Prof. Dr. Ralf Hinze
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Lösungshinweise/-vorschläge zum Übungsblatt 5: Grundlagen der Programmierung (WS 2023/24)

Listen in F# Auf dem letzten Übungsblatt haben wir bereits den Typ `Nats` für Listen natürlicher Zahlen verwendet. In der Vorlesung haben wir nun auch parametrisierte Typen kennengelernt. Unter anderem wurde der folgende Typ für Listen definiert: `type List<'a> = | Nil | Cons of 'a * List<'a>`

Diese Typdefinition kommt mit den bislang bekannten Sprachelementen aus: Rekursive Varianten und Paare. Wir können diese Typdefinition auch genauso in F# verwenden und z.B. die Liste der Zahlen `1N` und `2N` als `List<Nat>` durch `Cons (1N, Cons (2N, Nil))` darstellen. In F# gibt es jedoch auch einen vordefinierten Typ für Listen, für den es drei Notationen gibt: `'a list`, `list<'a>` sowie `List<'a>` (unglücklicherweise schreibt sich der in F# eingebaute Typ `List<'a>` gleich wie der Listentyp aus der Vorlesung, jedoch hat der F# Typ die Konstruktoren `[]` und `::`). Dieser Typ ist nicht kompatibel mit dem selbst definierten Typ `List<'a>` aus der Vorlesung. Es ist daher wichtig zu wissen, welchen Typ man gerade verwendet. Die Unterschiede finden Sie auf Vorlesungsfolie 402.

In den Übungen verwenden wir überwiegend den in F# vordefinierten Typ für Listen. Dies hat den Vorteil, dass wir eine kürzere Notation für die Darstellung der Listen nutzen können: `[1N; 2N; 3N]` statt `Cons (1N, Cons (2N, Cons (3N, Nil)))`. Außerdem werden wir einige vordefinierte Funktionen (sogenannte Bibliotheksfunktionen) kennenlernen, die auch nur mit dem vordefinierten Listen-Typ funktionieren.

Typparameter Einschränkungen in F# In der Vorlesung haben wir polymorphe Funktionen kennengelernt. Der Typ enthält hierbei einen Typparameter, sodass die Funktion für verschiedene Typen nutzbar wird. Oft ist man jedoch nicht ganz frei in der Wahl des Typs: Wenn Elemente des Typs miteinander verglichen werden (=, <, <=, usw.), dann muss der Typ diese Vergleichsoperation unterstützen. Eine polymorphe `contains` Funktion, die überprüft ob ein gegebener Wert vom Typ `'a` in einer Liste vom Typ `List<'a>` enthalten ist, setzt voraus, dass der Typ `'a` die Gleichheit unterstützt. Die meisten Typen unterstützen sowohl Gleichheit als auch Ordnungsvergleiche; Strings und Listen sind beispielweise lexikografisch geordnet. Es gibt aber auch Typen, die diese Vergleichsoperationen nicht unterstützen. Das sind insbesondere die Funktionstypen. Der Ausdruck `(fun (x: Nat) -> x + x) = (fun (x: Nat) -> 2N * x)` ist also nicht wohlgetypt, da der Typ `Nat -> Nat` die Gleichheit nicht unterstützt. Der Typparameter für polymorphe Funktionen kann wie folgt eingeschränkt werden: `let contains<'a when 'a : equality> (x: 'a) (xs: 'a list): Bool = ....` Dabei ist `equality` die Einschränkung, dass die Gleichheit (=) unterstützt werden muss. Für Ordnungsvergleiche (<, <=, `min`, ...) heißt die Einschränkung `comparison` und beinhaltet automatisch auch die Gleichheit. Sie brauchen sich nicht weiter damit zu befassen, jedoch werden manche Vorlagen derartige Typeinschränkungen enthalten. Diese müssen Sie so in der Vorlage stehen lassen, da der Code ansonsten nicht mehr kompilieren wird.

