6. Übungsblatt zu Programmierung

Prof. Gert Smolka, Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws00/

Abgabe: 8. Dezember 2000 in der Vorlesungspause (per Email)

Allgemeine Hinweise: Die Übungsblätter sollen in Zweiergruppen bearbeitet werden. Die Lösungen schicken Sie bitte Freitag ca 10 Uhr an Ihren Übungsgruppenleiter. Jede Gruppe soll nur eine Lösung einreichen, versehen mit den Namen und den Matrikelnummern der Gruppenmitglieder, sowie der Übungsgruppennummer.

Aufgabe 6.1: Binärbäume (1+2+2+2) Gegeben sei der Typkonstruktor `tree` für Binärbäume aus Abschnitt 6.5.

(a) Schreiben Sie eine polymorphe Prozedur *`root : 'a tree -> 'a`*, die die Marke der Wurzel eines Binärbaums liefert.

(b) Schreiben Sie eine Prozedur *`inner : 'a tree -> 'a list`*, die die Marken der inneren Knoten eines Binärbaums liefert. Dabei sollen die Marken in der Liste in derselben Reihenfolge wie bei der Präfixprojektion auftreten. Hinweis: Schreiben Sie zuerst eine Hilfsprozedur `inner'`, die die Marken aller Knoten liefert, die keine Blätter sind.

(c) Schreiben Sie eine Prozedur *`double : int tree -> int tree`*, die die Marken eines Binärbaums verdoppelt. Bespielsweise soll

```
double(N(5, L 3, L ~3)) = N(10, L 6, L ~6)
```

gelten.

(d) Schreiben Sie eine Prozedur *`mapTree : ('a -> 'b) -> 'a tree -> 'b tree`*, die eine Prozedur auf jede Marke eines Binärbaums anwendet (analog zu `map` für Listen).

Aufgabe 6.2: Symbolisches Differenzieren (2+4+4+2) Sie sollen eine Prozedur schreiben, die die Ableitung von Ausdrücken berechnet. Hier ist ein Beispiel für die Ableitung eines Ausdrucks:

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

Die zu betrachtenden Ausdrücke sollen gemäß dem folgenden Typ dargestellt werden:

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

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

(b) Schreiben Sie eine Prozedur `derive : expr -> expr`, 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 : expr -> expr`, 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 : expr -> expr`, 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 dann den zusammengesetzten Ausdruck mit den vereinfachten Unterausdrücken mithilfe von `simplify1`.

Aufgabe 6.3: Dreifach-Partition (3+3) Sie sollen Prozeduren schreiben, die eine Liste *xs* in drei Listen *us*, *vs*, *ws* zerlegen, sodass *us@vs@ws* eine Permutation von *xs* ist.

(a) Schreiben Sie mithilfe von `foldl` eine Prozedur

```
partition : ('a -> order) ->
              'a list -> 'a list * 'a list * 'a list
```

die für eine Prozedur f eine Liste wie folgt zerlegt:

(i) *us* enthält genau die Elemente von *xs*, für die f den Wert `LESS` liefert.

(ii) *ws* enthält genau die Elemente von *xs*, für die f den Wert `GREATER` liefert.

(b) Schreiben Sie mithilfe von `partition` eine Prozedur

```
myPartition : int * int ->
                int list -> int list * int list * int list
```

die für (m, n) mit $m \leq n$ eine Liste wie folgt zerlegt:

(i) *us* enthält genau die Elemente von *xs*, die echt kleiner als *m* sind.

(ii) *ws* enthält genau die Elemente von *xs*, die echt größer als *n* sind.

Aufgabe 6.4: Optionen (4) Schreiben Sie eine Prozedur

```
find : ('a -> bool) -> 'a tree -> 'a option
```

die zu einer Prozedur f und einem Binärbaum die erste Marke des Baums liefert (gemäß der Präfixordnung), für die f den Wert `true` liefert. Wenn der Baum keine solche Marke enthält, soll `NONE` geliefert werden.

Aufgabe 6.5: Schneller Test auf Doppelauftreten (0+3+3+3) 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 -> 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 6.6: Terme(2+2+2+3+3) Sie sollen Konzepte und Prozeduren, die Sie für Binärbäume kennengelernt haben, auf Terme übertragen, die gemäß der Typdeklaration

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

dargestellt werden. Benutzen Sie die Deklarationen

```
val t1 = T(1, [T(2,[]), T(7,[]), T(7,[])])
val t2 = T(3, [T(0,[t1]), T(4,[]), t1])
```

zum Testen ihrer Prozeduren.

(a) Schreiben Sie eine Prozedur

```
prefixp : 'a term -> 'a list
```

die die Präfixprojektion eines Terms liefert. Verwenden Sie dazu `map` und `List.concat`. Es soll gelten:

```
prefixp t2 = [3, 0, 1, 2, 7, 7, 4, 1, 2, 7, 7]
```

(b) Schreiben Sie eine Prozedur

```
postfixp : 'a term -> 'a list
```

die die Postfixprojektion eines Terms liefert. Es soll gelten:

```
postfixp t2 = [2, 7, 7, 1, 0, 4, 2, 7, 7, 1, 3]
```

(c) Schreiben Sie eine Prozedur

```
frontier : 'a term -> 'a list
```

die die Grenze eines Terms liefert. Es soll gelten:

```
frontier t2 = [2, 7, 7, 4, 2, 7, 7]
```

(d) Schreiben Sie eine Prozedur

```
subterm : 'a term -> int list -> 'a term
```

die zu einem Term und einem Knoten den von diesem Knoten aus erreichbaren Teilterm liefert. Verwenden Sie `List.nth`. Es soll gelten:

```
subterm t2 [1,1] = t1
```

Wenn der Knoten kein Knoten des Terms ist, soll die Ausnahme `Subscript` geworfen werden.

(e) Schreiben Sie eine Prozedur

```
balanced : 'a term -> bool
```

die testet, ob ein Term balanciert ist. Gehen Sie dabei wie beim schnellen Balanciertheitstest für Binärbäume vor (siehe Abschnitt 6.8).Verwenden Sie eine modifizierte Version der Prozedur `depth` in Abschnitt 6.7,die statt `Int.max` eine passend definierte Prozedur `forward` benutzt.