Musterlösung zur 1. Probeklausur
Programmierung 1
Wintersemester 2012/13

Das Team der Tutoren

08. Dezember 2012

Dieter Schlau Musterlösung

Name

2442424

Matrikelnummer

Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden.

Prüfen Sie dann zuerst, ob Sie alle **14** Seiten dieser Klausur erhalten haben.

Hilfsmittel sind nicht zugelassen. Am Arbeitsplatz dürfen nur Schreibgeräte, Getränke, Speisen und Ausweise mitgeführt werden. Taschen und Jacken müssen an den Wänden des Klausursaals zurückgelassen werden. Mobiltelefone sind ebenfalls dort ausgeschaltet aufzubewahren.

Das Verlassen des Saals ohne Abgabe des Klausurhefts gilt als Täuschungsversuch.

Wenn Sie während der Bearbeitung zur Toilette müssen, geben Sie bitte Ihr Klausurheft bei der Aufsicht ab. Es kann immer nur eine Person zur Toilette.

Alle Lösungen müssen auf den bedruckten rechten Seiten des Klausurhefts notiert werden. Die leeren linken Seiten dienen als Platz für Skizzen und werden **nicht korrigiert.** Notizpapier ist nicht zugelassen. Sie können mit Bleistift schreiben.

Für die Bearbeitung der Klausur stehen 90 Minuten zur Verfügung. Insgesamt können 90 Punkte erreicht werden. Zum Bestehen der Klausur genügen 45 Punkte.

Bitte legen Sie zur Identifikation Ihren Personalausweis bzw. Reisepass sowie Ihren Studierendenausweis neben sich.

Viel Erfolg!

Bitte lesen

Wir versuchen euch mit dieser Probeklausur eine Vorstellung von Ablauf und Aussehen der Klausur in Programmierung 1 zu geben. Dennoch ist sie, insbesondere was Stoffauswahl und erlaubte Hilfsmittel betrifft, nicht repräsentativ für die Klausur.

Während der Probeklausur dürfen Sie die linken Seiten der Klausur zum Schreiben benutzen.

1	2	3	4	5	6	7	8
10	13	10	21	10	9	17	(10)

Summe
90 + 10

Note

Hinweise

Folgende Prozeduren sind vordeklariert und dürfen benutzt werden, falls nicht anders angegeben:

- $iter : int \to \alpha \to (\alpha \to \alpha) \to \alpha$
- $first : int \to (int \to bool) \to int$
- $explode : string \to char\ list$
- $implode : char\ list \to string$
- $map : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$
- $length : \alpha\ list \to int$
- $rev : \alpha\ list \to \alpha\ list$
- $Int.compare : (int * int) \to order$
- $Int.min : (int * int) \to int$
- $Int.max : (int * int) \to int$
- $null : \alpha\ list \to bool$
- $hd : \alpha\ list \to \alpha$
- $tl : \alpha\ list \to \alpha\ list$
- $op:: : \alpha * \alpha\ list \to \alpha\ list$
- $op@ : \alpha\ list * \alpha\ list \to \alpha\ list$
- $op+ : (int * int) \to int$
- $op- : (int * int) \to int$
- $foldl : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$
- $foldr : (\alpha * \beta \to \beta) \to \beta \to \alpha\ list \to \beta$
- $List.concat : \alpha\ list\ list \to \alpha\ list$

Sie dürfen Prozeduren aus vorangehenden Aufgaben oder Aufgabenteilen nutzen, selbst wenn Sie diese Aufgabe nicht gelöst haben.

