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

**RPTU**
**Fachbereich Informatik**
**AG Programmiersprachen**

# Lösungshinweise/-vorschläge zum Übungsblatt 5: Grundlagen der Programmierung (WS 2024/25)

**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: `List<'a>`, `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: List<'a>): 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 378 bis 403 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 Parametrische Listen (Einreichaufgabe, 12 Punkte)

*Motivation:* In dieser Aufgabe sollen Sie sich mit den in F# eingebauten parametrischen Listen vertraut machen. Sie können sich an den Vorlesungsfolien 378 bis 403 sowie am Skript Kapitel 4.3 orientieren. Dies ist die Fortsetzung von Präsenzaufgabe 1.

Sie dürfen in vorherigen Aufgabenteilen definierte Funktionen verwenden.

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

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

a) Schreiben Sie eine Funktion `minAndMax<'a>: List<'a> -> Option<'a * 'a>`, die eine Liste `xs` nimmt und das Minimum und Maximum der Liste zurückgibt. Wenn die Liste leer ist, soll `None` zurückgegeben werden.

Beispiele:

```
minAndMax [] = None                              minAndMax [4N; 3N; 2N; 1N] = Some (1N, 4N)
minAndMax [1N; 2N; 3N; 4N] = Some (1N, 4N)       minAndMax [4N; 7N; 1N; 1N] = Some (1N, 7N)
```

```
let rec minAndMax<'a when 'a: comparison>(xs: List<'a>): Option<'a * 'a> =
    match xs with
    | [] -> None
    | x::xs ->
        match minAndMax xs with
        | None -> Some(x, x)
        | Some(low, high) -> Some(min x low, max x high)
```

b) Schreiben Sie eine Funktion `map<'a, 'b>: ('a -> 'b) -> List<'a> -> List<'b>`, die eine Funktion `f` und eine Liste `xs` nimmt und die Liste der Elemente zurückgibt, die durch Anwenden von `f` auf die Elemente von `xs` entstehen.

Beispiele:

```
map (fun x -> x + 1N) [] = []
map (fun x -> x + 1N) [1N; 2N; 3N] = [2N; 3N; 4N]
map (fun x -> x * 2N) [1N; 2N; 3N] = [2N; 4N; 6N]
map (fun x -> x * x) [4N; 7N; 1N; 1N] = [16N; 49N; 1N; 1N]
```

```
let rec map<'a, 'b> (f: 'a -> 'b) (xs: List<'a>): List<'b> =
    match xs with
    | [] -> []
    | x::xs -> f x :: map f xs
```

c) Schreiben Sie eine Funktion `duplicate<'a>: List<'a> -> List<'a>`, die eine Liste `xs` nimmt und die Liste zurückgibt, in der jedes Element von `xs` zweimal vorkommt.

Beispiele:

```
duplicate [] = []
duplicate [1N; 2N; 3N] = [1N; 1N; 2N; 2N; 3N; 3N]
duplicate [4N; 7N; 1N; 1N] = [4N; 4N; 7N; 7N; 1N; 1N; 1N; 1N]
```

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

d) Schreiben Sie eine Funktion `collect<'a, 'b>: ('a -> List<'b>) -> List<'a> -> List<'b>`, die eine Funktion `f` und eine Liste `xs` nimmt und die Liste der Elemente zurückgibt, die durch Anwenden von `f` auf die Elemente von `xs` entstehen.

Beispiele:

```
collect (fun x -> [x + 1N; x + 2N]) [] = []
collect (fun x -> [x + 1N; x + 2N]) [1N; 2N; 3N] = [2N; 3N; 3N; 4N; 4N; 5N]
collect (fun x -> [x * 2N; x * 3N]) [4N; 7N; 1N; 1N] = [8N; 12N; 14N; 21N; 2N; 3N; 2N; 3N]
```

```
// Hilfsfunktion um eine Liste von Listen zu verketten
let rec concatAll<'a> (xs: List<List<'a>>) : List<'a> =
    match xs with
    | [] -> []
    | x :: xs -> concat x (concatAll xs)

let rec collect<'a, 'b> (f: 'a -> List<'b>) (xs: List<'a>): List<'b> =
    concatAll (map f xs)
```

e) Schreiben Sie eine Funktion `intersperse<'a>: 'a -> List<'a> -> List<'a>`, die ein Element `sep` und eine Liste `xs` nimmt und die Liste zurückgibt, in der die Elemente aus `xs` durch `sep` getrennt sind.

