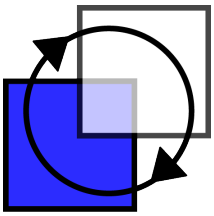
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Aufgaben für die Übungsgruppe B (Lösungsvorschläge)

Hinweis: Diese Aufgaben wurden von den Tutoren für die Übungsgruppe erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

Schnellkurs

Aufgabe TB.1 *(Argumentmuster)*
Deklarieren Sie eine Prozedur `flatten : (int * int) * int → int * int * int`, sodass `flatten ((a, b), c) = (a, b, c)` für beliebige Ganzzahlen `a`, `b` und `c` auf drei verschiedene Arten:

(a) mit einem kartesischen Argumentmuster

(b) mit einer lokalen Deklaration

(c) mit Projektionen

Lösungsvorschlag TB.1

(a)

```
fun flatten((a: int, b: int), c: int) : int*int*int = (a, b, c)
```

```
fun flatten (t: ((int*int)*int)) : int*int*int =
        let
                val ((a, b), c) = t
        in
                (a, b, c)
        end
```

(b)

```
fun flatten (t: (int*int)*int) : int*int*int = (#1(#1 t), #2(#1 t), #2 t)
```

Aufgabe TB.2 *(Max Mustermann)*
Schreiben Sie eine Prozedur `max : int * int * int → int`, die zu drei Zahlen die Größte liefert, auf zwei Arten:

(a) Benutzen Sie drei Konditionale und keine Hilfsprozedur.

(b) Benutzen Sie eine Hilfsprozedur und insgesamt nur ein Konditional.

Lösungsvorschlag TB.2

(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_help(x : int, y : int) = if x > y then x else y
fun max(x : int, y : int, z : int) = max_help(max_help(x, y), z)
```

Aufgabe TB.3 *(Schaltjahr)*
Schreiben Sie eine Prozedur `leapyear : int → bool`, die angibt, ob es sich bei einer Jahreszahl um ein Schaltjahr nach dem Gregorianischen Kalender[1] handelt. Nach diesem sind alle durch 4 teilbaren Jahre Schaltjahre, es sei denn eine Jahreszahl ist durch 100 teilbar, aber nicht durch 400, so liegt kein Schaltjahr vor.

Lösungsvorschlag TB.3

```
fun leapyear (x : int) : bool =
   if (x mod 4 <> 0) then false
                     else if (x mod 100 <> 0) then true
                                              else x mod 400 = 0
```

Aufgabe TB.4 *(https://www.youtube.com/watch?v=tRblwTsX6hQ)*
Dieter Schlau hat einen Hilferuf aus MfI1 erhalten. Niemand kann sich daran erinnern, wie die p-q-Formel aussieht.

(a) Helfen Sie Dieter Schlau dabei, eine Prozedur `pq: real * real → real * real` zu schreiben, die für Funktionen $f(x) = x^2 + px + q$ aus p und q die Nullstellen berechnet. Verwenden Sie dabei genau zwei verschiedene Standardprozeduren je genau einmal.

(b) Dieter Schlau ist zufrieden mit Ihrer Arbeit. Als er Ihre Prozedur verwendet, fällt ihm auf, dass man in SML nicht automatisch Ganzzahlen in Gleitkommazahlen umwandeln kann. Nehmen Sie ihm etwas Arbeit ab, indem Sie mithilfe der in (a) deklarierten Prozedur eine Prozedur `pqInt : int * int → real * real` schreiben.

Hinweis: Ihre Prozeduren müssen nur für Funktionen mit genau zwei Nullstellen das richtige Ergebnis liefern.

Lösungsvorschlag TB.4

(a)

```
fun pq (p:real, q:real):(real*real) =
  let
    val root = Math.sqrt(Math.pow(p/2.0, 2.0) - q)
  in
    (~p/2.0 + root, ~p/2.0 - root)
  end
```

(b)

