Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 7: Abstraktionsbildung mit Prozeduren höherer Ordnung

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

Überblick

1. Prozeduren als Parameter und Wert von Prozeduren
2. Currying
3. Funktionskomposition
4. Grundlegende Funktionen höherer Ordnung
5. Beispiel: Ein Rekursionsschema zur Akkumulation

Funktionen und Prozeduren höherer Ordnung

- In SML (und in anderen funktionalen Programmiersprachen) sind Funktionen Werte (Kapitel 2).
- Funktionen können (genauso wie andere Werte) als Parameter von Prozeduren (also auch von Funktionen) auftreten.
- Prozeduren bzw. Funktionen, die als Parameter oder als Wert Funktionen haben, werden „*Prozeduren bzw. Funktionen höherer Ordnung*“ genannt.

Hierarchie der Prozeduren höherer Ordnung

- Der Bezeichnung „*Prozeduren höherer Ordnung*“ liegt die folgende Hierarchie zu Grunde:
  - Die Ordnung `0` umfasst die Konstanten.
  - Die Ordnung `1` umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung `0` sind.
  - Die Ordnung *n* `+ 1` umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung *n* sind.

functionals und Operatoren

- „*Funktionen höherer Ordnung*“, die Funktionen unter ihren Parametern haben, werden manchmal „*functionals*“ und „*Operatoren*“ genannt.

Beispiel zur nützlichen Abstraktion von Funktionen höherer Ordnung

- Ist *f* eine (differenzierbare) mathematische Funktion R → R , so kann die Ableitung *f*′ von *f* wie folgt geschätzt werden (Δ (*Delta*) ist eine kleine reelle Zahl):

$$f'(x) = (f(x + \Delta) - f(x)) / \Delta$$

Der tatsächliche Wert von $f'(x)$ ist der Limes dieses Quotienten für Δ gegen 0.

Implementierung in SML

- Die Schätzung der Ableitung einer Funktion R → R lässt sich in SML so implementieren:

```
- val delta = 1e~5;
val delta = 1E~5 : real

- fun abl(f) = let
                   fun f´(x)=(f(x+delta)-f(x))/delta
               in
                   f´
               end;
val abl = fn : (real -> real) -> real -> real
```

abl

- Die Funktion `abl` hat als Parameter eine Funktion `f`.
- Innerhalb von `abl` wird eine neue Funktion `f´` mit Hilfe von `f` und `delta` definiert.
- Das Ergebnis von `abl(f)` ist die neue Funktion `f´` selbst.
- `abl(f)` kann wiederum auf eine reelle Zahl angewandt werden.

Typen von `abl`

- Da der Typkonstruktor „`->`“ rechtsassoziativ ist, bezeichnet der Typausdruck:

  ```
  (real -> real) -> real -> real
  ```

  den Typ:

  ```
  (real -> real) -> (real -> real).
  ```

- `abl` hat als Parameter eine Funktion vom Typ

  ```
  real -> real
  ```

  und liefert als Ergebnis eine Funktion vom Typ

  ```
  real -> real
  ```

Anwendung von `abl` auf eine reelle Zahl

- Beispiele:

```
- abl( fn x => x*x )(5.0);
val it = 10.0000099994 : real

- abl( fn x => x*x*x )(5.0);
val it = 75.0001499966 : real
```


Beeinflussung der Schätzung durch `delta`

- Die Ableitung der Funktion $x \rightarrow x^2$ ist die Funktion $x \rightarrow 2x$.
- Sie hat an der Stelle `5` den Wert `10`.
- Die Ableitung der Funktion $x \rightarrow x^3$ ist die Funktion $x \rightarrow 3x^2$
- Sie hat an der Stelle `5` den Wert `75 (= 3*25)`.
- Die mit SML berechneten Zahlen sind Schätzungen, die den richtigen Werten sehr nahe kommen.
- Mit anderen Werten für `delta` kann man die Schätzung verbessern.

