

Einführung in die funktionale Programmierung

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

21. November 2007

Monade

- ist ein Datentyp für (sequentielle) Aktionen
- i.a.: parametrisiert mit innerem Zustand und Ausgabetype der Aktionen
- gekapseltes Weiterreichen des Zustands
- Kombinatoren für Aktionen
- Gekapselt wird i.a. der Zustand und das Weiterreichen des Zustands

wichtige Verwendung von Monaden

- Sequentialisierung
- Konzepte der imperativen Programmierung und Seiteneffekte in purer, referentiell transparenter Form
- Ein-Ausgabe, d.h. IO
- zustandsbehaftete Berechnungen
- single-threaded zu verwendende Objekte

- Falls Außenwelt sich ändert ohne Einwirkung des Programms: Sequentialität hilft, aber das theoretische Modell passt nicht.
- Monaden-Modell bietet keine Unterstützung für vertauschbare Ein-Ausgaben.
- Implementierung der Monaden entspricht nicht dem reinen Lambda-Kalkül:
keine call-by-name-Theorie sondern call-by-need-Theorie
- Es gibt nicht-monadische Ansätze zu Ein-Ausgabe in funktionalen Programmiersprache

spezielle Monaden-Syntax; unterstützt durch den Kompiler

Monaden sind als Typklasse implementiert.

Das Typklassensystem und der Typchecker in Haskell sind wesentlich für die korrekte und sichere Verwendung von Monaden.

- Monade mit **Typkonstruktor** m
- Objekte vom Typ $(m\ a)$ sind Aktionen mit Ausgabety a
- Monaden-Methoden sind:
 - `>>=`: sequentielle Verknüpfung von zwei Aktionen
 - `return`: Erzeugung einer Aktion

Manchmal sind schon Aktionen vordefiniert

Beachte: Monad ist Konstruktorklasse

Programmieren = Konstruktion zusammengesetzter Aktionen.

Ergebnis erhält man durch Anwendung:
(Kombinierte Aktion) angewendet auf (Eingabe)

kein direkter Zugriff auf den inneren Zustand.

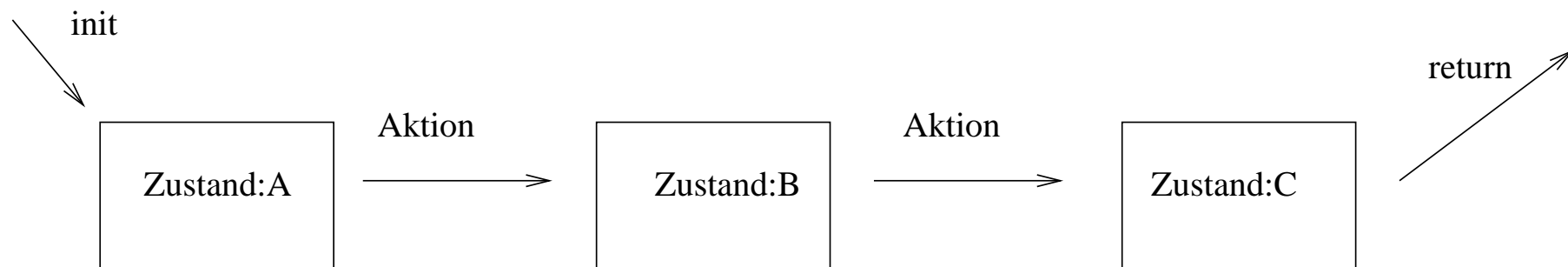
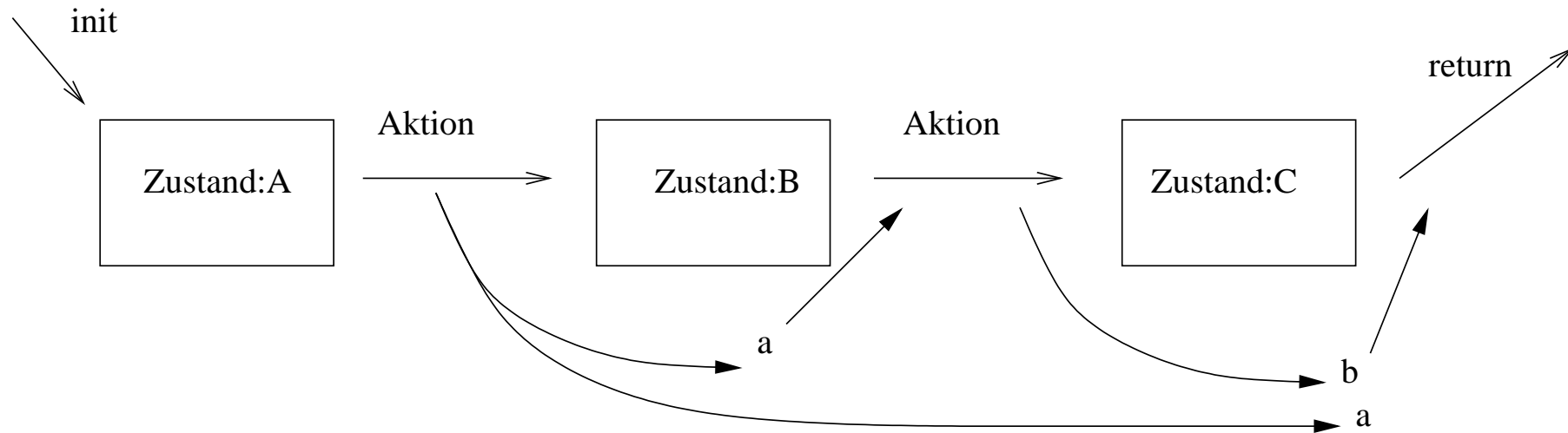


Bild mit Ausgabe der Aktionen:



Definition der Typklasse Monad im Haskell-Prelude:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b      --- (Bind)
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  -- Minimal complete definition: (>>=), return
  p >> q = p >>= \ _ -> q
  fail s = error s
```

Der Operator `>>=` wird auch „Bind“ genannt.

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

`return a` leere Aktion mit Ausgabe a
`a1 (>>=) a2` Hintereinanderausführung von a1 und a2
 wobei die Ausgabe von a1 an a2 weitergegeben wird.
`a1 (>>) a2` Hintereinanderausführung von a1 und a2
 ohne Weitergabe der Ausgabe

