Programmierung—
Eine Einführung in die Informatik mit Standard ML
Musterlösungen für ausgewählte Aufgaben

Gert Smolka

Vorbemerkungen

a) Ich danke den Assistenten der Vorlesungen in Saarbrücken für ihre Beiträge zu den Musterlösungen.
b) Musterlösungen für die Kapitel 8–16 sind in Vorbereitung.

1 Schnellkurs

Aufgabe 1.1 Betrachten Sie das folgende Programm:

```
val x = 7+4
val y = x*(x-1)
val z = ~x*(y-2)
```

Welche Bezeichner, Konstanten, Operatoren und Schlüsselwörter kommen in dem Programm vor? An welche Werte bindet das Programm die vorkommenden Bezeichner?

Lösung 1.1
Bezeichner: x, y, z
Konstanten: 7, 4, 1, 2
Operatoren: $+$, $*$, $-$, $\sim$
Schlüsselwörter: *val*, $=$, $($, $)$
Bindungen : $x = 11,\ y = 110,\ z = -1188$

Aufgabe 1.2 Deklarieren Sie eine Prozedur $p : int \to int$, die für x das Ergebnis $2x^2 - x$ liefert. Identifizieren Sie das Argumentmuster, die Argumentvariable und den Rumpf Ihrer Prozedurdeklaration.

Lösung 1.2

```
fun p (x:int) = 2*x*x-x
```

Argumentmuster: $(x : int)$
Argumentvariable: x
Rumpf: $2 * x * x - x$

Aufgabe 1.3 (Signum) Schreiben Sie eine Prozedur $signum : int \to int$, die für negative Argumente -1, für positive Argumente 1, und für 0 das Ergebnis 0 liefert.

Lösung 1.3 (Signum)

```
fun signum (x:int) =
if x<0 then ~1
else if x>0 then 1 else 0
```

Aufgabe 1.4 Schreiben Sie eine Prozedur $hoch17 : int \to int$, die zu einer Zahl x die Potenz x^{17} berechnet. Dabei sollen möglichst wenig Multiplikationen verwendet werden. Schreiben Sie die Prozedur auf zwei Arten: Mit einer Hilfsprozedur und mit lokalen Deklarationen.

Lösung 1.4
Mit Hilfsprozedur:

```
fun q (x:int)      = x * x
fun hoch17 (x:int) = x * q (q (q (q x)))
```

Mit lokalen Deklarationen:

```
fun hoch17 (x:int) =
let val a = x*x
    val b = a*a
    val c = b*b
    val d = c*c
in x*d
end
```

Aufgabe 1.5 a) Geben Sie ein Tupel mit 3 Positionen und nur einer Komponente an.
b) Geben Sie einen Tupelausdruck an, der den Typ $int * (bool * (int * unit))$ hat.
c) Geben Sie ein Paar an, dessen erste Komponente ein Paar und dessen zweite Komponente ein Tripel ist.

Lösung 1.5

a) $(1, 1, 1)$
b) $(1, (true, (1, ())))$
c) $((1, false), ((), (), true))$

Aufgabe 1.7 Schreiben Sie eine Prozedur $max : int * int * int \to int$, die zu drei Zahlen die größte liefert, auf zwei Arten:

a) Benutzen Sie keine Hilfsprozedur und drei Konditionale.
b) Benutzen Sie eine Hilfsprozedur und insgesamt nur ein Konditional.

Lösung 1.7

a)
```
fun max (x:int, y:int, z:int) =
  if x>y then if x>z then x else z
  else if y>z then y else z
```
b)
```
fun max' (x:int, y:int) = if x>y then x else y
fun max (x:int, y:int, z:int) = max'(x, max'(y,z))
```
c) Aus (b) erhält man eine Lösung mit 2 Konditionalen, indem man max' direkt bestimmt und das Ergebnis der geschachtelte Anwendung von max' mit einer lokalen Deklaration verfügbar macht (da es zweimal verwendet wird).

```
fun max (x:int, y:int, z:int) =
let val k = if y < z then z else y
in if x < k then k else x
end
```

Aufgabe 1.10 Schreiben Sie eine Prozedur *teilbar* : $int * int \to bool$, die für (x, y) testet, ob x durch y ohne Rest teilbar ist.

Lösung 1.10

```
fun teilbar (x:int, y:int) = (x mod y = 0)
```

Aufgabe 1.11 (Zeitangaben) Oft gibt man eine Zeitdauer im *HMS-Format* mit Stunden, Minuten und Sekunden an. Beispielsweise ist 2h5m26s eine hervorragende Zeit für einen Marathonlauf.

a) Schreiben Sie eine Prozedur *sec* : $int * int * int \to int$, die vom HMS-Format in Sekunden umrechnet. Beispielsweise soll $sec(1, 1, 6)$ die Zahl 3666 liefern.
b) Schreiben Sie eine Prozedur *hms* : $int \to int * int * int$, die eine in Sekunden angegebene Zeit in das HMS-Format umrechnet. Beispielsweise soll *hms* 3666 das Tupel $(1, 1, 6)$ liefern. Berechnen Sie die Komponenten des Tupels mithilfe lokaler Deklarationen.

Lösung 1.11 (Zeitangaben)

a)
```
fun sec (h:int, m:int, s:int) = 3600*h + 60*m + s
```
b)
```
fun hms (s:int) =
  let val h  = s div 3600
      val m  = (s mod 3600) div 60
      val s' = s mod 60
  in (h,m,s')
  end
```

Aufgabe 1.12 Sei die folgende rekursive Prozedurdeklaration gegeben:

```
fun f(n:int, a:int) : int = if n=0 then a else f(n-1, a*n)
```

a) Geben Sie die Rekursionsfolge für den Aufruf $f(3, 1)$ an.
b) Geben Sie ein verkürztes Ausführungsprotokoll für den Ausdruck $f(3, 1)$ an.
c) Geben Sie ein detailliertes Ausführungsprotokoll für den Ausdruck $f(3, 1)$ an. Halten Sie sich dabei an das Beispiel in Abbildung 1.1 (Buch S. 14). Wenn es mehrere direkt ausführbare Teilausdrücke gibt, soll immer der am weitesten links stehende zuerst ausgeführt werden. Sie sollten insgesamt 18 Ausführungsschritte bekommen.

