Funktionale Programmierung

Grundlagen der funktionalen Programmierung I

Hans-Wolfgang Loidl, Andreas Abel

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

April 23, 2009

Administratives

- Vorlesung: Do 12-14 Uhr, Oettingenstr. 67, Oe 0.33 (ab 23.4.2009)
- Übungen: Mo 12-14 Uhr, Oettingenstr. 67, Takla-Makan (Z11 im CIP-Pool) (ab 27.4.2009)
- URL:
  http://www.tcs.informatik.uni-muenchen.de/lehre/SS09/Fun/
- Schein: Mini-Projekt (50%) + mündliche Prüfung (50%)

Überblick

Schwerpunkte in diesem Kurs:

- Grundlagen funktionaler Programmiersprachen
- Konzepte funktionaler Programmiersprachen
- Fortgeschrittene Programmiertechniken
- Forschungsthemen im Design von Programmiersprachen

Überblick

Schwerpunkte in diesem Kurs:

- Grundlagen funktionaler Programmiersprachen
- Konzepte funktionaler Programmiersprachen
- Fortgeschrittene Programmiertechniken
- Forschungsthemen im Design von Programmiersprachen

Erläutert anhand der Programmiersprache Haskell.
Begleitet von Übungen um obige Techniken konkret anzuwenden.

Überblick

Schwerpunkte in diesem Kurs:

- Grundlagen funktionaler Programmiersprachen
- Konzepte funktionaler Programmiersprachen
- Fortgeschrittene Programmiertechniken
- Forschungsthemen im Design von Programmiersprachen

Erläutert anhand der Programmiersprache Haskell.
Begleitet von Übungen um obige Techniken konkret anzuwenden.
Voraussetzungen:

- Teil "funktionale Sprachen" aus Info I
- Grundkentnisse zur Programmierung in einer funktionalen Sprache (z.B. SML)

Struktur des Kurses

23.04. Grundlagen funktionaler Programmierung
30.04. Monaden und imperative Programmierung
07.05. Grundlagen II (Why functional programming matters)
14.05. Zirkuläre Programme
28.05. GUI-Programmierung in Haskell
04.06. Parallele funktionale Programmierung
18.06. Abhängige Typen
25.06. Generische Programmierung
02.07. Faltungen und verschränkte Datentypen
09.07. Systematisches Testen funktionaler Programme
16.07. Verifikation funktionaler Programme
23.07. Abstrakte Maschinen

Grundlagen funktionaler Programmierung

Philosophie *deklarativer Sprachen*

> *Spezifiziere* **was** *berechnet werden soll,* **nicht wie** *es berechnet werden soll.*

Grundlagen funktionaler Programmierung

Philosophie *deklarativer Sprachen*

> *Spezifiziere* **was** *berechnet werden soll,* **nicht wie** *es berechnet werden soll.*

Imperative Sprachen starten von low-level Code (z.B. Assembler) und führen Abstraktionen ein um die Programmierung zu vereinfachen (z.B. Schleifen).

Grundlagen funktionaler Programmierung

Philosophie *deklarativer Sprachen*

> *Spezifiziere* **was** *berechnet werden soll,* **nicht wie** *es berechnet werden soll.*

Imperative Sprachen starten von low-level Code (z.B. Assembler) und führen Abstraktionen ein um die Programmierung zu vereinfachen (z.B. Schleifen).
Deklarative Sprachen sind durch die Sprache der Mathematik motiviert und beschreiben Funktionen oder Beziehungen:

- der λ-Kalkül beschreibt Funktionen und ist die Basis für **funktionale Sprachen**
- die Prädikaten-Logik beschreibt Beziehungen und ist die Basis für **logische Sprachen**

Die Church-Rosser Eigenschaft

Die charakteristische Eigenschaft deklarativer Sprachen ist die **Church-Rosser Eigenschaft**.
Intuitiv besagt sie, dass die Reihenfolge der Auswertung unerheblich für das Resultat ist.

Die Church-Rosser Eigenschaft

Die charakteristische Eigenschaft deklarativer Sprachen ist die **Church-Rosser Eigenschaft**.
Intuitiv besagt sie, dass die Reihenfolge der Auswertung unerheblich für das Resultat ist.

Diese Eigenschaft ist die formale Basis um in einem Programm einen Ausdruck durch sein Ergebnis zu ersetzen (Gleichheitsschließen). Eine Sprache, die dies uneingeschränkt zulässt wird als **referentiell transparent** bezeichnet.

Vor- und Nachteile deklarativer Sprachen

Vorteile deklarativer Sprachen

- Korrektheitsbeweise von Programmen sind relativ einfach.
- Programme sind meist sehr viel kürzer als z.B. in imperativen Sprachen.
- Erste Prototypen des Codes können schnell erstellt werden ("rapid prototyping").
- Es gibt keine inhärent sequentielle Auswertung.

Vor- und Nachteile deklarativer Sprachen

Vorteile deklarativer Sprachen

- Korrektheitsbeweise von Programmen sind relativ einfach.
- Programme sind meist sehr viel kürzer als z.B. in imperativen Sprachen.
- Erste Prototypen des Codes können schnell erstellt werden ("rapid prototyping").
- Es gibt keine inhärent sequentielle Auswertung.

Nachteile deklarativer Sprachen

- Vorhersagen des Ressourcenverbrauchs sind schwierig.
- Es sind weniger Werkzeuge zur Programmerstellung verfügbar.

Literatur

- Graham Hutton, "*Functional Programming in Haskell*", Cambridge University Press, 2007, ISBN 0-52169269-5.
- Bryan O'Sullivan, Don Stewart, John Goerzen, "*Real World Haskell*", O'Reilly, November 2008, ISBN: 0-59651498-0.
- Simon Thompson, "*Haskell: The Craft of Functional Programming*", Second Edition, Addison-Wesley, 1999. ISBN 0-201-34275-8.
- Paul Hudak, "*The Haskell School of Expression*", Cambridge University Press, 2000. ISBN 0-52164408-9.

