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

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 6

(Kapitel 06)

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.

Aufgabe 6.1 (*Konstruktoren*)

(a) Deklariere einen Datentyp *Lecture*, der Vorlesungen beschreibt. Eine Vorlesung besteht aus ihrem Namen und der Anzahl der CP die man für sie kriegt.

(b) Deklariere jetzt unter Benutzung des Datentyps *Lecture* einen Datentyp *Student*. Ein Student besteht aus seiner Matrikelnummer(als Integer) und einer Liste der Vorlesungen die er besucht.

(c) Deklariere einen Datentyp, der Listen wie in SML beschreibt. Tipp: Du brauchst 2 Konstruktoren

Lösung 6.1:

(a) datatype Lecture = L of string*int

(b) datatype Student = S of int*Lecture list

(c) datatype 'a List = NIL | CONS of ('a*'a List)

Aufgabe 6.2 (*Bool im Eigenbau*)
Betrachte den folgenden Enumerationstyp:

```
datatype mybool = T | F
```

(a) Deklariere eine Prozedur

b2b : *mybool* → *bool*

die von der neuen Darstellung von bool in die in SML eingebaute übersetzt.

(b) Deklariere eine Prozedur

myand : *mybool* → *mybool* → *mybool*

die den logischen UND-Operator implementiert.

(c) Deklariere eine Prozedur

myor : *mybool* → *mybool* → *mybool*

die den logischen ODER-Operator implementiert.

Lösung 6.2:

(a)
```
fun b2b T = true
|b2b F = false
```

(b)
```
fun myand T x= x
|myand F _ = false
```

(c)
```
fun myor F x= x
|myor T _ = true
```

Aufgabe 6.3 (*Exceptions*)

(a) Schreibe eine Prozedur *position* : $'\alpha$ *list* $\rightarrow$ $'\alpha$ $\rightarrow$ *int*, die zu einer Liste und einem Element dieser Liste die Position zurückgibt an der dieses Element in der Liste steht. Wenn das Element mehrmals vorkommt soll das erste Vorkommen zurückgegeben werden. Kommt das Element nicht in der Liste vor so soll eine ElementNotFound Exception geworfen werden. Definiere diese vorher passend.
Beispiel: position [1,5,3,4,6] 1 soll 1 zurück geben, position [1,6,3,6,4] 6 soll 2 zurück geben.

Tipp: Deklariere eine passende Hilfsprozedur

(b) Manchmal möchte man lieber eine *option* zurückbekommen statt einer Exception. Schreibe eine Prozedur *positionOption* : $'\alpha$ *list* $\rightarrow$ $'\alpha$ $\rightarrow$ *int option* die die Prozedur aus Teil a) verwendet und NONE zurück gibt, falls die ElementNotFound Exception geworfen wird. Falls keine Exception auftritt soll der Rückgabewert von Position zurückgegeben werden (natürlich in einer Option verpackt)

(c) Verallgemeinere nun das Prinzip von b). Schreibe eine Prozedur
exception2option : $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta$ *option*),
die aus einer Prozedur die möglicherweise Exceptions wirft eine Prozedur generiert, die NONE zurück gibt falls eine beliebige Exception aufgetreten ist und SOME x (x Rückgabewert der übergebenen Prozedur) ansonsten.

Lösung 6.3:

(a)
```
exception ElementNotFound
fun positionHilf nil y _ = raise ElementNotFound
| positionHilf (x::xs) y n = if x=y then n else positionHilf xs y (n+1)
fun position xs y = positionHilf xs y 1
```

(b)
```
fun positionOption xs y = SOME (position xs y) handle ElementNotFound => NONE
```

(c)
```
fun exception2option f a = SOME (f a) handle _ => NONE
```

Aufgabe 6.4
In der Vorlesung wurde mithilfe von Konstruktoren der Typ *exp* eingeführt, mit dem wir einfache arithmetische Ausdrücke beschreiben können. Wir wollen bestimmen, ob innerhalb eines Ausdrucks einmal zwei verschiedene Variablen direkt miteinander addiert werden.
Beispiel: $2x + y$ oder $y + y$ erfüllen dies nicht, $x + y$ allerdings schon. Das Verfahren soll das Typmuster $exp \to bool$ erfüllen.

