Kapitel 4

Listen

Listen sind Darstellungen von endlichen Folgen, die in der Programmierung viele Anwendungen haben. Sie erforden einige Erweiterungen der bisher eingeführten Programmiersprache und geben uns die Möglichkeit, wichtige Programmiertechniken vorzuführen. Inbesondere werden wir drei Algorithmen zum Sortieren von Listen kennenlernen.

4.1 Grundbegriffe

Sei X eine Menge. Die Menge $\mathcal{L}(X)$ der *Listen über* X ist wie folgt rekursiv definiert:

$$\mathcal{L}(X) \stackrel{\text{def}}{=} \{\langle\rangle\} \cup X \times \mathcal{L}(X)$$

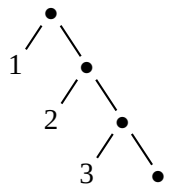
Eine Liste über X ist also entweder das leere Tupel oder ein Paar $\langle x, r \rangle$, dass aus einem $x \in X$ und einer Liste $r \in \mathcal{L}(X)$ besteht. Man kann die Menge $\mathcal{L}(X)$ auch durch zwei Inferenzregeln definieren:

$$\frac{}{\langle\rangle \in \mathcal{L}(X)} \qquad \frac{x \in X \qquad r \in \mathcal{L}(X)}{\langle x, r \rangle \in \mathcal{L}(X)}$$

Hier ist ein Beispiel für eine Liste über $\mathbb{Z}$:

$$\langle 1,\ \langle 2,\ \langle 3,\ \langle\rangle\rangle\rangle\rangle$$

Grafisch lässt sich diese Liste wie folgt darstellen:



Für die *leere Liste* $\langle\rangle$ schreiben wir auch *nil*. Für eine *nichtleere Liste* $\langle x, r\rangle$ schreiben wir auch $x :: r$ (lies x cons r). Gegeben eine nichtleere Liste $x :: r$, bezeichnet man x als den *Kopf* und r als den *Rumpf* der Liste.

Bei nach rechts geschachtelten Cons-Ausdrücken wie $x :: (y :: r)$ lassen wir die Klammern weg und schreiben $x :: y :: r$. Damit können wir die Liste $\langle 1, \langle 2, \langle 3, \langle\rangle\rangle\rangle\rangle$ lesbarer wie folgt schreiben:

$$1 :: 2 :: 3 :: 4 :: nil$$

Statt $x_1 :: x_2 :: \ldots :: x_n :: nil$ schreiben wir für $n \geq 0$ auch

$$[x_1, x_2, \ldots, x_n]$$

Beachten Sie, dass es sich bei der gerade eingeführte Notation für Listen nur um alternative Schreibweisen handelt.

Ein mathematisches Objekt x heißt *Liste* genau dann, wenn es $n \geq 0$ Objekte $x_1, \ldots, x_n$ gibt mit $x = [x_1, \ldots, x_n]$. Die Objekte $x_1, \ldots, x_n$ werden dabei als die *Elemente* der Liste bezeichnet. Für die Gleichheit von zwei Listen gilt:

$$[x_1, \ldots, x_n] = [y_1, \ldots, y_m] \iff n = m \wedge x_1 = y_1 \wedge \cdots \wedge x_n = y_n$$

Die *Länge* einer Liste $x = [x_1, \ldots, x_n]$ ist die Zahl n und wird mit $|x|$ bezeichnet. Die *Konkatenation* zweier Listen ist die wie folgt definierte Liste:

$$[x_1, \ldots, x_n] @ [y_1, \ldots, y_m] \stackrel{\text{def}}{=} [x_1, \ldots, x_n, y_1, \ldots, y_m]$$

4.2 Listen in Standard ML

Standard ML stellt für jeden Typ t einen Listentyp

```
t list
```

zur Verfügung. Die Werte dieses Typs sind alle Listen, die man über den Werten des Typs t bilden kann. Außerdem sind die im letzten Abschnitt eingeführten Notationen für Listen verfügbar:

```
[1, 2, 3]
[1, 2, 3] : int list

[1.0, 2.0, 3.0]
[1.0, 2.0, 3.0] : real list

[[2,3], [4,5,6], []]
[[2,3], [4,5,6], []] : int list list
```

```
nil
[] : 'a list

1::nil
[1] : int list

1::(2::(3::nil))
[1, 2, 3] : int list

1::2::3::nil
[1, 2, 3] : int list

1::2::3::nil = 1::[2,3]
true : bool

val xs = [3,4]
val xs = [3, 4] : int list

1::xs
[1, 3, 4] : int list

op::
fn : 'a * 'a list -> 'a list

op::(true,nil)
[true] : bool list
```

Die umgedrehte Schreibweise für Listentypen ist gewöhnungsbedürftig, eigentlich würde man statt

```
int list list
```

`list(list(int))` erwarten. Das Schlüsselwort `::` wird als *Cons-Konstruktor* bezeichnet.

Die Konstante `nil` und der Ausdruck `[]` sind polymorph, da die leere Liste ein Wert aller Listentypen ist. Die Typen von `nil` und `[]` werden durch das Schema

```
∀ 'a ('a list)
```

beschrieben. Hier sind weitere Beispiele für polymorphe Ausdrücke:

```
(1, nil, true)
(1, [], true) : int * 'a list * bool

(nil,nil)
([], []) : 'a list * 'b list
```

Das Typschema

```
∀ 'a 'b ('a list * 'b list)
```

für den Ausdruck `(nil,nil)` enthält zwei Typvariablen, da die polymorphe Konstante `nil` zweimal vorkommt. Die Instanz

```
int list * real list
```

instanziiert die Typvariablen des Schemas mit verschiedenen Typen.