Beispiele:

```
intersperse 0N [] = []
intersperse 0N [2N] = [2N]
intersperse 0N [1N; 2N; 3N; 4N]
            = [1N; 0N; 2N; 0N; 3N; 0N; 4N]
```

```
let rec intersperse<'a> (sep: 'a) (xs: List<'a>): List<'a> =
    match xs with
    | [] -> []
    | one & [_] -> one
    | x :: xs -> x :: sep :: intersperse sep xs
```

f) Schreiben Sie eine Funktion `runs<'a>: List<'a> -> List<List<'a>>`, die eine Liste `xs` nimmt und diese in Teillisten maximaler Länger aufteilt, sodass die Teillisten jeweils aufsteigend sortiert sind.

Beispiele:

```
runs [] = []
runs [1N; 2N; 3N] = [[1N; 2N; 3N]]
runs [3N; 2N; 1N] = [[3N]; [2N]; [1N]]
runs [4N; 7N; 1N; 1N] = [[4N; 7N]; [1N; 1N]]
```

```
let rec runs<'a when 'a: comparison> (xs: List<'a>) : List<List<'a>> =
    match xs with
    | [] -> []
    | x :: xs ->
        match runs xs with
        | rest & ((current & (y :: _)) :: ys) ->
            if x <= y then (x :: current) :: ys else [ x ] :: rest
        | rest -> [ x ] :: rest
```

## Aufgabe 3 Dynamische Semantik mit Rekursion (Einreichaufgabe, 8 Punkte)

*Motivation:* Wie bereits in Aufgabe 5 auf dem letzten Übungsblatt, sollen Sie Regeln der Dynamischen Semantik rückwärts anwenden. Zudem sollen Sie sich mit Rekursion auseinandersetzen. Sie benötigen die Regeln der Vorlesungsfolien 110, 144 und 196.

*Hinweis:* Wenn Sie in Übungsgruppe 1, 3, 5, 7, 9 oder 11 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Beweisbaum Werkzeug zur Verfügung, siehe Hinweis auf der ersten Seite von Übungsblatt 4. Sie müssen das Sprachfeature "Wertedefinitionen & Funktionen" aktivieren. Nach dem Einloggen im ExClaim-System führt dieser Link direkt zum vorbereiteten Baum für Teilaufgabe a) und dieser Link zu dem für Teilaufgabe c). Wenn Sie möchten, können Sie einen Screenshot des erstellten Beweisbaumes in Ihre Abgabe mit einbinden. Falls Sie bereits Erfahrung mit LaTeX haben, dürfen Sie die entsprechende Exportfunktion nutzen und eine kompilierte pdf-Datei abgeben. Der LaTeX-Quellcode oder die durch den Button "Baum speichern" erzeugte json-Datei wird jedoch nicht als Abgabe anerkannt! Alternativ können Sie die Abgabe wie gewohnt auf Papier aufschreiben.

a) Finden Sie einen Ausdruck $e$, der gemäß den Regeln der Dynamischen Semantik aus der Vorlesung zu folgendem Wert auswertet:

$$\langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle$$

Geben Sie einen Beweisbaum mit den Regeln der Dynamischen Semantik an, der

$$\emptyset \vdash e \Downarrow \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle$$

zeigt.

Ausdruck $e :=$ `let f = 5 in let rec f x = f x in f` Link zum Baum

$$\dfrac{\dfrac{}{\emptyset \vdash \mathtt{5} \Downarrow \mathtt{5}} \quad \dfrac{\dfrac{}{\{\mathtt{f} \mapsto 5\} \vdash \mathtt{let\ rec\ f\ x = f\ x} \Downarrow \{\mathtt{f} \mapsto \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle\}} \quad \dfrac{}{\{\mathtt{f} \mapsto \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle\} \vdash \mathtt{f} \Downarrow \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle}}{\emptyset \vdash \mathtt{let\ f = 5} \Downarrow \{\mathtt{f} \mapsto 5\}} \quad \dfrac{\{\mathtt{f} \mapsto 5\} \vdash \mathtt{let\ rec\ f\ x = f\ x\ in\ f} \Downarrow \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle}{\emptyset \vdash \mathtt{let\ f = 5\ in\ let\ rec\ f\ x = f\ x\ in\ f} \Downarrow \langle \{\mathtt{f} \mapsto 5\}, \mathtt{f}, \mathtt{x}, \mathtt{f\ x} \rangle}$$

*Hinweis:* Es sind auch andere Ausdrücke möglich, die `5` lässt sich beispielsweise durch `2 + 3` ersetzen, dann werden die Bäume jedoch größer.

b) Ist Ihr Ausdruck $e$ typkorrekt, d.h. gibt es einen Beweisbaum für die Statische Semantikmit, der

$$\emptyset \vdash e : \ldots$$

zeigt? **Sie müssen den Beweisbaum nicht abgeben!** "Ja" oder "Nein" genügt.