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik* (d.h. keine Seiteneffekte);

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik* (d.h. keine Seiteneffekte);
- *Algebraische Datenstrukturen* mit mächtigen Operationen wie "pattern matching", "list comprehensions";

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik* (d.h. keine Seiteneffekte);
- *Algebraische Datenstrukturen* mit mächtigen Operationen wie "pattern matching", "list comprehensions";
- *Funktionen höherer Ordnung ("higher-order functions")*;

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik* (d.h. keine Seiteneffekte);
- *Algebraische Datenstrukturen* mit mächtigen Operationen wie "pattern matching", "list comprehensions";
- *Funktionen höherer Ordnung ("higher-order functions")*;
- *Statisches Typsystem*;

Konzepte in Haskell

Haskell ist eine rein-funktionale, nicht-strikte Programmiersprache.
Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik* (d.h. keine Seiteneffekte);
- *Algebraische Datenstrukturen* mit mächtigen Operationen wie "pattern matching", "list comprehensions";
- *Funktionen höherer Ordnung ("higher-order functions")*;
- *Statisches Typsystem*;
- *Bedarfsauswertung ("lazy evaluation")*;

Preliminaries

Basistypen in Haskell:

- *Bool*: Boole'sche (logische) Werte: *True* und *False*
- *Char*: Zeichen
- *String*: Zeichenfolgen: Listen von Zeichen, d.h äquivalent zu [*Char*]
- *Int*: Ganze Zahlen ("fixed precision integers")
- *Integer*: Ganze Zahlen beliebiger Länge ("arbitrary precision integers")
- *Float*: Fließkommazahlen ("single-precision floating point numbers")

Preliminaries

Zusammengesetzte Typen:

- **Listen**: $[\cdot]$, z.B. $[Int]$ als Listen von Integers, mit Werten wie $[1, 2, 3]$
- **Tupel**: $(\cdot, \cdots)$, z.B. $(Bool, Int)$ als Tupel von booleschen Werten und Integers, mit Werten wie $(True, 2)$
- **Verbunde**: $\cdot \{ \cdot, \cdots \}$, z.B. $BI \{ b :: Bool, i :: Int \}$ mit Werten wie $BI \{b = True, i = 2\}$
- **Funktionen**: $a \to b$, z.B. $Int \to Bool$

Preliminaries

Zusammengesetzte Typen:

- **Listen**: $[\cdot]$, z.B. $[Int]$ als Listen von Integers, mit Werten wie $[1, 2, 3]$
- **Tupel**: $(\cdot, \cdots)$, z.B. $(Bool, Int)$ als Tupel von booleschen Werten und Integers, mit Werten wie $(True, 2)$
- **Verbunde**: $\cdot \{ \cdot, \cdots \}$, z.B. $BI \{ b :: Bool, i :: Int \}$ mit Werten wie $BI \{b = True, i = 2\}$
- **Funktionen**: $a \rightarrow b$, z.B. $Int \rightarrow Bool$

Typsynonyme werden wie folgt definiert:

type $IntList = [Int]$

Typdeklarationen

Typdeklarationen haben folgendes Format: $e :: \tau$
d.h. "*Ausdruck e hat den Typ* τ."

Typdeklarationen

Typdeklarationen haben folgendes Format: $e :: \tau$
d.h. "*Ausdruck e hat den Typ* τ."

Beispiele:

$$[1, 2, 3] :: [Int]$$
$$x :: Bool$$
$$bi :: BI\ \{\ b :: Bool,\ i :: Int\ \}$$
$$inc :: Int \rightarrow Int$$

Typdeklarationen

Typdeklarationen haben folgendes Format: $e :: \tau$
d.h. "*Ausdruck e hat den Typ* τ."

Beispiele:

$$[1,2,3] :: [Int]$$
$$x :: Bool$$
$$bi :: BI\ \{\ b :: Bool,\ i :: Int\ \}$$
$$inc :: Int \rightarrow Int$$

Typdeklaration	
SML:	**Haskell:**
$val\ inc : int \rightarrow int$	$inc :: Int \rightarrow Int$

I. Algebraische Datentypen

Haskell stellte mächtige Mechanismen der Datenabstraktion zur Verfügung.
Algebraische Datentypen definieren benannte Alternativen ("constructors"), die beliebige Typen als Argumente haben können.

I. Algebraische Datentypen

Haskell stellte mächtige Mechanismen der Datenabstraktion zur Verfügung.
Algebraische Datentypen definieren benannte Alternativen ("constructors"), die beliebige Typen als Argumente haben können.

Ein Beispiel einer Enumeration, d.h. eines algebraischen Datentyps mit Alternativen ohne Argumente:

$$\textbf{data}\ \mathit{Day} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Sun}$$

Algebraische Datentypen

Musterangleich ("**pattern matching**") über diesen Datentyp kann wie folgt verwendet werden:

$$
\begin{aligned}
&\mathit{next} && :: \mathit{Day} \to \mathit{Day} \\
&\mathit{next}\ \mathit{Mon} && = \mathit{Tue} \\
&\mathit{next}\ \mathit{Tue} && = \mathit{Wed} \\
&\mathit{next}\ \mathit{Wed} && = \mathit{Thu} \\
&\mathit{next}\ \mathit{Thu} && = \mathit{Fri} \\
&\mathit{next}\ \mathit{Fri} && = \mathit{Sat} \\
&\mathit{next}\ \mathit{Sat} && = \mathit{Sun} \\
&\mathit{next}\ \mathit{Sun} && = \mathit{Mon}
\end{aligned}
$$

Notation: Funktionsdefinition

Wert- und Funktionsdefinitionen	
SML:	**Haskell:**
val x = ...	x = ...
fun f x = ...	f x = ...

