Allgemein	(: <Name> (<inputSig1> … --> <outputSig>)) (define Sig (signature (combined natural (predicate (lambda (a) (null? (mod a 5)))) --> [natural + durch 5 teilbar]
one-of	(one-of <lit1><lit2>…) Keine Signaturen! Bsp: (one-of 1 2 3)
Mixed (mindest eins)	(mixed <sig1> <sig2>…) Bsp.: (mixed natural boolean)
combined (alle)	(combined <sig1> <sig2>…) Bsp.: (combined natural integer)
predicate	(predicate (lambda (a) (…. [bool als output]))

Build-In Signaturen
number, real, rational, integer, natural
boolean, true, false
string
empty-list
any

rekursiv	Wiederholte Aufrufen von sich selbst
Endrekursion Interation	Eine rekursive Funktion ist endrekursiv, wenn der letzte rekursive Funktionsaufruf, der zur Berechnung vom Ergebnis ist. Benötigt Akkumulator, Größe bleibt gleich
promise	Versprechen, verzögert die Auswertung eines Ausdrucks bis es zu einem "force" kommt
force	Wertet "promise" aus
Curry	Funktionen, die ihre Argumente nacheinander konsumieren und daher partielle Applikation ermöglichen. (((lambda (a) (lambda (b) (+ a b))) 1) 2)
H.O.F	Funktionen höherer Ordnung, Funktionen die Funktionen produzieren
Record-Procedures	Datenstrukturen, beschreibt den Namen der Record-Signatur, den Namen des Konstruktors, den Namen des Prädikats und die Namen der Selektoren der einzelnen Datenfeldern
Streams	Datenströme, bestehen aus einem ersten Element (head) und einem Promise den Rest des Streams generieren zu können. (tail)
Bäume / Binärbäume Suchbäume	Ein Baum ist eine Datenstruktur, mit der sich hierarchische Strukturen abbilden lassen. Er besteht aus einer Wurzel (root). Diese ist über Äste mit n-vielen Kindern (binär n=2) verbunden. Jedes Kind ist ebenfalls ein Baum und nennt sich, falls es ein oder mehrere Kinder besitzt Knoten (node). Ein Knoten, der keine Kinder besitzt nennt sich Blatt (leaf). Jeder Knoten besitzt exakt einen Vater. Suchbäume sind Bäume, an denen die Kinder eine gewisse Regel befolgen, die das Suchen erleichtern (bsp rechtes Kind größer als Knoten…)
Lexikalische-Bindung	Die Stärke der Bindung von an einen Identifier nimmt von innen nach außen ab. (define x 5) ((lambda () (let ([x 0]) x))) -> 0

Streams

(head s) => erstes Element
(head (tails s)) => Error! Da (tail s) kein stream
(Sondern Promise)
(force (tail s)) => rest-stream
Bsp.:

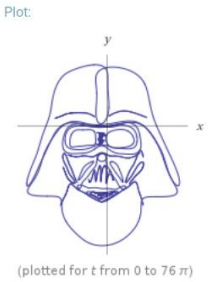
```
(define const-stream
  (lambda (x)
    (make-stream x (lambda () (const-stream x)))))
```

```
(define stream-intersect
  (lambda (s-one s-two)
    (letrec ([worker (lambda (s-one s-two x)
                       (if (= (head s-one) (head s-two))
                           x
                           (worker (force (tails s-one))
                                   (force (tails s-two))
                                   (+ x 1))))])
      (worker s-one s-two 0))))
```

Reduktionsschritte (⇝)

- Literal (1, #t, “abc”, …) [$eval_{lit}$]
  <l> ⇝ <l> (keine Red. möglich)
- Identifier (define…) [$eval_{ID}$]
  <id> ⇝ an id gebundener Wert
- Lambda (lambda (..)) [$eval_{\lambda}$]
  (lambda (…)..) ⇝ (lambda (…)..) (keine Red.)
- Applikation (<f> <e1> <e2> …)
  - <f>, <e1> , <e2>, … einzeln reduzieren ⇝ <f’>, <e1’>, <e2’> …
  - Falls <f’> primitive Operation, <f’> auf <e1’>, <e2’> … anwenden [$apply_{prim}$]
    Falls <f’> lambda Abstraktion, <e1’>, <e2’> … in Rumpf von <f’> einsetzen [$apply_{\lambda}$]

Bsp.: (sqr 9) ⇝ [$eval_{ID}$] ((lambda (x) (* x x) 9) ⇝ [$eval_{lit}$] ((lambda (x) (* x x) 9) ⇝ [$apply_{\lambda}$] (* 9 9)
⇝ [$eval_{ID}$] (#<procedure:*> 9 9) ⇝ [$eval_{lit}$] (#<procedure:*> 9 9) ⇝ [$apply_{prim}$] 81



https://www.wolframalpha.com/input/?i=darth+vader+curve

λ-Kalkül

Der Lambda-Kalkül ist eine formale Sprache zur Untersuchung von Funktionen

β-Redex Nächster Freier Term der Reduzierbar ist.

