Leonhard Fellermayr
Mat.Nr. 22128130XXXX
Tutorium Mi 14-16 (Reitmeier, HS 221 Hgb.)

WS 03/04

05.02.04

Übungen zur Vorlesung **Informatik I**

Blatt 14 - Lösungsversuch (keine Abgabe)

Programmieraufgabe P-54[1]

```
let rec while_loop p f x =                          (* while (p) do { ... }     *)
  if (p x) then while_loop p f (f x) else x ;;

let rec repeat_loop p f x =                         (* repeat { ... } until (p) *)
  let fx = f x in
    if (p fx) then fx else repeat_loop p f fx ;;

while_loop  (fun x -> x < 10000) (fun y -> y+5) 5 ;; (* - : int = 10000          *)

repeat_loop (fun x -> x > 10000) (fun y -> y+5) 5 ;; (* - : int = 10005          *)
```

Schriftliche Aufgabe S-55

Sehr gute URL zum Thema Komplexität und $O()$-Notation:
`http://www.mpi-sb.mpg.de/~ziegler/LEDATutorium/nonpublic/ch02s02s03.html`
Größe der Eingabe sei jeweils die Länge der Liste, die ge`fold`ed werden soll.

```
let rec fold_right f s = function
    []     -> s
  | h :: t -> f h (fold_right f s t)
```

Linear rekursiv (da 1 rekursiver Aufruf im Funktionskörper), **nicht** endrekursiv; damit fällt Platz für die Verwaltung der Rekursion an.
Die Anzahl der rekursiven Aufrufe ist $n-1$. Es wird eine $f()$-Schachtelung mit der maximalen Schachtelungstiefe n aufgebaut, die auf dem Stack abgelegt und erst bei Terminierung der Rekursion ausgewertet wird. Mit abnehmender Listenlänge n sieht das so aus:

```
  n:     f(h)                 (* Schachtelungstiefe 1 *)
n-1:     f(f(h))
n-2:     f(f(f(h)))
...      ...
  0:     f(f(f(...f(h)...)))  (* Schachtelungstiefe n *)
```

Für den Platzbedarf zur Ablage der Schachtelung auf dem Stack kann man also $O(n)$ veranschlagen, da die Schachtelungstiefe von n abhängig ist.
Der zur Auswertung der Schachtelung am Ende benötigte Platzaufwand ist identisch mit dem Aufwand zur Auswertung von f, da zuerst das innerste f ausgewertet wird, dann das nächstinnerste, usw. Der pro Auswertung benötigte Speicher wird nach jeder Auswertung wieder freigegeben und sozusagen wiederverwendet. Also $O(1)$.
Fazit: Die *Auswertung* der $f()$-Schachtelung ist mit $O(1)$ also nicht platzintensiv, wohl aber deren *Ablage* auf dem Stack ($O(n)$). Insgesamt bewegt sich der Platzbedarf von `fold_right` im Bereich von $O(1)+O(n)=O(n)$ (nach Aufgabe S-53 c).

[1] Quellcode auch im WWW: `http://www.slacky.de/files/uni/info1/schleifen.ml`

```
let rec fold_left f s = function
    []          -> s
  | [x]         -> x
  | h :: j :: t -> fold_left f s ((f h j) :: t)
```

Endrekursiv, keine Ansammlungen von geschachtelten Ausdrücken, sondern sofortige Auswertung von `(f h j)` im jeweiligen Rekursionsschritt. Der dazu allozierte Speicherplatz von $O(1)$ wird nach der Auswertung nicht mehr benötigt und wieder freigegeben.

Nachdem in jedem Schritt genau einmal f ausgewertet wird, bleibt der Platzbedarf pro Schritt für beliebig große n konstant $O(1)$.

Fazit: Also ist - dank Speicher-*re-using* - die Platzkomplexität von `fold_left` $O(1)$. Vgl. hierzu das empirisch bestimmte Speicherallokationsverhalten der *tail recursive*-Version des Siebs (Aufgabe 56) für verschiedene Eingabegrößen (10000 vs. 50000).

Abschlussbemerkung

Nach diesem Vergleich ist selbstverständlich die in `fold_left` eingesetzte Programmiertechnik zu bevorzugen. Schließlich belegt die zu bearbeitende Liste schon genug Speicher. Bei `fold_right` käme dazu noch ein größenordnungsmäßig gleich hoher Platzbedarf für die Aufbewahrung der Schachtelung (!).

