Proseminar Programmiersysteme

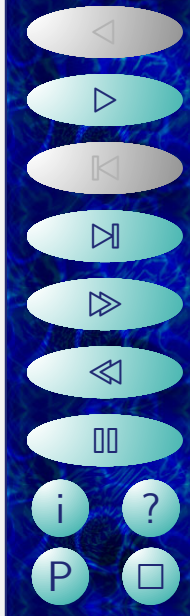
Lazy Programming

Wintersemester 2003 / 2004

Author
Benedikt Grundmann
bgrund@ps.uni-sb.de

Betreuer
Andreas Rossberg
rossberg@ps.uni-sb.de

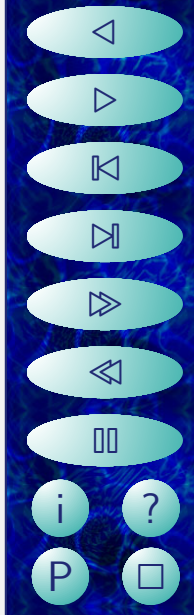
Professor
Gert Smolka
smolka@ps.uni-sb.de



Überblick

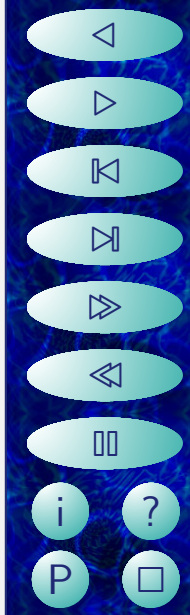
⇒ Haskell Geschichte + Überblick

- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



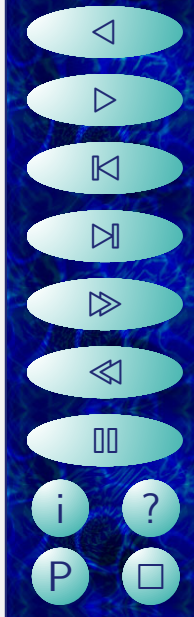
Haskell – Geschichte

- entwickelt in den späten 80er
- erste vollständige Definition (1990 – Haskell 1.0)
- entwickelte sich konstant weiter
  - Haskell 1.1 (August 1991)
  - Haskell 1.2 (März 1992)
  - Haskell 1.3 (Mai 1996)
  - Haskell 1.4 (April 1997)
- 1999 standardisiert als Haskell 98
- wird immer noch weiterentwickelt
  - Polymorphismus höherer Ordnung
  - Typklassen mit mehr als einem Parameter
  - ...



Haskell – Überblick

- reine funktionale Programmiersprache (keine Seiteneffekte)
- Bedarfsgesteuerte Auswertung mit Memoisation (lazy evaluation)
- polymorphes Typsystem mit Typinferenz und
- überladenen Bezeichnern basierend auf Typklassen
- Listenbeschreibungen (list comprehensions)

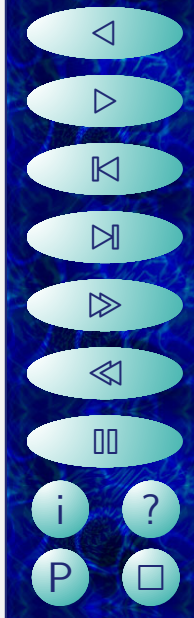


Überblick

- Haskell Geschichte + Überblick

⇒ Evaluierungsmodelle: Call by value, Call by name, Call by need

- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

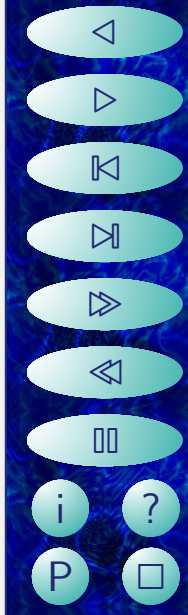
Definition

```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten



Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

Definition

```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten

```
5 * 5 + x + x
```

Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

Definition

```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten

```
5 * 5 + x + x  →  5 * 5 + 12 + 12
```

Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

Definition

```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten

```
5 * 5 + x + x  →  5 * 5 + 12 + 12
               →  25 + 12 + 12
```

Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

Definition

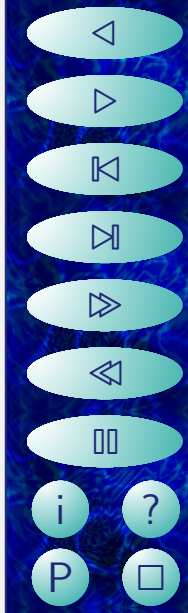
```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten

```
5 * 5 + x + x  →  5 * 5 + 12 + 12
               →  25 + 12 + 12
               →  37 + 12
```



Evaluierung 1. – Call by value

Erinnerung (Programmieren 1, Info 1) an Auswertungsregel in SML (call by value):

Definition

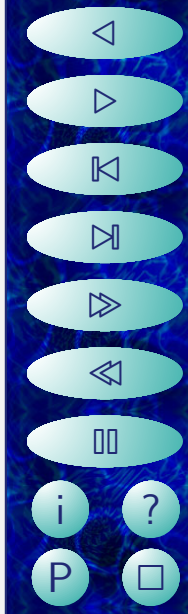
```
x = 3 * 4
```

Ausdruck

```
5 * 5 + x + x
```

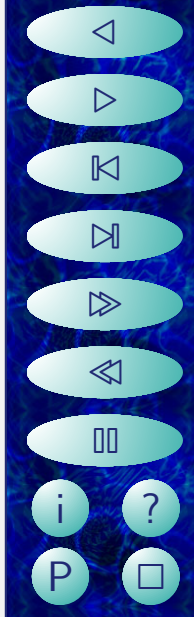
1. Bezeichner durch Werte ersetzen
2. Ausdruck von innen nach aussen und
3. von links nach rechts auswerten

```
5 * 5 + x + x  →  5 * 5 + 12 + 12
               →  25 + 12 + 12
               →  37 + 12
               →  49
```



Evaluierung 2. – Call by name

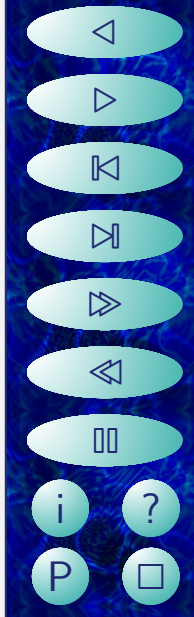
1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

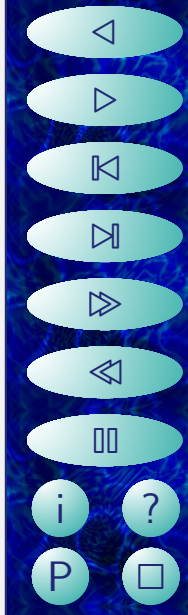
```
5 * 5 + x + x
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