Aufrufen von polymorphen Funktionen in F# Wie auf Vorlesungsfolie 398 beschrieben, kann der Typparameter beim Aufrufen polymorpher Funktionen meist weggelassen werden. Allerdings gilt dies nicht immer, wenn der Typparameter eingeschränkt ist (siehe vorheriger Abschnitt). Die vordefinierte Funktion `max<'a when 'a : comparison>: 'a -> 'a -> 'a` gibt das größere der beiden Argumente zurück. Wir können `max<List<Nat>> [5N; 6N] [7N; 1N]` ohne Typparameter aufrufen (`max [5N; 6N] [7N; 1N]`), da F# den Typparameter `List<Nat>` aus den Argumenten bestimmen kann. Sind die beiden Eingabelisten jedoch leer, dann ist diese Verkürzung nicht möglich: `max [] []` gibt einen Fehler, während `max<List<Nat>> [] []` funktioniert. Wenn Sie also den Fehler `FS0030: Value restriction` erhalten, müssen Sie beim Funktionsaufruf den Typparameter explizit angeben.

Vorbereitung auf die Klausur Wir legen Ihnen ans Herz, mit der Klausurvorbereitung rechtzeitig zu beginnen. Sie können sich mit Hilfe der alten GdP Klausuren im KAI System[1] einen Eindruck vom Aufbau der Klausur verschaffen.

Als Hilfsmittel für die Klausuren sind zwei beidseitig handschriftlich beschriebene DIN A4 Blätter zugelassen. Beginnen Sie möglichst schon jetzt damit diese vorzubereiten. Schreiben Sie Dinge auf, die Sie nicht auswendig lernen möchten, aber dennoch hilfreich bei der Bearbeitung von Klausuraufgaben sein könnten. Dies sind zum Beispiel die Regeln der statischen und dynamischen Semantik. Ansonsten könnten noch die Parameter- und Rückgabetypen einiger nützlicher Bibliotheksfunktionen, die Sie zum Lösen der Übungsaufgaben bereits benutzt haben, hilfreich sein. Beachten Sie, dass bereits das Erstellen dieser "Spickzettel" einen Lernprozess darstellt. Sie sollten sich also Ihre eigenen Blätter konzipieren und nicht von Kommilitoninnen und Kommilitonen abschreiben.

Aufgabe 1 Parametrische Listen (Präsenzaufgabe)

Motivation: In dieser Aufgabe sollen Sie sich mit den in F# eingebauten parametrischen Listen vertraut machen. Sie können sich an den Vorlesungsfolien 379 bis 404 sowie am Skript Kapitel 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Lists.fs` *aus der Vorlage* `Aufgabe-5-1.zip`.

Wir betrachten unter anderem einige aus Übungsblatt 4, Aufgabe 1 und 3 bekannte Funktionen noch einmal und verallgemeinern diese. Dazu werden die in F# eingebauten parametrischen Listen verwendet. Bitte beachten Sie die Hinweise zu Listen in F# auf der ersten Seite.

Hinweis: Verwenden Sie in Ihrer Lösung ***nicht*** *das List-Modul aus der Standardbibliothek.*

Wir verwenden bei einigen Teilaufgaben folgende Beispielliste:

```
let ex = [2N; 4N; 3N; 4N; 2N; 1N]
```

a) Schreiben Sie eine Funktion `plusOne: List<Nat> -> List<Nat>`, die eine Liste natürlicher Zahlen nimmt und zu jeder Zahl in der Liste die Zahl `1` addiert. Vergleichen Sie mit der gleichnamigen Funktion von Übungsblatt 4, Aufgabe 1.

Beispiele:

```
plusOne [] = []
plusOne ex = [3N; 5N; 4N; 5N; 3N; 2N]
```

```
let rec plusOne (xs: List<Nat>): List<Nat> =
    match xs with
    | []    -> []
    | x::ys -> (x + 1N)::(plusOne ys)
```

Gegenüber Aufgabe 1 von Übungsblatt 4 müssen nur die Konstruktoren ausgetauscht werden. Statt `Nil` haben wir jetzt `[]` und anstelle von `Cons (x, xs)` (Präfix) schreiben wir `x::xs` (Infix).

