

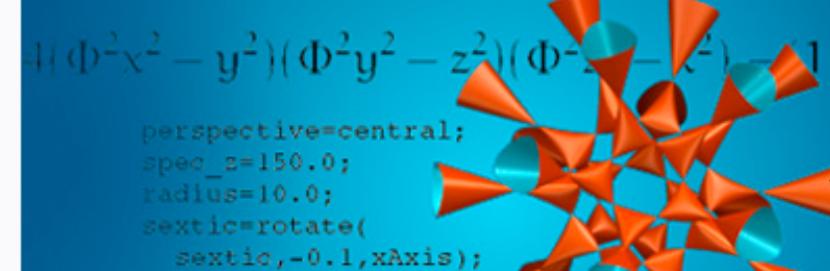
Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch

(Programmiersprachen und –werkzeuge)

Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[Skript 11]

Abstraktion

- Vermeidung von Redundanz (Don't Repeat Yourself)
- Bessere Lesbarkeit
- Bekannte Abstraktionsmechanismen
 - Konstanten
 - Funktionale Abstraktion

Funktionale Abstraktion

```

; (list-of String) -> Boolean
; does I contain "dog"
(define (contains-dog? I)
  (cond
    [(empty? I) false]
    [else
      (or
        (string=? (first I) "dog")
        (contains-dog?
          (rest I))))]))
```

```

; (list-of String) -> Boolean
; does I contain "cat"
(define (contains-cat? I)
  (cond
    [(empty? I) false]
    [else
      (or
        (string=? (first I) "cat")
        (contains-cat?
          (rest I))))]))
```

Einige Unterschiede

Funktionale Abstraktion

; String (list-of String) -> Boolean
; to determine whether l contains the string s

```
(define (contains? s l)
  (cond
    [(empty? l) false]
    [else ( or (string=? (first l) s)
                (contains? s (rest l)))])))
```

Unterschied als Parameter
für eine Funktion.
Wiederverwendung der
Gemeinsamkeit.

; (list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
 (contains? "dog" l))

; (list-of String) -> Boolean
; does l contain "cat"
(define (contains-cat? l)
 (contains? "cat" l))

Funktionale Abstraktion

; (list-of Number) -> Number

; adds all numbers in l

(define (add-numbers l)

(cond

[(empty? l) 0]

[else

(+ (first l)

(add-numbers (rest l))))]))

; (list-of Number) -> Number

; multiplies all numbers in l

(define (mult-numbers l)

(cond

[(empty? l) 1]

[else

(* (first l)

(mult-num-

Unterschiedliche Werte können als
Parameter gekapselt werden.

Was ist mit unterschiedlichen
Funktionsaufrufen?

Lösungsidee

```
(define (op-numbers op z l)
```

```
(cond
```

```
[(empty? l) z]
```

```
[else
```

```
  (op (first l))
```

```
    (op-numbers op z (rest l))))]))
```

```
(define (add-numbers l) (op-numbers + 0 l))
```

```
(define (mult-numbers l) (op-numbers * 1 l))
```

```
> (add-numbers (list 1 2 3 4))
```

Codieren der aufzurufenden Funktion als Parameter. Geht das?

function call: expected a function after the open parenthesis, but found a variable

Limitierung von BSL
(Beginning Student Language)

Funktionale Abstraktion

- Lösungsansatz
 - Codierung von Funktionsnamen als Parameter in Funktionsheader
 - Übergeben von Funktionsnamen als Argument bei Funktionsaufruf
 - Verwendung von Parameter anstatt Funktionsnamen in Funktions-Body
- Warum schlägt der Lösungsansatz fehl?
 - Bisher: Funktionen sind keine Werte
- Lösung: Funktionen wie Werte behandeln!
 - Umschalten des Sprachlevels auf "Zwischenstufe mit Lambda"

Funktionen als Parameter

```
(define (op-numbers op z l)
```

```
(cond
```

```
[(empty? l) z]
```

```
[else
```

```
  (op (first l)
```

```
    (op-numbers op z (rest l))))]))
```

```
(define (add-numbers l) (op-numbers + 0 l))
```

```
(define (mult-numbers l) (op-numbers * 1 l))
```

```
> (add-numbers (list 1 2 3 4))
```

```
10
```

Wir haben sogar die
Abhängigkeit von
Zahlen gänzlich
abstrahiert.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

Diagram illustrating the execution of the function:

- The call `(op-elements + 0 (list 5 8 12))` is shown.
- A callout box contains the arguments:
 - `op: +` (highlighted in blue)
 - `z: 0` (highlighted in green)
 - `l: (list 5 8 12)` (highlighted in brown)
- An arrow points from the `op` argument to the `(op (first l))` part of the function body.
- Another arrow points from the `l` argument to the `(op-elements op z (rest l))` part of the function body.
- A third arrow points from the `op` argument to the `(+ 5 (op-elements + 0 (list 8 12)))` part of the recursive call.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

Diagram illustrating the execution of the function:

- The function `(op-elements op z l)` is called with `op: +`, `z: 0`, and `l: (list 8 12)`.
- The first branch of the `cond` is taken because `l` is not empty.
- The expression `(op (first l))` is evaluated, which is `(+ 8 0)`.
- The expression `(op-elements op z (rest l))` is evaluated, which is `(op-elements + 0 (list 12))`.
- The final result is `(+ 8 (op-elements + 0 (list 12)))`.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 12)

(+ 12 (op-elements + 0 (list)))