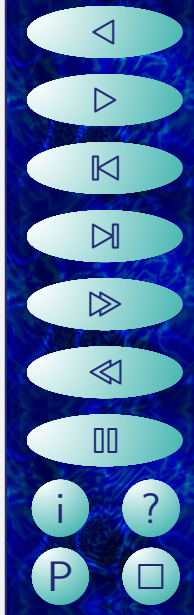
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

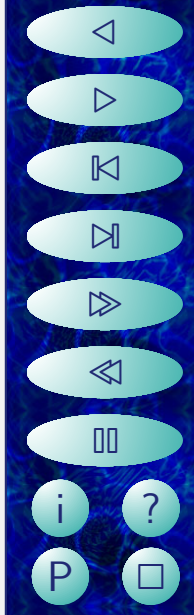
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

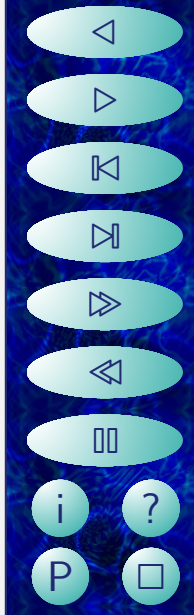
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

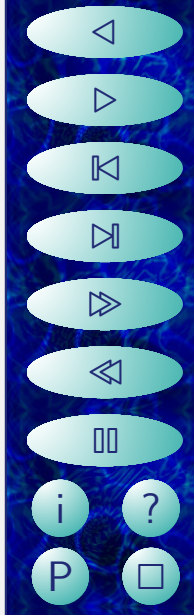
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
                 →   37 + 12
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

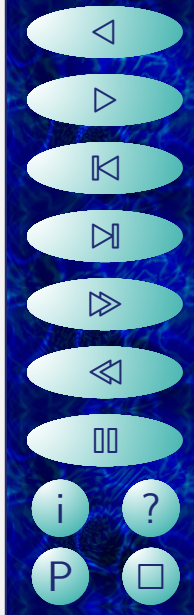
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
                 →   37 + 12
                 →   49
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

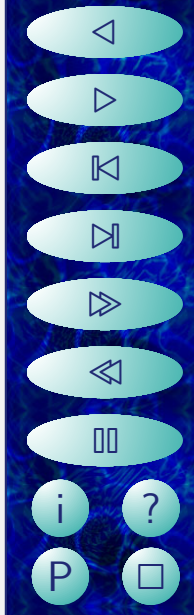
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
                 →   37 + 12
                 →   49
```



Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
                 →   37 + 12
                 →   49
```

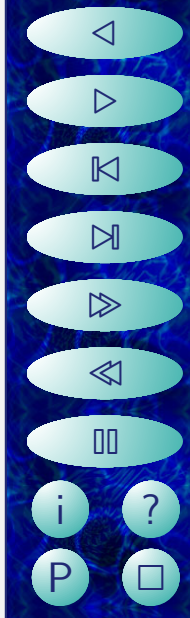


Evaluierung 2. – Call by name

1. Bezeichner durch *Ausdruck* ersetzen
2. Ausdrücke von aussen nach innen und
3. von links nach rechts auswerten

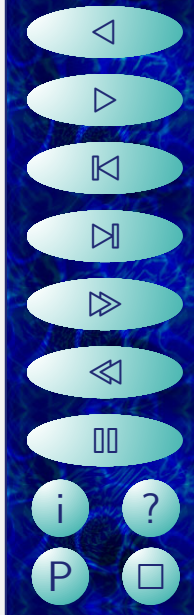
```
5 * 5 + x + x    →   25 + (3 * 4) + (3 * 4)
                 →   25 + 12 + (3 * 4)
                 →   37 + (3 * 4)
                 →   37 + 12
                 →   49
```

- Vorteil: Berechnung nur wenn wirklich nötig
- Nachteil: Berechnung unter Umständen *mehrfach*!



Evaluierung 3. – Call by need

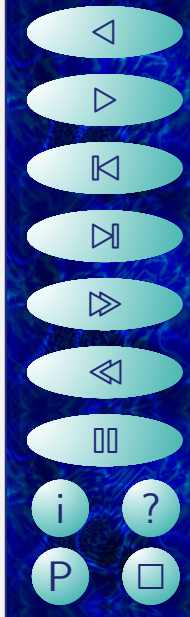
Verbesserung: Ausgewertete Ausdrücke merken $\Rightarrow$ nur *einmal* berechnen.



Evaluierung 3. – Call by need

Verbesserung: Ausgewertete Ausdrücke merken ⇒ nur *einmal* berechnen.

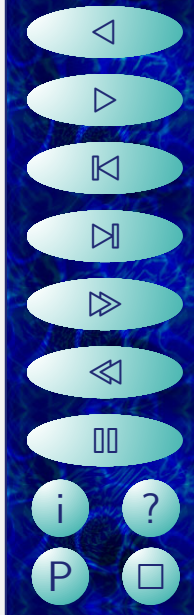
```
5 * 5 + x + x where x = 3 * 4
```



Evaluierung 3. – Call by need

Verbesserung: Ausgewertete Ausdrücke merken $\Rightarrow$ nur *einmal* berechnen.