`delta` als zweiter Parameter von `abl`

```
- fun abl(f, delta:real) =
   let
         fun f´(x) = (f(x+delta) - f(x)) / delta
   in
         f´
   end;
val abl = fn : (real -> real) * real -> real -> real

- abl( fn x => x*x, 1e~10 )(5.0);
val it = 10.0000008274 : real

- abl( fn x => x*x*x, 1e~10 )(5.0);
val it = 75.0000594962 : real
```


Anonyme Definition

- Man kann die neue Funktion auch anonym definieren (ohne ihr einen Namen zu geben, der außerhalb von `abl` nicht definiert ist).
- Damit kommt man zu folgender, gleichwertiger Definition:

```
- fun abl(f, delta:real) =
                fn x => (f(x+delta) - f(x)) / delta;
val abl = fn : (real -> real) * real -> real -> real
```

- Man kann auch ganz auf `fun` verzichten:

```
- val abl = fn(f, delta:real) => fn x =>
                        (f(x+delta) - f(x)) / delta;
val abl = fn : (real -> real) * real -> real -> real
```

Fazit zu `abl`

- Die letzten drei Definitionen von `abl` liefern als Ergebnis eine Funktion, die selbst wieder als Parameter von `abl` geeignet ist.
- Man kann `abl` also auch so verwenden:

```
- abl( abl(fn x => x*x, 1e~5), 1e~5 )(5.0);
val it = 2.00003569262 : real

- abl( abl(fn x => x*x*x, 1e~5), 1e~5 )(5.0);
val it = 30.0002511722 : real
```

Die Ableitung der Ableitung von $x \rightarrow x^2$ ist die Funktion $x \rightarrow 2$, und diese hat an der Stelle 5 den Wert 2.
Die Ableitung der Ableitung von $x \rightarrow x^3$ ist die Funktion $x \rightarrow 6x$, und diese hat an der Stelle 5 den Wert 30.

Beispiel: Identitätsfunktion `id`

- Die Identitätsfunktion `id` ist ein weiteres Beispiel für eine Funktion höherer Ordnung:

```
- val id = fn x => x;
val id = fn : 'a -> 'a

- id(2);
val it = 2 : int

- id(id)(2);
val it = 2 : int
```

Im Teilausdruck `id(id)` hat die Funktion `id` sich selbst als Parameter und liefert auch sich selbst als Ergebnis.

- Da die Funktion eine Funktion als Parameter und auch als Wert haben kann, ist sie eine Funktion höherer Ordnung.
- `id` ist polymorph und kann damit auf Werte verschiedener Typen angewandt werden.

Überblick

1. Prozeduren als Parameter und Wert von Prozeduren
2. Currying
3. Funktionskomposition
4. Grundlegende Funktionen höherer Ordnung
5. Beispiel: Ein Rekursionsschema zur Akkumulation

Prinzip

- Betrachte die folgende mathematische Funktion:

$$f: \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$$
$$(n_1, n_2, n_3) \rightarrow n_1 + n_2 + n_3$$

- f kann in SML so implementiert werden:

```
- fun f(n1, n2, n3):int = n1 + n2 + n3;
val f = fn : int * int * int -> int
```

$f\,1$

- Bezeichne die Menge der Funktionen von A in B mit $F(A, B)$.
- Aus der Funktion f lässt sich folgende Funktion definieren:

$$f1 : \mathbf{Z} \rightarrow F(\mathbf{Z} \times \mathbf{Z}, \mathbf{Z})$$
$$n_1 \rightarrow f1(n_1)$$

mit

$$f1(n_1) : \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$$
$$(n_2, n_3) \rightarrow f(n_1, n_2, n_3) = n_1 + n_2 + n_3$$

Implementierung von $f\,1$ in SML

```
- fun f1(n1)(n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int
- f1(1)(1, 1);
val it = 3 : int
- f1(1);
val it = fn : int * int -> int
```

- Wegen der Rechtsassoziativität des Typkonstruktors „`->`“ und der Präzedenzen zwischen „`*`“ und „`->`“ bezeichnet:

```
int -> int * int -> int
```

den Typ:

```
int -> ((int * int) -> int)
```

- Die Funktion `f1` bildet eine ganze Zahl auf eine Funktion vom Typ `((int *int) -> int)` ab.

