

FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG

TEIL 3: LAZINESS UND ZIRKULARITÄT

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

9. Mai 2017



1 LAZY EVALUATION

- Fallstricke
- Striktheit
- Faulheit
- Zusammenfassung Lazy Evaluation

2 ZIRKULARITÄT

- repmin Beispiel
- Memoisation
- Termination von zirkulären Programmen
- Produktivität
- Zusammenfassung



AUSWERTUNGSSTRATEGIE

- Eine **Auswertungsstrategie** beschreibt, wie ein Ausdruck ausgewertet wird. Insbesondere:
 - In welcher Reihenfolge werden die reduzierbaren Teilausdrücke (**Redex**) bearbeitet?
 - Werden Funktionsargumente ausgewertet, bevor die Funktion aufgerufen wird?
- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.
- Bei *terminierenden* Programmen ohne Seiteneffekten ist die Auswertereihenfolge egal.
Die Auswertereihenfolge kann aber beeinflussen, ob ein Programm terminiert (wenn z.B. bei der Auswertung eines unbenötigten Funktionsargumentes ein Fehler auftritt).



LAZY EVALUATION

Haskell nutzt Auswertestrategie **call-by-need** (Bedarfsauswertung), implementiert durch **lazy evaluation** (verzögerte Auswertung).

- Termination wie bei Call-By-Name
- Effizienz (fast) wie bei Call-By-Value

IDEE

- Anstelle eines Redex wird ein Verweis (**Thunk**) eingesetzt
- Bei der ersten Verwendung des Verweises wird der Redex ausgewertet und durch seinen Ergebniswert ersetzt; weitere Verwendungen des Verweises liefern sofort diesen Ergebniswert.

Nur möglich, da die Auswertereihenfolge prinzipiell egal ist!

NACHTEILE

- Mehr Speicher notwendig zum Merken von Redexe und Thunks
- Keine **Sequentialität** der Auswertung (erst das, dann das)

VERGLEICH AUSWERTUNGSSTRATEGIEN

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

```
import Debug.Trace      -- von Verwendung wird abgeraten
trace :: String -> a -> a -- Seiteneffekt: Textausgabe
```

```
foo x y z = y + y + z
```

```
z = foo (trace "first" 1)
      (trace "second" 2)
      (trace "third" 3)
```

```
CALL-BY-VALUE "first second third" 7
```

```
CALL-BY-NAME "second second third" 7
```

```
LAZY EVALUATION "second third" 7
```

Ausgaben hypothetisch! GHC: immer Lazy Evaluation



VERGLEICH AUSWERTUNGSSTRATEGIEN

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

```
import Debug.Trace          -- von Verwendung wird abgeraten
trace :: String -> a -> a  -- Seiteneffekt: Textausgabe
```

```
foo x y z = y + y + z
```

```
z = foo (trace "first" undefined)
      (trace "second" 2)
      (trace "third" 3)
```

CALL-BY-VALUE *** Exception: Prelude.undefined

CALL-BY-NAME "second second third" 7

LAZY EVALUATION "second third" 7

Ausgaben hypothetisch! GHC: immer Lazy Evaluation



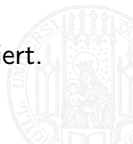
LAZY EVALUATION: BEISPIEL

Beispiel für Bedarfsauswertung:

```
foo x y z = if x<0 then abs x else x+y
```

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $x < 0$ und dieses wiederum ein Auswerten des Arguments x .
- Falls $x < 0$ wahr ist, wird der Wert von `abs x` zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $x+y$ zurückgegeben; dies erfordert die Auswertung von y .
- z wird in keinem Fall ausgewertet.
- Der Ausdruck `foo 1 2 (1 `div` 0)` ist daher wohldefiniert.



LAZY EVALUATION: BEISPIEL 2

```
g x y = let z = x `div` y
        in  if x > y * y then x else z
```

```
t2 = g 5 0
```

- `t2` liefert `5`. Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null.
- Nicht nur Funktionsargumente werden verzögert ausgewertet, sondern *beliebige Teilausdrücke*.
- **Sequentialität** der Auswertung (erst das, dann das) nicht gegeben, was sehr verwirrend sein kann.



POTENTIELL UNENDLICHE DATENSTRUKTUREN

Lazy Evaluation ermöglicht umgang mit unendlichen Datenstrukturen:

```
ones  = 1 : ones      -- ``unendliche'' Liste [1..]  
twos  = map (1+) ones -- [2..]
```

```
nums = iterate (1+) 0
```

```
iterate :: (a -> a) -> a -> [a]  
iterate f x = x : iterate f (f x)
```

```
take :: Int -> [a] -> [a]  
take n (x:xs) | n > 0 = x : take (n-1) xs  
take _ _ = []
```



POTENTIELL UNENDLICHE DATENSTRUKTUREN

Es wird immer nur soviel von der Datenstruktur ausgewertet wie benötigt wird:

```
> let nums = iterate (1+) 0
```

```
> take 10 nums
```

```
[0,1,2,3,4,5,6,7,8,9]
```

```
> :t (!!)
```

```
(!!) :: [a] -> Int -> a
```

```
> nums !! 999
```

```
999
```

⇒ Kontrollfluss unabhängig von Daten!



THUNKS IN GHCi

GHCi erlaubt die Inspektion des Speicher mit den Befehlen `:print` und `:sprint`, ohne das dabei Thunks ausgewertet werden:

```
> let x = map even [1..20]
> :sprint x
x = _
> x!!2
False
> :sprint x
x = _ : _ : False : _
```

Beid Befehle unterscheiden sich nur in der Ausführlichkeit der gezeigten Information, so zeigt `:print` z.B. auch gleich die jeweiligen Typen an.