(a) Entwerfe eine Prozedur, die für einen arithmetischen Ausdruck, gegeben als *exp*, entscheidet, ob dieser eine solche Addition enthält. Schreibe dazu die Rekursionsgleichungen auf.

(b) Formuliere die Gleichungen nun zu einer SML Prozedur aus. Was fällt dir auf? Welche der beiden Notationen ist allgemeiner, welche restriktiver?

(c) Entwerfe nun eine verbesserte Version des Verfahrens, das die beiden Variablen zurckgibt, falls eine gesuchte Addition existiert. Nutze dazu das Typschema $exp \to exp * exp\ option$.
Was ist der Nachteil, wenn man bei dieser Art von Problem auf Wahrheitswerte verzichtet?

Lösung 6.4:

(a) $find\ C\ x = false$
$find\ V\ x = false$
$findM(a, b) = find\ a \vee find\ b$
$findA(a, b) = true, falls\ a = Vv_1 \wedge b = Vv_1$
$findA(a, b) = find\ a \vee find\ b, sonst$

(b)
```
fun find C x = false
| find V x = false
| find M(a,b) = find a orelse find b
| find A(V v,V v2) = v = v2
```

Die SML Prozedur ist eine konkrete Umsetzung einer mathematischen Funktion. Die Syntax ist restriktiver, aber es gibt auch syntaktischen Zucker. Aber generell ist der Weg von den Rekursionsgleichungen bis zur konkreten Prozedur nicht mehr weit.

(c)
```
fun find C x = None
| find V x = None
| find M(a,b) = case (find a) of Some x ⇒ Some x | None ⇒ case (find b) of Some y ⇒ Some y
| None ⇒ None
| find A(a,b) = case a of (V v1) ⇒ case b of (V v2) ⇒ if v1 <> v2 then Some (v1, v2) else None
| x ⇒ find x | y ⇒ find (M (a,b))
```

Sobald man keine Basistypen mehr hat fällt es deutlich schwerer, Werte aus den Blättern des Rekursionsbaums nach oben zu geben bzw zu akkumulieren, da vorgefertigte Konstrukte wie die Operationen auf den ganzen Zahlen (+, *, div, ...) und boolesche Operatoren ($\vee$, $\wedge$, ...) nicht mehr verfügbar sind. Das führt bei eigenen Typen oft zu stark verschachtelten case Ausdrücken.

Aufgabe 6.5
Das Versandunternehmen T&S bietet die Filme *AmericanPie*, *DieHard*, *Skyfall* und *Saw*. Sie haben die Aufgabe bekommen den Onlinehandel zu verwalten.

(a) Überlegen Sie sich einen geeigneten Datentyp *movie*. Bei Filmen die aus mehreren Teilen bestehen soll es die Möglichkeit geben die Nummer mit anzugeben.

(b) Schreiben Sie eine Prozedur $compareMovie : movie * movie \rightarrow order$, wobei *AmericanPie* am größten ist und *Saw* am kleinsten. Testen Sie diese mit einem Sortieralgorithmus ihrer Wahl.

(c) Wir wollen jetzt herausfinden, wie viele Filme in unserem Warenkorb sind. Überlegen Sie sich zuerst wie man den Warenkorb am einfachsten beschreiben kann. Schreibe dazu eine Prozedur $getCount : shoppingcart \rightarrow int * int * int$ die ein Tupel liefert in dem steht wieviele *AmericanPie*, *DieHard* und *Saw* Filme im Warenkorb sind. Kommt der Film *Skyfall* im Warenkorb vor, soll die exception *Raubkopie* geworfen werden (da der Film noch nicht im Handel ist).

Lösung 6.5:

(a)
```
datatype movie = AmericanPie of int | DieHard of int | Skyfall | Saw of int
```

(b)
```
fun compareMovie (AmericanPie x, AmericanPie y) = Int.compare(x,y)
| compareMovie (AmericanPie _, _) = GREATER
| compareMovie (_, AmericanPie _) = LESS
| compareMovie (DieHard x, DieHard y) = Int.compare(x,y)
| compareMovie (DieHard _, _) = GREATER
| compareMovie (_, DieHard _) = LESS
| compareMovie (Skyfall, Skyfall) = EQUAL
| compareMovie (Skyfall, _) = GREATER
| compareMovie (_, Skyfall) = LESS
| compareMovie (Saw x, Saw y) = Int.compare(x,y)
```

