Kapitel 4

Listen

In diesem Kapitel lernen Sie, wie man mit Listen programmiert. Listen stellen endliche Folgen von Objekten dar. Hier sind einige typische Programmieraufgaben, die Sie mit Listen lösen können:

- Schreibe eine Prozedur, die zu $n \in \mathbb{N}$ die Liste der ersten n Primzahlen liefert. Beispielsweise soll für 5 die Liste $[2, 3, 5, 7, 11]$ geliefert werden.
- Schreibe eine Prozedur, die die Dezimaldarstellung einer Zahl liefert. Beispielsweise soll für 84711 die Liste $[8, 4, 7, 1, 1]$ geliefert werden.
- Schreibe eine Prozedur, die zu einer Liste die Liste liefert, die die Elemente der Liste in umgekehrter Reihenfolge enthält. Beispielsweise soll zu $[4, 7, 1, 1, 3]$ die Liste $[3, 1, 1, 7, 4]$ geliefert werden.
- Schreibe eine Prozedur, die zu einer Liste die Liste liefert, die die Elemente der Liste in sortierter Reihenfolge enthält. Beispielsweise soll zu $[4, 7, 1, 1]$ die Liste $[1, 1, 4, 7]$ geliefert werden.

Im Zusammenhang mit Listen lernen Sie in diesem Kapitel auch zwei neue programmiersprachliche Konzepte kennen: Regelbasierte Prozeduren und das Werfen von Ausnahmen.

4.1 Listen als mathematische Objekte

Endliche Folgen können durch Tupel $\langle x_1, \ldots, x_n \rangle$ dargestellt werden. Zum Programmieren besser geeignet ist allerdings die so genannte **Listendarstellung**, die nichtleere Folgen mithilfe von Paaren darstellt:

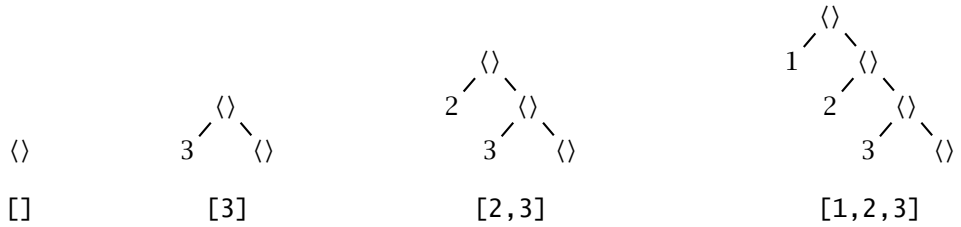
1. Die Listendarstellung der leeren Folge ist das leere Tupel.

2. Die Listendarstellung einer nichtleeren Folge $x_1, \ldots, x_n$ ist das Paar, das aus x_1 und der Listendarstellung der Folge $x_2, \ldots, x_n$ besteht.

Die Listendarstellung einer Folge $x_1, \ldots, x_n$ bezeichnen wir mit $[x_1, \ldots, x_n]$. Hier sind Beispiele:

$$\begin{aligned} [] &= \langle\rangle \\ [3] &= \langle 3, \langle\rangle\rangle \\ [2,3] &= \langle 2, \langle 3, \langle\rangle\rangle\rangle \\ [1,2,3] &= \langle 1, \langle 2, \langle 3, \langle\rangle\rangle\rangle\rangle \end{aligned}$$

Ein Objekt x heißt **Liste**, wenn es Objekte $x_1, \ldots, x_n$ mit $x = [x_1, \ldots, x_n]$ gibt. Die eindeutig bestimmten Objekte $x_1, \ldots, x_n$ werden als **Elemente** der Liste bezeichnet. Es ist hilfreich, sich die Baumdarstellung von Listen zu vergegenwärtigen:



Gegeben eine nichtleere Liste $\langle x, xs\rangle$, bezeichnen wir x als den **Kopf** und xs als den **Rumpf** der Liste.

