2.5 Typkonstruktoren

2.5.1 Funktionen [Wik, Kap.4]

2.5.1.1 Nicht–rekursive Funktionen

Funktionen werden in ML wie folgt deklariert:

`fun` <Fname> <Argname> = <Rumpf>`;`

wobei <Fname> der Name der Funktion,
<Argname> der Name des Arguments und
<Rumpf> die Rechenvorschrift der Funktion ist.

Das System gibt (nur) Namen der Funktion und den Typ zurück (nicht den Wert fn <Argname> ⇒ <Rumpf> der Funktion).

Der Typ einer Funktion hat die Form `fn : ´a –> ´b`, wobei `´a`, `´b` Typen sind.

Beispiele:

```
–   fun sum0to n = (n*(n+1)) div 2;
> val sum0to = fn : int –> int

–   fun sign x = if x<0  then ~1
      else if x>0 then 1
                  else 0;
> val sign = fn : int –> int

–   fun even x = (x mod 2) = 0;
> val even = fn : int –> bool
```

Funktionen werden in ML wie folgt aufgerufen.

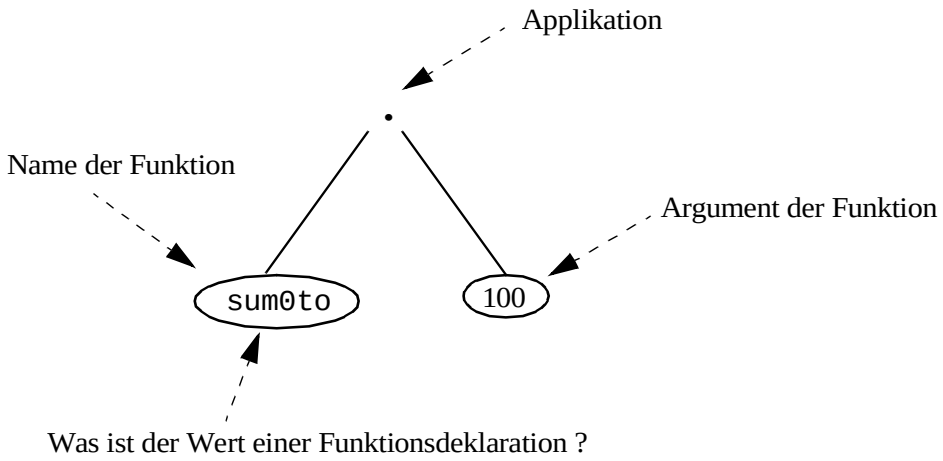
Syntax:

<Ausdruck> <Ausdruck>
↑ ↑
Funktionsname Argument

Beispiele:

```
– sum0to 100;
> 5050 : int
– sum0to(100);
> 5050 : int
```

Als Baum:



Der Wert einer Funktionsdeklaration ist eine anonyme Funktion.
Notation:

```
fn <Argname> => <Ausdruck>;
```

Beispiel: Summe

```
–   fn n => n*(n+1) div 2;
>   fn : int -> int
```

Bemerkung: Die übliche Schreibweise in der Mathematik lautet:
$(\lambda$ n. n*(n+1) div 2): int $\rightarrow$ int

Bindung:

```
<sum0to, fn n => n*(n+1) div 2>
```

Als Argumente können Ausdrücke vorkommen:

```
–    sum0to (1+1);
> 3 : int
```

```
–    sum0to 1+10; ! Klammerung von links: (sum0to 1)+10!
> 11 : int
```

Auswertung von Funktionen der Form: F E

(1) Auswertung des Operators F.

Der Wert muß ein Funktionsobjekt sein, das die folgende Form hat:

```
fn x => E'
   ↑    ↑
Parameter  Rumpf
```

(2) Auswertung des Operanden E mit Wert V.

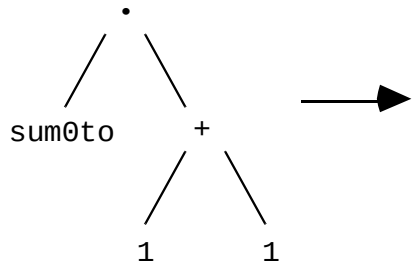
(3) Substitution von V für x überall im Rumpf E', mit dem Ergebnis E'[V/x].

(4) Auswertung von E'[V/x].

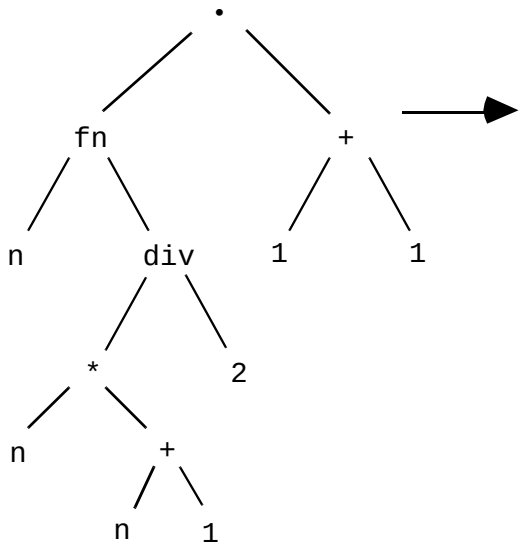
Beachte: x kann Struktur haben.

Beispiele:

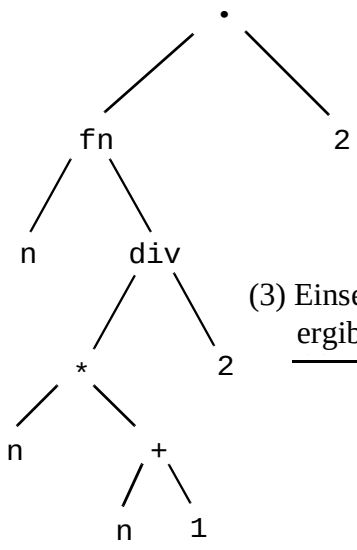
Auswertung von `sum0to (1+1)`:



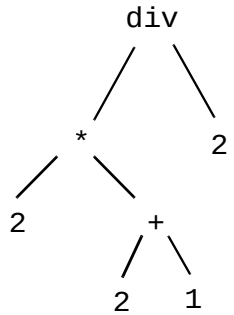
(1) Auswertung von sum0to im Environment mit der Bindung
`<sum0to, fn n =>...>`



