Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

# Programmierung 1 (Wintersemester 2012/13)

## Lösungsblatt 2

(Kapitel 2, Kaskadierung)

**Hinweis:** Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.

*Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant, noch irrelevant.*

## 1 Freie Bezeichner

**Aufgabe 2.1**
Schreiben Sie eine Prozedur, die einen freien Bezeichner enthält, aber auch in der leeren Umgebung ausgeführt werden könnte (wenn der Interpreter freie Bezeichner zuließe).

*Lösung 2.1:*
fun f x = if true then x else a

**Aufgabe 2.2**
Identifizieren Sie alle freien Bezeichner in folgendem Programm:

```
fun f (x:int) = a  + x
val a = 7
fun g (x:int) (y:int) :int = f(x) + f(y) + b
val b = c
val c = 7
```

*Lösung 2.2:*

- In der Phrase `fun f (x:int) = a + x` der Bezeichner `a`
- In der Phrase `fun g (x:int) (y:int) :int = f(x) + f(y) + b` der Bezeichner `b`
- In der Phrase `val b = c` der Bezeichner `c`

**Aufgabe 2.3** (*Zinkel, Zinkel, Zinkel...*)

Welche Umgebung liefert die Ausführung des folgenden Programmes in der leeren Umgebung?

```
val zinkel = ~37
fun g (x:int) (y:int) = x + y + zinkel
val zinkel = 0
fun zinkel' (a:int) = g zinkel
val x = zinkel' 7 3
```

*Lösung 2.3:*
[g := (fun g x y = x + y + zinkel, int → int → int, [zinkel := ∼ 37]),
zinkel := 0,
zinkel' := [fn y ⇒ g x y, int → int → int, [g := (fun g x y = x + y + zinkel, int → int → int, [zinkel := ∼ 37]), x := zinkel])]
x := ∼ 34]

## 2 Semantische Äquivalenz

**Aufgabe 2.4**
Warum nennen wir zwei Prozeduren semantisch äquivalent, wenn sie ich in Bezug auf statische und dynamische Semantik nach außen hin gleich verhalten? Warum nennen wir sie nicht äquivalent, wenn sie z.B. bei der Ausführung ihrer Deklaration die gleiche Umgebung liefern oder sie bis auf Umbenennung der Bezeichner gleich sind?

**Aufgabe 2.5** (*Semantische Zulässigkeit*)

(a) Nennen Sie zwei Bedingungen, die erfüllt sein müssen, damit ein Programm semantisch zulässig ist.

(b) Finden Sie je ein Beispielprogramm das nur genau eine der Bedingungen nicht erfüllt.

(c) Schreiben Sie je zwei Programme, die nicht wohlgetypt sind. Tauschen Sie diese mit Ihrem Nachbarn aus und finden Sie dann in den neuen Programmen die nicht wohlgetypten Stellen.

(d) Diskutieren Sie: Ist es sinnvoll, bei semantisch unzulässigen Programmen von semantischer Äquivalenz zu reden?

*Lösung 2.5:*

(a)
1. Das Programm muss **geschlossen** sein, also alle darin vorkommenden Bezeichnerauftreten mssen innerhalb des Programmes gebunden sein.
2. Das Programm muss **wohlgetypt** sein, also jeder Ausdruck muss den entprechenden Typregeln genügen.

(b)
- 1. erfüllt und 2. nicht erfüllt:

  ```
  fun f (x:int , y:int) = if x > y then x else false
  ```

  Dieses Programm ist geschlossen, weil sowohl x, als auch y im Prozedurrumpf auftreten und somit beim Aufruf der Prozedur an Werte gebunden werden. Es ist jedoch nicht wohlgetypt, da der Prozedurrumpf der Typregel für Konditionale

  $$\frac{e_1 : bool \quad e_2 : t \quad e_3 : t}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

  nicht genügt. Diese setzt nämich voraus, dass die Ausdrücke $e_1$ und $e_2$ den selben Typ $t$ haben. Da in obigem Programm `x` den Typ `int` hat und `true` den Typ `bool`, kann dem Konditional kein Typ zugeordnet werden.
- 1. nicht erfüllt und 2. erfüllt:

  Ein solches Programm existiert nicht, da man ungebundenen Bezeichnern keinen Typ zuordnen kann. Somit kann nicht überprüft werden, ob die Phrase, in der der ungebundene Bezeichner auftritt, der entprechenden Typregel entspricht. Eine Aussage über die Wohlgetyptheit kann also nicht getroffen werden.