Mit `:force` kann man die Auswertung von Thunks auserhalb der Reihenfolge erzwingen.



LAZY EVALUATION: BEISPIEL

Auswertung für `nums !! 2`:

~>

<code>nums !! 2</code>	\implies	<i>Ist Liste leer?</i>
<code>(iterate (1+) 0) !! 2</code>	\implies	
<code>(0:(iterate (1+) (1+0))) !! 2</code>	\implies	<i>Ist Index 0?</i>
<code>(iterate (1+) (1+0)) !! 1</code>	\implies	<i>Ist Liste leer?</i>
<code>((1+0):(iterate (1+) (1+(1+0)))) !! 1</code>	\implies	<i>Ist Index 0?</i>
<code>(iterate (1+) (1+(1+0))) !! 0</code>	\implies	<i>Ist Liste leer?</i>
<code>((1+(1+0)):iterate (1+) (...)) !! 0</code>	\implies	<i>Ist Index 0?</i>
<code>(1+(1+0))</code>	\implies	
<code>(1+1)</code>	\implies	
<code>2</code>		



BEISPIEL: TRENNUNG DATEN & KONTROLLE

Lazy Evaluation ermöglicht Trennung von Daten und Kontrollfluss, auch ohne den Einsatz von Zirkularität:

BEISPIEL

```
factors :: Integral a => a -> [a]
factors n = filter (\m -> n `mod` m == 0) [2 .. (n - 1)]

isPrime :: Integral a => a -> Bool
isPrime n = n > 1 && null (factors n)
```

`factors` berechnet alle Faktoren einer Zahl.

Die Berechnung von `isPrime` bricht sofort ab, so bald der erste Faktor gefunden wurde, trotz Verwendung von `factors` werden also nicht alle Faktoren berechnet!



BEISPIEL: SIEB DES ERATHOSTENES

Unendliche Liste aller Primzahlen:

```
primes :: [Integer]
primes = sieve [2..]
```

```
sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (xs `minus` [p,p+p..])
```

```
minus xs@(x:xt) ys@(y:yt) | LT == cmp = x : minus xt ys
                           | EQ == cmp =      minus xt yt
                           | GT == cmp =      minus xs yt
                           where cmp = compare x y
```

- Wir müssen uns nur um die Daten kümmern, also *wie* wir die Primzahlen berechnen.
- Kontrolle über Zahl der benötigten Primzahlen erfolgt später!
- Zu Effizienz-Betrachtungen siehe [Haskell-Wiki: Prime Numbers](#)

SPEICHERVERBRAUCH

Der Teufel kann aber hier im Detail stecken:

```
(++) :: [a] -> [a] -> [a]
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = x : xs ++ ys
```

```
cycleA xs = xs ++ cycleA xs
```

```
cycleB xs = xs' where xs' = xs++xs'
```

`cycleA 1` verbraucht potenziell unendlich viel Speicher; genauer: so viel Speicherzellen, wie Elemente von `cycleA 1` gelesen werden.

Für eine Liste `l` mit Länge `n` verbraucht `cycleB` jedoch nur maximal `n` Speicherzellen.

Für die weitere Verarbeitung spielt dies jedoch keine Rolle mehr!



FORMALE OPERATIONALE SEMANTIK

Um solche Eigenschaften zu untersuchen, muss man die Auswertung durch eine **operationale Semantik** formalisieren.

„A natural semantics for lazy evaluation“ von J. Launchbury POPL'93 wird oft als Basis verwendet, z.B. auch in **unseren Forschungsarbeiten** zu verzögerter Auswertung:

$$\begin{array}{c}
 \overline{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_m w \Downarrow w, \mathcal{H}} \quad (\text{WHNF}_\Downarrow) \\
 \\
 \frac{\ell \notin \mathcal{L} \quad \mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \cup \{\ell\} \vdash_m e \Downarrow w, \mathcal{H}'[\ell \mapsto e]}{\mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \vdash_m \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \quad (\text{VAR}_\Downarrow) \\
 \\
 \frac{\ell \text{ is fresh} \quad \mathcal{H}[\ell \mapsto e_1[\ell/x]], \mathcal{S}, \mathcal{L} \vdash_m e_2[\ell/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_m^{\text{let}} \text{let } x = e_1 \text{ in } e_2 \Downarrow w, \mathcal{H}'} \quad (\text{LET}_\Downarrow) \\
 \\
 \frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_m e \Downarrow \lambda x. e', \mathcal{H}' \quad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash_m^{\text{app}} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_m^{\text{app}} e \ell \Downarrow w, \mathcal{H}''} \quad (\text{APP}_\Downarrow) \\
 \\
 \frac{\mathcal{H}, \mathcal{S} \cup (\bigcup_{i=1}^n \{\bar{x}_i\} \cup \text{BV}(e_i)), \mathcal{L} \vdash_m e_0 \Downarrow c_k(\bar{\ell}), \mathcal{H}' \quad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash_m^{\text{match}} e_k[\bar{\ell}/\bar{x}_k] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash_m^{\text{match}} \text{match } e_0 \text{ with } \{c_i(\bar{x}_i) \rightarrow e_i\}_{i=1}^n \Downarrow w, \mathcal{H}''} \quad (\text{MATCH}_\Downarrow)
 \end{array}$$

Fig. 1. Instrumented version of Launchbury's operational semantics



FORMALISIERUNG DER VERZÖGERTEN AUSWERTUNG

Wir begnügen uns hier mit einer vereinfachten Version:

- Alle Teilausdrücke werden zuerst in den Speicher geschoben.
- Um mit Rekursion umzugehen, wird dabei die Variable durch einen frischen, einzigartigen Namen (Speicheradresse) ersetzt; im ursprünglichen Quelltext kommen diese nicht vor.

