Frage 1: 1. Funktionale Programmiersprache?

LISP (**Lis**t **P**rocessing)
LIPS ist die zweitälteste verwendete Programmiersprache und basiert auf Lambda Kalkülen.

b) Definieren Sie den Begriff *side effect und geben Sie ein Bsp.*:
Einige funktionale Programmiersprachen ermöglichen die Auswertung von Ausdrücken, um Aktionen zusätzlich zu den Rückgabewerten zu liefern. Diese Aktionen werden „side effects“ genannt.

Side Effects werden verwendet:

- für I/O
- Wenn die Programm Struktur klarer wird
- Wenn sie für die Effizient notwendig sind

Beispiel:

```
(define x 10) x
(set! x (+ x 1)) x
```

c) Wie nennt man Sprachen ohne Side Effects?
Antwort: pure functions

Sprachen die side effects verbieten nennt man *pure languages*.

Frage 2: Was sind typische Merkmale eines Programms welches dem funktionalen Paradigma entspricht?

Grundlage:	Lambda Kalkül
Abstraktionsprinzip:	Funktionen
Charakteristik:	Referentielle Transparenz
Bedeutung:	Basis vieler Programmiersprachen
Sprachen:	LISP, Scheme, Racket, Scala, Haskell, ...
Vorteile:	- Einfach zu lernen - Hohe Produktivität - Höhere Zuverlässigkeit
Nachteile:	- Geringe Performance - Teilw. unangemessen für Zustandsbasierte Anwendungen

Frage 3: Erklären Sie das Konzept Functional Interface (auch genannt SAM Interface) und seine Bedeutung bei der Umsetzung von funktionaler Programmierung in Java 8.

Eine funktionale Schnittstelle spezifiziert nur eine abstrakte Methode. Es kann jedoch eine beliebige Anzahl von Standard- oder statischen Methoden enthalten. Java 8 hat Lambda Expressions eingeführt, um die Schnittstelle zu instanziieren und um einfach auf die abstrakte Methode zugreifen zu können.

```
@FunctionalInterface
public interface AnnotationTest {
```

```
int foo();
}
```

Frage 4: Definieren Sie den Begriff High-Order-Procedure und geben Sie ein Beispiel.

High-Order-Procedures

- Akzeptieren Prozeduren als Argument
- Returnt Prozeduren

Diese Punkte erhöhen die Ausdruckskraft der Sprache.

Beispiel:

```
(define  (sum-integers  a  b)
        (if  (>  a  b)
                0
                (+  a  (sum-integers  (+  a  1)  b))))
```

Frage 5: Zeigen Sie, dass sich *compound data* ohne Datenstrukturen, jedoch mithilfe von first-class functions realisieren lassen. Implementieren Sie hierfür *mycons, mycar, mycdr,* ohne die jeweiligen built-in Funktionen oder *define-struct* zu verwenden.
z.B.: (mycar (mycons 3 4)) → 3.

```
(define  (mycons  x  y)
  (λ  (m) (m x y)))

(define  (mycar  z)  z (lambda p q) p)
(define  (mycdr  z)  z (lambda p q) q)
```

a) Implementieren Sie die Funktion *myfoldr*, die sich so verhält wie *foldr* jedoch ohne *foldr* zu verwenden.

```
(define (myfoldr  fun  init-val  list)
  (if  (null?  list)
     init-val
     (fun  (car  list)  (myfoldr  fun  init-val  (cdr  list)))))

(myfoldr + 5 '(1 2 3))
```

b) Geben Sie an, wie sich damit eine Funktion *sum*, welche die Summe einer Liste bildet, einfach implementieren lässt. z.B. (sum ,(1 2 3 4)) → 10.

```
(define (mySum list)
  (foldr + 0 list))
```

Frage 6: Erklären Sie die Funktionsweise von *stream-cons*, *stream-first*, und *stream-rest*. Alternativ können Sie auch die Implementierung der Funktionen angeben (mit Kommentaren).

Stream-cons:
(stream-cons *first-expr rest-expr*) // erzeugt einen Stream für das erste Element des Streams sowie einen Stream mit den restlichen Elementen des Streams

Stream-first:
Angenommen: (define s (stream 1 2 3))
(stream-first s) // liefert das erste Element des Streams (also 1)
(stream-first (stream-rest s)) //liefert das erste Element des Restes des Streams (also 2)

Stream-rest:
(stream-rest s) // liefert einen Stream mit allen Elementen außer dem ersten Element

a) Was ist die Idee von Streams und was sind die Vorteile gegenüber Listen?

- Streams arbeitet mit Sequenzmanipulationen, ohne die Kosten der Manipulation von Sequenzen als Listen zu verursachen
- Formuliert Programme elegant als Sequenzmanipulationen, während die Effizienz der inkrementellen Berechnung erreicht wird
- Es wird nur teilweise ein Stream konstruiert und die Teilkonstruktion wird an das Programm weitergegeben, das den Stream konsumiert.

