Kapitel 3

Funktionales Programmieren

Übersicht

Übersicht (2)

- Algorithmen auf Listen und Bäumen
  - Sortieren
  - Suchen
- Polymorphie und Funktionen höherer Ordnung
  - Typisierung
  - Funktionen höherer Ordnung
- Semantik, Testen und Verifikation
  - Zur Semantik funktionaler Programme
  - Testen und Verifikation

Abschnitt 3.1

Grundkonzepte funktionaler Programmierung

Unterabschnitt 3.1.1

Beispiele

Beispiele

Reichweite eines Geschosses

$$d = d(v, \theta, y_0) = \frac{v \cos\theta}{g}((v \sin\theta)^2 + 2gy_0)$$

$$g = 9.81$$

- als Ausdruck
- als Funktion
- Ersetzung

Beispiele (2)

Finbonacci's Kaninchen:

- Starte mit einem Kaninchenpaar zu Monat m=0
- Ein Monat, um fortpflanzungsfähig zu werden
- Jedes fortpflanzungsfähige Kaninchenpaar erzeugt ein Paar pro Monat

- kaninchen = alle aus dem vormonat + alle neuen
- alle neuen = Anzahl aus Vorvormonat
- $f_n = f_{n-1} + f_{n-2}$, $f_0 = 1$, $f_1 = 1$

Beispiele (3)

Fakultät

- $n! = \prod_{i=1}^{n} i = 1 \cdot \ldots \cdot n$
- $n! = (n-1)! \cdot n,\ n > 1$

$$n! = \begin{cases} 1 & , \text{ für } n = 0 \\ (n-1)!i \cdot n & , \text{ für } n > 0 \end{cases}$$

Beispiele (4)

Hypothekentilgung

- fange mit Betrag *b* an
- monatliche Abzahlung *a*
- monatliche Zinsen *p*

- Wie lange dauert die Tilgung?
- Wie viel wurde insgesamt abgezahlt?

Beispiele (5)

Quadratwurzel

- $y = \sqrt{x}$ bedeutet $y^2 = x$
- fang mit einem Versuch an g
- wenn $g^2 \approx x$ dann $g \approx \sqrt{x}$
- wenn nicht, versuche $\frac{g+x/g}{2}$

Unterabschnitt 3.1.2

Zentrale Begriffe und Einführung

Zentrale Begriffe und Einführung

Funktionale Programmierung im Überblick:

- Funktionales Programm:
  - partielle Funktionen von Eingabe- auf Ausgabedaten
  - besteht aus Deklarationen von (Daten-)Typen, Funktionen und (Daten-)Strukturen
  - Rekursion ist eines der zentralen Sprachkonzepte
  - in Reinform: kein Zustandskonzept, keine veränderlichen Variablen, keine Schleifen, keine Zeiger
- Ausführung eines funktionalen Programms: Anwendung einer Funktion auf Eingabedaten
- Zusätzliche Programmierkonstrukte, um die Kommunikation mit der Umgebung zu beschreiben

Definition: (partielle Funktion)

Ein Funktion heißt ***partiell***, wenn sie nur auf einer Untermenge ihres Argumentbereichs definiert ist.

Andernfalls heißt sie ***total***.

Welche der Beispielfunktionen sind partiell?

Beispiel: (partielle Funktion)

1. Bezeichne *Nat* die Menge der *natürlichen Zahlen* (0 und größer) und sei $fact :: Nat \to Nat$ wie folgt definiert:

$$fact(n) = \begin{cases} 1 & , \text{ für } n = 0 \\ fact(n-1) * n & , \text{ für } n > 0 \end{cases}$$

Dann ist *fact* wohldefiniert und total.

Beispiel: (partielle Funktion) (2)

2. Bezeichne *Float* die Menge der auf dem Rechner darstellbaren *Gleitkommazahlen*. Dann ist die Funktion

$$sqrt :: Float \to Float \quad ,$$

   die die Quadratwurzel (engl. square root) berechnet, partiell.