Notation: Funktionsdefinition

Wert- und Funktionsdefinitionen	
SML:	**Haskell:**
val x = ...	x = ...
fun f x = ...	f x = ...

Funktionsdefinition in **SML** und in **Haskell**:

```
val next : day → day          next :: Day → Day
fun next Mon = Tue            next Mon = Tue
  | next Tue = Wed            next Tue = Wed
  | next Wed = Thu            next Wed = Thu
  | next Thu = Fri            next Thu = Fri
  | next Fri = Sat            next Fri = Sat
  | next Sat = Sun            next Sat = Sun
  | next Sun = Mon            next Sun = Mon
```

Algebraische Datentypen

Ein weiteres Beispiel eines algeb. Datentyps sind komplexe Zahlen:

data *Complex*1 = *Comp*1 *Float Float*

Algebraische Datentypen

Ein weiteres Beispiel eines algeb. Datentyps sind komplexe Zahlen:

$$\textbf{data}\ \mathit{Complex1} = \mathit{Comp1}\ \mathit{Float}\ \mathit{Float}$$

Die folgende Funktion erzeugt eine komplexe Zahl aus Real- und Imaginärteil:

$$\begin{aligned} &\mathit{mkComp1} &&:: \mathit{Float} \to \mathit{Float} \to \mathit{Complex1} \\ &\mathit{mkComp1}\ r\ i &&= \mathit{Comp1}\ r\ i \end{aligned}$$

Algebraische Datentypen

Ein weiteres Beispiel eines algeb. Datentyps sind komplexe Zahlen:

$$\textbf{data}\ \mathit{Complex1} = \mathit{Comp1}\ \mathit{Float}\ \mathit{Float}$$

Die folgende Funktion erzeugt eine komplexe Zahl aus Real- und Imaginärteil:

$$\begin{aligned} &\mathit{mkComp1} :: \mathit{Float} \to \mathit{Float} \to \mathit{Complex1} \\ &\mathit{mkComp1}\ r\ i = \mathit{Comp1}\ r\ i \end{aligned}$$

Die folgenden Funktionen extrahieren Real- und Imaginärkomponenten einer komplexen Zahl:

$$\begin{aligned} &\mathit{realPart1} :: \mathit{Complex1} \to \mathit{Float} \\ &\mathit{realPart1}\ (\mathit{Comp1}\ r\ \_) = r \\ &\mathit{imagPart1} :: \mathit{Complex1} \to \mathit{Float} \\ &\mathit{imagPart1}\ (\mathit{Comp1}\ \_\ i) = i \end{aligned}$$

Algebraische Datentypen

Algebraische Datentypen können **Typvariablen** als Parameter verwenden.
Als Beispiel können wir komplexe Zahlen mit einem Basistyp parametrisieren:

data *Complex2 a* = *Comp2 a a*

Algebraische Datentypen

Algebraische Datentypen können **Typvariablen** als Parameter verwenden.
Als Beispiel können wir komplexe Zahlen mit einem Basistyp parametrisieren:

$$\textbf{data}\ \mathit{Complex2}\ a\ =\ \mathit{Comp2}\ a\ a$$

Die Funktionen sind **polymorph**, d.h. sie arbeiten für beliebige Instanzen der Typvariablen a

$$\begin{aligned}
&\mathit{mkComp2} &&:: a \to a \to \mathit{Complex2}\ a\\
&\mathit{mkComp2}\ r\ i &&= \mathit{Comp2}\ r\ i
\end{aligned}$$

$$\begin{aligned}
&\mathit{realPart2} &&:: \mathit{Complex2}\ a \to a\\
&\mathit{realPart2}\ (\mathit{Comp2}\ r\ _) &&= r\\
&\mathit{imagPart2} &&:: \mathit{Complex2}\ a \to a\\
&\mathit{imagPart2}\ (\mathit{Comp2}\ _\ i) &&= i
\end{aligned}$$

Algebraische Datentypen

Als verkürzende Schreibweise ist es möglich Infix-Konstrukturen zu verwenden (diese müssen mit einem : Symbol beginnen).

$$\textbf{data}\ \mathit{Complex3}\ a\ =\ a\ :\!+\ a$$

Algebraische Datentypen

Als verkürzende Schreibweise ist es möglich Infix-Konstrukturen zu verwenden (diese müssen mit einem : Symbol beginnen).

```
data Complex3 a = a :+ a


mkComp3     :: a -> a -> Complex3
mkComp3 r i = r :+ i


realPart3           :: Complex3 a -> a
realPart3 (r :+ _) = r

imagPart3           :: Complex3 a -> a
imagPart3 (_ :+ i) = i
```

Algebraische Datentypen

Wir können auch einen Verbund verwenden, um benannte Felder in der Datenstruktur zu erhalten.

$$\textbf{data}\ \mathit{Complex4}\ a\ =\ \mathit{Comp4}\ \{\ \mathit{real}\ ::\ a\ ,\ \mathit{imag}\ ::\ a\ \}$$

Algebraische Datentypen

Wir können auch einen Verbund verwenden, um benannte Felder in der Datenstruktur zu erhalten.