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

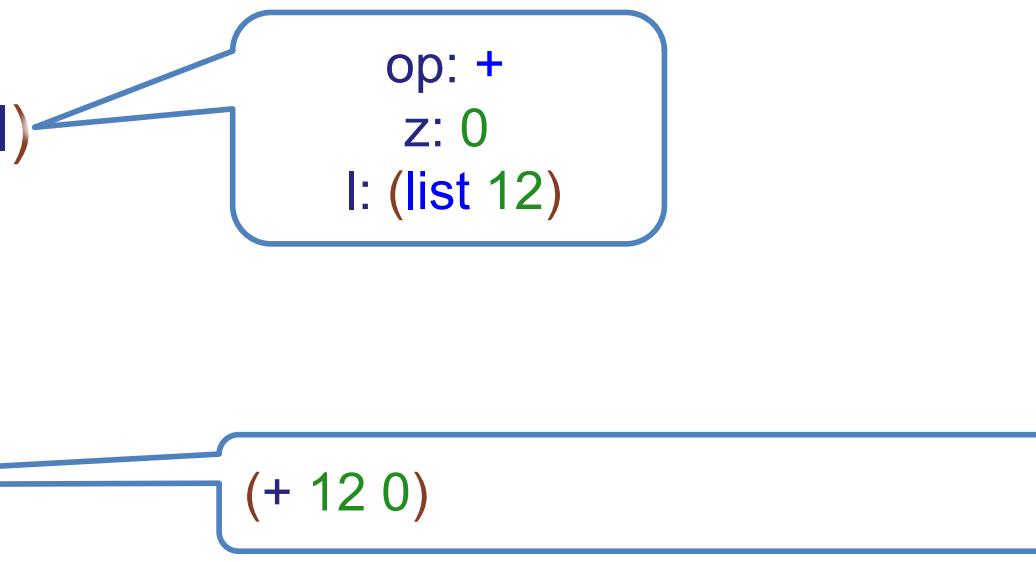
```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l)))]))
> (op-elements + 0 (list 5 8 12))
```

The diagram illustrates the evaluation of the function call `(op-elements + 0 (list 5 8 12))`. A box on the right contains the parameter bindings: `op: +`, `z: 0`, and `l: (list)`. Two arrows point from the variable `l` in the first argument of `op-elements` to the box, and one arrow points from the variable `z` in the second argument to the value `0`.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l))))]))
> (op-elements + 0 (list 5 8 12))
```



The diagram illustrates the execution of the function `(op-elements + 0 (list 5 8 12))`. It shows two callout boxes. The top box contains the parameter bindings: `op: +`, `z: 0`, and `l: (list 12)`. The bottom box contains the result of the first recursive step: `(+ 12 0)`.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l))))]))
> (op-elements + 0 (list 5 8 12))
```

The diagram illustrates the evaluation of the function call `(op-elements + 0 (list 5 8 12))`. It shows the parameter binding for `op-elements` and the arguments `+`, `0`, and `(list 5 8 12)`.

- A blue rounded rectangle contains the parameter binding:
 - `op: +`
 - `z: 0`
 - `l: (list 8 12)`
- An arrow points from the variable `l` in the first argument of `op-elements` to this binding.
- Another blue rounded rectangle contains the value of the second argument:
 - `(+ 8 12)`
- An arrow points from the variable `op` in the second argument of `op-elements` to this value.

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l))))]))
> (op-elements + 0 (list 5 8 12))
```

op: +
z: 0
l: (list 5 8 12)

(+ 5 20)

Mächtigkeit von funktionaler Abstraktion

- Allgemein: Funktion, die Elemente einer Liste und einen angegebenen Wert inkrementell verknüpft

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
      (op (first l)
          (op-elements op z (rest l)))])))
> (op-elements + 0 (list 5 8 12))
```

25

Mächtigkeit von funktionaler Abstraktion

```
> (op-elements string-append "" (list "ab" "cd" "ef"))  
"abcdef"
```

```
> (op-elements beside empty-image (list  
  (circle 10 "solid" "red")  
  (rectangle 10 10 "solid" "blue")  
  (circle 10 "solid" "green")))
```



Kopieren einer Liste per se nicht interessant. Varianten schon: Erzeugung von Listen basierend auf Listen.

```
> (op-elements cons empty (list 5 8 12 2 9))  
(list 5 8 12 2 9)
```

Mächtigkeit von funktionaler Abstraktion

```
(define (append-list l1 l2)
```

Zusammenfügen
von Listen

```
  (op-elements cons l2 l1))
```

```
> (append-list (list 1 2) (list 3 4))
```

```
(list 1 2 3 4)
```

Ergebnis bei
Rekursionsabbruch.

Ergebnis bei rekursivem Aufruf:
Liste bestehend aus erstem Element
und Ergebnis des rekursiven Aufrufs.

Mächtigkeit von funktionaler Abstraktion

> (op-elements

append-list

empty

(list (list 1 2) (list 3 4) (list 5 6)))

Flachklopfen einer
Liste von Listen.

(list 1 2 3 4 5 6)

Ergebnis bei
Rekursionsabbruch.

Ergebnis bei rekursivem Aufruf:
Zusammenfügung der Liste im ersten
Element und der Liste im Ergebnis
des rekursiven Aufrufs.