3. Bezeichne *String* die Menge der Zeichenreihen. Dann ist die Funktion abschneide2, die die ersten beiden Zeichen einer Zeichenreihe abschneidet partiell (warum?)

Definition: (Funktionsanwendung, -auswertung, Terminierung, Nichtterminierung)

Bezeichne f eine Funktion, a ein zulässiges Argument von f.

Die Anwendung von f auf a nennen wir eine ***Funktionsanwendung*** (engl. ***function application***); meist schreibt man dafür $f(a)$ oder $f\ a$.

Den Prozess der Berechnung des Funktionswerts nennen wir ***Auswertung*** (engl. ***evaluation***). Die Auswertung kann:

- nach endlich vielen Schritten *terminieren* und ein Ergebnis liefern (***normale Terminierung***, engl. ***normal termination***),
- nach endlich vielen Schritten *terminieren* und einen Fehler melden (***abrupte Terminierung***, engl. ***abrupt termination***),
- ***nicht terminieren***, d.h. der Prozess der Auswertung kommt (von alleine) nicht zu Ende.

Bemerkungen:

- Entsprechendes gilt in anderen Programmierparadigmen.
- Da Terminierung nicht *entscheidbar* ist, benutzt man in der Informatik häufig partielle Funktionen.
  Beispiel: (Zur Entscheidbarkeit der Terminierung)
  McCarthy's Funktion:
  Sei $m :: Nat \to Nat$ wie folgt definiert:

$$m(n) = \begin{cases} n - 10 & , \text{für } n > 100 \\ m(m(n+11)) & , \text{für } n \leq 100 \end{cases}$$

  Ist m für alle Argumente wohldefiniert?
- In der Theorie kann man durch Einführen eines Elements „jede“partielle Funktion total machen. Üblicherweise bezeichnet man das Element für „undefiniert “ mit $\perp$ (engl. „bottom “).

Begriffsklärung: (Wert, Value)

Werte (engl. ***values***) in der (reinen) funktionalen Programmierung sind
Schon gesehen:

- Elementare Daten (Zahlen, Wahrheitswerte, Zeichen, ...),

Weitere:

- zusammengesetzte Daten (Listen von Werten, Wertepaare, ...),
- (partielle) Funktionen mit Werten als Argumenten und Ergebnissen.

Also sind auch Listen von Funktionen Werte.

Bemerkungen:

- In anderen Sprachparadigmen gibt es auch Werte, allerdings werden Funktionen nicht immer als Werte betrachtet (z.B. in der objektorientierten Programmierung: „immutable objects“).
- Im Mittelpunkt der funktionalen Programmierung steht die Definition von Wertemengen (Datentypen) und Funktionen. Funktionale Programmiersprachen stellen dafür Sprachmittel zur Verfügung.
- Wie für abstrakte Objekte oder Begriffe typisch, besitzen Werte
  - keinen Ort,
  - keine Lebensdauer,
  - keinen veränderbaren Zustand,
  - kein Verhalten.

Begriffsklärung: (Typ, engl. type)

Ein **Typ** (engl. type) fasst Werte zusammen, auf denen die gleichen Funktionsanwendungen zulässig sind.

Typisierte Sprachen besitzen ein Typsystem, das für jeden Wert festlegt, welchen Typ er hat.

In funktionalen Programmiersprachen gibt es drei Arten von Werten bzw. Typen, mit denen man rechnen kann. Schon gesehen:

- Basisdatentypen (`Int`, `Float`, `Bool`, `String`, ...)