(c)
```
type movielist = movie list
exception Raubkopie
fun getCount ml = foldl (fn (AmericanPie _, (ar, dr, sr)) => (ar+1, dr, sr)
                         | (DieHard _, (ar, dr, sr)) => (ar, dr+1, sr)
                         | (Saw _, (ar, dr, sr)) => (ar, dr, sr+1)
                         | _ => raise Raubkopie)
                                                  (0, 0, 0) ml
```

Aufgabe 6.6 (*Ausdrücke*)

(a) Schreiben sie eine Prozedur $expToString : exp \rightarrow string$, die einen Ausdruck vom Typ *exp* nimmt und diesen als String ausgibt. Dabei soll die Ausgabe voll geklammert sein.

```
datatype exp = C of int | V of string | A of exp*exp | M of exp*exp

expToString (A(M(C 4, V "x") , M(V "x", C 2))) = "((4 * x) + (x * 2))"
```

Sie dürfen $Int.toString : int \rightarrow string$ verwenden.

(b) (**Achtung schwer!**) Schreiben sie die Prozedur $expToString2 : exp \rightarrow string$, wobei sie diesmal die minimal erforderliche Klammerung ausgeben.

```
expToString2 (A(A(C 4, V "x") , M(V "x", C 2))) = "4 + x + x*2"
```

Lösung 6.6:

(a)
```
fun expToString (C x) = Int.toString(x)
| expToString (V x) = x
| expToString (A (a,b)) = "("^expToString a ^"+ "^expToString b ^")"
| expToString (M (a,b)) = "("^expToString a ^"* "^expToString b ^")"
```

(b)
```
fun expToString2 (C x) = Int.toString(x)
| expToString2 (V x) = x
| expToString2 (A (a,b)) = expToString2 a ^"+ "^expToString2 b
| expToString2 (M (a,b)) = let
          val sa = case a of A _ => "("^expToString2 a ^")"
                | _ => expToString2 a
          val sb = case b of A _ => "("^expToString2 b ^")"
                | _ => expToString2 b
         in sa ^"*"^sb end
```

1 Knobelecke

Aufgabe 6.7 (*Kürzeste Wege*)
In dieser Aufgabe sollen Sie ein kleines Navigationssystem implementieren:

(a) Zunächst benötigen wir Repräsentationen für die Objekte in einer Landkarte: Deklarieren Sie Datentypen *node* und *edge*, so dass sie Städte, Dörfer und Sehenswürdigkeiten darstellen können, die durch Straßen, Pfade und Autobahnen miteinander verbunden sind. Die Position eines Knotens auf der Karte ist dabei vom Datentyp $point = int * int$

(b) Ihrer Liste von Deklarationen fügen wir noch

```
type map = node -> edge list
```

hinzu. Implementieren Sie nun einige Hilfsprozeduren:

(i) $getNodes : edge \to node * node$ soll eine Kante auf die durch sie verbundenen Knoten auf der Karte abbilden

(ii) $addNode : map \to node \to map$ soll einer Karte einen Knoten hinzufügen

(iii) $addEdge : map \to edge \to map$ soll einer Karte eine Verbindung hinzufügen

(iv) $getEdge : map \to node \to node \to node\ option$ soll die Kante zurückgeben, die zwei gegebene Knoten verbindet

(v) $getNeighbor : node \to edge \to node$ soll zu einem gegebenen Knoten den Knoten einer Kante zurückgeben, der ungleich dem gegebenen Knoten ist.

(c) Implementieren Sie eine Prozedur $dist : edge \to real$, die die Länge einer Verbindung berechnet

(d) Implementieren Sie eine Prozedur $time : edge \to real$, die die Zeit berechnet, die nötig ist, um eine Verbindung zu benutzen. Natürlich ist die Zeit, die für eine Verbindung benötigt wird, von ihrem Typ abhängig!

