Fakultät IV – Elektrotechnik und Informatik
MPGI 1 WS 2013/14
Pepper / Höger / Reicherdt / Zuber / Tutoren

Aufgabenblatt 6

letzte Aktualisierung: 20. 11, 13:18

Ausgabe: 20.11.2013
Abgabe: 28./29.11.2013

Thema: *Higher-Order Functions, Currying*

Hinweis: Ab diesem Aufgabenblatt erwarten wir von euch ein Testmodul, das Testfunktionen ähnlich der Hausaufgabe 3.2 des 5. Aufgabenblattes enthält. Es soll alle eure Module der Hausaufgabe importieren und deren Funktionen mit geeigneten Testwerten füttern und nachvollziehbare Ausgaben liefern.

1. Aufgabe: Funktionen höherer Ordnung (HOF)

Bei *Funktionen höherer Ordnung* (auch Higher-Order Functions oder Funktionale genannt) geht es darum, dass Funktionen selbst Parameter oder Resultate einer anderen Funktion sein können. Sie können genau wie andere Objekte (Zahlen, Zeichenketten...) behandelt werden. Anders als Funktionen in vielen anderen Sprachen oder auch Typen in Opal sind sie somit *First Class Citizens*.

1.1. (Tut) Funktionen als Parameter I Schreibt eine Funktion `apply`, die als Parameter zwei andere Funktionen sowie einen Wert aus `nat` erhält, und die beiden Ergebnisse der Funktionen für diesen Eingabewert als Tupel zurückgibt.

1.2. (Pflicht-Hausaufgabe) Funktionen als Parameter II Schreibt eine Funktion `applyToChunk`, die eine übergebene Funktion zeichenweise auf einen Ausschnitt einer übergebenen Denotation anwendet. Der Ausschnitt wird durch zwei übergebene Indizes ausgewählt.
Die Deklaration ist wie folgt:

```
FUN applyToChunk : denotation**(char → char)**nat**nat → denotation
```

Beispiel: `> e applyToChunk("lalala",upper,1,3)` ⇝ `lALAla`

1.3. (Tut) Produkt von Funktionsapplikationen Definiert eine Funktion `prodFun`, welche eine Funktion `f: nat -> nat` und zwei Grenzen `start` und `end` eines natürlichen Zahlenintervalles $[start, end] \subset \mathbb{N}$ übergeben bekommt. Die Funktion wendet `f` auf die Zahlen im Intervall an, multipliziert die Funktionsergebnisse $f(start), \ldots, f(end)$ auf und gibt dieses Produkt als Ergebnis zurück.

$$prodFun(f, start, end) = \prod_{i=start}^{end} f(i)$$

Welches Ergebnis liefert der Aufruf von `> e { prodFun(\\x.x,1,5) }`?
Zeigt das Zustandekommen des Ergebnisses.

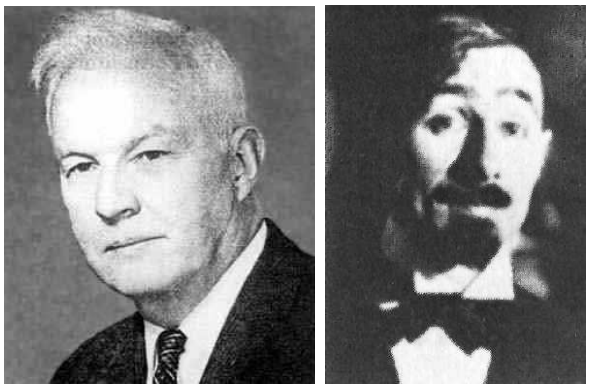

Abbildung 1: Haskell Brooks Curry und Moses Schönfinkel

Hinweis: Der Begriff **Currying** ist nicht etwa eine Anlehnung an die gleichnamige Gewürzmischung, sondern ehrt den amerikanischen Mathematiker **Haskell Brooks Curry**, der 1930 in Göttingen mit der Arbeit „Grundlagen der kombinatorischen Logik“ promovierte (`http://www.haskell.org/bio.html`). Curry selbst schrieb die Idee dem russischen Mathematiker **Moses Schönfinkel** zu (man könnte also auch vom **Schönfinkeln** sprechen). Das Prinzip ist noch älter und war bereits im 19. Jahrhundert **Frege** und **Cantor** bekannt (vgl. B. C. Pierce: „Types and Programming Languages“, S. 73).