Sortieren mit funktionaler Abstraktion

; A (sorted-list-of Number) is a (list-of Number) which is sorted by "<"

; (list-of Number) -> (sorted-list-of Number)

; returns a list containing all elements of I sorted by "<"

```
(define (sort-list I) (op-elements insert empty I))
```

; Number (sorted-list-of Number) -> (sorted-list-of Number)

; inserts x into a sorted list xs

```
(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))
```

```
(define (insert x xs) ...)
```

Sortieren mit funktionaler Abstraktion

; Number (sorted-list-of Number) -> (sorted-list-of Number)

; inserts x into a sorted list xs

(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))

(define (insert x xs)

(cond [(empty? xs) (list x)]

Basisfall
(Rekursionsabbruch)

[(cons? xs) (if (< x (first xs))

(cons x xs)

(cons (first xs) (insert x (rest xs))))]))

Rekursiver Fall

Rekursiver
Funktionsaufruf

Wiederverwendung durch Abstraktion

- Vermeidet Redundanz
- Fördert Lesbarkeit: Komplexität wird verborgen
- Spart Arbeit
- Wiederverwendung von Eigenschaften wie Korrektheit
 - Wiederverwendeter Code wurde bereits getestet
 - Mehr Clients bedeutet mehr Tests

Typen von Funktionen

- Wir haben gesehen:
Funktionen können Werte unterschiedlicher Typen verarbeiten
- Wie können wir das in der Signatur angeben?

```
; ...  
; (list-of String) -> String  
; (list-of Number) -> Number  
(define (second l) (first (rest l)))
```

Wir könnten alle möglichen Signaturen aufzählen.

Ergebnis: zweites Element der Liste unabhängig vom Elementtypen

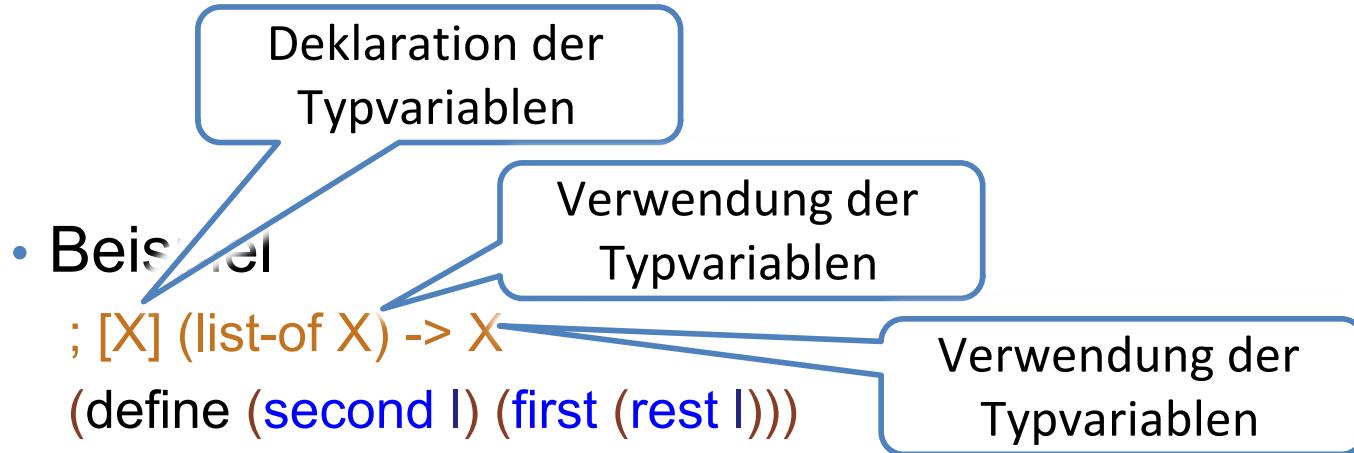
Typen von Funktionen

- Aufzählen der Signaturen ist keine gute Idee
 - Unmöglich alle aufzuschreiben:
 - Datentypen beliebig definierbar
→ unendlich viele Signaturen
 - Unpraktisch die verwendeten aufzuschreiben
 - Bei neuer Verwendung muss bestehender Code angepasst werden
- Bessere Möglichkeit: Typvariablen



Typvariablen

- Signatur mit Typvariablen steht für Menge der Signaturen, die sich aus allen gültigen Ersetzungen ergibt
 - Ersetzung durch konkreten Typ
 - Derselbe Typ bei allen Vorkommen der Typvariablen



Typvariablen

- Alle möglichen Ersetzungen müssen eine gültige Signatur ergeben
- Ist diese Signatur mit Typvariablen korrekt?

```
; [X Y] (list-of X) -> Y  
(define (second l) (first (rest l)))
```

Nein. Eine mögliche aber ungültige Ersetzung ist:
 $(\text{list-of Number}) \rightarrow \text{String}$

Typ von Funktionen

- Funktionen können als Argument bei Funktionsaufruf übergeben werden
 - Wie ist der Typ des entsprechenden Parameters?
 - Wie schreiben wir die Signatur von Funktionen mit Funktionsparametern?

```
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

Wie lautet die Signatur von add-circle?

