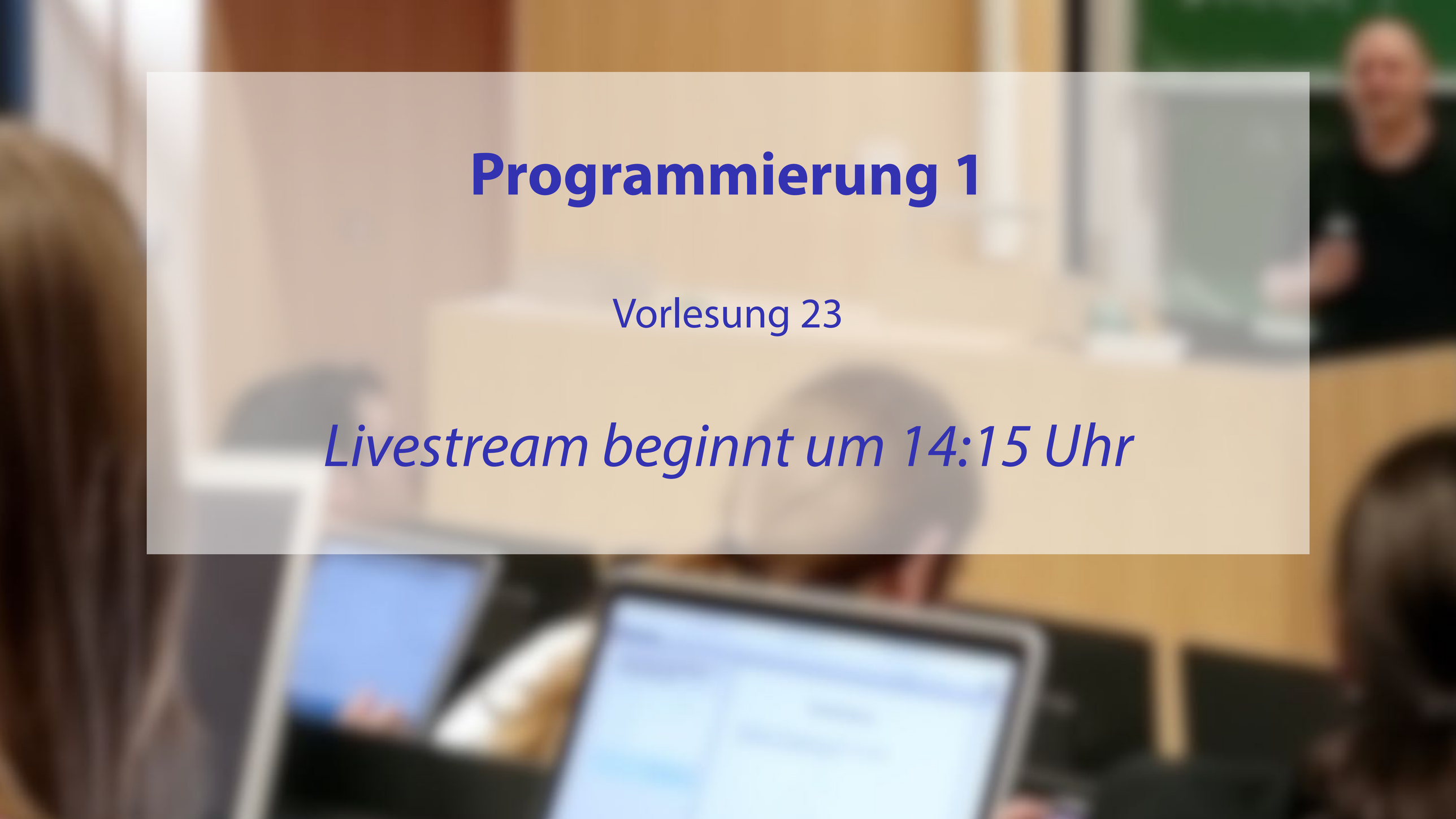


Programmierung 1

Vorlesung 23

Livestream beginnt um 14:15 Uhr

Speicher und veränderliche Objekte Teil 2

Programmierung 1

Unveränderliche und veränderliche Objekte

- **Mathematische Objekte,** wie z.B. 13, { λ *x* . *x*, λ *x* . 5 }
  sind **unveränderliche Objekte.**
- **Physikalische Objekte,** wie z.B. Uhren oder Computer,
  sind **veränderliche Objekte.**
  Sie ändern Ihren **Zustand** im Laufe der Zeit.