Programmieraufgabe P-56[2]

(a) Zur Zeitkomplexität

Wandelt man die Funktionen aus der Musterlösung von `sieb.ml` in endrekursive Varianten um, so bietet sich ein Akkumulator zur Speicherung der Rekursionsergebnisse an. Dann steht man schnell vor dem Problem, dass Listenelemente nicht mehr vorne, sondern hinten angehängt werden müssen, um die Reihenfolge beizubehalten. Die Nutzung des dazu notwendigen Listenkonkatenators `@` hat aber einen gewaltigen Nachteil:

Das Einfügen eines einzelnen Elements `k` via `::` am Beginn einer Liste geht **erheblich** schneller vonstatten als das Anhängen der einelementigen Liste `[k]` an das Ende der Liste via Listenkonkatenation `@`.

Dann werden die Elemente allerdings in umgekehrter Reihenfolge zurückgegeben. Diesem Umstand wird in der Funktion `genlist2` durch eine Änderung der Laufvariable (Dekrementierung statt einer Inkrementierung wie im Original- `genlist`) Rechnung getragen: die Reihenfolge stimmt wieder.

In den Funktionen `rest2` und `sieve2` wird die erstellte Liste bei Terminierung der Rekursion jeweils mittels `rev_clever` revertiert, was einen kleinen zusätzlichen Zeit- und Platzaufwand nach sich zieht.

Für kleine n ist das endrekursive Sieb etwas schneller. Hier organisiert OCAML offensichtlich intern die Rekursion effizienter. Für große n hat die Musterlösung knapp die Nase vorn, da die endrekursive Lösung `rev_clever` auf die einzelnen Listen anwenden muss.

Insgesamt ist hier aber zwischen linearer und repetitiver Rekursion kein signifikanter Laufzeitunterschied auszumachen.

(b) Zur Platzkomplexität

Der Platzbedarf der endrekursiven Variante ist erheblich geringer als der Platzbedarf der Musterlösung, da kein Verwaltungsaufwand für die Rekursion (Aufbewahrung der angehäuften unerledigten Terme) anfällt (vgl. [Kröger], S. 87ff.).

Die geringere Platzkomplexität äußert sich besonders anschaulich z. B. darin, dass sich mit `genlist2` Listen einer Länge von 1.000.000 erzeugen lassen (noch höhere Werte wurden nicht getestet), während `genlist` bereits bei einer Listenlänge von ca. 53.000 mit einem *stack overflow* die Segel streicht.

D. h. nur mit der endrekursiven Variante lassen sich z. B. die Primzahlen bis 100.000 berechnen.

[2] Quellcode auch im WWW: `http://www.slacky.de/files/uni/info1/endreksieb.ml`

(c) Fazit

Es lässt sich feststellen, dass - dank Endrekursion - die Platzeffizienz bei unveränderter Zeitkomplexität steigt.

	Eingabegröße n	linear rekursiv	endrekursiv
Laufzeit	10000	0,78s	0,65s
max. Speicherplatzverbrauch	10000	bis 10 MB	2 MB konstant
Laufzeit	50000	15,2s	17,1s
max. Speicherplatzverbrauch	50000	bis 180 MB	2 MB konstant

Der Speicherplatzverbrauch der tail-recursive Version ist sogar etwas kleiner als der Footprint des OCAML-Interpreters selbst (ca. 2,2 MB). Die linear rekursive Version aast geradezu mit Speicher.

(d) Code

```
#load "unix.cma" ;;
#use "sieb.ml" ;;

(* calc_limit gibt an, bis zu welcher Hoehe Primzahlen im Benchmark
   berechnet werden sollen *)

let calc_limit = 20000 ;;

let genlist2 =
  let rec genlist_aux acc k n =
    if k >= n then genlist_aux (k::acc) (k-1) n else acc
in genlist_aux [] ;;

let allnums2 n = genlist2 n 2 ;;

let rev_clever =
  let rec rc acc = function
    []   -> acc
  | h::t -> rc (h::acc) t
in rc[]

let rest2 =
  let rec rest_aux acc k l = match l with
    []                    -> rev_clever acc
  | h::t when h mod k = 0 -> rest_aux acc k t
  | h::t                  -> rest_aux (h::acc) k t
in rest_aux [] ;;

let sieve2 =
  let rec sieve_aux acc = function
      []   -> rev_clever acc
    | h::t -> sieve_aux (h::acc) (rest2 h t)
in sieve_aux [] ;;

let primzahlen2 n = sieve2 (allnums2 n) ;;
```