```
> (add-circle beside (rectangle 10 10 "solid" "blue"))
```



```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```



Typ von Funktionen

- Wir kennen bereits eine Beschreibung für Funktionen basierend auf Typen: die Signatur
- Wir können die Funktionssignatur als deren Typ ansehen

```
; (Image Image -> Image) Image -> Image
```

```
(define (add-circle f img)
```

```
  (f img (circle 10 "solid" "red")))
```

Signatur von beside:

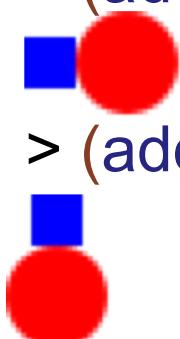
(Image Image -> Image)

```
> (add-circle beside (rectangle 10
```

Signatur von above:

(Image Image -> Image)

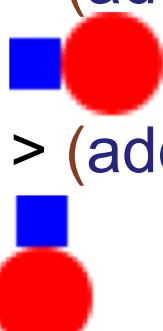
```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```



Typ von Funktionen

- Wir kennen bereits eine Beschreibung für Funktionen basierend auf Typen: die Signatur
- Wir können die Funktionssignatur als deren Typ ansehen

```
; (Image Image -> Image) Image -> Image
(define (add-circle f img)
  (f img (circle 10 "red")))
> (add-circle beside (rectangle 10 10 "solid" "blue"))
> (add-circle above (rectangle 10 10 "solid" "blue"))
```



Das erste Argument muss eine Funktion mit der Signatur `(Image Image -> Image)` sein.

Funktionen als Rückgabewert

- Wir können Funktionen als Argument übergeben
- Eine Funktion kann auch eine Funktion zurückgeben

; Color -> Image

```
(define (big-circle color) (circle 20 "solid" color))
```

; Color -> Image

```
(define (small-circle color) (circle 10 "solid" color))
```

; String -> (Color -> Image)

```
(define (get-circle-maker size)
```

```
  (cond [(string=? size "big") big-circle]
```

```
        [(string=? size "small") small-circle]))
```

Funktionssignatur als Typ
des Rückgabewerts

Funktion als Rückgabewert

Funktionen als Rückgabewert

- Aufruf einer Funktion mit Funktion als Rückgabewert
 - Aufruf steht an erster Stelle eines Funktionsaufruf-Ausdrucks

- Beispiel:

```
; String -> (Color -> Image)
```

```
(define (get-circle-maker size) ... )
```

Ausdruck ergibt eine Funktion

Aufruf der Ergebnisfunktion und Übergabe eines Arguments

```
> ((get-circle-maker "big") "cyan")
```

Klammern beachten!



Verschachtelte Funktionstypen

- Bei Funktionstyp für Parameter oder Rückgabewert
 - Parametertypen und Rückgabetypen dieser Funktion können wiederum Funktionen sein
- Verschachtelung kann beliebig tief sein

; ((Image Image -> Image) Image -> Image) Image Image -> Image

(define (add-two-imgs-with f img1 img2)

 (above (f beside img1) (f beside img2)))

> (add-two-imgs-with

 add-circle

 (circle 10 "solid" "green")

 (circle 10 "solid" "blue"))

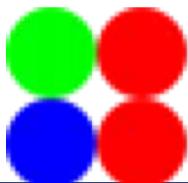
Verwendung des Parameters f.

1. Argument: Funktion

(Image Image -> Image)

2. Argument: Bild

Rückgabetyp: Bild



Ordnung von Funktionen

- Ordnung von Funktionen gibt an wie stark die Signatur verschachtelt ist
 - “Funktion erster Ordnung“ (“First-Order Function”)
 - Parameter- und Resultattypen sind keine Funktionen
 - Nur ein einziger Pfeil in Signatur
 - Funktion zweiter Ordnung
 - Mindestens ein Parameter- oder Resultattyp ist die Signatur einer Funktion erster Ordnung
 - Und so weiter
- In der Regel wird nur unterschieden zwischen
 - Funktionen erster Ordnung (First-Order Functions)
 - Funktionen höherer Ordnung (Higher-Order Functions)

Ordnung von Funktionen

- Ordnung von Funktionen gibt an wie stark die Signatur verschachtelt ist
 - “Funktion erster Ordnung“ (“First-Order Function”)
 - Parameter- und Resultattypen sind keine Funktionen
 - Nur ein einziger Typ
 - Funktion zweiter Ordnung
 - Mindestens eine Funktion als Parameter einer Funktion einer Funktion erster Ordnung
 - Und so weiter
 - In der Regel
 - Funktionen erster Ordnung (First-Order Functions)
 - Funktionen höherer Ordnung (Higher-Order Functions)

Wesentliches Unterscheidungsmerkmal
für Programmiersprachen:

- Unterstützung nur für Funktionen erster Ordnung
- Unterstützung auch für Funktionen höherer Ordnung

Polymorphe Funktionen

- Funktionen, deren Resultattyp von den übergebenen Argumenten abhängt heißen “polymorphe Funktionen”
 - Beispiel:

```
; [X] (list-of X) -> X
(define (second l) (first (rest l)))
```
- Auch Funktionen höherer Ordnung können Polymorph sein
 - Beispiel:
 - ```
(define (op-elements op z l) ...)
```



Wie ist hier die  
Signatur?

# Polymorphe Funktionen

```
; [A B C D] A B C -> D
(define (op-elements op z l)
(cond
 [(empty? l) z]
 [else
 (op (first l)
 (op-elements op z (rest l))))]))
```

Allgemein: eine Typvariable pro Parameter / Rückgabetyp

Jetzt:  
Abhängigkeiten erkennen.

# Polymorphe Funktionen

; [A B C D] A B C -> B

(define (op-elements op z l)

(cond

[(empty? l) z]

[else

(op (first l)

(op-elements op z (rest l))))]))

z kann Rückgabewert  
sein. Also muss der  
Rückgabetyp dem Typen  
von z entsprechen.

# Polymorphe Funktionen

; [A B C D E] A B (list-of E) -> B

(define (op-elements op z l)

(cond

[(empty? l) z]

[else

(op (first l))

(op-elements op z (rest l))))]))

I wird an empty?, first, rest übergeben. Es muss daher eine Liste sein.

# Polymorphe Funktionen