- Ein **Speicher** ist ein physikalisches Objekt,
  das in **Zellen** unterteilt ist.
  In jeder **Zelle** kann ein **Wert** dargestellt werden.
  Die Zellen des Speichers sind nummeriert.
  Die **Nummern** werden als **Referenzen** bezeichnet.

Zellen und Referenzen

- Für **Programme** ist der Speicher als **abstrakte Datenstruktur** sichtbar:

$eqtype\ \alpha\ ref$	**Referenztypen**
$ref\ :\ \alpha \rightarrow \alpha\ ref$	**Allokation**
$!\ :\ \alpha\ ref \rightarrow \alpha$	**Dereferenzierung**
$:=\ :\ \alpha\ ref * \alpha \rightarrow unit$	**Zuweisung**

- ***ref x*** wählt eine bisher **nicht benutzte Zelle** aus, **schreibt** die Darstellung des Wertes von *x* in die Zelle und **liefert die Referenz** der Zelle.
- ***!r*** liefert zu einer **Referenz** *r* den in der Zelle dargestellten **Wert.**
- ***r := x*** **schreibt** die Darstellung des Wertes x in eine **bereits allozierte** Zelle.

Funktionale und imperative Werte

- Ein **funktionaler Wert** beinhaltet **keine Referenzen.**
  Funktionale Werte sind **unveränderliche** Objekte.
- Ein **imperativer Wert** beinhaltet **Referenzen**.
  Imperative Werte sind **veränderliche** Objekte.
  Z.B. Referenzen, Paare von Referenzen sind imperative Werte
- Der **Zustand eines imperativen Objektes** ergibt sich aus dem **Speicherzustand.**
- Wenn die Ausführung einer Phrase den Speicherzustand verändert sprechen wir von dem **Speichereffekt.**
  - **Allokation** einer Zelle
  - **Zuweisung** eines Wertes an eine bereits allozierte Zelle
- Für Speichereffekte verwendet man oft **Prozeduren** mit Ergebnistyp **unit.**

Imperative Prozeduren

Die **imperative Prozedur** **counter** zählt in einer Speicherzelle mit, wie oft sie aufgerufen wird:

```
val r = ref 0
```
val r : int ref

```
fun counter () = (r:= !r+1 ; !r)
```
val counter : unit → int

```
counter()
```
1 : int

```
(counter(),  counter(), counter())
```
*(2, 3, 4) : int * int * int*

```
counter() + counter()
```
11 : int

Imperative Prozeduren

- Mithilfe eines **Let-Ausdrucks** kann die Prozedur die Zelle für den Zähler **einkapseln:**

```
val counter =
    let
        val r = ref 0
    in
        fn () => (r:= !r+1 ; !r)
    end
```

- Die **Prozedur** *counter i*st ein **imperativer Wert**, der die Referenz r beinhaltet.
- Die von *counter* benutzte Zelle ist nun von außen nicht mehr zugänglich.
- Der Let-Ausdruck wird nur **einmal (**bei der Ausführung der Deklaration für *counter*) **ausgeführt**!

Imperative Datenstrukturen

- Eine **funktionale Datenstruktur** ist eine Datenstruktur, deren Objekte **unveränderlich** sind.
- Eine **imperative Datenstruktur** ist eine Datenstruktur, deren Objekte **veränderlich** sind.
- **Speicherzellen** sind die Basis aller imperativen Datenstrukturen; daraus können weitere gebaut werden.
- Beispiel: **Reihungen** (engl. **arrays**)

  Eine **Reihung** ist im wesentlichen ein Vektor,
  Bei dem wir die Komponenten mittels
  Zuweisungsoperatoren verändern können.

  Wir implementieren Reihungen als
  Vektoren von Referenzen.