$$\textbf{data}\ \mathit{Complex4}\ a\ =\ \mathit{Comp4}\ \{\ \mathit{real}\ ::\ a\ ,\ \mathit{imag}\ ::\ a\ \}$$

$$\begin{aligned}&\mathit{mkComp4}\ \ \ ::\ a\ \rightarrow\ a\ \rightarrow\ \mathit{Complex4}\ a\\&\mathit{mkComp4}\ r\ i\ =\ \mathit{Comp4}\ \{\ \mathit{real}\ =\ r,\ \mathit{imag}\ =\ i\ \}\end{aligned}$$

$$\begin{aligned}&\mathit{realPart4}\ \ ::\ \mathit{Complex4}\ a\ \rightarrow\ a\\&\mathit{realPart4}\ x\ =\ \mathit{real}\ x\end{aligned}$$

$$\begin{aligned}&\mathit{imagPart4}\ \ ::\ \mathit{Complex4}\ a\ \rightarrow\ a\\&\mathit{imagPart4}\ x\ =\ \mathit{imag}\ x\end{aligned}$$

Weitere Beispiele für Algebraische Datentypen

data *Maybe a* = *Nothing* | *Just a* — vordefinerter Datentyp
data *Either a b* = *Left a* | *Right b* — vordefinerter Datentyp
data *Nat* = *Zero* | *Succ Nat* — rekursiver Datentyp

Weitere Beispiele für Algebraische Datentypen

data *Maybe a* = *Nothing* | *Just a* — vordefinerter Datentyp
data *Either a b* = *Left a* | *Right b* — vordefinerter Datentyp
data *Nat* = *Zero* | *Succ Nat* — rekursiver Datentyp

Algebraische Datentypen in **SML:**

datatype $'a$ *binTree* = *leaf*
| *node* **of** $'a * 'a$ *binTree* $*$ $'a$ *binTree*;

Weitere Beispiele für Algebraische Datentypen

```
data Maybe a = Nothing | Just a      — vordefinerter Datentyp
data Either a b = Left a | Right b   — vordefinerter Datentyp
data Nat = Zero | Succ Nat           — rekursiver Datentyp
```

Algebraische Datentypen in **SML:**

```
datatype 'a binTree = leaf
                    | node of 'a * 'a binTree * 'a binTree;
```

Algebraische Datentypen in **Haskell:**

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

Pattern Matching in Haskell

"Pattern matching" ist eine verkürzende Schreibweise für eine explizite Fallunterscheidung über den Typ.
Folgende Funktion erzeugt die **Liste der Quadrate** der Eingabeliste.

$$
\begin{array}{lcl}
sqs & :: & [Int] \to [Int] \\
sqs\ [] & = & [] \\
sqs\ (x : xs) & = & x * x\ :\ sqs\ xs
\end{array}
$$

Pattern Matching in Haskell

"Pattern matching" ist eine verkürzende Schreibweise für eine explizite Fallunterscheidung über den Typ.
Folgende Funktion erzeugt die **Liste der Quadrate** der Eingabeliste.

$$\begin{aligned} sqs &:: [Int] \to [Int] \\ sqs\ [] &= [] \\ sqs\ (x : xs) &= x * x\ :\ sqs\ xs \end{aligned}$$

Diese Definition ist äquivalent zu

$$\begin{aligned} sqs\ xs' = \textbf{case}\ xs'\ \textbf{of}& \\ [] &\to [] \\ (x : xs) &\to x * x\ :\ sqs\ xs \end{aligned}$$

Notation: case

Case-Ausdruck in **SML**:

$$\begin{aligned} \mathit{fun}\ \mathit{sqs}\ \mathit{xs}' \ =\ & \textbf{case}\ \mathit{xs}'\ \textbf{of} \\ & \quad [] \ \Rightarrow\ [] \\ & \mid (x :: \mathit{xs}) \ \Rightarrow\ (x * x) :: (\mathit{sqs}\ \mathit{xs}) \end{aligned}$$

Case-Ausdruck in **Haskell**:

$$\begin{aligned} \mathit{sqs}\ \mathit{xs}' \ =\ & \textbf{case}\ \mathit{xs}'\ \textbf{of} \\ & \quad [] \ \rightarrow\ [] \\ & (x : \mathit{xs}) \ \rightarrow\ x * x \ :\ \mathit{sqs}\ \mathit{xs} \end{aligned}$$

Pattern Matching in Haskell

$$
\begin{aligned}
sqs\ &:: [Int] \rightarrow [Int] \\
sqs\ [] &= [] \\
sqs\ (x : xs) &= x * x\ :\ sqs\ xs
\end{aligned}
$$

Pattern Matching in Haskell

$$
\begin{aligned}
sqs &:: [Int] \to [Int] \\
sqs\ [] &= [] \\
sqs\ (x : xs) &= x * x\ :\ sqs\ xs
\end{aligned}
$$

Das allgemeine Schema von pattern matching code lässt sich aus folgender Schreibweise ablesen:

$$
\begin{aligned}
&sqs\ xs' = \\
&\quad \textbf{if}\ null\ xs'\ \textbf{then}\ [] \\
&\quad \textbf{else let} \\
&\qquad\qquad x = head\ xs' \\
&\qquad\qquad xs = tail\ xs' \\
&\qquad \textbf{in} \\
&\qquad x * x\ :\ sqs\ xs
\end{aligned}
$$

Notation: Layout rule

Um Elemente einer Sequenz von Definitionen oder Case-Zweigen zusammenzufassen, muss jedes Element mindestens so weit wie das erste Element eingerückt sein ("**layout**" rule).

Notation: Layout rule

Um Elemente einer Sequenz von Definitionen oder Case-Zweigen zusammenzufassen, muss jedes Element mindestens so weit wie das erste Element eingerückt sein ("**layout**" rule).

Man kann auch die längere Notation $\{\ldots;\ldots\}$ verwenden, z.B:

$$
\begin{array}{l}
sqs\ xs' = \\
\quad \textbf{if}\ null\ xs'\ \textbf{then}\ [] \\
\quad \textbf{else let}\ \{\ x = head\ xs'\ ;\ xs = tail\ xs'\ \}\ \textbf{in}\ x * x\ :\ sqs\ xs
\end{array}
$$

Guarded Patterns

"**Guarded patterns**" erlauben es eine Bedingung zu spezifizieren, die gelten muss, damit ein Zweig einer Definition ausgeführt wird.