(2) Auswertung des Arguments

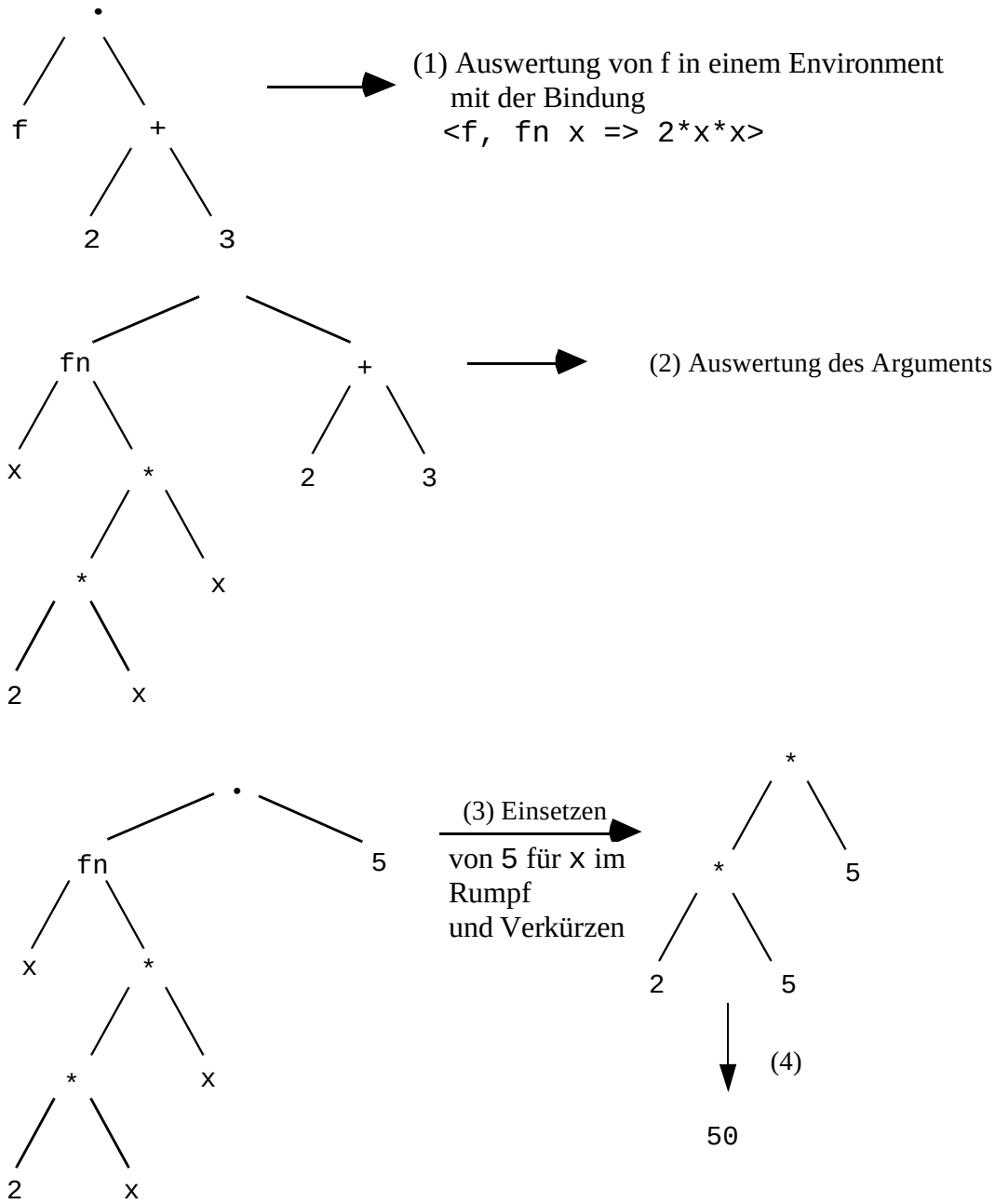


(3) Einsetzen und Verkürzen ergibt



(4) → `50`

(2) Auswertung von f(2+3)



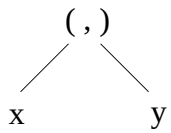
Kürzer in linearer Schreibweise:

```
f(2+3)→
(fn x => 2*x*x)(2+3) →
(fn x => 2*x*x) 5 →
2*5*5 →
50
```

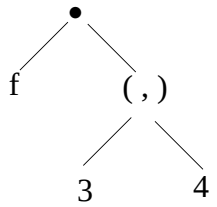
Auswertung zweistelliger Funktionen

Beispiel:
(* lineare Funktion *)
fun f(x, y) = x + 2*y;
Die Bindung ergibt:
<f, fn (x, y) ⇒ x + 2*y>

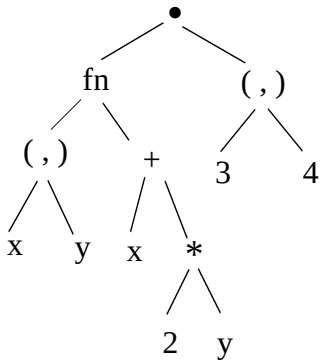
Ein Tupel (x, y) wird folgendermaßen als Baum repräsentiert:



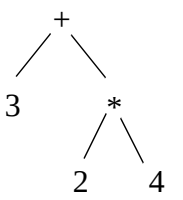
Auswertung von f(3, 4):



[Bindung auswerten]



[Substitution von 3 für x und 4 für y und Verkürzen]



[mehrere Schritte]

2.5.1.2 Rekursive Funktionen

Eine rekursive Funktionsdefinition `f` enthält mindestens einen Aufruf von `f` im

Rumpf.
Beispiel: Rekursive Definition der Summe

$$\text{sum } n = \sum_{i=0}^{n} i$$

```
–   fun sum n = if n=0 then 0
                else sum (n–1) + n;
```

Will man eine rekursive Funktion als Wertdeklaration schreiben, so muß der Wertdeklaration ein reserviertes Wort `rec` hinzugefügt werden.
Ohne das Schlüsselwort `rec` ergibt sich ein Fehler:

```
–   val sum = fn n => if n=0 then 0
                      else sum (n–1) + n;
Type checking error in: sum
Unbound value identifier in: sum
```

Mit `rec` erhält man:

```
–   val rec sum = fn n => if n=0 then 0
                          else sum (n+1) + n;
>   val sum = fn : int –> int
```

Eine rekursive Definition besteht aus zwei Arten von Fällen:

(1) **Abbruchfall** (Rekursionsanfang, Basis)
Argumente, deren Wert ohne Rekursion bestimmt wird (Grundelemente)

(2) **Rekursionsfall** (–schritt)
Argumente, bei denen sich der Funktionswert mit Hilfe eines Konstruktors berechnet, dessen Operanden näher an einem Abbruchfall liegen (rekursiver Aufruf, Induktionsannahme).

Beispiel: sum
– Abbruchfall (Rekursionsanfang): n = 0
– Rekursionsschritt: im Fall n>0 berechnet man sum(n) mit Hilfe von sum (n–1); hierbei liegt n–1 näher am Rekursionsanfang.
– Problem wenn $n \leq 0$: Rekursion auf n–1, das weiter vom Rekursionsanfang entfernt liegt, d.h. sum terminiert nicht für $n \leq 0$.