[1] https://kai.informatik.uni-kl.de/, Abruf nur aus dem Uni-Netz bzw. VPN https://rz.rptu.de/vpn/.

b) Schreiben Sie eine Funktion `filter<'a>: ('a -> Bool) -> List<'a> -> List<'a>`, die eine Funktion `p` und eine Liste `xs` nimmt und die Liste der Elemente aus `xs` zurückgibt, für die `p` **`true`** zurückgibt.

Beispiele:

```
filter (fun x -> x > 3N) [] = []
filter (fun x -> x > 3N) ex = [4N; 4N]
filter (fun x -> x <= 3N) ex = [2N; 3N; 2N; 1N]
```

```
let rec filter<'a> (p: 'a -> Bool) (xs: List<'a>): List<'a> =
    match xs with
    | []    -> []
    | x::xs ->
        if p x
          then x::(filter p xs)
          else filter p xs
```

c) Schreiben Sie eine Funktion `concat<'a>: List<'a> -> List<'a> -> List<'a>`, die zwei parametrische Listen `xs` und `ys` nimmt und deren Konkatenation berechnet, also die Liste in der zuerst alle Elemente aus `xs` und dann die Elemente aus `ys` kommen. Verwenden Sie nicht den in F# eingebauten Konkatenationsoperator `@`.

Beispiele:

```
concat [] ex = ex
concat ex [] = ex
concat [1N] [2N] = [1N; 2N]
```

```
let rec concat<'a> (xs: List<'a>) (ys: List<'a>): List<'a> =
    match xs with
    | [] -> ys
    | x::zs -> x::(concat zs ys)
```

d) Schreiben Sie eine Funktion `mirror<'a>: List<'a> -> List<'a>`, die eine Liste nimmt und die gespiegelte Liste berechnet, also eine Liste in der die Elemente in umgekehrter Reihenfolge enthalten sind.

Beispiele:

```
mirror [] = []
mirror ex = [1N; 2N; 4N; 3N; 4N; 2N]
```

```
// Lösung mit Laufzeit quadratisch in der Länge von xs
let rec mirror<'a> (xs: List<'a>): List<'a> =
    match xs with
    | [] -> []
    | x::ys -> concat (mirror ys) [x]

// Effizientere Lösung (lineare Laufzeit)
let mirror'<'a> (xs: List<'a>): List<'a> =
    // Hilfsfunktion berechnet concat (mirror xs) zs
    let rec mirrorConcat (xs: List<'a>) (zs: List<'a>): List<'a> =
        match xs with
        | [] -> zs
        | x::ys -> mirrorConcat ys (x::zs)
    mirrorConcat xs []
```

e) Schreiben Sie eine Funktion `sum: List<Nat> -> Nat`, die eine Liste natürlicher Zahlen nimmt und die Summe der Zahlen zurückgibt.

Beispiele:

```
sum [] = 0N                                    sum ex = 16N
```

```
let rec sum (xs: List<Nat>): Nat =
    match xs with
    | [] -> 0N
    | x::ys -> x + sum ys
```

Aufgabe 2 Prioritätswarteschlange (Einreichaufgabe, 12 Punkte)

Motivation: In dieser Aufgabe sollen Sie das Programmieren mit Records und parametrisierten Typen einüben. Sie können sich an den Vorlesungsfolien 281 bis 305 und 379 bis 404 sowie am Skript Kapitel 4.1 und 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `PriorityQueue.fs` *aus der Vorlage* `Aufgabe-5-2.zip`.

Eine Prioritätswarteschlange (*engl. priority queue*) ist eine Datenstruktur, die Werte anhand einer ihnen zugeordneten Priorität verwaltet. Die Einträge in der Warteschlange sollen damit geordnet nach ihrer Priorität in einer möglichst effizienten Weise abgerufen werden können.