Guarded Patterns

"**Guarded patterns**" erlauben es eine Bedingung zu spezifizieren, die gelten muss, damit ein Zweig einer Definition ausgeführt wird. Folgende Funktion ermittelt die Anzahl der Listenelemente, die größer sind als 0:

$$
\begin{array}{lll}
gt0\ [] & = & 0 \\
gt0\ (x : xs) & \mid\ x > 0 & = 1 + gt0\ xs \\
 & \mid\ otherwise & = gt0\ xs
\end{array}
$$

Auswertungsreihenfolge einer Funktionsapplikation $gt0\ xs$:

1. Stimmt die Struktur von xs mit dem Pattern überein?
2. Falls ja, binde die Variablen im Pattern.
3. Trifft die Bedingung im Guard zu?
4. Falls ja, werte die rechte Seite aus.

Die Regeln werden in der textuellen Reihenfolge geprüft.

List comprehensions in Haskell

"**List comprehensions**" sind der mathematischen Notation für Mengen entlehnt.

List comprehensions in Haskell

"**List comprehensions**" sind der mathematischen Notation für Mengen entlehnt.
Beispiel: Liste der Quadrate gerader Zahlen aus *xs*:

$$sq\_even\ xs\ =\ [x * x \mid x \leftarrow xs,\ even\ x]$$

List comprehensions in Haskell

"**List comprehensions**" sind der mathematischen Notation für Mengen entlehnt.
Beispiel: Liste der Quadrate gerader Zahlen aus *xs*:

$$sq\_even\ xs\ =\ [\underline{x * x}\ |\ x\ \leftarrow\ xs,\ even\ x]$$

Der Ausdruck $\underline{x * x}$ ist der Rumpf der list comprehension. Er bestimmt wie ein Listenelement berechnet wird.

List comprehensions in Haskell

"**List comprehensions**" sind der mathematischen Notation für Mengen entlehnt.
Beispiel: Liste der Quadrate gerader Zahlen aus *xs*:

$$sq\_even\ xs\ =\ [x * x \mid \underline{x\ \leftarrow\ xs},\ even\ x]$$

Der Ausdruck $\underline{x * x}$ ist der Rumpf der list comprehension. Er bestimmt wie ein Listenelement berechnet wird.
Der Ausdruck $\underline{x \leftarrow xs}$ ist ein Generator und weist nacheinander x die Elemente der Liste xs zu.

List comprehensions in Haskell

"**List comprehensions**" sind der mathematischen Notation für Mengen entlehnt.
Beispiel: Liste der Quadrate gerader Zahlen aus *xs*:

$$sq\_even\ xs\ =\ [x * x \mid x \leftarrow xs,\ \underline{even\ x}]$$

Der Ausdruck $\underline{x * x}$ ist der Rumpf der list comprehension. Er bestimmt wie ein Listenelement berechnet wird.
Der Ausdruck $\underline{x \leftarrow xs}$ ist ein Generator und weist nacheinander x die Elemente der Liste *xs* zu.
Die Bedingung $\underline{even\ x}$ entscheidet, ob der Wert von x in der erzeugten Liste verwendet werden soll.

II. Funktionen höherer Ordnung ("higher-order functions")

Viele Funktionen über Listen folgen einem generellen Format:

$$\begin{aligned} f\ [] &= v \\ f\ (x : xs) &= x \oplus f\ xs \end{aligned}$$

Zum Beispiel kann die Summe über eine Liste wie folgt definiert werden:

$$\begin{aligned} sum\ [] &= 0 \\ sum\ (x : xs) &= x + (sum\ xs) \end{aligned}$$

Die *foldr* Funktion

Die higher-order Funktion *foldr* abstrahiert diese Berechnungsstruktur

$$
\begin{aligned}
&\mathit{foldr} && :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ v\ [] && = v \\
&\mathit{foldr}\ f\ v\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ v\ xs)
\end{aligned}
$$

Die *foldr* Funktion

Die higher-order Funktion *foldr* abstrahiert diese Berechnungsstruktur

$$\begin{aligned} &\mathit{foldr} && :: (b \to a \to b) \to b \to [a] \to b \\ &\mathit{foldr}\ f\ v\ [] && = v \\ &\mathit{foldr}\ f\ v\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ v\ xs) \end{aligned}$$

Die *foldr* Funktion ersetzt jedes : in der Eingabeliste durch ein f, und jedes [] durch ein v:

Auswertung

$$\mathit{foldr}\ \oplus\ v\ (x_0 : \ldots : x_n : []) \Longrightarrow x_0 \oplus \ldots \oplus (x_n \oplus v)$$

Die *foldr* Funktion

Die higher-order Funktion *foldr* abstrahiert diese Berechnungsstruktur

$$\begin{aligned} &\mathit{foldr} && :: (b \to a \to b) \to b \to [a] \to b \\ &\mathit{foldr}\ f\ v\ [] && = v \\ &\mathit{foldr}\ f\ v\ (x : xs) && = f\ x\ (\mathit{foldr}\ f\ v\ xs) \end{aligned}$$

Die *foldr* Funktion ersetzt jedes : in der Eingabeliste durch ein f, und jedes [] durch ein v:

Auswertung

$$\mathit{foldr}\ \oplus\ v\ (x_0 : \ldots : x_n : []) \Longrightarrow x_0 \oplus \ldots \oplus (x_n \oplus v)$$

Wir können nun die Summe über eine Liste wie folgt definieren:

$$\mathit{sum} = \mathit{foldr}\ (+)\ 0$$

Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions") nehmen Funktionen als Argumente oder liefern Funktionen als Resultate.

Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions") nehmen Funktionen als Argumente oder liefern Funktionen als Resultate.