```
a1 >>= (\x -> a2 >>= (\y -> a3 >>= \_ -> return (x,y)))
```

kann man einfacher schreiben als:

```
do {x <- a1; y <- a2; a3; return (x,y)}
```

- Zuerst Aktion a1, dann Aktion a2, dann Aktion a3
- Die Ausgabe von Aktion a1 is x ;
- Die Ausgabe von Aktion a2 is y ;
- Ausgaben können von späteren Aktionen verwendet werden.
hier `return (x,y)`

```
do {x <- a1; y <- a2; a3; return (x,y)}
```

Vorsicht `x <- a1`
bedeutet **nicht**, dass `a1` and `x` gebunden wird!
sondern: der Wert der Aktion `a1` wird an `x` gebunden

Übersetzungsregeln für die do-Notation:

$$\begin{aligned} \text{do } \{x \leftarrow e; s\} &= e \gg= \backslash x \rightarrow \text{do } \{s\} \\ \text{do } \{e; s\} &= e \gg \text{do } \{s\} \\ \text{do } \{e\} &= e \end{aligned}$$

Monadengesetze:

$$1: \text{return } x \gg= f = f \ x$$

$$2: m \gg= \text{return} = m$$

$$3: m1 \gg= (\backslash x \rightarrow m2 \gg= \backslash y \rightarrow m3) = (m1 \gg= \backslash x \rightarrow m2) \gg= \backslash y \rightarrow m3$$

falls $x \notin FV(m3)$

Eigenschaft der Bind-Operation mittels `do`:

$$\begin{array}{l} \text{do } \{ \\ \quad x \leftarrow m_1; \\ \quad y \leftarrow m_2; \\ \quad m_3 \end{array} = \begin{array}{l} \text{do } \{ \\ \quad y \leftarrow \text{do } \{ \\ \quad \quad x \leftarrow m_1; \\ \quad \quad m_2 \}; \\ \quad m_3 \end{array}$$

Leider kann das Typsystem bzw. der Compiler die Gültigkeit dieser Gesetze nicht prüfen, wenn ein Datentyp zur Instanz von Monade erklärt wird. Für Fehlverhalten ist der Programmierer verantwortlich.

```
data Maybe a = Nothing | Just a
```

wird Monade:

```
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return  = Just
  fail s  = Nothing
```

$$1: \text{return } x \gg= f = f \ x$$

$$2: m \gg= \text{return} = m$$

$$3: m1 \gg= (\backslash x \rightarrow m2 \gg= \backslash y \rightarrow m3) = (m1 \gg= \backslash x \rightarrow m2) \gg= \backslash y \rightarrow m3$$

falls $x \notin FV(m3)$

instance Monad Maybe where

Just x >>= k = k x

Nothing >>= k = Nothing

return = Just

fail s = Nothing

Methode und Annahmen:

- call-by-need Variante eines Lambda-Kalküls
- `let` zur Modellierung des Sharing verwendet.
- Wir benutzen, dass Reduktion erhält die Gleichheit ebenso, dass einige Transformationsregeln korrekt sind.

1. `return x >>= f = f x`:

`Just x >>= f` ist als `f x` definiert.

2. `m >>= return = m`:

`m >>= Just`:

Fallunterscheidung über `m`

kann vom Typ her nur sein: `Just x`, `Nothing`, `bot`

- `m = Just x`. `Just x >>= Just` ergibt `Just x = m`.

- `m = Nothing` ergibt sich als Resultat `Nothing`, also ebenfalls `m`.

- `bot`: Ergibt Gleichheit

3. Wir verifizieren das dritte Monadengesetz; Fälle:

- $m_1 = \text{Nothing}$,
linke Seite Nothing .
rechte Seite, nach zweimaliger Anwendung der Definition:
 Nothing .
- $m_1 = \text{Just } s$,
Linke Seite: $\text{let } x = s \text{ in } m_2 \gg= \backslash y \rightarrow m_3$.
rechte Seite: $(\text{let } x = s \text{ in } m_2) \gg= \backslash y \rightarrow m_3$.
Stimmt, da diese let-Verschiebung korrekt ist.
- bot .

Der Datentype Liste kann als Monaden gesehen werden mit den Definitionen:

```
instance Monad [] where
  (x:xs) >>= f = f x ++ (xs >>= f)
  []          >>= f = []
  return x    = [x]
  fail s      = []
```

$[e \mid x \leftarrow a, \dots]$

$x \leftarrow a$ wird als Aktion gesehen:

Liste a hat als Ausgabe alle Listenelemente

1. $[x] \gg= f = f\ x \ ++\ ([] \gg= f) = f\ x.$

2. $m \gg= \lambda x \rightarrow [x]$ Fälle:

$m = []$ ergibt $[]$

$m = s:zs$ ergibt links:

$\text{let } x = s, xs = zs \text{ in } [x] \ ++\ (xs \gg= \lambda y \rightarrow [y]).$

rechte Seite ist gleich nach let-Verschiebung

$m = \text{bot}$ ergibt bot auf beiden Seiten

Für endliche Listen kann man Induktion durchführen,
für unendliche Listen: Induktion mit extra Basisfall bot

Auch hier ist der call-by-need-Kalkül zu verwenden!

Die `do`-Notation ist analog zu List-Comprehensions:

`[e | x <- xs; y <- ys]` entspricht der Notation:

```
do  {x <- xs;  
     y <- ys;  
     return e}
```

Dazu die äquivalente Bind-Definition,
die zu den Transformationsregeln für List-Comprehensions passt:

```
xs >>= f = concat (map f xs)
```

List Comprehensions können etwas mehr als die `do`-Notation:
Prädikate und Pattern.

Zustandsmonade kapselt Aktionen die einen Zustand verändern
erlaubt sequentielle Komposition von Aktionen

```
newtype StateTransformer s a = ST (s -> (a, s))