Wir verwenden dafür spitze Klammern in Beispielen, z.B. $\langle 3 \rangle$

- Jede Speicherzelle wird nur *einmal* verändert: Wird der Wert der Speicherzelle gebraucht und daher ausgewertet, so ersetzen wir den Ausdruck im Speicher durch seinen Wert.
- Speicherzellen werden nie gelöscht.
In der Praxis kümmert sich ein Garbage Collector um das freigeben unbenötigter Speicherzellen.



BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA ['a', 'b']			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	['a', 'b']	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	['a', 'b']	
<11> ++ (cycleA <11>)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	['a', 'b']	
<11> ++ <12>	<12>	cycleA <11>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
	<13>	'a'	✓
	<14>	['b']	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> (<14> ++ <12>)	<13>	'a'	✓
	<14>	['b']	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
	<14>	['b']	
	<15>	<14> ++ <12>	

$$\text{cycleA } xs = xs ++ (\text{cycleA } xs)$$

$$(++)\ [] \quad ys = ys$$

$$(++)\ (x:xs) \quad ys = (:) x (xs++ys)$$

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15>	<14>	['b']	
	<15>	<14> ++ <12>	

cycleA xs = xs ++ (cycleA xs)

(++) [] ys = ys

(++) (x:xs) ys = (:) x (xs++ys)

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	['b']	
	<15>	<14> ++ <12>	

cycleA xs = xs ++ (cycleA xs)

(++) [] ys = ys

(++) (x:xs) ys = (:) x (xs++ys)

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
	<15>	<14> ++ <12>	
	<16>	'b'	✓
	<17>	[]	✓

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	<14> ++ <12>	
	<16>	'b'	✓
	<17>	[]	✓

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	<14> ++ <12>	
(:) <16> <18>	<16>	'b'	✓
	<17>	[]	✓
	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
	<17>	[]	✓
	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18>	<17>	[]	✓
	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```


BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12>	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	<17> ++ <12>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	<17> ++ <12>	
<11> ++ (cycleA <11>)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	<17> ++ <12>	
<11> ++ <19>	<19>	cycleA <11>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	<17> ++ <12>	
<11> ++ <19>	<19>	cycleA <11>	
(:) <13> (<14> ++ <19>)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	cycleA <11>	
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	<17> ++ <12>	
<11> ++ <19>	<19>	cycleA <11>	
(:) <13> <20>	<20>	<14> ++ <19>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code><17> ++ <12></code>	
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code><14> ++ <19></code>	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code><14> ++ <19></code>	
<code><20></code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```


BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleA <11>	<11>	(:) <13> <14>	✓
<11> ++ <12>	<12>	(:) <13> <20>	✓
(:) <13> <15>	<13>	'a'	✓
<15> = <14> ++ <12>	<14>	(:) <16> <17>	✓
(:) <16> (<17> ++ <12>)	<15>	(:) <16> <18>	✓
(:) <16> <18>	<16>	'b'	✓
<18> = <17> ++ <12>	<17>	[]	✓
<12> = cycleA <11>	<18>	(:) <13> <20>	✓
<11> ++ <19>	<19>	cycleA <11>	
(:) <13> <20>	<20>	<14> ++ <19>	
<20> = <14> ++ <19>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code><14> ++ <19></code>	
<code><20> = <14> ++ <19></code>			
<code>(:) <16> (<17> ++ <19>)</code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code><14> ++ <19></code>	
<code><20> = <14> ++ <19></code>	<code><21></code>	<code><17> ++ <19></code>	
<code>(:) <16> (<17> ++ <19>)</code>			
<code>(:) <16> <21></code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code>(:) <16> <21></code>	✓
<code><20> = <14> ++ <19></code>	<code><21></code>	<code><17> ++ <19></code>	
<code>(:) <16> (<17> ++ <19>)</code>			
<code>(:) <16> <21></code>			
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code>(:) <16> <21></code>	✓
<code><20> = <14> ++ <19></code>	<code><21></code>	<code><19></code>	
<code>(:) <16> (<17> ++ <19>)</code>			
<code>(:) <16> <21></code>			
<code>⋮</code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code>(:) <16> <21></code>	✓
<code><20> = <14> ++ <19></code>	<code><21></code>	<code><19></code>	
<code>(:) <16> (<17> ++ <19>)</code>	<code><22></code>	<code>cycleA <11></code>	
<code>(:) <16> <21></code>			
<code>⋮</code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) [] ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>cycleA <11></code>	
<code>(:) <13> <20></code>	<code><20></code>	<code>(:) <16> <21></code>	✓
<code><20> = <14> ++ <19></code>	<code><21></code>	<code><19></code>	
<code>(:) <16> (<17> ++ <19>)</code>	<code><22></code>	<code>cycleA <11></code>	
<code>(:) <16> <21></code>	<code><23></code>	<code><14> ++ <22></code>	
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []      ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
<code>cycleA <11></code>	<code><11></code>	<code>(:) <13> <14></code>	✓
<code><11> ++ <12></code>	<code><12></code>	<code>(:) <13> <20></code>	✓
<code>(:) <13> <15></code>	<code><13></code>	<code>'a'</code>	✓
<code><15> = <14> ++ <12></code>	<code><14></code>	<code>(:) <16> <17></code>	✓
<code>(:) <16> (<17> ++ <12>)</code>	<code><15></code>	<code>(:) <16> <18></code>	✓
<code>(:) <16> <18></code>	<code><16></code>	<code>'b'</code>	✓
<code><18> = <17> ++ <12></code>	<code><17></code>	<code>[]</code>	✓
<code><12> = cycleA <11></code>	<code><18></code>	<code>(:) <13> <20></code>	✓
<code><11> ++ <19></code>	<code><19></code>	<code>(:) <13> <23></code>	✓
<code>(:) <13> <20></code>	<code><20></code>	<code>(:) <16> <21></code>	✓
<code><20> = <14> ++ <19></code>	<code><21></code>	<code>(:) <13> <23></code>	✓
<code>(:) <16> (<17> ++ <19>)</code>	<code><22></code>	<code>cycleA <11></code>	
<code>(:) <16> <21></code>	<code><23></code>	<code><14> ++ <22></code>	
<code>⋮</code>			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []      ys = ys
```

