Aufgabe 1: Coole Typen

Teil (a)

Ungetypte Sprachen

- Keine Datentypen (Interpretationssache)
- Operationen können auf allen Arten von Daten ausgeführt werden.
- Korrektheit liegt in der Hand des Programmierers.
- Beispiel: Assembler

Dynamisch getypte Sprachen

- Der Typ eines Wertes wird zur Laufzeit überprüft.
- Fehler können erst sehr spät entdeckt werden oder vielleicht gar nicht, da die Tests nicht alle Eventualitäten abdecken.

Statisch getypte Sprachen

- Datentypen werden vor der Ausführung festgelegt/überprüft.
- Der Typ eines zusammengesetzten Programmteils hängt von den Typen der Komponenten ab.

Teil (c)

Die Zuordnung eines *Contracts* zu einer Funktion ermöglicht die Typprüfung zur *Laufzeit*, damit ist also noch keine statische Typprüfung gegeben. Wie im Tutorium angesprochen, werden Contracts innerhalb einer Datei ohnehin nicht geprüft, aber auch beim Import über `(require "foobar.rkt")` in einer anderen Datei geschieht dies nicht sofort. Lediglich der Aufruf einer Funktion aus der importierten Datei löst die Typprüfung durch den Contract aus.

Teil (d)

Contracts stehen uns in lediglich in Standardracket zur Verfügung. Dazu also wieder die Sprache umstellen und die entsprechende Zeile einfügen lassen bzw. dies selber vornehmen. Gegebenenfalls muss dann noch `racket/contract` importiert werden.

```
#lang racket
(require racket/contract)

; Number (list-of Number) -> (list-of Number)
; Adds the given number to all the elements in the given list.
```

```
(define (add/all n lon) (map (lambda (x) (+ n x)) lon))

(provide
 (contract-out
  [add/all (-> number? (listof number?) (listof number?))]))
```

Contracts werden alle gesammelt in einer **provide**-Klausel definiert. Bei mehreren Funktionen sieht das etwa so aus:

```
(provide
 (contract-out
  [add/all (-> number? (listof number?) (listof number?))]
  [helloworld (-> string?)]
  [add2 (-> number? number?)]))

(define (helloworld) "Hallo Welt!")
(define (add2 n) (+ 2 n))
```

Die Position im Programm ist beliebig und nicht alle Funktionen müssen einen Contract besitzen, denn dieser soll lediglich eine Schnittstelle bilden. Spielt am besten nochmal ein bisschen mit verschiedenen Dateien rum, die ihr gegenseitig importiert.

Teil (e)

In Typed Racket ist die Syntax leicht anders, aber wie ich finde kompakter und übersichtlicher. Typen werden im Gegensatz zu den Contracts großgeschrieben und Fragezeichen kommen auch nicht vor. Mit dieser Regel im Kopf sollten die Unterschiede sollte das runterschreiben beider Varianten einfach von der Hand gehen.

```
#lang typed/racket

(: add/all (-> Number (Listof Number) (Listof Number)))
(define (add/all n lon) (map (lambda ([x : Number]) (+ n x)) lon))
```

Wenn ihr diesen Programmcode in aller Ausführlichkeit studiert habt, dann ist euch sicherlich aufgefallen, dass der Parameter der Lambdafunktion explizit als Zahl deklariert wurde. Das ist einer der Fallstricke von Typed Racket, besonderlich intelligent ist es nicht, denn eigentlich lässt sich an dieser Stelle ohne weiteres eine Typinferenz vornehmen:

Durch das `map` wird der Lambdafunktion in jedem Schritt ein Element der Liste übergeben, deren Typ ist aber durch die Signatur klar als Liste von *Zahlen* definiert und somit kann `x` auch nur eine Zahl sein. Aber nein, Typed Racket wüsste das gerne nochmal explizit von uns.

Somit gibt es zwei Grundregeln, wenn man Typed Racket benutzten möchte. Jede Funktionsdefinition (und Definition einer Lambdafunktion, die direkt einer Variablen zugeordnet wird) braucht eine Signatur und jedes Argument einer Lambdafunktion.