**Aufgabe 2.6**
Geben Sie je zwei syntaktisch verschiedene, aber semantisch äquivalente Prozedurdeklarationen an, die die folgenden Funktionen berechnen:

**Beispiel:** $f\colon \mathbb{Z} \to \mathbb{Z}, f(x) = x$ wird berechnet durch die folgenden beiden Prozeduren

```
fun f (x:int) = x fun g (x:int) = 2 * x - x
```

**Hinweis:** Benennen Sie die Prozeduren nicht einfach um. Achten Sie darauf, tatsächlich verschiedene Prozedurrümpfe zu haben.

(a) $f\colon \mathbb{Z} \to \mathbb{Z}, f(x) = (x-1)^3$

(b) $f\colon \mathbb{R} \to \mathbb{R}, f(x) = x + 5.0$

(c) $f\colon \mathbb{Z} \to \mathbb{R}, f(x) = 4.0$

*Lösung 2.6:*

(a) Zum Beispiel:

```
fun f (x:int) = (x - 1) * (x - 1) * (x - 1)     und
fun f (x:int) = x * x * x - 3 * x * x + 3 * x - 1
```

(b) Zum Beispiel:

```
fun f (x:real) = x + 5.0     und
fun f (x:real) = (x * x) / x + 2.0 * 2.5
```

(c) Zum Beispiel:

```
fun f (x:int) = 4.0     und
fun f (x:int) = x - x + 4.0
```

**Aufgabe 2.7** (*Offizielle Übungsaufgabe aus dem Wintersemester 2011/12*)
Entscheiden Sie, ob folgende Programme jeweils semantisch quivalent sind:

a)
```
fun a (x:int) = x + a
```
```
fun a(x:int) = a + x
```

b)
```
fun fac (x:int) =
    if x = 0 then 1
    else fac(x-1)
fun b (p:int*int) = (#2p,#1p)
val it = b(fac(~1),fac(1))
```
```
fun fac (x:int) =
    if x = 0 then 1
    else x * fac(x-1)
fun b(p:int*int) = p
val it = b(fac(1),fac(~1))
```

c)
```
val it = 111111111111 mod 1
```
```
val it = 111111 mod 1
```

d)
```
fun di (x:int) = di (x-1)
```
```
fun di (x:int) = 1 + di (x-1)
```

*Lösung 2.7:*

a) Beide Programme sind semantisch unzulssig, da eine Funktion nicht auf den Operator + angewendet werden darf.

b) Die Programme sind semantisch quivalent, obwohl sie *fac* und *b* an unterschiedliche Werte binden, da beide Programme divergieren.

c) Die Programme sind quivalent, da *it* in beiden Fllen an den Wert 0 gebunden wird.

d) Beide Programme divergieren, aber die Prozeduren `di` haben nicht den gleichen Ergebnistyp (unterscheiden sich also bzgl. der statischen Semantik), weshalb die Programme nicht semantisch quivalent sind.

## 3 Ausdrücke vs. Deklarationen

**Aufgabe 2.8**
Prozedurdeklarationen sind keine Ausdrücke. Deshalb ist z.B. für Aufgabe 2.7 des Übungsblattes

```
fun f (x:int) = x * x
```

keine gültige Lösung. Stattdessen ist

```
let fun f (x:int) = x * x in f end
```

eine gültige Lösung.

Ist dieses Vorgehen für alle Prozeduren möglich? Probieren Sie anhang einiger Beispiele aus, ob sie zu jeder Prozedur so einen Ausdruck erhalten, der semantisch zulässig ist und die gleiche Funktion berechnet.

**Aufgabe 2.9**
Aufgaben wie 3.7 des Übungsblattes sind typische Aufgabenstellungen. Warum wertet der Ausdruck zu einer Prozedur aus? Was müsste in der Prozeduranwendung noch übergeben werden, damit der Ausdruck zu einem expliziten Wert (also einer Zahl, einer booleschen Konstante, Tupe, Liste etc.) auswertet?
Überlegen Sie sich anhand dieser Aufgabe, wie man herausfindet, zu welchem expliziten Wert oder welcher Prozedur ein Ausdruck auswertet.