Mathematisch gesehen gilt

$$\langle 1, \langle\rangle\rangle = 1 :: nil$$

Standard ML liefert aber für den Ausdruck

```
(1, ())
(1, ()) : int * unit
```

keinen Listentyp. Dagegen liefert der Ausdruck

```
1::nil
[1] : int list
```

einen Listentyp. Das liegt an der bereits erwähnten Typisierung der Konstante `nil` und an der Typregel für *Cons-Ausdrücke*:

$$\frac{a_1 : t \qquad a_2 : t\ \texttt{list}}{a_1 :: a_2 : t\ \texttt{list}}$$

Die Typdisziplin ornet Ausdrücken also nur dann einen Listentyp zu, wenn sie mit den Notationen für Listen geschrieben werden.

4.3 Length, Append, Reverse, Concat und Tabulate

Eine Funktion *length*, die die Länge einer Liste liefert, können wir in mathematischer Notation wie folgt definieren:

$$\begin{aligned} length &\in \mathcal{L}(X) \to \mathbb{N} \\ length(nil) &= 0 \\ length(x :: xr) &= 1 + length(xr) \end{aligned}$$

Diese Definition können wir nach Standard ML übertragen:

```
fun length nil     = 0
  | length (x::xr) = 1 + length xr

val length : 'a list -> int
```

Die Prozedurdeklaration besteht wie die Funktionsdefinition aus zwei *Regeln*. Die erste Regel liefert die Länge der leeren Liste. Die zweite Regel liefert die Länge von nichtleeren Listen. Die linken Seite der zweiten Regel führt zwei Argumentvariablen `x` und `xr` ein, die in der rechten Seite der Regel gültig sind. Hier sind Beispiele für die Anwendung von `length`:

```
length [1, 4, 5, 6]
4 : int

length [true, true, false]
3 : int
```

Append

Hier ist eine Prozedur, die die Konkatenation zweier Listen berechnet:

```
fun append(nil,   ys) = ys
  | append(x::xr, ys) = x::append(xr,ys)

val append : 'a list * 'a list -> 'a list

append([2,3], [6,7,8])
[2, 3, 6, 7, 8] : int list

append([2.0,3.0], nil)
[2.0, 3.0] : real list
```

Da Listenkonkatenation oft benötigt wird, ist diese Operation auch über das Operationssymbols @ verfügbar:

```
[1,2,3] @ [3,5,6]
[1, 2, 3, 3, 5, 6] : int list

[1,2,3] @ [3,5,6] @ [2,6,9]
[1, 2, 3, 3, 5, 6, 2, 6, 9] : int list

op@
fn : 'a list * 'a list -> 'a list
```

Reverse

Die Prozedur

```
fun reverse nil     = nil
  | reverse (x::xr) = reverse xr @ [x]

val reverse : 'a list -> 'a list
```

berechnet zu einer Liste die Liste, die durch Umkehrung der Elementreihenfolge erhalten wird:

```
reverse [1, 2, 3, 4]
[4, 3, 2, 1] : int list
```

Man sagt, dass `reverse` Listen *reversiert*.

Concat

Die Prozedur

```
fun concat nil     = nil
  | concat (x::xr) = x @ concat xr

val concat : 'a list list -> 'a list
```

erwartet eine Listen von Listen und liefert die Konkatentation der Elementlisten:

```
concat [[2,3], [4,5,6], [], [7]]
[2, 3, 4, 5, 6, 7] : int list
```

Tabulate

Die Prozedur

```
fun tabulate(n,f) = if n<1 then nil
                    else tabulate(n-1, f) @ [f(n-1)]

val tabulate : int * (int -> 'a) -> 'a list
```

berechnet für n und f eine Liste der Länge n wie folgt:

$$\texttt{tabulate}\,(n,\,f)\;=\;[f(0),\ldots,f(n-1)]$$

Hier sind Beispiele:

```
tabulate(5, fn x => x)
[0, 1, 2, 3, 4] : int list
```

```
tabulate(5, fn x => x*x)
[0, 1, 4, 9, 16] : int list
```

Vordeklarierte Prozeduren

Damit man die grundlegenden Prozeduren für Listen nicht jedesmal neu deklarieren muß, sind diese beim Start des Interpreters bereits wie folgt vordeklariert:

```
length        : 'a list -> int
rev           : 'a list -> 'a list                (reverse)
List.concat   : 'a list list -> 'a list
List.tabulate : int * (int -> 'a) -> 'a list
```

4.4 Map, Filter, Exists und All

In diesem Abschnitt beschreiben wir die vordeklarierten Prozeduren

```
map         : ('a -> 'b)   -> 'a list -> 'b list
List.filter : ('a -> bool) -> 'a list -> 'a list
List.exists : ('a -> bool) -> 'a list -> bool
List.all    : ('a -> bool) -> 'a list -> bool
```

Diese Prozeduren werden auf Prozeduren angewendet und liefern Prozeduren, die auf Listen angewendet werden können.