Lösung 1.12

a) Rekursionsfolge:

$$f(3,1) \to f(2,3) \to f(1,6) \to f(0,6)$$

b) verkürztes Ausführungsprotokoll:

$$\begin{aligned} f(3,1) &= f(2,3) \\ &= f(1,6) \\ &= f(0,6) \\ &= 6 \end{aligned}$$

c) detailliertes Ausführungsprotokoll:

$$
\begin{aligned}
\underline{f(3,1)} &= \text{if } \underline{3=0} \text{ then } 1 \text{ else } f(3-1, 1\cdot 3)\\
&= \underline{\text{if } \mathit{false} \text{ then } 1 \text{ else } f(3-1, 1\cdot 3)}\\
&= f(\underline{3-1}, 1\cdot 3)\\
&= f(2, \underline{1\cdot 3})\\
&= \underline{f(2,3)}\\
&= \text{if } \underline{2=0} \text{ then } 3 \text{ else } f(2-1, 3\cdot 2)\\
&= \underline{\text{if } \mathit{false} \text{ then } 3 \text{ else } f(2-1, 3\cdot 2)}\\
&= f(\underline{2-1}, 3\cdot 2)\\
&= f(1, \underline{3\cdot 2})\\
&= \underline{f(1,6)}\\
&= \text{if } \underline{1=0} \text{ then } 6 \text{ else } f(1-1, 6\cdot 1)\\
&= \underline{\text{if } \mathit{false} \text{ then } 6 \text{ else } f(1-1, 6\cdot 1)}\\
&= f(\underline{1-1}, 6\cdot 1)\\
&= f(0, \underline{6\cdot 1})\\
&= \underline{f(0,6)}\\
&= \text{if } \underline{0=0} \text{ then } 6 \text{ else } f(0-1, 6\cdot 0)\\
&= \underline{\text{if } \mathit{true} \text{ then } 6 \text{ else } f(0-1, 6\cdot 0)}\\
&= 6
\end{aligned}
$$

Aufgabe 1.14 Der ganzzahlige Quotient *x div y* lässt sich aus *x* durch wiederholtes Subtrahieren von *y* bestimmen. Schreiben Sie eine rekursive Prozedur $mydiv : int * int \to int$, die für $x \geq 0$ und $y \geq 1$ das Ergebnis *x div y* liefert. Geben Sie zunächst Rekursionsgleichungen für *x div y* an.

Lösung 1.14
Rekursionsgleichungen:

$$
\begin{aligned}
\left\lfloor \tfrac{x}{y} \right\rfloor &= 0 && \text{für } x < y\\
\left\lfloor \tfrac{x}{y} \right\rfloor &= 1 + \left\lfloor \tfrac{x-y}{y} \right\rfloor && \text{sonst}
\end{aligned}
$$

Prozedur:

```
fun mydiv (x:int, y:int) : int =
if x<y then 0 else 1+mydiv(x-y,y)
```

Aufgabe 1.17 (Quersumme) Schreiben Sie eine rekursive Prozedur $quer : int \to int$, die die Quersumme einer ganzen Zahl berechnet. Die Quersumme einer Zahl ist die Summe ihrer Dezimalziffern. Beispielsweise hat die Zahl -3754 die Quersumme 19. Geben Sie zunächst die Rekursionsgleichungen für *quer* an. Verwenden Sie Restbestimmung modulo 10, um die letzte Ziffer einer Zahl zu bestimmen.

Lösung 1.17 (Quersumme)
Rekursionsgleichungen:

$$\begin{aligned} \mathit{quer}\ x &= \mathit{quer}\ (-x) && \text{für } x < 0 \\ \mathit{quer}\ 0 &= 0 \\ \mathit{quer}\ x &= (x \bmod 10) + \mathit{quer} \left\lfloor \tfrac{x}{10} \right\rfloor && \text{für } x > 0 \end{aligned}$$

Prozedur:

```
fun quer (x:int) : int =
if x<0 then quer(~x)
else if x=0 then 0
     else x mod 10 + quer(x div 10)
```

Aufgabe 1.23 (Reversion) Unter der Reversion $\mathit{rev}\, n$ einer natürlichen Zahl n wollen wir die natürliche Zahl verstehen, die man durch Spiegeln der Dezimaldarstellung von n erhält. Beispielsweise soll $\mathit{rev}\, 1234 = 4321$, $\mathit{rev}\, 76 = 67$ und $\mathit{rev}\, 1200 = 21$ gelten.

a) Schreiben Sie zunächst eine endrekursive Prozedur $\mathit{rev}' : \mathit{int} * \mathit{int} \to \mathit{int}$, die zu zwei natürlichen Zahlen m und n die Zahl liefert, die sich ergibt, wenn man die reversierte Dezimaldarstellung von n rechts an die Dezimaldarstellung von m anfügt. Beispielsweise soll $\mathit{rev}'(65,73) = 6537$ und $\mathit{rev}'(0,12300) = 321$ gelten. Die Arbeitsweise von rev' ergibt sich aus dem verkürzten Ausführungsprotokoll $\mathit{rev}'(65,73) = \mathit{rev}'(653,7) = \mathit{rev}'(6537,0) = 6537$.
b) Schreiben Sie mithilfe der Prozedur rev' eine Prozedur rev, die natürliche Zahlen reversiert.
c) Machen Sie sich klar, dass die entscheidende Idee bei der Konstruktion des Reversionsalgorithmus die Einführung einer Hilfsfunktion mit einem Akku ist. Überzeugen Sie sich davon, dass Sie rev nicht ohne Weiteres durch Rekursionsgleichungen bestimmen können.

Lösung 1.23 (Reversion)

a)
```
fun rev' (m:int, n:int) : int =
if n=0 then m else rev'(m*10 + n mod 10,n div 10)
```

b)
```
fun rev (n:int) = rev'(0,n)
```

c) Mit zwei Hilfsfunktionen (Potenz und Stelligkeit (Aufgabe 1.16)) kann man Rekursionsgleichungen für rev angeben.

$$\begin{aligned} \mathit{rev}\ n &= n && \text{für } n < 10 \\ \mathit{rev}\ n &= (n \bmod 10) \cdot 10^{(\mathit{stell}\ n)-1} + \mathit{rev} \left\lfloor \frac{n}{10} \right\rfloor && \text{für } x \geq 10 \end{aligned}$$

2 Programmiersprachliches

Aufgabe 2.1 Geben Sie die Baumdarstellungen der folgenden durch Zeichendarstellungen beschriebenen Phrasen an.

