



Technische
Universität
Braunschweig



Programmiersprachen und Übersetzer

12 - Das Funktionale Programmierparadigma

Christian Dietrich

Sommersemester 2024

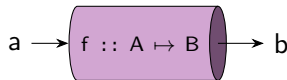
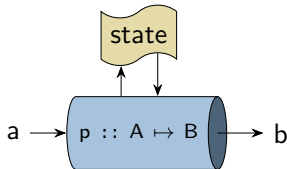
- Kleine Programme sind **einfach**, große Softwareprojekte sind **schwierig**.
 - Viele EntwicklerInnen, viele Standorte, viele Zeitzonen
 - Riesige Codebasis und **interagierende Komponenten**
 - Viele Prozeduren (Algorithmen) und viele Objekte (Daten)



- Begrenzte geistige Kapazität: **Veränderlichen Zustand** sieht man nicht.
 - ↔ Ausführung von Code ist meist sequentiell und lokal
 - Der aktuelle Zustand ist die **Akkumulation aller Änderungen**
 - Ohne Einhaltung von Invarianten leicht **unübersichtlich**

(Reine) funktionale Programmierung erhöht die Lokalität von Änderungen.

- **Rein funktional:** Prozeduren werden zu mathematischen Funktionen.
 - Das Ergebnis einer (echten) Funktion hängt nur von ihren Parametern ab.
 - Kein Seitenkanal über den globalen Zustand
 - Funktionsausführung hat **keine Seiteneffekte!**



- Mischformen mit veränderlichem Zustand

functional style

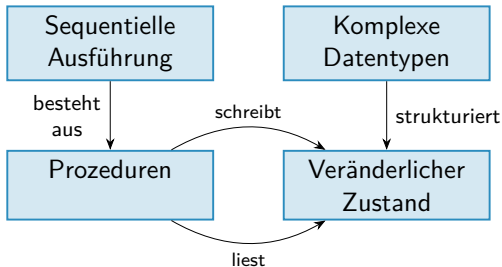
- Rein funktionale Programmierung ist auch schwierig
- Viele Programmiersprachen haben **funktionale Elemente**

➤ Verhältnis zu anderen Programmierparadigmen

Paradigmen

Das imperative
Paradigma

Konzepte



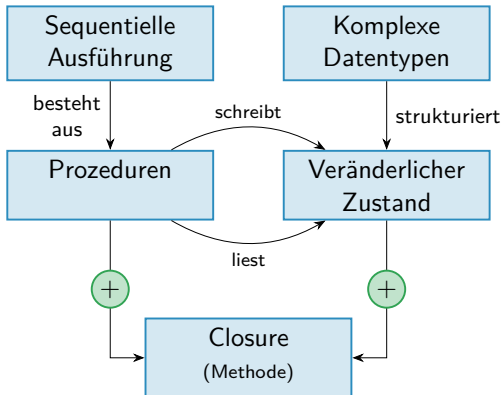
➤ Verhältnis zu anderen Programmierparadigmen