f 11

- Ähnlich lässt sich für jede ganze Zahl n_1 aus der Funktion $f\,1\,(n_1)$ die Funktion $f\,11$ definieren:

$$f\,11\,(n_1):\quad \mathbf{Z} \rightarrow F(\mathbf{Z}, \mathbf{Z})$$
$$n_2 \rightarrow f\,11\,(n_1)\,(n_2)$$

mit

$$f\,11\,(n_1)\,(n_2):\quad \mathbf{Z} \rightarrow \mathbf{Z}$$
$$n_3 \rightarrow f\,1\,(n_1)\,(n_2, n_3) = f\,(n_1, n_2, n_3) = n_1+n_2+n_3$$

Implementierung von *f* 11 in SML

```
- fun f11(n1)(n2)(n3) = f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

- Wegen der Rechtsassoziativität des Typkonstruktors „->“ bezeichnet:

```
int -> int -> int -> int
```

den Typ:

```
int -> (int -> (int -> int))
```

Beispiele zu *f* 11

```
- f11(1)(1)(1);
val it = 3 : int

- f11(1)(1);
val it = fn : int -> int

- f11(1);
val it = fn : int -> int -> int
```

So lässt sich jede *n*-stellige Funktion durch einstellige (unäre) Funktionen darstellen. In vielen Fällen werden wesentlich kompaktere und übersichtlichere Schreibweisen von Funktionsdefinitionen möglich.

Andere Syntax zur Deklaration von „curried“ Funktionen

- Erinnerung:

  In SML ist ein einelementiger Vektor mit seinem Element identisch!

Deklaration von `f1`

```
- fun f1 n1 (n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int
- f1(1)(1, 1);
val it = 3 : int
- f1 1 (1, 1);
val it = 3 : int
- f1(1);
val it = fn : int * int -> int
- f1 1;
val it = fn : int * int -> int
```


Deklaration von `f11`

```
- fun f11 n1 n2 n3 = f1 n1 (n2, n3);
val f11 = fn : int -> int -> int -> int
- f11(1)(1)(1);
val it = 3 : int
- f11 1 1 1;
val it = 3 : int
- f11(1)(1);
val it = fn : int -> int
- f11 1 1;
val it = fn : int -> int
- f11(1);
val it = fn : int -> int -> int
- f11 1;
val it = fn : int -> int -> int
```

Deklaration von `f1` und `f11` mit `fn`

```
- val f1 = fn n1 => fn (n2, n3) => f(n1, n2, n3);
val f1 = fn : int -> int * int -> int

- val f11 = fn n1 => fn n2 => fn n3 => f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

- Das (einfache!) Prinzip, das der Bildung von `f1` aus `f` und der Bildung von `f11` aus `f1` zugrunde liegt, wird nach dem Logiker *Haskell B. Curry* „*Currying*“ genannt.
- Die *n*-stellige Funktion, die sich aus der Anwendung des Currying auf eine (*n*+1)-stelligen Funktion *f* ergibt, wird „*curried*“ Form von *f* genannt.

Einfache Deklaration von curried Funktionen

```
- fun f11 n1 n2 n3 = ...;

- val f11 = fn n1 => fn n2 => fn n3 =>...;
```

zeigen, dass sich curried Funktionen einfach dadurch deklarieren lassen, dass ihre Parameter nicht als Vektor angegeben werden:

```
- fun f´ n1 n2 n3 : int = n1 + n2 + n3;
val f´ = fn : int -> int -> int -> int
```

- `f´n1` entspricht der Funktion `f1(n1)`.
- `f´n1 n2` entspricht der Funktion `f11(n1)(n2)`.