```
fun pqInt (p:int, q:int):(real*real) =
  pq(Real.fromInt(p), Real.fromInt(q))
```

[1] https://de.wikipedia.org/wiki/Schaltjahr#Gregorianischer_Kalender

Tupel und Typen

Aufgabe TB.5 *(Tolle Tupeltypen)*

(a) Konstruieren Sie ein Tupel an, welches aus genau einem Tripel und genau 5 Paaren zusammengesetzt ist.

(b) Nennen Sie 3 verschiedene zusammengesetzte Typen. Benutzten Sie dabei jeweils 3 mal den Typen `int` und sonst keine anderen Typen.

Lösungsvorschlag TB.5

(a) `((1, 1), ((2, 2), (2, 2)), (4, 4))`

(b)
```
int * int * int
int * (int * int)
(int * int) * int
```

Aufgabe TB.6 *(Tupel en détail)*
Geben Sie für folgende Ausdrücke ihre Baumdarstellung an, falls es sich um Tupel handelt! Welche Komponenten hat das Tupel? Wie lang ist es? Und geben Sie die Typen an!

(a) `(((1, 2, 3)), 1, 2, 3)`

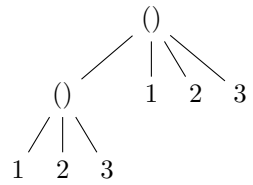
(b) `(((3)))`

(c) `((), (1.0, (2,3)), false, 42)`

(d) `(3 < 4, 1 + 2 + 3, (()))`

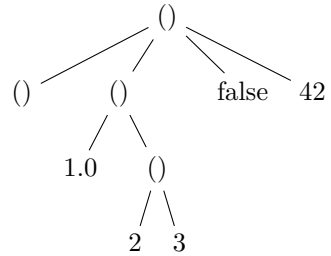
Lösungsvorschlag TB.6

(a) Das Tupel hat die Komponenten `(1, 2, 3)`, `1`, `2` und `3`. Es hat die Länge 4.
((int * int * int) * int * int * int)

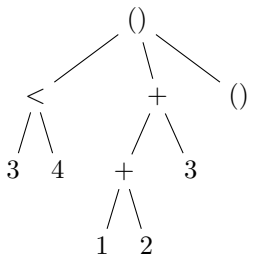


(b) Hierbei handelt es sich nicht um ein Tupel, sondern um die geklammerte Darstellung der Zahl 3.

(c) Das Tupel hat die Komponenten `()`, `((1.0, (2, 3))`, `false` und `42`. Es hat die Länge 4.
(unit * (real * (int * int)) * bool * int)



(d) Das Tupel hat die Komponenten `3 < 4`, `1 + 2 + 3` und `()`. Es hat die Länge 3.
(bool * int * unit)



Aufgabe TB.7 *(Typen-Tupel)*
Geben Sie jeweils ein Tupel mit den folgenden Typen an.

(a) `unit * int * int * real * bool`
(b) `real * (int * int)`
(c) `((real * int) * unit) * (unit * int)`
(d) `real * bool * bool * bool * bool * int * bool`
(e) `((unit * unit) * (unit * unit) * ((unit * unit) * (unit * unit))) * unit`

Lösungsvorschlag TB.7

(a) `((), 1, 2, 1.2, false)`
(b) `(4.4, (0, 1))`
(c) `(((1.0, 1), ()), ((), 1))`
(d) `(1.0, false, true, false, true, 1, true)`
(e) `((((), ()), ((), ()), (((), ()), ((), ()))), ())`

Aufgabe TB.8 *(Projezier mich nicht!)*

(a) Gegeben sei das Tupel `a`:

```
1 val a = (3, (if false then 4 else ~1, (27, false)), ((true, ()), 4))
```

Lesen Sie die folgenden Werte mithilfe von Projektionen aus `a` aus:

(i) `3` (ii) `()` (iii) `(27, false)` (iv) `4`

(b) Deklarieren Sie ein Tupel `b`, sodass die folgenden Ausdrücke zu `true` auswerten:

- `#1b = true`
- `#3b = ()`
- `#2(#4b) = 4`
- `#1(#2(#2b)) = ~1`

Lösungsvorschlag TB.8

(a) (i) `#1a` (ii) `#2(#1(#3a))` (iii) `#2(#2a)` (iv) `#2(#3a)`

(b) `val b = (true, (0, (~1, 0)), (), (0, 4))`

Aufgabe TB.9 *(Tupel-Typen)*
Geben Sie die Typen der folgenden Ausdrücke an.

(a) `(4, 2, 42, false, ())`

(b) `((), (4, (2, 42)), false)`

(c) `(if true then 6 else ~5, 3, (), true)`

(d) `#3(true, false, (4, 5, ()))`

(e)
```
let
    val x = 1 + 2
    val y = true
in
    (x, (y), ((), ()), ((((()))))
end
```

(f)
```
let
    fun f (x:int) : bool = true
    val a = if f 5 then 3 else 5
    fun g (x:bool, y:int) = (f y,a)
in
    ((()), g (true, 7), f 5)
end
```

(g)
```
let
    fun f (a : int, b : int) = a div b
in
    f
end
```

Lösungsvorschlag TB.9

(a) `(int * int * int * bool * unit)`

(b) `(unit * (int * (int * int)) * bool)`

(c) `int * int * unit * bool`

(d) `int * int * unit`

(e) `(int * bool * (unit * unit) * unit)`

(f) `(unit * (bool * int) * bool)`

(g) `int * int → int`

Rekursion

Aufgabe TB.10 *(Gaußsche Summe)*
Schreiben Sie eine rekursive Prozedur `gauss : int → int`, die die Gaußsche Summe (die Summe der Zahlen 0 bis n) einer beliebigen Zahl $n \in \mathbb{N}$ berechnet.

Beispiel: `gauss 5` $=$ `0 + 1 + 2 + 3 + 4 + 5` $=$ `15`

Machen Sie sich an einem vereinfachten Ausführungsprotokoll für den Aufruf `gauss 4` deutlich, wie Ihre Prozedur arbeitet.

Lösungsvorschlag TB.10

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

gauss 4 = (gauss 3) + 4
        = ((gauss 2) + 3) + 4
```

```
        = (((gauss 1) + 2) + 3) + 4
        = ((((gauss 0) + 1) + 2) + 3) + 4
        = (((0 + 1) + 2) + 3) + 4
        = 10
```