Paradigmen

Das imperative
Paradigma

Das objektorientierte
Paradigma

Konzepte



➤ Verhältnis zu anderen Programmierparadigmen

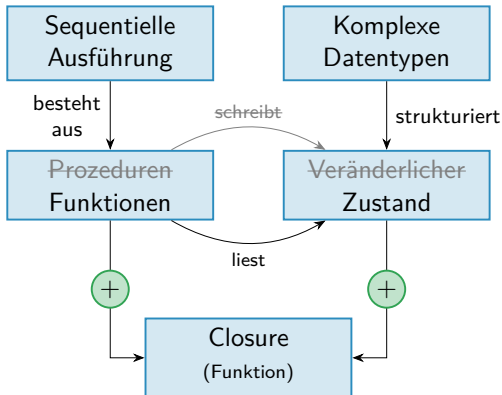
Paradigmen

Das imperative
Paradigma

Das objektorientierte
Paradigma

Das funktionale
Paradigma

Konzepte



- Einige Sprachkonzepte werden sinnlos...
 - Wiederbeschreibbare **Variablen** sind veränderlicher Zustand ⇒ **Verboten**
 - Veränderbare Objekte wären veränderlicher Zustand ⇒ **Verboten**
 - **Schleifen**iterationen kommunizieren über ihre Seiteneffekte ⇒ **Sinnlos**
 - Sequenzierte **Statements** haben keinen Rückgabewert ⇒ **Sinnlos**
- Andere Sprachkonzepte gewinnen an Wichtigkeit!
 - **Immutable Objekte**: Werte- und Referenzmodell fallen zusammen!
 - **Ausdrücke**: Jede Operation braucht einen Rückgabewert!
 - **Rekursion**: Wiederholte Ausführungen nur durch Rekursion!
 - **Funktionen**: Funktionsaufrufe modellieren abhängigkeiten Operationen!
- Weitere wichtige Fragen, die direkt aufkommen:
 - Wie soll man so bitte übersichtlich programmieren?!
 - Wie kann das bitte nicht ineffizient sein?!
 - Wie soll ich Ein- und Ausgabe machen?!

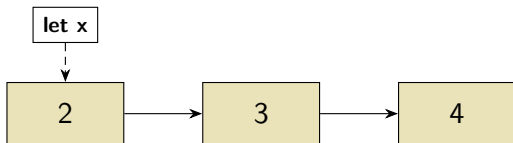
Viele Datentypen sind darauf ausgelegt, dass ihre Objekte während ihrer Lebenszeit verändert werden. Das haben wir leider **verboten!**

- Objekte werden nur initialisiert, aber nie verändert.
 - Jede Referenz erlaubt nur das Lesen des Objekts
 - "Modifikation" nur über partielle Kopien und parametrisierte Konstruktoren
 - Effizienz: Übersetzer kann viele dieser Kopien durch Optimierungen entfernen

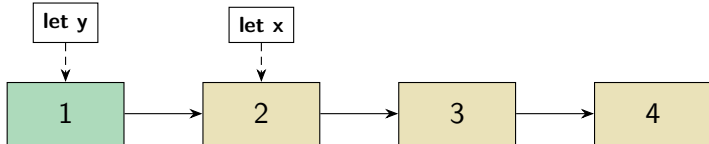
```
func add(a : Point, b : Point) : Point {  
    return new Point(a.x+b.x, a.y+b.y);  
}
```

Pseudocode

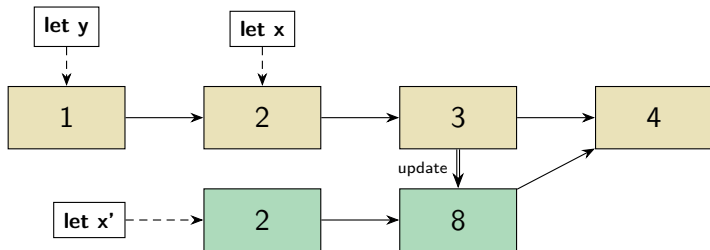
- Bei hierarchisch verzweigten Datentypen funktioniert das besonders gut
 - **Einfach verkettete Listen**: Vorne anhängen ist einfach
 - **Bäume**: Unterbäume können bei einem Knotenupdate übernommen werden



- Funktionale Programmierung arbeitet viel mit einfach verketteten Listen
 - **Vorne anhängen** erfordert keine Änderung der vorhandenen Listenelemente
 - **Gebundene Namen** bleiben valide und verändern ihren Wert nicht
 - **Updates** erfordern Elementkopien bis zur Änderungsstelle
- Einfach verkettete Listen sind eine **rekursive** Datenstruktur
⇒ Passt hervorragend zur rekursiven Struktur funktionaler Programme



- Funktionale Programmierung arbeitet viel mit einfach verketteten Listen
 - **Vorne anhängen** erfordert keine Änderung der vorhandenen Listenelemente
 - **Gebundene Namen** bleiben valide und verändern ihren Wert nicht
 - **Updates** erfordern Elementkopien bis zur Änderungsstelle
- Einfach verkettete Listen sind eine **rekursive** Datenstruktur
⇒ Passt hervorragend zur rekursiven Struktur funktionaler Programme