Korrektheit:

Zeige durch vollständige Induktion ($\forall\, n \in N.\ \text{sum } n = \sum_{i=0}^{n} i$)

Beweis:

Basis: (n = 0): sum 0
= {Definition}
0

Schritt: (n > 0): sum n

= {Definition}
sum (n–1) + n

= {Induktionsannahme}

$$\sum_{i=0}^{n-1} i + n$$

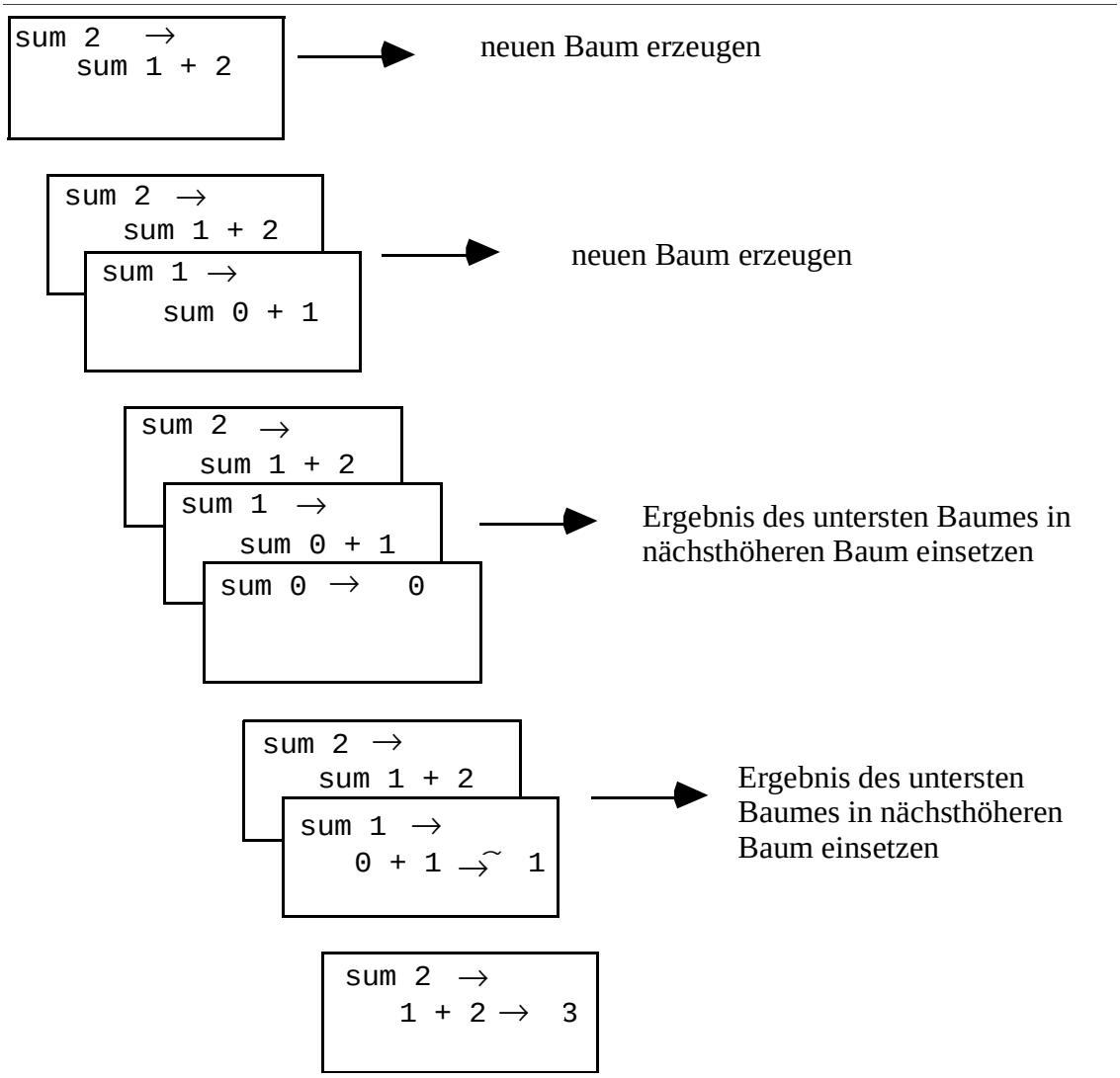
= {Arithmetik}

$$\sum_{i=0}^{n} i$$

Die Auswertung rekursiver Funktionen

Die Auswertung rekursiver Funktionen geschieht wie die Auswertung nichtrekursiver Funktionen, aber unter Berücksichtigung der Rekursion. Tritt ein rekursiver Aufruf auf, so wird die Berechnung angehalten, ein neuer Baum angelegt und dieser ausgewertet, bis wieder ein rekursiver Aufruf auftritt. Für diesen wird wieder ein Baum angelegt, usw. bis ein Ausdruck vollständig ausgewertet ist. Dieser Wert wird in den vorhergehenden Baum eingesetzt, der dann vollständig ausgewertet wird; der Wert wird in den diesem Baum vorhergehenden Baum eingesetzt, usw. bis der Wert des ursprünglichen Ausdrucks berechnet ist.

Beispiel:



Weitere Beispiele für rekursive Definitionen

(1) Fakultät: fac 0 = 1, (∀ n > 0 : fac n = 1 * 2 * ... * n = n!)

```
–    fun fac n = if n=0 then 1
                 else n * fac (n–1);
> val fac = fn : int –> int
```

(2) Exponentialfunktion: a^n für gegebenes a

```
–    fun exp (a, n) =   if n=0 then 1
                        else a * exp (a, n–1);
> val exp = fn : int*int –> int
```

(3) Test, ob eine Zahl gerade ist:

```
      fun even n =
            if n>=0
               then if n=0 then true
```

```
                else if n=1 then false
                     else even (n-2)
        else even (~n);
  > val even = fn : int -> bool
```

2.5.1.3. Exkurs: Strikte und nichtstrikte Abbildungen

Die Funktion fac terminiert nicht, wenn n<0:

```
fac(~1)          → *                [mehrere Schritte]
(~1) * fac(~2)   → *
(~1) * (~2) * fac(~3)→ ...
```

Man bezeichnet `fac` als partielle Funktion. Ebenso sind `sum` und `even` partielle Funktionen, eine weitere partielle Funktion ist die Division `div`, die für 0 im Nenner undefiniert ist.

Definition: Sei f: M → N eine Fkt., f heißt **total**, wenn f(x) für alle x ∈ M definiert ist. Sonst heißt f partiell.

Jede partielle Abbildung kann durch die Einführung eines künstlichen Wertes ⊥ auf einfache Weise totalisiert werden.

Definition:

Sei f: $M_1 \times \ldots \times M_n \to N$, und seien $\perp_0, \perp_1, \ldots, \perp_n \notin M_1 \cup \ldots \cup M_n \cup N$.

Eine ⊥-*Erweiterung* $f^{\perp}$ von f ist eine totale Abbildung

$$f^{\perp}: (M_1 \cup \{\perp_1\}) \times \ldots \times (M_n \cup \{\perp_n\}) \to (N \cup \{\perp_0\})$$

mit folgenden Eigenschaften, wobei $M := M_1 \times \ldots \times M_n$:

(i) für diejenigen $<x_1, \ldots, x_n> \in M$, für die f definiert ist, stimmen f und $f^{\perp}$ überein,

d.h. $f^{\perp}(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ für $<x_1, \ldots, x_n> \in D(f)$,

(ii) für diejenigen $<x_1, \ldots, x_n> \in M$, für die f nicht definiert ist, liefert $f^{\perp}$ den Wert $\perp_0$,

d.h. $f^{\perp}(x_1, \ldots, x_n) = \perp_0$ falls $<x_1, \ldots, x_n> \in M \backslash D(f)$.