In dieser Aufgabe modellieren wir die Elemente der Prioritätswarteschlange durch den Typ `QElem<'a>` ("Queue Element"), der als Record definiert ist. Dieser Record besteht aus einem Feld `priority`, welches die Priorität repräsentiert sowie dem Feld `value`, in dem der eigentliche Inhalt abgelegt wird.

```
type QElem<'a> =        // Element der Warteschlange
    { priority: Nat     // Priorität
      value: 'a }       // beliebiger Wert
```

Die Warteschlange selbst wird durch den Typ `PQ<'a>` modelliert. Wie Sie sehen, handelt es sich lediglich um einen kürzeren Namen für eine Liste von Warteschlangenelementen.

```
type PQ<'a> = List<QElem<'a>> // Die Prioritätswarteschlange
```

Wir verwenden hier die Konvention, dass niedrigere Werte von `priority` einer höheren Priorität entsprechen. Damit entspricht `priority=0N` der höchsten Priorität. Da wir die Prioritätswarteschlange durch eine Liste repräsentieren und Zugriffe auf das Element mit der höchsten Priorität möglichst effizient sein sollen, verlangen wir, dass die Liste immer aufsteigend nach dem Wert von `priority` sortiert sein soll (der Zugriff auf das Kopfelement ist in konstanter Zeit möglich). Wir betrachten in den Teilaufgaben also nur Warteschlangenlisten, die aufsteigend nach dem Wert von priority sortiert sind. Funktionen, die wir auf den Warteschlangen definieren, müssen diese Eigenschaft erhalten.

Falls mehrere Elemente dieselbe Priorität haben (Duplikate bezüglich `priority`), spielt deren Reihenfolge untereinander keine Rolle.

Hinweis: Sie dürfen das List-Modul aus der Standardbibliothek nicht verwenden.

Hinweis: Beachten Sie die Hinweise auf der ersten Seite.

In der Vorlage sind folgende Beispiele hinterlegt:

```
let exQueue1 =
    [ {priority=1N; value=4711N}
    ; {priority=3N; value=815N}
    ; {priority=9N; value=42N}
    ]

let exQueue2 =
    [ {priority=4N; value=123N}
    ; {priority=6N; value=456N}
    ; {priority=8N; value=789N}
    ]

let exElem = {priority=5N; value=7N}
```

a) Schreiben Sie eine Funktion `isEmpty<'a>: PQ<'a> -> Bool`, die eine Prioritätswarteschlange als Argument erwartet und zurückgibt, ob die Warteschlange leer ist oder nicht.

```
let isEmpty<'a> (xs: PQ<'a>): Bool =
    match xs with
    | [] -> true
    | _ -> false
```

Der Typ `PQ<'a>` der Prioritätswarteschlange ist ein Synonym für eine Liste mit Elementen vom Typ `QElem<'a>`. Wir können also mit Hilfe von **`match`** die entsprechende Fallunterscheidung durchführen. Wenn wir versuchen mit `xs = []` zu prüfen, ob die Liste leer ist, erhalten wir die Fehlermeldung `FS0001 : Einem Typparameter fehlt die Einschränkung "when 'a : equality"`. Die Prüfung auf Gleichheit ist nur möglich, wenn dies explizit für den Typparameter gefordert wird (s. Seite 1).

b) Schreiben Sie eine Funktion `insert<'a>: QElem<'a> -> PQ<'a> -> PQ<'a>`, welche ein Element `x` sowie eine Warteschlange `xs` nimmt. Das Element `x` soll in die Warteschlange `xs` eingefügt und die resultierende Warteschlange als Ergebnis zurückgegeben werden.

Beachten Sie, dass die Warteschlange bezüglich `priority` sortiert sein soll. Sie müssen also beim Einfügen sicherstellen, dass die Sortierung erhalten bleibt.

```
let rec insert<'a> (x: QElem<'a>) (xs: PQ<'a>): PQ<'a> =
    match xs with
    | [] -> [x]
    | y::ys ->
        if x.priority <= y.priority then x::xs
        else y::(insert x ys)
```

Wir unterscheiden beim Einfügen zwei Fälle:

1. Falls die Warteschlange leer ist, geben wir eine einelementige Liste zurück. Eine einelementige Liste ist bereits sortiert.
2. Falls die Warteschlange nicht leer ist, müssen wir zunächst prüfen, ob das einzufügende Element eine kleinere oder gleich große Priorität hat wie das Kopfelement. Wenn das der Fall ist, hängen wir es vorne an die Warteschlange an, ansonsten machen wir rekursiv mit der Restliste weiter.

c) Schreiben Sie eine Funktion `extractMin<'a>: PQ<'a> -> Option<QElem<'a>> * PQ<'a>`, die eine Warteschlange `xs` nimmt und daraus das Element mit der höchsten Priorität (also dem kleinsten `priority` Wert) entfernt. Als Resultat soll die Funktion ein Tupel zurückgeben, das aus dem entfernten Element besteht und der Warteschlange, die übrig bleibt, nachdem das Element entfernt wurde.

