1 Vollkommen in ML

```
fun dividedBy n m = (n mod m = 0);

fun sumDivisorsInner n 0 = 0
 |  sumDivisorsInner n m = ( if (n mod m = 0) then m else 0 ) + (
    sumDivisorsInner n (m-1));

 fun sumDivisors 0 = 0
 |  sumDivisors 1 = 0
 |  sumDivisors n = sumDivisorsInner n (n-1);

fun isPerfectNumber n = ( sumDivisors(n) = n);

isPerfectNumber (4);
isPerfectNumber (6);
isPerfectNumber (26);
isPerfectNumber (28);
isPerfectNumber (496);
isPerfectNumber (8128);
isPerfectNumber (33550336);
```

- Löst die Aufgabe 4 des Übungsblattes 2
- Wir bekommen eine “Out of memory exception” für das Überprüfen der Vollkommenheit von 33550336.
- Der Grund ist eine immer länger werdende Kette von Summanden der Form

$$0 + 0 + \text{Teiler1} + 0 + 0 + \cdots + \text{Teiler2} + \cdots$$

  in `sumDivisorsInner`, die das Programm auf Grund des Grundprinzips der *lazy evaluation* speichert bis alle Teiler durchiteriert sind.
- Dies führt zu Speicherproblemen.

2 Vollkommen in ML mit lazyness

```
datatype 'a seq = Nil
                | Cons of ('a  * (unit -> 'a seq))

fun filterq p Nil = Nil
  | filterq p (Cons(x, xf)) =
          if p x
                then Cons(x, fn () => filterq p (xf()))
                else filterq p (xf());

fun till 0 = Nil
 |  till k = Cons(k, fn () => till(k-1));

fun dividedBy n m = (n mod m = 0);

fun sumList Nil = 0
 |  sumList (Cons(x,xf)) = x + sumList (xf());

fun listDivisors 1 = Nil
 |  listDivisors 0 = Nil
 |  listDivisors n = filterq (dividedBy n) (till (n-1));

fun isPerfectNumber n = ( (sumList (listDivisors n)) = n);

isPerfectNumber (4);
isPerfectNumber (6);
isPerfectNumber (26);
isPerfectNumber (28);
isPerfectNumber (496);
isPerfectNumber (8128);
isPerfectNumber (33550336);
```

- Hier verwenden wir den Datentyp `seq`, eine "*lazy list*".
- Anstatt alle Elemente der Folge von Teilern auf einmal im Speicher zu halten, wird hier nur der aktuelle Wert und eine Vorschrift, wie man den nächsten Wert erhält zwischengespeichert (in Form der Funktion `listDivisors n`.
- Während wir die Liste in `sumList` aufsummieren, müssen wir zwar wieder alle Teiler im Speicher halten, bis alle Elemente der Liste der Teiler (`listDivisors n`) durchgegangen wurden, aber dies sind bedeutend weniger Zahlen.

3 Primzahlsieb in FC++

```
#include "fcpp.hh"

bool hatTeiler(int zahl, int tmp,int ende_sqared) {
  return cond(tmp*tmp>ende_sqared,
    false,
    cond(zahl%tmp==0,
      true,
      hatTeiler(zahl,tmp+1,ende_sqared)
        )
    );
}

bool isPrim(int zahl) {
  return !hatTeiler(zahl,2,zahl);
}

int main() {
  print(isPrim(3));
  print(isPrim(4));
  print(isPrim(9));
  print(isPrim(27));
  print(isPrim(31));
  print(isPrim(97));
}
```