Listen eignen sich gut zu Programmieren, da sie Schritt für Schritt durch Hinzufügen von Elementen konstruiert werden können:

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

Umgekehrt können Listen durch Zerlegen in Kopf und Rumpf schrittweise verarbeitet werden

Sei X eine Menge. Eine Liste heißt **Liste über** X, wenn alle ihre Elemente Elemente der Menge X sind. Die Listen über X lassen sich mit zwei Regeln konstruieren:

1. Das leere Tupel ist eine Liste über X.
2. Wenn $x \in X$ und xs eine Liste über X ist, dann ist $\langle x, xs\rangle$ eine Liste über X.

Die Menge aller Listen über X bezeichnen wir mit $\mathcal{L}(X)$. Die Konstruktionsregeln für $\mathcal{L}(X)$ können wir wie folgt formulieren:

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

Sei $[x_1, \ldots, x_n]$ eine Liste. Dann bezeichnen wir die Zahl n als die **Länge der Liste**. Die Länge einer Liste xs bezeichnen wir mit $|xs|$. Die **Konkatenation** zweier Listen ist die wie folgt definierte Liste:

$$[x_1, \ldots, x_m] @ [y_1, \ldots, y_n] \stackrel{\text{def}}{=} [x_1, \ldots, x_m, y_1, \ldots, y_n]$$

Offensichtlich gilt für zwei Listen xs und ys stets $|xs@ys| = |xs| + |ys|$.

Für die **leere Liste** $[\,]$ schreiben wir auch *nil*. Für eine **nichtleere Liste** $\langle x, xs \rangle$ schreiben wir auch $x :: xs$ (lies x cons xs). Bei nach rechts geschachtelten **Cons-Ausdrücken** wie $x :: (y :: xs)$ lassen wir die Klammern weg und schreiben $x :: y :: xs$. Damit können wir die Liste $[1, 2, 3, 4]$ auch als $1 :: 2 :: 3 :: 4 :: nil$ schreiben.

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 über den Werten des Typs t gebildet werden können. Listen können durch Aufzählen ihrer Elemente beschrieben werden:

```
[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
```

Die umgedrehte Schreibweise für Listentypen ist gewöhnungsbedürftig, statt `int list list` würde man eigentlich `list(list(int))` erwarten. Die Konstante für die leere Liste ist polymorph:

```
[]
[] : 'a list
```

Listen können auch mit der Konstante `nil` und dem **Cons-Konstruktor** `::` beschrieben werden:

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

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

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

Im Gegensatz zu der Klammersparregel für arithmetische Operatoren (siehe Abschnitt 1.10), gruppiert die Klammersparregel für den Cons-Konstruktor nach rechts:

$$e_1 :: e_2 :: e_3 \quad \rightsquigarrow \quad e_1 :: (e_2 :: e_3)$$

Hier sind weitere Beispiele:

```
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
```

Mathematisch gesehen gilt

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

Diese Gleichung ist in Standard ML nicht zulässig, da der linken und rechten Seite verschiedene Typen zugeordnet werden:

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

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

4.3 Length, Append, Reverse, Concat und Tabulate

Wir zeigen jetzt, wie man mit Listen programmiert. Wir beginnen mit einer polymorphen Prozedur

```
length : 'a list → int
```

die die Länge einer Liste liefert:

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

Die Deklaration der Prozedur besteht aus zwei **Regeln**, die durch das Schlüsselwort „|“ getrennt sind. Die erste Regel liefert die Länge der leeren Liste. Die zweite Regel liefert die Länge von nichtleeren Listen. Je nachdem, auf was für ein Argument die Prozedur angewendet wird, wird die richtige Regel gewählt. Das **Muster** `(x::xr)` der zweiten Regel führt zwei **Variablen** `x` und `xr` ein, die im **Rumpf** der zweiten Regel benutzt werden können. Hier sind Beispiele für die Anwendung von `length`:

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

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

Prozeduren, die so wie `length` mit mehreren Regeln beschrieben sind, bezeichnen wir als **regelbasiert**. Normale Prozeduren kann man als Prozeduren auffassen, die mit nur einer Regel beschrieben sind.