Weitere:

- benutzerdef., insbesondere rekursive Datentypen
- Funktionstypen, z.B. `Int` $\rightarrow$ `Bool` oder
  (`Int` $\rightarrow$ `Int`) $\rightarrow$ (`Int` $\rightarrow$ `Int`)

Datenstrukturen

- Was sind die Namen der Typen? Z.B. ganze Zahlen Z
- Was sind die zulässigen Werte? Z.B. $\{0, 1, -1, 2, -2, ...\}$
- Was sind definierte Funktionen? Z.B. $f : Z \rightarrow Z$

Unterabschnitt 3.1.3

Einfache Datenstrukturen in Haskell

Einfache Datenstruktur der ganzen Zahlen

Hier ist ein Beispiel für ganze Zahlen:

Typ: `Integer`

Funktionen:

`(+), (*), (-)`	::	`Integer` → `Integer` → `Integer`
`div, mod`	::	`Integer` → `Integer` → `Integer`

Konstanten:

– in Dezimaldarstellung: 0, 127, (-23)

Dem Typbezeichner `Integer` ist die Menge der ganzen Zahlen als Wertemenge zugeordnet. Schreibe negative Zahlen in Klammern.

Datenstrukturen

Eine *Struktur* fasst Typen und Werte zusammen, insbesondere also auch Funktionen.

Datenstrukturen sind Strukturen, die mindestens einen „neuen“*Datentyp* und alle seine wesentlichen Funktionen bereitstellen.

Eine ***Datenstruktur*** besteht aus einer oder mehrerer disjunkter Wertemengen zusammen mit den darauf definierten Funktionen.

In der Mathematik nennt man solche Gebilde *Algebren* oder einfach nur *Strukturen*.

In der Informatik spricht man auch von einer *Rechenstruktur*.

Definition: (Signatur einer Datenstruktur)

Die ***Signatur*** $(\mathbf{T}, \mathbf{F})$ ***einer Datenstruktur*** besteht aus

- einer endlichen Menge **T** von Typbezeichnern und
- einer endlichen Menge **F** von Funktionsbezeichnern,

wobei für jedes $f \in \mathbf{F}$ ein Funktionstyp

$$f :: T_1 \to \cdots \to T_n \to T_0\ , \quad T_i \in \mathbf{T}, \quad 0 \leq i \leq n,$$

definiert ist. n gibt die **Stelligkeit** von f an.

Definition: (Datenstruktur mit Signatur)

Eine (partielle) ***Datenstruktur mit Signatur*** $(\mathbf{T}, \mathbf{F})$ ordnet

- jedem Typbezeichner $T \in \mathbf{T}$ eine Wertemenge,
- jedem Funktionsbezeichner $f \in \mathbf{F}$ eine partielle Funktion zu,

so dass Argument- und Wertebereich von f den Wertemengen entsprechen, die zu $f's$ Funktionstyp gehören.

Bemerkungen:

- Wir betrachten zunächst die Basisdatenstrukturen, wie man sie in jeder Programmier-, Spezifikations- und Modellierungssprache findet.
- Die Basisdatenstrukturen (engl. basic / primitive data structures) bilden die Grundlage zur Definition weiterer Typen, Funktionen und Datenstrukturen.
- Als Beispiel dienen uns die Basisdatenstrukturen der funktionalen Sprache Haskell. Später lernen wir auch die Basisdatenstrukturen von Java kennen.
- Wir benutzen auch ***Operatorsymbole*** wie + und * um Funktionen zu bezeichnen.
- Nullstellige Funktionen nennen wir ***Konstanten***.

Die Datenstruktur der booleschen Werte:

Typ: `Bool`

Funktionen:

```
(==)   :: Bool → Bool → Bool
(/=)   :: Bool → Bool → Bool
(&&)   :: Bool → Bool → Bool
(||)   :: Bool → Bool → Bool
not    :: Bool → Bool
```

Konstanten:

```
True   :: Bool
False  :: Bool
```

Die Datenstruktur der booleschen Werte: (2)

Dem Typbezeichner `Bool` ist die Wertemenge {*True*, *False*} zugeordnet.

