III.2 Deklarationen

Dienstag, 12. Dezember 2017 10:00

Grundlegende Sprachkonstrukte von Haskell:

- Deklarationen
- Ausdrücke
- Patterns (Muster)
- Typen

2 Arten von Deklarationen:

- Typdeklaration
  (legt Definitions- und Wertebereich v. Funktionen fest)
- Funktionsdeklaration
  (legt Abbildungsvorschrift fest).

Kommentare in Haskell

-- ... Zeilenende

oder

{-
⋮
-}

Typdeklarationen

- Variablenbezeichner dienen auch als Funktionssymbole. Bsp. Strings, die mit Kleinbuchstaben anfangen, z.B. len, square.

```
square :: Int -> Int
```

(Int: Definitionsbereich; Int: Wertebereich)

- Typen: Int, Bool, Char, Float, Double, ...

[Int] : Typ der Listen v. ganzen Zahlen

[Bool]

`[[Int]]` : Typ der Listen v. Listen v. ganzen Zahlen.

z.B. `[[1,2], [3], []]`.

`Int -> Int` : Typ der Funktionen von Int nach Int

`[Int] -> Int` : ...

`[Int -> Int]` : ...

z.B. `[square, square, double]`

Typdeklarationen können weggelassen werden. Dann berechnet Haskell sie automatisch.
Aber: Es ist guter Prog-Stil, sie anzugeben.

Funktionsdeklarationen

```
square x = x * x
```

square x = x * x

Funktions-name

Form des erwarteten Arguments (Pattern)

Aus-druck, Ergebnis d. Funktion

Vordef. arithmet Grundoperationen:

+, −, *, /

Vordef. Vergleichsoperationen:

==, >, >=, ...

Vordef. Operationen auf Datenstruktur Bool (True, False):

&&, ||, not, ...

Man kann auch Funktionen ohne Argumente deklarieren (Konstanten):

```
one :: Int
one = 1
```

Ausführung eines fkt. Programms

Termersetzung:

- finde linke Seite einer Gleichung, die auf einen Teilausdruck des auszuwertenden Ausdrucks passt
  (wenn man die Var. der linken Seite geeignet instantiiert)
  Pattern Matching
- dann ersetze diesen Teilausdruck durch die entsprechend instantiierte rechte Seite

Es können mehrere Auswertungsschritte möglich sein. Auswertungsstrategie ent-

scheidet, welcher Schritt gewählt wird.

Java: macht immer call-by-value (call-by-reference kann durch Referenztypen simuliert werden).

Vor- und Nachteile

- strikte Auswertung ist schneller als nicht-strikte Auswertung, wenn man duplizierte nicht-ausgewertete Teilausdrücke anschließend separat auswertet.
-
```
three :: Int -> Int
three x = 3

non_term :: Int -> Int
non_term x = non_term (x+1)
```

three (non_term 0)

- terminiert nicht bei strikter Auswertung
- terminiert in 1 Schritt bei nicht-str. Auswertung und liefert 3.

Nicht-strikte Auswertung wertet Argumente nur dann aus, wenn sie benötigt werden (⇒ oftmals schneller als strikte Auswertung).

- Wenn irgendeine Auswertung terminiert, dann terminiert auch die nicht-strikte Auswertung.
- Wenn die strikte Auswertung terminiert, dann

terminiert jede Auswertung

- Alle Auswertungen, die terminieren, liefern dasselbe Ergebnis.

Haskell benutzt Lazy Evaluation

- nicht-strikte Auswertung (leftmost outermost)
- Aber: Wenn Ausdruck dupliziert wird, dann wird dies in Haskell durch Pointer realisiert und die duplizierten Ausdrücke werden dadurch parallel ausgewertet.

Bedingte Gleichungen + Tupel

(Int, Int) : Typ der Paare wobei beide Komponenten

den Typ Int haben.

d.h.: maxi hat den Typ $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

(Int→Int, [Bool], Int) ist auch ein Typ.

Bsp: (square, [True, False], 5)

Bedingte Gleichungen durch "|".

Dabei ist "otherwise" vordefiniert als:

```
otherwise :: Bool
otherwise = True
```

Mehrere Gleichungen für die gleiche Funktion werden von oben nach unten abgearbeitet und die erste passende Gleichung wird benutzt.

Currying

benannt nach Logiker

Haskell B. Curry

plus :: Int -> Int -> Int

bedeutet Int -> (Int -> Int)

plus 2 hat also den Typ Int -> Int.

Dies ist die Funktion, die Zahlen um 2 erhöht.

plus 2 3 = 2+3 = 5

Vorteile des Currying

- Spart Klammern
- erlaubt partielle Anwendungen auf "zu wenige" Argumente.
- Erst wenn alle benötigten Argumente da sind, kann Funktion ausgewertet werden.

```
suc :: Int -> Int
suc = plus 1
```

Dann kann man folgendermaßen auswerten:

suc 5

= plus 1 5

= 1 + 5

= 6

Funktionsdefinition durch Pattern Matching

- Mehrere definierende Gleichungen für dasselbe Funktionssymbol.
- Pattern beschreiben die erwartete Form des Arguments.
- Gleichungen werden v. oben n. unten überprüft.

- Jede Datenstruktur hat bestimmte Datenkonstruktoren, mit denen alle Elemente der Datenstruktur gebildet werden können. Pattern Matching führt Fallunterscheidung danach durch, mit welchem Datenkonstruktor ein Wert gebildet wurde.

Datenkonstruktoren von Bool:

True, False

Datenkonstruktoren von Listen:

[], :

D.h.: Pattern ≈ Ausdruck aus Datenkonstruktoren und Variablen

Lokale Deklarationen

Bsp: Gegeben a, b, c

Gesucht x, so dass

$$a \cdot x^2 + b \cdot x + c = 0$$

Lösung:

$$x = \frac{-b \pm \underbrace{\sqrt{b^2 - 4 \cdot a \cdot c}}_{d}}{\underbrace{2 \cdot a}_{e}}$$

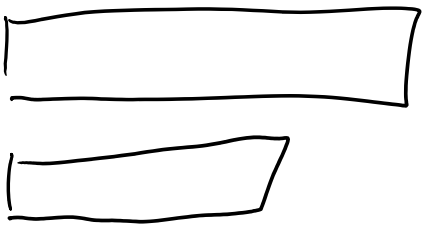
- Durch "where" wird ein lokaler Deklarationsblock eingeleitet (nur in dieser rechten Seite sichtbar).
- Vorteil: kürzer + effizienter (mehrfache Vorkommen von lokal deklarierten Ausdrücken werden parallel ausgewertet)

Man kann lokale Deklarationen ohne { ... } und ; .

schreiben durch Verwendung der Offside-Regel:

1. Das erste Symbol in Sammlg. v. Deklarationen bestimmt den linken Rand des Deklarationsblocks.

2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration in diesem Block.



3. Eine neue Zeile, die weiter rechts beginnt, ist eine Fortsetzung der Deklaration in der vorigen Zeile.

```
d = sqrt(b * b
    - 4 * a * c)
```

4. Eine neue Zeile, die weiter links beginnt als vorige Zeile, bedeutet, dass der momentane Deklarationsblock beendet ist. Neue Zeile gehört nicht mehr zu diesem Deklarationsblock.