Ja, unser Ausdruck $e$ ist typkorrekt. Das lässt sich mit dem F#-Interpreter leicht überprüfen. Dieser verrät uns auch, dass es sich um eine polymorphe Funktion vom Typ $'a \to 'b$ handelt. Wenn wir den Beweisbaum aufbauen, dann bleiben die Metavariablen $t_0$ und $t_1$ stehen, es ergibt sich der Typ $t_0 \to t_1$.

Hier ist der (nicht verlangte) Beweisbaum: Link zum Baum

$$\dfrac{\dfrac{}{\emptyset \vdash \texttt{5} : \text{Nat}}}{\emptyset \vdash \texttt{let f = 5} : \{f \mapsto \text{Nat}\}} \quad \dfrac{\dfrac{\dfrac{}{\{f \mapsto t_0 \to t_1, x \mapsto t_0\} \vdash \texttt{f} : t_0 \to t_1} \quad \dfrac{}{\{f \mapsto t_0 \to t_1, x \mapsto t_0\} \vdash \texttt{x} : t_0}}{\{f \mapsto t_0 \to t_1, x \mapsto t_0\} \vdash \texttt{f x} : t_1}}{\{f \mapsto \text{Nat}\} \vdash \texttt{let rec f (x:} \ t_0\texttt{):} \ t_1 \ \texttt{= f x} : \{f \mapsto t_0 \to t_1\}} \quad \dfrac{}{\{f \mapsto t_0 \to t_1\} \vdash \texttt{f} : t_0 \to t_1}$$

$$\dfrac{\emptyset \vdash \texttt{let f = 5} : \{f \mapsto \text{Nat}\} \qquad \{f \mapsto \text{Nat}\} \vdash \texttt{let rec f (x:} \ t_0\texttt{):} \ t_1 \ \texttt{= f x in f} : t_0 \to t_1}{\emptyset \vdash \texttt{let f = 5 in let rec f (x:} \ t_0\texttt{):} \ t_1 \ \texttt{= f x in f} : t_0 \to t_1}$$

c) Nun wollen wir den Wert aus Teilaufgabe a) auf der linken Seite der Semantikregeln verwenden. Betrachten Sie folgende Aussage der Dynamischen Semantik:

$$\{\texttt{f} \mapsto \langle \{\texttt{f} \mapsto 5\}, \texttt{f}, \texttt{x}, \texttt{f x} \rangle\} \vdash \texttt{f 0} \Downarrow \ldots$$

Versuchen Sie, einen Beweisbaum zu erstellen. Auf welches Problem stoßen Sie? Geben Sie einen unvollständigen Beweisbaum sowie eine Erläuterung Ihres Problems ab.

Definiere aus Platzgründen: $\delta_1 := \{\texttt{f} \mapsto \langle \{\texttt{f} \mapsto 5\}, \texttt{f}, \texttt{x}, \texttt{f x} \rangle\}$ und $\delta_2 := \{\texttt{f} \mapsto \langle \{\texttt{f} \mapsto 5\}, \texttt{f}, \texttt{x}, \texttt{f x} \rangle, \texttt{x} \mapsto 0\}$ .

Der unfertige Baum sieht wie folgt aus: Link zum Baum

$$\dfrac{\dfrac{}{\delta_1 \vdash \texttt{f} \Downarrow \langle \{\texttt{f} \mapsto 5\}, \texttt{f}, \texttt{x}, \texttt{f x} \rangle} \quad \dfrac{}{\delta_1 \vdash \texttt{0} \Downarrow 0} \quad \dfrac{\dfrac{}{\delta_2 \vdash \texttt{f} \Downarrow \langle \{\texttt{f} \mapsto 5\}, \texttt{f}, \texttt{x}, \texttt{f x} \rangle} \quad \dfrac{}{\delta_2 \vdash \texttt{x} \Downarrow 0} \quad \delta_2 \vdash \texttt{f x} \Downarrow v_0}{\delta_2 \vdash \texttt{f x} \Downarrow v_0}}{\delta_1 \vdash \texttt{f 0} \Downarrow v_0}$$

Das offene Beweisziel $\delta_2 \vdash \texttt{f x} \Downarrow v_0$ oben rechts kam die Ebene darunter schon einmal vor. Dieser Schritt wird sich also endlos wiederholen, wir können nacheinander immer wieder die Regel für rekursive Funktionsapplikation und zwei mal die Regel für einen Bezeichner anwenden. So wird der Baum nie fertig.

