4.11.3 Ausnahmen als Berechnungsmechanismus

```
datatype 'a List = Nil | Atom of 'a
                 | List of 'a List  * 'a List

val l = List(List(Atom 1,Atom 2),Atom 3)

exception OnEmpty
exception OnAtom
fun first l = case l of Nil => raise OnEmpty
                      | Atom _ => raise OnAtom
                      | List (first ,_) => first
first l;
val it = List (Atom 1,Atom 2) : int List
first (first (first l));
uncaught exception OnAtom raised at: stdIn:412.64-412.70
```

Ausnahmen als Berechnungsmechanismus

```
fun rest l = case l of Nil => raise OnEmpty
                     | Atom _ => raise OnAtom
                     | List (_,rest) => rest

fun atoms l =
 ((atoms (first l) handle OnEmpty => 0 | OnAtom => 1) +
  (atoms (rest l) handle OnEmpty => 0 | OnAtom => 1));

atoms (Atom 1);
val it = 2 : int

fun countAtoms l = (atoms l) div 2
countAtoms (Atom 1);
val it = 1 : int
countAtoms (List(List(Atom 1,Atom 2),Atom 3));
val it = 3 : int
```

4.12 Imperative Features

4.12.1 Referenzen

Der `ref`-Typ-Operator

Der postfixierte einstellige Typ-Operator `ref`

`ref : MT` $\mapsto$ `MT`

konstruiert ein Typ α `ref` aus einem Typ α.

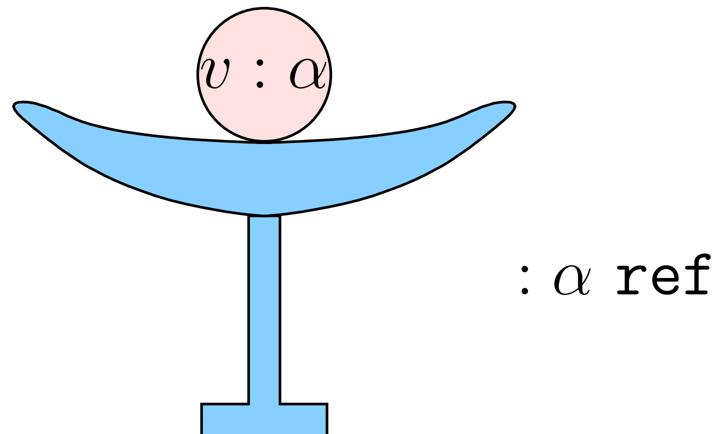
Z.B. `int ref`

`(int * bool) ref`

`int ref ref` (der Typ-Operator `ref` ist links-assoziativ)

Der `ref`-Konstruktor

- Der einzige Konstruktor des Datentyps `ref` heisst auch `ref`.
- Ein Wert vom Typ α `ref` ist ein Behälter ($\equiv$ eine Referenz) für Werte vom Typ α:




```
val p = ref 5;
val p = ref 5 : int ref
```

Der `ref`-Konstruktor

Da **`ref`** ein Konstruktor ist, kann man auf den Wert einer Referenz mit Pattern Matching zugreifen:

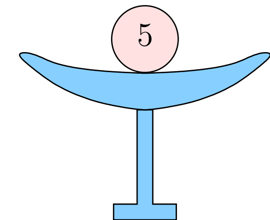
```
val p = ref 5;
val p = ref 5 : int ref
val ref x = p;
val x = 5 : int
```

Eine andere Möglichkeit ist der Dereferenzierungs-Operator **`!`**:

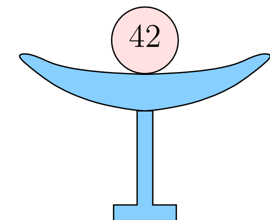
```
val x = !p;
val x = 5 : int
```

Zuweisungen

- Der
Behälter ist unveränderbar (wie alle funktionale Werte).



- Der Wert, auf
den eine Referenz zeigt, kann mit `:=` verändert werden:




```
p := 42;
val it = () : unit
p;
val it = ref 42 : int ref
```

Seiteneffekte

Das Setzen von `p` mittels `:=` ist ein Seiteneffekt und hat keinen Wert, d.h. es ergibt `()`.

```
p := 42;
val it = () : unit
op :=;
val it = fn : 'a ref * 'a -> unit
```

Gleichheit von Referenzen

Zwei Referenzen sind nur dann gleich, wenn sie derselbe Behälter (Zeiger) sind. Es genügt nicht, dass sie den gleichen Wert enthalten:

```
val p = ref 17;
val p = ref 17 : int ref
val q = ref (!p);
val q = ref 17 : int ref
p=q;
val it = false : bool
```

Gleichheit von Referenzen

```
val x=1;
val x = 1 : int
val (p,q) = (ref x,ref x);
val p = ref 1 : int ref
val q = ref 1 : int ref
p=q;
val it = false : bool
```

Gleichheit von Referenzen

```
val (p,q) = (ref (ref 1),ref (ref 2));
val p = ref (ref 1) : int ref ref
val q = ref (ref 2) : int ref ref
p := !q;
val it = () : unit
p=q;
val it = false : bool
!p = !q;
val it = true : bool
```

Sequenzen

Beim Arbeiten mit Seiteneffekten ist die Ausführungsreihenfolge wichtig.