Rekursive curried Funktionen Beispiel: ggT

```
- fun ggT a b = if a < b
                then ggT b a
                else if b = 0
                     then a
                     else ggT b (a mod b);
val ggT = fn : int -> int -> int

- ggT 150 60;
val it = 30 : int

- ggT 150;
val it = fn : int -> int
```

Die Schreibweise `ggT b a mod b` statt `ggT b (a mod b)` wäre inkorrekt: Wegen der Präzedenzen bezeichnet sie `((ggT b) a) mod b`.

curry

- Die Funktion `curry` ist eine Funktion höherer Ordnung.
- Man kann sie zur Berechnung der curried Form einer binären Funktion verwenden:

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

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

curry ist polymorph

```
- curry(fn (a,b) => 2*a + b);
val it = fn : int -> int -> int

- curry(fn (a,b) => 2*a + b) 3;
val it = fn : int -> int

- curry(fn (a,b) => 2*a + b) 3 1;
val it = 7 : int
```


Umkehrung der Funktion curry: uncurry

```
- fun uncurry(f) = fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

- val uncurry = fn f => fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

uncurry ist polymorph

```
- fun f x y = 2 + x + y;
val f = fn : int -> int -> int

- fun g x y = 2.0 + x + y;
val g = fn : real -> real -> real

- uncurry f;
val it = fn : int * int -> int

- uncurry f (1,1);
val it = 4 : int
```


uncurry ist die Umkehrung von curry

```
- curry(uncurry(f));
val it = fn : int -> int -> int

- curry(uncurry(f)) 1 1;
val it = 4 : int

- uncurry(curry(uncurry(f)))(1,1);
val it = 4 : int
```

Wenn man die Funktionen `curry` und `uncurry` zum ersten Mal sieht, denkt man oft, die Typen der beiden Funktionen seien miteinander verwechselt worden. Das ist nicht der Fall!

Nicht-curried und curried Funktionen im Vergleich

- Am Beispiel einer einfachen Funktion können die Unterschiede zwischen herkömmlichen mehrstelligen Funktionen und Funktionen in curried Form erkannt werden.
- Im folgenden Beispiel wird angenommen, dass die vordefinierte Funktion „*“ den Typ `int * int -> int` hat.

Vergleich

	Nicht-curried	**curried**

mögliche Deklaration	`fun mal(x,y) = x * y`
	`val mal = (fn(x,y)=>x*y)`
		`fun mal x y = x * y`
		`val mal = (fn x y => x*y)`
		`fun mal x = (fn y => x*y)`
		`val mal =`
		`(fn x => (fn y => x*y))`

Typ	`int * int -> int`	`int -> int -> int`
Aufruf	`mal(2,3)` (hat Wert 6)	`mal 2 3` (hat Wert 6)
Unterversorgung mit Aufrufparametern	-	`mal 2` (hat eine Funktion als Wert)
	`val doppelt =` `fn y => mal(2,y)`	`val doppelt = mal 2`

Überblick

1. Prozeduren als Parameter und Wert von Prozeduren
2. Currying
3. Funktionskomposition
4. Grundlegende Funktionen höherer Ordnung
5. Beispiel: Ein Rekursionsschema zur Akkumulation

Funktionskomposition

- Die SML-Standardbibliothek enthält eine Funktion höherer Ordnung, die so definiert werden kann:

```
- infix o;
infix o

- fun (f o g)(x) = f(g(x));
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- ((fn x => 2*x) o (fn x => x+1))(3);
val it = 8 : int

