5. Übungsblatt zu Programmierung WS 2002 / 03

Prof. Dr. Gert Smolka und Dipl.-Inform. Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws02/

Abgabe: Montag, 25. November 2002

Schicken Sie ihre Lösungen mit einer E-Mail an Ihren Übungsgruppenleiter. Geben Sie in der E-Mail zuerst ihren Name und die Nummer Ihrer Übungsgruppe an. Bitte reichen Sie alle Lösungen mit nur einer E-Mail ein.

Zusammengefügt sollen Ihre Lösungen ein semantisch zulässiges Programm ergeben. Prüfen Sie das, indem Sie ihre Lösungen in eine Datei `x.sml` schreiben und diese mittels der Eingabe `use "x.sml"` in einen neu gestarteten Interpreter einlesen. Wenn Sie eine Standardstruktur *X* benötigen, laden Sie diese am Anfang der Datei `x.sml` mit einer Zeile `; load "X" ;`. Trennen Sie ihre Lösungen durch Kommentare der Form `(* Aufgabe 5.1 *)`.

Aufgabe 5.1: Darstellung von Zahlen $(20 = 5 * 4)$ Mit den Werten des Typs

```
datatype nat = N | S of nat
```

kann man die natürlichen Zahlen darstellen: `N` $\mapsto 0$, `S N` $\mapsto 1$, `S(S N)` $\mapsto 2$, und so weiter.

(a) Schreiben Sie eine Prozedur *add: nat * nat* $\to$ *nat*, die Darstellungen addiert. Beispielsweise soll `add(S N, S(S N)) = S(S(S N))` gelten. Verwenden Sie dabei keine Operationen für ganze Zahlen.

(b) Schreiben Sie eine Prozedur *mul: nat * nat* $\to$ *nat*, die Darstellungen multipliziert. Verwenden Sie dabei die Prozedur `add`.

(c) Schreiben Sie zwei Prozeduren

```
to   : int -> nat  (* Subscript *)
from : nat -> int
```

die Zahlen in ihre Darstellungen und Darstellungen in die dargestellten Zahlen überführen. Die Prozedur `to` soll die Ausnahme `Subscript` werfen, falls ihr Argument keine natürliche Zahl ist.

(d) Definieren Sie mithilfe von `nat` einen Typ `integer`, mit dem sich die ganzen Zahlen darstellen lassen. Dabei soll für jede ganze Zahl genau eine Darstellung existieren.

(e) Geben Sie eine Prozedur *from' : integer* $\to$ *int* an, die zu einer Darstellung die dargestellte Zahl liefert. Verwenden Sie dabei die Prozedur `from`.

Aufgabe 5.2: B-Bäume $(12 = 3 * 4)$

(a) Schreiben Sie eine Prozedur

```
bsum : int btree → int
```

die die Summe aller Marken eines B-Baums liefert. Wenn eine Marke mehrfach auftritt, soll sie auch bei der Summenbildung mehrfach berücksichtig werden.

(b) Schreiben Sie eine Prozedur

```
bmap : ('a → 'b) → 'a btree → 'b btree
```

die eine Prozedur auf alle Marken eines B-Baums anwendet.

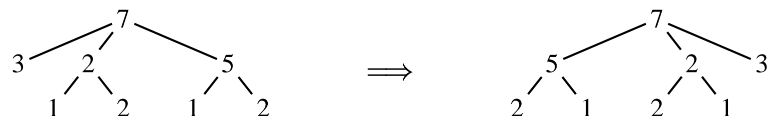
(c) Schreiben Sie eine Prozedur

```
bmember : ('a → bool) → 'a btree → bool
```

die für eine Prozedur p testet, ob ein Baum eine Marke enthält, für die p den Wert *true* liefert.

Aufgabe 5.3: L-Bäume ($12 = 3 * 4$) Schreiben Sie Prozeduren `lsum`, `lmap` und `lmember` für L-Bäume, die den Prozeduren für B-Bäume in Aufgabe 5.2 entsprechen.

Aufgabe 5.4: Spiegeln von Bäumen ($8 = 2 * 4$) Spiegeln reversiert die Ordnung der Teilbäume eines Baums:



Schreiben Sie Prozeduren

```
bmirror : 'a btree → 'a btree
lmirror : 'a ltree → 'a ltree
```

die B- und L-Bäume spiegeln.

Aufgabe 5.5: Schneller Test auf Doppelauftreten ($12 = 1 + 3 + 4 + 4$) Sie sollen eine Prozedur

```
test : int list → bool
```

schreiben, die testet, ob in einer Liste ein Element mehrfach auftritt. Dabei soll die Liste mit einer modifizierten Sortierprozedur sortiert werden, die eine Ausnahme wirft, sobald zwei Elemente der Liste miteinander verglichen werden, die gleich sind. Wenn man einen schnellen Sortieralgorithmus verwendet, bekommt man mit dieser Methode einen schnellen Test auf Doppelauftreten.