Schauen Sie sich nochmal die zweite Regel der Prozedur `length` an:

```
length (x::xr) = 1 + length xr
```

Da die Variable `x` im Rumpf der Regel nicht benutzt wird, können wir ihr Auftreten im Muster der Regel durch das **Wildcard-Symbol** ersetzen:

```
length (_::xr) = 1 + length xr
```

Die Verwendung des Wildcard-Symbols im Muster der Regel hat den stilistischen Vorteil, dass sofort klar ist, dass der Kopf der Liste bei der Anwendung der Regel keine Rolle spielt.

Append

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

```
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
```

Da Listenkonkatenation oft benötigt wird, ist diese Operation auch über den Operator `@` 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
```

Die Klammersparregel für den Konkatenationsoperator `@` gruppiert so wie die Klammersparregel für den Cons-Konstruktor nach rechts:

$$e_1 @ e_2 @ e_3 \quad \rightsquigarrow \quad e_1 @ (e_2 @ e_3)$$

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 Prozedur `map` liefert zu einer Prozedur f und einer Liste die Liste, die man durch Anwenden von f auf alle Elemente erhält:

`map` $f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$

Sie ist polymorph getypt

```
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 oft ambige Deklarationen (siehe Abschnitt 3.4):

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

Das liegt daran, dass die Applikation `map length` expansiv ist. Dieses Problem lässt sich durch Einfügen einer Abstraktion leicht 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. 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 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
```

Da Standard ML die Kurzform

e_1 `orelse` e_2

für Konditionale der Form

`if` e_1 `then true else` e_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

e_1 `andalso` e_2

als Abkürzung für

`if` e_1 `then` e_2 `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

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

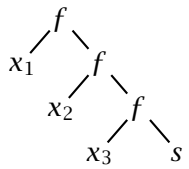
$$\texttt{foldl}\ f\ s\ [x_1,\ldots,x_n] = f(x_n, \ldots f(x_2, f(x_1, s)) \ldots)$$

charakterisiert und haben die Typen

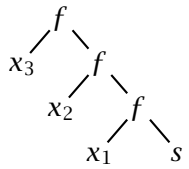
```
('a * 'b → 'b) → 'b → 'a list → 'b
```

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
```

Viele der bereits eingeführten Listenprozeduren können 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

`foldr` $f\ s\ xs\ =$ `foldl` $f\ s$ (`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, Null und Ausnahmen

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 bei der Ausführung eines Programms eine Ausnahme geworfen wird, bricht der Interpreter die Ausführung des Programms 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`:

```
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. Das ist aber keineswegs der Fall, da `xs=nil` im Gegensatz zu `null xs` nur dann zulässig ist, wenn der Elementtyp von `xs` ein Typ mit Gleichheit ist. Das liegt daran, dass der Operator = mit dem Schema

```
∀ ''a (''a * ''a → bool)
```

getypt ist (siehe Abschnitt 3.7). Also liefert die Deklaration

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

den entsprechend eingeschränkten Typ