Grund dafür ist, dass die Funktion sich rekursiv mit unverändertem Argument aufruft.

## Aufgabe 4 Zombieapokalypse (Einreichaufgabe, 6 Punkte)

*Motivation:* Als Teil einer vorlesungsbegleitenden Studie wollen wir untersuchen, welches mentale Modell von Rekursion die Studierenden zu verschiedenen Zeitpunkten im Verlauf des Semesters haben. Bitte bearbeiten Sie diese Aufgabe daher gewissenhaft und ohne fremde Hilfe.

*Hinweis:* Wenn Sie in Übungsgruppe 2, 4, 6, 8 oder 10 sind, dann steht Ihnen zur Bearbeitung dieser Aufgabe unser Rekursions-Tutor zur Verfügung, siehe Hinweis auf der ersten Seite von Übungsblatt 4.

Betrachten Sie die folgende Typdefinition:

```
type Zombies =
  | KeinZombie
  | Zombie of Nat * Zombies
```

a) Betrachten Sie nun die nachfolgende Funktion. Der erste Parameter soll eine Person durch eine eindeutige ID als natürliche Zahl beschreiben.

```
let rec zombify (p: Nat) (z: Zombies): Zombies =
  match z with
  | KeinZombie     -> Zombie (p, z)
  | Zombie (q, zs) -> if p = q then z
                      else Zombie (q, zombify p zs)
```

Beschreiben Sie in Ihren Worten, was die Funktion `zombify (p: Nat) (z: Zombies)` macht.

> Die Funktion `zombify` erweitert die Liste von zombifizierten Personen, die durch `Zombies` modelliert wird, um die Person `p`, sofern sie nicht bereits in der Liste stehen (keine Duplikate möglich).

b) Betrachten Sie nun die folgende Typdefinition, die eine Sequenz von Bissen modellieren soll.

```
type Bisse =
  | KeinBiss
  | Biss of Nat * Bisse
```

Beispiel: Die folgende Sequenz vom Typ `Bisse` bedeutet, dass Person 1 gebissen wurde, die dann Person 2 gebissen hat und die dann Person 3 gebissen hat, der niemanden mehr gebissen hat.

```
Biss (1, Biss (2, Biss (3, KeinBiss)))
```

Um während einer Zombieapokalypse überwachen zu können, wer bereits zombifiziert wurde, soll eine Funktion definiert werden, die, gegeben eine Sequenz von Bissen, alle Personen in dieser Sequenz zombifiziert. Betrachten Sie nun abschließend die folgenden drei Versuche, eine Lösung für dieses Problem zu definieren.

Schauen Sie sich jede Funktion an und beantworten Sie jeweils kurz: (A) Hat die jeweilige Funktion das beschriebene Verhalten? (B) Falls ja, wie funktioniert die Funktion? Falls nein, wieso funktioniert die Funktion nicht?

```
let rec solution1 (bs: Bisse) (z: Zombies): Zombies =
    match bs with
    | KeinBiss -> z
    | Biss (p, ps) -> solution1 ps (zombify p z)

let rec solution2 (bs: Bisse) (z: Zombies): Zombies =
    match bs with
    | KeinBiss -> z
    | Biss (p, ps) -> let zz = (zombify p z) in solution2 ps z

let rec solution3 (bs: Bisse) (z: Zombies): Zombies =
    match bs with
    | KeinBiss -> z
    | Biss (p, ps) -> zombify p (solution3 ps z)
```

Sowohl `solution1` als auch `solution3` haben das beschriebene Verhalten. Der einzige Unterschied ist die Reihenfolge, in der die Personen zombifiziert werden. In `solution1` wird die erste Person zombifiziert und dann bereits die aktualisierte Liste dem rekursiven Aufruf übergeben. In `solution3` wird erst der rekursive Abstieg bis zur letzten Person gemacht, und dann von innen nach außen bzw. von hinten nach vorne zombifiziert. Für das Endergebnis macht die Reihenfolge keinen Unterschied. `solution2` führt nicht zum richtigen Ergebnis, weil im rekursiven Aufruf stets die unveränderte Liste übergeben wird.

## Aufgabe 5 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-5-5.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 6 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-5-6.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)
```