III.2. Deklarationen

Mittwoch, 5. Dezember 2018 09:30

Typdeklarations:

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

(square ist Fkt. von $\mathbb{Z} \to \mathbb{Z}$)

```
len :: [Int] -> Int
```

↑ Datentyp der Listen von Integern

- Layout hat in Haskell eine Semantik (genauer in nächster Vorlesung).
- Typdeklarationen müssen nicht geschrieben werden (dann inferiert Haskell sie selbst). Aber: Typdekl. sind guter Stil.
- Kommentare in Haskell:

-- bis Zeilenende oder

{-

⋮

-}

- Prog: linksbündige Folge von decls.
- Als Funktionssymbole werden Variablenbezeichner var benutzt.
- var: beliebige Strings aus Bchst. u. Zahlen mit Kleinbuchstaben am Anfang
- type: Int, Bool, Double, Float, ...

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

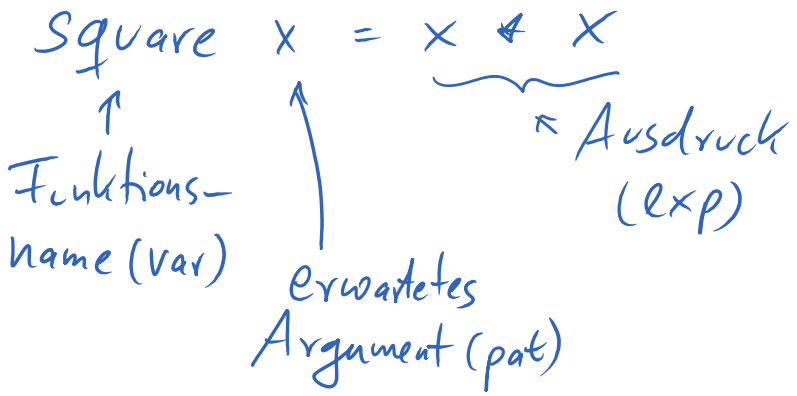
  [Int]: Typ der Listen mit Elementen v. Typ Int

[[Int]] : z.B.

[[15,70], [36], [3]]

[Int → Int] : [square, ...]

Funktionsdeklaration:



Pattern: spezielle Ausdrücke für erwartete Argumente, z.B. Variablen, [], kopf : rest, ...

Vordef. Funktionen in Bibliotheken,
Standard-Bibliothek: Prelude

- +, -, *, /, ...
- >, <, >=, <=, ==, ...
- Datenstruktur Bool mit &&, ||, not, ...

Es sind auch Funktionen

ohne Argumente möglich:

```
one :: Int

one = 1
```

Auswertung von Ausdrücken in Haskell

durch Termersetzung

reducible expression ↓ "Redex"

1. Suche Teilausdruck, der mit linker Seite einer definierenden Gleichung übereinstimmt, wenn Var. der Gleichung geeignet instantiiert werden. ← "Pattern Matching"
2. Ersetze diesen Teilausdruck durch rechte Seite der def. Gleichung, wobei Variablen

wie in Schritt 1 instantiiert werden.

Auswertungsstrategie

legt fest, welcher Teilausdruck zuerst ausgewertet werden soll.

Call-by-value =
eager Auswertung =
strikte Auswertung

wird in Java u. den meisten anderen Prog-spr. benutzt

Vorteil von strikter Auswertung: Duplizierte Teilausdrücke müssen nicht mehrfach ausgewertet werden.

Nachteil von strikter Auswertung: Argumente werden auch dann aus-

gewertet, wenn sie gar nicht für Gesamtergebnis gebraucht werden.

```
three :: Int -> Int
three x = 3

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

three (non_term 0)
```

strikte Auswertung: terminiert nicht

nicht-strikte Auswertung: liefert 3 in einem Auswertungsschritt

Haskell verwendet Lazy Evaluation:

- im Prinzip nicht-strikte Auswertung (leftmost outermost)
- Wenn ein Teilausdruck bei

der Auswertung dupliziert wird, dann "merkt" Haskell das u. wertet diese Teilausdrücke später gleichzeitig aus.

