III.2 Deklarationen

Mittwoch, 11. Januar 2017 09:00

Grundlegende Sprachkonstrukte von Haskell:

- Deklarationen (III.2)
- Ausdrücke (III.3)
- Patterns (III.4)
- Typen (III.5)

- Eigenart von Haskell:
  Layout hat Auswirkungen auf die Semantik.
- 2 Arten von Deklarationen
  Typdeklarationen (legen Definitionsbereich + Wertebereich v. Fkt fest)
  Funktionsdeklarationen (genaue Abbildungsvorschrift)
- Syntax für Kommentare:
  `--` ... bis Zeilenende
  oder
  `{-` ⋮ `-}`
- Typdeklarationen:

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

Variablen-bezeichner (String, der mit Kleinbuchstaben beginnt)

Typ (z.B. Int, Bool, Char, Int -> Int, [Int], [Int] -> [Bool] [Int -> Bool], ...)

Typdeklarationen müssen nicht mit angegeben werden (dann berechnet Haskell ihn selbst). Aber für Lesbarkeit ist es guter Stil, zu jeder Funktion auch die Typdekl. mit anzugeben.

- Funktionsdeklarationen

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

square: Variablenbezeichner

x: Pattern (beschreibt bestimmte Arten von

x * x: Ausdruck

Eingaben)

Hierbei müssen die Typen passen.

Auf diese Weise kann man auch Konstanten definieren.

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

In Ausdrücken können neben selbstdefinierten Funktionen auch vordef. Funktionen auftreten: +, -, *, /, ...

<, >, <=, >=, ==, ...

&&, ||, not, ...

Ausführung v. fkt. Programmen

Termersetzung:

- Computer sucht Teilausdruck der der linken Seite einer definierenden Gleichung entspricht. Hierbei müssen die Variablen durch geeignete Ausdrücke instantiiert werden.
- Dann ersetzt man diesen Teilausdruck durch die entsprechend

instantierte rechte Seite.

Auswertungsstrategie entscheidet, welcher Teilausdruck als nächstes ausgewertet wird.

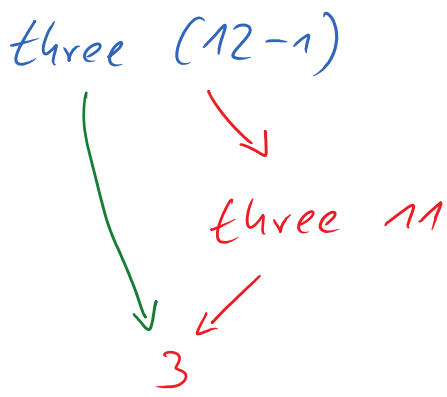
- strikte Auswertung
  (eager evaluation,
  leftmost innermost evaluation,
  call-by-value):
  Werte immer soweit innen links aus, wie möglich.
- nicht-strikte Auswertung
  (leftmost outermost evaluation,
  call-by-name)

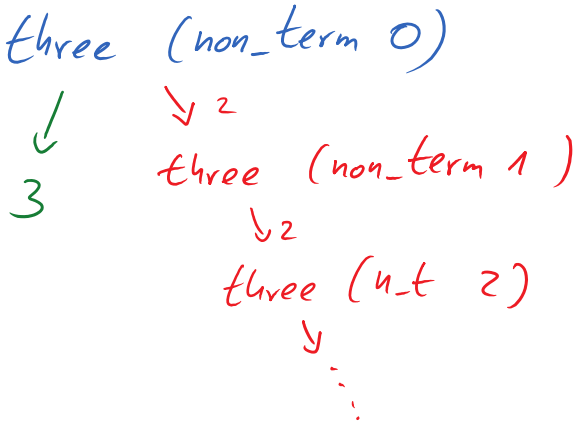
- Die Auswertungsstrategie beeinflusst nicht das Ergebnis (wenn man ein Ergebnis erhält, dann ist es immer dasselbe, unabh. v.d. Auswertungsstrategie).
- Aber: Die Auswertungsstrategie beeinflusst die Länge und das Terminierungsverhalten der Auswertung.

- Vorteil der strikten Auswertung: Man vermeidet mehrfache Auswertungen von duplizierten Ausdrücken (im Bsp benötigt strikte Ausw. nur 3 statt 4 Schritte)
- Vorteil der nicht-strikten Auswertung: Es werden nur Teilausdrücke ausgewertet, die für das Ergebnis benötigt werden.