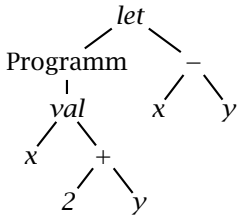
a) `int * int -> bool`
b) `if x<3 then 3 else p 3`

c) `let val x = 2+y in x-y end`

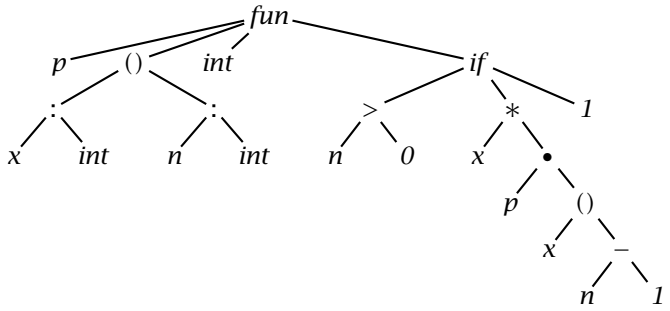
d) `fun p(x:int,n:int):int=if n>0then x*p(x,n-1)else 1`

Lösung 2.1

c)



d)



Aufgabe 2.2 Geben Sie die Baumdarstellungen der folgenden durch Zeichendarstellungen beschriebenen Phrasen an.

a) `x+3*f x-4`

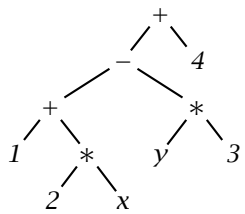
b) `1+2+3+4`

c) `1+2*x-y*3+4`

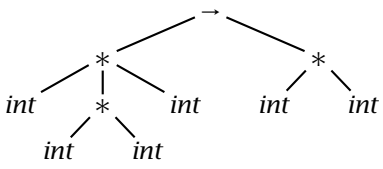
d) `int*(int*int)*int->int*int`

Lösung 2.2

c)



d)



Aufgabe 2.4 Welche Bindungen berechnet das folgende Programm?

```
fun f (x:bool) = if x then 1 else 0
val x = 5*7
fun g (z:int) = f(z<x)<x
val x = g 5
```

Lösung 2.4

$f := (fun\ f\ x = if\ x\ then\ 1\ else\ 0,\ bool \to int,\ [])$
$g := (fun\ g\ z = f(z < x) < x,\ int \to bool,$
$\quad [f := (fun\ f\ x = if\ x\ then\ 1\ else\ 0,\ bool \to int,\ []),$
$\quad\ x := 35])$
$x := true$

Aufgabe 2.6 Betrachten Sie das folgende Programm:

```
val x = 3+2
fun f (y:int) = x+y
fun g (y:int) : int = if y<x then 0 else y+g(y-1)
```

a) Geben Sie die Umgebung an, die die Ausführung des Programms in der Umgebung [] liefert (§ 2.7.4).
b) Geben Sie die Umgebung an, in der der Rumpf der Prozedur f bei der Ausführung des Aufrufs $f\ 7$ ausgeführt wird (§ 2.7.2).
c) Geben Sie die Umgebung an, in der der Rumpf der Prozedur g bei der Ausführung des Aufrufs $g\ 13$ ausgeführt wird (§ 2.7.2).

Lösung 2.6

a) $[x := 5,$
$f := (fun\ f\ y = x + y,\ int \to int,\ [x := 5])$
$g := (fun\ g\ y = if\ y < x\ then\ 0\ else\ y + g(y - 1),\ int \to int,\ [x := 5])]$
b) $[x := 5,\ f := (fun\ f\ y = x + y,\ int \to int,\ [x := 5]),\ y := 7]$
c) $[x := 5,\ g := (fun\ g\ y = if\ y < x\ then\ 0\ else\ y + g(y - 1),\ int \to int,\ [x := 5]),\ y := 13]$

Aufgabe 2.7 Geben Sie einen geschlossenen Ausdruck an, der eine Prozedur $int \to int$ beschreibt, die zu x das Ergebnis x^2 liefert. Geben Sie die Tripeldarstellung der durch Ihren Ausdruck beschriebenen Prozedur an. Hinweis: Verwenden Sie einen Let-Ausdruck.

Lösung 2.7
Ausdruck:

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

Tripeldarstellung:
$(fun\ f\ x = x * x,\ int \to int,\ [])$

3 Höherstufige Prozeduren

Aufgabe 3.2 Deklarieren Sie zwei Prozeduren, die die Operation *div* (ganzzahlige Division) kartesisch und kaskadiert darstellen.

Lösung 3.2

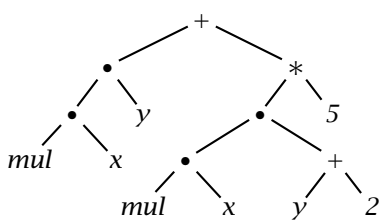
```
fun divkar (x:int, y:int) = x div y
fun divkas (x:int) (y:int) = x div y
```

Aufgabe 3.3 Geben Sie die Baumdarstellung des Ausdrucks

```
mul x y + mul x (y + 2) * 5
```

an. Überprüfen Sie die Richtigkeit Ihrer Darstellung mit einem Interpreter.

Lösung 3.3

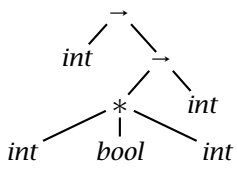


Aufgabe 3.4 Geben Sie die Baumdarstellungen der folgenden Typen an:

a) $int \to real \to int \to bool$
b) $int \to int * bool * int \to int$

Lösung 3.4

b)



Aufgabe 3.5 Geben Sie zu den folgenden Abstraktionen semantisch äquivalente Ausdrücke an, die ohne die Verwendung von Abstraktionen gebildet sind.

a) $fn\ (x : int) \Rightarrow x * x$
b) $fn\ (x : int) \Rightarrow fn\ (y : int) \Rightarrow x + y$

Hilfe: Verwenden Sie Let-Ausdrücke und Prozedurdeklarationen.