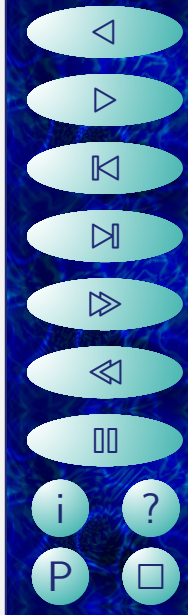
```
5 * 5 + x + x where x = 3 * 4   →   25 + x + x where x = 12
```



Evaluierung 3. – Call by need

Verbesserung: Ausgewertete Ausdrücke merken ⇒ nur *einmal* berechnen.

```
5 * 5 + x + x where x = 3 * 4   →   25 + x + x where x = 12
                                →   25 + 12 + 12
```



Evaluierung 3. – Call by need

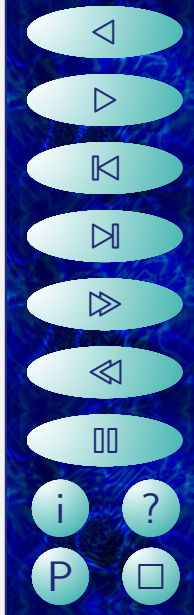
Verbesserung: Ausgewertete Ausdrücke merken $\Rightarrow$ nur *einmal* berechnen.

```
5 * 5 + x + x where x = 3 * 4   →   25 + x + x where x = 12
                                →   25 + 12 + 12
                                →   37 + 12
```

Evaluierung 3. – Call by need

Verbesserung: Ausgewertete Ausdrücke merken ⇒ nur *einmal* berechnen.

```
5 * 5 + x + x where x = 3 * 4   →   25 + x + x where x = 12
                                →   25 + 12 + 12
                                →   37 + 12
                                →   49
```

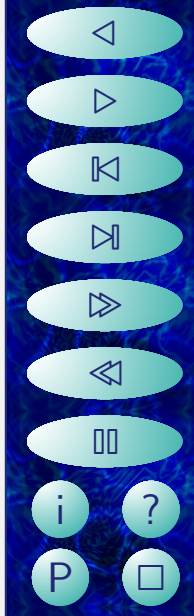


Evaluierung 3. – Call by need

Verbesserung: Ausgewertete Ausdrücke merken ⇒ nur *einmal* berechnen.

```
5 * 5 + x + x where x = 3 * 4   →   25 + x + x where x = 12
                                →   25 + 12 + 12
                                →   37 + 12
                                →   49
```

Auswertung somit *wirklich* “lazy”.

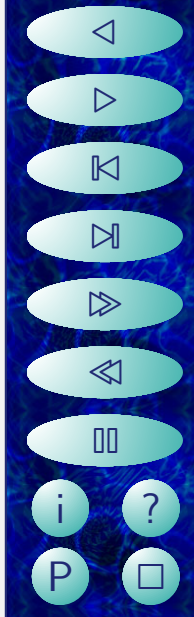


Beispiele

Definition

```
my_if True  a b = a
my_if False a b = b

switch []                 = error "switch failed"
switch ((True, e) : _)    = e
switch ((False, _) : l)   = switch l
```

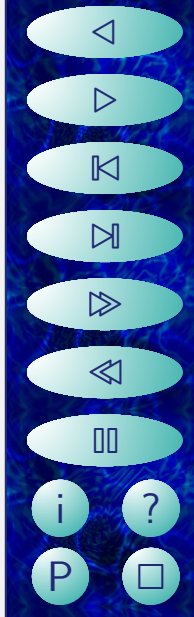


Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need

$\Rightarrow$ Definition: strikte Funktionen

- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



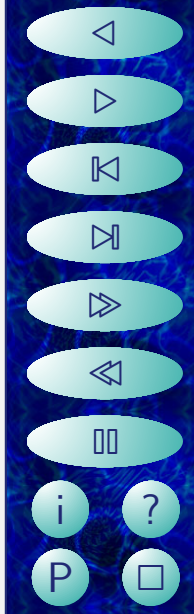
Theorie

Definition:

Für jede Funktion f gilt:

$$f \text{ ist } \textit{strikt} \Leftrightarrow f\perp = \perp$$

wobei $\perp$ (bottom) = Wert divergierender Ausdrücke



Theorie

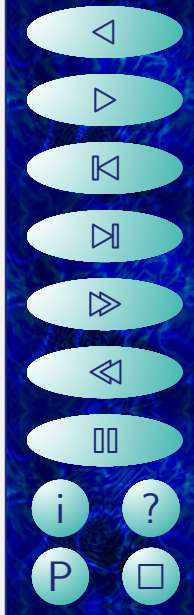
Definition:

Für jede Funktion f gilt:

$$f \text{ ist } \textit{strikt} \Leftrightarrow f \perp = \perp$$

wobei $\perp$ (bottom) = Wert divergierender Ausdrücke

Bedarfsgesteuerte Programmiersprachen haben *nicht* strikte Funktionen



Theorie

Definition:

Für jede Funktion f gilt:

$$f \text{ ist } \textit{strikt} \Leftrightarrow f \perp = \perp$$

wobei $\perp$ (bottom) = Wert divergierender Ausdrücke

Bedarfsgesteuerte Programmiersprachen haben *nicht* strikte Funktionen

Beispiel:

Definitionen

```
bot = bot
f x = 42
```

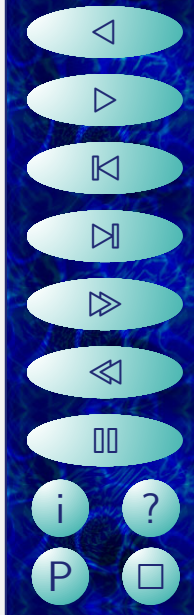
Ausdruck

```
f bot ⟹ 3
```

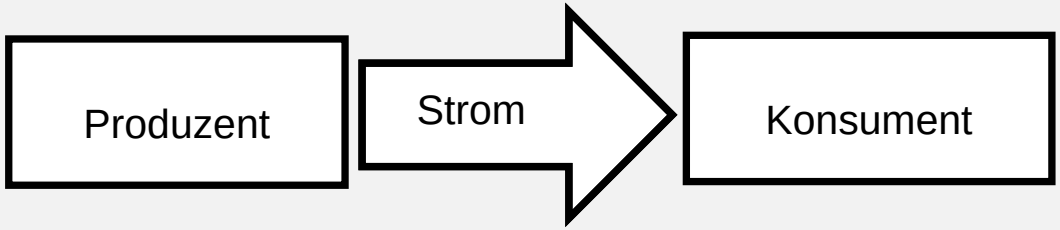
Unendliche Datenstrukturen

- unendliche Datenstrukturen möglich
  - Listen
  - Bäume
  - ...
- unendliche Listen (Ströme) sind besonders nützlich

$\Rightarrow$ neue Arten der Codestrukturierung



Strukturierung durch Ströme



Beispiele für Produzenten

Ein Strom von Einsen

```
ones = 1 : ones
```

Die natürlichen Zahlen

```
enum n = n : enum (n + 1)
nats = enum 1
```

syntaktischer Zucker: `nats = [1 .. ]`

Die Fibonacci Zahlen in linearer Zeit

```
fibs = 0 : 1 : (zipWith (+) fibs (tail fibs))
```

Ströme: Teil 2

Im allgemeinen:

Das Sieb des Eratosthenes in 2 Zeilen Haskell.

```
primes          = sieve [2 .. ]
sieve (x:xs)    = x : sieve [ y | y <- xs, y `mod` x > 0]
```

Ströme: Teil 3

Ein Strom von Annäherungen an die Wurzel einer Zahl `n`.

```
approxSqrt a0 n = iterate (next n) a0
  where
  next n x = (x + n/x) / 2
```

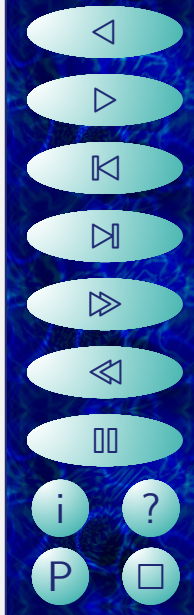
Beispiel

```
approxSqrt (81/2) 81 = [8.0,5.0,4.1,...,4.0,4.0,...]
```

Konsument entscheidet wann Annäherung gut genug

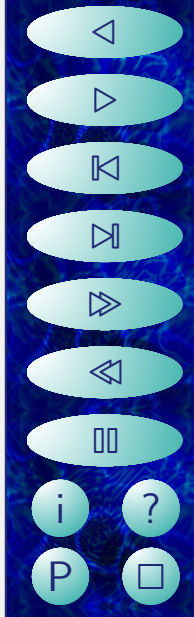
```
within eps (a : b : rest)
  | abs (a - b) <= eps = b
  | otherwise          = within eps (b : rest)

sqrt n = within 0.0001 (approxSqrt (n/2) n)
```



Ströme: Zusammenfassung

- Struktur: Produzent → Transformator → Konsument
- Einsatzmöglichkeiten
  - Numerik: Berechnung immer besserer Annäherungen
  - AI: mögliche Züge in einem Spiel
  - Fakten in einem Logikinterpreter
  - ...
- benötig bedarfsgesteuerte Auswertung
- Vorteil: Trennung des Programmes in unabhängige Teile

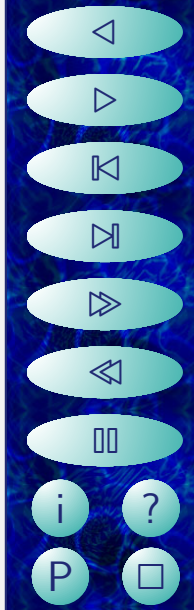


Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme

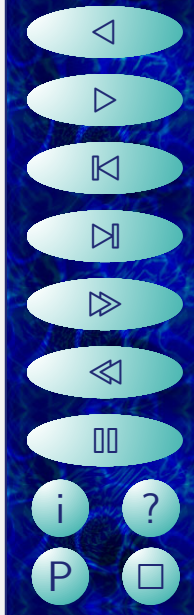
$\Rightarrow$ Seiteneffekte

  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



Aber wenn . . .

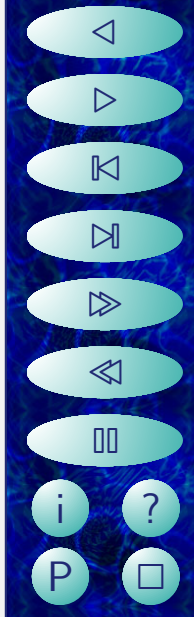
. . . ich gar nicht weiß wann ein Ausdruck ausgewertet wird, führt das dann nicht zu Problemen?



Aber wenn . . .

. . . ich gar nicht weiß wann ein Ausdruck ausgewertet wird, führt das dann nicht zu Problemen?

Antwort: Ja aber nur wenn der Ausdruck nicht nur etwas berechnet, sondern auch etwas verändert (Seiteneffekte).



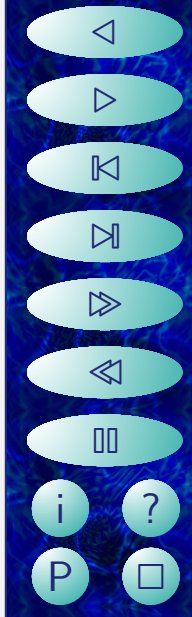
Aber wenn . . .

. . . ich gar nicht weiß wann ein Ausdruck ausgewertet wird, führt das dann nicht zu Problemen?

Antwort: Ja aber nur wenn der Ausdruck nicht nur etwas berechnet, sondern auch etwas verändert (Seiteneffekte).

Beispiel:

```
a      = inputInt ()
b      = inputInt ()
result = a + b
```



Aber wenn . . .

. . . ich gar nicht weiß wann ein Ausdruck ausgewertet wird, führt das dann nicht zu Problemen?

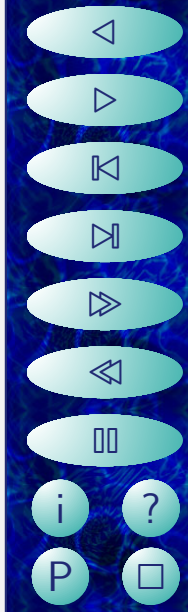
Antwort: Ja aber nur wenn der Ausdruck nicht nur etwas berechnet, sondern auch etwas verändert (Seiteneffekte).

Beispiel:

```
a      = inputInt ()
b      = inputInt ()
result = a + b
```

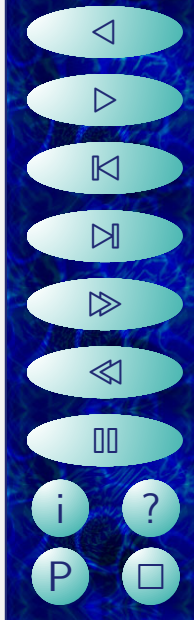
Auch das Verändern einer Variable ist ein Seiteneffekt! In Haskell nicht möglich.

```
a   =   ref 3
b   =   !a
a   :=  !a + 1
b   =   ?
```



Konsequenz

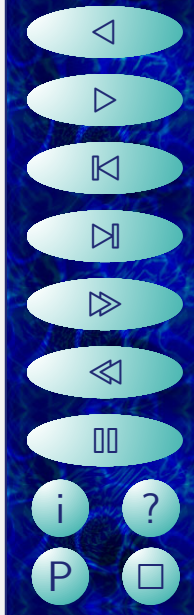
Keine Seiteneffekte



Konsequenz

Keine Seiteneffekte

Definition: Sprachen ohne Seiteneffekte heißen rein/pur funktional



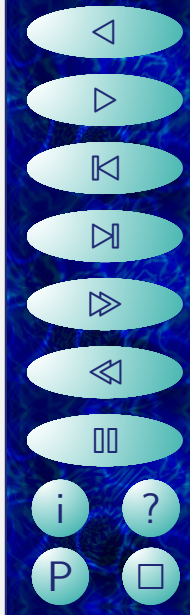
Konsequenz

Keine Seiteneffekte

Definition: Sprachen ohne Seiteneffekte heißen rein/pur funktional

Wichtige Vertreter (neben Haskell):