Map

Die Idee von `map` ist durch die Gleichung

$$\texttt{map}\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

gegeben. Also hat `map` die Typen

```
map : ('a -> 'b) -> 'a list -> 'b list
```

und berechnet zu einer Prozedur

```
f : 'a -> 'b
```

eine Prozedur

```
'a list -> 'b list
```

die zu einer Liste

$$[x_1, \cdots, x_n]$$

die Liste

$$[f\ x_1, \ldots, f\ x_n]$$

liefert. Hier sind Beispiele:

```
map (fn x => 2*x) [2, 4, 11, 34]
[4, 8, 22, 68] : int list

map op~ [2, 4, 11, 34]
[~2, ~4, ~11, ~34] : int list

val minus5 = map (fn x => x-5)
val minus5 : int list -> int list

minus5 [3, 6, 99, 72]
[~2, 1, 94, 67] : int list

map rev [[3, 4, 5], [2, 3], [7, 8, 9]]
[[5, 4, 3], [3, 2], [9, 8, 7]] : int list list
```

Die Prozedur `map` kann wie folgt deklariert werden:

```
fun map f nil     = nil
  | map f (x::xr) = (f x) :: (map f xr)

map : ('a -> 'b) -> ('a list -> 'b list)
```

Ambige Deklarationen

Wenn man mit `map` polymorphe Prozeduren deklariert, bekommt man manchmal ambige Deklarationen:

```
val maplen = map length
val maplen : 'a list list -> int list
! Warning: Value polymorphism, free type variable 'a
```

Dieses Problem kann man durch Einfügen einer eigentlich unnötigen Abstraktion lösen:

```
val maplen = fn xs => map length xs
val maplen : 'a list list -> int list
```

Die Abstraktion sorgt dafür, dass die rechte Seite der Deklaration kein expansiver Ausdruck ist. Etwas kürzer geht es mit einer Prozedurdeklaration:

```
fun maplen xs = map length xs
val maplen : 'a list list -> int list
```

Filter

Die vordeklarierte Prozedur

```
List.filter : ('a -> bool) -> 'a list -> 'a list)
```

liefert zu einer Prozedur f eine Prozedur, die aus einer Liste alle Elemente x herausfiltert, für die f den Wert `true` liefert:

```
List.filter (fn x => x<0) [0, ~1, 2, ~3, ~4]
[~1, ~3, ~4] : int list

List.filter (fn x => x>=0) [0, ~1, 2, ~3, ~4]
[0, 2] : int list

List.filter (fn x => x<0) [1, 2, 3]
[] : int list
```

Eine entsprechende Prozedur kann wie folgt deklariert werden:

```
fun filter f nil     = nil
  | filter f (x::xr) = if f x then x :: filter f xr
                       else filter f xr

val filter : ('a -> bool) -> 'a list -> 'a list
```

Exists und Orelse

Die vordeklarierte Prozedur

```
List.exists : ('a -> bool) -> ('a list -> bool)
```

liefert zu einer Prozedur f eine Prozedur, die für eine Liste testet, ob f für mindestens ein Element der Liste `true` liefert. Hier sind Beispiele:

```
List.exists (fn x => x<0) [1, 2, 3]
false : bool

List.exists (fn x => x<0) [1, ~2, 3]
true : bool
```

Eine entsprechende Prozedur kann wie folgt deklariert werden:

```
fun exists f nil     = false
  | exists f (x::xr) = if f x then true else exists f xr

val exists : ('a -> bool) -> ('a list -> bool)
```

Da Standard ML die Kurzform

a_1 `orelse` a_2

für Konditionale der Form

`if` a_1 `then true else` a_2

bereitstellt, kann die Deklaration von `exists` kürzer geschrieben werden:

```
fun exists f nil     = false
  | exists f (x::xr) = f x orelse exists f xr
```

All und Andalso

Die vordeklarierte Prozedur

```
val List.all : ('a -> bool) -> ('a list -> bool)
```

liefert zu einer Prozedur f eine Prozedur, die für eine Liste testet, ob f für jedes Element der Liste `true` liefert. Hier sind Beispiele:

```
List.all (fn x => x>0) [1, 2, 3]
true : bool

List.all (fn x => x>0) [1, ~2, 3]
false : bool
```

Eine entsprechende Prozedur kann wie folgt deklariert werden:

```
fun all f nil     = true
  | all f (x::xr) = if f x then all f xr  else false
```

Da Standard ML

a_1 `andalso` a_2

als Abkürzung für

```
if a1 then a2 else false
```

bereitstellt, geht es auch kürzer:

```
fun all f nil     = true
  | all f (x::xr) = f x andalso all f xr
```

4.5 Faltungsprozeduren

Mithilfe der sogenannten Faltungsprozeduren können viele Listenprozeduren ohne Rekursion definiert werden. Die Faltungsprozeduren `foldr` und `foldl` sind durch die Gleichungen

$$\texttt{foldr}\ f\ s\ [x_1, \ldots, x_n] \ = \ f(x_1,\ \ldots\ f(x_{n-1},\ f(x_n, s))\ \ldots)$$

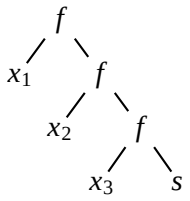
$$\texttt{foldl}\ f\ s\ [x_1, \ldots, x_n] \ = \ f(x_n,\ \ldots\ f(x_2,\ f(x_1, s))\ \ldots)$$

charakterisiert und haben die Typen

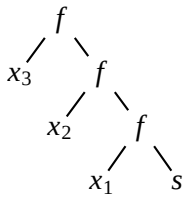
```
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Die Arbeitsweise der Faltungsprozeduren wird klarer, wenn wir den Spezialfall $n = 3$ betrachten und die rechten Seiten der Gleichungen grafisch darstellen:

`foldr` $f\ s\ [x_1, x_2, x_3] =$



`foldl` $f\ s\ [x_1, x_2, x_3] =$



Der Unterschied zwischen `foldr` und `foldl` besteht also lediglich in der Reihenfolge mit der die Listenelemente verarbeitet werden: `foldr` beginnt rechts und `foldl` beginnt links.

Eine Prozedur, die die Elemente einer Liste addiert, können wir mit `foldl` wie folgt schreiben:

```
val plus = foldl op+ 0
val plus : int list -> int

plus [4,3,6,2,8]
23 : int
```

Die folgenden Deklarationen zeigen, dass die meisten der bereits eingeführten Listenprozeduren mithilfe der Faltungsprozeduren ohne Rekursion definiert werden können:

```
fun length xs     = foldl (fn (x,n) => n+1) 0 xs

fun append(xs,ys) = foldr op:: ys xs

fun rev xs        = foldl op:: nil xs

fun map f         = foldr (fn (x,yr) => (f x)::yr) nil

fun concat xs     = foldr op@ nil xs

fun exists f      = foldl (fn (x,b) => b orelse f x) false

fun all f         = foldl (fn (x,b) => b andalso f x) true
```

Da die Gleichung

$$\texttt{foldr}\ f\ s\ xs\ =\ \texttt{foldl}\ f\ s\ (\texttt{rev}\ xs)$$

gilt, können wir sogar `foldr` auf `foldl` zurückführen:

```
fun foldr f s xs = foldl f s (foldl op:: nil xs)
```

Die Faltungsprozeduren können wie folgt deklariert werden:

```
fun foldl f s nil     = s
  | foldl f s (x::xr) = foldl f (f(x,s)) xr

fun foldr f s nil     = s
  | foldr f s (x::xr) = f(x, foldr f s xr)
```

4.6 Hd, Tl und Null

Die vordeklarierten Prozeduren

```
hd : 'a list -> 'a

tl : 'a list -> 'a list
```

liefern den Kopf (englisch „head“) und den Rumpf (englisch „tail“) einer nichtleeren Liste:

```
hd [1,2,3,4,5]
1 : int
```

```
tl [1,2,3,4,5]
[2, 3, 4, 5] : int list
```

Wenn sie auf eine leere Liste angewendet werden, werfen sie die Ausnahme `Empty`:

```
hd nil
! Uncaught exception: Empty

tl nil
! Uncaught exception: Empty
```

Die Prozeduren `hd` und `tl` können wie folgt deklariert werden:

```
fun hd nil     = raise Empty
  | hd (x::xr) = x

val hd : 'a list -> 'a

fun tl nil     = raise Empty
  | tl (x::xr) = xr

val tl : 'a list -> 'a list
```

Der Ausdruck

```
raise Empty
```

wirft beim Auswerten die Ausnahme `Empty`. Er ist polymorph und kann jeden Typ annehmen. Wenn die Auswertung eines Teilausdrucks eine Ausnahme wirft, bricht der Interpreter die laufende Auswertung ab und liefert eine entsprechende Fehlermeldung:[1]

```
[1,2] @ tl nil
! Uncaught exception: Empty
```

Die vordeklarierte Prozedur

```
fun null nil     = true
  | null (x::xr) = false

val null : 'a list -> bool
```

testet, ob eine Liste leer ist.

Mithilfe der Prozeduren `hd`, `tl` und `null` kann man jede Listenprozedur ohne Regeln schreiben. Wir zeigen das am Beispiel von `append`:

[1] Später werden wir ein Sprachkonstrukt kennenlernen, mit dem geworfene Ausnahme gefangen werden können.

```
fun append(xs,ys) = if null xs then ys
                    else hd xs :: append(tl xs, ys)

val append : 'a list * 'a list -> 'a list
```

Auf den ersten Blick scheint es, dass man statt

```
null xs
```

auch

```
xs = nil
```

schreiben kann. Im Allgemeinen ist dies nicht der Fall, wie die Deklaration

```
fun empty xs = xs=nil

val empty : ''a list -> bool
```

zeigt. Die Prozedur `empty` kann im Gegensatz zu `null` nur auf Listen angewendet werden, die über einem Typ mit Gleichheit gebildet sind. Das restriktive Typschema für `empty` kommt dadurch zustande, dass das Operationssymbol `=` mit dem Schema

```
∀ ''a (''a * ''a -> bool)
```

getypt ist (siehe Abschnitt 3.7).

Betrachten Sie die (eigentlich unsinnige) Deklaration

```
val x = hd nil

! Uncaught exception: Empty
```

Die Deklaration ist zulässig. Sie ist ambig, da die rechte Seite polymorph und expansiv ist. (Dem Ausdruck `hd nil` kann jeder Typ zugeordnet werden.) Die Auswertung der rechten Seite wird mit der Ausnahme `Empty` abgebrochen. Das bedeutet, dass der Bezeichner `x` nicht an einen Wert gebunden werden kann. Daher merkt sich der Interpreter auch keine statische Bindung für `x`.

4.7 Sortieren von ganzzahligen Listen

Eine Liste $[x_1, \ldots, x_n] \in \mathcal{L}(\mathbb{Z})$ heißt *sortiert* genau dann, wenn $x_i \leq x_{i+1}$ für alle $i \in \{1, \ldots, n-1\}$ gilt. Die Prozedur

```
fun sorted nil        = true
  | sorted [x]        = true
  | sorted (x::y::zs) = x<=y andalso sorted(y::zs)

val sorted : int list -> bool
```

testet, ob eine ganzzahlige Liste sortiert ist:

```
sorted [2, 5, 9, 33]
true : bool

sorted [2, 5, 4, 33]
false: bool
```

Man kann die Prozedur `sorted` auch kürzer deklarieren:

```
fun sorted (x::y::zs) = x<=y andalso sorted(y::zs)
  | sorted xs         = true
```

Diese Deklaration ist zu der obigen Deklaration gleichwertig, da die Regeln einer Prozedur entsprechend ihrer Reihenfolge priorisiert sind: Eine Regel wird nur dann angewendet, wenn die vor ihr stehenden Regeln nicht anwendbar sind.

Argumentvariablen, die nur in der linken Seite einer Regel vorkommen, sollte man *anonymisieren*, indem man sie durch das Zeichen $\langle\_\rangle$ (lies „wildcard“) ersetzt:

```
fun sorted (x::y::zs) = x<=y andalso sorted(y::zs)
  | sorted _          = true
```

Zwei Listen heißen *bis auf Permutation gleich* genau dann, wenn sie bis auf die Reihenfolge ihrer Elemente gleich sind. Eine Liste *xs* heißt *Permutation* einer Liste *ys* genau dann, wenn *xs* und *ys* bis auf Permutation gleich sind.

Zu jeder Liste in $\mathcal{L}(\mathbb{Z})$ existiert offensichtlich genau eine sortierte Permutation. Folglich gibt es genau eine *Sortierfunktion*

$$sort \in \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$$

die zu einer Liste die sortierte Permutation liefert. Es gilt:

$$\forall xs \in \mathcal{L}(\mathbb{Z})\ \ \forall ys \in \mathcal{L}(\mathbb{Z}):$$
$$xs \text{ und } ys \text{ sind bis auf Permutation gleich} \iff sort(xs) = sort(ys)$$

Sortieren durch Einfügen

Wir überlegen uns jetzt, wie man eine *Sortierprozedur* schreiben kann, die die Sortierfunktion berechnen. Da die leere Liste sortiert ist, brauchen wir für eine Sortierprozedur lediglich eine Gleichung, die für nichtleere Listen $x :: xr$ beschreibt, wie man aus $sort(xr)$ die sortierte Liste $sort(x :: xr)$ erhalten kann. Wir

werden mit der Gleichung

$$sort(x :: xr) \;=\; insert(x,\; sort(xr))$$

arbeiten. Diese gilt für alle Funktionen $insert \in \mathbb{Z} \times \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$, die zu x und einer sortierten Liste xs die sortierte Permutation von $x :: xs$ liefern. Es ist nicht schwer, eine entsprechende Prozedur `insert` zu schreiben. Dazu müssen wir lediglich x an der richtigen Stelle der bereits sortierten Liste xs einfügen:

```
fun insert (x, nil)   = [x]
  | insert (x, y::yr) = if x<=y then x::y::yr
                        else y::insert(x,yr)

val insert : int * int list -> int list

insert(3, [0, 1, 2, 3, 4, 5])
[0, 1, 2, 3, 3, 4, 5] : int list

insert(3, [0, 5, 2])
[0, 3, 5, 2] : int list
```

Mit `insert` können wir die folgende Sortierprozedur formulieren:

```
fun isort nil     = nil
  | isort (x::xr) = insert(x, isort xr)

val isort : int list -> int list

isort [5, 2, 2, 13, 9, 9, 13, ~2]
[~2, 2, 2, 5, 9, 9, 13, 13]
```

Den durch `isort` realisierten Sortieralgorithmus bezeichnet man als *Sortieren durch Einfügen*.

Mit `foldl` kann man `isort` eleganter schreiben:

```
val isort = foldl insert nil
val isort : int list -> int list
```

Sortieren durch Mischen

Wir entwickeln jetzt eine zweite Sortierprozedur, die einen Sortieralgorithmus realisiert, der als Sortieren durch Mischen bekannt ist. Diesmal verwenden wir die Gleichung

$$sort(xs@ys) \;=\; merge(sort(xs),\; sort(ys))$$

Diese gilt für jede Funktion $merge \in \mathcal{L}(\mathbb{Z}) \times \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$, die zu zwei sortierten Listen xs und ys die sortierte Permutation von $xs@ys$ liefert. Wir benötigen also eine Prozedur `merge`, die zwei sortierte Listen in eine sortierte Liste verschmilzt. Wir deklarieren `merge` wie folgt:

```
fun merge (nil  , ys   ) = ys
  | merge (xs   , nil  ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)

val merge : int list * int list -> int list
```

Als nächstes brauchen wir eine Prozedur

```
split : 'a list -> 'a list * 'a list
```

die eine Liste in zwei etwa gleich große Teillisten zerlegt. Präziser gesagt soll `split` zu einer Liste *xs* zwei Listen *ys* und *zs* berechnen, sodass gilt:

1. *ys@zs* ist eine Permutation von *xs*.
2. Für die Längen gilt $|ys| \leq \frac{1}{2}(|xs|+1)$ und $|zs| \leq \frac{1}{2}(|xs|+1)$.

Hier ist eine entsprechende Deklaration für `split`:

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys))
                     (nil, nil) xs

val split : 'a list -> 'a list * 'a list
```

```
split[5, 2, 2, 13, 4, 9, 9, 13, ~2]
([13, 9, 13, 2], [~2, 9, 4, 2, 5]) : int list * int list
```

Jetzt können wir die folgende Sortierprozedur schreiben:

```
fun msort nil      = nil
  | msort (x::nil) = x::nil
  | msort xs       = let
                         val (ys,zs) = split xs
                       in
                         merge(msort ys, msort zs)
                       end

val msort : int list -> int list
```

Die Rekursion terminiert, da `split` Listen mit mindestens zwei Elementen in echt kürzere Listen zerlegt. Die lokale Wertdeklaration

```
val (ys,zs) = split xs
```

in der dritten Regel bindet die Bezeichner `ys` und `zs` an die von `split xs` gelieferten Teillisten.

