

Algorithmen und Programmierung

3. Aufgabenblatt

Aufgabe 3.1

- a) Funktionsnotation bedeutet Präfixschreibweise.

$$\begin{array}{ll} a + b \cdot 3 \cdot c - (d + 3) & \text{Infixschreibweise} \\ = (-) ((+) a ((\cdot) ((\cdot) b 3)c) ((+) d 3) & \text{Präfixschreibweise} \end{array}$$

Da + und * kommutativ sind, gibt es mehrere korrekte Lösungen

- b) $3+2*5$ ist **korrekt**.

```
*Temp> 3+2*5  
13
```

- c) $(+) 3 (2*5)$ ist **korrekt**

```
*Temp> (+) 3 (2*5)  
13
```

- d) $(+) 3 (2^*) 5$ ist **falsch**

$(+)$ ist eine Funktion, die als 2ten Parameter eine Funktion (2^*) bekommt
=> Fehler

- e) $(+) 3 2*5$ ist **korrekt**

```
*Temp> (+) 3 2*5  
25
```

- f) $(3+) (2+) 5$ ist **falsch**

Der Abschnitt $(3+)$ bekommt eine Funktion als Eingabe $(2+)$ => Fehler

- g) $(3+2^*) 5$ ist **falsch**

* bindet stärker als +. Daher wird versucht das * zuerst aufzulösen. Auf Grund der Klammerung ist das nicht möglich => Fehler

- h) $(3+2) * 5$ ist korrekt

```
*Temp> (3+2) *5  
25
```

- i) Gemeinsame Abstraktionen von $2^7 + 2^3 + 1$ und $2^9 + 2^2 + 1$ sind:

1te Abstraktion: $(\lambda x \rightarrow \lambda y \rightarrow 2^x + 2^y + 1)$
2te Abstraktion z.B.: $(\lambda x \rightarrow \lambda y \rightarrow \lambda f \rightarrow 2^x `f 2^y `f 1)$

3te Abstraktion z.B.: $(\lambda x \rightarrow \lambda y \rightarrow \lambda f \rightarrow \lambda z \rightarrow z^x \text{ `f } z^y \text{ `f } 1)$

j) Gemeinsame Abstraktionen von 3^2+5 und 3^3*5

1te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow 3^a \text{ `f } 5)$
2te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda b \rightarrow b^a \text{ `f } 5)$
3te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda g \rightarrow 3 \text{ `g` a `f } 5)$
4te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda c \rightarrow 3^a \text{ `f } c)$
5te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda b \rightarrow \lambda g \rightarrow b \text{ `g` a `f } 5)$
6te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda c \rightarrow \lambda g \rightarrow 3 \text{ `g` a `f } c)$
7te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda b \rightarrow \lambda c \rightarrow b^a \text{ `f } c)$
8te Abstraktion: $(\lambda a \rightarrow \lambda f \rightarrow \lambda b \rightarrow \lambda c \rightarrow \lambda g \rightarrow b \text{ `g` a `f } c)$

Aufgabe 3.2

Definition von power

$2^n = 2^{2n/2} = (2^{(n/2)})^2$ für gerade n
 $2^n = 2^{2n/2+1} = 2(2^{(n/2)})^2$ für ungerade n

Implementierung

```
--Implementierung von power
-- n > 0
power x 0 = 1
power x n | odd n = 2 * ((power x k) ^ 2) -- ungerade n
          | otherwise = (power x k) ^ 2 -- gerade n
          where k = div n 2

-- Kurzschreibweise von power
-- n > 0
shortpower x 0 = 1
shortpower x n = 2 ^ (mod n 2) * ((power x (div n 2)) ^ 2)

-- Langsame Implementierung von power
-- n > 0
slowpower x 0 = 1
slowpower x n = 2 * slowpower x (n-1)
```

Testläufe

```
*Programme> :set +s

*Programme> power 2 100000
9990020930143845079440327643300335909804291390541816917715
2927386314583246425734832748733133244965040316439444555585
4930018799660765617656290847135424749...
(1.99 secs, 243190232 bytes)

*Programme> slowpower 2 100000
```

```

9990020930143845079440327643300335909804291390541816917715
2927386314583246425734832748733133244965040316439444555585
4930018799660765617656290847135424749...
(20.58 secs, 1044048796 bytes)

```

Aufgabe 3.3

a)

```

<Prog_stützen> := <stützregel>; | <stützregel>;<Prog_stützen>
<stützregel>   := <Prog_bezeichner> -> <stützen>
<stützen>     := <bez> | <bez>,<stützen>
<bez>         := <zeichenkette> |<zeichenkette>'
<<zeichenkette> := .....

```

b) Algorithmus "stützen eines Programms P", Parameter

```

stützen {} = {}
stützen {P,Q,R,...} = s(P) ∪ stützen {Q,R,...}
  wobei s(P) = {} wenn keine Regel P -> ... existiert
             s(P) = stützen {P1,P2,...} wenn P1,P2,... auf der
                rechten Seite der Regel vorkommen, deren linke Seite P ist.

```

{...} bezeichnen Multimengen oder Listen, das heißt ein Element kann mehrfach vorkommen.

c) Statt Multimengen oder Listen Mengen als Datenstruktur wählen! Damit ist gesichert, dass kein P mehrfach auftritt. In einer Pseudocode-Implementierung ist ein solcher "Trick" erlaubt. Da Programmiersprachen in aller Regel von Haus aus keine Mengen sondern nur Listen kennen, müssen Mengen aber in der Implementierung programmiert werden. Das kann auf ganz unterschiedliche Arten geschehen.

Aufgabe 3.4

- a) Im Tutorium besprochen
- b) Curry und Uncurry

Eselsbrücke

Curry-Notation
 add a b = (add a) b d.h. durch Currying auswertbar.

Nicht in Curry-Notation
 add (a,b) d.h. nicht durch Anwenden der einzelnen
 Parameter a und b auswertbar.

Implementierung

```

-- Curry nimmt eine Funktion in Nicht-Curry-Notation
-- und bringt sie in Curry-Notation
haskellCurry :: ((a, b) -> c) -> a -> b -> c

```

```

haskellCurry f a b = f(a,b)

-- Uncurry nimmt eine Funktion in Curry-Notation
-- und bringt sie in Nicht-Curry-Notation
haskellUncurry :: (a -> b -> c) -> ((a, b) -> c)
haskellUncurry f (a,b) = f a b

add1 (x,y) = x+y
add2 x y   = x+y

```

Testläufe

```

*Programme> :t haskellCurry add1
haskellCurry add1 :: forall t2. (Num t2) => t2 -> t2 ->
t2

```

add1 ist somit in Curry-Notation und kann auf 2 Parameter angewandt werden:

```

*Programme> add1 3 4
...Fehler...
*Programme> haskellCurry add1 3 4
7

```

```

*Programme> :t haskellUncurry add2
haskellUncurry add2 :: forall t2. (Num t2) => (t2, t2) -
> t2

```

add2 ist somit in Nicht-Curry-Notation und kann auf ein Tupel angewandt werden:

```

*Programme> add2 (3,4)
...Fehler...
*Programme> haskellUncurry add2 (3,4)
7

```

Aufgabe 3.5

- a) Newton Iteration siehe Musterlösung auf der Veranstaltungshomepage
<http://www.inf.fu-berlin.de/lehre/WS04/ALP1/uebungen/newton.lhs>
<http://www.inf.fu-berlin.de/lehre/WS04/ALP1/uebungen/testNewton.txt>