Aufgabe TB.11 *(Fack ju Gauss)*
Gegeben sei die folgende rekursive Prozedurdeklaration:

```
fun foo (n : int) = if n = 0 then 1 else n * foo(n − 1)
```

(a) Geben Sie die Rekursionsfolge für den Aufruf `f(2)` an.

(b) Geben Sie ein vollständiges Ausführungsprotokoll für den Ausdruck `f(2)` an.

(c) Geben Sie ein verkürztes Ausführungsprotokoll für den Aufruf `f(2)` an.

(d) Welche mathematische Funktion berechnet die Prozedur?

Lösungsvorschlag TB.11

(a) `foo(2)` → `foo(1)` → `foo(0)`

(b)
```
foo(2) = if 2 = 0 then 1 else 2 * foo(2 − 1)
       = if false then 1 else 2 * foo(2 − 1)
       = 2 * foo(2 − 1)
       = 2 * foo(1)
       = 2 * (if 1 = 0 then 1 else 1 * foo(1 − 1))
       = 2 * (if false then 1 else 1 * foo(1 − 1))
       = 2 * (1 * foo(1 − 1))
       = 2 * (1 * foo(0))
       = 2 * (1 * (if 0 = 0 then 1 else 0 * foo(0 − 1)))
       = 2 * (1 * (if true then 1 else 0 * foo(0 − 1)))
       = 2 * (1 * 1)
       = 2
```

(c)
```
foo(2) = 2 * foo(1)
       = 2 * (1 * foo(0))
       = 2 * (1 * 1)
       = 2
```

(d) Die Prozedur berechnet die Fakultät der Zahl `n`.

Aufgabe TB.12 *(Addieren in langsam)*
Gegeben sei folgende Prozedur:

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

Schreiben Sie ein *vollständiges* Ausführungsprotokoll des Prozeduraufrufs `add(4, ~2)`.

Lösungsvorschlag TB.12

```
add(4, ~2) = if ~2 = 0 then 4 else if ~2 < 0 then add(4 − 1, ~2 + 1) else add(4 + 1, ~2 − 1)

= if false then 4 else if ~2 < 0 then add(4 − 1, ~2 + 1) else add(4 + 1, ~2 − 1)

= if ~2 < 0 then add(4 − 1, ~2 + 1) else add(4 + 1, ~2 − 1)

= if true then add(4 − 1, ~2 + 1) else add(4 + 1, ~2 − 1)

= add(4 − 1, ~2 + 1)
```

```
10
11 = add(3, ~2 + 1)
12
13 = add(3, ~1)
14
15 = if ~1 = 0 then 3 else if ~1 < 0 then add(3 - 1, ~1 + 1) else add(3 + 1, ~1 - 1)
16
17 = if false then 3 else if ~1 < 0 then add(3 - 1, ~1 + 1) else add(3 + 1, ~1 - 1)
18
19 = if ~1 < 0 then add(3 - 1, ~1 + 1) else add(3 + 1, ~1 - 1)
20
21 = if true then add(3 - 1, ~1 + 1) else add(3 + 1, ~1 - 1)
22
23 = add(3 - 1, ~1 + 1)
24
25 = add(2, ~1 + 1)
26
27 = add(2, 0)
28
29 = if 0 = 0 then 2 else if 0 < 0 then add(2 - 1, 0 + 1) else add(2 + 1, 0 - 1)
30
31 = if true then 2 else if 0 < 0 then add(2 - 1, 0 + 1) else add(2 + 1, 0 - 1)
32
33 = 2
```