--      s      Zustands-Typ
--      a      Typ des Ausgabewerts
--      s -> (a, s)  Typ einer Aktion

apply :: StateTransformer s a -> s -> (a, s)
apply (ST f) x = f x

instance Monad (StateTransformer s) where
  return x      = ST $ \s -> (x, s)
  a1 >>= aa2    = ST action
    where
      action    = \s -> let (res, s') = apply a1 s
                          in apply (aa2 res) s'
```

Beispiel: Kontostand, der verändert werden kann:

--- Operationen auf dem Zustand:

```
incr = ST $ \s -> ((), s+1)
```

```
decr = ST $ \s -> ((), s-1)
```

```
hole = ST $ \s -> (s, s)
```

```
add n = ST $ \s -> ((), s +n)
```

```
sub n = ST $ \s -> ((), s - n)
```

```
clear = ST $ \_ -> ((), 0)
```

```
check p = ST $ \s -> (p s, s)
```

```
anfang = ST (\ein -> ((),ein))
```

```
test = (anfang >> incr >> incr >> add 3 >> hole)
zeigetest = (apply test 4444)
> (4449,4449) :: (Integer,Integer)
```

```
test2 = anfang >> incr >> add 3 >> sub 5
zeigetest2 = (apply test2 3333)
> ((),3332) :: ((),Integer)
```

Beispiele zu Kontrollstrukturen:

```
forever :: Monad m => m a -> m b
```

```
forever a = a >> forever a
```

```
repeatN :: (Monad a, Num b) => b -> a c -> a ()
```

```
repeatN 0 a = return ()
```

```
repeatN n a = a >> repeatN (n-1) a
```

Beispiele

```
repeatN :: (Monad m, Num a) => a -> m b -> m ()
```

```
repeatN 0 a = return ()
```

```
repeatN n a = a >> repeatN (n-1) a
```

```
test3 = anfang >> incr >> (repeatN 5 (add 3)) >> sub 5
```

```
zeigetest3 = (apply test3 6666)
```

```
> (((),6677) :: ((),Integer)
```

`for` hat eine Liste als Eingabe
und eine Funktion fa ,
die aus jedem Listenelement eine Aktion erzeugt:

```
-- Aktion pro Element einer Liste
for :: Monad m => [a] -> (a -> m c) -> m ()
for []      fa = return ()
for (n:ns)  fa = fa n >> for ns fa

testprintZahlen = for [1..10] print
```

```
for []      fa = return ()  
for (n:ns)  fa = fa n >> for ns fa
```

Kann so nicht programmiert werden:

```
forl xs f    = do {x <- xs; f x; return ()}
```

Grund: Kombination von 2 Monaden ist nicht möglich
Liste und IO

Typ von $\gg=$ $:: \text{forall } m \ b \ a. (\text{Monad } m) \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$

`sequence`: Ergebnisse der Aktionen in einer Liste

```
sequence []           = return []
sequence (a:as)      = do { r <- a;
                          rs <- sequence as;
                          return (r:rs) }
```

```
while :: Monad m => m Bool -> m a -> m ()
while check a = do {
    b <- check;
    if b then a >> while check a
    else return ()
}
```

```
test4 = anfang >> (add 10) >> while (check (\x -> x > 0)) decr
zeigetest4 = (apply test4 7777)
> ((()),0) :: ((()),Integer)
```

Ein/-Ausgabe mittels Aktionen

- Aktionen vom Typ `IO a`,
- Aktionen operieren auf “`World`”
- `a` ist der Typ des Ausgabe-Wertes

- Haskell-Programm = Aktion vom Typ `IO ()`.

Programm wird auf “`World`” angewendet

Keine Ausgabe im Haskell-Sinne

Return-Wert: `World` implizit verändert (z.B. eine DB wurde verändert)

Modell-Vorstellung

Der Typ `IO a` ist eine Abkürzung für Zustandsmonadentyp:

```
type IO a = World -> (a, World)
```

Eine Aktion vom Typ `IO a` gibt ein Paar zurück:

- Wert vom Typ `a`
- `World`: (veränderte Außenwelt)

Monadisches Programmieren erzwingt

single-threaded Benutzung von `World`:

`World` ist nicht kopierbar im Programm.

```
instance Monad IO where
  (>>=) = ...      (primbindIO)
  return = ...     (primretIO)
  fail s = ioError (userError s)
```

Die nicht angezeigten Funktionen sind vorgegeben und können **nicht** selbst implementiert bzw. ersetzt werden.

Nur vordefinierte Aktionen können direkt auf `World` angewendet werden

Vordefiniert sind:

```
getChar :: IO Char           -- World -> (Char,World)
putChar :: Char -> IO ()     -- Char -> (World -> ((),World))
```

IO-Programmierung setzt auf den vordefinierten Basis-IO-Aktionen auf

Es gibt weitere vordefinierte Aktionen

Die Aktionen `getChar` und `putChar` kann man kombinieren zu `echo` mittels `>>=`:

```
echo :: IO ()
echo = do x <- getChar
         putChar x
-- echo = getChar >>= putChar
```

Bind bewirkt sequentielle Ausführung mit Argumentübergabe

Benutzte Typinstanz von `>>=`

```
IO Char -> (Char -> IO ()) -> IO ()
```

Doppelecho:

```
echo = do x <- getChar
        putChar x
        putChar x
```

eine Zeile einlesen und ausgeben:

```
getline :: IO [Char]
getline = do c <- getChar
            if c == '\n' then return []
            else do cs <- getline
                  return (c:cs)

putLine :: [Char] -> IO ()
putLine [] = return ()
putLine (c:cs) = do {putChar c; putLine cs}
-- putLine (c:cs) = putChar c >> putLine cs
```

(Aus Haskell Hierarchical Libraries)

Externe Speicherzelle:

```
data IORef a    -- wie in ML
newIORef  :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef:: IORef a -> a -> IO ()

count :: Int -> IO Int
count n = do { r <- newIORef 0;
              loop r 1 }
  where
    loop r i | i > n    = readIORef r
              | otherwise = do { v <- readIORef r;
                                writeIORef r (v+i);
                                loop r (i+1)}
```

Testen der Aktion count:

```
*Main> do {n <- count 0;print n}
```

```
0
```

```
*Main> do {n <- count 10;print n}
```

```
55
```

Alle Interaktionen mit „`World`“ nur durch IO-Aktionen.

Bedingung

Es gibt immer nur ein Objekt `World` im gerade aktuellen Zustand

- `World` darf nicht kopiert werden
- D.h. es dürfen keine alten bzw, konkurrierenden Versionen aufgehoben werden
- Dies wird gewährleistet durch eine single-threaded Behandlung von `World`

Damit Ein-/Ausgabe in Haskell korrekt funktioniert und das Programmieren pur bleibt, gelten folgende **Eigenschaften** bzw. **Beschränkungen**:

- Ein Haskell-Programm ist eine IO-Aktion, d.h. vom Typ `IO ()`.
D.h. `main :: IO ()`
- IO-Aktionen sind nur kombinierbar mit `>>=` bzw `do`
Einzige benutzerdefinierbare Basisaktion mittels `return`
⇒ zuverlässige Sequentialisierung der IO-Aktionen
`do`-Notation hält die Beschränkungen ein.
- IO-Aktionen können “beliebig” im Programm verwendet werden.
- IO-Monade hat kein „Loch“. Das Ergebnis von `main` erhält man erst nach Beendigung des Programms

unsichere, nichtpure, Möglichkeit:

`unsafePerformIO`, umgeht alle Beschränkungen

Allerdings wird hierbei die **Reihenfolge der IOs und Zugriffe** **nicht** vom Compiler gewährleistet.

Implementierung von Bind ($\gg=$) und return
(unter der Annahme, dass World nicht kopierbar ist)

```
return :: a -> IO a
```

```
return a = \w -> (a,w)
```

```
(\gg=) :: IO a -> (a -> IO b) -> IO b
```

```
m \gg= k = \w -> case (m w) of (r,w') -> k r w'
```

Hier ist w' die neue World.

\Rightarrow Es gelten die **Monadengesetze**
unter verzögerter Auswertung (lazy evaluation)

Beta-Reduktion (d.h. call-by-name)
ist **nicht verträglich** mit IO-Aktionen und der IO-Monade.

Insbesondere gilt der Satz von Church-Rosser **nicht mehr**
für die Kombination der (kopierenden) Beta-Regel mit IO-Aktionen.

Erforderlich: verzögerte Auswertung mit Sharing.
d.h. ein call-by-need Lambda-Kalkül

Beispiel von Simon Peyton Jones.

```
do {c <- getChar; putChar c; putChar c}
```

Übersetzen unter Benutzung der Definition von `do` und `>>=`:

```
\w -> case getChar w of
  (c,w1) -> case putChar c w1 of
    (_,w2) -> putChar c w2
```

Würde der Satz von Church-Rosser gelten, wäre es dem Compiler erlaubt, Ersetzung von gleichen Ausdrücken zu machen:

```
\w -> case getChar w of
  (c,w1) -> case putChar c w1 of
    (_,w2) -> putChar (fst (getChar w)) w2
```

Fehler: die World-Referenz w wurde kopiert

verzögerte Auswertung mit Sharing verhindert diesen Effekt

Ein einfacher Taschenrechner wird programmiert
als [Beispiel](#) für
die Kombination einer Zustandsmonade mit der IO-Monade.

[File: Calculator2.hs](#)

Ein-Ausgabe der Einfachheit halber über die Konsole.

Taschenrechner: die Aufgabe

Eingabe:

- Zahlen aus Ziffern '0'-'9'
- Operatoren '+', '-', '*' und '/',
Dezimalpunkt, 'c' und '='.

Ergebnis: arithmetisches Ergebnis nach Linksklammerung

“123*456=” [ergibt](#): 56088.0

Zustand: speichert Operator und die beiden Operanden
Implementierung: speichert Funktion und Operator.

```
module Main where
```

```
type CalcState = (Float -> Float, Float)
```

```
type CalcTransformer a = CalcState -> (a, CalcState)
```

Folgende CalcTransformer benötigen wir für die Simulation

```
clear :: CalcTransformer ()
```

```
clear (g, 0.0) = ((), (id, 0.0))
```

```
clear (g, z)   = ((), (g, 0.0))
```

```
digit :: Float -> CalcTransformer ()
```

```
digit d (g, z) = ((), (g, z * 10.0 + d))
```

```
oper :: (Float -> Float -> Float) -> CalcTransformer ()
```

```
oper o (g, z) = ((), (o (g z), 0.0))
```

```
total :: CalcTransformer ()  
total (g, z) = ((), (id, g z))
```

```
readResult :: CalcTransformer Float  
readResult (g, z) = (g z, (g, z))
```

```
startState :: CalcState  
startState = (id, 0.0)
```

passende Operation zum Eingabe-Operator:

```
calcStep :: Char -> CalcTransformer ()
```

```
calcStep x
```

```
  | isDigit x = digit (fromInt (ord x - ord '0'))
```

```
  | x == '+'  = oper (+)
```

```
  | x == '-'  = oper (-)
```

```
  | x == '*'  = oper (*)
```

```
  | x == '/'  = oper (/)
```

```
  | x == 'c'  = clear
```

```
  | x == '='  = total
```

Implementierung:

```
calc :: String -> CalcTransformer Float
calc "" s      = readResult s
calc (x:xs) s = let (_,newState) = calcStep x s
                  in calc xs newState

main = fst (calc "123*456=" startState)
```

Vorher:

```
type CalcState = (Float -> Float, Float)
type CalcTransformer a = CalcState -> (a, CalcState)
```

Verallgemeinerung von CalcTransformer:

```
newtype StateTransformer s a = ST (s -> (a, s))
```

s: Typ des Zustands; a: Typ der Ausgabe

newtype: Aktionen, wie in einer Zustandsmonade
gekapselt mit ST

```
newtype StateTransformer s a = ST (s -> (a, s))
```

Hilfsfunktion zum Auspacken:

```
apply :: StateTransformer s a -> s -> (a, s)  
apply (ST action) = action
```

State-Transformatoren als Monade

StateTransformer s als Instanz von Monad

Wir definieren ($\gg=$) und return:

```
instance Monad (StateTransformer s) where
  return x = ST $ \s -> (x, s)
```

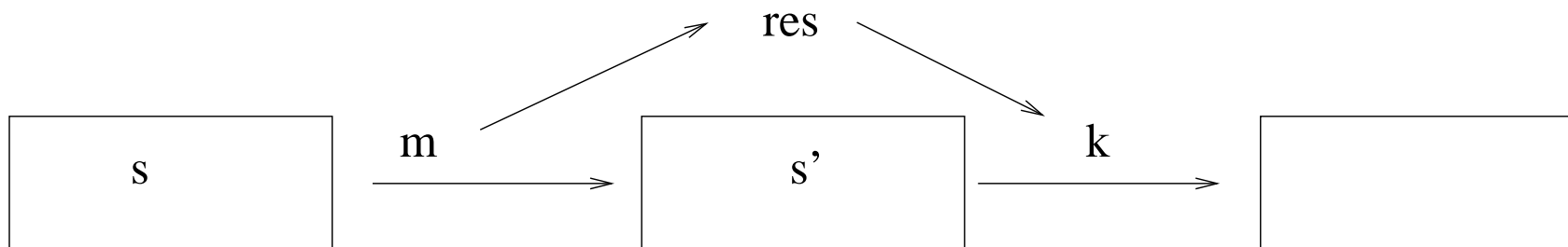
```
m >>= k = ST action
```

```
  where
```

```
    action s = apply (k res) s'
```

```
    where
```

```
      (res, s') = apply m s
```



(>>) automatisch definiert durch die Instanz-Definition

```
(>>) :: StateTransformer s a -> StateTransformer s b  
      -> StateTransformer s b
```

```
m >> k = m >>= \_ -> k
```

Rechner mittels StateTransformer s a:

```
type CalcTransformer = StateTransformer CalcState
```

```
clear :: CalcTransformer ()
```

```
clear = ST action
```

```
  where
```

```
    action (g, 0.0) = ((), (id, 0.0))
```

```
    action (g, z)   = ((), (g, 0.0))
```

```
digit :: Float -> CalcTransformer ()
```

```
digit d = ST $ \ (g, z) -> ((), (g, z * 10.0 + d))
```

```
oper :: (Float -> Float -> Float) -> CalcTransformer ()
```

```
oper o = ST $ \ (g, z) -> ((), (o (g z), 0.0))
```

Taschenrechner, monadisch (2)

```
total :: CalcTransformer ()
total = ST $ \(g, z) -> ((), (id, g z))

readResult :: CalcTransformer Float
readResult = ST $ \(g, z) -> (g z, (g, z))
```

calcStep bleibt; Neu ist:

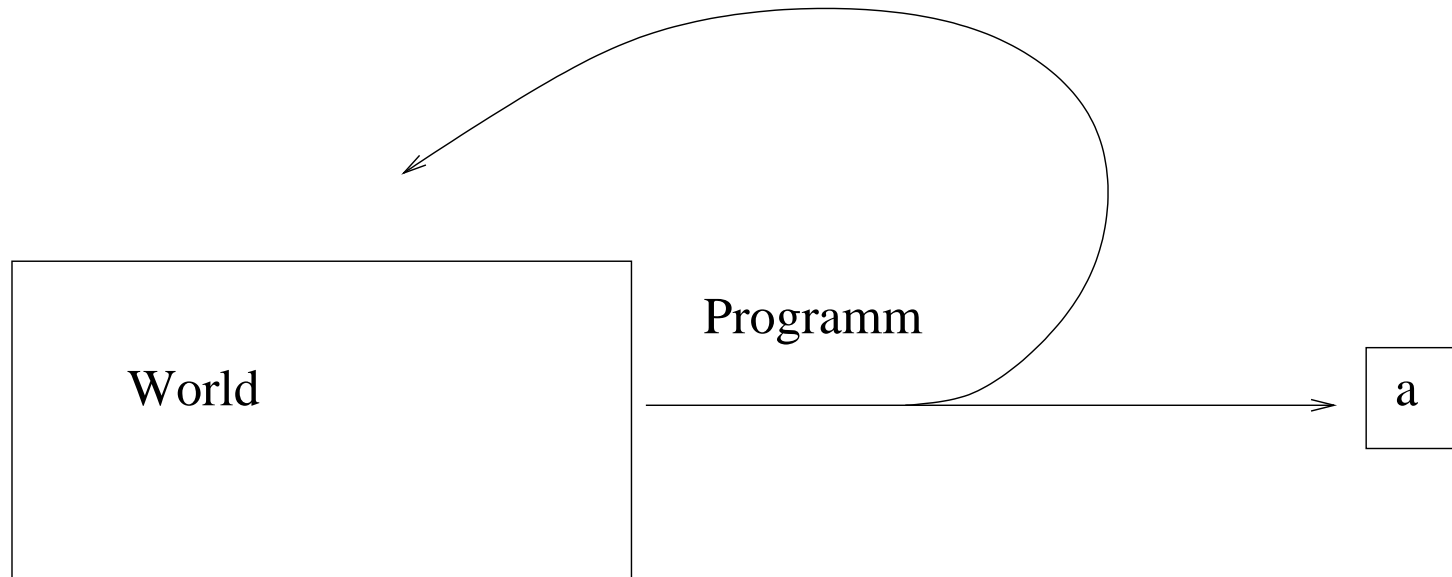
```
calc :: String -> CalcTransformer Float
calc ""      = readResult
calc (x:xs) = calcStep x >>
               calc xs
```

mit do-Notation:

```
calc ""      = readResult
calc (x:xs) = do calcStep x
                  calc xs
```

bzw.