- Math.sqrt o ~;
val it = fn : real -> real

- val f = Math.sqrt o ~;
val f = fn : real -> real

- f(~4.0);
val it = 2.0 : real
```

Anwendungsreihenfolge / Diagrammreihenfolge

- Die Infixfunktion ∘ leistet eine Funktionskomposition in sogenannter „*Anwendungsreihenfolge*“ (oder Funktionalreihenfolge), d.h.:

  $(f \circ g)(x) = f(g(x))$

- In der Mathematik ist auch eine Funktionskomposition in sogenannter „*Diagrammreihenfolge*“ üblich, die so definiert ist:

  $(f \circ_d g)(x) = g(f(x))$

Um Verwechslung mit dem SML-Funktionskompositionsoperator ∘ zu vermeiden, verwenden wir hier die Notation $\circ_d$ für den Funktionskompositionsoperator in Diagrammreihenfolge, der in der Mathematik üblicherweise ∘ notiert wird.

Diagrammreihenfolge

- Die Bezeichnung „*Diagrammreihenfolge*“ kommt aus der folgenden Darstellung der Komposition zweier Funktionen, wobei *f* eine Funktion von *A* in *B* und *g* eine Funktion von *B* in *C* sind:

$$A \xrightarrow{f} B \xrightarrow{g} C$$

- $(f \circ_d g)(x)$ ist eine Funktion von *A* in *C*.
- Die Reihenfolge *f* vor *g* in der Notation $(f \circ_d g)$ folgt der Reihenfolge im Diagramm, aber nicht der Reihenfolge der Funktionsanwendungen:

  $(f \circ_d g)(x) = g(f(x))$

Anwendungsreihenfolge

- Die „*Anwendungsreihenfolge*“ entspricht der Reihenfolge der Funktionsanwendungen:
  $(f \circ g)(x) = f(g(x))$
- Der Funktionskompositionsoperator ○ ist in SML vordefiniert.

Überblick

1. Prozeduren als Parameter und Wert von Prozeduren
2. Currying
3. Funktionskomposition
4. Grundlegende Funktionen höherer Ordnung
5. Beispiel: Ein Rekursionsschema zur Akkumulation

map

- Die Funktion höherer Ordnung `map` wendet eine unäre Funktion auf alle Elementen einer Liste an und liefert als Wert die Liste der (Werte dieser) Funktionsanwendungen:

```
- fun quadrat(x : int) = x * x;
val quadrat = fn : int -> int

- map quadrat [2, 3, 5];
val it = [4, 9, 25]

- map Math.sqrt [4.0, 9.0, 25.0];
val it = [2.0, 3.0, 5.0] : real list
```


Definition von `map` in SML

```
- fun map f nil = nil
    | map f (h :: t) = f(h) :: map f t;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- map (fn x:int => x*x) [1,2,3,4,5];
val it = [1,4,9,16,25] : int list
```

- Das Pattern Matching auf Funktionen in curried Form ist genauso anwendbar wie auf Funktionen mit einem Vektor als Parameter.

Warum ist `map` in curried Form definiert?

- Durch diese Form ist `map` einfach auf Listen von Listen anzuwenden.
- Im Ausdruck :

  `map (map quadrat) [[1,2], [3,4], [5,6]]`

  wird die Funktion `(map quadrat)` auf jedes (Listen-) Element der Liste `[[1,2],[3,4],[5,6]]` angewandt.

```
- map quadrat;
val it = fn : int list -> int list

- map (map quadrat) [[1,2],[3,4],[5,6]];
val it = [[1,4],[9,16],[25,36]] : int list list
```


Vorteil der curried Form am Beispiel der Funktion `map`

- Wenn `map` nicht in curried Form definiert wäre, müsste statt `(map quadrat)` ein komplizierterer Ausdruck verwendet werden, in dem die anonyme Funktion:

  `(fn L => map quadrat L)`

  vorkommt.

map´ (1)

- Sei map´ eine nicht-curried Version von map:

```
- fun    map´(f, nil) = nil
   |     map´(f, h :: t) = f(h) :: map´(f,t);
val map´ = fn : ('a -> 'b) * 'a list -> 'b list

- map´(quadrat, [2, 3, 5]);
val it = [4,9,25] : int list
```


map´ (2)

- Unter Verwendung von map´ anstelle von

  `map(map quadrat) [[1,2], [3,4], [5,6]]`

  muss der kompliziertere Ausdruck

  `map´(fn L => map´(quadrat,L),[[1,2],[3,4],[5,6]])`

  verwendet werden:

```
- map´(fn L => map´(quadrat,L),[[1,2],[3,4],[5,6]]);
val it = [[1,4],[9,16],[25,36]] : int list list
```

- Die Verwendung der anonymen Funktion

  `(fn L => map´(quadrat,L))` ist nötig, da im Gegensatz zu map die Funktion map´ zwei Parameter verlangt.

filter

- `filter` ist eine Funktion höherer Ordnung.
- `filter` filtert aus einer Liste alle Elemente heraus, die ein Prädikat erfüllen.

Beispiel

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

- filter ist_gerade [1,2,3,4,5];
val it = [2,4] : int list

- fun filter pred nil = nil
    | filter pred (h :: t) =  if pred(h)
                              then h :: filter pred t
                              else filter pred t;
val filter = fn : ('a -> bool) -> 'a list -> 'a list

- filter (fn x => (x mod 2) = 0) [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- filter (not o (fn x =>(x mod 2)=0)) [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

Warum ist `filter` in curried Form definiert?

- Durch die curried Form sind kompaktere und übersichtlichere Definitionen möglich:

```
- val gerade_elemente = filter ist_gerade;
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente = filter (not o ist_gerade);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```


Vorteil der curried Form am Beispiel der Funktion `filter`

- Sei `filter´` eine Version von `filter`, die nicht in curried Form ist:

```
fun   filter´(pred, nil) = nil
 |    filter´(pred, (h :: t)) =
            if pred(h)
            then h :: filter´(pred, t)
            else filter´(pred, t);
val filter´ = fn : ('a -> bool) * 'a list -> 'a list
```

Verwendung von `filter´`

- Unter Verwendung von `filter´` statt `filter` müssen die Funktionen `gerade_elemente` und `ungerade_elemente` so definiert werden:

```
- val gerade_elemente = fn L => filter´(ist_gerade, L);
val gerade_elemente = fn : int list -> int list

- val ungerade_elemente=fn L => filter´(not o ist_gerade,L);
val ungerade_elemente = fn : int list -> int list

- gerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list

- ungerade_elemente [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```


`foldl` und `foldr`

- Die Funktionen höherer Ordnung
  - `foldl` (zusammenfalten von links her)
  - `foldr` (zusammenfalten von rechts her)

  wenden eine Funktion auf die Elemente einer Liste folgendermaßen an:

```
foldl f z [x1, x2, ..., xn] entspricht:
f(xn,...,f(x2, f(x1, z))...)

foldr f z [x1, x2, ..., xn] entspricht:
f(x1, f(x2, ..., f(xn, z)...))
```

Beispiele

```
- foldl (op +) 0 [2,3,5];    (*entspricht 5+(3+(2+0))*)
val it = 10 : int

- foldr (op +) 0 [2,3,5];    (*entspricht 2+(3+(5+0))*)
val it = 10 : int

- foldl (op -) 0 [7,10];     (*entspricht 10-(7-0)*)
val it = 3 : int

- foldr (op -) 0 [7,10];     (*entspricht 7-(10-0)*)
val it = ~3 : int
```


Implementierung in SML

```
- fun foldl f z nil = z
    | foldl f z (x::L) = foldl f (f(x,z)) L;
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

- Im zweiten Fall von `foldl` ist die Klammerung um `f(x,z)` notwendig, weil `foldl` eine curried Funktion ist.
- Im Ausdruck `foldl f f (x,z) L` würde `foldl` auf `f` angewendet und eine Funktion liefern, die dann auf `f` angewendet würde und nicht wie beabsichtigt auf `f(x,z)`.

```
- fun foldr f z nil = z
    | foldr f z (x::L) = f(x, foldr f z L);
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Neutrales Element

- Viele binäre Funktionen haben neutrale Elemente, z.B.:
  - `0` ist das neutrale Element für `+`
  - `1` ist das neutrale Element für `*`
  - `nil` ist das neutrale Element für `@` (Listenkonkatenation)
  - `""` das neutrale Element für `^` (Stringkonkatenation).
- Die Funktionen `foldl` und `foldr` werden typischerweise so verwendet, dass das zweite Argument `z` das neutrale Element des ersten Arguments `f` ist.

Die wichtigste Anwendung von `foldl` und `foldr`

- Die wichtigste Anwendung von `foldl` und `foldr` ist die Definition von neuen Funktionen auf Listen mit Hilfe von binären Funktionen auf den Elementen der Listen:

```
- val listsum = fn L => foldl (op +) 0 L;
val listsum = fn : int list -> int

- val listprod = fn L => foldl (op * ) 1 L;
val listprod = fn : int list -> int

- val listconc = fn L => foldr (op ^) "" L;
val listconc = fn : string list -> string