- Funktionale Programmierung arbeitet viel mit einfach verketteten Listen
 - **Vorne anhängen** erfordert keine Änderung der vorhandenen Listenelemente
 - **Gebundene Namen** bleiben valide und verändern ihren Wert nicht
 - **Updates** erfordern Elementkopien bis zur Änderungsstelle
- Einfach verkettete Listen sind eine **rekursive** Datenstruktur
⇒ Passt hervorragend zur rekursiven Struktur funktionaler Programme

Problem: Wir wollen viele Objekte erzeugen und wieder zerlegen.

```
enum PointKind {  
    XY,  
    XYZ,  
};
```

Rust

- Funktionale Sprachen bieten Aufzählungstypen (`enum`) mit Nutzlast
 - Endliche (aufzählbare) Anzahl an Varianten

Problem: Wir wollen viele Objekte erzeugen und wieder zerlegen.

```
enum Point {  
    XY(i32, i32),  
    XYZ(i32, i32, i32)  
};
```

Rust

- Funktionale Sprachen bieten Aufzählungstypen (`enum`) mit Nutzlast
 - Endliche (aufzählbare) Anzahl an Varianten
 - Jede Variante kann unterschiedliche Felder mit sich führen

Problem: Wir wollen viele Objekte erzeugen und wieder zerlegen.

```
enum Point {  
  XY(i32, i32),  
  XYZ(i32, i32, i32)  
};
```

Rust

```
fn handle(p : Point);  
let p1 = Point::XY (0, 3);  
let p2 = Point::XYZ(0, 3, 0)  
handle(p1); handle(p2);
```

Rust

- Funktionale Sprachen bieten Aufzählungstypen (**enum**) mit Nutzlast
 - Endliche (aufzählbare) Anzahl an Varianten
 - Jede Variante kann unterschiedliche Felder mit sich führen
 - Aufzählungsname wird Konstruktor
 - Tagged variante Records (**union**) sind typsicher und polymorph

Problem: Wir wollen viele Objekte erzeugen und wieder zerlegen.

```
enum Point {  
  XY(i32, i32),  
  XYZ(i32, i32, i32)  
};
```

Rust

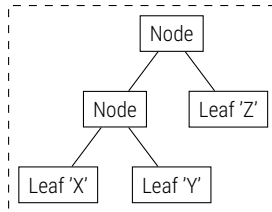
```
fn handle(p : Point);  
let p1 = Point::XY (0, 3);  
let p2 = Point::XYZ(0, 3, 0)  
handle(p1); handle(p2);
```

Rust

- Funktionale Sprachen bieten Aufzählungstypen (**enum**) mit Nutzlast
 - Endliche (aufzählbare) Anzahl an Varianten
 - Jede Variante kann unterschiedliche Felder mit sich führen
 - Aufzählungsname wird Konstruktor
 - Tagged variante Records (**union**) sind typsicher und polymorph
- Rekursive generisch-polymorphe typsichere Aufzählungstypen

```
data Tree T = Leaf T  
            | Node (Tree T) (Tree T)  
freeTree :: Tree Char  
freeTree = Node  
    (Node  
        (Leaf 'X')  
        (Leaf 'Y'))  
    (Leaf 'Z')
```

Haskell



Tree Char
==
Ein Baum
aus Char



Deklaration und Konstruktion von Aufzählungsobjekten ist sehr kompakt.

- **Destructuring Bind:** Zerlegen eines Objekts in seine Einzelteile
 - Muster mit der gleichen Struktur aber freien Namen
 - Objekt wird auf dieses Muster aufgeteilt
 - Freie Namen werden an die zerlegten Subobjekte gebunden

```
struct ObjT { a : i32, b : i32 }  
let ObjT {a: y0, b: y1} = obj;  
  
let (x0, x1) = (1, 2);
```

Rust



Deklaration und Konstruktion von Aufzählungsobjekten ist sehr kompakt.