```
append : ''a list * ''a list → ''a list
```

4.7 Regelbasierte Prozeduren

Eine Prozedur heißt **regelbasiert**, wenn sie mit mindestens zwei Regeln definiert ist. In diesem Kapitel haben wir bereits einige Beispiele für regelbasierte Prozedu-

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

ren gesehen. Wir wollen jetzt genauer auf die programmiersprachlichen Aspekte dieser Definitionsform eingehen.

Allgemein haben die **Regeln** einer regelbasierten Prozedur die Form

⟨*Prozedurbezeichner*⟩ ⟨*Muster*⟩ ... ⟨*Muster*⟩ = ⟨*Rumpf*⟩

Mehrere Muster gibt es nur bei kaskadierten Prozeduren (siehe beispielsweise die Deklarationen von `map` und `foldl`). Die vor dem Schlüsselwort = stehenden Konstituenten einer Regel werden zusammenfassend als **Kopf der Regel** bezeichnet. Bei dem nach dem Schlüsselwort = folgenden **Rumpf der Regel** handelt es sich um einen Ausdruck. Die Muster einer Regel können **Variablen** einführen, die im Rumpf der Regel benutzt werden können.

Muster sind syntaktische Objekte, die sich wie folgt definieren lassen:

1. Konstanten und Bezeichner sind Muster.
2. Das **Wildcard-Symbol** _ ist ein Muster.
3. Wenn M ein Muster und t ein Typ ist, dann ist $(M : t)$ ein Muster.
4. Wenn M_1 und M_2 Muster sind, dann ist $(M_1 :: M_2)$ ein Muster.
5. Wenn $M_1, \ldots, M_n$ Muster sind, dann sind $(M_1, \ldots, M_n)$ und $[M_1, \ldots, M_n]$ Muster.

Zusätzlich gilt, dass ein Bezeichner in einem Muster nicht mehr als 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 **Variablen** der Regel.

Ein Wert v **trifft ein Muster** M, wenn eine der folgenden Bedingungen erfüllt ist:

1. M ist ein Bezeichner oder das Wildcard-Symbol.
2. M ist eine Konstante und v ist der Wert, den die Konstante beschreibt.
3. M hat die Form $(M' : t)$ und v trifft M' (der Typ t spielt also keine Rolle).
4. M hat die Form $(M_1 :: M_2)$ und v ist eine Liste, deren Kopf M_1 und deren Rumpf M_2 trifft.
5. M hat die Form $[M_1, \ldots, M_n]$ und v ist eine n-elementige Liste $[v_1, \ldots, v_n]$, deren Elemente v_i die entsprechenden Muster M_i treffen.
6. M hat die Form $(M_1, \ldots, M_n)$ und v ist eine n-stelliges Tupel $\langle v_1, \ldots, v_n \rangle$, dessen Komponente v_i die entsprechenden Muster M_i treffen.

Hier sind Beispiele:

- Der Wert $[1,2,3]$ trifft das Muster (`x::xr`). Dabei wird `x` an 1 und `xr` an $[2,3]$ gebunden.
- Der Wert $[1,2,3,4]$ trifft das Muster (`_::x::_`). Dabei wird `x` an 2 gebunden.

Wir sagen, dass endlich viele Muster

- **disjunkt** sind, wenn kein Wert mehr als eines der Muster trifft.
- einen Typ **erschöpfen**, wenn jeder Wert des Typs mindestens eines der Muster trifft.

Beispielsweise sind die Muster `nil` und `(x::xr)` disjunkt und erschöpfen jeden Listentyp *t list*. Ein Muster, das nur aus einem Bezeichner oder dem Wildcard-Symbol besteht, erschöpft jeden Typ.

Die Regeln einer Prozedur heißen **disjunkt**, wenn ihre Muster disjunkt sind, und **erschöpfend**, wenn ihre Muster den Argumenttyp der Prozedur erschöpfen.

Wenn die Regeln einer Prozedur disjunkt sind, spielt die Reihenfolge, in der sie angegeben sind, keine Rolle. Statt

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

können wir also genauso gut

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

schreiben. Wenn die Regeln dagegen nicht disjunkt sind, ist ihre Reihenfolge von Bedeutung. In diesem Fall ist eine Regel nämlich 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.

Wenn die Regeln einer Prozedur nicht erschöpfend sind, gibt der Interpreter eine Warnung aus:

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

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

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

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

In der Praxis sollte man auf Prozeduren mit nichterschöpfenden Regeln verzichten und eventuell unerwünschten Argumente mithilfe einer zusätzlichen **Ausnahmeregel** 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.

4.8 Sortieren von ganzzahligen Listen

Eine Liste $[x_1, \ldots, x_n] \in \mathcal{L}(\mathbb{Z})$ heißt **sortiert**, wenn $x_1 \leq \cdots \leq x_n$ gilt. Die Prozedur

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

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
```

Zwei Listen heißen **bis auf Permutation gleich**, 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. Für alle Listen $xs, ys \in \mathcal{L}(\mathbb{Z})$ gilt: *xs* und *ys* sind bis auf Permutation gleich $\iff sort(xs) = sort(ys)$.