Wir müssen jedoch berücksichtigen, dass die Prioritätswarteschlange leer sein kann. In diesem Fall kann kein Element entnommen werden. Wir verwenden daher den sogenannten Optionstyp, der in F# bereits wie folgt vordefiniert ist: **`type`** `Option<a> = | None | Some` **`of`** `a`. Wenn die gegebene Prioritätswarteschlange leer ist, dann soll die erste Komponente des Rückgabe-Tupels `None` sein und die zweite Komponente die unveränderte (leere) Prioritätswarteschlange.

```
let extractMin<'a> (xs: PQ<'a>): Option<QElem<'a>> * PQ<'a> =
    match xs with
    | [] -> (None, xs)
    | y::ys -> (Some y, ys)
```

Die Aufgabenbeschreibung weist darauf hin, dass der Zugriff auf das Element mit der höchsten Priorität besonders effizient möglich sein soll. Daher haben wir gefordert, dass die Liste, welche die Prioritätswarteschlange repräsentiert aufsteigend bezüglich `priority` sortiert ist. Das Element mit der größten Priorität, d.h. dem kleinsten `priority` Wert befindet sich also im Kopf der Liste. Wenn die

Warteschlange leer, ist können wir kein kleinstes Element entfernen und zurückgeben, das Ergebnis ist also `(None, xs)` mit der leeren Warteschlange `xs`. Wenn die Warteschlange nicht leer ist, geben wir das Kopfelement `y` als `Some y` und die Warteschlange ohne das Kopfelement `ys` zurück.

d) Schreiben Sie eine Funktion `merge<'a>: PQ<'a> -> PQ<'a> -> PQ<'a>`, welche zwei Warteschlangen als Argumente erwartet und als Ergebnis eine Warteschlange zurückgibt, welche die Einträge aus beiden Warteschlangen enthält.

Hinweis: Beachten Sie, dass die resultierende Warteschlange ebenfalls bezüglich `priority` sortiert sein muss. Es genügt also nicht die Warteschlangen einfach „aneinanderzuhängen“.

```
let rec merge<'a> (xs: PQ<'a>) (ys: PQ<'a>): PQ<'a> =
    match (xs, ys) with
    | ([], zs) | (zs, []) -> zs
    | (x::xss, y::yss) ->
        if x.priority < y.priority then x::(merge xss ys)
        else y::(merge xs yss)

// alternativ mit insert
let rec merge2<'a> (xs: PQ<'a>) (ys: PQ<'a>): PQ<'a> =
    match xs with
    | [] -> ys
    | x::xss -> insert x (merge2 xss ys)
```

Die beiden Prioritätswarteschlangen verschmelzen wir, indem wir die Kopfelemente beider Listen vergleichen und das bezüglich `priority` kleinere Element vorne im Ergebnis anstellen. Dann machen wir rekursiv weiter bis eine der Listen leer ist. Wir nutzen bei der Implementierung aus, dass beide Prioritätswarteschlangen bereits sortiert sind. Beachten Sie, dass das größere der beiden Kopfelemente auch mit in den rekursiven Aufruf aufgenommen werden muss, so würden `x:y:merge(xss yss)` bzw. `y:x:merge(xss yss)` im Allgemeinen nicht zum richtigen Ergebnis führen (`exQueue1` und `exQueue2` sind so konstruiert, dass Sie damit das Problem nachvollziehen können).