■ Destructuring Bind: Zerlegen eines Objekts in seine Einzelteile

- Muster mit der gleichen Struktur aber freien Namen
- Objekt wird auf dieses Muster aufgeteilt
- Freie Namen werden an die zerlegten Subobjekte gebunden

```
struct ObjT { a : i32, b : i32 }  
let ObjT {a: y0, b: y1} = obj;  
  
let (x0, x1) = (1, 2);
```

Rust

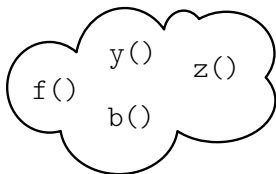
```
let norm = match p1 {  
  Point::XYZ(x,y,z) => x+y+z,  
  Point::XY (x,y)   => x+y,  
}
```

Rust

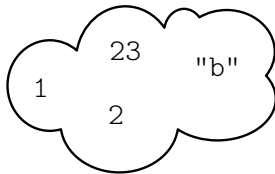
■ Pattern Matching: Selektionsoperation mittels Destructuring Bind

- Ähnlich, aber mächtiger als ein switch-case in C
- Vergleich eines Objekts gegen mehrere Muster und Destructuring Bind
- Der entsprechende Arm wird mit den gebundenen Namen ausgeführt
- Semantische Analyse kann prüfen, ob alle Fälle abgedeckt sind

Funktionen



Objekte



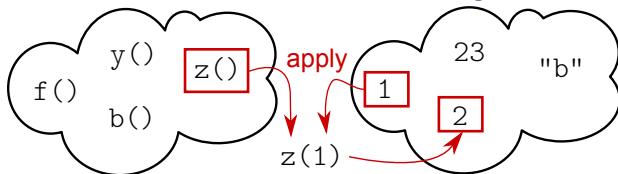
■ **Bisher:** Wir hatten hauptsächlich **Funktionen erster Ordnung**

- Funktionen und Objekte sind zwei getrennte Universen
- Funktionsanwendung kombiniert eine Funktion und mehrere Objekte

⇒ Definition: Eine Funktion erster Ordnung hat nur passive Objekte als Argument

Funktionen

Objekte



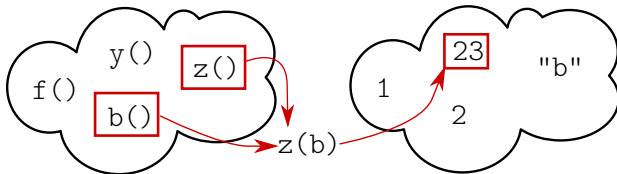
■ **Bisher:** Wir hatten hauptsächlich Funktionen erster Ordnung

- Funktionen und Objekte sind zwei getrennte Universen
- Funktionsanwendung kombiniert eine Funktion und mehrere Objekte

⇒ Definition: Eine Funktion erster Ordnung hat nur passive Objekte als Argument

Funktionen

Objekte



■ **Bisher:** Wir hatten hauptsächlich Funktionen erster Ordnung

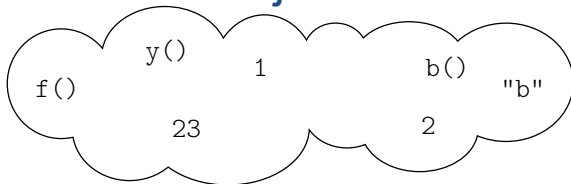
- Funktionen und Objekte sind zwei getrennte Universen
- Funktionsanwendung kombiniert eine Funktion und mehrere Objekte

⇒ Definition: Eine Funktion erster Ordnung hat nur passive Objekte als Argument

■ Funktionen höherer Ordnung haben Funktionen als Parameter

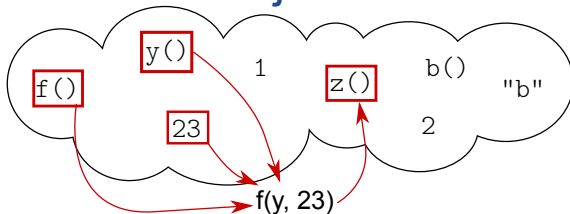
- Viele imperative Sprachen bieten bereits Funktionszeiger
- **Aber:** Funktionen sind immer noch nicht wirklich gleichberechtigt!

Objekte



- Im funktionalen Paradigma sind Funktionen wirklich gleichberechtigt.
 - Kein kategorieller Unterschied zwischen Funktionen und Objekten
 - Funktionen können Argument **und** Rückgabewert sein!
 - Funktionen sind **WIRKLICH** gleichberechtigt!
 - Funktionen können während der Laufzeit neu erzeugt werden.
 - Funktionen können komponiert und partiell angewendet werden.
- ⇒ Im funktionalen Paradigma werden die algorithmischen Aspekte einer Sprache gestärkt!