Aufgabe TB.13 *(Divergenz)*
Machen Sie sich klar, warum die unten stehende Prozedur für x > 0 divergiert. Erstellen Sie sich hierfür ein Ausführungsprotokoll, welches die ersten Rekursionsaufrufe beinhaltet.

```
1 fun f (x:int):int = if x > 0 then 1+f(x div 2 + 2) else x
```

Lösungsvorschlag TB.13
Betrachten Sie das Ausführungsprotokoll

$$f(6) = if(6 > 0)\ then\ 1 + f(6\ div\ 2 + 2)\ else\ 6 = 1 + f(6\ div\ 2 + 2) = 1 + f(3 + 2)$$

$$= 1 + f(5) = 1 + if(5 > 0)\ then\ 1 + f(5\ div\ 2 + 2)\ else\ 5 = 1 + 1 + f(5\ div\ 2 + 2) = 1 + 1 + f(2 + 2) = 1 + 1 + f(4)$$

$$= 1 + 1 + if(4 > 0)\ then\ 1 + f(4\ div\ 2 + 2)\ else\ 4 = 1 + 1 + 1 + f(4\ div\ 2 + 2) = 1 + 1 + 1 + f(2 + 2)$$

$$= 1 + 1 + 1 + f(4) = ...$$

Durch den rekursiven Aufruf f(5) wird dann letztendlich wieder f(5) aufgerufen, die Ausführung wird also niemals terminieren.

Aufgabe TB.14 *(Blazingly Fast Division . . . not ;))*

(a) Schreiben Sie eine Prozedur `mydiv : int * int → int`, die für $x \geq 0$ und $y \geq 1$ das Ergebnis x div y liefert, ohne den Operator `div` zu verwenden.

(b) Können Sie Ihre Prozedur so erweitern, dass sie auch für negative $x \in \mathbb{Z}$ und $y \in \mathbb{Z} \setminus \{0\}$ gemäß der Definition aus dem Buch funktioniert?

Lösungsvorschlag TB.14

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

(b)
```
1 fun fulldiv (x : int, y : int) : int =
2    if x * y >= 0 then mydiv(abs x, abs y)
3                  else fulldiv (x + y, y) - 1
```

Aufgabe TB.15 *(This is Mathe)*

(a) Schreiben Sie eine Prozedur `potenz : int * int → int`, die b^x für beliebige $b, x \in \mathbb{N}$ berechnet.

(b) Schreiben Sie nun eine Prozedur `length : int → int`, die gegeben einer beliebigen Zahl $n \in \mathbb{N}$ die Länge dieser Zahl, also die Anzahl der Ziffern, zurückgibt.

> **Beispiel:**
> - `length (0) = 1`
> - `length (15) = 2`
> - `length (01337) = 4`

(c) Die Konkatenation zweier Zahlen ist die Aneinanderreihung dieser. Implementieren Sie eine Prozedur `concat : int * int → int`, die zwei beliebige Zahlen $\in \mathbb{N}$ konkateniert.

> **Beispiel:**
> - `concat (1337, 0) = 13370`
> - `concat (0, 42) = 42`
> - `concat (012, 034) = 1234`

Lösungsvorschlag TB.15

```
fun potenz ((b, x) : int * int) : int = if x < 1 then 1 else b * potenz(b, x - 1)
fun length (n : int) : int = if n < 10 then 1 else 1 + length(n div 10)
fun concat ((n, m) : int * int) : int = if m = 0 then n else potenz(10, length(m)) * n + m
```

Endrekursion

Aufgabe TB.16 *(Rekursion vs. Endrekursion)*
Die folgenden Prozeduren berechnen für beliebige x und $n \geq 0$ mit $a = 0$ dasselbe Ergebnis:

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

(a) Welches ist die endrekursive Variante?

(b) Geben Sie jeweils ein vollständiges Ausführungsprotokoll für beide Prozeduren mit den Argumenten $a = 0,\ x = 10$ und $n = 1$ an.

(c) Machen Sie sich klar, dass beide Ausführungsprotokolle für beliebige $a,\ x$ und $n \geq 0$ immer gleichlang sind.