Funktion, Algorithmus und Prozedur

Am Beispiel des Sortierproblems für ganzzahlige Listen können wir die Begriffe Funktion, Algorithmus und Prozedur klar voneinander abgrenzen:

1. Es gibt genau eine Sortierfunktion $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$.
2. Sortieren durch Einfügen und Sortieren durch Mischen sind zwei verschiedene Algorithmen (Berechnungsverfahren) für die Berechnung der Sortierfunktion $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$. Diese Algorithmen können in verschiedenen Programmiersprachen verschieden realisiert werden.
3. Die Prozedur `isort` ist eine mögliche Realisierung des Algorithmus „Sortieren durch Einfügen“. Die Prozedur `msort` ist eine mögliche Realisierung des Algorithmus „Sortieren durch Mischen“.

In Kapitel 7 werden wir zeigen, dass die Prozedur `msort` schneller sortiert als die Prozedur `isort`.

4.8 Polymorphe Sortierprozeduren

Die Sortierprozeduren des letzten Abschnitts sortieren ganzzahlige Listen in aufsteigender Ordnung. Wir wollen jetzt eine polymorphe Sortierprozedur schreiben, mit der wir unter anderem die folgenden Aufgaben lösen können:

1. Sortieren einer Liste über `int` in aufsteigender oder absteigender Ordnung.
2. Sortieren einer Liste über `real` in aufsteigender oder absteigender Ordnung.
3. Sortieren einer Liste über `int*int` in aufsteigender oder absteigender Ordnung gemäß

$$(x_1, y_1) \leq (x_2, y_2) \iff x_1 < x_2 \ \vee \ (x_1 = x_2 \wedge y_1 \leq y_2)$$

Ein geeigneter Typ für eine solche polymorphe Sortierprozedur ist

```
('a * 'a -> bool) -> 'a list -> 'a list
```

Dabei soll das erste Argument eine Prozedur f sein, die zu einem Paar (x, y) genau dann `true` liefert, wenn x in einer sortierten Liste vor y stehen darf. Die Prozedur f legt also fest, nach welcher *Ordnung* sortiert werden soll.

Es ist einfach, die im letzten Abschnitt vorgestellten Sortierprozeduren zu polymorphen Sortierprozeduren zu verallgemeinern. Abbildung 4.1 zeigt eine polymorphe Sortierprozedur `pisort`, die durch Einfügen sortiert. Hier sind Beispiele für die Anwendung von `pisort`:

```
pisort op<=
fn : int list -> int list
```

```
fun pisort order =
    let
        fun insert (x, nil)   = [x]
          | insert (x, y::yr) = if order(x,y) then x::y::yr
                                else y::insert(x,yr)
    in
        foldl insert nil
    end

val pisort : ('a * 'a -> bool) -> 'a list -> 'a list
```

Abbildung 4.1: Polymorphes Sortieren durch Einfügen

```
pisort op<= [5, 2, 2, 13, 4, 9, 9, 13, ~2]
[~2, 2, 2, 4, 5, 9, 9, 13, 13] : int list

pisort op>= [5, 2, 2, 13, 4, 9, 9, 13, ~2]
[13, 13, 9, 9, 5, 4, 2, 2, ~2] : int list

pisort op<= [5.0, 2.0, 2.0, 13.0, 4.0, 9.0]
[2.0, 2.0, 4.0, 5.0, 9.0, 13.0] : real list

fun order((x,y), (x',y')) = x<x' orelse x=x' andalso y<=y'
val order : (int * int) * (int * int) -> bool

pisort order [(7,5), (4,8), (3,9), (7,4), (3,12)]
[(3, 9), (3, 12), (4, 8), (7, 4), (7, 5)] : (int * int) list
```

4.9 Prozeduren mit mehreren Regeln

In diesem Kapitel haben wir Prozeduren oft mithilfe von zwei oder drei Regeln definiert. Wir wollen jetzt die programmiersprachlichen Aspekte dieser Definitionsform zusammenfassend diskutieren.

Als erstes Beispiel betrachten wir

```
fun length nil     = 0
  | length (x::xr) = 1 + length xr

val length : 'a list -> int
```

Die erste Regel ist nur auf die leere Liste und die zweite Regel nur auf nichtleere Listen anwendbar. Wir sagen, dass sich die beiden Regeln *gegenseitig ausschließen*. Wenn sich die Regeln einer Prozedur gegenseitig ausschließen, spielt die Reihenfolge, in der die Regeln angegeben werden, keine Rolle. Wir können `length` also auch wie folgt deklarieren:

```
fun length (x::xr) = 1 + length xr
  | length nil     = 0
```

Die Regel für nichtleere Listen führt zwei Argumentvariablen ein. Diese sind nur in der rechten Seite der Regel gültig. Da die Argumentvariable `x` auf der rechten Seite der Regel nicht benutzt wird, sollte man sie der besseren Lesbarkeit halber mithilfe des Wildcard-Symbols *anonymisieren*:

```
fun length (_::xr) = 1 + length xr
  | length nil     = 0
```

In Kapitel 3 haben wir gelernt, dass Prozedurdeklarationen auf rekursive Wertdeklarationen und Abstraktionen zurückgeführt werden. Dies gilt auch für Prozedurdeklarationen mit mehreren Regeln:

```
val rec length =
    fn _::xr => 1 + length xr
     | nil   => 0
```

Dabei werden *Abstraktionen mit mehreren Regeln* verwendet. Normale Abstraktionen bestehen aus genau einer Regel. Die Regeln einer Abstraktion haben die Form

Muster `=>` *Ausdruck*

Die rechte Seite einer Regel heißt *Rumpf* der Regel. *Muster* sind spezielle Ausdrücke, die sich wie folgt rekursiv definieren lassen:

1. Konstanten und Bezeichner sind Muster.
2. Wenn $a_1, \ldots, a_n$ Muster sind, dann sind $(a_1, \ldots, a_n)$ und $[a_1, \ldots, a_n]$ Muster.
3. Wenn a_1 und a_2 Muster sind, dann ist $a_1 :: a_2$ ein Muster.