```
(++) (x:xs) ys = (:) x (xs++ys)
```


BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB ['a', 'b']			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	['a', 'b']	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	['a', 'b']	
<12>	<12>	<11> ++ <12>	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	['a', 'b']	
<12>	<12>	<11> ++ <12>	
<11> ++ <12>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	<11> ++ <12>	
<11> ++ <12>	<13>	'a'	✓
	<14>	['b']	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	<11> ++ <12>	
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	['b']	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	<11> ++ <12>	
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	['b']	
(:) <13> <15>	<15>	<14> ++ <12>	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	['b']	
(:) <13> <15>	<15>	<14> ++ <12>	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```


BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	['b']	
(:) <13> <15>	<15>	<14> ++ <12>	
<15>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	['b']	
(:) <13> <15>	<15>	<14> ++ <12>	
<15> = <14> ++ <12>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	<14> ++ <12>	
<15> = <14> ++ <12>	<16>	'b'	✓
	<17>	[]	✓

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	<14> ++ <12>	
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	<14> ++ <12>	
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	<17> ++ <12>	

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	(:) <16> <18>	✓
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	<17> ++ <12>	
<18>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	(:) <16> <18>	✓
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	<17> ++ <12>	
<18> = <17> ++ <12>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	(:) <16> <18>	✓
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	<17> ++ <12>	
<18> = <17> ++ <12>			
<12>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```


BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	(:) <16> <18>	✓
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	<17> ++ <12>	
<18> = <17> ++ <12>			
<12>			
(:) <13> <15>			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

BEISPIEL

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB <11>	<11>	(:) <13> <14>	✓
<12>	<12>	(:) <13> <15>	✓
<11> ++ <12>	<13>	'a'	✓
(:) <13> (<14> ++ <12>)	<14>	(:) <16> <17>	✓
(:) <13> <15>	<15>	(:) <16> <18>	✓
<15> = <14> ++ <12>	<16>	'b'	✓
(:) <16> (<17> ++ <12>)	<17>	[]	✓
(:) <16> <18>	<18>	(:) <13> <15>	✓
<18> = <17> ++ <12>			
<12>			
(:) <13> <15>			
Alles vollständig ausgewertet!			

```
cycleB xs = xs'
  where xs' = xs++xs'
```

```
(++) []      ys = ys
(++) (x:xs)  ys = (:) x (xs++ys)
```

SPEICHERVERBRAUCH LAZY EVALUATION

BEISPIEL

```
sumWithL :: [Int] -> Int -> Int
sumWithL [] acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4] 1 ~>
sumWithL [3,4] (1+2) ~>
sumWithL [4] ((1+2)+3) ~>
sumWithL [] (((1+2)+3)+4) ~>
                    ((1+2)+3)+4 ~>
                    ((3)+3)+4 ~> (6)+4 ~> 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹️



SPEICHERVERBRAUCH LAZY EVALUATION

BEISPIEL

```
sumWithL :: [Int] -> Int -> Int
sumWithL [] acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4] 1 ~>
sumWithL [3,4] (1+2) ~>
sumWithL [4] ((1+2)+3) ~>
sumWithL [] (((1+2)+3)+4) ~>
                    ((1+2)+3)+4 ~>
                    ((3)+3)+4 ~> (6)+4 ~> 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹️



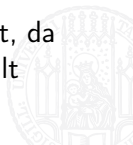
SPEICHERVERBRAUCH LAZY EVALUATION

BEISPIEL

```
sumWithL :: [Int] -> Int -> Int
sumWithL [] acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4] 1 ~>
sumWithL [3,4] (1+2) ~>
sumWithL [4] ((1+2)+3) ~>
sumWithL [] (((1+2)+3)+4) ~>
                ((1+2)+3)+4 ~>
                ((3)+3)+4 ~> (6)+4 ~> 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹️



STRIKTHEIT

Eine Argument einer Funktion heißt **strikt**, wenn es auf jeden Fall ausgewertet wird, egal welchen Wert die anderen Argumente haben.

BEISPIEL:

```
bar x y z = if 1 < succ x then z else x+y
```

Hier ist x strikt, y und z aber nicht.



STRIKTHEIT ERZWINGEN

Haskell bietet einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

$$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$$

Der Ausdruck $f \$! x$ erzwingt die Auswertung von x vor der Anwendung von f .