Alternativ ist es auch möglich rekursiv `insert` aufzurufen.

e) *Freiwillige Zusatzaufgabe* Schreiben Sie eine Funktion `deleteNth<'a>: Nat -> PQ<'a> -> PQ<'a>`, welche eine natürliche Zahl `n` und eine Prioritätswarteschlange `pq` nimmt und das `n`-te Element (bei `0` beginnend) aus `pq` löscht. Hat `pq` weniger als `n-1` Elemente, wird `pq` unverändert zurückgegeben.

```
let rec deleteNth<'a> (n: Nat) (xs: PQ<'a>): PQ<'a> =
    if n = 0N then
        match xs with
        | [] -> []
        | _::ys -> ys
    else
        match xs with
        | [] -> []
        | y::ys -> y::(deleteNth (n-1N) ys)
```

Wir gehen nach dem Peano-Entwurfsmuster vor: Ist `n = 0`, löschen wir das Kopfelement, falls die Warteschlange nicht leer ist. Ist `n > 0`, und die Liste nicht-leer, so löschen wir das `n-1`-te Element aus der Restliste und fügen das Kopfelement vorne an.

Aufgabe 3 Ausdrücke vereinfachen (Einreichaufgabe, 3 Punkte)

Motivation: Wir haben in den Klausuren die Erfahrung gemacht, dass Studierende häufig unnötig komplexe Ausdrücke schreiben. Einerseits vermindert ein solch komplexer Ausdruck die Lesbarkeit, andererseits kostet er wertvolle Zeit beim Aufschreiben. Daher wollen wir zu diesem frühen Zeitpunkt schon einüben, wie man gängige Ausdrücke möglichst kurz darstellen kann.

Schreiben Sie Ihre Lösungen in die Datei `Simplify.fs` *aus der Vorlage* `Aufgabe-5-3.zip`.

Geben Sie für die folgenden Ausdrücke jeweils einen vereinfachten (also möglichst kurzen) Ausdruck an, der auf jeden Fall zu demselben Wert wie der ursprüngliche Ausdruck auswertet.

Beispiel: `false = (a = true)` lässt sich vereinfachen zu `not a`.

a) `if a then b else false`

```
a && b
```

b) `if (a = true) then 2N else 3N`

```
if a then 2N else 3N
```

c) `if (x <> 0N) then false else true`

```
x = 0N
```

Aufgabe 4 Balanciertes Ternärsystem (Trainingsaufgabe)

Motivation: In dieser Aufgabe sollen Sie mit Listen und Variantentypen arbeiten. Sie können sich an den Vorlesungsfolien 306 bis 331 und 379 bis 401 bzw. am Skript Kapitel 4.2 und 4.3 orientieren.

Schreiben Sie Ihre Lösungen in die Datei `Ternary.fs` *aus der Vorlage* `Aufgabe-5-4.zip`.

Zur Zahlendarstellung verwenden wir in dieser Aufgabe ein balanciertes ternäres Stellenwertsystem, welches wir mit Hilfe von Listen modellieren. Als Ziffern werden `M` („minus eins“), `Z` („zero“ bzw. „null“) und `P` („plus eins“) verwendet.

```
type Ternary = | M | Z | P // -1, 0, 1
```

Wir legen fest, dass die niederwertigste Ziffer vorne in der Liste steht und entsprechend die höchstwertige Ziffer am Ende der Liste. Damit repräsentiert die Liste `[M; Z; M; P]` die Zahl $(-1)\cdot 3^0+0\cdot 3^1+(-1)\cdot 3^2+1\cdot 3^3 = 17$. Wir brauchen im balancierten Ternärsystem kein Vorzeichen, um negative Zahlen darzustellen.

Beachten Sie, dass die Darstellung einer Zahl aufgrund von führenden Nullen nicht eindeutig ist. Teilaufgabe b) stellt eine Hilfsfunktion bereit, mit der Sie das Problem in den darauffolgenden Teilaufgaben umgehen können.

