Exercise Session Informatik III

3. Going Further with SML

reduce and filter

```
fun reduce f nil a = a
  | reduce f (hd::tl) a = f hd (reduce f tl a);
val reduce = fn : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

fun filter f nil = nil
  | filter f (hd::tl) = if (f hd) then hd::filter f tl
                                  else filter f tl;
val nth = filter : ('a -> bool) -> 'a list -> 'a list
```

Application of reduce

```
fun min x a = Int.min(String.size x, a);
val min = fn : string -> int -> int

reduce min ["C", "Java", "ML"] (Option.valOf Int.maxInt);
val it = 1 : int
```

avlength

```
fun sum x y = x + y;
val sum = fn : int-> int -> int

fun avlength nil = 0.0
  | avlength ls = real(reduce sum (map String.size ls) 0)
                / real(length ls);
val avlength = fn : string list -> real
```

mapTree

```
fun mapTree f empty = empty
  | mapTree f (node(x, left, right)) =
      node(f x, (mapTree f left), (mapTree f right));
val mapTree = fn : ('a -> 'b) -> 'a tree -> 'b tree

mapTree String.size myFamilyTree;
```

redTree

```
fun redTree f a empty = a
  | redTree f a (node(x, left, right)) =
      redTree f (redTree f (f a x) left) right;
val redTree = fn : ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a
```

Important: Function **f** is called an aggregation function. Aggregation functions by definition take two values, the aggregated value so far and the next value.

```
fun join s1 s2 = s1 ^ " " ^ s2;
val join = fn : string -> string -> string

val treeToString = redTree join "my family members : ";
val treeToString = fn : string tree -> string
```

iter

```
fun iter 0 f x = x
  | iter n f x = iter (n-1) f (f x);
```

Genreral Form:	`'a -> 'b -> 'c -> 'd`
First Parameter:	`int`
Second Parameter:	`('c -> 'c)`
Third Parameter:	`'c`
Result:	`'c`
Complete Type:	`int -> ('c -> 'c) -> 'c -> 'c`
Rewritten Type:	`int -> ('a -> 'a) -> 'a -> 'a`

```
val iter = fn : int -> ('a -> 'a) -> 'a -> 'a
```

Application of iter

```
fun power4 x = iter 2 (fn x => x * x) x;
val power4 = fn : int -> int

power4 2;
val it = 16 : int
```

app app

```
fun app g f = f (f g);
```

To determine the type-string we have a look at the arguments of **app**.

- Argument **g** is a value. Its type cannot be inferred, so we assume **'a**.
- Argument **f** is a function. Function **f** takes the type of **g** as input type. As **f** is applied to itself, input and output have to be of the same type. Hence, the type of **f** is **('a -> 'a)**.
- The result type of **app** has to be the same as the result type of **f** which we found to be **'a**.

The whole type string of app therefore is **'a -> ('a -> 'a) -> 'a**.

```
val app = fn : 'a -> ('a -> 'a) -> 'a
```

app app

Again we apply our standard technique to find the type of an SML expression, which is to assume arguments and apply the definitions of the occurring functions.

app app x = x(x app) to evaluate the outermost **app**, we assume **x**

From this we can derive the general of the type-string $\tau(x) \rightarrow \tau(x(x\ app)$

Since **x** is the only unknown type in the above type string we try to find out more about **x**.

- **x** is a function, that is applied to itself, hence its type is **('b -> 'b)**.
- the input type of **x** is the type of **app**. The type of **x** therefore is **('a -> ('a -> 'a) -> 'a) -> ('a -> ('a -> 'a) -> 'a)**.

app app

Putting everything together we find the complete type-string to be

```
(('a -> ('a -> 'a) -> 'a) -> ('a -> ('a -> 'a) -> 'a)) ->
('a -> ('a -> 'a) -> 'a)
```

Binary Trees

```
fun insert(x:int,empty) = node(x,empty,empty)
  | insert(x:int,node(y,left,right)) =
              if x > y then node(y,left,insert(x,right))
                       else node(y,insert(x,left),right);
val insert = fn : int * int btree -> int btree

fun lookup(x:int,empty) = false
  | lookup(x:int,node(y,left,right))=
              if x = y then true
                       else if x > y then lookup(x,right)
                                     else lookup(x,left);
val lookup = fn : int * int btree -> bool
```

Binary Trees

```
fun merge(empty,T) = T
  | merge(T,empty) = T
  | merge(node(x,left,right),T) =
      node(x,left,merge(right,T));
val merge = fn : 'a btree * 'a btree -> 'a btree

fun delete(x,empty) = empty
  | delete(x,node(y,left,right)) =
      if x = y then merge(left,right)
        else if x < y then node(y,delete(x,left),right)
                      else node(y,left,delete(x,right));
val delete = fn : int * int btree -> int btree
```

globis

That's all folks!

Aufgaben für nächste Woche:

- Beide Versionen vom Interpreter organisieren
- Quelltext durchlesen und verstehen
- Entscheiden welche Version verständlicher ist
- Eine Version zur Bearbeitung der Übung auswählen

globis