Aufgabe 5.5

Schreiben Sie eine polymorphe Prozedur

$$member : \ ''a \rightarrow ''a \ list \rightarrow bool$$

die testet, ob ein Wert als Element in einer Liste vorkommt.

Lösen Sie dies auf drei Arten:

1. durch eine regelbasierte Prozedurdeklaration (formulieren Sie zunächst passende Rekursionsgleichungen),
2. mithilfe der vordeklarierten Prozedur *List.exists*,
3. mithilfe der Prozedur *foldl*.

Aufgabe 5.9

Deklarieren Sie die Faltungsprozedur *foldr* mithilfe der Faltungsprozedur *foldl*.

Verwenden Sie dabei keine weitere rekursive Hilfsprozedur.

Tipp: Reversieren Sie die Liste zunächst (mit foldl).
Es geht aber auch ohne.

Aufgabe 5.10

Die Faltungsprozedur *foldl* kann mithilfe der Faltungsprozedur *foldr* deklariert werden, ohne dass dabei eine weitere rekursive Hilfsprozedur verwendet wird.

Das können Sie sehen, wenn Sie wie folgt vorgehen:

1. Deklarieren Sie *append* mithilfe von *foldr*.
2. Deklarieren Sie *rev* mithilfe von *foldr* und *append*.
3. Deklarieren Sie *foldl* mithilfe von *foldr* und *rev*.
4. Deklarieren Sie *foldl* nur mithilfe von *foldr*.

Aufgabe 5.11

Die Rückführung von *foldl* auf *foldr* gelingt auf besonders elegante Weise, wenn man *foldr* auf einen prozeduralen Startwert anwendet.

1. Finden sie eine Abstraktion *e*, welche uns foldl mit folgender Deklaration foldl definieren lsst:

   ```
   fun foldl f s xs = (foldr e (fn g => g) xs) s
   ```

   Die Abstraktion soll keine Hilfsprozeduren verwenden.
2. Überzeugen Sie sich davon, dass die obige Deklaration auch umgekehrt funktioniert. Wir können einfach die Bezeichner *foldl* und *foldr* vertauschen.

Ausnahmen

- Deklaration von Ausnahmen
  - `exception <Bezeichner`
  - `exception <Bezeichner> of <Typ>`
- Werfen von Ausnahmen

  `raise <Ausdruck>`
- Fangen von Ausnahmen

  `<Ausdruck> handle <Regel> | ... | <Regel>`
- Typregel

$$\frac{e : t \qquad e_1 : t \qquad \ldots e_n : t \qquad M_1 : exn \ldots M_n : exn}{e \text{ handle } M_1 \Rightarrow e_1 | \ldots | M_n \Rightarrow e_n : t}$$

Aufgabe 3.1

Schreiben Sie eine Prozedur $mymod : int \to int \to int$, die für $x \geq 0$ und $y \geq 1$ dasselbe Ergebnis wie $x \ mod \ y$ liefert.

Variation: Die Prozedur soll geeignete Ausnahmen werfen falls die Annahmen verletzt sind.

Sequentialisierung

`(<Ausdruck> ; ... ; <Ausdruck>)`

$(e_1; \ldots e_n)$ zurückgeführt auf $\#n(e_1, \ldots e_n)$

Beispiel: Test auf Überlauf:

```
fun zuGross(x,y) = (x*y; false) handle Overflow =>
true; (<Ausdruck> ; ... ; <Ausdruck>)
```

$(e_1; \ldots e_n)$ zurückgeführt auf $\#n(e_1, \ldots e_n)$

Beispiel: Test auf Überlauf:

```
fun zuGross(x,y) = (x*y; false) handle Overflow =>
true;
```

Beispiel: Test auf Überlauf

```
fun zuGross(x,y) = (x*y; false) handle Overflow =>
true;
```

Suche nach Duplikaten

```
fun pisort compare =
let
   fun insert (x,nil) = [x]
   | insert(x,y::yr) = case compare(x,y) of
     GREATER => y::insert(x,yr)
     _ => x::y::yr
in foldl insert nil

fun hatDuplikat compare xs =
let
   exception Duplicate
   fun mask compare p = compare p of
      EQUAL => raise Duplicate | v => v
in
   (pisort (mask compare) xs; false)
      handle Duplicate => true
end
```

Konstruktortypen

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

- ```
  A = fn :  exp * exp -> exp
  C = fn :  int -> exp
  M = fn :  exp * exp -> exp
  V = fn :  string -> exp
  ```

Strukturelle Rekursion

- ```
  fun subexp e = e ::  (case e of
                  A (x,y) => subexp x @ subexp y
                | M (x,y) => subexp x @ subexp y
                | _ => nil )
  ```

- ```
  fun eval env (C c) = c
    | eval env (V v) = env v
    | eval env (A(x,y)) = eval env x + eval env y
    | eval env (M(x,y)) = eval env x * eval env y
  ```