Klausur Programmierung WS2003/04

Prof. Dr. Andreas Podelski und Dipl. Inf. Andrey Rybalchenko

13. Dezember 2003

____
Vor- und Nachname

____
Matrikelnummer Sitz-Nr.

- Bitte öffnen Sie das Klausurheft erst dann, wenn Sie dazu aufgefordert werden (gleiche Bearbeitungszeit für alle).
- Sie können die Klausur nur auf dem für Sie vorgesehen Platz mitschreiben.
- Sie dürfen die Hilfsprozeduren `foldl`, `foldr`, `map` und `iter` verwenden, falls die Aufgabenstellung nichts über die Hilfsprozeduren aussagt.
- Hilfsmittel sind nicht zugelassen. Am Arbeitsplatz dürfen nur Schreibgerte, Getränke, Speisen sowie Ausweise mitgeführt werden. Taschen, Jacken und Mntel müssen an den Wänden des Klausursaals zurückgelassen werden.
- Verlassen des Saals ohne vorherige Abgabe des Klausurhefts gilt als Täuschungsversuch.
- Wenn Sie während der Bearbeitung auf die Toilette müssen, geben Sie bitte Ihr Klausurheft bei der Aufsicht ab. Es kann immer nur eine Person auf die 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 150 Minuten zur Verfügung. Insgesamt sind 150 Punkte erreichbar. Die für jede Aufgabe angegebene Punktzahl gibt Ihnen also einen Anhaltspunkt, wieviel Zeit Sie auf die Bearbeitung der Aufgabe verwenden sollten. Zum Bestehen der Klausur genügen 75 Punkte.
- Keine Abgaben während der letzten 30 Min. der Bearbeitungszeit.
- Bitte legen Sie Ihren Personalausweis oder Reisepass sowie Ihren Studierendenausweis zur Identifikation neben sich.
- Viel Erfolg!

Version I

1	2	3	4	5	6	7	8	*	$\sum$	Note
12	14	21	12	24	30	12	25	24	150	

Aufgabe 1 (Sequenzialisierung). $(12 = 5 + 7)$
Wir nehmen an, dass ein Prozeduraufruf zuerst sein Argument auswertet und dann den Rumpf. Vergessen Sie die Variablen nicht, die in den Ausdrucken frei vorkommen.
(a) Führen Sie eine zweistellige Sequenzialisierung $(e_1; e_2)$ auf Abstraktionen und Applikationen zurück (ohne Tupel).
(b) Führen Sie eine dreistellige Sequenzialisierung $(e_1; e_2; e_3)$ auf Abstraktionen und Applikationen zurück (ohne Tupel).

Aufgabe 2 (Kaskadierte und kartesische Prozeduren). $(14 = 6 + 8)$
a) Geben Sie zwei Prozeduren

```
kask2: ('a * 'b -> 'c) -> 'a -> 'b -> 'c

kart2: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

an, sodass `kask2` zur kartesischen Darstellung einer zweistelligen Operation die kaskadierte Darstellung und `kart2` zur kaskadierten Darstellung die kartesische Darstellung liefert.
b) Geben Sie zwei Prozeduren

```
kask3: ('a * 'b * 'c -> 'd) -> 'a -> 'b -> 'c -> 'd

kart3: ('a -> 'b -> 'c -> 'd) -> 'a * 'b * 'c -> 'd
```

an, sodass `kask3` zur kartesischen Darstellung einer dreistelligen Operation die kaskadierte Darstellung und `kart3` zur kaskadierten Darstellung die kartesische Darstellung liefert.

Aufgabe 3 (Höherstufige Prozeduren). $(21 = 5 + 8 + 8)$
a) Geben Sie eine *rekursive* Prozedur

```
multRec: (int -> int) -> int -> int
```

an, die für eine Prozedur f und eine Zahl $n \geq 0$ das Produkt $1 \cdot (f\ 1) \cdot \ldots \cdot (f\ n)$ berechnet.
b) Geben Sie eine *nicht rekursive* Prozedur

```
multNotRec: (int -> int) -> int -> int
```

an, die für eine Prozedur f und eine Zahl $n \geq 0$ das Produkt $1 \cdot (f\ 1) \cdot \ldots \cdot (f\ n)$ berechnet.
c) Geben Sie eine Prozedur

```
mult: (int -> int) -> int -> int -> int
```

an, die für $k \geq 0$ und $m \geq 0$ die Gleichung

$$\mathit{mult}\ f\ m\ k = 1 \cdot f(m+1) \cdot \ldots \cdot f(m+k)$$

erfüllt.

Aufgabe 4 (Binding). $(12 = 9 + 3)$
Betrachten Sie den Ausdruck:

```
let
  val (x , y ) = (1, x )
  fun f  x  x  z  = (x , y , z )
  fun g  g  = f  g  z
in
  g (y , z )
end
```

a) Stellen Sie die lexikalischen Bindungen des Ausdrucks durch Überstreichen und Indizieren von Bezeichnerauftreten dar.
b) Welche Bezeichner kommen in dem Ausdruck frei vor?

Aufgabe 5 (Listen). $(24 = 6 + 6 + 6 + 6)$
a) Geben Sie eine Prozedur

```
drop: int list -> int list
```

an, die aus einer Liste alle auf ungeraden Positionen vorkommenden Elemente entfernt, z.B., `drop [1,2,3,4]` ist `[2,4]`.
b) Geben Sie eine Prozedur