- val listapp = fn L => foldr (op @) nil L;
val listapp = fn : 'a list list -> 'a list
```

Beispiele

```
- listsum [1,2];
val it = 3 : int

- listsum [1,2,3,4];
val it = 10 : int

- listprod [1,2];
val it = 2 : int

- listprod [1,2,3,4];
val it = 24 : int

- listconc ["abc", "de", "fghi", "j"];
val it = "abcdefghij" : string

- listapp [[1,2], [10,20,30], [100]];
val it = [1,2,10,20,30,100] : int list
```


Mächtige Hilfsmittel

- Man kann neue Funktionen wie `listsum`, `listprod` usw. auch explizit rekursiv definieren.
- Die Definitionen wären alle sehr ähnlich zueinander und würden sich nur an wenigen Stellen unterscheiden.
- Die Funktionen `foldl` und `foldr` sind Abstraktionen, die die Gemeinsamkeiten all dieser Definitionen darstellen und direkt zur Verfügung stellen, so dass man sie nicht bei jeder neuen Funktion wiederholen muss.
- Die Funktionen `foldl` und `foldr` sind damit sehr mächtige und grundlegende Hilfsmittel.

Kompakte Definition von `listsum`, `listprod`, `listconc`

- Da `foldl` und `foldr` curried Funktionen sind, kann man neue Funktionen mit Hilfe von `foldl` und `foldr` definieren, ohne Namen für jedes Argument der neuen Funktionen erfinden zu müssen.
- Die kompakteste (und übersichtlichste) Definition lautet:

```
- val listsum = foldl (op +) 0;
val listsum = fn : int list -> int

- val listprod = foldl (op * ) 1;
val listprod = fn : int list -> int

- val listconc = foldr (op ^) "";
val listconc = fn : string list -> string
```


Rätsel

- Welche bekannten Listenfunktionen werden wie folgt mittels `foldl` und `foldr` definiert?

```
- val a = fn L => foldl (op ::) nil L;
val a = fn : 'a list -> 'a list

- a [1,2,3];
val it = [3,2,1] : int list

- val b = fn (L1, L2) => foldr (op ::) L2 L1;
val a = fn : 'a list * 'a list -> 'a list

- b ([1,2,3],[4,5,6]) ;
val it = [1,2,3,4,5,6] : int list
```

exists

- Die Funktion höherer Ordnung `exists` überprüft, ob ein Prädikat für manche Elemente einer Liste erfüllt ist:

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

- exists ist_gerade [1,2,3,4,5];
val it = true : bool

- exists ist_gerade [1,3,5];
val it = false : bool
```


Implementierung in SML

```
- fun exists pred nil = false
    | exists pred (h::t) = (pred h) orelse exists pred t;
val exists = fn : ('a -> bool) -> 'a list -> bool
```

all

- Die Funktion höherer Ordnung `all` überprüft, ob ein Prädikat für alle Elemente einer Liste erfüllt ist:

```
- all ist_gerade [1,2,3,4,5];
val it = false : bool

- all ist_gerade [2,4];
val it = true : bool
```


Implementierung in SML

```
- fun all pred nil = true
    | all pred (h::t) = (pred h) andalso all pred t;
val all = fn : ('a -> bool) -> 'a list -> bool
```

Wiederholte Funktionsanwendung mit `repeat`

- Angewandt auf eine Funktion `f` und eine natürliche Zahl `n` ist die Funktion höherer Ordnung `repeat` eine Funktion, die `n` Mal die Funktion `f` anwendet:

```
- fun    repeat f 0 x = x
   |     repeat f n x = repeat f (n-1) (f x);
val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a

- repeat (fn x => x+1) 3 4;
val it = 7 : int

- repeat (fn x:int => x*x) 3 2;
val it = 256 : int
```


