Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Weihnachtsübungsblatt

Hinweis: Dieses Zusatzübungsblatt wird von den Tutoren der Vorlesung "Programmierung 1" (WS 2012/13) erstellt. Es enthält unter anderem Aufgaben von den Übungsblättern der gleichen Vorlesung aus dem Wintersemester 2011/12 gehalten von Prof. Hermanns, sowie des Nachklausurtutoriums aus dem gleichen Jahr. Außerdem enthält es einige neue, von den Tutoren erstellte Aufgaben. *Die hier gestellten Aufgaben und die damit abgedeckten Themenbereiche sind weder für die Klausur relevant, noch irrelevant. Sie dienen vor allem dazu Ihnen andere Blickrichtungen auf den Stoff zu präsentieren. Ferner sollen sie als Möglichkeit dienen, den in der ersten Semesterhälfte gelernten Stoff über die Weihnachtsferien zu wiederholen.*

Aufgabe CH.1 (*Verteufelte Baumfaltung*)
Dieter Schlau hat eine Prozedur geschrieben, die zählen soll, wie oft ein Baum t in einem Baum $T\,ts$ vorkommt.

```
fun  count s (T ts)  = fold (fn cs => if s = T ts then 1 else (foldl op+ 0 cs)) (T ts)
```

Dabei ist etwas schief gegangen. Können Sie herausfinden was?

Aufgabe CH.2
Ein Quine ist ein Programm, das haargenau seinen eigenen Quellcode ausgibt (inklusive aller Zeilenumbrüche). Schreiben Sie ein Quine in SML! Verwenden Sie dabei keine dunkle Magie (wie zum Beispiel obskure Bibliotheken...)! Sie dürfen die Prozedur $print : string \to unit$ verwenden. Falls Sie diese Prozedur nicht kennen: Probieren Sie zunächst aus, was Sie tut!

Aufgabe CH.3 (*In-Order Teilbaumliste*)
Sie haben bereits Prä- und Postordnung auf Bäumen kennengelernt. Nun wollen wir auf Binärbäumen noch eine weitere Ordnung realisieren: Die In-Order. Dazu wollen wir die Liste aller Teilbäume eines Baumes in In-Order berechnen. Bei der In-Order werden zunächst die Knoten des linken Unterbaums, dann die Wurzel und zuletzt die Knoten des rechten Unterbaums besucht. Z.B. hätte der Baum $T[T[T[],T[]],T[]]$ folgende In-Order-Teilbaumliste: $[T[],T[T[],T[]],T[],T[T[T[],T[]],T[]],T[]$. Implementieren Sie nun eine Prozedur *iost*, die die Teilbäume eines Baumes in In-Order berechnet. Falls der Baum kein Binärbaum ist, soll die Ausnahme Subscript geworfen werden.

Aufgabe CH.4
Sie wollen sich überlegen, wie Sie ein Haus möglichst schön, aber mit wenig Aufwand weihnachtlich dekorieren können. Dazu implementieren Sie sich schnell ein kleines Modell in SML.

(a) Ein Haus hat Fenster, Türen, Wände, ein Dach und unter Umständen mehrere Stockwerke. Deklarieren Sie einen Datentyp *haus*, der all diese Umstände berücksichtigt.
Hinweis: Es kann hilfreich sein, mehrere Datentypen anzulegen, z.B. einen Datentyp für Fenster, einen für Türen, einen für Wände und so nach und nach die einzelenen Komponenten des Hauses zusammenzusetzen.

(b) Deklarieren Sie nun einen Datentyp *dekoration*. Fenster und Türen kann man mit einfachen Lichterketten und Fensterbildern dekorieren, an der Tür kann ein Mistelzweig hängen, auf dem Dach kann ein Rentierschlitten stehen oder können Lichterketten, die auch für außen geeignet sind, angebracht sein. Sie können den Datentypen natürlich auch beliebig erweitern. Machen Sie sich nur jetzt schon bereits klar, wo Ihre Dekoration angebracht werden soll.

(c) Erweitern Sie nun Ihren Datentyp *haus*, damit ihr Haus weihnachtlich dekoriert werden kann. Überlegen Sie sich, wie Sie sicherstellen, dass mehr als nur eine Dekoration pro Fenster oder Tür möglich ist.

(d) Was fehlt nun noch für ein weihnachtliches Haus? Richtig, der Schnee. Erweitern Sie nun also Ihren Datentypen erneut, so dass auf dem Dach Schnee liegen kann.

(e) Schreiben Sie nun eine Prozedur $richtigDekoriert\colon haus \to bool$, die testet, dass nur Lichterketten am Dach angebracht wurden, die auch für draußen geeignet sind und die überprüft, ob die Dekorationen nur an Ihrem dafür vorgesehenen Platz sind.