Lösung 3.5

b) $let\ fun\ f\ (x : int)\ (y : int) = x + y\ in\ f\ end$

Aufgabe 3.6 Geben Sie die Tripeldarstellung der Prozedur an, zu der der folgende Ausdruck auswertet:

```
(fn (x:int) => fn (b:bool) => if b then x else 7) (2+3)
```

Lösung 3.6 $(fn\ b \Rightarrow if\ b\ then\ x\ else\ 7,\ bool \to int,\ [x := 5])$

Aufgabe 3.7 Geben Sie die Tripeldarstellung der Prozedur an, zu der der folgende Ausdruck auswertet:

```
let val a = 7
    fun f (x:int) = a + x
    fun g (x:int) (y:int) : int = g (f x) y
in
    g (f 5)
end
```

Lösung 3.7
$(fn\ y \Rightarrow g(f\ x)y,\ int \to int,$
$[f := (fun\ f\ x = a + x,\ int \to int,\ [a := 7]),$
$g := (fun\ g\ x\ y = g(f\ x)y,\ int \to int \to int,$
$[f := (fun\ f\ x = a + x,\ int \to int,\ [a := 7])]),$
$x := 12])$

Aufgabe 3.8 Deklarieren Sie eine Prozedur *power* : $int \to int \to int$, die zu x und $n \geq 0$ die Potenz x^n liefert, wie folgt:

a) Mit einer kaskadierten Deklaration.
b) Mit einer Deklaration mit *val* und Abstraktionen.

Lösung 3.8

a)
```
fun power (x:int) (n:int) : int =
  if n=0 then 1 else x * power x (n-1)
```

b)
```
val rec power : int -> int = fn (x:int) => fn (n:int) =>
  if n=0 then 1 else x * power x (n-1)
```

Aufgabe 3.10 Deklarieren Sie mithilfe der höherstufigen Prozedur *sum* eine Prozedur sum' : $(int \to int) \to int \to int \to int$, die für $k \geq 0$ die Gleichung

$$sum'\, f\, m\, k = 0 + f(m+1) \cdots + f(m+k)$$

erfüllt. Die Prozedur sum' soll nicht rekursiv sein.

Lösung 3.10

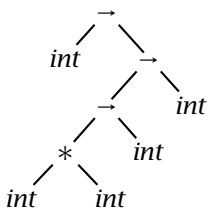
```
fun sum' (f:int->int) (m:int) = sum (fn (i:int) =>  f(m+i))
```

Aufgabe 3.11 Geben Sie die Baumdarstellung des folgenden Typs an:

$$int \to (int * bool \to int) \to int$$

Lösung 3.11

b)



Aufgabe 3.12 Geben Sie geschlossene Abstraktionen an, die die folgenden Typen haben:

a) $(int * int \to bool) \to int \to bool$
b) $(int \to real) \to (bool \to int) \to int * bool \to real * int$

Die Abstraktionen sollen nur mit Prozeduranwendungen, Tupeln und Bezeichnern gebildet werden. Konstanten und Operatoren sollen nicht verwendet werden.

Lösung 3.12

c)
```
fn f:int->bool => fn g:bool->real => fn x:int => g(f x)
```

Aufgabe 3.13 Schreiben Sie zwei Prozeduren

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

sodass *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 Prozeduren, die das Maximum zweier Zahlen liefern:

```
fun maxCas (x:int) (y:int) = if x<y then y else x
fun maxCar (x:int, y:int) = if x<y then y else x
val maxCas' = cas maxCar
val maxCar' = car maxCas
```

Wenn Sie *cas* und *car* richtig geschrieben haben, verhält sich *maxCas′* genauso wie *maxCas* und *maxCar′* genauso wie *maxCar*. Hinweis: Die Aufgabe hat eine sehr einfache Lösung.

Lösung 3.13

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

Aufgabe 3.15 Geben Sie eine Abstraktion *e* an, sodass die Ausführung des Ausdrucks *first x e* für alle *x* divergiert.

Lösung 3.15 `fn _ => false`

Aufgabe 3.17 Deklarieren Sie mit *iter* eine Prozedur *fac*, die zu $n \geq 0$ die n-te Fakultät $n!$ liefert.

Lösung 3.17

```
fun fac  (n:int) : int = #2(iter n (0,1) (fn (k:int, a:int) => (k+1, a*(k+1))))
```

alternativ:

```
fun fac (n:int) : int = #1(iter n (1,n) (fn (a:int, k:int) => (a*k, k-1)))
```

Aufgabe 3.20 Deklarieren Sie polymorphe Prozeduren wie folgt:

$$cas : \quad \forall \alpha\ \beta\ \gamma.\ (\alpha * \beta \to \gamma) \to \alpha \to \beta \to \gamma$$
$$car : \quad \forall \alpha\ \beta\ \gamma.\ (\alpha \to \beta \to \gamma) \to \alpha * \beta \to \gamma$$

Ihre Prozeduren sollen nicht rekursiv sein. Machen Sie sich klar, dass die angegebenen Typschemen das Verhalten der Prozeduren eindeutig festlegen.

Lösung 3.20

```
fun cas f x y = f(x,y)
fun car f (x,y) = f x y
```

Aufgabe 3.21 Dieter Schlau ist ganz begeistert von polymorphen Prozeduren. Er deklariert die Prozedur

```
fun 'a pif (x:bool, y:'a, z:'a) = if x then y else z
val α pif : bool * α * α → α
```

und behauptet, dass man statt eines Konditionals stets die Prozedur *pif* verwenden kann:

$$\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \quad \rightsquigarrow \quad \textit{pif}(e_1, e_2, e_3)$$

Was übersieht Dieter? Denken Sie an rekursive Prozeduren und daran, dass der Ausdruck, der das Argument einer Prozeduranwendung beschreibt, vor dem Aufruf der Prozedur ausgeführt wird.