Hier bezeichnet D(f) den Definitionsbereich von f. Totalisierungen sind nicht eindeutig bestimmt, man kann den Wert von $f^{\perp}(\perp)$ frei wählen.

Definition: Seien $\perp_0, \perp_1, \ldots, \perp_n \notin M_1 \cup \ldots \cup M_n \cup N$.

(1) Eine Funktion

$g:(M_1 \cup \{\perp_1\}) \times \ldots \times (M_n \cup \{\perp_n\}) \to (N \cup \{\perp_0\})$ heißt

strikt im i-ten Argument, $1 \le i \le n$,

falls $g(\ldots, \perp_i, \ldots) = \perp_0$ gilt.

(2) g heißt *strikt*, falls g in allen Argumenten strikt ist; sonst heißt g nicht strikt.

Beispiel:

Die Interpretationen der arithmetischen Funktionen `+, -, *, div, mod` sind strikt.

Die Interpretationen von `if-then-else, andalso, orelse` sind nicht strikt.

2.5.1.4 Terminierungsbeweise

Sei f: $T_1 \rightarrow T_2$ eine rekursive Funktion mit der Deklaration

fun f(x) = E [$f(E_1)/y_1$, ..., $f(E_k)$]

so daß die rek. Aufrufe $f(E_1)$, ..., $f(E_k)$ unter den Bedingungen $C_i(x)$, ..., $C_k(x)$ stattfinden. Auch E_1, ..., E_k hängen im Allgemeinen von x ab.

Die Terminierung von f kann nach folgendem Schema bewiesen werden:

Existiert eine **totale** Funktion[1]

h: $NF_{T1} \rightarrow NF_{int}$

so daß für alle $x \in T_1$ gilt

(1) $C_i(x) \Rightarrow h(x) > h(E_i(x))$ für i = 1, ..., k

(2) $h(x) \geq 0$

dann terminiert f für alle $x \in T_1$ (d.h. genauer für alle Normalformen $x \in NF_{T1}$).

Beispiel 1: „sum“

```
fun sum x = if x ≤ 0 then 0
                     else x + sum(x-1)
```

Es gilt $C_1(x) = (x > 0)$, $E_1(x) = (x-1)$

`sum` terminiert für alle $x \in NF_{int}$:

$$\text{Sei } h(x) =_{def} \begin{cases} x & \text{falls } x \geq 0 \\ 0 & \text{sonst} \end{cases}$$

Der Rekursionsfall gilt für $C_1(x) = (x > 0)$

Dann gilt:

(1) $C_1(x) \Rightarrow h(x) > h(E_1(x))$

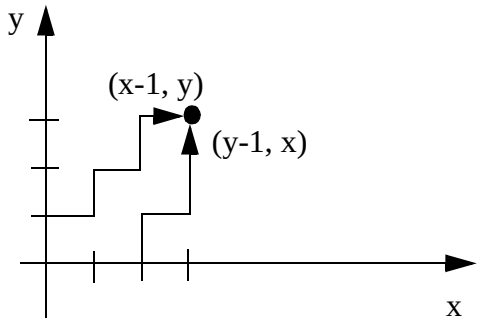
denn für $x > 0$ gilt $h(x) = x > x-1 = h(x-1)$.

(2) Für alle $x \in NF_{int}$ $h(x) \geq 0$

Beispiel: 2 „Alle Wege“

Die Funktion „alleWege“ berechnet die Anzahl der Wege, die vom Ursprung (0, 0) zu einem Punkt (x, y) führen. Dabei ist nur erlaubt, daß man auf dem Raster nach rechts oder senkrecht nach oben läuft.

1.Die Terminierungsfunktion wird hier wie in der Mathematik aufgefaßt, nicht als SML-Programm.




```
fun alleWege (x, y) =
    if x ≤ 0 orelse y ≤ 0 then 1
    else alleWege(x-1, y) + alleWege(x, y-1)
```

Die Fkt. `alleWege` hat den Typ

```
alleWege: int * int → int
```

Zur Terminierung
kann man wählen

$$h(x, y) = \begin{cases} x+y & \text{falls } x \geq 0 \text{ und } y \geq 0 \\ 0 & \text{sonst} \end{cases}$$

Dann gilt

(1) $x > 0 \wedge y > 0 \Rightarrow h(x, y) = x + y > x\text{-}1 + y$
$\wedge\, h(x, y) = x + y > x + y\text{-}1$

(2) Für alle $x, y \in NF_{int}$. $h(x, y) \geq 0$

Terminierungsbeweise für partielle Funktionen

Ist f eine partielle Funktion, die für eine Teilmenge M (von Normalformen) von T1 als terminierend bewiesen werden soll, benötigt man eine zusätzliche Bedingung:
Sei M eine Teilmenge von NF_{T1}.

f **terminiert** auf M, wenn
eine Funktion $h: M \rightarrow NF_{int}$ existiert mit

(1) f ist abgeschlossen unter M, d.h.
Für i = 1, ..., n, für alle $x \in M$
$C_i(x) \Rightarrow E_i(x) \in M$

(2) h fällt streng monoton für rek. Aufrufe.
Für i = 1, ..., n, für alle $x \in M$
$C_i(x) \Rightarrow h(E_i(x)) < h(x)$

(3) $h(x) \geq 0$ für alle $x \in M$

Beispiel:

```
fun fac n = if n = 0 then 1
            else n * fac(n-1)
```

d. h. $C_1(n) \equiv n \neq 0$, $E_1 \equiv n\text{-}1$

Wähle $h(n) =_{def} n$ für $n \geq 0$,

$M = \{n \mid \geq 0\}$

Dann gilt

(1) Für $n \geq 0$ gilt

$n \neq 0 \Rightarrow n\text{-}1 \geq 0$

denn $n \geq 0 \wedge n \neq 0 \Rightarrow n > 0$, d.h. $n\text{-}1 \geq 0$

(2) Sei $n \in M$, d.h. $n \geq 0$

Es gilt auch ohne Berücksichtigung von C_1:

$n\text{-}1 < n$

(3) $h(n) = n \geq 0$ für $n \in M$

2.5.2 Kartesische Produkte und Polymorphie [Wik, Kap.8; Stansifer, Kap.2.3]

2.5.2.1 Paare und Polymorphie

ML hat auch zusammengesetzte Strukturen (Typen). Die einfachste ist Paarbildung.

Die Rechenstruktur "Paar" besteht aus

- dem Datentyp `´a * ´b` für gegebene Datentypen (Typ aller Paare mit erster Komponente vom Typ `´a` und zweiter Komponente vom Typ `´b`),
- dem Paarkonstruktor

  `( . , . ) : ´a -> ´b -> ´a * ´b`,

  der aus zwei Argumenten ein Paar bildet.

Hier sind `´a` , `´b` **Typvariablen**. Syntaktisch besteht eine Typvariable aus „`´<Name>`“, d.h. aus einem Hochkomma gefolgt von einem Namen.
Für eine Typvariable können beliebige Typen eingesetzt werden.