Aufgabe 1: Zum Aufwärmen

Teilaufgabe 1.1 (*Fast bekannt*) **3 Punkte**
Deklarieren Sie eine Prozedur *supercas*, die eine übergebene kartesische Prozedur in eine kaskadierte Prozedur umwandelt. Ihre Prozedur soll das Typschema $supercas : (\alpha * \beta * \gamma \to \delta) \to \alpha \to \beta \to \gamma \to \delta$ haben.

```
fun supercas f x y z = f (x,y,z)
```

Teilaufgabe 1.2 (*Mengen hoch zwei*) **4 Punkte**
Deklarieren Sie eine Prozedur $powerlist : \alpha\ list \to \alpha\ list\ list$, die die Potenzmenge einer gegebenen Liste berechnet.

```
fun powerlist  nil    = [nil]
  | powerlist (x::xr) = powerlist xr @ (map (fn y => x::y) (powerlist xr))
```

Teilaufgabe 1.3 (*Eigentlich ganz einfach*) **3 Punkte**
Ein Palindrom ist ein Wort, das von vorne und von hinten gelesen identisch ist, z.B. „OTTO“.

Schreiben Sie eine Prozedur $palindrom : string \to bool$, die prüft, ob ein Wort ein Palindrom ist.

Hinweis: Ihre Prozedur soll Groß- und Kleinschreibung beachten, „Otto“ soll also nicht als Palindrom erkannt werden.

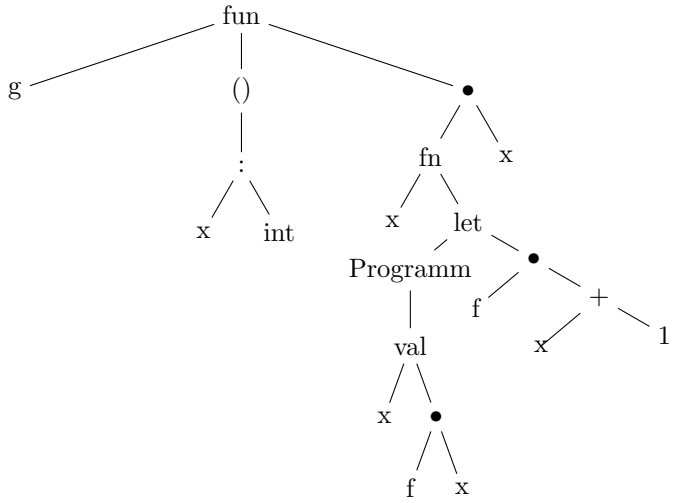
```
fun palindrom s = explode s = rev (explode s)
```

Aufgabe 2: Programmiersprachliches

Teilaufgabe 2.1 (*Lexikalische Bindungen*) **4 + 1 Punkte**
Betrachten Sie die folgende Prozedurdeklaration:

```
fun g (x:int) = (fn x => let val x = f x in f (x+1) end) x
```

(a) Geben Sie die Baumdarstellung zu der oben in Zeichendarstellung gegebenen Prozedurdeklaration an.



(b) Bereinigen Sie die Prozedurdeklaration durch konsistente Umbenennung der gebundenen Bezeichner.

```
fun g (x1 : int) = (fn x2 => let val x3 = f x2 in f (x3 + 1) end) x1
```

Teilaufgabe 2.2 (*Semantische Bindungen*) **5 Punkte**
Gegeben seien folgende Deklarationen:

```
val f = let val a = 7 in fn x => if x = 1337 then 2*a else 42 end
fun g (x : int) (y : int) : bool = g (f x) y
```

Geben Sie an, wozu der Ausdruck $g\ (f\ 1337)$ auswertet.

Hinweis: Sie dürfen auf bereits angegebene Zwischenergebnisse verweisen.

- $f :=$ (fn x => if x = 1337 then 2*a else 42, $int \to int$, $[a := 7]$)
- $g :=$ (fun g x y = g (f x) y, $int \to int \to bool$, $[f :=$ s. oben$]$)
- (fn y => g (f x) y, $int \to bool$, $[g :=$ s. oben, $f :=$ s. oben, $x := 14]$)

Teilaufgabe 2.3 (*Typen, hurra!*) **3 Punkte**

Geben Sie eine geschlossene Abstraktion an, die den folgenden Typ hat. Verzichten Sie dabei auf die Angabe von Konstanten und explizite Typangaben.