Vorteile gegenüber Listen:
- Der Unterschied ist der Zeitpunkt zu welchem Elemente bewertet werden
- **Stream**: Restliche werden in der selection-time ausgewertet
- **Listen**: Erste und restliche werden in der construction-time ausgewertet

b) Geben Sie ein Beispiel für einen unendlichen Stream (in Racket Code).

```
(define  (fibgen  x  y)
  (stream-cons  x (+  x y)))

(define fibs  (fibgen  0  1))
```

Frage 7: Closure (Funktionsabschluss)

Ein Funktionsabschluss ist eine Datenstruktur, die

- Eine Funktion zusammen mit einer Umgebung speichert
- Jede freie Variable der Funktion mit dem Wert verknüpft, zu dem der Name zum Zeitpunkt der Erstellung des Abschlusses gebunden war

Beispiel:

```
(define (startAt x)
  (define (incrementBy y)
    (+ x y))
  incrementBy)

(define closure1 (startAt 1))
(define closure5 (startAt 5))

(closure1 9)
(closure5 9)
```

Frage 8: Normal evaluation vs. applicative evalution mit Beispiel.

Scheme ist eine applicative-order language.

Applicative evaluation:
Wertet Unterausdrücke aus, kurz bevor die Prozedur angewendet wird.

Normal evaluation:
Durchläuft Subexpressionen wie sie sind, ohne Auswertung und geht mit der Auswertung nur weiter, wenn der entsprechende formale Parameter tatsächlich selbst ausgewertet werden soll.

Beispiel:

```
 (define (p) (p) )

(define (test x y) (if (= x 0)
                       0
                       y))

(test 0 p)
```

Bei applicative evaluation von (Test 0 (p)) werden zuerst die Argumentunterausdrücke auswerten also 0 und (p).

Bei der normal evaluation werden die Argumentunterausdrücke nicht berührt. Der Aufruf (test 0 p) wird durch den Rumpf des Funktionstests ersetzt.

Frage 9: Lambda Kalküle und Currying

Lambda Kalküle:
Funktionale Programmierung basiert auf Lambda Kalkülen.

- Ist ein mathematischer Formalismus
- Eingeführt durch Alonzo Church (1941)

Einen anonyme racket Funktion mit Lambda schaut beispielsweise so aus:

```
(lambda (x) (* 2 x))
```

Merkmale:

- Der Body einer Abstraktion kann ein beliebiger gültiger Lambda-Ausdruck sein.
- Alle Lambda Abstraktionen beinhalten ein einzelnes *lambda* und eine einzelne gebundene Variable
- Nur ein Parameter ist erlaubt

Currying:
beschreibt die Konvertierung einer Multi-Argument-Funktion in eine Kette von einzelnen Argumentfunktionen.

Notation:
f: A x B x C → D non curried
f: A → B → C → D curried

Beispiel:

```
(define (add3 x y z) (+ x y z))
(define curried_add3 (curry add3))
(define curried_add3_2 (curry add3 1 2))
(curried_add3_2 6)

Ausgabe: 9
```

Partielle Anwendung:
beschreibt die Konvertierung einer Funktion mit mehreren Argumenten in eine, die weniger Argumente akzeptiert.

- Es wendet teilweise einige Argumente auf eine Funktion an und gibt eine Funktion mit einer Signatur zurück, die aus den restlichen Argumenten besteht

Frage 10: Tail-Recursion

Eine rekursive Funktion ist tail-recursive, wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung der Funktion ist.

Beispiel:

```
(define (tail-factorial n [acc 1])
  (if (= n 1)
      acc
      (tail-factorial (- n 1) (* n acc)))) ; tail recursive

(tail-factorial 3)
```

Ausgabe: 6

Indem wir einen Akkumulator verwenden, können wir die Multiplikation innerhalb des rekursiven Aufrufs verschieben, wodurch diese Version tail rekursiv wird.

Frage 11: Last implementieren. Last liefert das letzte Element einer Liste zurürck.

```
(define (last xs)
  (if (null? (cdr xs))
      (car xs)
      (last (cdr xs))))
```

Frage 12: count-leaves implementieren. Count-leave liefert die Anzahl an Blätter in einem Baum.

```
(define x (cons (list 1 2) (list 3 4)))
(length x)

(define (leave? x)
  (not (pair? x)))

(define (count-leaves x)
  (cond [(null? x) 0]
        [(leave? x) 1]
        [else (+ (count-leaves (first x)) (count-leaves (rest x)))]))

;; Reverse for trees
```

```
(define (deeprev l)
  (if (list? l)
      (reverse (map deeprev l))
      l))

(define y (list '(1 2) '(3 4) 5 '(6 7 8)))
(deeprev y)
```

Zusatz:

Append:

```
(define (myappend xs ys)
  (if (null? xs)
      ys
      (cons (first xs) (myappend (rest xs) ys))))
```

Reverse:

```
(define (rev xs)
  (if (null? xs)
      null
      (append (rev (cdr xs)) (list (car xs)))))
```

Mapping over Lists:

```
(define (map proc items)
  (if (null? items)
      null
      (cons (proc (car items))
            (map proc (cdr items)))))
```

Letztes Element abschneiden:

```
(define mylist '(1 2 3 4 5 6 7))

(define (init xs)
  (cond ((null? xs) (error "empty list"))
        ((null? (cdr xs)) null)
        (else (cons (car xs) (init (cdr xs))))))

Aufruf: (init mylist)
Ausgabe: '(1 2 3 4 5 6)
```