(d) Dieter Schlau hat gehört, dass Endrekursion schneller als „normale“ Rekursion sein soll. Die Aussage ist tatsächlich im Allgemeinen richtig. Woran könnte das liegen? 🤔

Lösungsvorschlag TB.16

(a) `p'` ist die endrekursive Variante.

(b)

```
p (10, 1)
= if 1 > 0 then 10 * 1 + p (10, 1 - 1) else 0
= if true then 10 * 1 + p (10, 1 - 1) else 0
= 10 * 1 + p (10, 1 - 1)
= 10 + p (10, 1 - 1)
= 10 + p (10, 0)
= 10 + (if 0 > 0 then 10 * 1 + p (10, 0 - 1) else 0)
= 10 + if false then 10 * 1 + p (10, 0 - 1) else 0
= 10 + 0
= 10
```

```
p' (10, 1, 0)
= if 1 > 0 then p' (10, 1 - 1, 10 * 1 + 0) else 0
= if true then p' (10, 1 - 1, 10 * 1 + 0) else 0
= p' (10, 1 - 1, 10 * 1 + 0)
= p' (10, 0, 10 * 1 + 0)
= p' (10, 0, 10 + 0)
= p' (10, 0, 10)
= if 0 > 0 then p' (10, 0 - 1, 10 * 0 + 10) else 10
= if false then p' (10, 0 - 1, 10 * 0 + 10) else 10
= 10
```

(c) Man kann relativ leicht sehen, dass die Wahl von a und x keinen Einfluss auf die Anzahl der Rekursionsaufrufe und damit die Länge des Ausführungsprotokolls hat. Nun können wir zwei Fälle für n betrachten:

Wenn $n = 0$, dann muss für beide Prozeduren bloß das Konditional ausgewertet werden, was jeweils genau einen Schritt benötigt.

Wenn $n > 0$, dann werten wir ebenfalls zunächst das Konditional aus. Anschließend verhalten sich p und p' zwar etwas unterschiedlich, werten jedoch beide die Multiplikation `x * n` und die Subtraktion `n - 1` aus. Auch die Addition von dem Ergebnis von `x * n` und `p(x, n - 1)` bzw. `a` kommt in beiden Prozeduren vor, bei p erst am Ende, bei p' schon vor dem nächsten rekursiven Aufruf. Mit der Annahme, dass die Prozeduranwendung mit $n - 1$ in gleichlangen Ausführungsprotokollen resultiert, wäre gezeigt, dass die Ausführungsprotokolle tatsächlich gleichlang sind.

Die erwähnte Annahme liefert die Idee der Induktion, die wir uns später anschauen werden. Dann werden wir auch in der Lage sein, aus diesen Überlegungen einen formalen Beweis zu konstruieren.

(d) Die endrekursive Variante ist tatsächlich auch hier schneller, was man natürlich allerdings erst bei sehr großen Argumenten merken kann. Praktisch kann man das bei unserem Programm kaum ausprobieren, da die Zahlen, mit denen gerechnet wird, zu schnell zu groß werden und die Ausführung deshalb abbricht.

Der Geschwindigkeitsvorteil hat grob gesagt damit zu tun, dass sich der Interpreter nicht immer das Ergebnis von `x * n` zwischenspeichern muss sondern es direkt mit `a` verrechnen kann. Dadurch kann der Interpreter im Wesentlichen mit drei „Speichern“ arbeiten anstatt ungefähr `n` vielen. Immer wiederkehrende Speicherzugriffe an dieselbe Stelle kann der Computer üblicherweise schneller ausführen als solche an unterschiedliche Stellen.

Es gibt auch noch Vorteile bezüglich der Rückkehr von der Berechnung rekursiver Aufrufe, dafür müsste man allerdings tiefer in die Art und Weise einsteigen, wie Computer tatsächlich arbeiten. Dazu gehören Aspekte, die in den Vorlesungen *Systemarchitektur* und *Programmierung 2* behandelt werden. Einige Punkte findet man auch schon in Kapitel 16 des Buchs.

Aufgabe TB.17 *(Addition durch Inkrementieren)*
Schreiben Sie eine endrekursive Prozedur `add : (int * int) * int → int`, die durch wiederholtes Inkrementieren zwei positive Zahlen addiert.

Hinweis: Verwenden Sie hierbei keine Addition, außer +1.