Beispiele:

```
–    (3,true);
> (3,true) : int * bool
–    (5–2,not false);
> (3,true) : int * bool
–    (true,false);
> (true,false) : bool * bool;
```

Paare können geschachtelt werden:

```
–    ((3,true),7);
> ((3,true),7) : (int * bool) * int;
```

Funktionen sind ebenso Datenobjekte:

```
–    (sum0to,3);
```

```
> (fn,3) : (int -> int) * int
```

Paare können benannt werden:

```
-    val x=(7,12);
> val x=(7,12) : int * int
```

Die **Wertemenge eines Paartyps** wird folgendermaßen bestimmt:
Seien T_1, T_2 Typen mit Normalformmengen NF_{T1}, NF_{T2}.
Dann ist

$$NF_{T1*T2} = \{(n_1, n_2)\} \mid n_1 \in NF_{T1}, n_2 \in NF_{T2}\}$$
$$\cong NF_{T1} \times NF_{T2}$$

Beispiel:
(1) $NF_{int * bool} = \{(z, b) \mid z \in Z, b \in \{true, false\}\}$

Bemerkung:
Wir abstrahieren hier von der Beschränktheit des Zahlenbereichs auf konkreten Maschinen.

(2) $NF_{(int*bool) * (int \to int)} = \{((z, b), fn\ x: int => E(x)) \mid z \in Z, b \in \{true, false\},$
x Name, E(x) Ausdruck vom Typ int}

Man nennt eine Funktion **polymorph**, wenn ihr Typ Typvariablen enthält. Damit kann die Funktion Argumente beliebiger Datentypen verwenden. Durch die Namensgebung von Typvariablen können Zusammenhänge zwischen Argument- und Resultatstypen ausgedrückt werden.

Beispiel:

```
fst:    'a * 'b → 'a
fst:    (int * bool) * 'b → (int * bool)
fst:    (int → bool) * ((real → real) * int)
        → (int → bool)
```

Polymorphie bedeutet, daß das gleiche Rechenverfahren (Algorithmus) für Argumente mit **verschiedenen Typen** angewendet werden kann.
Überladen bedeutet, daß der **gleiche Name** für zwei oder mehr verschiedene Rechenverfahren angewendet wird. Die Funktion fst ist polymorph, die Multiplikation „*“ ist eine überladene Funktion, die auf int und real durch verschiedene Algorithmen realisiert wird.

Mit Hilfe des Paarkonstruktors können wir zweistellige Funktionen definieren.
Beispiel: [Wik, S.206]

Wir definieren die Exponentiation als eine Funktion mit zwei Parametern:

```
-   fun tuple_exp (a,n) =
        if n=0 then 1 else a * tuple_exp (a,n-1);
```

```
>   val tuple_exp = fn : (int * int) -> int
```

Diese Funktion kann in zwei Hinsichten verbessert werden:

a) **syntaktisch** können wir den gängigen Infixoperator `**` einführen,

b) **semantisch** können wir die rekursive Aufruffolge verkürzen.

– `infix 8 **;` (Die Funktion `**` ist infix mit Präzedenz 8)

–
```
fun a**n = if n=0 then 1
           else if n mod 2 = 0
                then   (a*a)**(n div 2)
                else a*(a*a)**(n div 2);
>   val ** = fn : (int * int) -> int
```

Weitere Standardoperationen auf Paaren

= , <> auf Paaren, die keine Funktionstypen beinhalten.

!! <, > (kleiner, größer) sind auf Paaren nicht vordefiniert.

Die Projektionen heißen `#1` und `#2` und können auch explizit definiert werden:

```
–   fun fst (x,y) = x
>   val fst = fn : (´a * ´b) -> ´a
–   fun snd (x,y) = y
>   val snd = fn : (´a * ´b) -> ´b
```

2.5.2.2. Tupel

Ein Tupel repräsentiert das Kartesische Produkt von einer beliebigen, festen Anzahl von Datentypen. Damit kann man mehrstellige Funktionen definieren.

Standardoperationen:

– Konstruktor: `(,...,) :`
`´a1 -> ´a2 -> ... -> ´an -> (´a1 * ... *´an)`

– Selektion: `#i : (´a1 * ... * ´an) -> ´ai (0 < i ≤ n)`

Beachte: Selektion existiert **nicht** auf allen Maschinen.

Beispiel:

```
–   (1,17,3);
> val it = (1,17,3) : int * int * int
– #2 (1,17,3);
> val it = 17 : int

–   (1,(2,3),4);
> val it = (1,(2,3),4) : int * (int * int) * int
```

```
–   #2(1,(2,3),4);
>   val it = (2,3) : int * int
```

Normalformen für Tupel:
Seien T_1, ..., T_n Typen.

$NF_{T1*...*Tn} =_{def} \{(m_1, ..., m_n) \mid m_i \in NF_{Ti} \text{ für } i = 1, ..., n\}$

$\cong NF_{T1} \times ... \times NF_{Tn}$

2.5.2.3. Records

Tupelelemente können auch mit individuellen (aber eindeutigen) Namen anstatt durch Position definiert werden; dann ist die Position beliebig.

```
–   val employee = {name = "Smith", first_name = "John",
    age = 25};
>   val employee = {age = 25, first_name =  "John",
                    name = "Smith"} : {age : int,
    first_name : string, name : string}
–   employee =
=      {first_name =  "John", age = 25, name = "Smith"};
>   true : bool
```

Normalformen für Records

$NF_{\{id1: T1, ..., idn: Tn\}} =$

$\{\{id_{i1} = m_{i1}, ..., id_{in} = m_{in}\} \mid (i1, ..., in)$ ist Permutation von $(1, ..., n)$ und $m_j \in NF_{Tj}$ für $j = 1, ..., n\}$

Tupel können als eine Spezialisierung von Records angesehen werden: in Tupeln sind die Komponentennamen Integer.

```
–   val x = {3=1,2=true,1=()}
>   val x = ((),true,1) : unit * bool * int
```

Beachte: Das geht auch **nicht** auf allen Maschinen!

Indizieren in Records funktioniert wie mit Tupeln, nur per Namen:

```
–   #age employee;
>   25 : int
```

2.5.3 Listen [Wik, Kap.13]

Eine Liste ist eine (endliche) Folge von Datenobjekten gleichen Typs. Listen sind polymorph, d.h. jeder Datentyp ist als Typ für die Elemente einer Liste zugelassen, jedoch müssen alle Elemente einer Liste vom selben Typ sein. (Diese Eigenschaft

nennt man auch **Homogenität**; Tupel dürfen **heterogen** sein.)

Listenkonstruktoren:

`[] : ´a list` "die leere Liste"
`:: : ´a * ´a list -> ´a list` (Präzedenz 5, assoziiert nach rechts!)

Beispiele:

```
– nil;
> val it = [] : ´a list
– 1 :: nil;
> val it = [1] : int list
– 1 :: 2 :: 3 :: nil;
> val it = [1,2,3] : int list
```

$[x_1, ..., x_n]$ ist die Normalform für endliche Listen, d.h. die Normalformmenge eines Typs T list ist folgendermaßen definiert:

$NF_{T\ list} = \{[m_1, ..., m_n] \mid m_j \in NF_T \text{ für } j = 1, ..., n, n \geq 0\}$

Weitere Beispiele:
Listen von Paaren und Listen von Listen:

```
[2,3] : int list
[(2,3), (3,4)] : (int * int) list
[[2,3], [2,3,4]] : int list list
  ↑         ↑
   int list
```

!! **Nicht** korrekt ist:

```
[(2,3), (2,3,4)]
  ↑     ↑
```

verschiedene Typen: `int * int` bzw. `int * int * int`

2.6 ML Programmierstil

2.6.1 Mustervergleich (Pattern Matching) [Wik, Kap.12]

Die Konstruktoroperatoren von Datentypen erlauben es in einfacher Weise, Fallunterscheidungen bei der Beschreibung des Funktionsrumpfes anzugeben.