*Lösung 2.9:*
Innerhalb dieses let-Ausdrucks wird folgende Umgebung erstellt, in der der Ausdruck `g (f 5)` ausgewertet wird:

$\Big[$ $a := 7$,
$f :=$ (`fun f x = a + x` , int `->` int , $[a := 7]$),
$g :=$ (`fun g x y = g (f x) y` , int `->` int `->` $\alpha$ ,
$\big[f :=$ (`fun f x = a + x` , int `->` int , $[a := 7]$)$\big]$)$\Big]$

Bei der Auswertung des Prozeduraufrufs `g (f 5)` wird zuerst das Argument `(f 5)` ausgewertet. Betrachtet man die Bindung von `f` in obiger Umgebung, so sehen wir, dass `f` für ein Argument `x` das Ergebnis der Addition `a + x` zurückliefert. Wir können die Bindung von `a` auf der dritten Komponente der Tripeldarstellung von `f` als 7 herauslesen, also wissen wir, dass die Prozeduranwendung `f x` für `x`:= 5 das Ergebnis 12 zurckliefert.

Mit diesem Wert wird nun die Prozedur `g` angewendet, also der Ausdruck `g 12`. Da `g` kaskadiert deklariert ist, können wir den Code auch folgendermaen darstellen:

```
fun g x = fn y => g (f x) y
```

Wir sehen also, dass `g x` zu einer Prozedur auswertet. In diesem Fall nämlich zu einer Prozedur, die ein Argument `y` nimmt und folgenden Ausdruck mit der Bindung `x`:= 12, sowie obigen Bindungen von `g` und `f` auswertet:

```
g (f x) y
```

Damit die Prozeduranwendung `g (f 5)` zu einem expliziten Wert auswertet, müsste noch ein weiterer Ausdruck, der zu `int` auswertet angehängt werden, z.B. `g (f 5) 9` oder `g (f 5) (2 * a)`.

## 4 Kaskadierung

**Aufgabe 2.10**
Schreiben Sie die folgenden Prozeduren kaskadiert auf. Erläutern Sie Vorteile, die diese Darstellung bringt.

**Beispiel:**

```
fun f (g:int -> bool,x:int) :bool= g(x)
```

Für jeden neuen Wert von $x$, selbst wenn die Prozedur $g$ gleichgelassen wird, muss $f(g, x)$ neu aufgerufen werden. Es ist viel praktischer, einmal $g$ zu übergeben und dann für jeden neuen Wert von $x$ nur noch mit diesem Aufrufen zu müssen. Dies ist möglich durch:

```
fun f (g:int->int) (x:int) = g(x)
fun g ...
val a = f g
val b = a 5
val c = a 7
val d = a 10345678
```

(a) `fun mul (x:int,y:int) = x * y`

(b) `fun konditional (f: int * int -> bool, x:int, y:int) = if f(x,y) then x else y`

*Lösung 2.10:*

(a) `fun mul (x:int) (y:int) = x * y`

Mit dieser Deklaration von `mul` könnte man Prozeduren erhalten, die einen Wert immer mit einer bestimmten Zahl multipliziert. Zum Beispiel

```
val mulSeven = mul 7
val mulTwenty = mul20
```

Hier würde zum Beispiel `mulSeven 13` das Ergebnis der Multiplikation von 7 und 13 zurückliefern.

(b) `fun konditional (f: int * int -> bool) (x:int) (y:int) = if f(x,y) then x else y`

Wenn man nun folgendes Programm ausfhrt

```
val max = konditional (fn (x:int,y:int) => x > y)
val maxOfXandSeven = max 7
val min = konditional (fn (x:int,y:int) => x < y)
```

erhält man die Prozedur `max`, die zu zwei Argumenten immer das Maximum zurückliefert.
`maxOfXandSeven` liefert für ein Argument `x` immer das Maximum von `x` und `7` zurück. `min` liefert entprechend das Minimum zweier Zahlen.

**Aufgabe 2.11**
Betrachten Sie die folgenden Deklarationen:

```
val even = fn x => x mod 2 = 0
fun odd n = not (even n)
fun greater m n = n > max
```

Dieter Schlau möchte eine Prozedur schreiben, die alle natürlichen Zahlen kleiner gleich $n$ aufsummiert, für die ein gegebenes Prädikat $p$ erfüllt ist. Weil er die Prozedur nicht für jedes mögliche Prädikat neu schreiben will, sucht er einen guten Ausweg. Wie kann er vorgehen?