Objekte



- Im funktionalen Paradigma sind Funktionen wirklich gleichberechtigt.
 - Kein kategorieller Unterschied zwischen Funktionen und Objekten
 - Funktionen können Argument **und** Rückgabewert sein!
- Funktionen sind **WIRKLICH** gleichberechtigt!
 - Funktionen können während der Laufzeit neu erzeugt werden.
 - Funktionen können komponiert und partiell angewendet werden.

⇒ Im funktionalen Paradigma werden die algorithmischen Aspekte einer Sprache gestärkt!

Obwohl Python keine rein(!) funktionale Programmiersprache ist, werde ich die Beispiele, soweit es geht, als Python-Code zeigen.

- Funktionen höherer Ordnung können **Auswertungsstrategien** abstrahieren
 - „Äußere“ Funktion steuert die Auswertung, die übergebene macht die Arbeit.
 - Die übergebene Funktion kann mehrfach und mit unterschiedlichen Parametern aufgerufen werden. Die Ergebnisse können kombiniert werden.
 - Die Auswertungsstrategie ist parametrisierbar und wird so **wiederverwendbar**!

```
def outer(fn, arr):  
    return fn(arr[0]) + fn(arr[1])  
  
outer(innerA, [23, 42]) # => 6500  
outer(innerB, [23, 42]) # => -65
```

```
def innerA(x):  
    return x * 100  
  
def innerB(x):  
    return x * -1
```

- Wir hatten dies bereits bei `traverse()` und `fixpoint()` benutzt.

- Bei funktionaler Programmierung braucht man ständig kleine Funktionen
 - Entfernte Funktionsdefinition an anderer Stelle zerstört die Code-Lokalität
 - Außerdem ist es aufwendig, sich ständig neue Funktionsnamen auszudenken

`lambda` `p1, p2, p3` : `p1 * p2 + p3`

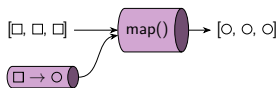
Parameter Ein Ausdruck

- **Lösung:** `Lambda-Ausdrücke` erzeugen ein Funktionsobjekt ohne Namen
 - Inspiriert vom, aber nicht immer äquivalent zum, Lambda-Kalkül
 - Bestandteile: Parameterliste und **ein** Ausdruck
 - Funktionsobjekt kann später aufgerufen werden

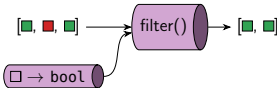
```
outer(lambda x: x * 100, [23,42]) # => 6500
outer(lambda x: x * -1, [23,42]) # => -65
```

```
add = lambda x, y: x+y
add(1,2) # => 3
```

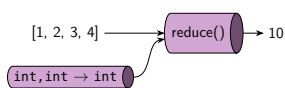

- Listen sind die funktionale „Brot-und-Butter“-Datenstruktur
 - Funktionen höherer Ordnung bieten standardisierte Auswertungsstrategien
 - Diese Funktionen ersetzen teilweise iterative Sprachelemente



`map()` bildet jedes Element mithilfe der übergebenen Funktion ab und konstruiert daraus eine neue Ergebnisliste.



`filter()` prüft jedes Element mittels eines Prädikats und konstruiert eine Liste mit „erfolgreichen“ Elementen.

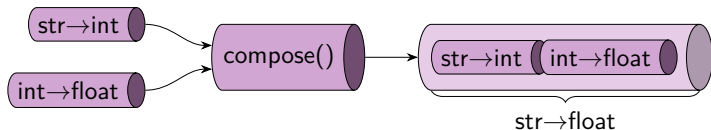


`reduce()` reduziert eine Liste zu einem Wert, indem die übergebene Funktion als Infixoperator angewendet wird.