Lösungsvorschlag TB.17

```
1 fun add ((x : int, y : int), a : int) : int =
2     if x = 0 then
3         (if y = 0 then a else add((x,y - 1), a + 1))
4     else
5         add((x - 1,y), a + 1)
```

Aufgabe TB.18 *(Größte Ziffer)*
Schreiben Sie eine Prozedur `groessteZiffer : int → int` welche die größte Ziffer einer Zahl ausgibt. Verwenden Sie hierzu eine endrekursive Hilfsprozedur. Beachten Sie, dass Ihre Prozedur auch für negative Zahlen korrekt sein soll.

Beispiel: `groessteZiffer (12392)` $= 9$ `groessteZiffer (~13)` $= 3$.

Sie dürfen folgende Hilfsprozeduren verwenden:

```
1 fun abs (x : int) : int = if x < 0 then ~x else x
2 fun max (a : int, b : int) : int = if a > b then a else b
```

Lösungsvorschlag TB.18

```
fun groessteZiffer (x : int) : int =
   let
      fun helper (x : int, z : int) : int =
         if x < 10 then max (x, z)
                   else helper (x div 10, max(x mod 10, z))
   in
      helper(abs (x), 0)
   end
```

Aufgabe TB.19 *(Endrekursive Prozeduren)*
Welche der folgenden Prozeduren sind endrekursiv?

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

(b) `fun g (a : int, x : int) : int = if x = 0 then a else g(a + x, x - 1)`

(c) `fun fac (x : int) : int = if x = 0 then 1 else x * fac(x - 1)`

(d) `fun id (x : int) : int = x`

(e) `fun h (x : int) : bool = if x <=1 then x = 0 else h(x - 2)`

(f)
```
fun abs (a : int, x : int) : int =
        if x = 0 then a else
                if x > 0 then abs(a + 1, x - 1) else abs(a, x + 1) - 1
```

(g) `fun pow2 (a : int, n : int) : int = if n = 0 then a else pow2(a * 2, n - 1)`

(h) `fun i (a : int) : int = if false then i a else 0`

(i)
```
fun Fun (a : int, n : int) : int =
        if n = 0 then a else Fun(Fun(a + 1, n - 1), n - 1)
```

Lösungsvorschlag TB.19
Endrekursiv: Teilaufgaben a, b, e, g, h

Aufgabe TB.20 *(Fibonacci)*
Schreiben Sie eine Prozedur `fib : int → int`, die mittels einer endrekursiven Hilfsprozedur die n-te Stelle der Fibonacci Folge berechnet.

Die Fibonacci Folge ist definiert als:

$$\begin{aligned} &\mathit{fib}\ 0 = 1 \\ &\mathit{fib}\ 1 = 1 \\ &\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \qquad \text{für} \quad n > 1 \end{aligned}$$

Schreiben Sie außerdem ein Ausführungsprotokoll für den Aufruf `fib(5)`.

Lösungsvorschlag TB.20

```
fun fibi (n : int, a : int, b : int) : int =
   if n < 1 then a else fibi(n - 1, b, b + a)

fun fib (n : int) = fibi(n, 1, 1)
```

Ausführungsprotokoll:

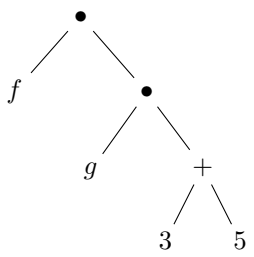
$$\begin{aligned} \mathit{fib}(5) &= \mathit{fibi}(5,1,1) \\ &= \mathit{fibi}(4,1,2) \\ &= \mathit{fibi}(3,2,3) \\ &= \mathit{fibi}(2,3,5) \\ &= \mathit{fibi}(1,5,8) \\ &= \mathit{fibi}(0,8,13) \\ &= 8 \end{aligned}$$

Programmiersprachliches

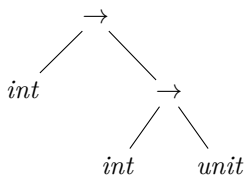
Aufgabe TB.21 *(Minimale Klammerung)*
Geben Sie die folgenden in der Baumdarstellung gegebenen Ausdrücke in minimal geklammerter Zeichendarstellung an:

Hinweis: Sie finden Klammersparregeln in Anhang A, sowie auf Seite 36 des Buches.