Reihungen

```
signature ARRAY = sig
    eqtype 'a array
    val array    : int * 'a -> 'a array
    val fromList : 'a list -> 'a array
    val sub      : 'a array * int -> 'a  (* Subscript *)
    val length   : 'a array -> int
    val foldl    : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val foldr    : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
    val app      : ('a -> unit) -> 'a array -> unit
    val update   : 'a array * int * 'a -> unit  (* Subscript *)
    val modify   : ('a -> 'a) -> 'a array -> unit
end
```

Implementierung von Reihungen durch Vektoren

```
structure Array :> ARRAY = struct
    type 'a array = 'a ref vector
    fun array (n,x) = Vector.tabulate(n, fn _ => ref x)
    fun fromList xs = Vector.fromList (map ref xs)
    fun sub (v,i) = !(Vector.sub(v,i))
    fun length v = Vector.length v
    fun foldl f s v = Vector.foldl (fn (x,a) => f(!x,a)) s v
    fun foldr f s v = Vector.foldr (fn (x,a) => f(!x,a)) s v
    fun app p v = Vector.app (fn x => p(!x)) v
    fun update (v,i,x) = Vector.sub(v,i) := x
    fun modify f a = iterup 0 (length a - 1) ()
        (fn (i,_) => update(a, i, f(sub(a,i))))
end
```

Was ist das Ergebnis von

```
let val a = Array.fromList [1,2,3]
val b = ref 0
in (Array.modify (fn x => (b:=x;!b)) a;
array2list a) end ?
```

A: [1,2,3] ✓

B: [3,2,1] ✗

C: [0,1,2] ✗

D: [2,1,0] ✗

Intervalltest mit Logbuch

interval xs s g prüft ob die Liste *xs* alle Zahlen
im Intervall von *s* bis *g* enthält.

```
fun interval xs s g = let
  val log = Array.array(g-s+1, false)
  fun test x = if s<=x andalso x<=g
               then Array.update(log, x-s, true)
               else ()
in
  List.app test xs;
  Array.foldl (fn (b,b') => b andalso b') true log
end;
```

$$val\ interval : int\ list \rightarrow int \rightarrow int \rightarrow bool$$

Reihungen

sind **vorteilhaft** bei

- Datenmengen **fester Anzahl**
- **algorithmischen Problemen** die
  - **konstante Zugriffszeit auf Komponenten** nutzen können
  - **an Ort und Stelle** (engl. **in place**) arbeiten können
  - Beispiele:
    - “in-place” **Reversion**
    - “in-place” **Sortieren**

Reversieren von Reihungen

```
fun reverse a = let
    fun swap i j =
        Array.update(a, i, #1(Array.sub(a,j),
                              Array.update(a, j, Array.sub(a,i))))
    fun reverse' l r =
        if l>=r then ()
        else (swap l r; reverse' (l+1) (r-1))
in
    reverse' 0 (Array.length a - 1)
end
```

val reverse : α array → unit

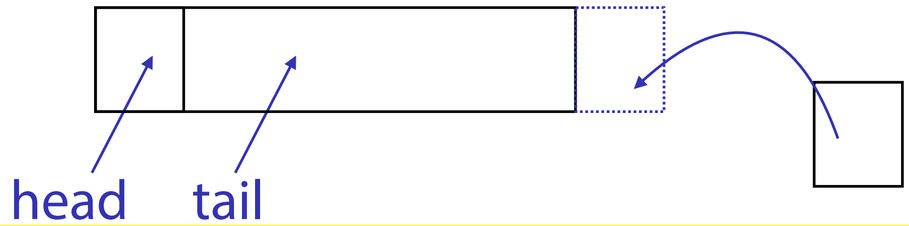
Beispiel:

	[1, 2, 3, 4, 5, 6, 7]	Vertausche Positionen 0 und 6
→	[7, 2, 3, 4, 5, 6, 1]	Vertausche Positionen 1 und 5
→	[7, 6, 3, 4, 5, 2, 1]	Vertausche Positionen 2 und 4
→	[7, 6, 5, 4, 3, 2, 1]	

Funktionale Schlangen

```
signature QUEUE = sig
    type 'a queue
    val empty  : 'a queue
    val snoc : 'a queue -> 'a -> 'a queue
    val head   : 'a queue -> 'a                (* Empty *)
    val tail   : 'a queue -> 'a queue          (* Empty *)
end

structure Queue :> QUEUE = struct
    type 'a queue = 'a list
    val empty = nil
    fun snoc q x = q@[x]
    val head = hd
    val tail = tl
end
```