```
sorted: int list -> bool
```

an, die testet, ob eine Liste aufsteigend sortiert ist. Verwenden Sie dabei keine Hilfsprozeduren.
c) Geben Sie eine Prozedur

```
insert: int -> int list -> int list
```

an, die ein Element in eine aufsteigend sortierte Liste einfügt, so dass das Ergebnis wieder eine aufsteigend sortierte Liste ist.
d) Geben Sie eine Prozedur

```
sort: int list -> int list
```

an, die eine Liste aufsteigend sortiert. Benutzen Sie dabei die Prozedur `insert`.

Aufgabe 6 (Konstruktortypen). $(30 = 8 + 8 + 14)$
Seien die folgenden Deklarationen gegeben:

```
datatype expr = Const of int | Sum of expr list
                             | Prod of expr list
exception NotYetImplemented
```

Die Werte des Typs `expr` stellen arithmetische Ausdrücke über die ganzen Zahlen dar, die mit Konstanten, Addition und Multiplikation (von beliebig vielen Operanden) gebildet sind. Die leere Summe `Sum []` hat den Wert 0; das leere Produkt `Prod []` hat den Wert 1.
a) Geben Sie Prozeduren

```
eval_sum: expr -> int
```

und

```
eval_prod: expr -> int
```

an, die den Wert eines `Sum` und `Prod` Ausdrucks liefern, falls dies keine `Sum` und `Prod` Teilausdrucke enthält. Sonst soll die Ausnahme `NotYetImplemented` geworfen werden.
b) Geben Sie eine Prozedur

```
eval: expr -> int
```

an, die den Wert eines Ausdrucks liefert.

Aufgabe 7 (Ausnahmen). $(12 = 6 + 6)$
a) Geben Sie eine Prozedur

```
test: int -> bool
```

an, die testet, ob das Quadrat einer ganzen Zahl im darstellbaren Zahlbereich liegt. Wir nehmen an, dass die Ausnahme `Overflow` geworfen wird, falls eine ganze Zahl nicht darstellbar ist.
b) Geben Sie eine Prozedur

```
max_squared: unit -> int
```

an, die die größte ganze Zahl liefert, deren Quadrat im darstellbaren Zahlbereich liegt.

Aufgabe 8 (Bäume). $(25 = 9 + 16)$
Sei die folgende Deklaration gegeben:

```
datatype 'a tree = T of 'a * 'a tree list
```

a) Geben Sie eine Prozedur

```
size: 'a tree -> int
```

an, die die Größe eines Baums berechnet. Die Größe eines Baums ist die Anzahl seiner Knoten.
b) Geben Sie eine Prozedur

```
odd_size: 'a tree -> int
```

an, die die Anzahl der Knoten eines Baums liefert, die eine ungerade Tiefe haben. Vorschlag: Verwenden Sie eine Hilfsprozedur, in der Sie explizit angeben, auf welcher Tiefe im Baum die Berechnung stattfindet.

Bonus-Aufgabe 1 (Tautologie-Test). $(24 = 6 + 6 + 6 + 6)$
Seien die folgenden Deklarationen gegeben:

```
type var = int
datatype bexp = Const of bool | Var of var
              | Neg of bexp | And of bexp * bexp
              | Or of bexp * bexp
```

Die Werte des Typs `bexp` stellen boolsche Ausdrücke dar, die mit boolschen Konstanten, Variablen, Negation, Konjunktion und Disjunktion gebildet sind.
Ein boolscher Ausdruck ist eine *Tautologie*, wenn er unter allen boolschen Variablenbelegungen zu `true` evaluiert.
Sei φ ein boolscher Ausdruck mit den freien Variablen $(x_1, \ldots, x_n)$. Wir geben eine induktive Definition einer Tautologie über die Anzahl von den belegten Variablen.

$$\begin{aligned} \mathit{taut}\ 0\ \varphi &\equiv \mathit{eval}\ \varphi \\ \mathit{taut}\ i\ \varphi &\equiv \mathit{taut}\ (i-1)\ \varphi[x_i := \texttt{true}] \\ &\quad \wedge \mathit{taut}\ (i-1)\ \varphi[x_i := \texttt{false}] \end{aligned}$$

Hier: $\wedge$ bedeutet Konjunktion, $\varphi[x_i := \texttt{value}]$ ersetzt die Variable x_i in dem Ausdruck φ durch den Wert `value`. Der Ausdruck φ ist eine Tautologie genau dann, wenn $(\mathit{taut}\ n\ \varphi) = \texttt{true}$.
Wir nehmen an, dass die Variablen $(x_1, ..., x_n)$ durch die Terme `Var(1)`, ..., `Var(n)` dargestellt sind.
a) Geben Sie eine Prozedur

```
subst: bexp -> int -> bool -> bexp
```

an, so dass `subst phi id value` in dem boolschen Ausdruck `phi` jedes Vorkommen der Variable `Var(id)` durch den Wert `value` ersetzt.
b) Geben Sie eine Prozedur

```
eval: bexp -> bool
```

an, die geschlossene boolsche Ausdrucke (ohne freien Variablen) evaluiert.
c) Geben Sie die Prozedur

```
taut: int -> bexp -> bool
```

an, die testet, ob ein boolscher Ausdruck eine Tautologie ist.
d) Ein boolscher Ausdruck ist *erfüllbar*, falls eine Variablenbelegung existiert, bei der der Ausdruck den Wert `true` hat. Wie kann die Prozedur `taut` geändert werden, damit sie einen Erfüllbarkeitstest implementiert?