Normalform nicht reduzierbarer Term

free[]

- free[v] = {v}
- free[(e1.e2)] = free[e1] U free[e2]
- free[(λv.e1)] = (free[e1] \ {v})

bound[]

- bound[v] = ∅
- bound[(e1.e2)] = bound[e1] U bound[e1]
- bound[(λv.e1)] = bound[e1] U {v}

Wichtige Funktionen

Identität: (λx.x) Bsp.: ((λx.x) v) = v

Ω -Funktion: ((λ x (x x)) (λ x (x x))) Reproduziert sich bei β-Reduktion selbst!

???: ((λx.a) b) = a Nimmt zwar b, gibt aber a aus, da b nirgends hingetan wird

Capturing

Einfangen von freien Variablen (nicht erlaubt!)

((λx.(λy.(x y))) y)) ≠ ((λy.(y y))) !Achtung

= ((λa.(y a))) (Die innere Variablen werden Umbenannt)

B-Reduktion

1. Redex finden
2. Schauen ob es zu Capturing kommt, ggf. Variablen umbenennen
3. Freie! Variablen ersetzten Bsp.: ((λx.(λy. (y x)) a) = (λy. (y a))
4. Solange wiederhohlen bis Term in Normalform erreicht ist

- Notation <e/v>[<v> wird reduziert mit -> <a>]
- x[x->a] = a
- v[x->a] = v; v≠x
- (e1 e2) [x->a] = (e1[x->a] e2[x->a])

Church-Rosser-Konfluenz: Alle Reduktionswege führen zum selben Ausdruck

Wichtige Definitionen

Records

```
(define-record-procedures name
  make-name
  name?
  (name-selektor1 name-selektor2 ...))
(define-record-procedures-parametric name name-of
  make-name
  name?
  (name-selektor1 name-selektor2 ...))
```

Curry

```
(define curry
  (lambda (f)
    (lambda (a)
      (lambda (b)
        ((f a) b)))))
```

Uncurry

```
(define uncurry
   (lambda (f)
     (lambda (a b) ((f a) b))))
```

Signatur

```
(define list-of-10
  (lambda (t)
    (signature (combined
                (list-of t)
                (predicate (lambda (l) ((= (length l)) 10)))
))))

(define searchtree-of
   (lambda (bt)
     (signature (combined
     (btree-of bt)
      (predicate search-tree?))))))   !Achtung Benötigt btree-search
```

Binärbäume

```
(: btree-fold (%b (%b %a %b -> %b) (btree-of %a) -> %b))
(define btree-fold
  (lambda (z c t)
    (cond
      ((empty-tree? t) z)
      ((node? t)
            (c (btree-fold z c (node-left-branch t))
               (node-label t)
               (btree-fold z c (node-right-branch t)))))))

(: btree-map ((%a -> %b) (btree-of %a) -> (btree-of %b)))
(define btree-map
 (lambda (f t)
  (btree-fold empty-tree
          (lambda (l x r) (make-node l (f x) r))
          t)))
```

Wichtige Signaturen

```
(: map ((%a -> %b) (list-of %a) -> (list-of %b))
(: fold (%a (%b %a -> %a) (list-of %b) -> any))
(: btree-fold (%b (%b %a %b -> %b) (btree-of %a) -> %b))
```

```
(: btree-depth ((btree-of %a) -> natural))
(define tree–depth
  (lambda (bt)
    (cond
      ((empty–tree? bt) 0)
      ((node? bt ) (+ 1 (max
                          (tree–depth(node–left–branch bt))
                          (tree–depth(node–right–branch bt))
 ) ) ) ) ) )

(: search-tree? ((btree-of real) -> boolean))
(define search-tree?
   (lambda (bt)
     (match bt
       ((make-empty-tree) #t)
       ((make-node l n r) (and (< (btree-max l) n (btree-min r))
                               (search-tree? l)
                               (search-tree? r))
                               ))))

!Achtung benötigt btree-max / min
```