$(\alpha * `\beta \to \gamma) \to (`\beta \to \alpha) \to `\beta \to bool \to \gamma$

```
fn f => fn g => fn x => fn b => (b andalso x=x; f ((g x), x))
```

Aufgabe 3: Rekursion und Endrekursion, Iteration

Teilaufgabe 3.1 (*Rekursion und Endrekursion*) **2 + 3 + 1 Punkte**
Betrachten Sie die folgende Prozedur:

```
fun f x = let
            fun g x = if x = 0 then 1 else if x mod 2 = 0 then f (x-1) else 2 * g (x-1)
          in g x end
```

(a) Zu welchen Werten werten die Prozeduraufrufe $f\,1$, $f\,4$ und $f\,5$ aus?

2, 4 und 8

(b) Geben Sie eine endrekursive Prozedur $h : (int * int) \to int$ an, so dass $f\,x$ und $h\,(1, x)$ für alle Werte von x zum gleichen Wert auswerten.

```
fun h (a,x) = if x = 0 then a else if x mod 2 = 0 then h (a,x-1) else h (a*2, x-1)
```

(c) Deklarieren Sie mit h eine Prozedur i, die selbst nicht rekursiv ist, so dass f und i semantisch äquivalent sind.

```
fun i x = h (1,x)
```

Teilaufgabe 3.2 (*Knobeln mit iter*) **4 Punkte**
Deklarieren Sie eine nicht rekursive Prozedur $iterfac : int \to int\,list$, so dass $iterfac\,n$ für $n \geq 1$ zur Liste $[1!, 2!, 3!, ..., n!]$ auswertet. Verwenden Sie keine Hilfsprozedur außer *iter*.

```
fun iterfac n = #1 (iter n (nil, 1, 1) (fn (l,n,a) => (l @ [n*a], n+1, n*a)))
```

Aufgabe 4: Listen und Sortierung

Teilaufgabe 4.1 (*ICE*) **4 Punkte**
Betrachten Sie folgende Prozedurdeklaration:

```
fun f  nil      = nil
  | f (x :: xs) = (f xs) @ [x, x+1]
```

Schreiben Sie eine semantisch äquivalente Prozedur g, die ohne Konkatenation von Listen auskommt.

f erfüllt folgende Gleichung:

$$f\ [x_1, ..., x_n] = [x_n, x_n + 1, ..., x_1, x_1 + 1]$$

Wir deklarieren zuerst eine Hilfsprozedur g', die folgende Gleichung erfüllt:

$$g'\ [x_1, ..., x_n]\ a = [x_n, x_n + 1, ..., x_1, x_1 + 1]\ @\ a$$

Bei der Implementierung können wir :: statt @ verwenden, analog zur Implementierung der schnellen Reversion (*rev*).

```
fun g'  nil    a = a
  | g' (x::xr) a = g' xr (x::(x+1)::a)
fun g xs = g' xs nil
```

Teilaufgabe 4.2 (*Sortieren durch Mischen*) **5 Punkte**
Ergänzen Sie die folgende Deklaration, die Sortieren durch Mischen implementiert:

```
fun merge cmp  xs      nil    =

  | merge cmp  nil     ys     =

  | merge cmp (x::xr) (y::yr) = case                    of

                                   GREATER =>

                                 | LESS    =>

                                 | EQUAL   =>

fun split xs = foldl (fn (x, (ys, zs)) =>                ) (nil, nil) xs

fun msort cmp []  =

  | msort cmp [x] =

  | msort cmp xs  = let val (ys, zs) =

                    in  merge cmp (msort cmp ys) (msort cmp zs) end
```

```
fun merge cmp  xs      nil    = xs
  | merge cmp  nil     ys     = ys
  | merge cmp (x::xr) (y::yr) = case cmp (x,y) of
                                  GREATER => y :: (merge cmp (x::xr) yr)
                                | LESS    => x :: (merge cmp xr (y::yr))
                                | EQUAL   => x :: y :: (merge cmp xr yr)

fun split xs = foldl (fn (x, (ys, zs)) => (zs, x::ys)) (nil, nil) xs

fun msort cmp []  = []
  | msort cmp [x] = [x]
  | msort cmp xs  = let val (ys, zs) = split xs
                    in  merge cmp (msort cmp ys) (msort cmp zs) end
```