Weitere Beispiele:

```
[M; P; M] // -7     [M; M] // -4      [M] // -1          [M; P] // 2       [M; M; P] // 5
[Z; P; M] // -6     [Z; M] // -3      []  //  0          [Z; P] // 3       [Z; M; P] // 6
[P; P; M] // -5     [P; M] // -2      [P] //  1          [P; P] // 4       [P; M; P] // 7
```

Hinweis: Wir verwenden in dieser Aufgabe den Typ `Int` der ganzen Zahlen. Zahl-Literale dieses Typs haben keinen `N`-Suffix, die Zahl 42 ist also einfach `42` und -42 ist `-42`.

a) Schreiben Sie eine Funktion `bedeutung: List<Ternary> -> Int`, die für eine gegebene Repräsentation im ternären Stellenwertsystem die entsprechende ganze Zahl berechnet.

```
let rec bedeutung (n: List<Ternary>): Int =
    match n with
    | [] -> 0
    | M::ns -> 3 * bedeutung ns - 1
    | Z::ns -> 3 * bedeutung ns // + 0
    | P::ns -> 3 * bedeutung ns + 1
```

b) Implementieren Sie die Funktion `zCons` (einen „smarten Konstruktor“), die eine Null (`Z`) an eine Zahl im balancierten Ternärsystem anhängt, sofern deren Darstellung nicht der leeren Liste entspricht. Verwenden Sie diesen smarten Konstruktor in den folgenden Teilaufgaben, sofern Sie ein `Z` an eine Zahl im balancierten Ternärsystem anfügen möchten.

Hinweis: Es genügt, wenn mit `zCons` nur der Fall behandelt wird, dass die übergebene Liste leer ist. Sie müssen nicht prüfen, ob es weitere führende Nullen gibt. Damit ist z. B. `zCons [Z] = [Z; Z]`. Allerdings tritt dieser Fall nicht auf, wenn statt `Z::` stets `zCons` verwendet wird. Für oben genanntes Beispiel erhalten wir also mit `zCons (zCons []) = []` das erwartete Ergebnis.

```
let zCons (ns: List<Ternary>): List<Ternary> =
    match ns with
    | [] -> []
    | _ -> Z::ns
```

c) Schreiben Sie eine Funktion `inc`, die eine Zahl im balancierten Ternärsystem um den Wert eins erhöht.

```
let rec inc (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> [P]
    | M::ns -> zCons ns
    | Z::ns -> P::ns
    | P::ns -> M::(inc ns)
```

d) Schreiben Sie eine Funktion `dec`, die eine Zahl im balancierten Ternärsystem um den Wert eins verringert.

```
let rec dec (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> [M]
    | M::ns -> P::(dec ns)
    | Z::ns -> M::ns
    | P::ns -> zCons ns
```

e) Schreiben Sie eine Funktion `fromInt: Int -> List<Ternary>`, die eine ganze Zahl ins balancierte Ternärsystem überführt. Orientieren Sie sich am Leibniz Entwurfsmuster.

```
let rec fromInt (n: Int): List<Ternary> =
    if n = 0 then []
    else if n % 3 =  2 then M::(inc (fromInt (n/3)))
    else if n % 3 =  1 then P::(fromInt (n/3))
    else if n % 3 = -1 then M::(fromInt (n/3))
    else if n % 3 = -2 then P::(dec (fromInt (n/3)))
    else (* n % 3 =  0 *)   zCons (fromInt (n/3))
```

f) Schreiben Sie eine Funktion `add: List<Ternary> -> List<Ternary> -> List<Ternary>`, die zwei Zahlen im balancierten Ternärsystem addiert.

```
let rec add (m: List<Ternary>) (n: List<Ternary>): List<Ternary> =
    match (m, n) with
    | ([], x) | (x, []) -> x
    | (M::ms, M::ns) -> P :: (add (dec ms) ns)
    | (P::ms, P::ns) -> M :: (add (inc ms) ns)
    | (M::ms, Z::ns) | (Z::ms, M::ns)                  -> M :: (add ms ns)
    | (M::ms, P::ns) | (P::ms, M::ns) | (Z::ms, Z::ns) -> zCons (add ms ns)
    | (P::ms, Z::ns) | (Z::ms, P::ns)                  -> P :: (add ms ns)
```

g) Schreiben Sie eine Funktion `negative: List<Ternary> -> List<Ternary>`, die das Vorzeichen einer Zahl im balancierten Ternärsystem umkehrt.

```
let rec negative (n: List<Ternary>): List<Ternary> =
    match n with
    | [] -> []
    | M::ns -> P::(negative ns)
    | Z::ns -> zCons (negative ns)
    | P::ns -> M::(negative ns)
```