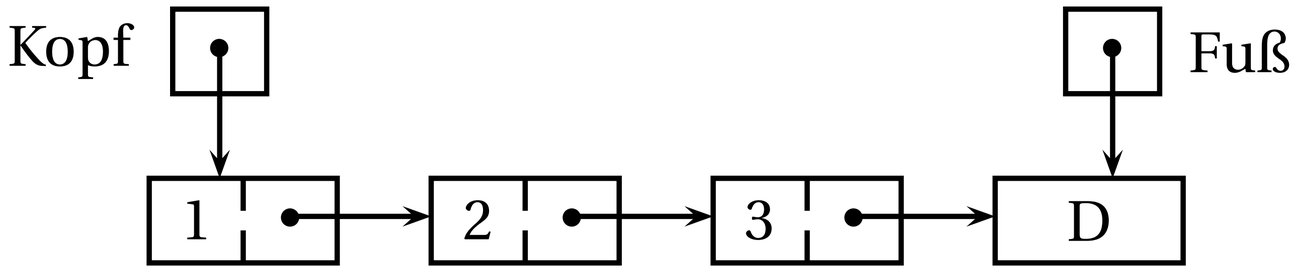


Imperative Schlangen

```
signature QUEUE = sig
    type 'a queue
    val queue : unit -> 'a queue
    val snoc : 'a queue -> 'a -> unit
    val head : 'a queue -> 'a      (* Empty *)
    val tail : 'a queue -> unit    (* Empty *)
end
```

Imperative Schlangen

- Bei **imperativen Schlangen** haben **alle Operationen** (auch einzeln betrachtet) **konstante Laufzeit.**
- Wir stellen Schlangen durch **verzeigerte Zellen** dar.
  Beispiel: Schlange mit Einträgen 1,2,3:




```
datatype 'a cell = D | E of 'a * 'a cell ref
type 'a queue = 'a cell ref ref * 'a cell ref ref
```

Imperative Schlangen

```
structure Queue :> QUEUE = struct
    datatype 'a cell = D | E of 'a * 'a cell ref
    type 'a queue = 'a entry ref ref * 'a entry ref ref

    fun queue () = let
        val dummy =  ref D
    in
        (ref dummy, ref dummy)
    end

    fun snoc (_,f) x = let
       val foot = ref D
    in
       !f:= E(x,foot) ; f:= foot
    end

    fun tail (h,_) = case !(!h) of D => raise Empty
                                 | E(x,n) => h:= n

    fun head (h,_) = case !(!h) of D => raise Empty
                                 | E(x,n) => x
end
```

Agenden

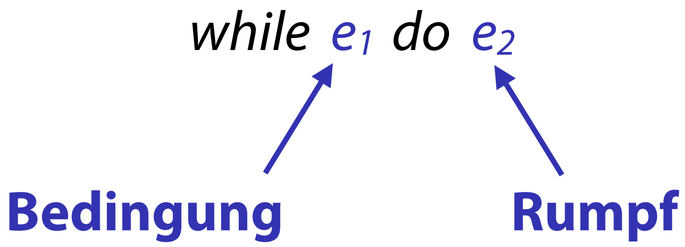
- Eine **Agenda** ist ein veränderliches Objekt, in das Werte eingetragen und wieder gelöscht werden können.
- Agenden mit **FIFO**-Strategie heißen **Schlangen**,
  Agenden mit **LIFO**-Strategie heißen **Stapel**.

$$
\begin{array}{lll}
\multicolumn{3}{l}{eqtype\ \alpha\ agenda} \\
val\ agenda & : & unit \to \alpha\ agenda \\
val\ insert & : & \alpha\ agenda \to \alpha \to unit \\
val\ remove & : & \alpha\ agenda \to unit \qquad (*\ Empty\ *) \\
val\ first & : & \alpha\ agenda \to \alpha \qquad (*\ Empty\ *) \\
val\ empty & : & \alpha\ agenda \to bool
\end{array}
$$

- *agenda* liefert eine neue Agenda, die zunächst leer ist.
- *insert* trägt einen Wert in eine Agenda ein.
- *remove* nimmt den ersten Eintrag aus einer Agenda.
- *first* liefert den Wert des ersten Eintrags einer Agenda.
- *empty* testet, ob eine Agenda leer ist.