Komposition von Funktionen

- **Funktionskomposition** ist die Hintereinanderausführung von Funktionen
 - Aus zwei Funktionsobjekten kreieren wir ein neues Funktionsobjekt.
 - Die neue Funktion wendet beide Funktionen in Reihe an.
Das Ergebnis der ersten wird zur Eingabe der zweiten.
 - Rückgabetypen und Parametertypen müssen zueinander passen.



```
def inc(x):  
    return x + 1  
  
def compose(f, g):  
    return lambda x: f(g(x))  
  
x = compose(inc, inc)  
x(0) # => 2
```

Python

```
inc = \x -> x +1  
x   = inc . inc
```

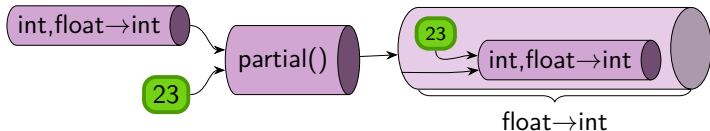
x(0) # => 2

Haskell

Haskell bietet Komposition sogar als **Infixoperator** „.“, angelehnt an $f \circ g$:

■ Partielle Anwendung bindet einige Parameter einer Funktion

- **Beobachtung:** Funktionen brauchen all ihre Parameter zur Ausführung
- **Idee:** Wir könnten einige Parameter mit bekannten Argumenten vorbelegen
- Erzeugen eines Funktionsobjekts mit geringerer Stelligkeit



```
def add(a,b):  
    return a + b  
  
def partial(f, *a0):  
    return lambda *a1: f(*(a0+a1))  
  
inc = partial(add, 1)  
inc(0) # => 1
```

Python

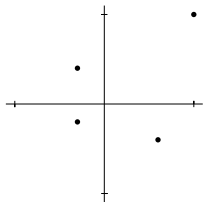
```
add :: Int -> (Int -> Int)  
add a b = a + b
```

```
inc = (add 1)  
inc(1) -- => 2
```

Haskell

Haskell bietet **Currying**, wo jede Funktion standardmäßig partiell anwendbar ist.

➤ Beispiel: Eine Verarbeitungskette



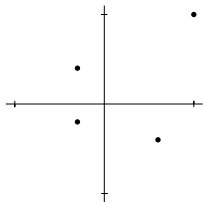
Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3,0.4),  (-0.3, -0.2),  
          (0.6,-0.4),  (1, 1)]
```

points

List(Point)

➤ Beispiel: Eine Verarbeitungskette



Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3,0.4),  (-0.3, -0.2),  
          (0.6,-0.4),  (1, 1)]
```

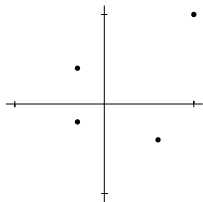
```
def norm(N, point): ...
```

■ Generische Normierungsfunktion $\| \cdot \|_N$

points

List(Point)

➤ Beispiel: Eine Verarbeitungskette



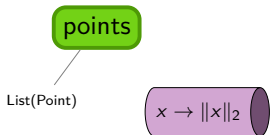
Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3,0.4),  (-0.3, -0.2),  
          (0.6,-0.4),  (1, 1)]
```

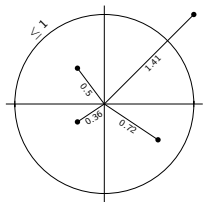
```
def norm(N, point): ...
```

```
    partial(norm, 2)
```

- Generische Normierungsfunktion $\|\cdot\|_N$
- Die Distanz zum Ursprung ist die 2-Norm



➤ Beispiel: Eine Verarbeitungskette



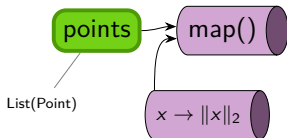
Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3,0.4), (-0.3, -0.2),  
          (0.6,-0.4), (1, 1)]
```

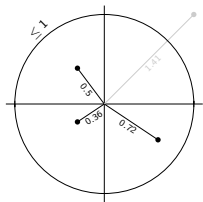
```
def norm(N, point): ...
```

```
map(partial(norm, 2),  
    points)
```

- Generische Normierungsfunktion $\| \cdot \|_N$
- Die Distanz zum Ursprung ist die 2-Norm
- Bilde jeden Punkt auf seine Distanz ab