`(==)`	bezeichnet die Gleichheit auf Wahrheitswerten
`(/=)`	bezeichnet die Ungleichheit auf Wahrheitsw.
`(&&)`	bezeichnet das logische Und
`(||)`	bezeichnet das logische Oder
`not`	bezeichnet die logische Negation
`True`	bezeichnet den Wert *True*
`False`	bezeichnet den Wert *False*

Bemerkungen:

- Operatorsymbole werden meist mit ***Infix-Notation*** verwendet:
  `34 + 777 , True || False , True && False == True`
- Ist $\odot$ ein Operatorsymbol, kann man $(\odot)$ in Haskell wie einen Funktionsbezeichner verwenden:
  `(+) 34 777 , (||) True False ,`
  `(==) ((&&) True False) True`
- Ist *f* ein (mindestens) zweistelliger Funktionsbezeichner, kann man `` `f` `` in Haskell mit Infix-Notation verwenden:

  ```
  34 `div` 777
  ```

- Im Folgenden unterscheiden wir nur noch dann zwischen Funktionsbezeichner und bezeichneter Funktion, wenn dies aus Gründen der Klarheit nötig ist.

Die Datenstruktur der ganzen Zahlen:

Die *Datenstruktur der ganzen Zahlen* erweitert die Datenstruktur der booleschen Werte, d.h. sie umfasst den Typ `Bool` und die darauf definierten Funktionen. Zusätzlich enthält sie u. a.:

Typ: `Integer`

Funktionen:

`(==), (/=)`	::	`Integer → Integer → Bool`
`(<), (<=), (>), (>=)`	::	`Integer → Integer → Bool`
`(+), (*), (-)`	::	`Integer → Integer → Integer`
`div, mod`	::	`Integer → Integer → Integer`
`negate, signum, abs`	::	`Integer → Integer`

Die Datenstruktur der ganzen Zahlen: (2)

Konstanten:

- in Dezimaldarstellung: 0, 127, -23
- in Hexadezimaldarstellung: 0x0, 0x7F, −0x17
- in Oktaldarstellung: 0o0, 0o177, −0o27

Dem Typbezeichner `Integer` ist die Menge der ganzen Zahlen als Wertemenge zugeordnet.

Die Funktionen der Datenstruktur bezeichnen die üblichen Funktionen auf den ganzen Zahlen; `div` bezeichent die ganzzahlige Division, `mod` liefert den Rest der ganzzahligen Division.

Die Datenstruktur der beschränkten ganzen Zahlen:

Die *Datenstruktur der beschränkten ganzen Zahlen* erweitert die Datenstruktur der booleschen Werte. Zusätzlich enthält sie u. a.:

Typ: `Int`

Funktionen:

`(==), (/=)`	::	`Int → Int → Bool`
`(<), (<=), (>), (>=)`	::	`Int → Int → Bool`
`(+), (*), (-)`	::	`Int → Int → Int`
`div, mod`	::	`Int → Int → Int`
`negate, signum, abs`	::	`Int → Int`

Konstanten:

`minBound, maxBound` :: `Int`

- in Dezimaldarstellung: 0, 127, -23
- in Hexadezimaldarstellung: 0x0, 0x7F, -0x17
- in Oktaldarstellung: 0o0, 0o177, -0o27

Die Datenstruktur der beschränkten ganzen Zahlen: (2)

Dem Typbezeichner `Int` ist eine rechnerabhängige Wertemenge zugeordnet, die mindestens die ganzen Zahlen von -2^{29} bis $2^{29} - 1$ enthalten muss.

Innerhalb der Wertemenge sind die Funktionen der Datenstruktur der beschränkten ganzen Zahlen verlaufsgleich mit den Funktionen auf den ganzen Zahlen.

Außerhalb der Wertemenge ist ihr Verhalten nicht definiert. Insbesondere können $(+)$, $(*)$, `abs`, `negate` partiell sein.

Begriffsklärung:

Wenn unterschiedliche Funktionen oder andere Programmelemente den gleichen Bezeichner haben, spricht man vom ***Überladen*** des Bezeichners (engl. Overloading).