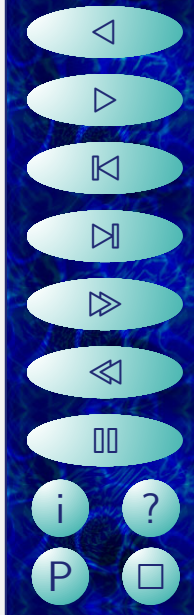
- Miranda entwickelt 1985 (kommerziell)
- Clean entwickelt 1987



Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte

  $\Rightarrow$ Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



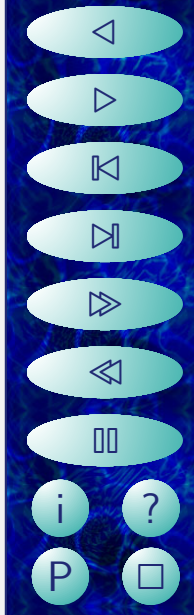
Referentielle Transparenz

In Sprachen ohne Seiteneffekte gilt:

- alle Ausdrücke haben einen Wert (evtl. $\perp$)
- ein Bezeichner kann immer durch den von ihm definierten Wert ersetzt werden
- der Wert eines Ausdruckes ist nur von den Werten seiner Teilausdrücke abhängig

Solche Sprachen nennt man

referentiell transparent



Referentielle Transparenz

In Sprachen ohne Seiteneffekte gilt:

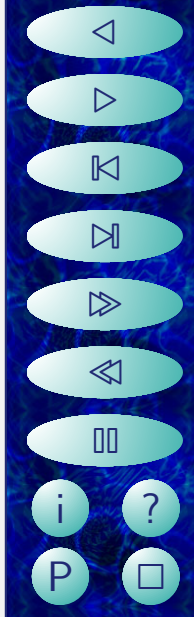
- alle Ausdrücke haben einen Wert (evtl. $\perp$)
- ein Bezeichner kann immer durch den von ihm definierten Wert ersetzt werden
- der Wert eines Ausdruckes ist nur von den Werten seiner Teilausdrücke abhängig

Solche Sprachen nennt man

referentiell transparent

In nicht strikten Programmiersprachen gilt außerdem:

- ein Bezeichner kann immer durch den von ihm definierten *Ausdruck* ersetzt werden
- die Reihenfolge der Auswertung der Teilausdrücke ist irrelevant



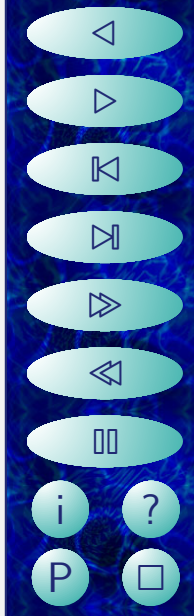
Vorteile der referentiellen Transparenz

- einzelne Funktionen können unabhängig vom Rest eines Programmes analysiert werden

$\Rightarrow$ einfachere Programmverifikation

- Äquivalenz von Ausdrücken ist einfacher beweisbar

$\Rightarrow$ Programmtransformationen möglich (z. Bsp. Optimierung durch Compiler)

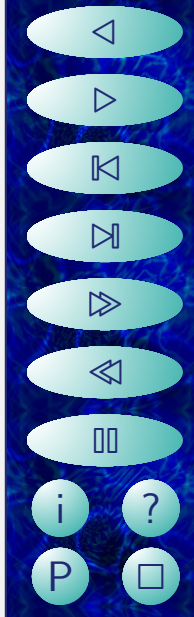


Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz

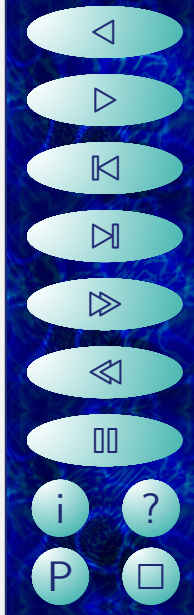
$\Rightarrow$ Seiteneffekte und Aktionen in puren funktionalen Sprachen

  - Zustand durchfädeln
  - Monade
- Laziness in strikten Sprachen



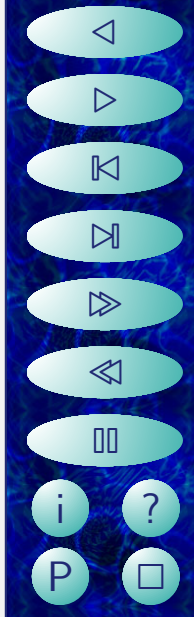
Aktionen und Seiteneffekte

- Seiteneffekte manchmal nötig
  - Zustand verändern (Zuweisung)
  - Ein- und Ausgabe
  - GUI
  - Ausnahmenbehandlung



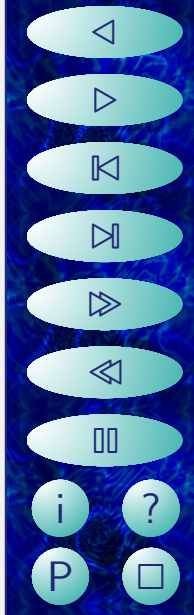
Aktionen und Seiteneffekte

- Seiteneffekte manchmal nötig
  - Zustand verändern (Zuweisung)
  - Ein- und Ausgabe
  - GUI
  - Ausnahmenbehandlung
- Wie können wir Zustand ausdrücken?



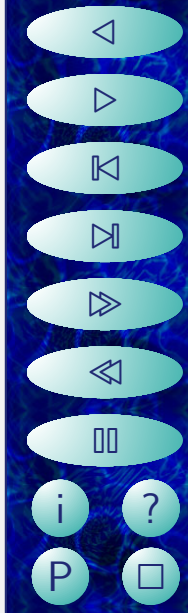
Aktionen und Seiteneffekte

- Seiteneffekte manchmal nötig
  - Zustand verändern (Zuweisung)
  - Ein- und Ausgabe
  - GUI
  - Ausnahmenbehandlung
- Wie können wir Zustand ausdrücken?
- ohne Seiteneffekte einzuführen?