Wir schreiben:

```
fun <name> <pattern_1> =   <Ausdruck_1>
  | <name> <pattern_2> =   <Ausdruck_2>
      :          :
  | <name> <pattern_k> =   <Ausdruck_k>;
```

Hier bezeichnet <name> den namen der Funktion, <pattern$_i$> sind **aus Konstruktoren** gebildete Ausdrücke, <Ausdruck$_i$> sind beliebige Ausdrücke, die den gleichen Typ besitzen müssen. In <pattern$_i$> tritt die gleiche Variable nie mehrmals auf.

Beispiele:

(1) Prüfen auf leere Liste

```
–   fun null nil     = true
=   |   null (x :: xl) = false;
>   val null = fn : ´a list -> bool
```

(2) Quadrieren der Elemente

```
–   fun quad x:int = x * x;
–   fun quadlist nil = nil
=   |   quadlist ((x:int) :: xl) = quad x :: quad-
    list(xl);
```

(3) Zusammenfügen von zwei Listen

```
–   fun app nil       yl = yl
=   |   app (x :: xl) yl = x :: app xl yl;
```

Zusammenfügen wird mit der @ Standardfunktion realisiert (Präzedenz 5):

```
    xl @ yl = app xl yl
```

(4) Revertieren einer Liste

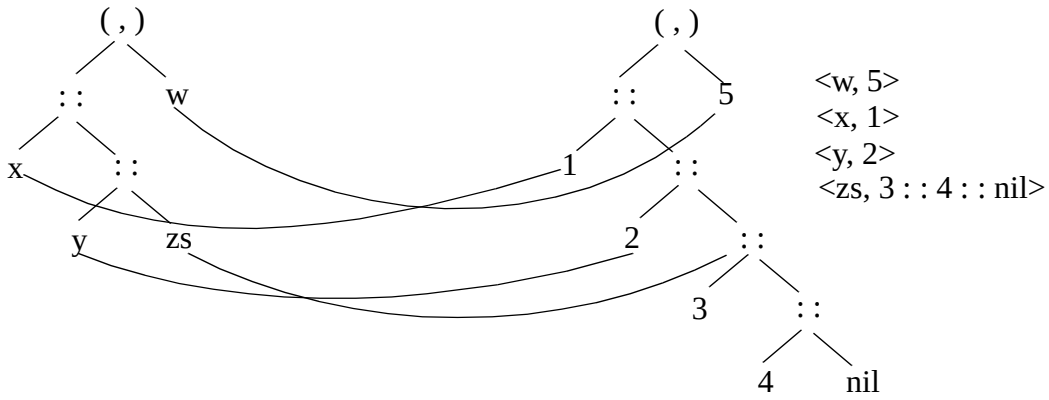
```
–   fun rev nil                  = nil
=   |   rev (x :: xl) = (rev xl) @ (x :: nil);
```

Pattern matching

Um zu prüfen, ob ein Ausdruck auf ein Pattern paßt, verwendet man den Vergleich der assoziierten Bäume.

Beispiel:

Pattern: (x : : y : : zs, w)
Ausdruck: ([1 : : 2 : : 3 : : 4], 5)
Als Bäume geschrieben erhält man



Um ein Pattern mit einem Ausdruck zu vergleichen, überlagern wir die beiden Bäume und vergleichen zunächst die Wurzeln und dann induktiv die Teilbäume. Dabei werden Bindungen für die Variablen des Pattern aufgebaut. Beim Vergleich unterscheidet man die folgenden Fälle:

1) Das Pattern ist eine Variable x. Dann wird x an den korrespondierenden Ausdruck gebunden.

2) Das Pattern ist eine Konstante. Dann muß der Ausdruck die gleiche Konstante sein. Sonst schlägt der Vergleich fehl.

(3) Die Wurzel des Pattern ist ein Funktionssymbol. Dann muß die Wurzel des Ausdrucks damit übereinstimmen; außerdem müssen alle Teilbäume des Ausdrucks mit denen des Baums „matchen“. Sonst schlägt der Vergleich fehl.

2.6.2 Fehlerbehandlung [Stansifer, Kap.6]
In SML wie in (fast) jeder anderen Programmiersprache gibt es drei grundsätzliche Arten von Fehlern:
- Syntaxfehler
- Laufzeitfehler
- semantische Fehler

Als **Syntaxfehler** bezeichnet man ein grammatikalisch unkorrektes Programm, d.h. ein Programm, das die Grammatik der Programmiersprache verletzt.
Beispiele: Falsch geschriebene Schlüsselwörter (lexikalischer Fehler), falsch gesetzte Kommata, Strichpunkte (Fehler der kontextfreien Syntax), Typfehler (kontextsensitive Fehler).
Syntaxfehler werden statisch vor Ausführung des Programms (durch den Compiler)

erkannt.

Tritt während der Ausführung eines syntaktisch korrekten Programms ein Fehler auf, so spricht man von **Laufzeitfehler**.

Beispiele:
- Division von 0
- Aufruf eines aktuellen Parameters außerhalb des Definitionsbereichs einer Funktion

Als **semantisch nicht korrekt** bezeichnet man ein Programm, das (mindestens) einen anderen Wert liefert als es seine Spezifikation (Anforderungsbeschreibung) verlangt. Umgekehrt bezeichnet man ein Programm, das seine Spezifikation erfüllt als (semantisch) korrekt. (siehe später)

Beispiel:

Spezifikation: abs: int → int, abs(x) = |x|

Implementierung

```
fun abs x = if x = 0 then 1
            else if x > 0 then x
                 else -x
```

Die Implementierung ist nicht semantisch korrekt, da sie für x = 0 einen falschen Wert liefert.

Um Laufzeitfehler exakt zu behandeln und Programme „robust“ zu gestalten, gibt es die folgende Fehlerbehandlung in ML:

Mit

```
exception <name>;
```

wird eine "einfache" Ausnahme definiert, die nur den Namen <name> angibt und den trivialen Typ `unit` besitzt, der genau ein Objekt () als Element hat.

Mit

```
raise <name>
```

wird die Berechnung abgebrochen und die Fehlerbehandlung aufgerufen. Damit läßt sich auch für andere Funktionen die Ausnahmebehandlung genau eingrenzen. Fehlerbehandler werden mit der Direktive `handle` definiert, auf die hier aus Zeitgründen nicht näher eingegangen wird. Die im folgenden deklarierten Ausnahmen werden vom System als unbehandelt gemeldet.