Verwenden Sie die oben angegebenen Prädikate und ihre Summierungsprozedur, um eine Prozedur $sumEven : int \to int \to int$ zu schreiben, die alle natürlichen, geraden Zahlen in einem Bereich von $a$ bis $b$ summiert!

*Lösung 2.11:*
Die Lösung von Dieters Problem ist die Verwendung einer Prozedur, die eine Prozedur $p$ als Argument nimmt (Höherstufigkeit):

```
fun sum p n =
let
     fun validate n = if p n then n else 0
     fun sum' (s, n) = if n = 0 then s else sum'(s + validate n, n-1)
in
     sum'(0, n)
end
```

Die Prozedur *sum* mit dem Typ $(int \to int) \to int \to int$ kann nun mit beliebigen Prädikaten aufgerufen werden:

```
sum even 10 (* ergibt 30 *)
sum odd 10 (* ergibt 25 *)
sum (greater 7) 10 (* ergibt 27 *)
```

Um alle natürlichen, geraden Zahlen im Bereich von $a$ bis $b$ zu summieren, kann man folgendermaßen vorgehen:

```
fun sumEven a b = sum (fn x => even x andalso greater (a-1) x) b
```

Außerdem interessant ist die folgende Prozedur:

```
fun sumAll a b = sum (greater (a-1)) b
```

Man beachte, dass die Implementierungen von *sumEven* und *sumAll* eine sehr schlechte Performanz haben. Beispiel:

```
sumAll 1000000000 1000000000
```

## 5 Knobelaufgaben

**Aufgabe 2.12**
Schreiben Sie zwei Prozeduren

$$cas : (int * int \to int) \to int \to int \to int$$
$$car : (int \to int \to int) \to int * int \to int$$

so dass *cas* zur kartesischen Darstellung einer zweistelligen Operation die kaskadierte Darstellung und *car* zur kaskadierten Darstellung die kartesische Darstellung liefert. Erproben Sie *cas* und *car* mit einem Interpreter!

*Lösung 2.12:*

```
fun cas (f:int * int -> int) (x:int) (y:int) = f (x, y) fun car (f:int -> int -> int) (x:int, y:int)
= f x y
```

**Aufgabe 2.13**
Betrachten Sie das folgende Programm:

```
val u = 42
val n = 2
fun f (x:int) = n*x
val n = 3
val g = fn (x:int) => (f x) * n
val n = 4
val h = f
```

(a) Wozu wertet $f\ 5$ aus?

(b) Wozu wertet $g\ 5$ aus?

(c) **Hinweis:** Bearbeiten Sie diesen Aufgabenteil zuerst alleine, und vergleichen Sie Ihre Ergebnisse dann mit Ihren Kommilitonen!

Woran bindet dieses Programm $u$, $n$, $f$, $g$ und $h$?

*Lösung 2.13:*

(a) 10

(b) 30

(c)
- $u := 42$
- $n := 4$
- $f := (fun\ f\ x = n * x, int \to int, [n := 2])$
- $g := (fn\ x \Rightarrow (f\ x) * n, int \to int, [n := 3, f := (fun\ f\ x = n * x, int \to int, [n := 2])])$
- $h := (fun\ \underbrace{f}_{?????}\ x = n * x, int \to int, [n := 2])$

**Aufgabe 2.14**
Wie sieht die Tripeldarstellung der Prozedur

```
fun f f = f 5
```

aus? Mit was wird die Prozedur aufgerufen?

*Lösung 2.14:*
Die Tripeldarstellung sieht folgendermaßen aus:

$$(\texttt{fun f f = f 5}\ ,\ (\text{int}\ \texttt{->}\ \alpha)\ \texttt{->}\ \alpha\ ,\ [])$$

Man muss die Prozedur mit einer anderen Prozedur aufrufen und sie liefert den Wert der Anwendung der übergebenen Prozedur auf den Wert 5 zurück.
Zum Beispiel liefert folgender Aufruf

```
f (fn x => x * x)
```

den Wert 25 zurück. Die oben deklarierte Prozedur `f` ist entgegen des ersten Anscheins **nicht** rekursiv, sondern lediglich höherstufig.