BEISPIEL:

```
sumWithS :: [Int] -> Int -> Int
sumWithS [] acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h
```

```
sumWithS [2,3,4] 1 ~>
sumWithS [3,4] $! (1+2) ~> sumWithS [3,4] 3
sumWithS [4] $! (3+3) ~> sumWithS [4] 6
sumWithS [] $! (6+4) ~> sumWithS [] 10 ~> 10
```



STRIKTHEIT ERZWINGEN

Haskell bietet einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

$$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$$

Der Ausdruck $f \$! x$ erzwingt die Auswertung von x vor der Anwendung von f .

BEISPIEL:

```
sumWithS :: [Int] -> Int -> Int
```

```
sumWithS [] acc = acc
```

```
sumWithS (h:t) acc = sumWithS t $! acc+h
```

```
sumWithS [2,3,4] 1 ~>
```

```
sumWithS [3,4] $! (1+2) ~> sumWithS [3,4] 3
```

```
sumWithS [4] $! (3+3) ~> sumWithS [4] 6
```

```
sumWithS [] $! (6+4) ~> sumWithS [] 10 ~> 10
```



STRIKTHEIT ERZWINGEN

Haskell bietet einen Mechanismus, welcher die **strikte Auswertung** erzwingt:

$$(\$!) :: (a \rightarrow b) \rightarrow a \rightarrow b$$

Der Ausdruck $f \$! x$ erzwingt die Auswertung von x vor der Anwendung von f .

BEISPIEL:

```
sumWithS :: [Int] -> Int -> Int
```

```
sumWithS [] acc = acc
```

```
sumWithS (h:t) acc = sumWithS t $! acc+h
```

```
sumWithS [2,3,4] 1 ~>
```

```
sumWithS [3,4] $! (1+2) ~> sumWithS [3,4] 3
```

```
sumWithS [4] $! (3+3) ~> sumWithS [4] 6
```

```
sumWithS [] $! (6+4) ~> sumWithS [] 10 ~> 10
```



STRIKTHEIT ERZWINGEN

`$!` ist definiert durch das Primitiv `seq :: a -> b -> b`

```
($!) :: (a -> b) -> a -> b
```

```
f $! x = x `seq` f x
```

`seq` erzwingt die Auswertung seines ersten Argumentes und liefert danach das zweite Argument zurück.

```
foo x =  
  let zwischenwert = bar x  
      ergebnis     = goo zwischenwert  
  in  seq zwischenwert ergebnis
```

Von einigen Funktionen bietet die Standardbibliothek auch strikte Varianten, z.B. macht `foldl'` das gleiche wie `foldl`, erzwingt jedoch die Auswertung des Akkumulators in jedem Schritt.



STRIKTHEIT ERZWINGEN

Das erste Argument von `seq` wird nur soweit ausgewertet, bis dessen äußere Form klar ist, darin kann auf weitere Thunks verwiesen werden:

INT, BOOL: werden vollständig ausgewertet

LISTEN: ausgewertet bis klar ist, ob Liste leer ist oder nicht. Kopf und der Rumpf werden nicht ausgewertet!

TUPEL: ausgewertet bis Tupel-Konstruktor fest steht, d.h. Elemente des Tupels werden nicht ausgewertet.

MAYBE: ausgewertet bis `Nothing` oder `Just`, das Argument von `Just` wird noch nicht ausgewertet.

Generell wertet `seq` bis zum äußeren Konstruktor aus. Argumente des Konstruktors werden nicht weiter ausgewertet.

⇒ **Schwache Kopf-Normalform** Weak Head Normal Form

WHNF

`Control.DeepSeq` bietet Kontrolle über Auswertungsreihenfolge.

DEMO

Demo `sumWith.hs`:

Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!



DEMO

Demo `sumWith.hs`:

Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

- Wir haben gesehen, dass der Kompilier automatisch eine Striktheit-Analyse durchführt, d.h. wir müssen nur selten eingreifen.
- Wenn ein Stack-Overflow eintritt, dann sollte man über Endrekursion und auch über Striktheit nachdenken.



STRIKTHEIT ERZWINGEN

Eine weitere Alternative bietet die Spracherweiterung `BangPatterns`. Damit können wir `($!)` ebenfalls definieren:

```
($!) :: (a -> b) -> a -> b
f $! x! = f x
```

Patterns, welche mit `!` beginnen, müssen immer zuerst ausgewertet werden:

```
> ghci -XBangPatterns

> let foo (x,!y) = [x,y]
> drop 1 $ foo (undefined,2)
[2]
> take 1 $ foo (2,undefined)
*** Exception: Prelude.undefined
```



PROBLEME MIT STRIKTTHEIT

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr (&&) True (repeat False)
False
```

```
> foldr' (&&) True (repeat False) -- aus Data.Foldable
<interactive>: Heap exhausted;
```



PROBLEME MIT STRIKTTHEIT

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr (&&) True (repeat False)
```

```
False
```

```
> foldr' (&&) True (repeat False) -- aus Data.Foldable
```

```
<interactive>: Heap exhausted;
```

```
> let foo = const 42 -- const x y = x
```

```
> foo $ undefined
```

```
42
```

```
> foo $! undefined
```

```
*** Exception: Prelude.undefined
```

Eine Funktion, welche für `undefined` als n -tes Argument immer einen Fehler liefern, nennt man auch **strikt im n -ten Argument**.

FAULHEIT ERZWINGEN

Man kann auch faule Pattern-Matches erzwingen:

```
> let bar (x,y) = 42
> bar undefined
*** Exception: Prelude.undefined
> let baz ~(x,y) = 42
> baz undefined
42
```

Auch in verschachtelten Patterns möglich, z.B. $(x, \sim y)$.