Beispiel: (Überladung von Bezeichnern)

Wie in den obigen Datenstrukturen gezeigt, können Funktionsbezeichner und Operatorbezeichner in Haskell **überladen** werden, d.h. in Abhängigkeit vom Typ ihrer Argumente bezeichnen sie unterschiedliche Funktionen.
Beispiele: `negate`, `(==)`, `(+)`

Die Datenstruktur der Gleitkommazahlen:

Die *Datenstruktur der Gleitkommazahlen* erweitert die Datenstruktur der ganzen Zahlen und bietet u. a.:

Typ: `Float`

Funktionen:

`(==), (/=)`	::	`Float → Float → Bool`
`(<), (<=), (>), (>=)`	::	`Float → Float → Bool`
`(+), (*), (-), (/)`	::	`Float → Float → Float`
`negate, signum, abs`	::	`Float → Float`
`fromInteger`	::	`Integer → Float`
`truncate, round`	::	`Float → Integer`
`ceiling, floor`	::	`Float → Integer`
`exp, log, sqrt`	::	`Float → Float`
`(**), logBase`	::	`Float → Float → Float`
`sin, cos, tan`	::	`Float → Float`

Die Datenstruktur der Gleitkommazahlen: (2)

Konstanten:

```
pi :: Float
```

– mit Dezimalpunkt: 0.0, 1000.0, 128.9, -2.897
– mit Exponenten: 0e0, 1e3, 1289e-1, -2897e-3

Dem Typbezeichner `Float` ist in Haskell eine rechnerabhängige Wertemenge zugeordnet.

Entsprechendes gilt für die präzise Bedeutung der Funktionen und Konstanten.

Bemerkung:

- Die ganzen Zahlen sind in der Programmierung keine Teilmenge der reellen Zahlen!
- In Haskell gibt es weitere Zahlentypen (number types):
  - `Double` (vordefiniert): doppelt präzise Gleitkommazahlen
  - `Rational` (definiert in Standardbibliothek): rationale Zahlen

Die Datenstruktur der Zeichen:

Die *Datenstruktur der Zeichen (engl. character)* erweitert die Datenstruktur der beschränkten ganzen Zahlen. Zusätzlich enthält sie u. a.:

Typ: `Char`

Funktionen:

`(==), (/=)`	::	`Char → Char → Bool`
`(<), (<=), (>), (>=)`	::	`Char → Char → Bool`
`succ, pred`	::	`Char → Char`
`toEnum`	::	`Int → Char`
`fromEnum`	::	`Char → Int`

Die Datenstruktur der Zeichen: (2)

Konstanten:

- in Zeichendarstellung: 'A', 'a', '0', 'ß', ''
- spezielle Zeichen: '\"', '\n', '\t', '\b', '\\'
- in numerischer Darstellung: '\65', '\x41', '\o101'

```
minBound, maxBound :: Char
```

Dem Typbezeichner `Char` ist die Menge der *Unicode-Zeichen* zugeordnet. Jedes Unicode-Zeichen besitzt eine Nummer im Bereich von 0 bis 1.114.111 .

Die Vergleichsoperationen stützen sich auf die Nummerierung.

Die Funktionen `succ` bzw. `pred` liefern das Nachfolger- bzw. Vorgängerzeichen entsprechend der Nummerierung.

Die Funktionen `fromEnum` bzw. `toEnum` liefern die Nummer eines Zeichens bzw. das Zeichen zu einer Nummer.