Sortieren durch Einfügen

Wir überlegen uns jetzt, wie man eine Prozedur schreiben kann, die die Sortierfunktion berechnet. 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 Gleichung gilt für die Funktion $insert \in \mathbb{Z} \times \mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$, die zu x und einer sortierten Liste xs die Liste $sort(x :: xs)$ liefert. 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

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 die Prozedur `isort` elegant als Einzeiler schreiben:

```
val isort = foldl insert nil
```

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 Gleichung gilt für die 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 Liste $sort(xs@ys)$ liefert. Wir benötigen also eine Prozedur `merge`, die zwei sortierte Listen zu einer 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* liefern, sodass gilt:

1. *ys@zs* ist eine Permutation von *xs*.
2. $|ys| \leq \frac{1}{2}(|xs|+1)$ 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 []  = []
  | msort [x] = [x]
  | 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 Deklaration

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

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

```
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

4.9 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

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
```

Bemerkungen

In diesem Kapitel haben wir unser programmiersprachliches Repertoire um Listen erweitert. In Standard ML sind Listen gemäß eines sehr einfachen mathematischen Modells (siehe Abschnitt 4.1) realisiert. Der wesentliche Punkt an der Listendarstellung ist, dass eine nichtleeren Folge durch ein Paar dargestellt wird, das aus dem ersten Element (der Kopf) und der Listendarstellung der restlichen Elemente (derRumpf) der Folge besteht.

Im Zusammenhang mit Listen haben wir regelbasierte Prozeduren und Ausnahmen kennengelernt. Dabei handelt es sich um grundlegende Sprachkonstrukte von Standard ML, die wir im Folgenden auch in anderen Zusammenhängen regelmäßig benutzen werden.

Mit „Sortieren durch Einfügen“ und „Sortieren durch Mischen“ haben wir zwei grundlegende Algorithmen besprochen, die jeder Informatiker kennen sollte. Wir können jetzt 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 für die Berechnung der Sortierfunktion $\mathcal{L}(\mathbb{Z}) \to \mathcal{L}(\mathbb{Z})$. Diese Algorithmen können in verschiedenen Programmiersprachen verschieden realisiert werden. Sie lassen sich auch problemlos auf andere Typen übertragen (beispielsweise `real`).
3. Bei den in diesem Kapitel angegebenen Prozeduren `isort` und `msort` handelt es sich jeweils um eine mögliche Realisierung der entsprechenden Sortieralgorithmen in Standard ML.

Um das Ergebnis eines Prozeduraufrufs 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.

Wir betrachten 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)
```

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 die Prozedur `reverse'` ihr Ergebnis sofort liefert. Die Erklärung dieser überraschenden Beobachtung werden wir in Kapitel 7 geben.

Glossar

Listen (Listendarstellung einer endlichen Folge; Kopf, Rumpf, Elemente, Länge $|xs|$, Konkatenation $xs@ys$, Reversion; Darstellung mit $[\ldots]$; Darstellung mit Cons $x::xr$ und nil; Listen über X, $\mathcal{L}(X)$).

Vordeklarierte Prozeduren für Listen (hd, tl, null; length, rev, map; Faltungsprozeduren foldl, foldr; List.all, List.exists, List.filter; List.concat, List.tabulate).

Andalso und Orelse.

Ausnahmen werfen (raise Empty, raise Match).

Muster (Wildcard-Symbol, Wert trifft Muster; disjunkt, erschöpfen)

Regelbasierte Prozeduren (Regel (Kopf, Rumpf, Variablen)).

Sortieren (sortierte Liste, Permutation, Sortierfunktion; bis auf Permutation gleich; Sortieren durch Einfügen, Sortieren durch Mischen).

Funktion, Algorithmus, Prozedur; Laufzeit einer Prozedur.