Typische Benutzungen:

<table>
<tr><th>vor Berechnung</th><th>nach Berechnung</th></tr>
<tr><td><pre>let val _ = <effect>
    val x = <value>
in x
end</pre></td><td><pre>let val x = <value>
    val _ = <effect>
in x
end</pre></td></tr>
</table>

Abkürzung

vor Berechnung	nach Berechnung
`(<effect>; <value>)`	`<value> before <effect>`

Beispiel: Objekte mit Hilfe von Referenzen und Abschlüsse

```
val konto =
  let
    val betrag = ref 100
  in
    {einzahlen = fn b => betrag := !betrag + b,
     auszahlen = fn b => betrag := !betrag -b,
     auszug = fn () => !betrag}
  end;
val konto = auszahlen=fn,auszug=fn,einzahlen=fn
  : {auszahlen:int -> unit, auszug:unit -> int, einzahlen:int -> unit}
#einzahlen konto 500;
val it = () : unit
#auszug konto ();
val it = 600 : int
```

Beispiel: Objekte mit Hilfe von Referenzen und Abschlüsse

```
val konto =
  let
    val betrag = ref 100
  in
    {einzahlen = fn b => betrag := !betrag + b,
     auszahlen = fn b => betrag := !betrag -b,
     auszug = fn () => !betrag}
  end;
```

- Die Referenz `betrag` ist im `konto` Objekt in den funktionalen Abschlüssen eingekapselt.
- Der Betrag kann nur mit den Methoden `einzahlen` und `auszahlen` manipuliert werden.

4.12.2 Vektoren

Ein Vektor ist eine Liste fester Länge, auf deren Elemente in konstanter Zeit zugegriffen werden kann:

```
val vec = #[1,3,5,7];
val vec = #[1,3,5,7] : int vector
Vector.sub(vec,3);
val it = 7 : int
```

Wird außerhalb der Vektorgrenzen zugegriffen, wird die Exception `Subscript` geworfen:

```
Vector.sub(vec,4);
uncaught exception subscript out of bounds raised at: stdIn:1426.1-1426.11
```

Vektoren

Ein Vektor kann aus einer Liste oder oder als Wertetabelle für eine Funktion erzeugt werden:

```
Vector.fromList [1,2,3];
val it = #[1,2,3] : int vector
Vector.tabulate (6,fn x => x*x);
val it = #[0,1,4,9,16,25] : int vector
```

Ähnliche Funktionale wie bei Listen sind vordefiniert (**`map`**, **`foldl`**, **`foldr`**, u.v.m.):

```
Vector.foldri (fn (i,x,xs) => (x+i)::xs) []
                    (#[0,1,2,3],1,NONE);
val it = [2,4,6] : int list
```

4.12.3 Arrays

Vektoren kann man, wenn sie einmal erzeugt sind, nicht mehr verändern. Dafür muß man Arrays verwenden. Arrays kann man im Gegensatz zu Vektoren nicht direkt hinschreiben:

```
val arr = Array.fromList [11,12,13];
val arr = [|11,12,13|] : int array
[|11,12,13|];
stdIn:1433.2-1433.5 Error: syntax error: deleting BAR INT COMMA
```

Ähnlich wie bei Vektoren kann auf Elemente eines Arrays mit Hilfe von `Array.sub` zugreifen:

```
Array.sub(arr,2);
val it = 13 : int
```

Arrays

Zur Modifizierung der Array-Einträge benutzt man die Funktion `Array.update`:

```
(Array.update(arr,1,4); arr);
val it = [|11,4,13|] : int array
Array.update(arr,5,4);
uncaught exception subscript out of bounds raised at: stdIn:1.1-1364.2
```

Wenn man ein Array nicht mehr verändern will, kann man es in einen Vektor transformieren:

```
Array.extract(arr,0,SOME 2);
val it = #[11,4] : int vector
Array.extract(arr,0,NONE);
val it = #[11,4,13] : int vector
```

4.12.4 Ein- und Ausgabe

Die einfachste Funktion zur Bildschirmausgabe ist
`print: string -> unit`:

```
print "Palim-palim!\n";
Palim-palim!
val it = () : unit
```

Will man einen Wert ausgeben, so muß man diesen zunächst in den `string`-Typ konvertieren. Für die Basis-Typen bietet SML vordefinierte Funktionen an, z.B. `Int.toString: int -> string`:

```
print (Int.toString (7*6)^"\n");
42
val it = () : unit
```

Ein- und Ausgabe

Will man strukturierte Daten ausgeben, muß man sich selber entsprechende Funktionen schreiben.

- Erste Möglichkeit: Eine Funktion die einen String produziert benutzen:

```
datatype 'a Tree = Leaf of 'a
                 | Node of 'a Tree * 'a * 'a Tree

fun Tree2String a2String t =
 case t of Leaf a => "Leaf "^a2String a
     | Node (l,a,r) => "Node("^(Tree2String a2String l)^ ","^
                                 (a2String a)^",",
                                 (Tree2String a2String r)^")"
val Tree2String = fn : ('a -> string) -> 'a Tree -> string

fun printTree a2String = print o (Tree2String a2String)
val printTree = fn : ('a -> string) -> 'a Tree -> unit
```