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

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





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

Haskell macht Lazy Evaluation:

- nicht-strikte Auswertung (leftmost outermost)
- Haskell realisiert duplizierte Ausdrücke durch Pointer, so dass diese nur einmal ausgewertet werden.

Es existieren aber auch viele fkt. Sprachen mit innermost Strategie (ML, Lisp, Scheme, ..)

- Mehrere Eingaben + bedingte Gleichungen

(Int, Int) ist der Typ der 2-Tupel, bei dem beide Komponenten den Typ Int haben ($\hat{=}$ Int x Int)

z.B. auch (Bool, [Int], Int->Int). Dies wäre der Typ des Ausdrucks (True, [1,2,3,4], square).

Gleichungen können durch Bedingungen eingeschränkt werden. Bedg. werden von oben u. unten überprüft und

man erhält das Ergebnis der ersten Gleichung, bei der Bedg. zu True auswertet. ("otherwise" ist vordef. und liefert immer True).

- Currying

(benannt nach Logiker Haskell B. Curry)

(analog dazu spricht man von Currying und Uncurrying)

Kann man weglassen, "→" assoziiert nach rechts

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

plus x y = x + y
```

plus x y stehe für (plus x) y

Klammern kann man weglassen, denn Funktionsanwendung assoziiert nach links

plus 2 : Int → (Int → Int), Int

hat Typ Int → Int

Wenn (plus 2) ein Arg. 5 bekommt, wird es auf 2+5 abgebildet

Vorteile von Currying

- Funktionen können partiell angewendet werden:

```
suc :: Int -> Int

suc = plus 1
```

Dann könnte man folgendes auswerten:

suc 7
= plus 1 7
= 1+7
= 8

- Currying spart Klammern

- Funktionsdefinition durch Pattern Matching
  - mehrere Funktionsdeklarationen für das gleiche Funktionssymbol
  - Patterns beschreiben die Form des erwarteten Arguments
  - Patterns ≈ Ausdrücke aus Variablen und Datenkonstruktoren
  - Datenkonstruktoren: spezielle Funktionssymbole, die nicht weiter ausgewertet werden. Diese Funktionssymbole dienen dazu, alle Objekte

einer Datenstruktur
zu repräsentieren,
z.B. True, False
sind die Datenkonstruktoren
der Datenstruktur Bool.

Datenkonstruktoren beginnen
(normalerweise) mit Großbuch-
staben. Auswertbare Funk-
tionen beginnen (normaler-
weise) mit Kleinbuchstaben.

- Pattern Matching:
Um einen Ausdruck $f\ t_1 \dots t_n$
auszuwerten, werden die
definierenden Gleichungen von f
von oben nach unten überprüft und
man verwendet die erste Gleichung

$$f\ pat_1 \dots pat_n = exp,$$

bei der die Pattern "passen",
d.h., bei der die Variablen so in-
stantiiert werden können, dass
pat_1 zu t_1 wird, ..., pat_n zu t_n
wird.
- Pattern gibt an, mit welchen
Datenkonstruktoren ein Ar-

gument gebildet wurde.
Bei Listen: Patterns erlauben eine Fallunterscheidung anhand der Datenkonstruktoren [] und :

[1,2] ist nur eine Kurzschreibweise für
1 : (2 : [])
↑ ↑
Klammern kann man weglassen, : assoziiert nach rechts

- Lokale Deklarationen

Bsp: $a, b, c \in \mathbb{Q}$ oder $\mathbb{R}$

Gesucht: Für welche x gilt

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

Schule:

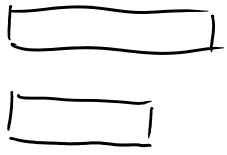
$$x = \frac{-b \pm \overbrace{\sqrt{b^2 - 4ac}}^{d}}{\underbrace{2a}_{e}}$$

Vorteil: Haskell wird dadurch effizient, da die beiden

duplizierten Vorkommen von d (und von e) parallel ausgewertet werden.

Haskell hat die sogenannte Offside-Regel, mit der man "{", "}", ";" sparen kann. D.h: das Layout des Programms beeinflusst die Semantik:

1. Das erste Symbol einer Sammlung von Deklarationen bestimmt den linken Rand des Deklarationsblocks.
2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration des gleichen Blocks.



3. Eine neue Zeile, die weiter rechts als der linke Rand beginnt, gehört zur selben Deklaration.

```
d = sqrt (bob -
```

4 * a * c)

e = 2 * a

4. Eine neue Zeile, die weiter links beginnt, beendet den aktuellen Deklarationsblock. Sie gehört nicht mehr zu diesem Dekl-Block.