Im Allgemeinen können Funktionen wie Daten verwendet werden, z.B. als Elemente von Datenstrukturen ("*functions are first-class values*").

Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions") nehmen Funktionen als Argumente oder liefern Funktionen als Resultate.

Im Allgemeinen können Funktionen wie Daten verwendet werden, z.B. als Elemente von Datenstrukturen ("*functions are first-class values*").

Funktionen höherer Ordnung werden benutzt, um häufig verwendete Berechnungsstrukturen zu abstrahieren.

Die *foldl* Funktion

Die higher-order Funktion *foldl* ist die links-assoziative Version von *foldr*:

$$
\begin{aligned}
&\mathit{foldl} && :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ v\ [] && = v \\
&\mathit{foldl}\ f\ v\ (x : xs) && = \mathit{foldl}\ f\ (f\ v\ x)\ xs
\end{aligned}
$$

Die *foldl* Funktion

Die higher-order Funktion *foldl* ist die links-assoziative Version von *foldr*:

$$
\begin{array}{lll}
\mathit{foldl} & :: & (a \to b \to b) \to b \to [a] \to b \\
\mathit{foldl}\ f\ v\ [] & = & v \\
\mathit{foldl}\ f\ v\ (x : xs) & = & \mathit{foldl}\ f\ (f\ v\ x)\ xs
\end{array}
$$

Auswertung

$$\mathit{foldl}\ \oplus\ v\ (x_0 : \ldots : x_n : []) \Longrightarrow (\ldots(v \oplus x_0) \oplus \ldots \oplus x_n)$$

Die *map* Funktion

Die vordefinierte, higher-order Funktion *map* wendet eine Funktion f auf alle Elemente der Eingabeliste an, d.h. es gilt

Auswertung

$$map\ f\ [x_0, x_1, \ldots, x_n] \Longrightarrow [f\ x_0,\ \ldots,\ f\ x_n]$$

Die *map* Funktion

Die vordefinierte, higher-order Funktion *map* wendet eine Funktion f auf alle Elemente der Eingabeliste an, d.h. es gilt

Auswertung

$$map\ f\ [x_0, x_1, \ldots, x_n] \Longrightarrow [f\ x_0,\ \ldots,\ f\ x_n]$$

Wir können nun unser Beispiel der Quadrate aller Zahlen in *xs* wie folgt definieren:

```
sqs     :: [Int] → [Int]
sqs xs =  map square xs
  where
    square x = x * x
```

Lambda Ausdrücke

Im vorangegangenen Beispiel haben wir die *square* Funktion nur einmal verwendet. In solchen Fällen ist es bequemer eine lokale Definition zu vermeiden, und statt dessen einen Lambda Ausdruck zu verwenden.

Lambda Ausdrücke

Im vorangegangenen Beispiel haben wir die *square* Funktion nur einmal verwendet. In solchen Fällen ist es bequemer eine lokale Definition zu vermeiden, und statt dessen einen Lambda Ausdruck zu verwenden.
Ein **Lambda Ausdruck** ist eine anonyme Funktion, d.h. er definiert wie das Resultat in Abhängigkeit von der Eingabe berechnet wird ohne der Funktion einen Namen zu geben.

$$sqs :: [Int] \rightarrow [Int]$$
$$sqs = map\,(\lambda x \rightarrow x * x)$$

Lambda Ausdrücke

Im vorangegangenen Beispiel haben wir die *square* Funktion nur einmal verwendet. In solchen Fällen ist es bequemer eine lokale Definition zu vermeiden, und statt dessen einen Lambda Ausdruck zu verwenden.
Ein **Lambda Ausdruck** ist eine anonyme Funktion, d.h. er definiert wie das Resultat in Abhängigkeit von der Eingabe berechnet wird ohne der Funktion einen Namen zu geben.

$$sqs :: [Int] \rightarrow [Int]$$
$$sqs = map\,(\lambda x \rightarrow x * x)$$

Lambda Ausdrücke	
SML:	**Haskell:**
`fn x => ...`	`\ x -> ...`

Funktionskomposition

Da Funktionen Werte erster Klasse sind, können sie auch miteinander kombiniert werden, z.B $(f \,.\, g)$ wobei folgendes gilt:

$$(f \,.\, g)\, x \;=\; f\,(g\, x)$$

Wir definieren die Liste der Quadrate aller geraden Zahlen:

$$\begin{aligned} &\mathit{sq\_even} :: [\mathit{Int}] \;\to\; [\mathit{Int}] \\ &\mathit{sq\_even} = \; \mathit{map}\,(\lambda\, x \;\to\; x * x) \,.\, \mathit{filter}\ \mathit{even} \end{aligned}$$

Die Funktion *filter p xs* liefert alle Elemente der Liste *xs*, für die das Prädikat *p True* liefert.

Funktionskomposition

Da Funktionen Werte erster Klasse sind, können sie auch miteinander kombiniert werden, z.B $(f \,.\, g)$ wobei folgendes gilt:

$$(f \,.\, g)\, x \;=\; f\,(g\, x)$$

Wir definieren die Liste der Quadrate aller geraden Zahlen:

$$\begin{aligned} &\mathit{sq\_even} :: [\mathit{Int}] \;\to\; [\mathit{Int}] \\ &\mathit{sq\_even} = \; \mathit{map}\,(\lambda\, x \;\to\; x * x)\,.\,\mathit{filter\ even} \end{aligned}$$

Die Funktion *filter p xs* liefert alle Elemente der Liste *xs*, für die das Prädikat *p True* liefert.