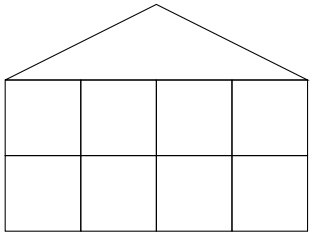
(e) Implementieren Sie eine Prozedur $getPath : map \to (edge \to real) \to (node \to bool) \to node \to node \to nodelist*real$ die beim Aufruf $getPath\ m\ w\ v\ a\ b$ den kürzesten Pfad von a nach b berechnet und Kanten dabei mit der Prozedur w gewichtet. Da in einem kürzesten Pfad kein Knoten zweimal vorkommt, ist es für die Rekursion hilfreich, auch noch einen Parameter v zu übergeben, der angibt, welche Knoten schon besucht wurden. Sie sollten den kürzesten Pfad auf naive Weise suchen: Sie beginnen bei Stadt a und betrachten (per Rekursion) für jeden Nachbarn a' den kürzesten Pfad von a' nach b.

(f) Erzeugen Sie sich mit *addNode* und *addEdge* eine Beispielkarte und testen Ihre Prozedur *getPath* darauf!

Lösung 6.7:

```
type point = int * int
datatype node = City of string * point | Village of string * point | Sight of string * point
exception Double of string
exception Empty of string
datatype edge = Road of node * node | Path of node * node | Highway of node * node
fun getNodes (Road t) = t | getNodes (Path t) = t | getNodes (Highway t) = t
type map = node -> edge list
fun emptyMap (_ : node) : edge list = raise Empty ``Node not found!''
fun addNode (m:map) c x = if c = x then nil else m x
fun addEdge (m:map) e x = let val (a, b) = getNodes e in if a = x orelse b = x then e :: (m x)
else m x end
fun contains e c = let val (a, b) = getNodes e in a = c orelse b = c end
exception Found of edge
fun getEdge (m:map) a b = (foldl (fn (e, _ ) => if contains e b then raise Found e else ()) ()
(m a); NONE) handle Found e => SOME e
fun getNeighbor c e = let val (a, b) = getNodes e in if a = c then b else a end
fun pos (City(_ , p)) = p | pos (Village(_ , p)) = p | pos (Sight (_ , p)) = p
fun dist e = let val (a, b) = getNodes e val (x1, y1) = pos a val (x2, y2) = pos b in (Math.sqrt(Real.fromInt
+ (y1-y2)*(y1-y2)))) end
fun time e = (dist e * (case e of Road(_ ) => 1.0 | Path (_ ) => 3.0 | Highway(_ ) => 0.75))
fun adj env k v x = if x = k then v else env x
fun getPath' m w v a b = if a = b then (nil, 0.0) else let val edges = m a in foldl (fn (e, (p,
l)) => let val n = (getNeighbor a e) in if v n then (p, l) else let val (p', l') = getPath' m w
(adj v n true) n b val (p', l') = (e:: p', l' + w e) in if l' < l then (p', l') else (p, l) end
end)
([hd edges], Real.fromInt(valOf Int.maxInt)) edges end
fun getShortestPath m a b = #1(getPath' m dist (fn _ =>false) a b)
fun getFastestPath m a b = #1(getPath' m time (fn _ =>false) a b )
```

Aufgabe 6.8 (*Dr. Schlau, gelernter Häuslebauer*)
Wie sie sicher schon wussten, lebt ihr bester Freund, Dieter Schlau, in einen kleinen und nahezu unbekannten, zweidimensionalen Viertel von Saarbrücken. In Ermangelung einer dritten Dimension leben die Menschen dort in Mehrfamilienhäusern mit quadratischen Wohnungen, die alle gleich groß sind.



Wie Sie sehen, besteht ein Haus zumindest aus einem Dach. Ein Haus besteht aus einem Haus, unter das noch ein Stockwerk gebaut wurde.[1]

```
datatype haus = Dach | Haus of haus * stockwerk
```

Stockwerke werden natürlich nach dem gleichen Prinzip gebaut: Es gibt leere Stockwerke und Stockwerke, die aus einer Wohnung (links) und noch einem Reststockwerk (rechte daneben) gebaut wurden.

```
datatype stockwerk = Nix | Wohnung of bewohner * stockwerk | Leerstand of stockwerk
```

Bewohner haben ihre Wohnung gemietet oder gekauft, und werden mit ihrem Namen identifiziert. Natürlich könnte eine Wohnung aus leer stehen.