```
; [A B C D E] (E B -> B) B (list-of E) -> B
```

```
(define (op-elements op z l)
```

```
(cond
```

```
[(empty? l) z]
```

```
[else
```

```
 (op (first l))
```

```
 (op-elements op z (rest l))))]))
```

op wird aufgerufen mit dem ersten Element von l und dem Resultat des rekursiven Aufrufs. Und sein Ergebnis kann Ergebnis von op-elements sein.

# Polymorphe Funktionen

```
; [X Y] (X Y -> Y) Y (list-of X) -> Y
```

```
(define (op-elements op z l)
```

```
(cond
```

```
[(empty? l) z]
```

```
[else
```

```
 (op (first l))
```

```
 (op-elements op z (rest l))))]))
```

Noch ein bisschen  
aufräumen

# Signatur von polymorphen Funktionen

- Vorsicht beim Ableiten der Signatur aus Beispielen
  - (op-elements + 0 (list 5 8 12))
  - (op-elements string-append "" (list "ab" "cd" "ef"))
- In diesen Beispielen scheint die Signatur zu sein:  
; [X] (X X -> X) X (list-of X) -> X
  - Diese Signatur ist gültig, aber zu eingeschränkt
  - Ein anderes gültiges Beispiel ist aber:
    - (op-elements cons empty (list 5 8 12 2 9))
    - Signatur:  
; (Number (list-of Number) -> (list-of Number)) (list-of Number)  
(list-of Number) -> (list-of Number)
    - Gültig und möglichst allgemein ist die zuvor ermittelte Signatur:  
; [X Y] (X Y -> Y) Y (list-of X) -> Y

# Typvariablen bei Datendefinitionen

- Bei generischen Funktionen oder Datendefinitionen
    - Bei bestimmten Elementen sind mehrere Typen unterstützt
    - Alle Varianten hierfür aufzuzählen ...
    - ... einsetzen einer Typvariablen
  - Typvariablen bei Signaturen
    - Deklariert durch vorangestellte eckige Klammern
  - Typvariablen bei Datendefinitionen
    - Deklariert durch Verwendung im Namen der Datendefinition
- ; a (list-of X) is either  
; - empty  
; - (cons X (list-of X))

Macht deutlich, dass X  
eine Typvariable ist.

# Typvariablen bei Datendefinitionen

- Durch Typvariablen können Eigenschaften ausgedrückt werden  
; a (nonempty-list-of X) is: (cons X (list-of X))

Sowohl erstes Element  
als auch Elemente der  
Rest-Liste haben  
denselben Typ.

# Grammatik der Typen und Signaturen

- **<Typ>** ::= **<Basistyp>**
  - | **<Datentyp>**
  - | ‘(’ **<TypKonstruktor>** **<Typ>+ ‘)**
  - | ‘(’ **<Typ>+ ‘->’ <Typ> ‘)’**
  - | **<X>**
  - | ‘[’ **<X>+ ’]** **<Typ>**
- **<Basistyp>** ::= ‘Number’
  - | ‘String’
  - | ‘Boolean’
  - | ‘Image’
- **<Datentyp>** ::= ‘Posn’
  - | ‘WorldState’
  - | ...
- **<TypKonstruktor>** ::= ‘list-of’
  - | ‘tree-of’
  - | ...
- **<X>** ::= ‘X’
  - | ‘Y’
  - | ...

Typen werden rekursiv konstruiert.

Signaturen sind Typen

Typvariablen sind Typen.  
Aber Vorsicht: ihre Verwendung ergibt nur Sinn, wenn Sie vorher deklariert sind.

# Funktionstypen

- Wir kennen jetzt die Syntax für die Deklaration von Typen inklusive Funktionstypen
- Bedeutung eines Typs:
  - Menge aller Werte mit gemeinsamen Eigenschaften
  - ... über die gemeinsam Funktionen definiert sind
- Was ist die Bedeutung eines Funktionstyps?
  - Informell: Menge aller Funktionen mit entsprechender Signatur

# Lokale Definitionen

- Bisher:
  - Definitionen sind im gesamten Programm sichtbar
  - Ab der Definition
- Manche Definitionen werden nur in einem kleinen Programmteil verwendet
  - Solche Definitionen füllen unnötig die globale Umgebung
  - Erschweren Programmverständnis
  - Verhindern andere Definition unter demselben Namen an anderer Stelle

# Lokale Definitionen

- Beispiel

; (list-of String) -> String

; appends all strings in l with blank space between elements

(check-expect (append-all-strings-with-space (list "a" "b" "c"))  
                  "a b c ")

(define (append-all-strings-with-space l)

  (foldr string-append-with-space

    "

  l))

; String String -> String

; juxtapoint two strings and prefix with space

(define (string-append-with-space s t)

  (string-append s " " t))

# Lokale Definitionen

- Beispiel

; (list-of String) -> String

; appends all strings in I with blank space between elements

(check-expect (append-all-strings-with-space (list "a" "b" "c"))  
                  "a b c ")

(define (append-all-strings-with-space I)

  (foldr string-append-with-space

    foldr ist eine primitive  
    Funktion, die unserem  
    op-elements entspricht.

Die Funktion  
string-append-with-space  
wird nur hier verwendet.

; String

; juxtapoint two strings and prefix with space

(define (string-append-with-space s t)

  (string-append s " " t))

# Lokale Definitionen

- Beispiel