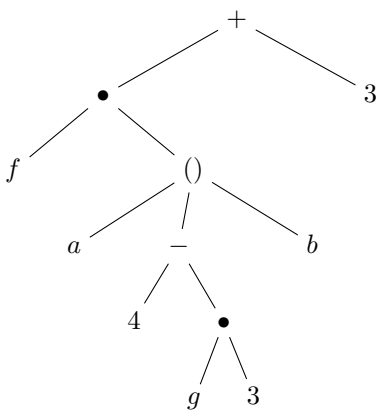
(a)



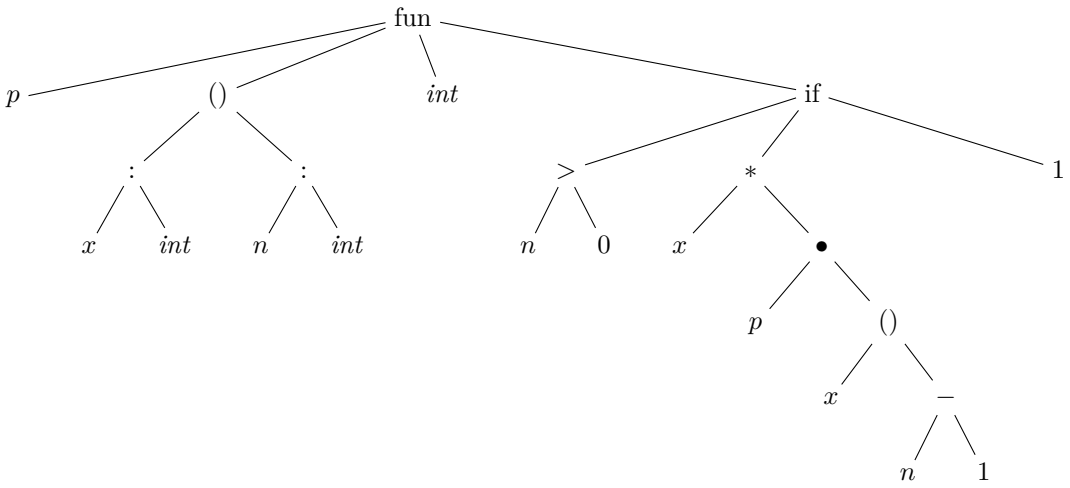
(b)



(c)



(d)



Lösungsvorschlag TB.21

(a) `f(g(3 + 5))`

(b) `int → int → unit`

(c) `f(a, 4 − g 3, b) + 3`

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

Aufgabe TB.22 *(Bäume)*
Geben Sie die Baumdarstellung der folgenden durch Zeichendarstellungen beschriebenen Phrasen an:

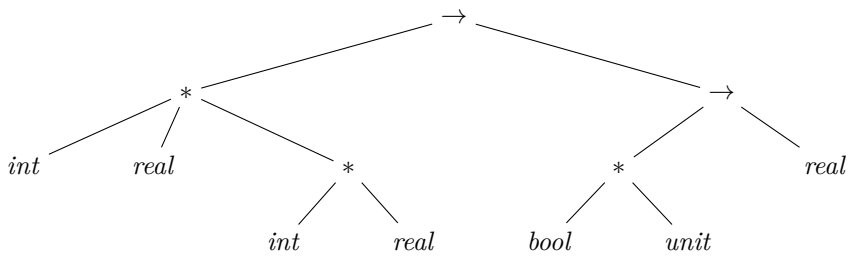
(a) `int * real * (int * real) → bool * unit → real`

(b) `if x > 4 then prozedur 5 else 0`

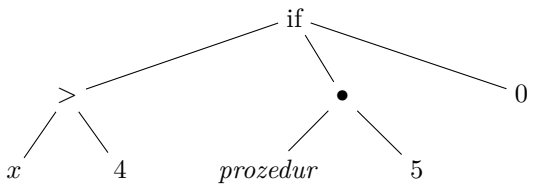
(c)
```
fun fib (n : int) : int =
    if n = 0 then 0 else
        if n <= 2 then 1 else fib (n − 1) + fib (n − 2)
```