Die Datenstruktur der Zeichenreihen:

Zeichenreihen sind in Haskell als ***Listen von Zeichen*** realisiert und erweitern die Datenstruktur der Zeichen. Alle auf Listen verfügbaren Funktionen (siehe Datenstruktur der Listen) können für Zeichenreihen verwendet werden, insbesondere:

Typ: `String` oder `[Char]`

Funktionen:

`(==), (/=)`	::	`String → String → Bool`
`(<), (<=), (>), (>=)`	::	`String → String → Bool`
`head`	::	`String → Char`
`tail`	::	`String → String`
`length`	::	`String → Int`
`(++)`	::	`String → String → String`

Die Datenstruktur der Zeichenreihen: (2)

Konstanten:

- Zeichenreihendarstellung in doppelten Hochkommas:

```
"Ich bin ein String!!"
"Ich \098\105\110 ein String!!"
""      (die leere Zeichenreihe)
"Mein Lehrer sagt: \"Nehme die Dinge genau!\""
"String vor Zeilenumbruch \nNach Zeilenumbruch"
```

- Zeichenreihendarstellung als Liste von Zeichen:

```
[ 'H', 'a', 's', 'k', 'e', 'l', 'l']
```

Dem Typbezeichner `String` ist die Menge der Zeichenreihen/Listen über der Menge der Zeichen zugeordnet.

Die Datenstruktur der Zeichenreihen: (3)

Den Vergleichsoperationen liegt die lexikographische Ordnung zugrunde, wobei die Ordnung auf den Zeichen auf deren Nummerierung basiert (siehe Datenstruktur Char).

Bemerkung:

- Es wird unterschieden zwischen Zeichen und Zeichenreihen der Länge 1.
- Jede Programmier-, Modellierungs- und Spezifikationssprache besitzt Basisdatenstrukturen. Die Details variieren aber teilweise deutlich.
- Wenn Basisdatenstrukturen implementierungs- oder rechnerabhängig sind, entstehen Portabilitätsprobleme.
- Der Trend bei den Basisdatenstrukturen geht zur Standardisierung.

Unterabschnitt 3.1.4

Aufbau funktionaler Programme: Ausdrücke

Aufbau funktionaler Programme

Im Kern, d.h. wenn man die Modularisierungskonstrukte nicht betrachtet, bestehen funktionale Programme aus:

- der Beschreibung von Werten:
  - z.B. `(7+23)`, `30`
- Vereinbarung von Bezeichnern für Werte (einschließlich Funktionen):
  - `x = 7;`
- der Definitionen von Typen:
  - `type String = [Char]`
  - `data MyType = ...`

Beschreibung von Werten:

- mittels Konstanten oder Bezeichnern für Werte:

```
23
"Ich bin eine Zeichenreihe"
True
x
```

- durch direkte Anwendung von Funktionen:

```
abs (-28382)
"Kaese" ++ "stinkt"
not True
```

Beschreibung von Werten: (2)

- durch geschachtelte Anwendung von Funktionen:

```
45.67 + 6857 * (-9)
floor (-3.4) * truncate (-3.4)
toEnum(((fromEnum (last("Kaese"++"stinkt")))+2))::
    Char
```

- durch Verwendung des ***bedingten*** Ausdrucks (engl. ***conditional*** expression):

```
if <boolAusdruck> then <Ausdruck>
                  else <Ausdruck>
```

Begriffsklärung: (Ausdruck, expression)

Ausdrücke sind das Sprachmittel zur Beschreibung von Werten. Ein ***Ausdruck*** (engl. ***expression***) in Haskell ist

- eine Konstante,
- ein Bezeichner (Variable, Name),
- die Anwendung einer Funktion auf einen Ausdruck,
- ein bedingter Ausdruck gebildet
- oder ist mit Sprachmitteln aufgebaut, die erst später behandelt werden.

Begriffsklärung: (Ausdruck, expression) (2)

Jeder *Ausdruck hat einen Typ*:

- Der Typ einer Konstanten ergibt sich aus der Signatur.
- Der Typ eines Bezeichners ergibt sich aus dem Wert, den er bezeichnet.
- Der Typ einer Funktionsanwendung ist der Ergebnistyp der Funktion.
- Der Typ eines `if-then-else`-Ausdrucks ist gleich dem Typ des Ausdruck im `then`- bzw. `else`-Zweig.