FALLSTRICK: Fauler Patterns sind immer irrefutable, sollten also als letztes kommen. Verwendung kann man sich so vorstellen:

```
quu ~(x,y) = x+y
```

```
quu p= (fst p)+(snd p)
```

```
qux ~(h:t) = 2*h:(qux t)
```

```
qux l= 2*(head l):(qux(tail l))
```

FAULHEIT ERZWINGEN: BEISPIEL

DEMO: `splitAt_lazypattern.hs`

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n ls
  | n <= 0      = ([], ls)
splitAt _ []   = ([], [])
splitAt n (y:ys) =
  case splitAt (n-1) ys of
    ~(prefix, suffix) -> (y : prefix, suffix)
```

Damit steht der erste Teil des Ergebnis-Paares sofort zur Verfügung; der Rest muss dagegen noch später ausgewertet werden.



FAULHEIT IN STRIKTEN SPRACHEN

In anderen Programmiersprachen ist die Idee nicht unbekannt:
z.B. bieten C und Java faule Varianten von logischen Operatoren:
Der Java-UND-Operator `&` wertet immer beide Argumente aus;
aber `&&` wertet das zweite Argument nicht aus, wenn das erste
bereits `False` ergibt. \Rightarrow short-circuit (Kurzschluß)

Programmiersprachen, in denen im Gegensatz zur verzögerten
Auswertung alle Ausdrücke sofort ausgewertet werden, nennt man
strikt engl. *eager*. Fast alle imperativen Sprachen sind strikt.

Faule Auswertung kann in strikten Sprachen simuliert werden, in
dem man Ausdrücke zu Funktionen mit Scheinargumenten macht:

```
foo x = let e' = (\_ -> e) -- keine Auswertung von e  
        in ... e' () ...   -- Auswertung von e
```



ZUSAMMENFASSUNG

- Haskell verwendet Auswertestrategie Lazy Evaluation:
Maximal einmal auswerten, wenn überhaupt
- Terminations- und Ausnahmeverhalten können von der Auswertestrategie abhängen.
- Lazy Evaluation erlaubt Verwendung unendlicher oder zirkulärer Datenstrukturen
- Lazy Evaluation erlaubt gute Modularisierung durch Trennung von Daten und Kontrollfluss;
kann aber Speicherverbrauch erhöhen.
- Ein Funktionsargument, welches auf jeden Fall ausgewertet wird, heißt strikt.
- Annotationen zur Erzwingung von Striktheit möglich:
Strikt: `!p` Faul: `~p` (irrefutable)
- Übermäßige Striktheit verhindert Verwendung unendlicher Datenstrukturen.



DEFINITION

Ein *zirkuläres* Programm erzeugt eine Datenstruktur, deren Berechnung von sich selbst abhängt.

- Solche Programme erfordern nicht-strikte Datenstrukturen.
- Diese werden in Sprachen mit nicht-strikter Semantik unterstützt, und können in Sprachen mit strikter Semantik simuliert werden.
- Zirkuläre Programme werden oft verwendet um mehrfaches Traversieren einer Datenstruktur oder den Aufbau von Zwischen-Datenstrukturen zu vermeiden.



ZIRKULÄRE DATENSTRUKTUREN

Ein einfaches Beispiel einer zirkulären Datenstruktur ist die Liste *aller* natürlichen Zahlen:

```
nums2 = 0 : (map (+1) nums2)
```

Beachte:

Programm benutzt die erzeugte Datenstruktur selbst als Eingabe!

REKURSIVE VARIANTE ZUR ERINNERUNG:

```
nums1 = iterate (+1) 0
```

```
iterate f x = x : iterate f (f x)
```



BEISPIEL: ELIMINIERUNG VON DUPLIKATEN

Die Funktion `nub1` eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []          = []
nub1 (x : xs)   = x : (nub1 (filter (x/=) xs))
```



BEISPIEL: ELIMINIERUNG VON DUPLIKATEN

Die Funktion `nub1` eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []           = []
nub1 (x : xs)    = x : (nub1 (filter (x/=) xs))
```

Diese Funktion erzeugt für jedes Listenelement der Eingabeliste eine Liste als Zwischen-Datenstruktur:

- 1 Liste, bei der all doppelten des ersten Elements entfernt wurden
- 2 Liste, bei der alle doppelten des ersten und zweiten Elements entfernt wurden;

usw. ☹



BEISPIEL: ELIMINIERUNG VON DUPLIKATEN

Eine bessere Implementierung vermeidet diese Zwischen-Listen:

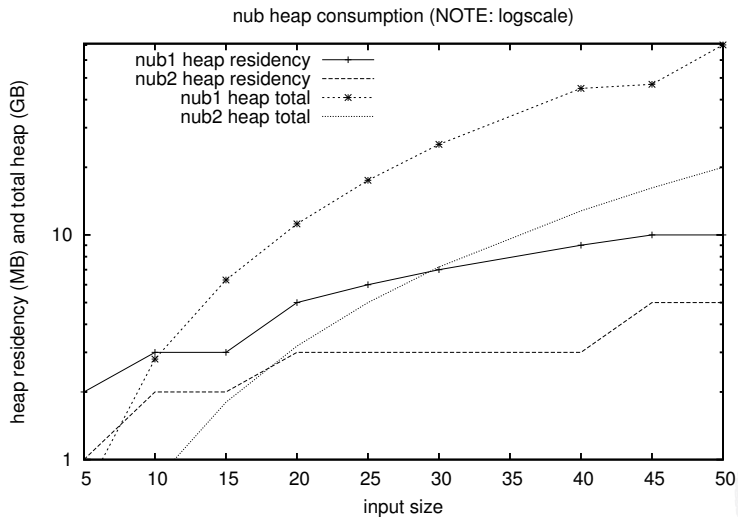
```
nub2 :: [Integer] -> [Integer]
nub2 xs = res
  where
    res = build xs 0

    build [] _ = []
    build (x : xs) n
      | mem x res n = build xs n
      | otherwise   = x : (build xs (n + 1))