Beispiele:

(1) Fakultät

```
–   exception Fac;

–       fun fac 0 =  1
=        |  fac n = if n>0 then n * fac (n–1)
=                  else raise Fac;
```

(2) Erstes Element einer Liste

```
–       exception Hd;
–       fun   hd nil            = raise Hd
=        |    hd (x :: xl)      = x;
```

(3) Liste ohne das erste Element

```
–       exception Tl
–       fun   tl nil            = raise Tl
=        |    tl (_ :: xl)      = xl;
```

Beachte: `hd` und `tl` sind in ML Standardfunktionen, die hier zur Illustration neu deklariert werden.

(4) Letztes Element einer Liste

```
–       exception Last
–       fun   last nil              = raise Last
=        |    last (x :: nil)       = x
=        |    last (x :: y :: yl)   = last (y :: yl);
```

(5) Sortieren mit dem "Insertion Sort"

```
–       fun   insert x nil          = [x:int]
=        |    insert x (y :: yl)    =
=               if x<=y then x :: y :: yl
=                       else y :: insert x yl;

–       fun   sort nil              = nil
=        |    sort (x :: xl)        = insert x (sort xl);
```

2.6.3 Funktionen höherer Ordnung

Funktionen sind Daten, können also als Argumente und Werte von Funktionen (höherer Ordnung) vorkommen. Ein Typ heißt **höherer Ordnung**, wenn mindestens zwei Pfeile darin vorkommen.
Wir betrachten zunächst den

Unterschied von `int -> (int -> int)` und `(int * int) -> int`

```
–   fun add x y = x+y;
>   val add = fn : int -> (int -> int)
–   fun plus (x,y) = x+y;
>   val plus = fn : (int * int) -> int
```

`add` ist eine Funktion mit einem Argument, deren Wert eine Funktion mit einem Argument ist; die Argumente werden nacheinander gegeben.
`plus` ist eine Funktion mit einem Paar als Argument; beide Argumente werden gleichzeitig gegeben.

Beispiel:

```
–   fun pr1 x y = x;
```

```
>   val pr1 = fn :´a -> (´b -> ´a)
–   fun fst (a,b) = a;
>   val fst = fn : a * b -> a
```

d.h. `fst` hat ein Paar als Argument;
`pr1` ist eine einstellige Funktion, deren Werte Funktionen sind.

Der Übergang von $a \times b \to c$ nach $a \to b \to c$ wird auch als **Currying** bezeichnet (nach H. B. Curry, obwohl von M. Schönfinkel erfunden).
Es gibt eine Funktion, die den Übergang herstellt:

```
– fun curry f = fn x => (fn y => f(x,y));
> val curry = fn : (´a * ´b -> ´c) -> (´a -> (´b -> ´c))
```

Beispiel:

```
–   curry fst;
>   fn : ´a ->(´b -> ´a) (* mit Wert fn x =>(fn y => fst (x,y))*)
–   fun uncurry f = fn (x,y) => f x y;
>   val uncurry = fn : (´a -> ´b -> ´c) -> (´a * ´b) -> ´c
```

Beispiel:

```
–   curry fst 2 10;
>   2 : int
–   uncurry pr1 (2,10);
>   2 : int
```

Bei "gecurryten" Funktionen ist die Reihenfolge der Argumente wichtig; insbesondere kommen im allgemeinen die Funktionsargumente vor den Nichtfunktionen.

Folgerung:
Für alle Funktionen `f : ´a -> (´b -> ´c)`
und `g : (´a * ´b) -> ´c` gilt:

```
curry (uncurry f) = f
uncurry (curry g) = g
```

Ein weiteres Beispiel für eine Funktion höherer Ordnung ist die Iteration von Funktionsapplikationen:

Beispiel:

(1)

```
– fun twice  f x = f (f x);
> twice = fn : (´a -> ´a) -> (´a -> ´a)

– fun square x:int = x * x;
> val square = fn : int -> int
– twice square 5
= (* = (twice square) 5 = square (square 5) = *);
> 625 : int

– fun pow4 x = twice square x;
```

```
> val pow4 = fn : int -> int
```

Im Folgenden betrachten wir 2 wichtige Operationen auf Listen.
Die Funktion **map** wendet eine Funktion f alle Argumente einer Liste an:

map f $[x_1, ..., x_n] = [f\ x_1, ..., f\ x_n]$

```
– fun map f nil            = nil
=   | map f (x :: xl)      = (f x) :: (map f xl);
> val map = fn : (´a -> ´b) -> ´a list -> ´b list
```

Beispiele: Die Auswertung von map square geschieht folgendermaßen:

```
(1)   map square [1,2,3]
  →   square 1 :: map square [2,3]
  →   1 :: map square [2,3]
  →   1 :: square 2 :: map square [3]
  →   …
  →   [1,4,9]
(2)   fun quadlist xl = map square xl;
(3)   - map (fn x => x+1) [1,2,3];
      > val it = [2, 3, 4]
(4)   - map size ["a", "", "vier"];
      > val it = [1, 0, 4]
(5)   val text =
        [["Aus","eins","mach","zehn"],
         ["Und","zwei","lass","gehn"],
         ["Und", "drei","mach", "gleich"],
         ["So", "bist","Du","reich!"]];
```

(6) Anzahl der Elemente in einer Liste von Listen

```
      - map (map size) text;
      > val it =
        [[3, 4, 4, 4],
         [3, 4, 4, 4],
         [3, 4, 4, 6],
         [2, 4, 2, 6]];
```

Die Operation **foldr** verknüpft die Elemente einer Liste mit Hilfe einer binären Operation # und einer Konstante `u:`

```
foldr # u [x1,...,xn] =
        x1 #(x2 #(...(xn # u)...))
```

Die rekursive Definition von `foldr` lautet:

```
- fun foldr f u nil     = u
   |  foldr f u (x::xl) = f(x, foldr f u xl);
```

```
> val foldr = fn: (´a -> ´b) -> ´a -> ´a list -> ´b
```

Beispiele:

(1) Die Operation `sum [x1,...,xn] =`
`x1+(x2 +(...(xn + 0)...))`
mit der rekursiven Definition

```
sum nil = 0
sum (x::xl) = x + sum xl;
```

läßt sich folgendermaßen mit `foldr` implementieren[2]:

```
- val sum = foldr op+ 0;
> val sum = fn: int list -> int

- sum [1, 2, 3];
> val it = 6
```

(2) Die Operation `implode ["x1",...,"xn"] =`
`"x1"^("x2"^(...("xn" ^"")...))`
mit der rekursiven Definition

```
fun implode nil     = ""
  |  implode (x::xl) = x ^ implode xl;
```

läßt sich folgendermaßen mit `foldr` implementieren:

```
- val implode = foldr op^ "";
> val implode = fn: string list -> string

- implode ["Aus ","eins ","mach ","zehn"];
> val it = "Aus eins mach zehn"
```

(3) Die Operation `rev [x_1,...,x_n] = [x_n,...,x_1] =`
`[x_n] @([x_n-1] @(...([x_1] @[])...)) =`
`(rev [x_2,...,x_n]) @ [x]`
mit der rekursiven Definition

```
rev nil = nil
rev (x::xl) = (rev xl) @ [x];
```

läßt sich folgendermaßen mit `foldr` implementieren:

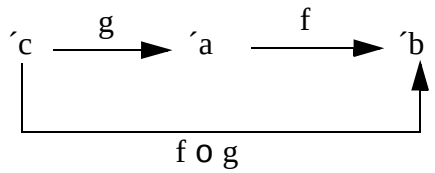
```
- fun rev l = foldr (fn (x,y) => y@[x]) nil l;
> val rev: ´a list -> ´a list

- rev ["Aus ","eins ","mach ","zehn"];
> val it = ["zehn", "mach", "eins", "Aus"]
```

Die **Funktionskomposition** erlaubt es zwei Funktionen hintereinander anzuwenden:
Sei f: ´a → ´b, g: ´c → ´a, dann ist f o g: ´c → ´b wie folgt definiert:

$$(f \circ g)(x) =_{def} f(g(x))$$

---

2. Ist # eine Funktion in Infix-Notation, so wird durch `op#` die Infixschreibweise lokal aufgehoben und `op#` als zweistellige Funktion aufgefaßt.




```
– fun comp (f,g) x = f (g x);
> val comp = fn ((´a -> ´b) * (´c -> ´a)) -> ´c -> ´b
```

`comp` wird Infix mit `o` geschrieben und hat Präzedenz 3.

Beispiele:

```
– fun twice f = f o f
– fun last x = (hd o rev) x;
– fun second x = (hd o tl) x;
```

Mehr zum Thema Ausführungseffizienz im zweiten Semester.

2.6.3 Lokale Deklarationen in Ausdrücken

Mit einem `let`–Ausdruck kann der Bindungsbereich einer Variablen beschränkt werden, ohne daß eine Neudeklaration erfolgt. (Dieses Prinzip der Bindungsbereichsbeschränkung ist im Gegensatz zur Neudeklaration funktional.) Eine mit `let` formulierte Wertdeklaration schafft eine Bindung, die nur für den `let`–Ausdruck gültig ist. Eine vorherige Bindung eines in `let` gebundenen Namens bleibt nach Auswertung des `let`–Ausdrucks erhalten.

```
–   val a = 3 and b = 5;
>   val b = 5: int
>   val a = 3: int
–   let val asq = a*a in asq + asq end;
>   18: int
–   asq;
Typechecking error in: asq
Unbound value identifier: asq
```

Wenn ein Name vor dem `let`–Ausdruck gebunden war, so ist er es auch danach.

Die syntaktische Form des `let`–Ausdrucks ist

`let` <Deklaration> `in` <Ausdruck> `end;`

Der `let` Ausdruck ist nur eine syntaktische Umschreibung (syntactic sugar) für die Funktionsapplikation:

`let val` <Name> `=` <Ausdruck1> `in` <Ausdruck2> `end;`

entspricht `(fn` <Name> `=>` <Ausdruck2>) <Ausdruck1>;

Mehrere Namen werden durch verschachtelte `let`–Ausdrücke gebunden.

Mit `let` gebundene Namen heißen auch **lokale Variablen**. Die Benutzung lokaler Variablen hat mehrere Vorteile:

(1) Variablen brauchen nicht länger behalten zu werden als notwendig. Dadurch werden die Programme kürzer; die Effizienz des Programms wird verbessert, da ein mehrfach auftretender Ausdruck nur einmal ausgewertet wird.
(2) Lokale Bindungen können das Programm nur in Teilen beeinflussen, für die sie relevant sind. Die Möglichkeit unbeabsichtigter Fehler wird so eingeschränkt. Das erleichtert Modifikationen des Programms.

Eine bestehende Bindung einer Variablen kann durch einen `let`–Ausdruck verdeckt werden. Mit den gegebenen Bindungen für `a` und `b`.

```
–   let val a = 7 in a+b end;
>   12: int
–   a;
>   3: int
```

`a` erhält einen neuen Wert im `let`–Ausdruck, aber der alte Wert von `a` bleibt erhalten.

Das Semikolon und `and` haben innerhalb eines `let` dieselbe Wirkung wie außerhalb[3]:

```
–   let val a = b and b = a in (a,b) end;
>   (5,3) : int * int
–   let val a = b; val b = a in (a,b) end;
>   (5,5) : int * int
```

Man kann auf den Strichpunkt auch verzichten:

```
–   let val a = b
=       val b = a
      in (a,b)
    end;
>   (5,5) : int * int
```

Die globalen Werte von `a` und `b` wurden durch die lokalen Deklarationen nicht verändert.

```
–   a; b;
>   3: int
>   5: int
```

3. Beachten Sie aber die unterschiedliche Bedeutung: `val a =` E_1 `and b =` E_2 erzeugt **simultan** die Bindungen von a und b, während `val a =` E_1`; val b =` E_2 die Bindungen **sequentiell** erzeugt.

Ein `let`–Ausdruck kann überall dort stehen, wo ML–Ausdrücke stehen dürfen.

```
–   val f = let val c = 200.0        (Konversion von Celsius
            in (c/5.0)*9.0 + 32.0     nach Fahrenheit)
            end
>   392.0: real
```

Als Anwendungsbeispiel betrachten wir einen schnellen Sortieralgorithmus, **Sortieren durch Verschmelzen**. Dieses Sortierverfahren wurde 1945 von John von Neumann entwickelt.

Es verwendet zwei Hilfsfunktionen:
Die Funktion `split` spaltet eine Liste in zwei Listen auf, die Funktion `merge` verschmilzt zwei aufsteigend geordnete Listen zu einer geordneten Liste:

```
(*Aufspalten einer Liste:
split [x_1,...,x_n] =
([x_1, x_3,...],[x_2, x_4,...]) *)

fun split nil       = (nil, nil)
 |  split (x::nil)  = (x::nil, nil)
 |  split (x::y::xl)=
            let val (l,r) = split xl
            in (x::l, y::r)
            end;
> val split = fn: ´a list -> ´a list * ´a list

- val list = [17,10,23,35,13,2,7];
- split list;
> val it = ([17, 23, 13, 7], [10, 35, 2])

(*Mischen zweier geordneter Listen zu einer geordneten
Liste*)
fun merge (xl, nil) = xl
 |  merge (nil, yl) = yl
 |  merge (x::xl, y::yl) =
            if x<=y then x::merge (xl, y::yl)
            else         y::merge (x::xl, yl);
> val merge = fn: ´a list * ´a list -> ´a list

- val list1 = [17,23,35];
- val list2 = [12,14,48,53];
- merge (list1, list2);
> val it = [12, 14, 17, 23, 35, 48, 53]
```

Beim Sortieren durch Verschmelzen wird eine Liste mit mindestens zwei Elementen zunächst in zwei Listen aufgespalten und diese nach dem gleichen Verfahren (rekursiv) sortiert. Die beiden (geordneten) Teilergebnisse werden dann zu einer geordneten Liste verschmolzen.

```
(*Sortieren durch Mischen*)
fun mergeSort nil    = nil
 |  mergeSort (x::nil) = x::nil
 |  mergeSort (x::y::xl) =
     let val (l,r) = split (x::y::xl)
     in merge(mergeSort l, mergeSort r)
     end;
> val mergeSort = fn: ´a list -> ´a list

- mergeSort [17,10,23,35,13,2];
> val it = [2, 10, 13, 17, 23, 35]
```