Teilaufgabe 4.3 (*Ein wenig komplizierter*) **3 + 3 + 4 + 2 Punkte**

Sortieren wir mit *msort* die Liste $[(1,a),(7,a),(4,a),(7,b),(3,d),(1,b),(4,b)$ nach den ersten Stellen der Tupel, so erhalten wir das folgende Ergebnis:

$$[(1,b),(1,a),(3,d),(4,a),(4,b),(7,b),(7,a)]$$

Wie Sie sehen, wurde die Reihenfolge der laut compare-Prozedur gleichen Elemente nicht beibehalten. Ein Algorithmus, der diese Reihenfolge immer beibehält, heißt *stabil*. Wir werden im Folgenden versuchen, Sortieralgorithmen zu stabilisieren.

Ein stabiler Algorithmus müsste die erste Beispielliste wie folgt sortieren:

$$[(1,a),(1,b),(3,d),(4,a),(4,b),(7,a),(7,b)]$$

(a) Schreiben Sie eine Prozedur $positions : \alpha\ list \to (\alpha * int)\ list$, die jedem Element der Eingabeliste seine Position zuordnet. Beginnen Sie mit 0.

Beispiel: $positions\ [a,b,c,d] = [(a,0),(b,1),(c,2),(d,3)]$

```
fun positions' n  nil    = nil
  | positions' n (x::xr) = (x,n) :: positions' (n+1) xr

fun positions xs = positions' 0 xs
```

(b) Schreiben Sie eine Prozedur $oldlist : (\alpha * \beta)\ list \to \alpha\ list$, so dass für alle Listen xs gilt: $oldlist\ (positions\ xs) = xs$

Beispiel: $oldlist\ [(a,0),(b,1),(c,2),(d,3)] = [a,b,c,d]$

```
fun oldlist xs = map (fn (a,b) => a) xs
```

(c) Schreiben Sie eine Vergleichsprozedur $modifycompare : (\alpha * \beta \to order) \to (\alpha * int) * (\beta * int) \to order$. Diese nimmt eine Vergleichsprozedur *compare* und zwei Tupel (a, n) und (b, m).

Ihre Prozedur soll die Tupel anhand ihrer ersten Komponenten mit der gegebenen Vergleichsprozedur vergleichen. Falls die ersten Stellen gleich sind, soll nach der zweiten Stelle sortiert werden.

```
fun modifycompare compare ((a,b),(c,d)) = case compare (a,c) of
                                            EQUAL => Int.compare (b,d)
                                          | s     => s
```

(d) Schreiben Sie nun eine Prozedur *stability*, die eine gegebene Sortierprozedur $sort : (\alpha * \alpha \to order) \to \alpha\, list \to \alpha\, list)$ stabilisiert.

```
fun stability sort (compare : 'a * 'a -> order) (xs : 'a list) : 'a list =


oldlist (sort (modifycompare compare) (positions xs))
```

Aufgabe 5: Umgebungen und Fehler

Teilaufgabe 5.1 (*Umgebungen*) **4 + 3 Punkte**

Wir benutzen im Folgenden die Typsynonyme *var* und *env*, die wie in der Vorlesung definiert sind:

```
type var = string
type env = var -> int
```

Außerdem sei folgende Ausnahme vordefiniert:

```
exception Unbound
```

(a) Schreiben Sie eine Prozedur $getValue : env \to var \to int\ option$, die eine Prozedur und einen Bezeichner nimmt. Ist der Bezeichner in der Umgebung an v gebunden, soll $SOME\, v$ ausgegeben werden, sonst soll $NONE$ ausgegeben werden.

```
fun getValue env x = SOME (env x) handle Unbound => NONE
```

(b) Geben Sie einen Ausdruck an, der zu einer Umgebung auswertet, in der alle Bezeichner, die mehr als 10 Zeichen haben, an 42 gebunden sind. Alle anderen Bezeichner sollen ungebunden bleiben.