Damit ist Typed Racket also klar von Sprachen wie Scala, Haskell und Rust (yay!) zu unterscheiden, bei denen Typen auch selbst inferiert werden.

Aufgabe 2: Advanced Dicing Technologies

Teil (b)

Für die Verwendung des `define-type`-Konstrukts, wie wir es in dieser Vorlesung verwenden ist wieder der Einsatz der ISL oder höher und ein neues Teachpack `abstraction.rkt` notwendig. In Typed Racket gibt es auch ein `define-type`, dies ist allerdings nicht das richtige.

Wir wollen nun also einen ADT der folgenden Form implementieren:

```
; A BST is one of:
; - (make-leaf Number)
; - (make-node Number BST BST)
; interp. as binary search tree for numbers
```

Das Grundschema dazu sieht so aus:

```
(define-type BST
  [leaf ...]
  [node ...])
```

`BST`, `leaf`, `node` sind alles frei wählbare Begriffe. Wenn wir – warum auch immer – zwei verschiedene Arten von Blättern haben wollten, wäre dies einfach wie folgt möglich.

```
(define-type BST
  [leaf1 ...]
  [leaf2 ...]
  [node ...])
```

Jetzt müssen wir allerdings die Einträge noch mit Inhalt/Signaturen füllen. Dazu schauen wir uns eine Alternative des ADTs genauer an:

```
  [leaf (value number?)]
```

Damit ist festgelegt, dass sofern wir einen BST der Form `leaf` haben, dieser nur einen Eintrag enthält, der den Namen `value` besitzt und vom Typ eine Zahl sein muss. Der Name ist dabei auch wieder frei wählbar. Knoten wollen natürlich mehr als nur aus einer Zahl zu bestehen:

```
  [node (value number?) (left BST?) (right BST?)]
```

Deshalb liegen hier insgesamt drei Einträge vor, mit den Namen `value`, `left` und `right`, die letzteren beide müssen dabei vom Typ `BST` sein. Unser Baum ist also rekursiv definiert. Wieder gilt hier, dass die Namen beliebig sein können. Ebenso können – wie ihr seht – beliebig viele Einträge pro Alternative des ADTs definiert werden, womit sie sich wie Strukturen verhalten. Das sehen wir auch dann schön, wenn wir mal eine konkrete Instanz des BSTs definieren:

```
(define EINBAUM (make-node 2 (make-leaf 1) (make-leaf 3)))
```

Nun müssen wir noch Funktionen implementieren, die mit dieser Datenstruktur umgehen können. Dazu gibt es die Funktion `type-case`. Die Verwendung ist ganz einfach und analog zu **`match`**. Das erste Argument ist der Name des definierten Typs mithilfe dessen wir ein Pattern Matching durchführen wollen, in unserem Fall also `BST`. Das zweite Argument ist eine konkrete Instaz dieses Typs und dann kommen die verschiedenen Fälle, natürlich nur soviele, wie unser ADT an Alternativen besitzt. In den verschiedenen Fällen können wie beim **`match`** direkt Namen für die gematchten Parameter vergeben werden. Wie man sieht wurde hier etwa `val` für den Inhalt eines Knoten/Blattes gewählt, um eine Überdeckung des ursprünglichen Funktionsparameters `value` zu vermeiden.

```
(define-type BST
  [leaf (value number?)]
  [node (value number?) (left BST?) (right BST?)])

; Number BST -> BST
; Inserts the given number into the tree.
(define (tree/insert value tree)
  (type-case BST tree
    [leaf (val)
          (if (<= value val)
              (make-node value (make-leaf value) (make-leaf val))
              (make-node val   (make-leaf val)   (make-leaf value)))]
    [node (val left right)
          (if (<= value val)
              (make-node val (tree/insert value left) right)
              (make-node val left (tree/insert value right)))]))

; Number BST -> Boolean
; Checks if the given tree contains the given Number.
(define (tree/contains value tree)
  (type-case BST tree
    [leaf (val)            (= value val)]
    [node (val left right) (if (<= value val)
                               (tree/contains value left)
                               (tree/contains value right))]))
```