```
; (list-of String) -> String
```

```
; appends all strings in l with blank space between elements
```

```
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
 "a b c ")
```

```
(define (append-all-strings-with-space l)
```

```
 (local
```

```
 [; String String -> String
```

```
 ; juxtapoint two strings and prefix with space
```

```
 (define (string-append-with-space s t)
```

```
 (string-append s " " t))]
```

```
 (foldr string-append-with-space
```

```
 ""
```

```
 l))
```

```
)
```

# Lokale Definitionen

- Beispiel

```
; (list-of String) -> String
```

; appends all strings in l with blank space between elements

```
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
 "a b c ")
```

```
(define (append-all-strings-with-space l)
```

(local

[ ; String String -> String

: juxtapoint two strings and return their concatenation

```
define (string-append-with-space s t))
```

dr string-append-with-space

" "

l))

Schlüsselwort  
für lokale  
Definitionen

lokale  
Definitionen

Gültigkeits-  
bereich  
lokaler  
Definitionen

Hier kann **string-append-with-space** nicht  
mehr aufgerufen werden.

# Lokale Definitionen

- (local ...) sind Ausdrücke
  - Kann überall verwendet werden wo Ausdruck erwartet wird
  - Erster Unterausdruck enthält eine oder mehrere Definitionen
  - Zweiter Unterausdruck
    - Kann lokale Definitionen verwenden
    - Resultat ist das Ergebnis des local-Ausdrucks

- Beispiele

```
> (local [(define (f x) (+ x 1))] (+ (* (f 5) 6) 7))
```

43

```
> (+ (local [(define (f x) (+ x 1))] (* (f 5) 6)) 7)
```

43

```
> (+ (* ((local [(define (f x) (+ x 1))] f) 5) 6) 7)
```

43



# Lokale Definitionen

- (local ...) sind Ausdrücke
  - Kann überall verwendet werden wo Ausdrücke erwartet werden
  - Erster Unterausdruck enthält eine oder mehrere Definitionen
  - Zweiter Unterausdruck
    - Kann lokale Definitionen verwenden
    - Resultat ist das Ergebnis des local-Ausdrucks

- Beispiele

```
> (local [(define (f x) (+ x 1)) 1 (+ (* (f 5) 6) 7))
```

43

Wert des local-Ausdrucks ist die  
lokal definierte Funktion selbst.

```
> (+ (local [(define (
```

43

```
> (+ (* ((local [(define (f x) (+ x 1)))] f) 5) 6) 7)
```

43

# Lokale Definitionen

- Auch Konstanten können lokal definiert werden
- Lokale Definitionen können auf lokale Umgebung zugreifen
  - Zum Beispiel auf Argument-Werte

# Lokale Definitionen - Kontext

- Beispiel:

```
(define (power8 x)
 (* x x)))))))))
> (power8 2)
```

256

- Hier werden acht Multiplikationen ausgeführt
- Umschreiben durch bekannte Rechenregel:  $a^{2^b} = a^b * a^b$

Wenn wir  $a^b$  nicht mehrfach berechnen müssen, benötigen wir weniger Multiplikationen.

# Lokale Definitionen - Kontext

```
(define (power8-fast x)
```

```
(local
```

```
[(define r1 (* x x))
 (define r2 (* r1 r1))
 (define r3 (* r2 r2))]
```

```
r3))
```

$$r1 = x * x = x^2$$

$$r2 = r1^2 = x^2 * x^2 = x^4$$

$$r3 = r2^2 = x^4 * x^4 = x^8$$

Insgesamt 3 statt 7  
Multiplikationen.

# Lokale Definitionen - Kontext

```
(define (power8-fast x)
 (local
 [(define r1 (* x x))
 (define r2 (* r1 r1))
 (define r3 (* r2 r2))])
 r3))
```

Definition einer lokalen Konstanten.

Neu: bei der Definition einer lokalen Konstanten dürfen wir die Parameter der umgebenden Definition verwenden.

Gilt für alle lokalen Definitionen.

Wir wissen bereits: bei der Definition einer Konstanten dürfen wir bereits definierte Funktionen und Konstanten verwenden.

# Lokale Konstanten

- Konstanten werden nur einmal bei der Definition berechnet
- Zwischenergebnisse können so gespeichert werden
- Globale Konstanten helfen in dem Beispiel nicht: sie können nicht von Funktionsparametern abhängen
- Abstraktion durch lokale Konstanten
  - Vermeidung von Redundanz
    - Im Programmtext
    - In der Berechnung
  - Vergeben eines Namens an ein Zwischenergebnis

Statische Redundanz.  
(Don't Repeat Yourself)

Dynamische Redundanz.

# Namen für Zwischenergebnisse

```
(define (posn+vel p q)
 (make-posn (+ (posn-x p) (vel-delta-x q))
 (+ (posn-y p) (vel-delta-y q))))
```

```
(define (posn+vel p q)
 (local [(define new-x (+ (posn-x p) (vel-delta-x q)))
 (define new-y (+ (posn-y p) (vel-delta-y q)))]
 (make-posn new-x new-y)))
```

Bedeutung der Werte  
wird deutlich. Ausdruck  
weniger verschachtelt.

# Vermeidung von Dynamischer Redundanz

- Successive Squaring
  - Algorithmus zur Berechnung von Potenzen
  - Verallgemeinerung des Ansatzes von power8-fast
- Erster Anlauf
  - Exponent ist eine natürliche Zahl
  - Auffassen von natürlichen Zahlen als rekursiven Datentyp
  - Implementierung als rekursive Funktion