Generell:

- Wenn die strikte Auswertung terminiert, dann terminiert jede Auswertung.
- Jede terminierende Auswertung (beim gleichen Start-Ausdruck) hat das gleiche Ergebnis.

Man kann die outermost-Auswertungsstrategie von Haskell nutzen, um Programme mit unendlichen Datenstrukturen zu schreiben (→ später).

Tupel + bedingte def. Gleichungen

Tupel-Typen: (Int, Int)

ist der Typ der Paare von Int-Zahlen ($\hat{=} \mathbb{Z} \times \mathbb{Z}$).

(3,5) hat den Typ (Int, Int)

([1,2], True, 5) hat den Typ

([Int], Bool, Int)

Bedingte Gleichungen:

Def. Gleichung kann mit boolescher Bedingung eingeschränkt werden.

Die Gleichungen werden von oben nach unten getestet und die erste passende wird ausgeführt.

"otherwise" ist vordefiniert:

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

Currying

benannt nach Logiker

Haskell B. Curry

Untere plus-Version:

```
plus :: Int -> (Int -> Int)
```

(Kann man weglassen, denn "->" assoziiert nach rechts)

plus ist jetzt eine einstellige Funktion.

plus 2 ist eine Funktion von Int -> Int die Zahlen um 2 erhöht.

(plus 2) 3 = 2+3 = 5

↑ ↑

Kann man weglassen, denn Funktionsanwendung assoziiert nach links.

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

Auswertung von:

```
  suc 5
= plus 1 5
= 1 + 5
= 6
```

Vorteil des Currygings:

- weniger Klammern
- partielle Anwendung von Funktionen (z.B. Anwendung von plus auf nur 1 Argument)

Funktionsdefinition durch Pattern Matching

Man kann mehrere def.
Gleichungen für die gleiche
Fkt haben, die sich durch
die Form der erwarteten
Argumente unterscheiden
(d.h. unterschiedliche Patterns).

True, False: Datenkonstruk-
toren der Datenstruk-
tur Bool.

Datenkonstruktoren sind Funk-
tionssymbole zur Darstellung
der Werte einer Datenstruk-
tur. Sie werden nicht weiter
ausgewertet (beginnen in
Haskell mit Großbuchstaben).

Patterns ≙ Ausdrücken aus
Variablen und Daten-
konstruktoren.

Haskell überprüft von oben nach unten, welche definierende Gleichung passt und wendet die erste passende Gleichung an.

Datenkonstruktoren für Listen: `[]`, `:`

Lokale Deklarationen

Bsp: Gegeben a, b, c

Gesucht: x, so dass

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

Lösung:

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

"where" leitet einen lokalen Deklarationsblock ein, der nur in der entsprechenden

rechten Seite der def. Gleichung sichtbar ist.

Realisierung: wird durch Pointer realisiert, d.h.:
Wenn d ausgewertet wird, dann wertet es gleichzeitig an allen Stellen aus, an denen es in der rechten Seite auftritt.

locdecls = Sammlung von (lokalen) Deklarationen

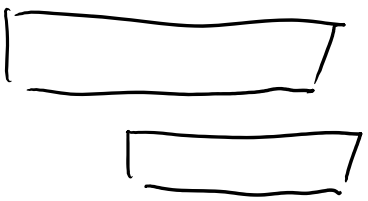
Haskell nutzt das Layout des Programms aus, um Klammern zu vermeiden und damit die Lesbarkeit zu erhöhen ("offside-Regel"):

1. Das erste Symbol in einer Sammlung von 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 als der linke Rand gehört zur selben Deklaration wie die Zeile darüber.



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

4. Eine neue Zeile, die

weiter links beginnt, beendet den Deklarations-Block (d.h. ist neue Deklaration, die nicht zum Block der Zeile darüber gehört).

⇒ Dann kann man auf { ... } und ; verzichten.