```
calc (x:xs) = do { calcStep x; calc xs }
```



Programm :: IO a wobei IO a = World -> (a,World)

Beispiele für Ausgabe-Funktionen:

```
putChar :: Char -> IO ()  
putStr  :: String -> IO ()  
putStrLn :: String -> IO ()  
print   :: Show a => a -> IO ()
```

Beispiele für Eingabe-Funktionen:

```
getChar :: IO Char
```

```
getLine :: IO String
```

```
getContents :: IO String
```

```
interact :: (String -> String) -> IO ()
```

```
readIO :: Read a => String -> IO a
```

```
getLine :: Read a => IO a
```

Beispiele für und für Funktionen auf Dateien

```
writeFile :: FilePath -> String -> IO ()
```

```
appendFile :: FilePath -> String -> IO ()
```

```
readFile :: FilePath -> IO String
```

Eine Variation des Rechners mit IO ist

```
main = do k <- getLine
         let (g, z) = apply (calc k) startState
         print $ g
```

Zustandsmonade + andere Monade m:

```
newtype STT s m a = STT (s -> m (a, s))
```

```
apply :: STT s m a -> s -> m (a, s)
```

```
apply (STT f) = f
```

```
instance Monad m => Monad (STT s m) where
```

```
  return x = STT $ \ s -> return (x, s)
```

```
  a >>= k = STT action
```

```
    where action s = do (x, s') <- apply a s
```

```
      apply (k x) s'
```

```
type CalcTransformer = STT CalcState IO
```

IOMonad als Verallgemeinerung:
ioPrint und ioGetChar als Methoden

```
class Monad m => IOMonad m where  
  ioGetChar :: m Char  
  ioPrint :: Show a => a -> m ()
```

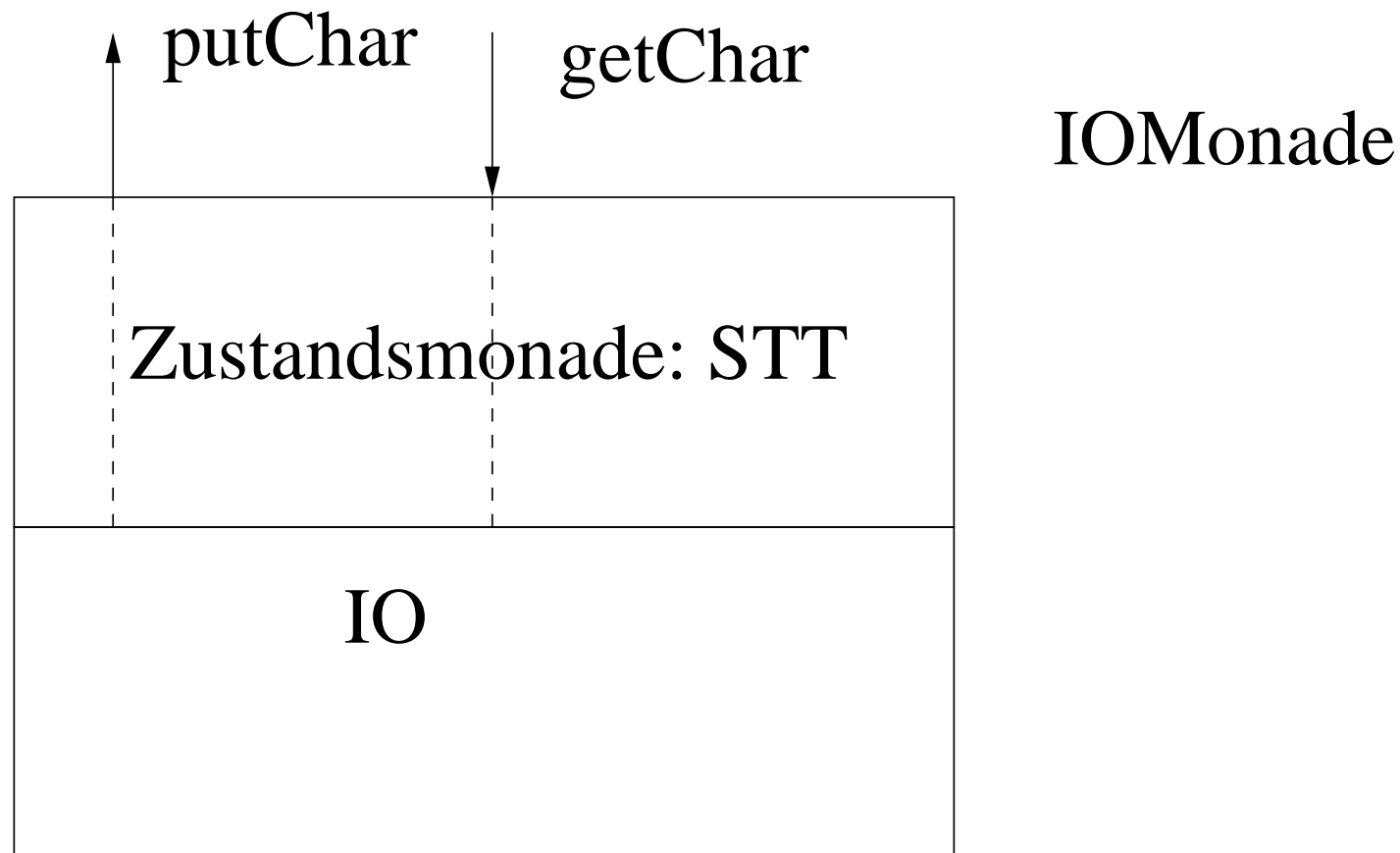
IO soll Instanz sein:

```
instance IOMonad IO where  
  ioGetChar = getChar  
  ioPrint = print
```

STT s m soll Instanz von IOMonad sein:

```
instance IOMonad m => IOMonad (STT s m) where
  ioGetChar = promote ioGetChar
  ioPrint x = promote $ ioPrint x
```

rekursive Instanz-Definition,
wobei s jeweils verschieden sein kann



apply und promote
als Entpacken bzw. Liften der Aktionen

```
class MonadTransformer t where
  promote :: Monad m => m a -> t m a

instance MonadTransformer (STT s) where
  promote g = STT $ \ s -> do {x <- g; return (x, s)}

promote :: (MonadTransformer t, Monad m) => m a -> t m a
```

Eine Monade,

die Zustandsaktionen und IO-Aktionen kombiniert,

und typsicher ist

File: Calculator6.hs

Leichte Erweiterungen:

Erweiterter Zustand,
Gleitkommazahlen als Eingabe