Aktionen und Seiteneffekte

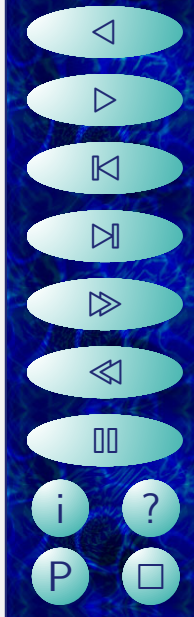
- Seiteneffekte manchmal nötig
  - Zustand verändern (Zuweisung)
  - Ein- und Ausgabe
  - GUI
  - Ausnahmenbehandlung
- Wie können wir Zustand ausdrücken?
- ohne Seiteneffekte einzuführen?
  - Einfach: Zustand durchfädeln



Zustand “durchfädeln”

Imperativ (C)

```
int fac (int n) {
    int i, res = 1;
    for (i = 1; i <= n; i++) res *= i;
    return res;
}
```

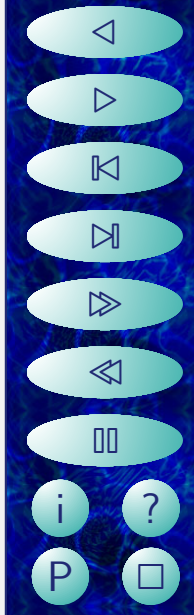


Zustand “durchfädeln”

Imperativ (C)

```
int fac (int n) {
    int i, res = 1;
    for (i = 1; i <= n; i++) res *= i;
    return res;
}
```

Kann in funktionalen Sprachen durch “Durchfädeln” des Zustandes erreicht werden.



Zustand “durchfädeln”

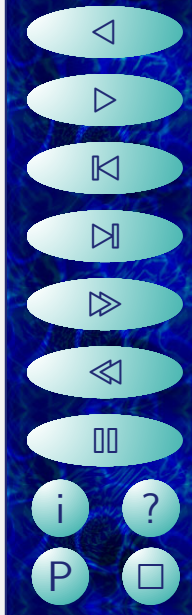
Imperativ (C)

```
int fac (int n) {
    int i, res = 1;
    for (i = 1; i <= n; i++) res *= i;
    return res;
}
```

Kann in funktionalen Sprachen durch “Durchfädeln” des Zustandes erreicht werden.

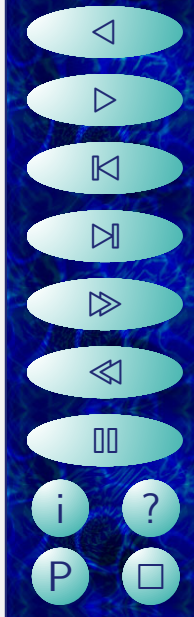
Funktional (Haskell)

```
fac n = fac' 1 1
where
fac' i res = if i <= n then fac' (i + 1) (res * i)
             else res
```



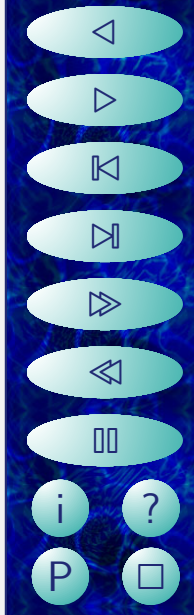
Die Welt “durchfädeln”: Ein- und Ausgabe

- Zustand können wir ausdrücken.
- Können wir etwas ähnliches auch für E/A machen?



Die Welt “durchfädeln”: Ein- und Ausgabe

- Zustand können wir ausdrücken.
- Können wir etwas ähnliches auch für E/A machen?
- Ja wir können! Ein Programm das E/A macht verändert einfach die Welt!



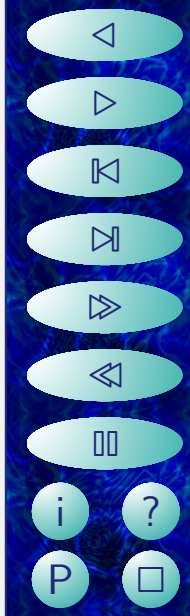
Die Welt “durchfädeln”: Ein- und Ausgabe

- Zustand können wir ausdrücken.
- Können wir etwas ähnliches auch für E/A machen?
- Ja wir können! Ein Programm das E/A macht verändert einfach die Welt!

Beispiel (PSEUDO-Haskell)

```
inputLine :: World -> (String, World)
print :: String -> World -> World

main :: World -> World
main world =
  let world'           = print "Enter your name" world
      (name, world'') = inputLine world'
  in print "Hello " ++ name world''
```

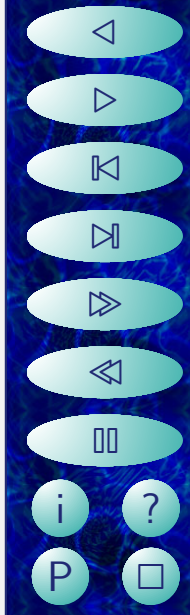


Probleme

Was passiert in folgendem Program?

PSEUDO-Haskell

```
main :: World -> World
main world =
  let _       = print "a" world
  in print "b" world
```



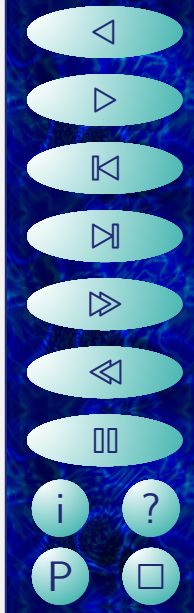
Probleme

Was passiert in folgendem Program?

PSEUDO-Haskell

```
main :: World -> World
main world =
  let _      = print "a" world
  in print "b" world
```

- Problem: Es darf *immer* nur einen Zustand der Welt geben!



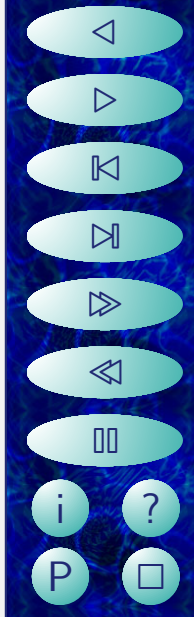
Probleme

Was passiert in folgendem Program?