```
; NaturalNumber Number -> Number
(define (exponential n x)
 (if (zero? n)
 1
 (* x (exponential (sub1 n) x))))
```

# Vermeidung von Dynamischer Redundanz

; NaturalNumber Number -> Number

(define (exponential n x)

  (if (zero? n)

    1

    (\* x (exponential (sub1 n) x))))

- Expansion des Funktionsaufrufs z.B. für  $n = 8$

• (exponential 8 x)

$$\equiv (* x x)))))))$$

$$\equiv (* (* (* x x) (* x x)) (* (* x x) (* x x)))$$

Assoziativität

(\* x x) wird mehrfach berechnet:  
dynamische Redundanz.

# Vermeidung von Dynamischer Redundanz

- Wie kann die bekannte Lösung verallgemeinert werden?
- Bisherige Annahme: Exponent ist durch zwei teilbar (gerade)
- Verallgemeinerung: benötigen Fallunterscheidung und Strategie für ungerade Exponenten

```
(define (exponential-fast n x)
```

```
 (if (zero? n)
```

```
 1
```

```
 (local
```

n gerade:  $y = x^{n/2}$   
 n ungerade:  $y = x^{(n-1)/2}$

```
 [(define y (exponential-fast (quotient n 2) x))
```

```
 (define z (* y y))]
```

```
 (if (odd? n) (* x z) z))))
```

quotient:

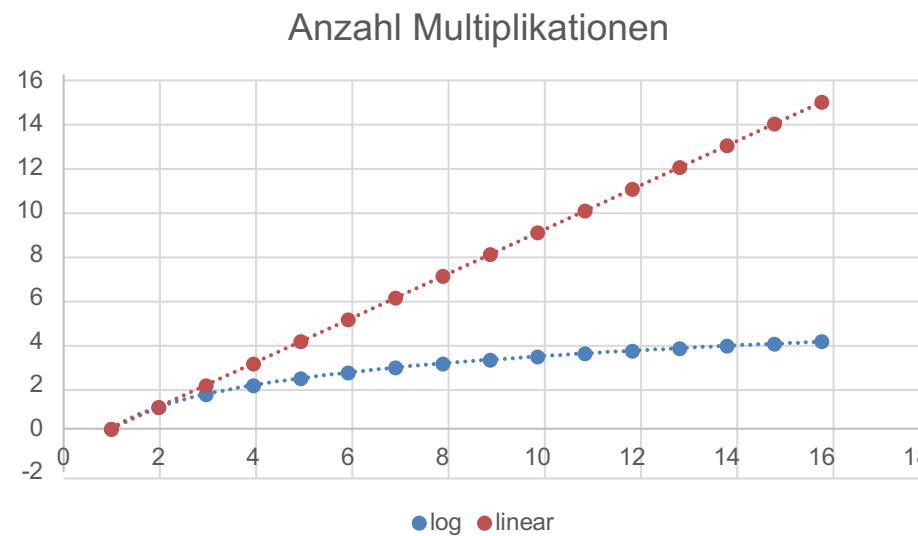
ganzzahlige Division

n gerade:  $z = x^{n/2} * x^{n/2} = x^n$

n ungerade:  $z = x^{(n-1)/2} * x^{(n-1)/2} = x^{n-1}$

# Vermeidung von Dynamischer Redundanz

- Successive Squaring benötigt ca.  $\log_2(n)$  Multiplikationen gegenüber  $(n - 1)$  Multiplikationen der naiven Implementierung
  - Vermeidung von dynamischer Redundanz
  - Schnellere Ausführung



- Demo Laufzeitperformance

# Scope lokaler Definitionen

- Bereich im Programm, wo eine Definition verwendet werden darf: “Gültigkeitsbereich“ oder “Scope“
- In unserem Fall: ”lexikalisches Scoping“ bzw. ”statisches Scoping“
  - Scope hängt ab von dem Platz der Definition im Quelltext
  - In Teilausdrücken des ”local“ Ausdrucks

# Scope lokaler Definitionen

```
(local [(define (f n) (if (zero? n)
 0
 (+ x (f (sub1 n))))))
 (define x 5)]
(+ x (f 2)))
```

(+  
  (local [(define x 5)] (+ x 3))  
  x)

Verwendung im Scope.

Verwendung im Scope.

Verwendung außerhalb des Scopes.

# Geschachtelte Scopes

- Der Scope einer lokalen Definition ist eingebettet in den Scope der globalen Definitionen
- (local ...) sind Ausdrücke
  - Dürfen überall stehen wo Ausdrücke erlaubt sind
  - Können auch in Unterausdrücken von (local ...) vorkommen
- Scopes sind geschachtelt!
  - Was passiert, wenn Namen wiederholt werden?

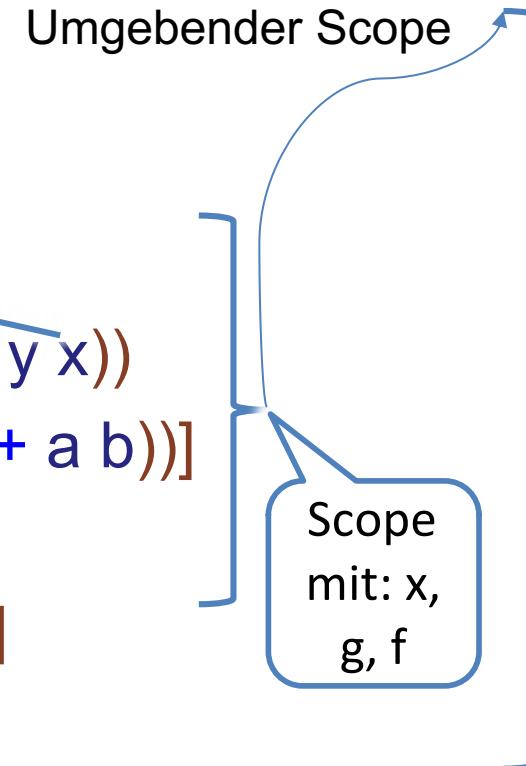
# Geschachtelte Scopes

(add1 local