Funktionskomposition	
SML:	**Haskell:**
`f o g`	`f . g`

Partielle Applikation

In Sprachen wie SML werden die Argumente von Funktionen meist zu Tupeln zusammengefasst:

$$\begin{aligned} add' &:: (Int, Int) \rightarrow Int \\ add'\,(x, y) &= x + y \end{aligned}$$

Partielle Applikation

In Sprachen wie SML werden die Argumente von Funktionen meist zu Tupeln zusammengefasst:

$$\begin{aligned} add' &:: (Int, Int) \to Int \\ add'\,(x, y) &= x + y \end{aligned}$$

Funktionen, die ihre Argumente schrittweise verwenden, werden als "**curried functions**" bezeichnet, z.B:

$$\begin{aligned} add &:: Int \to (Int \to Int) \\ add\,x\,y &= x + y \end{aligned}$$

In dieser Version nimmt die *add* Funktion ein Argument und liefert eine Funktion $Int \to Int$ als Resultat. Der Ausdruck $add\ 1\ 2$ ist wie $(add\ 1)\ 2$ zu lesen. Man bezeichnet $add\ 1$ als eine **partielle Applikation**.

Partielle Applikation

Partielle Applikationen sind nützlich, wenn man Argumente, die sich selten ändern, fixieren will, z.B:

$$inc :: Int \rightarrow Int$$
$$inc = add\ 1$$

Partielle Applikation

Partielle Applikationen sind nützlich, wenn man Argumente, die sich selten ändern, fixieren will, z.B:

$$inc :: Int \to Int$$
$$inc = add\ 1$$

Um die Werte aller Listenelemente zu erhöhen definieren wir

$$incAll :: [Int] \to [Int]$$
$$incAll = map\ inc$$

Partielle Applikation

Partielle Applikationen sind nützlich, wenn man Argumente, die sich selten ändern, fixieren will, z.B:

$$\begin{aligned} &inc :: Int \to Int \\ &inc = add\ 1 \end{aligned}$$

Um die Werte aller Listenelemente zu erhöhen definieren wir

$$\begin{aligned} &incAll :: [Int] \to [Int] \\ &incAll = map\ inc \end{aligned}$$

Wir können diese Definition auch kürzer schreiben:

$$\begin{aligned} &incAll :: [Int] \to [Int] \\ &incAll = map\ (+1) \end{aligned}$$

Durch das Weglassen eines Arguments eines binären Operators wird eine partielle Applikation definiert, z.B. ist der Ausdruck $(+1)$ äquivalent zu $\lambda x \to x + 1$.

Beispiel: Cesar Cipher

Das Prinzip von Caesar's Verschlüsselungsfunktion ("cesar cipher"):

Verschlüsselung (*encode*):
Gegeben: Eine Liste von Zeichen *xs*.
Gesucht: Eine Liste von Zeichen *encode n xs*, in der jedes *m*-te Zeichen des Alphabets durch das $m + n$-te Zeichen ersetzt ist.

Entschlüsselung (*crack*):
Gegeben: Eine Liste von Zeichen *xs*.
Gesucht: Eine Liste von Zeichen *crack xs*, sodass

$$\exists n.xs = encode\ n\ (crack\ xs).$$

Cesar Cipher: Hilfsfunktionen

Idee: Jedes Zeichen (Kleinbuchstabe) muss um einen gegebenen Betrag verschoben werden.

Cesar Cipher: Hilfsfunktionen

Idee: Jedes Zeichen (Kleinbuchstabe) muss um einen gegebenen Betrag verschoben werden.

Hilfsfunktionen:

$$\begin{aligned} &\mathit{let2int} :: \mathit{Char} \to \mathit{Int} \\ &\mathit{let2int}\ c = \mathit{ord}\ c - \mathit{ord}\ 'a' \end{aligned}$$

$$\begin{aligned} &\mathit{int2let} :: \mathit{Int} \to \mathit{Char} \\ &\mathit{int2let}\ n = \mathit{chr}\ (\mathit{ord}\ 'a' + n) \end{aligned}$$

Cesar Cipher: Verschlüsselung

Verschiebefunktion:

$$
\begin{aligned}
&\mathit{shift} &&:: \mathit{Int} \to \mathit{Char} \to \mathit{Char} \\
&\mathit{shift}\ n\ c &&= \mathit{int2let}\ ((\mathit{let2int}\ c + n)\ `\mathit{mod}`\ 26)
\end{aligned}
$$

Cesar Cipher: Verschlüsselung

Verschiebefunktion:

$$
\begin{aligned}
&\mathit{shift} :: \mathit{Int} \to \mathit{Char} \to \mathit{Char}\\
&\mathit{shift}\ n\ c = \mathit{int2let}\ ((\mathit{let2int}\ c + n)\ `\mathit{mod}`\ 26)
\end{aligned}
$$

Verschlüsselungsfunktion:

$$
\begin{aligned}
&\mathit{encode} :: \mathit{Int} \to \mathit{String} \to \mathit{String}\\
&\mathit{encode}\ n\ cs = [\ \mathit{shift}\ n\ c \mid c \leftarrow cs\ ]
\end{aligned}
$$

Cesar Cipher: Entschlüsselung

Idee: Verschiebe den Eingabestring so, dass die Häufigkeiten der Zeichen möglichst nah an den bekannten Häufigkeiten von Kleinbuchstaben im Alphabet liegen.

Cesar Cipher: Entschlüsselung

Idee: Verschiebe den Eingabestring so, dass die Häufigkeiten der Zeichen möglichst nah an den bekannten Häufigkeiten von Kleinbuchstaben im Alphabet liegen.

Wir benötigen

- eine Tabelle der Häufigkeiten von Kleinbuchstaben im Alphabet;
- Hilfsfunktionen zum Ermitteln der Häufigkeiten und deren Distanz;
- eine Entschlüsselungsfunktion *crack*, gemäß obiger Idee.