Tipp: Nutzen Sie *length* und *explode*.

```
(fn x => if length (explode x) > 10 then 42 else raise Unbound)
```

Teilaufgabe 5.2 (*Test von Argumenten*) **3 Punkte**

Schreiben Sie eine Prozedur $check : (\alpha \to \beta) \to \alpha\ list \to bool$. Dieser wird eine Prozedur f sowie eine Liste von Argumenten übergeben. Ihre Prozedur soll nun überprüfen, ob beim Anwenden von f auf eines dieser Argumente eine Ausnahme geworfen wird. In diesem Fall soll sie $true$ zurückgeben, sonst $false$.

Benutzen Sie weder Hilfsprozeduren noch Konditionale.

```
fun check f nil     = false
    check f (x::xs) = (f x; check f xs) handle e => true
```

Aufgabe 6: Konstruktoren

Teilaufgabe 6.1 (*Logische Ausdrücke*) **2 + 3 + 4 Punkte**

Im Folgenden sehen Sie die Datentypen *mybool* und *lexp*, die logische Ausdrücke darstellen:

```
datatype mybool = True | False
datatype lexp = C of mybool | Not of lexp | And of lexp * lexp | Or of lexp * lexp
```

Dabei steht *and* für das logische Und ($\wedge$), *or* für das logische Oder ($\vee$). *Not* steht für die boolesche Negation ($\neg$).

(a) Schreiben Sie eine Prozedur $neg : mybool \to mybool$, die die boolesche Negation ($\neg$) realisiert.

```
fun neg True  = False
  | neg False = True
```

(b) Das Gesetz der doppelten Negation besagt, dass die Negation eines negierten logischen Ausdrucks seine Bejahung ist: $\neg\neg\varphi = \varphi$

Schreiben Sie eine Prozedur $dNeg : lexp \to lexp$, die doppelte Negationen in einem Ausdruck und in seinen Teilausdrücken entfernt.

Beispiel: $dNeg\,(\neg(\neg\neg true \wedge false))$ soll auswerten zu $\neg(true \wedge false)$

```
fun dNeg (Not (Not x)) = dNeg x
  | dNeg (Not x)       = Not (dNeg x)
  | dNeg (C x)         = C x
  | dNeg (And (x,y))   = And (dNeg x, dNeg y)
  | dNeg (Or  (x,y))   = Or  (dNeg x, dNeg y)
```

(c) Wir haben in unseren logischen Ausdrücken keine Variablen, sondern nur Konstanten erlaubt. Dadurch können wir den Wahrheitswert jedes Ausdrucks bestimmen.

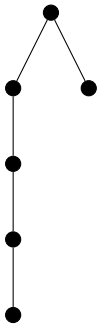
Deklarieren Sie eine Prozedur $eval : lexp \to mybool$, die den Wahrheitswert eines Ausdruckes berechnet.

```
fun eval (C x)       = x
  | eval (Not x)     = neg (eval x)
  | eval (And (x,y)) = case (eval x, eval y) of (True, True) => True | _ => False
  | eval (Or  (x,y)) = case (eval x, eval y) of (False, False) => False | _ => True
```

Aufgabe 7: Bäume

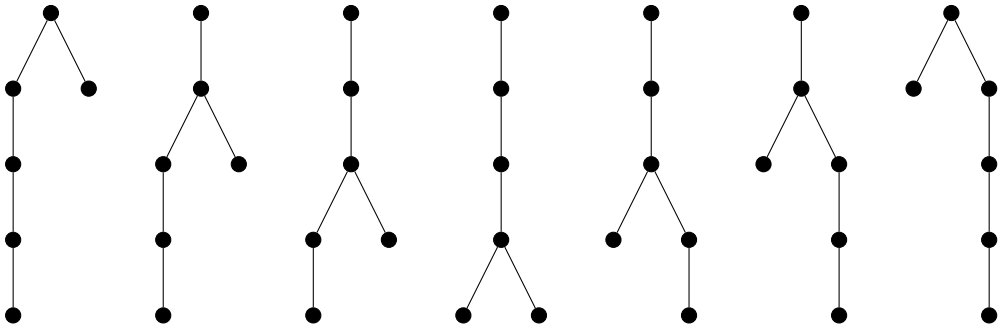
Teilaufgabe 7.1 (*Im Wald*) **3 Punkte**

(a) Geben Sie *einen* Baum mit Größe 6 und Tiefe 4 an.