Überblick

1. Prozeduren als Parameter und Wert von Prozeduren
2. Currying
3. Funktionskomposition
4. Grundlegende Funktionen höherer Ordnung
5. Beispiel: Ein Rekursionsschema zur Akkumulation

summe

- Die Funktion höherer Ordnung `summe` entspricht dem Summenzeichen, für das in der Mathematik die Sigma-Notation (∑) üblich ist:

```
- fun summe(von,bis,schritt,funktion,akk) =
            if von > bis
            then akk
            else summe(von+schritt, bis, schritt,
                       funktion, funktion(von)+akk);
val summe = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x), 0);
val it = 10 : int
```


produkt

- Die Funktion höherer Ordnung `produkt` entspricht dem Produktzeichen, für das in der Mathematik die Pi-Notation (∏) üblich ist:

```
- fun produkt(von,bis,schritt,funktion,akk) =
            if von > bis
            then akk
            else produkt(von+schritt, bis, schritt,
                       funktion, funktion(von)*akk);
val produkt = fn : int * int * int * (int -> int) * int -> int

- produkt(1,4,1, (fn x => x), 1);
val it = 24 : int
```

Das gemeinsame Rekursionsschema (1)

- Das gemeinsame Rekursionsschema der Definitionen von `summe` und `produkt` kann unabhängig von der verwendeten arithmetischen Operation formuliert werden:

```
- fun akkumulieren(operation)(von,bis,schritt,
                              funktion,akk) =
    if von > bis
    then akk
    else akkumulieren(operation)(von+schritt,bis,
      schritt,funktion,operation(funktion(von),akk));

val akkumulieren =
   fn ('a * 'b -> 'b) -> int * int * int * (int -> 'a) * 'b -> 'b
```


Das gemeinsame Rekursionsschema (2)

```
- val summe = akkumulieren (op +);
val summe = fn : int * int * int * (int -> int) * int -> int

- val produkt = akkumulieren (op * );
val produkt = fn : int * int * int * (int -> int) * int -> int

- summe(1,4,1, (fn x => x),0);
val it = 10 : int

- produkt(1,4,1,(fn x => x),1);
val it = 24 : int
```

Rätsel

- Welche Funktion wird wie folgt mittels `akkumulieren` definiert?

```
fun a n = akkumulieren (op * )(1,n,1,(fn x => x),1);
```


Anwendung zur Integralschätzung

- Eine Funktion `real_akkumulieren` kann eingesetzt werden, um eine Schätzung des Integrals einer Funktion **R** → **R** zu definieren.
- Erinnerung:

  Das Integral einer Funktion *f* zwischen *a* und *b* kann (unter gewissen mathematischen Voraussetzungen wie der Stetigkeit von *f*) durch folgende Summe abgeschätzt werden:

$$\Delta * f(a + \Delta) + \Delta * f(a + 2\Delta) + \Delta * f(a + 3\Delta) + \ldots$$

real_akkumulieren (1)

```
- fun real_akkumulieren(operation)( von:real,
                                    bis:real,
                                    schritt:real,
                                    funktion,
                                    akk:real) =
      if von > bis
      then akk
      else real_akkumulieren(operation)(von+schritt,
      bis,schritt,funktion,operation(funktion(von),akk));

val real_akkumulieren = fn : ('a * real -> real) ->
              real * real * real * (real -> 'a) * real -> real
```


real_akkumulieren (2)

```
- fun integral(f,von,bis,delta) = real_akkumulieren
    (op +)(von+delta,bis,delta,(fn x => delta*f(x)),0.0);
val integral = fn : (real -> real) * real * real * real -> real

- integral((fn x => 1.0), 0.0, 3.0, 0.5);
val it = 3.0 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.5);
val it = 10.5 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.1);
val it = 8.7 : real

- integral((fn x => 2.0*x), 0.0, 3.0, 0.0001);
val it = 8.9997 : real
```

Beeinflussung der Schätzung durch `delta`

- Das unbestimmte Integral der Funktion $x \rightarrow 2\,x$ nach x ist die Funktion $x \rightarrow x^2$.
- Das bestimmte Integral von 0 bis 3 hat also den Wert $3^2 - 0^2 = 9$.
- Die Größe von `delta` beeinflusst, wie nahe der Schätzwert bei diesem Wert liegt.