```
module Main where

import Char
import IO

main::IO ()
main = do
    hSetBuffering stdin NoBuffering
    hSetBuffering stdout NoBuffering
    apply calc startState
    return ()
```

```
type CalcState = ((String,Int), Double -> Double, Double)

startState :: CalcState
startState = ((" ",0),id, 0.0)

calcStep :: Char -> CalcTransformer ()
calcStep x
  | isDigit x = digit (fromInteger (toInteger (ord x - ord '0')))
  | x == '+'  = oper (+)
  | x == '-'  = oper (-)
  | x == '*'  = oper (*)
  | x == '/'  = oper (/)
  | x == 'c'  = clear
  | x == '='  = total
  | x == '.'  = komma
  | otherwise = tunichts
```

```
newtype STT s m a = STT (s -> m (a, s))
```

```
apply :: STT s m a -> s -> m (a, s)
```

```
apply (STT f) x = f x
```

```
instance Monad m => Monad (STT s m) where
```

```
  return x = STT $ \ s -> return (x, s)
```

```
  k1 >>= k2 = STT action
```

```
    where action s = do (x, s') <- apply k1 s  
                      apply (k2 x) s'
```

```
type CalcTransformer = STT CalcState IO
```

```
class Monad m => IO Monad m where
  ioGetChar :: m Char
  ioPrint :: Show a => a -> m ()
```

```
instance IO Monad IO where
  ioGetChar = getChar
  ioPrint = print
```

```
instance IO Monad m => IO Monad (STT s m) where
  ioGetChar = promote ioGetChar
  ioPrint x = promote $ ioPrint x
```

```
class MonadTransformer t where
  promote :: Monad m => m a -> t m a
```

```
instance MonadTransformer (STT s) where
  promote g = STT $ \ s -> do {x <- g; return (x, s)}
```

```
clear :: Monad m => STT CalcState m ()
clear = STT action
  where
    action (str,g, 0.0) = return ((), (("c",0),id, 0.0))
    action (str,g, z)   = return ((), (("c",0), g, 0.0))

total :: IOMonad m => STT CalcState m ()
total  = STT $ \(_,g, z) ->
  do {ioPrint $ g z; return ((), (("t",0),id, g z))}

digit :: Monad m => Double -> STT CalcState m ()
digit d = STT $ \(str,dignum),g, z) ->
  if str == "t" then return ((), (("d",0),g, d))
  else if str == "." then
    return ((), (("d",0),g, z + (0.1^dignum)*d))
  else return ((), (("d",0), g, z*10.0 + d))
```

```
komma :: Monad m => STT CalcState m ()
komma  = STT $ \((str,_) ,g, z) ->
    if str == "d" || str == "c" || str == " " || str == "o"
    then return ((), ((".",1),g, z))
    else if str == "t"
    then return ((), ((".",1),g, 0.0))
    else return ((), ((".",30),g, z))
```

```
tunichts :: Monad m => STT CalcState m ()
tunichts  = STT $ \(sta,g, z) -> return ((), (sta,g, z))
```

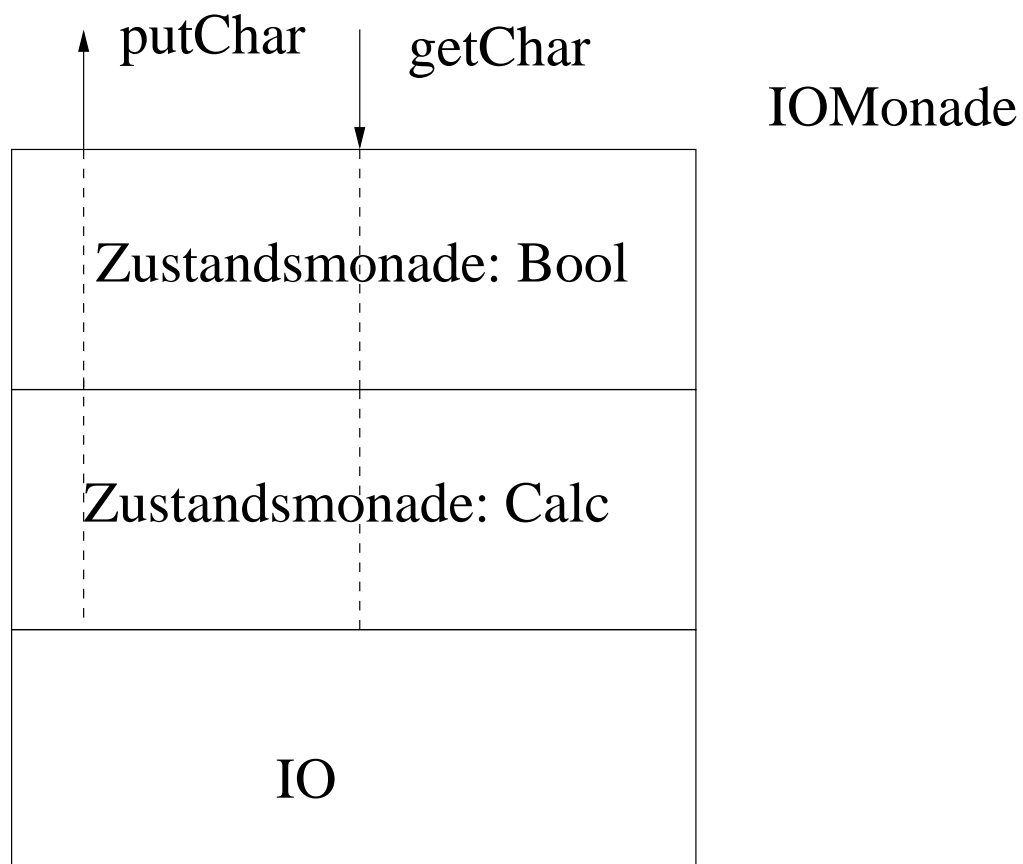
```
oper :: Monad m => (Double -> Double -> Double) -> STT CalcState m ()
oper o  = STT $ \(_,g, z) -> return ((), (("o",0), o (g z), 0.0))
```