Zusätzlich gilt, dass ein Bezeichner in einem Muster höchstens einmal auftreten darf. Die Bezeichnerauftreten im Muster einer Regel sind alle definierend und haben den Rumpf der Regel als Gültigkeitsbereich. Die durch das Muster einer Regel eingeführten Bezeichner heißen *Argumentvariablen* der Regel.

Die Vorschrift, nach der bei einer Prozeduranwendung entschieden wird, mit welcher Regel das Argument behandelt werden soll, nennt man *pattern matching*.

Wenn sich die Regeln einer Prozedur nicht gegenseitig ausschließen, werden sie entsprechend der angegebenen Reihenfolge *priorisiert*. Eine Regel ist in diesem Fall nur dann anwendbar, wenn die vor ihr stehenden Regeln nicht anwendbar sind. Hier ist ein Beispiel:

```
fun test (_::_::_) = true
  | test _         = false

val test : 'a list -> bool
```

Diese Prozedur testet, ob eine Liste mindestens zwei Elemente hat.

Die Regeln einer Prozedur heißen *erschöpfend* (englisch „exhaustive“), wenn auf jeden Wert des Argumenttyps mindestens eine Regel anwendbar ist. Wenn die Regeln einer Prozedur nicht erschöpfend sind, gibt der Interpreter eine Warnung aus:

```
fun test nil = 0
  | test [_] = 1

! Warning: pattern matching is not exhaustive
val test : 'a list -> int
```

Wenn eine Prozedur auf ein Argument angewendet wird, auf das keine Regel anwendbar ist, wird die Ausnahme `Match` geworfen:

```
test [1,2]
! Uncaught exception: Match
```

In der Praxis sollte man auf unvollständig definierte Prozeduren verzichten und die unerwünschten Argumente mithilfe einer zusätzlichen Regel behandeln, die eine geeignete Ausnahme wirft:

```
fun test nil = 0
  | test [_] = 1
  | test _   = raise Match

val test : 'a list -> int
```

In Kapitel 6 werden wir lernen, wie man neue Ausnahmen definiert.

4.10 Laufzeit von Prozeduren

Um das Ergebnis einer Prozeduranwendung zu berechnen, benötigt der Interpretierer eine gewisse Zeit. Man spricht von der Laufzeit einer Prozedur. Oft ist die Laufzeit einer Prozedur von ihrem Argument abhängig. Wenn eine Prozedur für bestimmte Argumente eine zu hohe Laufzeit hat (zum Beispiel einige Jahre), ist sie für diese Argumente praktisch unbrauchbar. Zwei Prozeduren, die die dieselbe Funktion berechnen, können sich in ihrer Laufzeit drastisch unterscheiden. Gegeben eine zu berechnende Funktion, interessieren wir uns natürlich für Prozeduren,

die die Funktion für die praktisch relevanten Argumente mit kleiner Laufzeit berechnen.

In Kapitel 7 werden wir lernen, wie man die Laufzeit von Prozeduren abschätzen kann. Wir zeigen an zwei Beispielen, dass sich die Laufzeit von zwei Prozeduren, die dieselbe Funktion berechnen, drastisch unterscheiden kann.

Zunächst betrachten wir zwei Prozeduren, die zu einer Liste die umgedrehte Liste liefern:

```
fun reverse nil     = nil
  | reverse (x::xr) = reverse xr @ [x]

fun reverse' xs = foldl op:: nil xs
```

Wenn wir die beiden Prozeduren auf die Liste

```
val xs = List.tabulate(5000, fn x=> x)
```

der Länge 5000 anwenden, können wir beobachten, dass `reverse'` sehr viel schneller als `reverse` ist. Der Unterschied wird noch größer, wenn wir längere Listen betrachten. Für eine Liste der Länge 50000 liefert `reverse` auch nach einigen Minuten noch kein Ergebnis, wohingegen `reverse'` ihr Ergebnis sofort liefert.

Als zweites Beispiel vergleichen wir die Sortierprozeduren `isort` (die Version ohne `foldl`) und `msort` aus Abschnitt 4.7 für die Liste

```
val xs = rev(List.tabulate(5000, fn x=> x))
```

Während `msort` diese Liste sofort sortiert, braucht `isort` einige Sekunden. Für die Liste

```
val xs = List.tabulate(5000, fn x=> x)
```

sind `isort` und `msort` dagegen ungefähr gleich schnell.

Die Erklärungen dieser überraschenden Beobachtungen finden Sie in Abschnitt 7.14.

Übungsaufgaben

Aufgabe 4.1 (Last) Schreiben Sie eine Prozedur

```
last : 'a list -> 'a
```

die das letzte Element einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme `Empty` geworfen werden.

Aufgabe 4.2 (Enum) Schreiben Sie eine Prozedur

```
enum : int * int -> int list
```

die für zwei Zahlen $m \leq n$ die Liste $[m, m+1, \ldots, n]$ liefert. Für $m > n$ soll `enum` die leere Liste liefern.