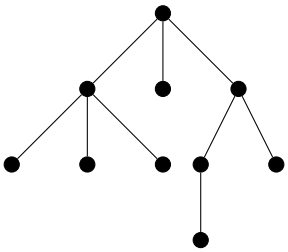
Wie viele solche Bäume gibt es?

Es gibt 7 solche Bäume:



Teilaufgabe 7.2 (*Linearisierungen*) **2 Punkte**

Geben Sie die Postlinearisierung des folgenden Baumes an:



$[0, 0, 0, 3, 0, 0, 1, 0, 2, 3]$

Teilaufgabe 7.3 (*Nicht minimalistisch*) **4 Punkte**
Schreiben Sie eine Prozedur $findmin : int\,ltr \to int$, die die minimale Marke eines markierten Baumes zurückgibt.

```
fun findMin (L (y, xs)) = foldl Int.min y (map findMin xs)
```

Teilaufgabe 7.4 (*Wir lieben Bäume!*) **5 Punkte**
Schreiben Sie eine Prozedur $postlin : tree \to int\,list\,list$, die die Adressen eines Baums in Postordnung liefert.

- Lösung mit Faltung, aus der Besprechung:

```
fun prepend x xs = (map (fn ys => x::ys) xs)

fun postlin (T ts) = (#2 (foldl (fn (t, (x, xs)) =>
                                    (x+1, xs @ (prepend x (postlin t))))
                                (1, nil) ts)) @ [nil]
```

- Lösung mit eigener Variante von *map*, mit Rekursionsgleichungen:

$$mapi\ f\ [x_1, ..., x_n] = [f(1, x_1), ..., f(n, x_n)]$$
$$postlin(T[t_1, ..., t_n] = (prepend(postlin\ t_1)@...@(postlin\ t_n))@[nil]$$

```
fun prepend x xs = (map (fn ys => x::ys) xs)
fun mapi f xs = #2 (foldl (fn (x,(n,a)) => (n+1,a @ [f (n,x)])) (1,nil) xs)
fun postlin (T ts) = (List.concat (mapi (fn (x,t) => prepend x (postlin t)) ts))
                     @ [nil]
```

- Alternative Lösung nur mit Hilfsprozedur:

```
fun postlin' a n (T  nil   ) = [a]
  | postlin' a n (T (x::xr)) = (postlin' (a @ [n]) 1 x) @ (postlin' a (n+1) (T xr))

fun postlin ts = postlin' nil 1 ts
```

Teilaufgabe 7.5 (*Für SML-Magier und Zauberlehrlinge*) **3 Punkte**
Deklarieren Sie eine Prozedur $parsePreLin : int\ list \to tree$, die aus einer gegebenen Prälinearisierung den Baum rekonstruiert.

Wenn die übergebene Liste keine gültige Prälinearisierung ist, soll eine beliebige Exception geworfen werden.

- Lösung mit *iter*:

  $parseTree\ (ls, ts)$ soll von der Teil-Linearisierung ls den ersten Baum „weglesen“ und vorne an ts hängen. So soll z.B. $parseTree\ ([\overbrace{1,0}^{\text{Prälinearisierung von } T[T[]]}, 0], [T[]])$ die Liste $[1,0]$ zu $T[T[]]$ auswerten und deshalb ($\underbrace{[0]}_{\text{Rest der Linearisierung}}, [\underbrace{T[T[]]}_{\text{neuer Baum}}, \underbrace{T[]}_{\text{alte Bäume}}]$) ausgeben.

```
exception Error

fun parseTree ( nil    , _ ) = raise Error
  | parseTree (n :: xs, ts) = let
                                val (xs, ls) = (iter n (xs, nil) parseTree)
                              in
                                (xs, T ls :: ts)
                              end

fun parsePreLin xs = case parseTree (xs, nil) of
                       (nil, [res]) => res
                     |  _           => raise Error
```