Lösung 3.21 Das Problem mit *pif* haben wir bereits in Aufgabe 1.25 kennengelernt. Betrachten Sie die folgenden Prozeduren:

```
fun p n = if n>0 then p(n-1) else 0
fun q n = pif(n>0,q(n-1),0)
```

Während p für alle Argumente n den Wert 0 liefert, divergiert q für alle n.

Aufgabe 3.22 Geben Sie ein Typschema für den Bezeichner f an, sodass der Ausdruck $(f\ 1,\ f\ 1.0)$ wohlgetypt ist.

Lösung 3.22 $\forall\alpha.\ \alpha \to \alpha$

Aufgabe 3.23 Warum gibt es keinen Typen t, sodass die Abstraktion $\textit{fn } f : t \Rightarrow (f\ 1,\ f\ 1.0)$ wohlgetypt ist?

Lösung 3.23 Die Argumentvariable f wird in Standard ML monomorph getypt.

Aufgabe 3.24 (Let und Abstraktionen) Dieter Schlau ist begeistert. Scheinbar kann man Let-Ausdrücke mit Val-Deklarationen auf Abstraktion und Applikation zurückführen:

$$\textit{let val } x = e \textit{ in } e' \textit{ end} \quad \rightsquigarrow \quad (\textit{fn } x\colon t \Rightarrow e')\,e$$

Auf der Heimfahrt von der Uni kommen ihm jedoch Zweifel an seiner Entdeckung, da ihm plötzlich einfällt, dass Argumentvariablen nicht polymorph getypt werden können. Können Sie ein Beispiel angeben, das zeigt, dass Dieters Zweifel berechtigt sind? Hilfe: Geben Sie einen semantisch zulässigen Let-Ausdruck an, dessen Übersetzung gemäß des obigen Schemas scheitert, da Sie keinen passenden Typ t für die Argumentvariable x finden können.

Lösung 3.24 (Let und Abstraktionen)

```
let val id = fn x => x in (id 1,id 1.0) end
```

Aufgabe 3.27 Geben Sie Deklarationen an, die monomorph getypte Bezeichner wie folgt deklarieren:

a : $int * unit * bool$

b : $unit * (int * unit) * (real * unit)$

c : $int \to int$

d : $int * bool \to int$

e : $int \to real$

f : $int \to real \to real$

g : $(int \to int) \to bool$

Verzichten Sie dabei auf explizite Typangaben und verwenden Sie keine Operator- und Prozeduranwendungen.

Lösung 3.27

```
fun c x = if true then x else 1
```

Aufgabe 3.29 Deklarieren Sie eine Identitätsprozedur $''a\ ideq : ''a \to ''a$, deren Typschema auf Typen mit Gleichheit eingeschränkt ist. Verzichten Sie dabei auf explizite Typangaben.

Lösung 3.29 `fun ideq x = #1(x,x = x)`

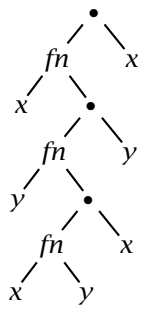
Aufgabe 3.30 Betrachten Sie den Ausdruck

```
(fn x => (fn y => (fn x => y) x) y) x
```

a) Geben Sie die Baumdarstellung des Ausdrucks an.
b) Markieren Sie die definierenden Bezeichnerauftreten durch Überstreichen.
c) Stellen Sie die lexikalischen Bindungen durch Pfeile dar.
d) Geben Sie alle Bezeichner an, die in dem Ausdruck frei auftreten.
e) Bereinigen Sie den Ausdruck durch Indizieren der gebundenen Bezeichnerauftreten.

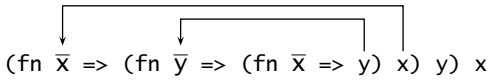
Lösung 3.30

a)



b) `(fn x => (fn y => (fn x => y) x) y) x`

c)



d) x, y

e) `(fn x1 => (fn y1 => (fn x2 => y1) x1) y) x`

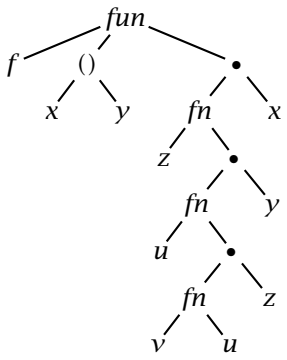
Aufgabe 3.32 Betrachten Sie die bereinigte Deklaration

```
fun f (x, y) = (fn z => (fn u => (fn v => u) z) y) x
```

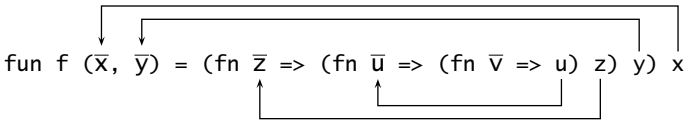
a) Geben Sie die Baumdarstellung der Deklaration an.
b) Stellen Sie die lexikalischen Bindungen der Deklaration durch Pfeile dar.
c) Geben Sie die statischen Bezeichnerbindungen an, die durch die semantische Analyse bestimmt werden.
d) Geben Sie eine möglichst einfache, semantisch äquivalente Deklaration an, die ohne Abstraktionen gebildet ist.

Lösung 3.32

a)



b)



c)

$f : \forall\, \alpha\ \beta.\ \alpha * \beta \to \beta$

$x : \alpha$

$y : \beta$

$z : \alpha$

$u : \beta$

$v : \alpha$

d) `fun f (x, y) = y`

Aufgabe 3.37 Deklarieren Sie mit *iterup* eine Prozedur

a) *power*, die zu x und n die Potenz x^n liefert.
b) *fac*, die zu $n \geq 0$ die n-te Fakultät $n!$ liefert.
c) *sum*, die zu f und n die Summe $0 + f\ 1\ \ldots + f\ n$ liefert.
d) *iter'*, die zu n, s und f dasselbe Ergebnis liefert wie *iter n s f*.

Lösung 3.37

```
fun sum f n = iterup 1 n 0 (fn(i,a)=>a+f i)
```

Aufgabe 3.38 Deklarieren Sie mithilfe der Prozedur *iter* eine Prozedur

a) *iterup′*, die zu *m*, *n*, *s* und *f* dasselbe Ergebnis wie *iterup m n s f* liefert.
b) *iterdn′*, die zu *n*, *m*, *s* und *f* dasselbe Ergebnis wie *iterdn n m s f* liefert.