(a) Deklarieren Sie eine Ausnahme `Double`.

(b) Schreiben Sie eine Prozedur

```
order : int * int → order  (* Double *)
```

die zu (m, n)

(i) den Wert `LESS` liefert, wenn $m < n$.

(ii) den Wert `GREATER` liefert, wenn $m > n$.

(iii) die Ausnahme `Double` wirft, wenn $m = n$.

(c) Schreiben Sie mithilfe der Sortierprozedur

```
Listsort.sort : ('a * 'a → order) → 'a list → 'a list
```

aus der Standardstruktur `Listsort` eine modifizierte Sortierprozedur

```
dsort : int list → int list  (* Double *)
```

die die Ausnahme `Double` wirft, sobald sie zwei gleiche Elemente miteinander vergleicht.

(d) Schreiben Sie eine Prozedur

$test : int\ list \to bool$

die testet, ob in einer Liste ein Element mehrfach auftritt. Schreiben Sie `test` mit einem Let-Ausdruck, in dem Sie `Double`, `order` und `dsort` lokal deklarieren.

Aufgabe 5.6: Programmieren mit Ausnahmen und Optionen ($10 = 4 + 6$)

(a) Sei die Deklaration

exception Found of int

gegeben. Schreiben Sie eine Prozedur

$findi : (int \to bool) \to int\ btree \to unit$ *(* Found *)*

die zu einer Prozedur und einem Baum die erste Marke liefert, für die die Prozedur *true* liefert. Dabei soll gemäß der Präfixordnung gesucht werden. Wenn eine Marke gefunden wird, soll sie mithilfe von `Found` als Ausnahme geworfen werden.

(b) Schreiben Sie eine Prozedur

$find : ('a \to bool) \to 'a\ btree \to 'a\ option$

die zu einer Prozedur und einem Baum die erste Marke liefert, für die die Prozedur *true* liefert. Wenn der Baum keine solche Marke enthält, soll `NONE` geliefert werden. Die Prozedur `find` soll intern so wie im ersten Aufgabenteil mit Ausnahmen arbeiten. Verwenden Sie dazu die lokale Deklaration

exception Found of 'a

und eine lokale Hilfsprozedur `find'`, die analog zu der obigen Prozedur `findi` arbeitet.

Aufgabe 5.7: Symbolisches Differenzieren ($26 = 2+10+8+6$) Sie sollen eine Prozedur schreiben, die arithmetische Ausdrücke nach der Variable x ableitet. Hier ist ein Beispiel:

$$(x^3 + 3x^2 + x + 2)' = 3x^2 + 6x + 1$$

Die zu betrachtenden Ausdrücke sollen gemäß des folgenden Typs dargestellt werden:

```
datatype exp = Con of int                  c
             | X                           x
             | Add of exp * exp            u + v
             | Mul of exp * exp            u · v
             | Pow of exp * int            u^n
```

(a) Schreiben Sie eine Deklaration, die den Bezeichner `exp` an eine Darstellung des Ausdrucks $x^3 + 3x^2 + x + 2$ bindet.

(b) Schreiben Sie eine Prozedur

```
derive : exp → exp
```

die die Ableitung eines Ausdrucks gemäß den folgenden Regeln berechnet:

$$\begin{aligned} c' &= 0 \\ x' &= 1 \\ (u+v)' &= u' + v' \\ (u \cdot v)' &= u' \cdot v + u \cdot v' \\ (u^n)' &= n \cdot u^{n-1} \cdot u' \end{aligned}$$

Die Ableitung darf vereinfachbare Teilausdrücke enthalten (z.B. $0 + u$ oder $0 \cdot u$).

(c) Schreiben Sie eine Prozedur

```
simplify1 : exp → exp
```

die versucht, einen Ausdruck auf oberster Ebene durch die Anwendung einer der folgenden Regeln zu vereinfachen:

$$\begin{array}{rcl@{\qquad\qquad}rcl} 0+u & \to & u & u+0 & \to & u \\ 0 \cdot u & \to & 0 & u \cdot 0 & \to & 0 \\ 1 \cdot u & \to & u & u \cdot 1 & \to & u \\ u^0 & \to & 1 & u^1 & \to & u \end{array}$$

Wenn keine der Regeln auf oberster Ebene anwendbar ist, wird der Ausdruck unverändert zurückgeliefert.

(d) Schreiben Sie eine Prozedur

```
simplify : exp → exp
```

die einen Ausdruck gemäß der obigen Regeln solange vereinfacht, bis keine Regel mehr anwendbar ist. Gehen Sie bei zusammengesetzten Ausdrücken wie folgt vor:

(i) Vereinfachen Sie zuerst die Unterausdrücke.

(ii) Vereinfachen Sie danach den zusammengesetzten Ausdruck mithilfe von `simplify1`.