Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Überblick

1. **Seiteneffekte**
   - Referenzen
   - Sequenzen
   - Vektoren
   - Arrays
   - Ein- und Ausgabe

Referenzen

- Der postfixierte **Typ-Operator** `ref`

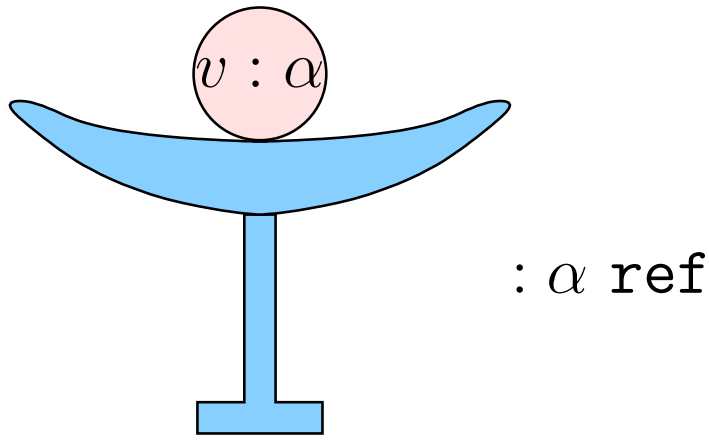
$$\texttt{ref} \; : \; \texttt{MT} \mapsto \texttt{MT}$$

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

- Z.B.
  - `int ref`
  - `(int * bool) ref`
  - `int ref ref` $\equiv$ `((int ref) ref)` (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
```

Dereferenzierung

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

Dereferenzierung = Zugriff auf den Inhalt (Wert) einer Referenz:

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

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

Dereferenzierung

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

Dereferenzierung = Zugriff auf den Inhalt (Wert) einer Referenz:

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

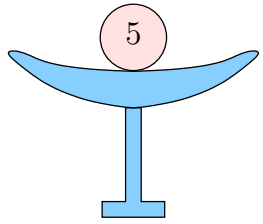
```
val ref x = p;
val x = 5 : int
```

- oder mit dem **Dereferenzierung-Operator** "!":

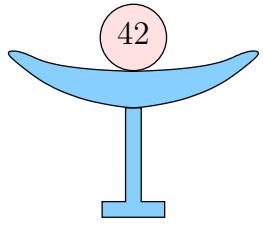
```
val x = !p;
val x = 5 : int
```

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:

vor Berechnung	nach Berechnung

`let val _ = <effect>`
`    val x = <value>`
`in x`
`end`
	`let val x = <value>`
	`    val _ = <effect>`
	`in x`
	`end`

- Abkürzung

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

Beispiel: Objekte mit Referenzen und Abschlüssen

```
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 Referenzen und Abschlüssen

```
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.

Vektoren

- Der postfixierte **Typ-Operator** `vector` : `MT` $\mapsto$ `MT`
- 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
```

Arrays

- Der postfixierte **Typ-Operator** `array` : `MT` $\mapsto$ `MT`

- Vektoren kann man, wenn sie einmal erzeugt sind, nicht mehr verändern. Dafür muß man **Arrays** verwenden.

```
val arr = Array.fromList [11,12,13];
val arr = [|11,12,13|] : int array
```

- Arrays kann man im Gegensatz zu Vektoren nicht direkt hinschreiben:

```
[|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
```

Ein- und Ausgabe

- Die einfachste Funktion zur Bildschirmausgabe ist

  `print: string -> unit`

```
print "Palim-palim world!\n";
Palim-palim world!
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 selbst entsprechende Funktionen definieren.

- Erste Möglichkeit: **toString-Funktion**

```
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
```

Ein- und Ausgabe

- Zweite Möglichkeit: direkte Ausgabe auf dem Bildschirm

```
fun printTree printA t =
 case t of Leaf a => (print "Leaf "; printA a)
         | Node(l,a,r) =>
                    (print "Node(";
                     printTree printA l;
                     print ",";
                     printA a;
                     print ",";
                     printTree printA r;
                     print ")")
val printTree = fn : ('a -> unit) -> 'a Tree -> unit
```

Ein- und Ausgabe aus Dateien

- Zur (Text-)Ein- und Ausgabe aus Dateien stellt das Modul `TextIO` eine Kollektion von Typen und Funktionen zur Verfügung:

```
type elem = char
type vector = string
type instream
type outstream

val stdIn : instream
val stdOut : outstream
val stdErr : outstream
```

Einlesen aus Text-Dateien

- Der Typ `instream` repräsentiert Dateien, aus denen man nur lesen kann. Als Sonderfall kann man auch einen String zum Lesen öffnen.