Lösung 3.38

```
fun iterup' m n s f = #2(iter (n-m+1) (m,s) (fn (k,a) => (k+1,f(k,a))))
```

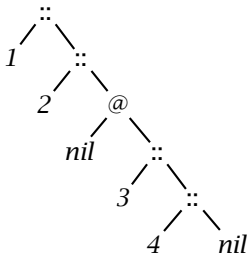
4 Listen und Strings

Aufgabe 4.2 Betrachten Sie den Ausdruck 1::2::*nil*@3::4::*nil*.

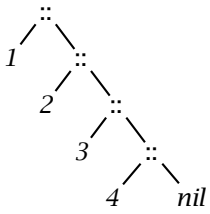
a) Geben Sie die Baumdarstellung des Ausdrucks an.
b) Geben Sie die Baumdarstellung der beschriebenen Liste an.
c) Geben Sie die beschriebene Liste mit „[...]“ an.

Lösung 4.2

a)



b)



c) [1, 2, 3, 4]

Aufgabe 4.9 Schreiben Sie mithilfe von *foldl* eine polymorphe Prozedur *member* : *''a → ''a list → bool*, die testet, ob ein Wert als Element in einer Liste vorkommt.

Lösung 4.9 `fun member x = foldl (fn (y,b) => x=y orelse b) false`

Aufgabe 4.12 Deklarieren Sie die Faltungsprozedur *foldr* mithilfe der Faltungsprozedur *foldl*. Verwenden Sie dabei keine weitere rekursive Hilfsprozedur.

Lösung 4.12

```
fun foldr f s xs = foldl f s (foldl op:: nil xs)
```

oder kürzer:

```
fun foldr f s = foldl f s o foldl op:: nil
```

Aufgabe 4.13 Deklarieren Sie die Faltungsprozedur *foldl* mithilfe der Faltungsprozedur *foldr*. Gehen Sie wie folgt vor:

a) Deklarieren Sie *append* mithilfe von *foldr*.
b) Deklarieren Sie *rev* mithilfe von *foldr* und *append*.

c) Deklarieren Sie *foldl* mithilfe von *foldr* und *rev*.
d) Deklarieren Sie *foldl* nur mithilfe von *foldr*.

Lösung 4.13

b)
```
fun rev xs = foldr (fn (x,ys) => append(ys,[x])) nil xs
```

Aufgabe 4.15 (Last) Deklarieren Sie eine Prozedur $last : \alpha\ list \to \alpha$, die das Element an der letzten Position einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme *Empty* geworfen werden.

Lösung 4.15 (Last)

```
fun last xs = hd (rev xs)
```

Aufgabe 4.16 (Max) Schreiben Sie mit *foldl* eine Prozedur $max : int\ list \to int$, die das größte Element einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme *Empty* geworfen werden. Verwenden Sie die vordefinierte Prozedur $Int.max : int * int \to int$.

Lösung 4.16 (Max)

```
fun max xs = foldl Int.max (hd xs) (tl xs)
```

Aufgabe 4.17 (Take und Drop) Schreiben Sie zwei polymorphe Prozeduren *take* und *drop*, die gemäß $\alpha\ list * int \to \alpha\ list$ getypt sind und die folgende Spezifikation erfüllen:

a) $take(xs, n)$ liefert die ersten n Elemente der Liste xs. Falls $n < 0$ oder $|xs| < n$ gilt, soll die Ausnahme *Subscript* geworfen werden.
b) $drop(xs, n)$ liefert die Liste, die man aus xs erhält, wenn man die ersten n Elemente weglässt. Falls $n < 0$ oder $|xs| < n$ gilt, soll die Ausnahme *Subscript* geworfen werden.

Hinweis: Verwenden Sie *orelse* und die Prozeduren *hd*, *tl* und *null*.

Lösung 4.17 (Take und Drop)

a)
```
fun take(xs,n) = if n=0 then nil
                 else if n<0 orelse null xs
                      then raise Subscript
                      else hd xs :: take(tl xs, n-1)
```

b)
```
fun drop(xs,n) = if n=0 then xs
                 else if n<0 orelse null xs
                      then raise Subscript
                      else drop(tl xs, n-1)
```

Aufgabe 4.19 Entscheiden Sie für jeden der folgenden Werte, ob er das Muster $(x,\ y :: \_ :: z,\ (u, 3))$ trifft. Geben Sie bei einem Treffer die Bindungen für die Variablen des Musters an.

a) $(7,\ [1],\ (3, 3))$
b) $([1, 2],\ [3, 4, 5],\ (11, 3))$

Lösung 4.19

a) Das Muster wird nicht getroffen.
b) $x := [1,2]$, $y := 3$, $z := [5]$, $u := 11$

Aufgabe 4.22 Schreiben Sie eine Prozedur *reverse* : *string* → *string*, die Strings reversiert (z.B. *reverse* "*hut*" = "*tuh*").

Lösung 4.22

```
fun reverse s = implode (rev (explode s))
```

oder kürzer:

```
val reverse = implode o rev o explode
```

Aufgabe 4.23 Schreiben Sie eine Prozedur *isDigit* : *char* → *bool*, die mithilfe der Prozedur *ord* testet, ob ein Zeichen eine der Ziffern $0, \ldots, 9$ ist. Nützen Sie dabei aus, dass *ord* die Ziffern durch aufeinander folgende Zahlen darstellt.

Lösung 4.23

```
fun isDigit c = ord c >= ord #"0" andalso ord c <= ord #"9"
```

5 Sortieren

Aufgabe 5.4 Schreiben Sie eine Prozedur *issort* : *int list* → *int list*, die eine Liste sortiert und dabei Mehrfachauftreten von Elementen eliminiert. Beispielsweise soll für $[3,1,3,1,0]$ die Liste $[0,1,3]$ geliefert werden.