Unterabschnitt 3.1.5

Operatorenpräzedenz

Präzedenzregeln:

Wenn Ausdrücke nicht vollständig geklammert sind, ist im Allg. nicht klar, wie ihr Syntaxbaum aussieht.

Beispiele:

```
3 == 5 == True
False == True || True
False && True || True
```

Präzedenzregeln: (2)

Präzedenzregeln legen fest, wie Ausdrücke zu strukturieren sind:

- Am stärksten binden Funktionsanwendungen in Präfixform.
- Regeln für Infix-Operatoren:
  ```
  infixl 7 *, /, div, mod
  infixl 6 +, -
  infix 4  ==, /=, <, >, <=, >=
  infixr 3 &&
  infixr 2 ||
  ```
  Je höher die Präzedenzzahl, desto stärker binden die Operationen.
- Mit “infixl”/“infixr” gelistete Operatoren sind links-/rechtsassoziativ, d.h. sie werden von links/rechts her geklammert.
- Mit “infix” gelistete Operatoren müssen geklammert werden.

Unterabschnitt 3.1.6

Deklarationen und Bindungen

Deklaration und Bezeichnerbindung:

Bisher haben wir Ausdrücke formuliert, die sich auf die vordefinierten Funktions- und Konstantenbezeichner von Haskell gestützt haben. Syntaktisch gesehen heißt Programmierung:

- neue Typen, Werte und Funktionen zu definieren,
- die neu definierten Elemente unter Bezeichnern zugänglich zu machen.

Begriffsklärung: (Vereinbarung, Deklaration, Bindung)

In Programmiersprachen dienen ***Vereinbarungen*** oder ***Deklarationen*** (engl. ***declaration***) dazu, den in einem Programm verwendeten Elementen Bezeichner/Namen zu geben.

Dadurch entsteht eine ***Bindung*** (n, e) zwischen dem Bezeichner n und dem bezeichneten Programmelement e.

An allen Programmstellen, an denen die Bindung sichtbar ist, kann der Bezeichner benutzt werden, um sich auf das Programmelement zu beziehen.

Bemerkung:

- Die verschiedenen Arten an Programmelementen, die in Deklarationen vorkommen können, hängen von der Programmiersprache ab.
- In Haskell sind es im Wesentlichen:
  1. Bezeichnervereinbarungen (nur zusammen mit Wert-/Funktionsvereinbarung)
  2. Wertvereinbarungen
  3. Vereinbarungen (rekursiver) Funktionen
  4. Vereinbarungen benutzerdeklarierter Typen
- Die Regeln, die die Sichtbarkeit von Bindungen bzw. Bezeichern festlegen, sind ebenfalls sprachabhängig und können sehr komplex sein. Wir führen die Sichtbarkeitsregeln schrittweise ein.

Wertvereinbarungen:

- Wertvereinbarungen haben (u.a.) die Form:

```
<Bezeichner>  =  <Ausdruck> ;
```

- Wertvereinbarungen kann man eine Bezeichnervereinbarung voranstellen, um den Typ des Bezeichners zu deklarieren:

```
<Bezeichner> :: <Typ> ;
<Bezeichner> =  <Ausdruck> ;
```

  Der Typ des Ausdrucks muss gleich dem vereinbarten Typ sein.

- Der rechtsseitige Ausdruck darf nur sichtbare Bezeichner enthalten.

Beispiele: (Wertvereinbarungen)

```
b = 56 ;

a::Int
a = 7

sieben :: Float ;
sieben = 7.0 ;
flag   =  floor sieben == truncate (- sieben)
dkv    =  "Deutscher Komiker Verein e.v."
```

Beispiele: (Wertvereinbarungen) (2)

Einzeilige Vereinbarungen im Interpreter ghci:

```
let b = 56 ;
```

Mehrzeilige Vereinbarungen im Interpreter ghci:

```
:{
let {
    a::Int ;
    a = 7  ;
    sieben :: Float ;
    sieben = 7.0 ;
    flag   =  floor sieben == truncate (- sieben) ;
    dkv    =  "Deutscher Komiker Verein e.v."
}
:}
```

Funktionsvereinbarungen:

Zwei Probleme:

1. Bisher haben wir keine Ausdrücke, die eine Funktion als Ergebnis liefern (Ausnahme: Funktionsbezeichner)
2. Funktionen können rekursiv sein, d.h. der Funktionsbezeichner kommt im definierenden Ausdruck vor.