```
readResult :: Monad m => STT CalcState m Double
readResult = STT $ \(_,g, z) -> return (g z, (("r",0),g, z))
```

```
calc :: CalcTransformer ()
calc =
  do {k <- ioGetChar;
      if k == '\n'
      then do {x <- readResult;
               ioPrint x}
      else do {calcStep k;
               calc}}
```

- Taschenrechner als Zustandsmonade
- Boolescher Taschenrechner als Zustandsmonade
Kommandos: T,F,S,N,A,O
für True, False, Show, Not, And, Or
- IO
- Sourcefile: `calculator6-comb.hs`

Beispiel: Kombination von 3 Monaden



Vermitteln und Durchreichen mittels:
apply und promote

```
module Main where
import Char
import IO

main::IO ()
main = do
    hSetBuffering stdin NoBuffering
    hSetBuffering stdout NoBuffering
    combapply calcComb startStateBool startStateCalc
    return ()

-- combapply calcComb startStateBool startStateCalc ::
--           IO (((), BoolState), CalcState)

type CalcState = ((String,Int), Double -> Double, Double)

type BoolState = (Bool, Bool -> Bool)
```

```
newtype STT s m a = STT (s -> m (a, s))
```

```
apply :: STT s m a -> s -> m (a, s)
```

```
apply (STT f) x = f x
```

```
combapply (STT f) b x = apply (f b) x
```

```
instance Monad m => Monad (STT s m) where
```

```
  return x = STT $ \ s -> return (x, s)
```

```
  k1 >>= k2 = STT action
```

```
    where action s = do (x, s') <- apply k1 s  
                      apply (k2 x) s'
```

```
type CalcTransformer = STT CalcState IO
```

```
type CalcTransformerComb = STT BoolState CalcTransformer
```

```
class Monad m => IO Monad m where
  ioGetChar :: m Char
  ioPrint :: Show a => a -> m ()
```

```
instance IO Monad IO where
  ioGetChar = getChar
  ioPrint = print
```

```
instance IO Monad m => IO Monad (STT s m) where
  ioGetChar = promote ioGetChar
  ioPrint x = promote $ ioPrint x
```

```
class MonadTransformer t where
  promote :: Monad m => m a -> t m a
```

```
instance MonadTransformer (STT s) where
  promote g = STT $ \ s -> do {x <- g; return (x, s)}
```

```
startStateCalc :: CalcState
```

```
startStateCalc = ((" ",0),id, 0.0)
```

```
calcStep :: Char -> CalcTransformer ()
```

```
calcStep x
```

```
  | isDigit x = digit (fromInteger (toInteger (ord x - ord '0')))
```

```
  | x == '+'  = oper (+)
```

```
  | x == '-'  = oper (-)
```

```
  | x == '*'  = oper (*)
```

```
  | x == '/'  = oper (/)
```

```
  | x == 'c'  = clear
```

```
  | x == '='  = total
```

```
  | x == '.'  = komma
```

```
  | otherwise = tunichts
```

```
startStateBool :: BoolState
```

```
startStateBool = (True,id)
```

```
calcStepComb :: Char -> CalcTransformerComb ()
```

```
calcStepComb x
```

```
  | x == 'T' = STT $ \(b,g) -> return ((),(g True,id))
```

```
  | x == 'F' = STT $ \(b,g) -> return ((),(g False,id))
```

```
  | x == 'N' = STT $ \(b,g) -> return ((),(not b,id))
```

```
  | x == 'O' = STT $ \(b,g) -> return ((),(b,\x-> b || x) )
```

```
  | x == 'A' = STT $ \(b,g) -> do {print "=";
                                   return ((),(b,\x-> b && x) )}
```

```
  | x == 'S' = STT $ \(b,g) -> do {ioPrint "="; ioPrint b;
                                   return ((),(b,g))}
```

```
  | otherwise = promote ( calcStep x)
```

```
clear :: Monad m => STT CalcState m ()
```

```
clear = STT action
```

```
  where
```

```
    action (str,g, 0.0) = return ((), (("c",0),id, 0.0))
```

```
    action (str,g, z)   = return ((), (("c",0), g, 0.0))
```

```
total :: IO Monad m => STT CalcState m ()
```

```
total = STT $ \(_,g, z) -> do {ioPrint $ g z; return((),(("t",0),id,g z))}
```

```
digit :: Monad m => Double -> STT CalcState m ()
```

```
digit d    = STT $ \((str,dignum),g, z) ->
```

```
  if str == "t" then return ((), (("d",0),g, d))
```

```
  else if str == "." then return ((), ((".", dignum+1),g, z + (0.1^dignum))
```

```
  else return ((), (("d",0), g, z*10.0 + d))
```

```
komma :: Monad m => STT CalcState m ()
komma   = STT $ \((str,_) ,g, z) ->
  if str == "d" || str == "c" || str == " " || str == "o"
  then return ((), ((".",1),g, z))
  else if str == "t"
    then return ((), ((".",1),g, 0.0))
    else return ((), ((".",30),g, z))

tunichts :: Monad m => STT CalcState m ()
tunichts   = STT $ \ (sta,g, z) -> return ((), (sta,g, z))

oper :: Monad m => (Double -> Double -> Double) -> STT CalcState m ()
oper o     = STT $ \ (_,g, z) -> return ((), ( ("o",0), o (g z), 0.0))

readResult :: Monad m => STT CalcState m Double
readResult = STT $ \ (_,g, z) -> return (g z, ( ("r",0),g, z))

readResultComb :: Monad m => STT BoolState m Bool
readResultComb = STT $ \ (b,g) -> return (g b, (g b,g))
```

```
calcComb :: CalcTransformerComb ()
calcComb =
  do {k <- ioGetChar;
      if k == '\n'
      then do {x <- promote readResult;
               y <- readResultComb;
               ioPrint (y,x)}
      else do {calcStepComb k;
               calcComb}}
```

```
*Main> main
```

```
20+5*4=100.0
```

```
TOFS=""
```

```
True
```

```
TAFS=""
```

```
False
```

```
(False,100.0)
```

```
*Main>
```