Cesar Cipher: Entschlüsselung

Die bekannten Häufigkeiten von Kleinbuchstaben im Alphabet sind durch folgende Tabelle definiert:

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2,
         2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4,
         6.7, 7.5, 1.9, 0.1, 6.0, 6.3,
         9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

Cesar Cipher: Hilfsfunktionen

Berechnung des Prozentsatzes:

$$
\begin{aligned}
&\mathit{percent} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Float}\\
&\mathit{percent}\ n\ m &&= (\mathit{fromIntegral}\ n\ /\ \mathit{fromIntegral}\ m) * 100
\end{aligned}
$$

Berechnung der Häufigkeiten aller Kleinbuchstaben im String *cs*:

$$
\begin{aligned}
&\mathit{freqs} &&:: \mathit{String} \to [\mathit{Float}]\\
&\mathit{freqs}\ cs &&= [\mathit{percent}\ (\mathit{count}\ c\ cs)\ n \mid c \leftarrow ['a'..'z']\ ]\\
&\quad \textbf{where} &&\\
&\qquad n &&= \mathit{lowers}\ cs
\end{aligned}
$$

Cesar Cipher: Hilfsfunktionen

Links-Rotation einer Liste: $rotate\ n\ xs\ =\ drop\ n\ xs\ ++\ take\ n\ xs$
Anzahl der Kleinbuchstaben in einem String:

$$\begin{aligned} lowers\ &::\ String\ \to\ Int \\ lowers\ cs\ &=\ length\ [\ c\ \mid\ c\ \leftarrow\ cs,\ isLower\ c] \end{aligned}$$

Häufigkeit eines Zeichens in einem String:

$$\begin{aligned} count\ &::\ Char\ \to\ String\ \to\ Int \\ count\ c\ cs\ &=\ length\ [\ c'\ \mid\ c'\ \leftarrow\ cs,\ c\ ==\ c'] \end{aligned}$$

Auftreten eines Zeichens in einem String:

$$\begin{aligned} positions\ &::\ Eq\ a\ \Rightarrow\ a\ \to\ [a]\ \to\ [Int] \\ positions\ x\ xs\ &=\ [\ i'\ \mid\ (x',\ i')\ \leftarrow\ zip\ xs\ [0..n],\ x\ ==\ x'\ ] \\ &\quad \textbf{where} \\ &\qquad n\ =\ length\ xs\ -\ 1 \end{aligned}$$

Cesar Cipher: Vergleich von Häufigkeiten

Zum Vergleich einer Sequenz von beobachteten Werten *os* mit einer Sequenz von erwarteten Werten *es* wird die *Chi-Quadrat* Funktion χ verwendet. Diese Funktion bestimmt die Distanz, d.h. je geringer der Wert ist desto näher sind die beobachteten Werte an den erwarteten Werten.

$$\chi\ os\ es\ = \sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

Cesar Cipher: Vergleich von Häufigkeiten

Zum Vergleich einer Sequenz von beobachteten Werten *os* mit einer Sequenz von erwarteten Werten *es* wird die *Chi-Quadrat* Funktion χ verwendet. Diese Funktion bestimmt die Distanz, d.h. je geringer der Wert ist desto näher sind die beobachteten Werte an den erwarteten Werten.

$$\chi\ os\ es\ = \sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

Diese Funktion kann wie folgt implementiert werden:

$$\begin{aligned} &chisqr && :: && [Float] \to [Float] \to Float \\ &chisqr\ os\ es && = && sum\,[((o-e)\uparrow 2)/e \mid (o,e) \leftarrow zip\ os\ es] \end{aligned}$$

Cesar Cipher: Entschlüsselung

Die Entschlüsselungsfunktion bestimmt für jeden Verschiebungsfaktor *n* die *chisqr* Distanz, und wählt den Faktor *factor* mit der geringsten Distanz.

Cesar Cipher: Entschlüsselung

Die Entschlüsselungsfunktion bestimmt für jeden Verschiebungsfaktor *n* die *chisqr* Distanz, und wählt den Faktor *factor* mit der geringsten Distanz.

$$
\begin{aligned}
&\mathit{crack} \;::\; \mathit{String} \to \mathit{String}\\
&\mathit{crack}\ \mathit{cs} = \mathit{encode}\ (-\mathit{factor})\ \mathit{cs}\\
&\quad \textbf{where}\\
&\qquad \mathit{factor} = \mathit{head}\ (\mathit{positions}\ (\mathit{minimum}\ \mathit{chitab})\ \mathit{chitab})\\
&\qquad \mathit{chitab} = [\ \mathit{chisqr}\ (\mathit{rotate}\ n\ \mathit{table}')\ \mathit{table} \mid n \leftarrow [0..25]\ ]\\
&\qquad \mathit{table}' = \mathit{freqs}\ \mathit{cs}
\end{aligned}
$$

Zusammenfassung der Konzepte in Haskell

Die wichtigsten Konzepte in Haskell sind:

- *Rein funktionale Semantik*: keine Seiteneffekte;
- *Statisches Typsystem*: Typen können bereits zur Compilezeit geprüft werden;
- *Algebraische Datenstrukturen* mit "pattern matching";
- *Parametrischer Polymorphismus*: die gleiche Funktion kann auf Werte unterschiedlicher Typen angewendet werden;
- *"overloading"*: verschiedene Funktionen können durch dasselbe Symbol repräsentiert werden;
- *Funktionen höherer Ordnung ("higher-order functions")*: Funktionen können als Argumente an andere Funktionen weitergegeben und als Resultatwerte zurückgegeben werden;
- *Bedarfsauswertung ("lazy evaluation")*

Ressourcen

Weiterführendes Material:

- Haskell Page: http://haskell.org/
- Haskell Report: http://haskell.org/onlinereport/
- Haskell Libraries: http://www.haskell.org/ghc/dist/current/docs/libraries/
- Wikibooks: http://en.wikibooks.org/wiki/Haskell