Lösungen:

Zu 1. Erweitere die Menge der Ausdrücke, so dass Ausdrücke Funktionen beschreiben können. Dann kann die obige Wertvereinbarung genutzt werden.

Zu 2. Erlaube selbstbezügliche Deklarationen (Haskells Lösung) oder benutze spezielle Syntax für rekursive Funktionsdeklarationen.

Genaueres dazu in Unterabschnitt 3.1.2 (Folien 241ff).

Beispiele: (Funktionsvereinbarungen)

```
myDivision :: Integer -> Integer -> Interger
myDivision =  div

fac :: Integer -> Integer
-- Argument n muss >= 0 sein
fac n =  if n==0 then 1 else n * fac (n-1)

plus2 :: Integer -> Integer
plus2 = (+) 2
```

Typvereinbarungen:

Zwei Probleme:

1. Bisher haben wir keine Ausdrücke, die Typen als Ergebnis liefern.
2. Typen können rekursiv sein, d.h. der vereinbarte Typbezeichner kommt im definierenden Typausdruck vor.

Lösungen:

Zu 1. Führe “Ausdrücke” für Typen ein (z.B. `Int -> Int`).

Zu 2. Benutze spezielle Syntax für rekursive Typdeklarationen.

Genaueres dazu in Unterabschnitt 3.1.4. (Folien 298ff).

Beispiele: (Typvereinbarungen)

```
type IntPaar        =  (Int,Int) ;
type CharList       =  [Char] ;
type Telefonbuch    =
        [((String,String,String,Int),[String])] ;

type IntegerNachInteger  =   Integer -> Integer ;

fakultaet :: IntegerNachInteger ;
-- Argument muss >= 0 sein
fakultaet = fac ;
```

Begriffsklärung: (Bezeichnerumgebung)

Eine ***Bezeichnerumgebung*** ist eine Abbildung von Bezeichnern auf Werte (einschl. Funktionen) und Typen, ggf. auch auf andersartige Programmelemente.

Oft spricht man auch von ***Namensumgebung*** oder einfach von ***Umgebung*** (engl. ***environment***).

Bemerkungen:

- Programmiersprachen stellen üblicherweise eine Standard-Umgebung bereit mit den vordefinierten Programmelementen (Werten, Funktionen, Typen, etc.). In Haskell ist die Standard-Umgebung durch das Modul `Prelude` definiert.
- Eine Bezeichnerumgebung wird häufig als Liste von Bindungen modelliert (vgl. Folie 226).
- Jede Datenstruktur und jedes Modul definiert eine Bezeichnerumgebung.

Unterabschnitt 3.1.7

Programme

Begriffsklärung: (Programm)

Der Programmbegriff entsteht aus Benutzesicht: Software, die irgendwie ein Problem löst; an den Details ist der Benutzer i.A. nicht interessiert.
Ein ***Programm*** besteht in der Regel aus

- einer Menge von Deklarationen und
- einer (durch einen besonderen Namen) ausgezeichneten Funktion bzw. Prozedur (oder ähnlichem Konstrukt), die angibt, wie die Auswertung bzw. Ausführung des Programms zu starten ist.

Beispiel: (funktionales Programm)

```
import System.IO

fac :: Integer -> Integer
-- Argument n muss >= 0 sein
fac n =  if n==0 then 1 else n * fac (n-1)

main = do {
  hSetBuffering stdout NoBuffering;
  putStr "Eingabe x (x>=0): ";
  a <- readLn;
  putStr "Ergebnis (fac x): ";
  print (fac a);
}
```