➤ Beispiel: Eine Verarbeitungskette



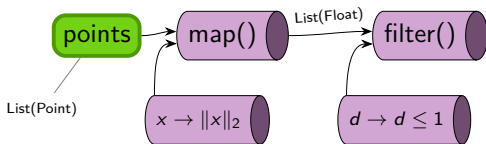
Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3, 0.4), (-0.3, -0.2),  
          (0.6, -0.4), (1, 1)]
```

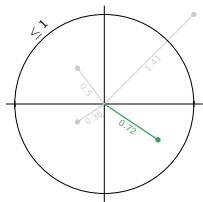
```
def norm(N, point): ...
```

```
filter(lambda d: d <= 1.0,  
       map(partial(norm, 2),  
          points))
```

- Generische Normierungsfunktion $\| \cdot \|_N$
- Die Distanz zum Ursprung ist die 2-Norm
- Bilde jeden Punkt auf seine Distanz ab
- Filtere Distanzen, die ≤ 1 sind



➤ Beispiel: Eine Verarbeitungskette

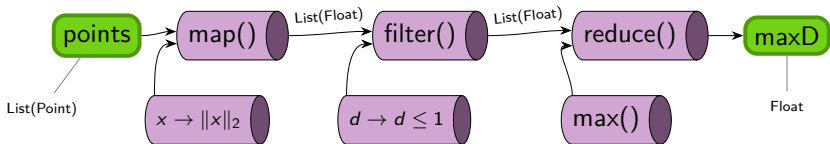


Aufgabe: Gegeben sei eine 2-dimensionale Punktwolke. Finden Sie innerhalb des Einheitskreises die größte Distanz, die ein Punkt zum Ursprung hat.

```
points = [(-0.3,0.4), (-0.3, -0.2),  
          (0.6,-0.4), (1, 1)]
```

```
def norm(N, point): ...  
maxD = \  
    reduce(max,  
        filter(lambda d: d <= 1.0,  
            map(partial(norm, 2),  
                points)))
```

- Generische Normierungsfunktion $\| \cdot \|_N$
- Die Distanz zum Ursprung ist die 2-Norm
- Bilde jeden Punkt auf seine Distanz ab
- Filtere Distanzen, die ≤ 1 sind
- Reduziere Distanzen zur maximalen Distanz



→ Keine Seiteneffekte → Keine Schokolade

- **Erinnerung:** Im rein-funktionalen Paradigma gibt es keine Seiteneffekte
 - Kein veränderlicher Zustand, keine Variablen, ausschließlich **pure Funktionen**
- ⇒ Jegliche Art der Ein- oder Ausgabe sind verbotene Seiteneffekte
- ⇒ Keine Terminalausgabe, kein Dateisystem, kein Netzwerk

Wie soll man damit nützliche Programme schreiben?

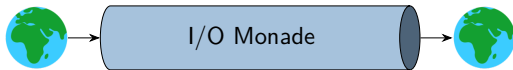
- **Functional-Style Programming** als funktionales Paradigma Light
 - Programmierstil, der funktionale Elemente in anderen Sprachen verwendet
 - Komposition, partielle Applikation, oder **map()** geht auch mit Seiteneffekten
 - Trennung des Programms in Funktionen mit und ohne Seiteneffekte
- ⇒ Ein effektiver Entwickler setzt Seiteneffekte gezielt ein
- Funktionale Elemente verbreiten sich unter den Sprachen
 - Skriptsprachen haben meist funktionale Aspekte (Python, Ruby, Lua,...)
 - Die Tendenz bei übersetzten Sprachen geht auch zu funktionalen Elementen.

➤ Und bei rein funktionalen Sprachen?

- Nur wenige Sprachen verbieten Seiteneffekte völlig
 - **Haskell** ist das prominenteste Beispiel
 - Es gibt noch weitere Forschungssprachen: Clean, Mercury

➤ Und bei rein funktionalen Sprachen?

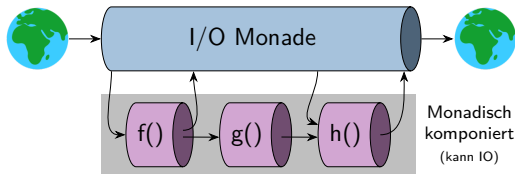
- Nur wenige Sprachen verbieten Seiteneffekte völlig
 - **Haskell** ist das prominenteste Beispiel
 - Es gibt noch weitere Forschungssprachen: Clean, Mercury
- In Haskell wird I/O über die **I/O-Monade** gesteuert



- **Intuition:** Die I/O-Monade transformiert das Universum
- Sie steuert die Ausführung monadisch komponierter Funktionen
- Ruft Funktionen mit Welt-Informationen auf und koppelt Ergebnisse zurück
- Unten in der Aufrufhierarchie sind nur pure Funktionen

➤ Und bei rein funktionalen Sprachen?

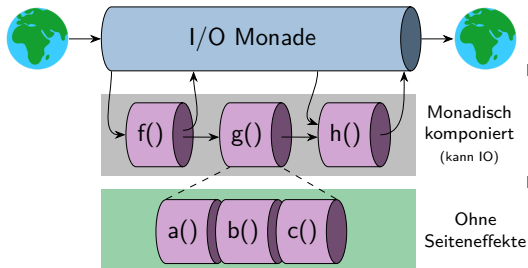
- Nur wenige Sprachen verbieten Seiteneffekte völlig
 - **Haskell** ist das prominenteste Beispiel
 - Es gibt noch weitere Forschungssprachen: Clean, Mercury
- In Haskell wird I/O über die **I/O-Monade** gesteuert



- **Intuition:** Die I/O-Monade transformiert das Universum
- Sie steuert die Ausführung monadisch komponierter Funktionen
- Ruft Funktionen mit Welt-Informationen auf und koppelt Ergebnisse zurück
- Unten in der Aufrufhierarchie sind nur pure Funktionen

➤ Und bei rein funktionalen Sprachen?

- Nur wenige Sprachen verbieten Seiteneffekte völlig
 - **Haskell** ist das prominenteste Beispiel
 - Es gibt noch weitere Forschungssprachen: Clean, Mercury
- In Haskell wird I/O über die **I/O-Monade** gesteuert



- **Intuition:** Die I/O-Monade transformiert das Universum
- Sie steuert die Ausführung monadisch komponierter Funktionen
- Ruft Funktionen mit Welt-Informationen auf und koppelt Ergebnisse zurück
- Unten in der Aufrufhierarchie sind nur pure Funktionen