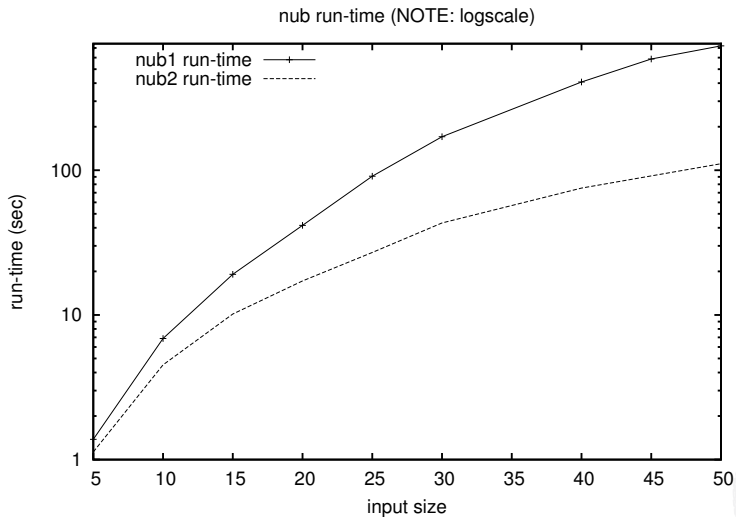
    mem _ _ys 0 = False
    mem x (y : ys) n
      | x == y   = True
      | otherwise = mem x ys (n - 1)
```



PERFORMANZ



PERFORMANZ



BEISPIEL: REPMIN

GEGEBEN: Binärer Baum von Integer Werten.

GESUCHT: Binärer Baum mit der gleichen Struktur in dem die Werte der Blätter durch das kleinste Element im Baum ersetzt sind.



NAIVE IMPLEMENTIERUNG

Die naive Implementierung verwendet 2 separate Funktionen:

```
treemin :: (Ord a) => BinTree a -> a
treemin (Leaf x)      = x
treemin (Node l r)    = min (treemin l) (treemin r)
```

```
replace :: BinTree a -> a -> BinTree a
replace (Leaf _) m    = Leaf m
replace (Node l r) m = Node (replace l m) (replace r m)
```

```
transform2p :: (Ord a) => BinTree a -> BinTree a
transform2p t = replace t (treemin t)
```

Dieser Code traversiert den Baum 2-mal: einmal in `treemin` und einmal in `replace`



VERBESSERTE IMPLEMENTIERUNG

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.

Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x) m = (Leaf m, x)
repmin (Node l r) m =
  let (l', ml) = repmin l m
      (r', mr) = repmin r m
  in (Node l' r', min ml mr)
```



VERBESSERTE IMPLEMENTIERUNG

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```

repm (Leaf x) m = (Leaf m, x)
repm (Node l r) m = (l', ml) `comb` (r', mr)
  where
    (l', ml) = repm l m
    (r', mr) = repm r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)

```



VERBESSERTE IMPLEMENTIERUNG

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.

Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x) m = (Leaf m, x)
```

```
repmin (Node l r) m = (l', ml) `comb` (r', mr)
```

```
  where
```

```
    (l', ml) = repmin l m
```

```
    (r', mr) = repmin r m
```

```
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```



VERBESSERTE IMPLEMENTIERUNG

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```

repmin (Leaf x) m = (Leaf m, x)
repmin (Node l r) m = (l', ml) `comb` (r', mr)
  where
    (l', ml) = repmin l m
    (r', mr) = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)

```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

Nun können wir eine zirkuläre Datenstruktur verwenden um das Resultat der `treemin` Komponente mit der Eingabe der `replace` Komponente zu verknüpfen:

```
transform1p t = fst p where p = repmin t (snd p)
```



MEMOISATION

- **Memoisation** ist eine Programmier- oder Implementierungstechnik, in der die Resultate früherer Aufrufe einer Funktion gespeichert (“memoised”) werden, um wiederholtes Auswerten zu vermeiden.
- Im Allgemeinen verwendet man dazu eine Datenstruktur, wie eine Hash-Tabelle, und legt dort die bereits berechneten Werte ab.
- Dieser Ansatz ist von Vorteil, wenn die einzelnen Berechnungen sehr teuer sind, und den Administrationsaufwand der Tabellenverwaltung rechtfertigen.



LIGHT-WEIGHT MEMOISATION

Zirkuläre Datenstrukturen bieten eine leichtgewichtige Alternative zu generellen Hash-Tabellen.

Anstatt eine Funktion direkt zu definieren, wird eine unendliche Liste aller Resultatwerte definiert. Der Funktionsaufruf wird durch eine Indexing-Operation auf der Liste ersetzt.

```
fibs :: [Integer]
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))

fib :: Int -> Integer
fib n = fibs !! n
```

Wiederholte Aufrufe liefern deutlich schneller ein Ergebnis!