[1] Wie, Sie dachten, Häuser werden von unten nach oben gebaut? Nicht mit Dieter Schlau!

```
type name = string
datatype bewohner = Mieter of name | Besitzer of name
```

(a) Beschreiben Sie Dieter Schlaus Eigenheim, das nur aus einer Wohnung besteht, mit dem Typ *haus*. Es sieht so aus:

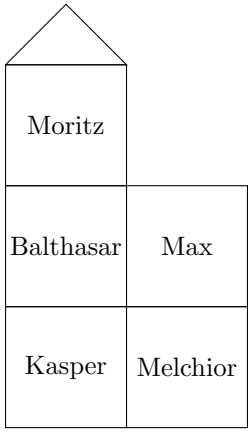
(b) Dieter Schlau ist von Beruf Häuslebauer. Leider muss er immer viel Zeit darauf verwenden, die Häuser zu planen. So wird Dieter nie reich werden! Helfen Sie ihm!

Schreiben Sie eine Prozedur $fuerDieter : name\ list \to int \to haus$, die Dieter reich macht. Die Prozedur bekommt eine Liste von zukünftigen Mietern und die Größe (Breite) des Grundstücks, auf dem Sie bauen dürfen.

Um den Gewinn zu maximieren, müssen Sie darauf achten, ein möglichst niedriges Haus zu bauen (also die Breite voll auszunutzen). Außerdem sollen Sie keine Leerstände erzeugen.

Damit die Mieter auch ihre Wohnung finden, müssen die Wohnungen von links unten nach rechts oben gemäß der gegebenen Reihenfolge der Liste erstellt werden.

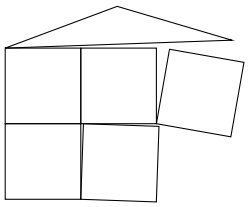
Beispielsweise soll $fuerDieter$ [„Kasper“, „Melchior“, „Balthasar“, „Max“, „Moritz“] 2 zu folgendem Hochhaus auswerten:



(c) Endlich reich und berühmt! In seinem Übermut versucht sich Dieter als Künstler. Allerdings schlägt er dabei oft über die Stränge.

Helfen Sie ihm, in dem Sie eine Prozedur $wirfAuge : haus \to unit$ schreiben, die ein Auge auf Dieter wirft. Diese soll das ihr übergebene Haus auf Planungsfehler überprüfen.

Eine Planungsfehler besteht z.B. darin, dass in einem Haus Wohnungen überstehen:



Auch dürfen aus Gründen des Brandschutzes in einem Haus maximal 25 Wohnungen (inklusive Leerständen) existieren.

Auch vergisst Dieter gelegentlich, überhaupt eine Wohnung einzubauen. Dies ist ebenfalls nicht erlaubt.

Zuletzt sei erwähnt, dass ein Dach auf mindestens einer Wohnung liegen muss, das Stockwerk darunter also mindestens eine Wohnung haben muss.

Wenn ein Planungsfehler gefunden wird, soll die Ausnahme

```
exception SoGehtDasNicht of haus
```

mit dem ungültigen Haus geworfen werden. Ist alles in Ordnung, soll () ausgegeben werden.

Lösung 6.8:

(a)
```
val dietershaus = Haus (Dach, Wohnung (Besitzer Dieter Schlau, Nix));
```

(b)
```
fun trySublist _ 0 0 = nil | trySublist (x::xr) 0 m = x :: trySublist xr 0 (m-1) | trySublist
(x::xr) n m = trySublist xr (n-1) m | trySublist nil _ _ = nil
fun bauStockwerk nil = Nix | bauStockwerk (x :: xr) = Wohnung (Mieter x, bauStockwerk xr)
fun fuerDieter xs n = iterdn ((length xs - 1) div n) 0 Dach (fn (x,h) => Haus (h, bauStockwerk
(trySublist xs (x * n) n)))
```

(c)
```
fun countStock Nix = 0 | countStock (Wohnung (_ , s)) = 1 + countStock s | countStock (Leerstand
s ) = 1 + countStock s
fun countHaus Dach = 0 | countHaus (Haus (h, s)) = countStock s + countHaus h
fun verifyCounts Dach _ = true | verifyCounts (Haus (h, s)) n = countStock s <= n andalso countStock
s > 0 andalso verifyCounts h (countStock s)
fun wirfAuge Dach = raise SoGehtDasNicht Dach | wirfAuge (Haus (h, s)) = if verifyCounts h (countStock
s) andalso countHaus (Haus (h, s)) > 0 andalso countHaus (Haus (h, s)) < 25 then () else raise
SoGehtDasNicht (Haus (h, s))
```