```
val openIn : string -> instream
val openString : string -> instream
val closeIn : instream -> unit
val input1 : instream -> elem option
val inputN : instream * int -> vector
val endOfStream : instream -> bool
```

Schreiben in Text-Dateien

- Ein `outstream` dagegen dient zum Schreiben:

```
val openOut : string -> outstream
val openAppend : string -> outstream
val closeOut : outstream -> unit
val output : outstream * vector -> unit
val output1 : outstream * elem -> unit
```

Überblick

2. Kontrollfluß-Manipulieren

Ausnahmen

Ausnahmen

- **Ausnahmen** (*exceptions*) sind Werte eines **vordefinierten Typs** `exn` $\in MT$.
- Konstruktoren für die Werte des Typen `exn` können vom Benutzer definiert werden:

  `exception` *AusnahmeKonstruktor* `[of` *Typ*`]`

- Beispiel:

```
exception LeereListe
exception BannedWords of string list
```

- Dadurch wurde der exn-Datentyp um zwei Ausnahmen-Konstruktoren erweitert. Das geht mit keinem anderen Datentyp!

Vordefinierte Ausnahmekonstruktoren

Div	bei Division durch Null
Empty	bei Zugriff auf eine leere Liste (`hd []`)
Match	bei unvollständigem Match in einem case-Ausdruck oder im Funktionskopf
Fail `of string`	ohne bestimmte Bedeutung, zur Benutzung durch den Programmierer

```
- 1 div 0;
uncaught exception divide by zero raised at: <stdin>

- tl (tl [1]);
uncaught exception Empty raised at: boot/list.sml:37.38-37.43
```

Der Typ `exn`

- Der Typ `exn` ist das selbe unabhängig vom Typ des eingebetteten Wertes.

```
LeereListe;
val it = LeereListe(-) : exn

- BannedWords ["viagra","rolex","medication"];
val it = BannedWords(-) : exn
```

$\Longrightarrow$ `exn` ist kein polymorpher Typ.

Ausnahmenverarbeitung

- **Ausnahmen Werfen:**
  - Eine Ausnahme `ex` kann bei der Laufzeit während einer Ausdrucksauswertung *geworfen* werden.
  - `ex` reist in die *Vergangenheit* zu den noch nicht zu Ende ausgewerteten Ausdrucksauswertungen.

- **Ausnahmen Behandeln:**
  - Eine Ausnahme `ex`, die in die Vergangenheit reist, kann abgefangen (gehandlet) werden.

Ausnahmen Werfen

raise : exn -> 'a

- kann an einer beliebigen Stelle in einem Ausdruck E vorkommen
- liefert nichts zurück
- simuliert nur einen Rückgabewert für den Wert von E
  $\Longrightarrow$ der virtuelle Rückgabewert hat den Typ von E
  $\Longrightarrow$ der Rückgabetyp von `raise` muss polymorph sein

```
1 + (raise Div);
uncaught exception divide by zero raised at: stdIn:363.12-363.15

1::(raise Div);
uncaught exception divide by zero raised at: stdIn:263.1-263.4
```

Ausnahmen Behandeln

$$
\begin{array}{rcl}
E \texttt{ handle } P_1 & \texttt{=>} & E_1 \\
\mid P_2 & \texttt{=>} & E_2 \\
\ldots & \texttt{=>} & \\
\mid P_n & \texttt{=>} & E_n
\end{array}
$$

- P_1, P_2, ..., P_n **sind Muster** ähnlich wie bei einem case Ausdruck.
- Terminiert das Auswerten von E normal, wird dessen Wert geliefert.
- Wirft das Auswerten von E eine Ausnahme `ex`, liefert der Ausdruck den Wert von E_i, wenn P_i das erste passende Muster ist.
  $\Longrightarrow$ E_i müssen den selben Typ wie E haben
- Sonst wird `ex` weitergeworfen und evt. bei der Auswertung eines umgebenden Ausdrucks (insbesondere Funktionsaufrufs) behandelt
- Das Laufzeitsystem behandelt nicht gefangene Ausnahmen.

Ausnahmen Verwenden

Ausnahmen können verwendet werden:

- zur **Fehlerbehandlung**
- als *lange Sprünge* (*longjumps*)
- als **Berechnungsmechanism**

Ausnahmen zur Fehlerbehandlung