Aufgabe 4.3 (Nth, Take, Drop) Schreiben Sie drei Prozeduren

```
nth  : 'a list * int -> 'a
take : 'a list * int -> 'a list
drop : 'a list * int -> 'a list
```

wie folgt:

- `nth(`*xs*`,`*n*`)` liefert das *n*-te Element der Liste *xs*. Dabei sollen die Elemente von *xs* mit 0 beginnend nummeriert sein. Falls $n < 0$ oder $\mathit{length}(xs) \leq n$ gilt, soll die Ausnahme `Subscript` geworfen werden.
- `take(`*xs*`,`*n*`)` liefert die ersten *n* Elemente der Liste *xs*. Falls $n < 0$ oder $\mathit{length}(xs) < n$ gilt, soll die Ausnahme `Subscript` geworfen werden.
- `drop(`*xs*`,`*n*`)` liefert die Liste, die man aus *xs* erhält, wenn man die ersten *n* Elemente weglässt. Falls $n < 0$ oder $\mathit{length}(xs) < n$ gilt, soll die Ausnahme `Subscript` geworfen werden.

Hinweis: Verwenden Sie `orelse` und die Prozeduren `hd`, `tl` und `null`.

Aufgabe 4.4 (Max) Schreiben Sie mit `foldl` eine Prozedur

```
max : int list -> int
```

die das größte Element einer Liste liefert. Wenn die Liste leer ist, soll die Ausnahme `Empty` geworfen werden.

Aufgabe 4.5 (Member) Sie sollen drei Varianten einer Prozedur

```
member : ''a -> ''a list -> bool
```

schreiben, die testet ob ein Wert in einer Liste vorkommt.

1. Die erste Variante soll keine andere Prozedur benutzen.
2. Die zweite Variante soll mit `List.exists` und ohne Rekursion geschrieben werden.

3. Die dritte Variante soll mit `foldl` und ohne Rekursion geschrieben werden.

Aufgabe 4.6 (Count) Schreiben Sie mit `foldl` eine Prozedur

```
count : ''a -> ''a list -> int
```

die für einen Wert die Anzahl seiner Auftreten in einer Liste berechnet.

Aufgabe 4.7 (Dezimaldarstellung) Die Dezimaldarstellung einer natürlichen Zahl ist die Liste ihrer Ziffern. Beispielsweise hat 7856 die Dezimaldarstellung [7, 8, 5, 6].

1. Schreiben Sie eine Prozedur

   ```
   dec : int -> int list
   ```

   die die Dezimaldarstellung einer natürlichen Zahl berechnet. Verwenden Sie `div` und `mod`.

2. Schreiben Sie mithilfe von `foldl` eine Prozedur

   ```
   int : int list -> int
   ```

   die zu einer Dezimaldarstellung die dargestellte natürliche Zahl berechnet.

Aufgabe 4.8 (Permutationen) Sei eine Sortierprozedur

```
sort : int list -> int list
```

gegeben. Schreiben Sie damit eine Prozedur

```
perm : int list * int list -> bool
```

die testet, ob zwei Listen bis auf Permutation gleich sind.

Aufgabe 4.9 (Partition) Schreiben Sie zwei Varianten einer Prozedur

```
partition : int -> int list -> int list * int list
```

die zu einer Zahl *x* und einer Liste *xs* zwei Listen *us* und *vs* wie folgt berechnet:

1. *us@vs* ist eine Permutation von *xs*.
2. Alle Elemente von *us* sind echt kleiner als *x* und alle Elemente von *vs* sind größer gleich als *x*.

Die erste Variante soll keine andere Prozedur benutzen. Die zweite Variante soll mit `foldl` und ohne Rekursion geschrieben werden.

Aufgabe 4.10 (Quicksort) Quicksort ist ein klassischer Sortieralgorithmus, der auf der folgenden Beobachtung beruht.

Seien $x \in \mathbb{Z}$ und $xs, us, vs \in \mathcal{L}(\mathbb{Z})$ wie folgt:

1. $us@vs$ ist eine Permutation von xs.
2. Alle Elemente von us sind echt kleiner als x und alle Elemente von vs sind größer gleich als x.

Weiter sei *sort* die Sortierfunktion $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$. Dann gilt die Gleichung

$$sort(x :: xs) \;=\; sort(us) \;@\; [x] \;@\; sort(vs)$$

Schreiben Sie mithilfe dieser Gleichung und der Prozedur `partition` aus Aufgabe 4.9 eine Sortierprozedur `qsort : int list -> int list`.

Aufgabe 4.11 (Polymorphe Sortierprozedur) Schreiben Sie eine polymorphe Sortierprozedur

```
pmsort : ('a * 'a -> bool) -> 'a list -> 'a list
```

die Listen durch Mischen sortiert.

Aufgabe 4.12 (Primzahlen) Eine natürliche Zahl $n \geq 2$ heißt Primzahl, wenn sie nur durch 1 und sich selbst teilbar ist.

1. Schreiben Sie mithilfe von `List.exists` eine Prozedur

   ```
   divisible: int * int list -> bool
   ```

   die testet, ob eine positive Zahl durch mindestens ein Element einer Liste positiver Zahlen teilbar ist.

2. Schreiben Sie eine Prozedur

   ```
   nextPrime: int * int list -> int
   ```

   die zu $x > 1$ und zu einer Liste *ps* von Primzahlen die kleinste Primzahl $p \geq x$ liefert, vorausgesetzt, *ps* enthält alle Primzahlen, die kleiner als x sind.

3. Schreiben Sie eine Prozedur

   ```
   primes: int -> int list
   ```

   die zu $n \in \mathbb{N}$ die Liste der ersten n Primzahlen berechnet (die Reihenfolge ist beliebig).