- Alternative Lösung, aus der Besprechung:

```
fun popStack n stack = iter n (stack, nil)
                      (fn (stack, res) => (tl stack, hd stack :: res))

fun parsePreLinToStack stack  nil      = stack
  | parsePreLinToStack stack (x::rlin) = let
                                           val (rstack, children)
                                               = popStack x
                                                (parsePreLinToStack stack rlin)
                                         in
                                           T (rev children) :: rstack
                                         end

fun parsePreLin lin = case parsePreLinToStack nil lin of
                        [t] => t
                      |  _  => raise Overflow
```

Aufgabe 8: Mathematische Prozeduren (Zusatz)

Hinweis: Da die Aufgaben zu Kapitel 9 in den Übungen noch nicht hinreichend besprochen wurden, ist dies nur eine Zusatzaufgabe. Für die echte Klausur ist dieses Kapitel relevant.

Teilaufgabe 8.1 (*Korrektheitssatz*) **2 + 2 + 1 + 5 Punkte**

(a) Gegeben sei die folgende mathematische Prozedur:

$$\begin{array}{rcl} p : (\mathbb{N} \times \mathbb{Z}) \to \mathbb{Z} & & \\ p\,(0, y) & = & 0 \\ p\,(x, y) & = & p\,(x-1, y) + y \quad \text{für } x > 0 \end{array}$$

Geben Sie ein Ausführungsprotokoll für $p\,(3, 7)$ an.

$$\begin{array}{rcl} p\,(3,7) & = & p\,(2,7) + 7 \\ & = & (p\,(1,7) + 7) + 7 \\ & = & ((p\,(0,7) + 7) + 7) + 7 \\ & = & 21 \end{array}$$

(b) Geben Sie die Rekursionsrelation zu p an.

$\{((x, y), (x-1, y)) | x \in \mathbb{N} \wedge y \in \mathbb{Z} \wedge x > 0\}$

(c) Geben Sie die Ergebnisfunktion f von p an.

$f : (\mathbb{N} \times \mathbb{Z}) \to \mathbb{Z}$
$f(x, y) = x \cdot y$

(d) Beweisen Sie mit dem Korrektheitssatz, dass die Prozedur p Ihre Ergebnisfunktion f berechnet.

Hinweis: Denken Sie daran, *alle* zur Anwendung des Korrektheitssatzes notwendigen Voraussetzungen zu beweisen.

Nutzen Sie die linke Seite der Klausur für Ihre Lösung.

Um den Korrektheitssatz anzuwenden, müssen wir Folgendes zeigen:

(i) *Dom* $f \subseteq$ *Dom* p
Der Definitionsbereich unserer Funkion f stimmt mit dem Argumentbereich unserer Prozedur p überein. Wir müssen noch zeigen, dass dieser auch dem *Definitionsbereich* unserer Prozedur entspricht, unsere Prozedur also immer terminiert.
Dazu geben wir eine natürliche Terminierungsfunktion an: $\lambda(x, y) \in \mathbb{N} \times \mathbb{Z}\,.\,x$

(ii) f erfüllt die definierenden Gleichungen von p für alle $x \in$ *Dom* f.
Um dies zu zeigen, ersetzen wir in den definierenden Gleichungen von p den Bezeichner p durch f und zeigen dann die Korrektheit.

- Erste Gleichung:

$$\begin{array}{rcll} f(0,y) & = & 0 \cdot y & \text{Definition } f \\ & = & 0 & \text{Arithmetik} \end{array}$$

- Zweite Gleichung:

$$\begin{array}{rcll} f(x,y) & = & x \cdot y & \text{Definition } f \\ & = & (x-1) \cdot y + y & \text{Arithmetik} \\ & = & f(x-1, y) + y & \text{Definition } f \end{array}$$

Damit können wir den Korrektheitssatz anwenden: Die Prozedur p berechnet unsere Ergebnisfunktion f. ■