```
printUpCase :: String -> IO ()
printUpCase l = putStrLn (upCase l)

main :: IO ()
main = (getLine >=> printUpCase)
      >> main
```

```
main = do
  l <- getLine
  putStrLn (upCase l)
  main
```

- Die I/O-Monade ist „magisch“ im Haskell Universum
 - Im Programm komponiert man nur Funktionen zusammen ($>>$, $>=>$)
 - Virtuelle Haskell-Maschine entnimmt das Monadenobjekt `main` und führt es aus.
 - **Syntaktischer Zucker** erleichtert die Monadenkomposition
- Monaden sind ein allgemeineres und mächtigeres Konzept
 - Monaden sind ein Konzept aus der **Kategorientheorie**
 - Monaden kapseln Strategien zur Auswertungsreihenfolge und zum Datenfluss
 - Es gibt viele spannende Monadentypen: Maybe, State, Writer
 - Wir können und werden keine davon hier besprechen...



- **Veränderlicher Zustand** ist problematisch in großen Softwareprojekten
 - Viele Prozeduren interagieren mit und verändern eine Vielzahl an Objekten
 - Zustand ist im Quellcode nicht sichtbar und daher unübersichtlich
- Das (rein) **funktionale Paradigma** verzichtet auf veränderlichen Zustand
 - Variablen können nur einmal an ein Objekt gebunden werden
 - Listen und Aufzählungstypen mit Nutzlast sind nützliche Datentypen
- Seiteneffektfreie Funktionen werden **Bürger erster Klasse**
 - Funktionsobjekte sind als Argumente und Rückgabewerte gleichberechtigt
 - Komposition, partielle Applikation und Listenfunktionen
- **Ein- und Ausgabe** kann nicht Seiteneffektfrei sein
 - Der funktionale Programmierstil erlaubt veränderlichen Zustand
 - Rein funktionale Sprachen transformieren die Welt über **Monaden**