Lösungsvorschlag TB.22

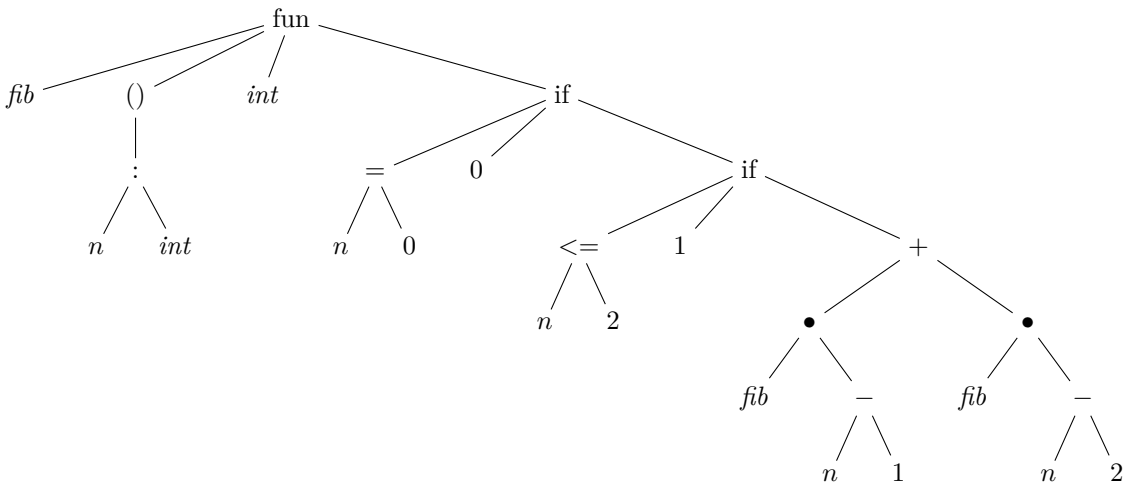
(a)



(b)



(c)



Aufgabe TB.23 *(Viele Syntaxbäume)*
Geben Sie die Baumdarstellung zu folgenden Phrasen an:

(a) `int * int * bool → int * (bool * bool)`

(b)
```
1 if 2 < 3 then if 4 * x > 8 then 3 + y * 4 else x + 4
2                            else x * (2 − 1)
```

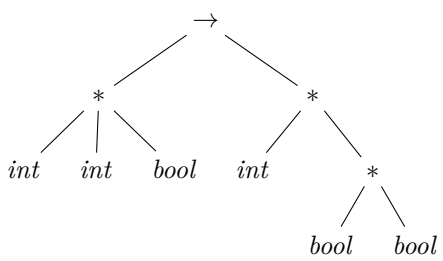
(c) `(((1, 2), (3), ((4, 5)), (6, (7, (8, 9)))))`

(d)
```
1 (fn x ⇒ fn z ⇒ x (z + 4))(fn x ⇒ x)
```

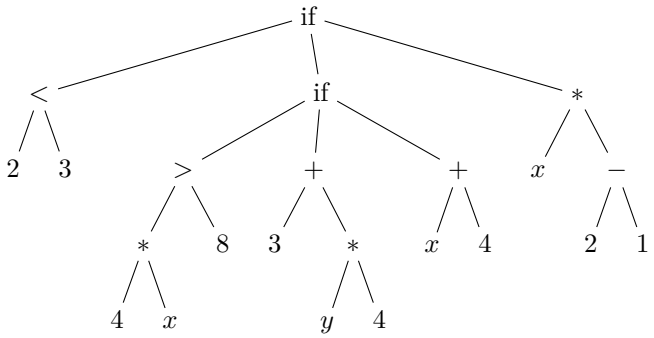
(e)
```
1 #2 ((x * 3 > 0, f (2 + 1)), 42)
```

Lösungsvorschlag TB.23

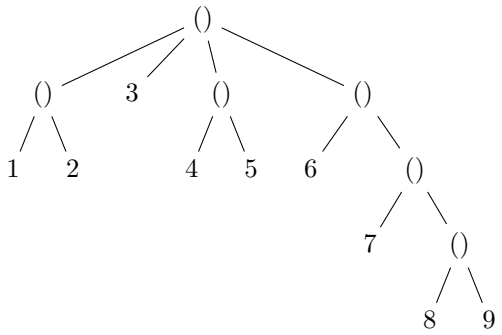
(a)



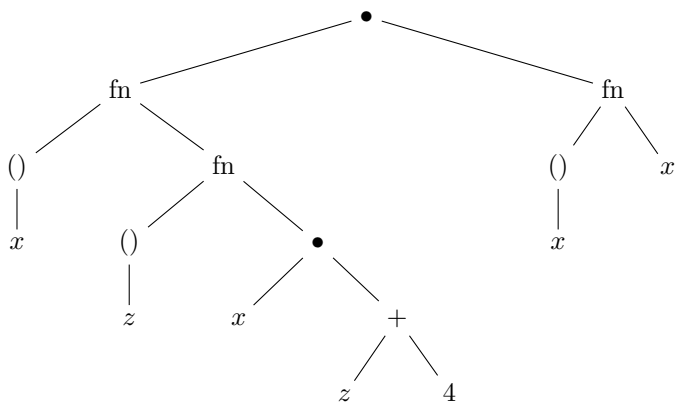
(b)



(c)



(d)



(e)