Schleifen

Eine **Schleife** ist ein programmiersprachliches Konstrukt zur Beschreibung von **iterativen Berechnungen** die den **Speicherzustand** verändern.



Semantik: *while* e_1 *do* e_2 = *if* e_1 *then* (e_2; *while* e_1 *do* e_2)
else ()

Was ist das Ergebnis von f 4 ?

```
fun f x =
let val a = ref 1
    val i = ref 1
in while !i <= x do
  (a := !a * !i ; i := !i + 1 ); !a
end
```

A: 4 ✗

B: 6 ✗

C: 24 ✓

D: 120 ✗

Schleifen

Schleifen sind eine **abgeleitete Form,** die sich auf **Endrekursion** zurückführen lässt.

$$\text{while } e_1 \text{ do } e_2 \quad \rightsquigarrow \quad \begin{array}{l} \text{let} \\ \quad \text{fun } \mathit{loop}\ () = \text{if } e_1 \text{ then } (e_2\ ;\ \mathit{loop}\ ()) \text{ else } () \\ \text{in} \\ \quad \text{loop } () \\ \text{end} \end{array}$$

Umgekehrt lässt sich die Anwendung einer endrekursiven Prozedur auch durch eine Schleife beschreiben.

Berechnung der Potenzfunktion

imperativ

```
fun poweri x n = let
    val a = ref 1
    val i = ref 1
in
    while !i <= n do (
        a:= !a*x ;
        i:= !i+1
    );
    !a
end
```

funktional

```
fun power x n = let
    fun power' a i =
        if i<=n
        then power' (a*x) (i+1)
        else a
in
    power' 1 1
end
```

Lineare Speicher

- Eine **Halde** (engl. **heap**) ist ein linearer Speicher, mit dem wir Zellen blockweise allozieren können.
- Die Positionen der Reihung bezeichnen wir als **Adressen**.
- Wir unterscheiden zwischen **freien** und **allozierten** Zellen der Halde. **Zunächst** sind alle Zellen **frei**.

```
signature HEAP = sig
    exception Address
    exception OutOfMemory
    type address = int
    type index = int
    val new     : int -> address  (* Address, OutOfMemory *)
    val sub     : address -> index -> int  (* Address *)
    val update  : address -> index -> int -> unit  (* Address *)
    val release : address -> unit
    val show    : unit -> (address * int) list
end
```

Operationen

- ***new n*** alloziert einen **Block** der Länge $n \geqq 1$ und liefert die Adresse des Blocks (= der ersten Zelle).
  Die **relativen Positionen** der Zellen des Blocks heissen **Indizes**.
  Die Blöcke werden aufeinanderfolgend, beginnend mit der Adresse 0, in der Halde alloziert.
- ***sub a i*** liefert den Wert in der Zelle mit Adresse $a+i$
- ***update a i x*** legt den Wert x in die Zelle mit Adresse $a+i$
- ***release a*** dealloziert alle Zellen der Halde ab Adresse a.
- ***show*** liefert die Werte der allozierten Zellen als Liste.

```
structure Heap :> HEAP = struct
    val size = 1000
    val array = Array.array(size,~1)
    val lar = ref ~1  (* last allocated address *)
    exception Address
    exception OutOfMemory
    type address = int
    type index = int
    fun new n = if n<1 then raise Address
                else if !lar+n >= size then raise OutOfMemory
                     else #1(!lar+1, lar:= !lar+n)
    fun check a = if a<0 orelse a > !lar then raise Address else a
    fun sub a i = Array.sub(array, check(a+i))
    fun update a i x = Array.update(array, check(a+i), x)
    fun release a = lar:= (if a=0 then a else check a) - 1
    fun show () = iterdn (!lar) 0 nil
                  (fn (a,es) => (a, Array.sub(array,a)) :: es)
end
```

Beispiel

Die Prozedur *new'* legt eine Folge von Zahlen in einem neu allozierten Block ab:

```
fun new' xs = let
    val a = new (length xs)
in
    foldl (fn (x,i) => (update a i x ; i+1)) 0 xs ;
    a
end
```

```
new' [4, 7, 8]
```

0 : address

```
new' [~1, 2]
```

3 : address

```
show()
```

*[(0, 4), (1, 7), (2, 8), (3, ~1), (4, 2)] : (address * int) list*

www.prog1.saarland