Lösung 5.4

```
fun sinsert (x,nil)   = [x]
  | sinsert (x,y::yr) = case Int.compare(x,y) of
                          LESS  => x::y::yr
                        | EQUAL => y::yr
                        | _     => y::sinsert(x,yr)

val issort = foldl sinsert nil
```

Aufgabe 5.7 Deklarieren Sie eine Prozedur

$$intPairCompare : (int * int) * (int * int) \to order$$

die die lexikalische Ordnung für Paare des Typs $int * int$ darstellt. Zum Beispiel soll $[(3,4), (3,5), (4,0)]$ gemäß dieser Ordnung sortiert sein.

Lösung 5.7

```
fun intPairCompare ((x,x'),(y,y')) =
    case Int.compare(x,y) of EQUAL => Int.compare(x',y')
                           | v     => v
```

Aufgabe 5.14 (Striktes Sortieren durch Mischen)

a) Schreiben Sie eine polymorphe Prozedur *smerge*, die zwei strikt sortierte Listen zu einer strikt sortierten Liste kombiniert. Beispielsweise soll gelten: *smerge Int.compare* $([1,3],\ [2,3]) = [1,2,3]$.
b) Schreiben Sie eine polymorphe Prozedur *ssort*, die Listen strikt sortiert. Beispielsweise soll *ssort Int.compare* $[5,3,2,5] = [2,3,5]$ gelten.
c) Machen Sie sich klar, dass es sich bei *ssort Int.compare* um eine Prozedur handelt, die zu einer Liste *xs* die eindeutig bestimmte strikt sortierte Liste liefert, die dieselbe Menge wie *xs* darstellt.

Lösung 5.14 (Striktes Sortieren durch Mischen)

a)
```
fun smerge _ (nil,ys) = ys
  | smerge _ (xs,nil) = xs
  | smerge compare (x::xr,y::yr) = case compare(x,y) of
         LESS    => x::smerge compare (xr,y::yr)
       | EQUAL   => x::smerge compare (xr,yr)
       | GREATER => y::smerge compare (x::xr,yr)
```

b)
```
fun ssort compare =
    let fun ssort' nil = nil
          | ssort' [x] = [x]
          | ssort' xs = let val (ys,zs) = split xs
                        in smerge compare (ssort' ys,ssort' zs)
                        end
    in ssort' end
```

6 Konstruktoren und Ausnahmen

Aufgabe 6.1 Deklarieren Sie eine Prozedur *variant* : *shape* → *int*, die die Variantennummer eines geometrischen Objekts liefert. Beispielsweise soll *variant*(*Square* 3.0) = 2 gelten.

Lösung 6.1

```
fun variant (Circle _)   = 1
  | variant (Square _)   = 2
  | variant (Triangle _) = 3
```

Aufgabe 6.7 Deklarieren Sie eine Prozedur *check* : *exp* → *exp* → *bool*, die für zwei Ausdrücke *e* und e' testet, ob *e* ein Teilausdruck von e' ist.

Lösung 6.7

```
fun check e e' = e=e' orelse
    case e' of
        A(e1,e2) => check e e1 orelse check e e2
      | M(e1,e2) => check e e1 orelse check e e2
      | _        => false
```

Aufgabe 6.8 Schreiben Sie eine Prozedur *instantiate* : *env* → *exp* → *exp*, die zu einer Umgebung *V* und einem Ausdruck *e* den Ausdruck liefert, den man aus *e* erhält, indem man die in *e* vorkommenden Variablen gemäß *V* durch Konstanten ersetzt. Beispielsweise soll

für die in § 6.4.2 deklarierte Umgebung *env* und den Ausdruck $A(V\,\text{"}x\text{"},\ V\,\text{"}y\text{"})$ der Ausdruck $A(C\ 5,\ C\ 3)$ geliefert werden. Orientieren Sie sich an der Prozedur *eval*.

Lösung 6.8

```
fun instantiate env (V s)     = C(env s)
  | instantiate env (A(e,e')) = A(instantiate env e,instantiate env e')
  | instantiate env (M(e,e')) = M(instantiate env e,instantiate env e')
  | instantiate env e         = e
```

Aufgabe 6.13 Führen Sie zweistellige Sequenzialisierungen $(e_1; e_2)$ auf Abstraktionen und Applikationen zurück.

Lösung 6.13 $(fn\ \_ \Rightarrow e_2)e_1$

7 Bäume

Aufgabe 7.1 Schreiben Sie eine Prozedur *compound* : *tree* → *bool*, die testet, ob ein Baum zusammengesetzt ist.

Lösung 7.1

```
fun compound (T[]) = false
  | compound _ = true
```

Aufgabe 7.3 Geben Sie Ausdrücke an, die Bäume mit den unten gezeigten grafischen Darstellungen beschreiben. Dabei können Sie die Bezeichner $t1 = T[]$ und $t2 = T[t1, t1, t1]$ verwenden.

Lösung 7.3

- $T[T[t1]]$
- $T[T[t1, t1], T[t1]]$
- $T[t2, t1, t1, t2]$

Aufgabe 7.4 Sei die grafische Darstellung eines Baums gegeben. Nehmen Sie an, dass die Darstellung $n \geq 1$ Kanten enthält und beantworten Sie die folgenden Fragen:

a) Wie viele Knoten enthält die Darstellung mindestens/höchstens?
b) Wie viele Blätter enthält die Darstellung mindestens/höchstens?
c) Wie viele innere Knoten enthält die Darstellung mindestens/höchstens?

Lösung 7.4

a) Ein Baum mit n Kanten hat exakt $n + 1$ Knoten
b) Die Darstellung enthält mindestens 1 und höchstens n Blätter
c) Aus b) folgt, dass es mindestens 1 und höchstens n innere Knoten gibt

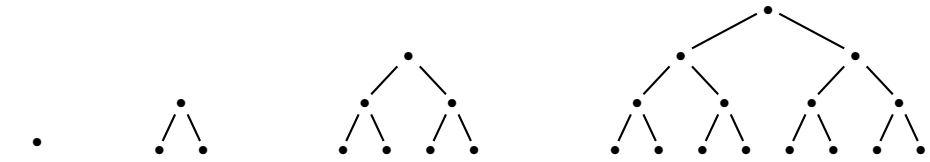
Aufgabe 7.7 Ordnen Sie die folgenden Bäume gemäß der lexikalischen Ordnung an: $t1$, $t2$, $t3$, $T[T[t1]]$, $T[t1, T[T[t1]]]$, $T[T[T[T[t1]]]]$.