```
fun head l = case l of nil => raise Empty | h::_ => h
fun tail l = case l of nil => raise Empty | _::r => r

fun member x l = if x=head l then true
                 else member x (tail l)
                 handle Empty => false


member 2 [1,2,3];
val it = true : bool

member 4 [1,2,3];
val it = false : bool
```

Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

Ausnahmen als Longjumps

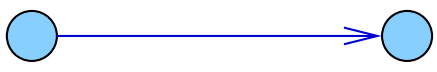
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

member x $[a, b, c]$

Ausnahmen als Longjumps

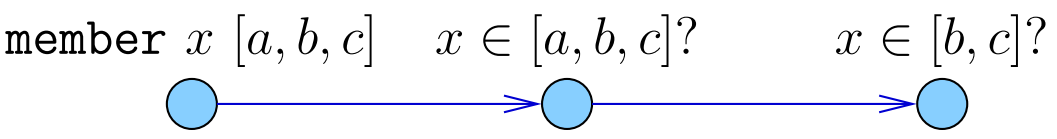
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```

member x $[a, b, c]$ $x \in [a, b, c]$?



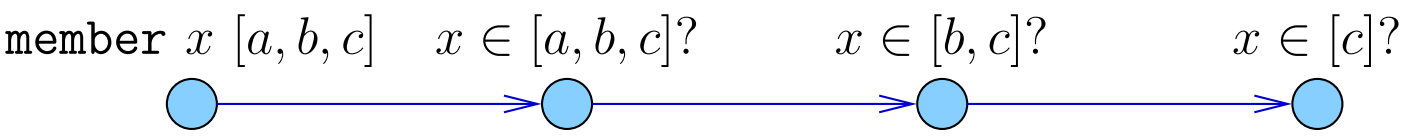
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



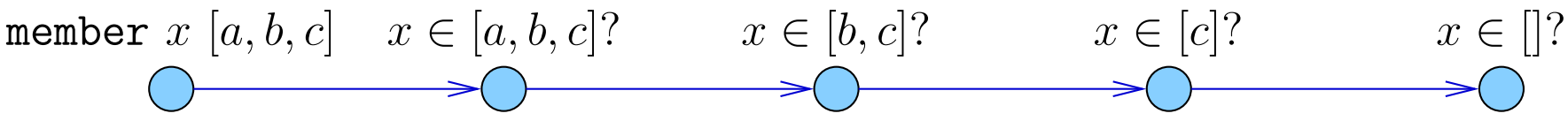
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



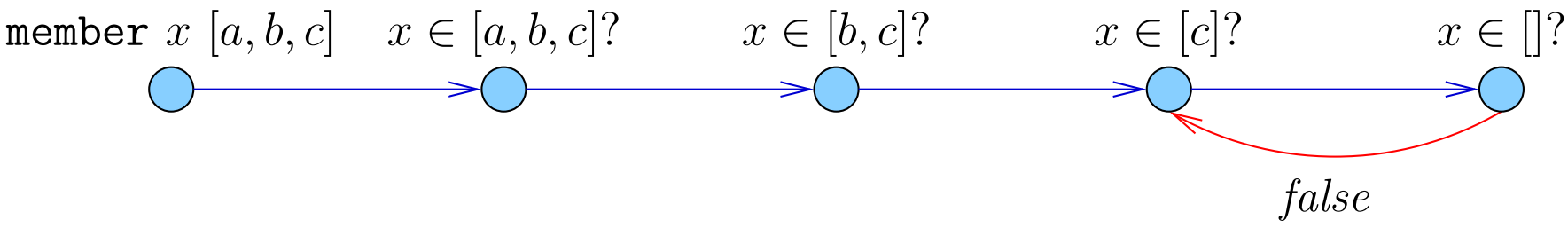
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