PSEUDO-Haskell

```
main :: World -> World
main world =
  let _      = print "a" world
  in print "b" world
```

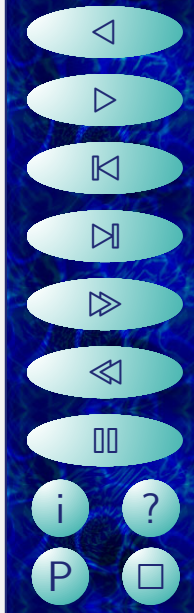
- Problem: Es darf *immer* nur einen Zustand der Welt geben!
- Lösung: IO Monad



Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln

  $\Rightarrow$ Monade
- Laziness in strikten Sprachen



Erst die Theorie

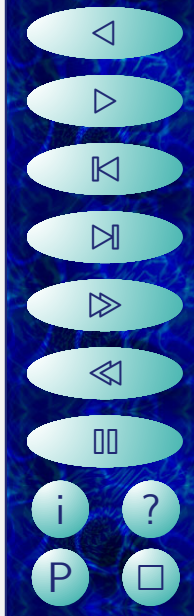
Monad: Typkonstruktor auf den die Funktionen `return` und `>>=` (bind) definiert sind.

Definitionen

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  ...
```

Die drei Regeln genügen müssen:

- `(return x >>= f) == (f x)`
- `(m >>= return) == m`
- `((m >>= f) >>= g) == (m >>= \x -> f x >>= g)`



Was gibt es für Monade?

`IO a`

1. führt Ein-/Ausgabe Aktion durch (z.Bsp Lesen oder Schreiben einer Datei)
2. gibt danach Wert vom Typ `a` zurück.

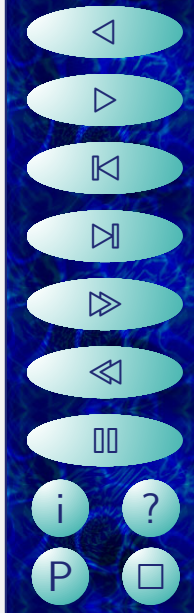
`GUI a`

1. führt eine GUI Aktion durch (z.Bsp Popupmenu zeichnen) und
2. gibt danach einen Wert vom Typ `a` zurück.

`State t a`

1. führt eine Aktion aus die auf eine Variable vom Typ `t` zugreift
2. gibt danach einen Wert vom Typ `a` zurück

...



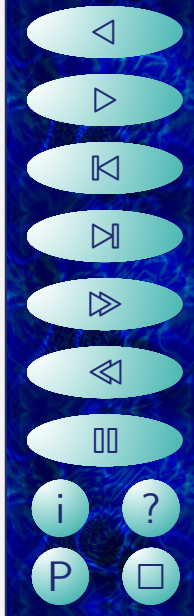
Beispiel

Definition

```
readLine =
  getChar >>=                    -- lese ein Zeichen
  \c -> if c == '\n'             -- wenn neue Zeile
          then return ""         -- dann leeren String
          else                   -- sonst das zeichen und
            readLine >>= \l -> return (c : l)  -- rest der Zeile
```

Hier sehen wir folgendes:

- `return :: a -> M a` nimmt einen wert und "steckt" ihn in einen Monad `M`. In unserem Fall `IO`.
- `>>= :: M a -> (a -> M b) -> M b` kombiniert Monade zu neuen Monaden
- `readLine :: IO String` ist ein Monad – *keine* Funktion!



bind aka >>=

Type

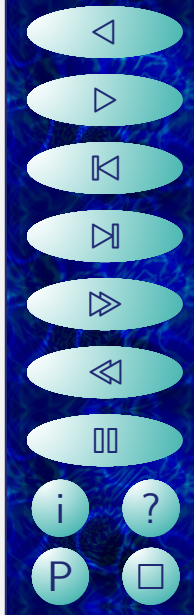
```
>>= :: M a -> (a -> M b) -> M b
```

Argument 1 Monad `m`

Argument 2 Prozedur `f` von Wert nach Monad

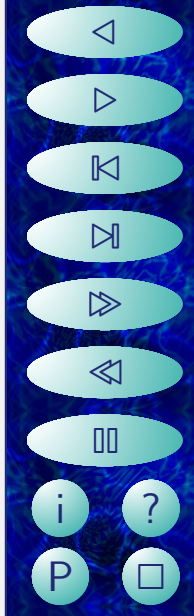
Resultat Monad der (sofern er ausgewertet wird):

- Monad `m` auswertet
- Wert `w` aus `m` nimmt
- `f w` auswertet



Syntaktischer Zucker: do

```
do x <- action                    action >>=
   y <- action2 x                   ( \x -> action2 x >>=
   return (f y)                     ( \y -> return (f y) ) )
```



Syntaktischer Zucker: do

```
do x <- action                  action >>=
   y <- action2 x                 ( \x -> action2 x >>=
   return (f y)                   ( \y -> return (f y) ) )
```

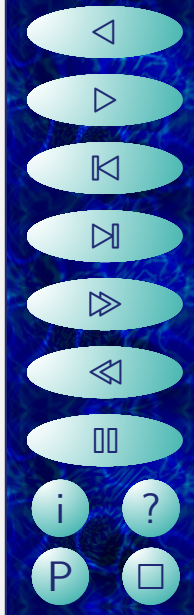
readLine

```
readLine :: IO String
readLine =
  do c <- getChar                -- lese ein Zeichen
     if c == '\n'                -- wenn Neue Zeile
       then return ""            -- dann leeren String
       else do l <- getLine      -- ansonsten rekursiv
               return (c : l)    -- zeichen und rest der Zeile
```

Überblick

- Haskell Geschichte + Überblick
- Evaluierungsmodelle: Call by value, Call by name, Call by need
- Definition: strikte Funktionen
- Unendliche Datenstrukturen
  - Strukturierung durch Ströme
- Seiteneffekte
  - Referentielle Transparenz
- Seiteneffekte und Aktionen in puren funktionalen Sprachen
  - Zustand durchfädeln
  - Monade

$\Rightarrow$ Laziness in strikten Sprachen



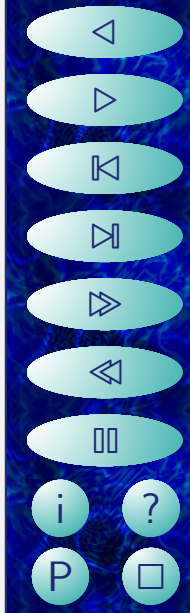
Ausblick: Lazyness in strikten Sprachen

Einfachste Möglichkeit: bestehende Sprachmittel (closures, o.ä) + evtl. syntaktischer Zucker

- Auswerten verzögerter Ausdrücke explizit durch Funktionsaufruf
- Verzögerte Ausdrücke haben eigenen Typ

Beispiel (OCAML)