Lösung 7.7 $t1 < t2 < T[t1, T[T[t1]]] < T[T[t1]] < t3 < T[T[T[T[t1]]]]$

Aufgabe 7.9 Geben Sie einen Baum mit 5 Knoten an, der genau zwei Teilbäume hat. Wie viele solche Bäume gibt es?

Lösung 7.9 Der einzige Baum mit 5 Knoten, der genau zwei Teilbäume hat, sieht wie folgt aus:

Aufgabe 7.11 Schreiben Sie eine Prozedur *tree* : *int* → *tree*, die für $n \geq 0$ binäre Bäume wie folgt liefert:



Achten Sie darauf, dass die identischen Unterbäume der zweistelligen Teilbäume jeweils nur einmal berechnet werden. Das sorgt dafür, dass Ihre Prozedur auch für $n = 1000$ schnell ein Ergebnis liefert. Verwenden Sie die Prozedur *iter* aus § 3.4.

Lösung 7.11
```
fun tree n = iter n (T[]) (fn t => T[t,t])
```

Aufgabe 7.13 Betrachten Sie die grafische Darstellung des Baums $t3$ in Abbildung

a) Geben Sie die Adresse der Wurzel an.
b) Geben Sie die Adressen der inneren Knoten an (es gibt genau 4).
c) Geben Sie die Adressen der Auftreten des Teilbaums $t2$ an.

Lösung 7.13

a) $[]$
b) $[], [1], [1, 1], [2], [3]$
c) $[1, 1], [3]$

Aufgabe 7.15 Schreiben Sie Prozeduren des Typs *tree* → *int list* → *bool* wie folgt:

a) *node* testet, ob eine Adresse einen Knoten eines Baums bezeichnet.
b) *root* testet, ob eine Adresse die Wurzel eines Baums bezeichnet.
c) *inner* testet, ob eine Adresse einen inneren Knoten eines Baums bezeichnet.
d) *leaf* testet, ob eine Adresse ein Blatt eines Baums bezeichnet.

Lösung 7.15

```
fun node t xs = (ast t xs;true) handle Subscript => false
fun root (T _) (xs:int list) = null xs
fun inner t xs = compound (ast t xs) handle Subscript => false
fun leaf t xs = compound (ast t xs) = false handle Subscript => false
```

Aufgabe 7.23 Schreiben Sie die oben angegebene Prozedur *size* so um, dass sie ohne *map* auskommt. Hilfe: Erledigen Sie die rekursive Anwendung von *size* in der Verknüpfungsprozedur für *foldl*.

Lösung 7.23 `fun size (T ts) = foldl (fn (t,s) => size t + s) 1 ts`

Hinweis : Aufgabe 7.27 hat ein Beispiel im Buch

Aufgabe 7.32 Schreiben Sie eine Prozedur *post'* : *tree* → *int* → *tree*, die zu einem Baum und einer Postnummer den entsprechenden Teilbaum liefert. Realisieren Sie die Agenda von *post* mit der folgenden Typdeklaration:

```
datatype 'a entry = I of 'a | F of 'a
```

Lösung 7.32

```
datatype 'a entry = I of 'a | F of 'a

fun lift (T ts) = map I ts
fun post nil _ = raise Subscript
  | post (F t::_) 0 = t
  | post (F _::tr) k = post tr (k-1)
  | post (I t::tr) k = post (lift t @ F t :: tr) k
fun post' t = post [I t]
```

Aufgabe 7.33 Geben Sie die Prä- und die Postlinearisierung des Baums $T[T[\,], T[T[\,]], T[T[\,], T[\,]]]$ an.

Lösung 7.33
Prälinearisierung: $[3,0,1,0,2,0,0]$
Postlinearisierung: $[0,0,1,0,0,2,3]$

Aufgabe 7.34 Gibt es Listen über $\mathbb{N}$, die gemäß der Prä- oder Postlinearisierung keine Bäume darstellen?

Lösung 7.34 Ja. Beispielsweise stellt die Liste $[1]$ weder gemäß der Prä- noch Postlinearisierung einen Baum dar.

Aufgabe 7.37 Deklarieren Sie eine Prozedur *direct* : *tree* → *tree*, die zu einem Baum einen gerichteten Baum liefert, der die gleiche Menge darstellt. Verwenden Sie die polymorphe Sortierprozedur aus Aufgabe 5.14 auf S. 107 und die Vergleichsprozedur *compareTree* aus § 7.1.3.

Lösung 7.37 `fun direct (T ts) = T(ssort compareTree (map direct ts))`

Aufgabe 7.42 Schreiben Sie Prozeduren *leftmost*, *rightmost* : α *ltr* → α, die das linkeste beziehungsweise rechteste Blatt eines Baums liefern.

Lösung 7.42

```
fun rightmost (L(a,nil)) = a
  | rightmost (L(a,ts)) = rightmost (last ts)
```

Aufgabe 7.49 Schreiben Sie eine Prozedur *find* : $(\alpha \to bool) \to \alpha\ ltr \to \alpha\ option$, die zu einer Prozedur und einem Baum die gemäß der Präordnung erste Marke des Baums liefert, für die die Prozedur *true* liefert. Orientieren Sie sich an der Prozedur *prest* aus § 7.6.1.

Lösung 7.49

```
fun find' p nil = NONE
  | find' p (L(x,ts)::es) = if p x then SOME x else find' p (ts@es)
fun find p t = find' p [t]
```

Aufgabe 7.54 Die **Grenze** eines markierten Baums ist die Liste der Marken seiner Blätter, in der Ordnung ihres Auftretens von links nach rechts und mit Mehrfachauftreten. Die Grenze des Baums *t3* (§ 7.9) ist $[2, 7, 7, 4, 2, 7, 7]$. Schreiben Sie eine Prozedur *frontier* : $\alpha\ ltr \to \alpha\ list$, die die Grenze eines Baums liefert.

Lösung 7.54

```
fun frontier (L(x,nil)) = [x]
  | frontier (L(x,ts)) = List.concat (map frontier ts)
```