```

[(define (f y)
 (local [(define x 2)
 (define (g y) (+ y x))
 (define (f a b) (+ a b))])
 (f 1 (g y))))
(define (add1 x) (sub1 x))]
(f (add1 2)))

```



- Namen des umgebenden Scope dürfen wiederverwendet (überschrieben) werden
  - Suche nach Definition von innen nach außen

# Funktionen als Werte: Closures

- Bisher: lokal definierte Konstanten können auf Funktionsparameter zugreifen
  - Geht das auch bei lokal definierten Funktionen?
  - Beispiel
    - Eine Funktion derivative soll die Ableitung der als Argument übergebenen Funktion zurückliefern
    - Die Ergebnis-Funktion soll bspw. mittels einer plot-Funktion ausgegeben werden
- ```
; (Number -> Number) -> Image
( define (plot-function f) ...)
; (Number -> Number) -> (Number -> Number)
( define (derivative f) ...)
```

Funktionen als Werte: Closures

- Ein Versuch:

; (Number -> Number) -> (Number -> Number)

(define (derivative f)

(local

[(define delta-x 0.001)

(define (delta-f-x x) (- (f (+ x delta-x)) (f x)))

(define (g x) (/ (delta-f-x x) delta-x))]

g))

Dürfen wir lokal definierte
Funktionen zurückgeben?
Was könnten Probleme sein?

Funktionen als Werte: Closures

- Ein Versuch:

`; (Number -> Number) -> (Number -> Number)`

`(define (derivative f)`

`(local`

`[(define delta-x 0.001)`

`(define (delta-f-x x) (- (f (+ x delta-x)) (f x)))`

`(define (g x) (/ (delta-f-x x) delta-x))`

`g))`

Die lokal definierte Funktion greift auf Argumente von `derivative` zu. Wie kann die Funktion außerhalb des Funktionsaufrufs ausgewertet werden, der sie erzeugt hat?

Funktionen als Werte: Closures

- Das vorige Programm ist korrekt!
- Aber:
 - Es wird nicht eine Funktion zurückgegeben
 - Sondern eine “Closure”
- Closure
 - Kombination aus Funktionsdefinition und
 - Einer lokalen Umgebung
- Eine Closure ist ein sog. “Funktionsabschluß“

Funktionen als Werte: Closures

- Verwendung einer Closure wie eine Funktion
- Aufruf der Closure mit Argumenten möglich
- Beim Auswerten der Closure werden die Definitionen aus der gebundenen lokalen Umgebung verwendet

```
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
    g))
(local [(define f (derivative exp))
        (define delta-x 10000)]
       (f 5))
```

Wird **nicht** bei der Auswertung von f verwendet.

Lambdas

- Lokal definierte Funktionen werden oft nur einmal verwendet
- Lohnt sich der Aufwand?
 - Vergeben eines Namens
 - Aufwendige Syntax

```
(local [(define (double x) (* 2 x))]  
       (map double lon))
```

Lambdas

- “Anonyme“ Funktionen: “lambdas“
- Werden direkt an der Stelle definiert, wo sie verwendet werden
- `(map (lambda (x) (* 2 x)) lon)`

Lambdas

- Allgemeine Form:
 - $(\lambda(x_1 \dots x_n) \text{exp})$
 - Entspricht: $(\text{local } [(\text{define } (f x_1 \dots x_n) \text{exp})] f)$
- Abkürzend: Verwendung der griechischen Buchstaben λ
 $(\text{map } (\lambda(x) (* 2 x)) \text{lon})$
- Lambdas und lokale Funktionen sind nicht äquivalent:
 - Lambdas haben keinen Namen
 - Sie können sich daher auch nicht rekursiv aufrufen

Sprachvereinfachung mit Lambdas

- Allerdings sind Lambdas zusammen mit Konstantendefinitionen ein Ersatz für Funktionsdefinitionen
 - $(\text{define } (\text{f } \text{x-1} \dots \text{x-n}) \text{ exp})$  $(\text{define f } (\lambda \text{x-1} \dots \text{x-n}) \text{ exp})$
- Bisher
 - Funktionsaufrufe beginnen entweder mit einem Funktionsnamen
 - Oder haben die Form $(\text{exp}_0 \text{ exp}_1 \dots \text{exp}_n)$
 - Wobei exp_0 zu einer Funktion ausgewertet wird
 - Genauer gesagt wird exp_0 zu einer Closure ausgewertet
- Mit Lambdas und Konstanten
 - Bei einem Funktionsaufruf kann exp_0 der Name einer Konstanten sein mit einer Closure als Wert

Lambda Kalkül

- Lambda Kalkül
 - Minimales Modell von Programmiersprachen
 - Vollständig durch formelle Regeln beschrieben
- Idee
 - Alle Sprachkonstrukte sind syntaktischer Zucker und lassen sich zu Lambda-Ausdrücken transformieren
 - Eigenschaften, die sich am Lambda-Kalkül beweisen lassen, gelten dann auch für diese Sprachkonstrukte
- Sprachkonstrukte, die sich durch Lambda-Ausdrücke realisieren lassen, z.B.:
 - Zahlen
 - Wahrheitswerte
 - Konstantendefinitionen
 - Listen
 - Strukturen