2. Aufgabe: Funktionen als Resultate anderer Funktionen

In den bisherigen Aufgaben waren Funktionen nur Eingaben für andere Funktionen. Natürlich können Funktionen aber auch als Rückgabewerte auftreten.

2.1. (Tut) Funktionskomposition Deklariert und definiert eine Funktion `o`, die zwei Funktionen miteinander kombiniert und die dabei entstehende Funktion als Resultat liefert. Definitions- und Wertebereich beider Eingabe-Funktionen soll `nat` sein. Die erste Funktion soll nach der zweiten angewendet werden, d.h.
$(f \circ g)(x) = f(g(x))$.

Welche Ergebnisse erwartet ihr bei folgenden Aufrufen:

- `> e (double o half) (1)`
- `> e (half o double) (1)`
- `> e (double o half)`

2.2. (Pflicht-Hausaufgabe) Mehrfachanwendung Deklariert und definiert eine Funktion `^`, die eine Funktion n-fach mit sich selbst komponiert. Definitions- und Wertebereich der Eingabefunktion sind `nat`. Die 0-fache Anwendung einer Funktion soll dabei der Identität der entsprechen, also $(f^0)(x) = x$.

Welches Ergebnis liefert der Aufruf `> e (half ^ 2) (16)`?
Führt eine Handsimulation wie in Aufgabe 2.1 durch und gebt diese als Textdatei mit ab!

2.3. (Tut) Currying Betrachten wir eine Definition für die Potenzfunktion `pow`:

```
FUN pow : nat ** nat -> nat
DEF pow(x,y) ==
    IF y = 0
    THEN 1
    ELSE x * pow(x,y-1)
    FI
```

Schreibt diese Funktion nun in Curry-Schreibweise! Schreibt sowohl die Funktions- als auch die Lambdaschreibweise auf!

2.4. (Tut) Currying angewandt Letzte Woche habt ihr die Mersenne-Zahlen $M(p) = 2^p - 1$ kennengelernt. Definiert eine Funktion `mersenne`, die $M(p)$ berechnet, ausschließlich mithilfe von `o`, `pred` und `pow` von diesem Aufgabenblatt!

Wie sieht die Definition mit und wie sieht sie ohne Currying bei `pow` aus?

2.5. (Pflicht-Hausaufgabe) Spezielle Mehrfachanwendung Deklariert und definiert eine Funktion `specCombine`, welche zwei Funktionen (beide Definitions- und Wertebereich `nat`) kombiniert, so dass die erste Eingabefunktion m-mal und die zweite n-mal angewendet wird. Diese kombinierte Funktion soll zurückgegeben werden.

Beispiel:
$m = 2, n = 3 \rightarrow f \circ f \circ g \circ g \circ g$
$m = 2, n = 0 \rightarrow f \circ f \circ id$

Testet eure Funktion `specCombine` mit folgenden Aufrufen und erläutert die Ergebnisse:

- `> e specCombine(pred,1)(pred,1)(10)`
- `> e specCombine(succ,5)(pred,4)(1)`
- `> e specCombine(succ,5)(pred,4)(max)`

Hinweis: Benutzt eure Lösung aus Aufgabe 2.2!
Beachtet, dass die Funktionsdeklaration zu den aufgeführten Beispielen passen muss!

2.6. (Freiwillige Hausaufgabe) Potenzieren und Tetration mittels Mehrfachanwendung Vielleicht ist euch schon einmal aufgefallen, dass Addition, Multiplikation und Potenzieren aufeinander aufbauen:

- $x \cdot y$ ist x mit sich selbst y-mal addiert.
- x^y ist x mit sich selbst y-mal multipliziert.

Wenn man die Reihe fortsetzt, gelangt man zur sogenannten Tetration (`https://en.wikipedia.org/wiki/Tetration`), geschrieben ${}^y x$.

- ${}^y x$ ist x mit sich selbst y-mal exponenziert: ${}^y x = \underbrace{x^{x^{\cdot^{\cdot^{x}}}}}_{y\text{-mal}}$

Implementiert eine Funktion `tetration`, die ${}^y x$ berechnet mithilfe der Mehrfachanwendung `^` aus Aufgabe 2.2 und des gecurryten `pow`.

Wenn ihr Lust habt, könnt ihr auch `pow` noch einmal neu-definieren mithilfe von Multiplikation und Mehrfachanwendung.