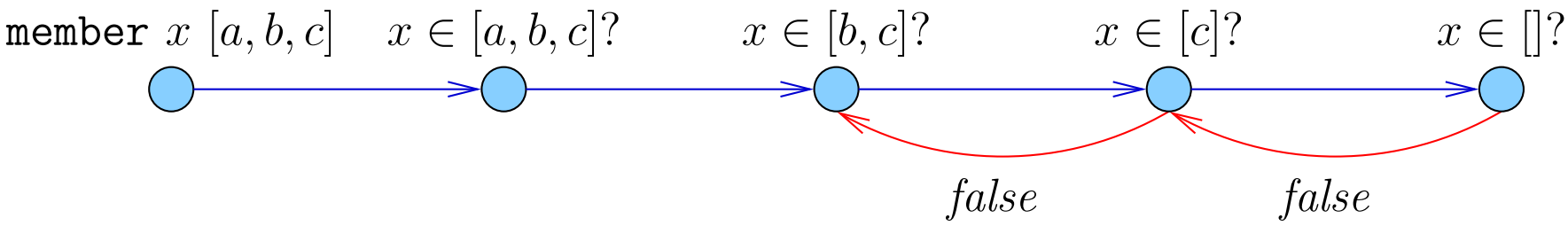
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



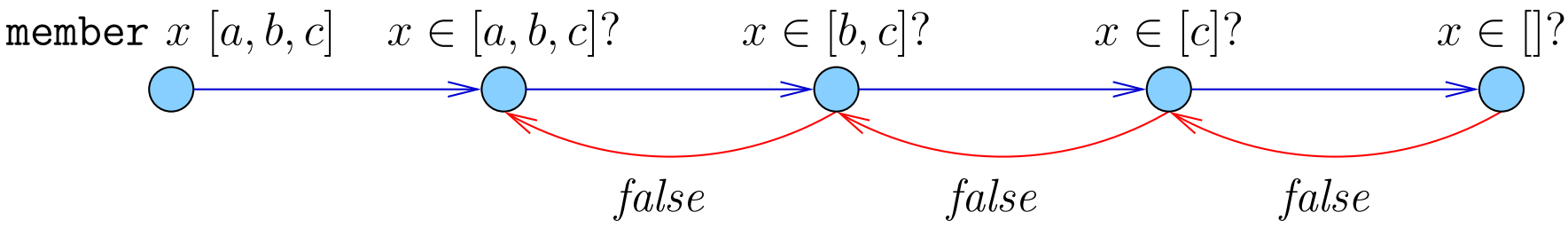
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



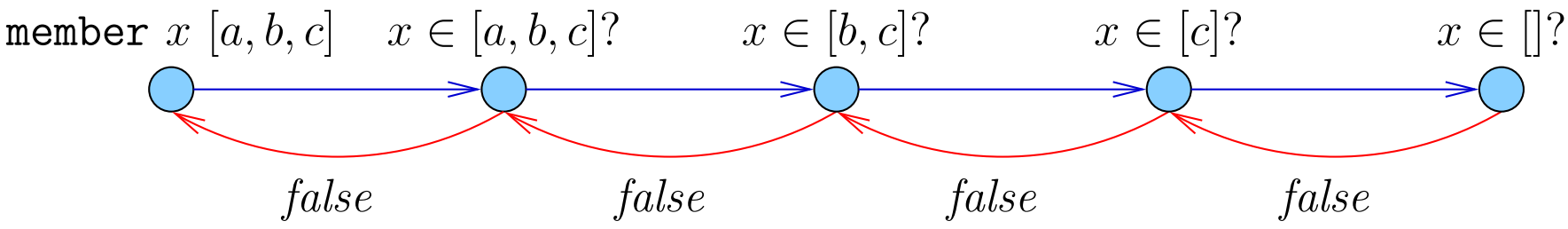
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



Ausnahmen als Longjumps

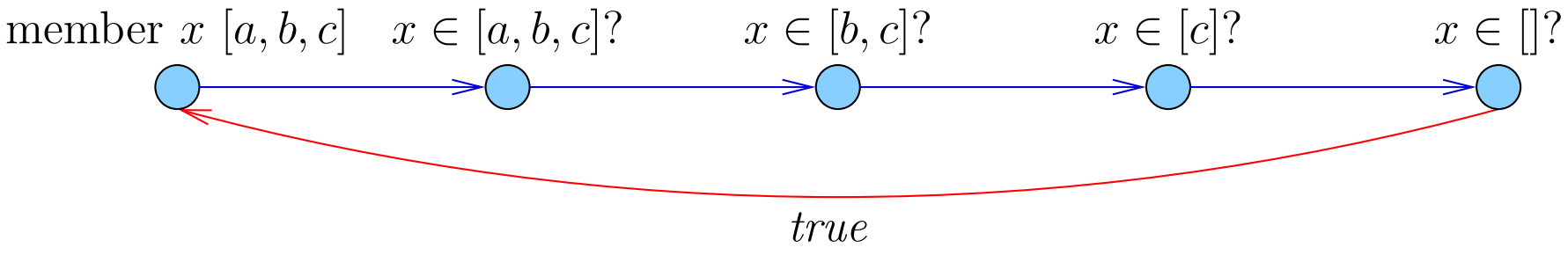
```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



Ausnahmen als Longjumps

```
exception Result of bool

fun member x list =
  let fun search l =
        case l of nil => raise (Result false)
                | h::r => if h=x then raise (Result true)
                          else search r
  in search list handle (Result r) => r
  end
```



Ausnahmen als Berechnungsmechanismus

Inhomogene Listen

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

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

Ausnahmen als Berechnungsmechanismus

Inhomogene Listen

```
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
```