BEISPIEL: HAMMING NUMBERS

Liste aller Vielfachen von 2,3 und 5, auch bekannt als harmonische Zahlen.

```
hamming = 1 : (merge (map (2*) hamming) $
               merge (map (3*) hamming)
                     (map (5*) hamming))
```

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs@(x:xt) ys@(y:yt)
  | x==y = x : merge xt yt
  | x<y  = x : merge xt ys
  | x>y  = y : merge xs yt
```



TERMINATION VON ZIRKULÄREN PROGRAMMEN

Der Beweis der Termination von zirkulären Programmen ist oft nicht trivial.

Im vorangegangenen Beispiel kann man zeigen, dass immer nur auf vorherige Elemente der Datenstruktur zugegriffen wird, i.e. das n -te Element hängt von allen i -ten Elementen, mit $i < n$ ab.

Verallgemeinert nutzt man das Konzept der Produktivität um die Termination von zirkulären Programmen zu zeigen.



PRODUKTIVITÄT

DEFINITION (MAXIMAL)

Gegeben eine partielle Ordnung \sqsubseteq auf einer Menge D . Ein Element $x \in D$ ist *maximal* wenn

$$\forall y \in D. y \sqsubseteq x$$

\sqsubseteq auf Funktionen und Listen wird komponentenweise definiert.
Produktivität von $x \in \sigma$ induktiv über Struktur des Typs σ :

DEFINITION (PRODUKTIV)

$x \in \sigma$ ist *produktiv*, wenn

- $\sigma = Int \vee \sigma = Bool$: x ist maximal
- $\sigma = \tau \rightarrow \tau'$: x bildet produktive Elemente in produktive Elemente ab
- $\sigma = [\tau]$: $\forall i. 0 \leq i \leq \#x \implies x!!i$ ist produktiv

PRODUKTIVITÄT

DEFINITION (MENGEN-PRODUKTIVITÄT)

Eine Liste $x \in [\sigma]$ ist A -produktiv ($A \subseteq \mathbb{N}$), gdw. $\forall a \in A . x!!a$ ist produktiv.

Intuition:

A ist die Menge der Indizes von berechneten Listen-Elementen.



PRODUKTIVITÄT

DEFINITION (SEGMENT-PRODUKTIVITÄT)

Eine Liste $x \in [\sigma]$ ist n -produktiv ($n \in \mathbb{N}$), gdw. x ist $\{0, 1, \dots, n-1\}$ produktiv.

Intuition: die ersten n Elemente der Liste sind berechnet.

DEFINITION (SEGMENT-PRODUKTIVITÄT)

Eine Funktion $f \in [\sigma] \rightarrow [\tau]$ ist v -produktiv für $v \in \mathbb{N} \rightarrow \mathbb{N}$, gdw. $\forall k \in \mathbb{N}, x \in [\sigma]. x \text{ ist } k \text{ produktiv} \implies (f \ x) \text{ ist } (v \ k) \text{ produktiv}$

Intuition: die v Funktion gibt an, wie sich die Größe des berechneten Segments verändert.



PRODUKTIVITÄT

BEISPIEL 1:

Die Funktion `map f` ist `id` produktiv,
d.h. die Segmentgröße bleibt unverändert.

BEISPIEL 2:

Die List-cons Funktion `(:)` ist `(+1)` produktiv,
d.h. die Segmentgröße steigt um 1.



PRODUKTIVITÄT

THEOREM (LISTEN-PRODUKTIVITÄT)

Sei eine Funktion $f \in [\sigma] \rightarrow [\sigma]$ welche $v \in \mathbb{N} \rightarrow \mathbb{N}$ -produktiv ist, und $\forall k \in \mathbb{N} . (v\ k) > k$. Dann ist der Fixpunkt von f eine produktive, unendliche Liste.

Intuition: Um zu zeigen, dass alle Element einer unendlichen Liste berechenbar sind, muss man zeigen, dass die Änderung der Segmentgröße immer ansteigt. D.h. in jeder Iteration steigt der berechnete Teil der Liste.



BEISPIEL: BEWEIS DER PRODUKTIVITÄT DER MEMOISIERENDEN FIBONACCI FUNKTION.

- Die Funktion `zipWith` ist \min produktiv, da sie eine binäre Funktion f komponentenweise auf die beiden Argumentlisten `xs`, `ys` anwendet. Wenn die ersten m Elemente von `xs` und die ersten n Elemente von `ys` berechnet sind, dann sind die ersten $\min m n$ Elemente der Resultatliste berechnet.
- Die List-cons Funktion `(:)` ist $(+1)$ produktiv.
- Sei `fibs` eine k produktive Liste, und `vtl` die Produktivitätsfunktion von `tail`. Dann ist der Rumpf

$$1 : 1 : (\text{zipWith } (+) \text{ fibs } (\text{tail fibs}))$$
 $1 + 1 + \min k (vtl k)$ produktiv.
- Weiterhin ist $vtl k = k - 1$ für alle $k > 0$.
- Daher ist die Produktivität des Rumpfs von `fibs`:

$$1 + 1 + \min k (vtl k) = 2 + k - 1 = k + 1 > k \quad \square$$



ZUSAMMENFASSUNG

- Zirkularität erfordert nicht-strikte Datenstrukturen
- Zirkuläre Programme können wiederholtes Traversieren von Datenstrukturen vermeiden
- Zirkularität ermöglicht eine einfache Form von Memoisation
- Man muss bei zirkulären Programmen auf die Produktivität achten, d.h. es werden pro konsumierten Element einer Datenstruktur mindestens ebenso so viele Elemente erzeugt.