```
# let n = lazy (factorial 12);;
val n : int lazy_t = <lazy>
# Lazy.force n + 1;;
- : int = 479001601
```



Lazyness in strikten Sprachen: Teil 2

Implementierung (vereinfacht)

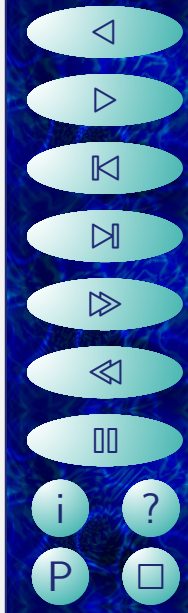
```
type 'a status =
  | Delayed of (unit -> 'a)
  | Value of 'a

type 'a t = 'a status ref

let force l =
  match !l with
  | Value v   -> v
  | Delayed f -> let v = f () in l := Value v; v

let _lazy f = ref (Delayed f)

lazy (3 + 4) ==> _lazy (fun () -> 3 + 4)
```



Lazyness in strikten Sprachen Teil 3

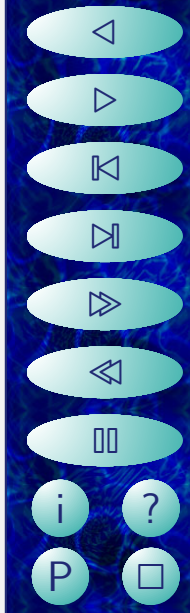
Probleme der OCAML Lösung

- Konstrukt orthogonal zu anderen Sprachelementen
- Eigener Typ $\Rightarrow$ bestehende Funktionen (z.Bsp für Listen) können nicht verwendet werden.

Lösung Semantik um bedarfsgesteuerte Ausdrücke erweitern

Beispiel (Alice)

```
- val n = lazy (factorial 12);
val n : int = _lazy
- n + 1;
val it : int = 479001601
```



Literatur

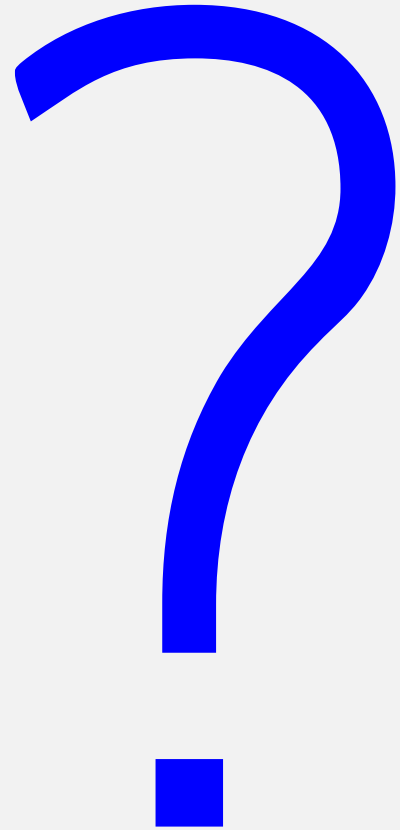
[1] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[2] S. P. Jones, J. Hughes, et al. *Report on the Programming Language Haskell 98*, February 1999.

[3] R. Plasmeijer and M. van Ekelen. *Clean Version 2.0 Language Report*, December 2001. Draft.

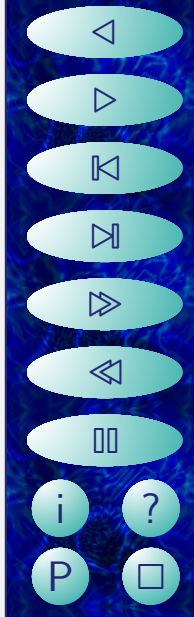
[4] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, New York;Amsterdam;Bonn, 1999.

[5] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 1–16, Nancy, France, September 1985. Springer.



Anhang

- Lazy programming im Alltag
- Der `State` Monad
- foldr versus foldl



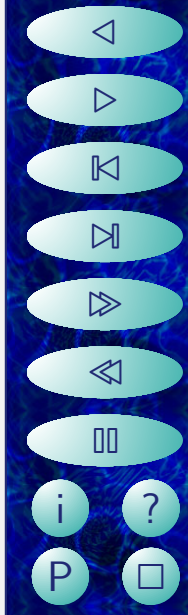
Lazy programming im Alltag


```
grep 'printf' foo.c | wc -l
```

- Zählt Anzahl der Zeilen in denen "printf" vorkommt
- Beide Befehle werden nebenläufig ausgeführt ⇒ keine temporäre Datei

```
grep 'printf' foo.c | head 5
```

- Zeigt maximal 5 Zeilen in denen "printf" vorkommt



Der State Monad

```
import Control.Monad.State
import Data.List

data Tree a = Nil | Node a (Tree a) (Tree a)

type Table = [String]

numberNode :: String -> State Table Int

numberNode s = do table <- get
    case elemIndex s table of
      Nothing -> do put (table ++ [s])
                    return (length table)
      Just i  -> return i

numberTree' :: Tree String -> State Table (Tree Int)

numberTree' Nil = return Nil
numberTree' (Node s l r) = do num <- numberNode s
                l'  <- numberTree' l
                r'  <- numberTree' r
                return (Node num l' r')

numberTree t = evalState (numberTree' t) []
```

```
import Data.List

data Tree a = Nil | Node a (Tree a) (Tree a)

type Table = [String]

numberNode :: String -> Table -> (Int, Table)

numberNode s table = case elemIndex s table of
         Nothing -> (length table, table ++ [s])
         Just i  -> (i, table)

numberTree' :: Tree String -> Table -> (Tree Int, Table)

numberTree' Nil table          = (Nil, table)
numberTree' (Node s l r) table =
    let (num, table')  = numberNode s table
        (l', table'')  = numberTree' l table'
        (r', table''') = numberTree' r table''
    in  (Node num l' r', table''')


numberTree t = fst (numberTree' t [])
```

`foldl` versus `foldr`

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs
```

- `foldl` bearbeitet immer die *ganzen* Liste
- unabhängig von `f`

```
foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)
```

- `foldr f z l` ist *nicht* strikt in `l` sofern `f` nicht strikt im zweiten Argument ist