(f) Schreiben Sie schließlich eine Prozedur $schneien\colon haus \to haus$, die es auf ein Haus schneien lässt und eine Ausnahme wirft, falls die Dekoration falsch angebracht wurde.

Aufgabe CH.5 (*Plätzchenbacken mit Konstruktoren*)
Seien die Datentypen $form$ und $teig$ wie folgt gegeben.

```
datatype form = KREIS | QUADRAT | HERZ

datatype teig = T of int
```

wobei die ganze Zahl des Teigs anzeigt, für wie viele Plätzchen der Teig reicht. Ignorieren Sie, dass man für verschiedene Plätzchen eigentlich verschiedene Teigsorten benötigt.

- Deklarieren Sie einen Datentyp *gebaeck*, mit dem man Plätzchen und Zimtwaffeln darstellen kannst. Plätzchen müssen ihre Form beinhalten, Zimtwaffeln brauchen das nicht (da sie immer quadratisch sind).
- Deklarieren Sie eine Prozedur $zimtwaffeleisen : teig \to teig * gebaeck\ list$, die einen Teig bekommt und eine Liste mit 8 Zimtwaffeln zurückgibt, zusammen mit dem übrig gebliebenen Teig (d.h. wenn der Eingabeteig für 10 Zimtwaffeln reicht soll der Rückgabeteig für 2 Zimtwaffeln reichen). Reicht der Teig nicht mehr für 8 Zimtwaffeln, so sollen nur so viele Zimtwaffeln zurückgegeben werden wie Teig da ist, zusammen mit dem jetzt leeren Teig, also $T\,0$.
- Wir können das Ausstechen der Plätzchen durch Prozeduren vom Typ $teig \to teig * gebaeck$ darstellen. Hier soll der zurückgegebene Teig für ein Plätzchen weniger reichen als der Eingabeteig. Falls der Eingabeteig nicht mehr für ein Plätzchen reicht soll eine NotEnoughDough

Exception geworfen werden. Deklarieren Sie diese vorher entsprechend. Definiere einen Typ $plaetzchenform$ das Ausstechen der Plätzchen auf diese Art darstellt.

Hinweis: Sie sollen hier das Schlüsselwort *type* verwenden, nicht *datatype*.

- Deklarieren Sie eine Prozedur $makeform : form \to plaetzchenform$, die zu einer gegebenen Form eine Plätzchenform (gemäß des obigen Typen) zurückgibt.
- Deklarieren Sie eine Prozedur $bake : teig \to form\ option \to teig * gebaeck\ list$, die ein Gebäck der entsprechenden Form herstellt oder bis zu 8 Zimtwaffeln backt falls die übergebene Form $NONE$ ist. Sollte während des Backens der Teig ausgehen (es werden also z.B. nur 5 Zimtwaffeln gebacken), so soll die NotEnoughDough Exception geworfen werden. Bleibt Teig übrig, so soll dieser zurückgegeben werden.
- Und nun zum krönenden Abschluss: Deklarieren Sie eine Prozedur $bakeAutonomous : teig \to form\ option\ list \to teig * gebaeck\ list$ die (unter Verwendung von *bake*) entsprechend der $form\,option\,list$ Gebäck produziert und zurückgibt. Falls während dem Backen der Teig ausgeht soll automatisch neuer Teig nachbestellt werden. Das wir dadurch simuliert dass einfach mit val newdough = T of 20 immer neuer Teig zur Verfügung steht. Es dürfen maximal 7 Einheiten Teig weggeworfen werden, d.h. Sie dürfen sich nicht einfach für jeden Backvorgang neuen Teig nehmen.

Aufgabe CH.6 (*Datentypen und Exceptions*)
Eine Schweizer Bank bietet ihren Kunden Schließfächer und Konten an. Ein Schließfach gehört einem Kunden (Typ: *string*), ein Konto gehört einem Kunden (*string*) und enthält Euros (*int*). Den Inhalt eines Schließfachs werden wir igonieren.

(a) Deklarieren Sie einen Datentyp *item*, der ein Schließfach oder ein Konto darstellt. Deklarieren Sie dazu passend eine Prozedur $getKunde : item \to string$, die den Kunden zu einem Konto oder Schließfach liefert.

(b) Die Bank wird als $item\ list$ dargestellt, die alle Konten und Fächer enthält, die die Bank anbietet. Schreiben Sie nun folgende Prozeduren, um die Verwaltung der Bank zu erleichtern.

- $isKunde : item\,list \to string \to bool$, überprüft, ob eine Person Kunde bei der Bank ist (also ein Fach oder Konto hat).
- Die Prozedur $countFaecher : itemlist \to string \to int$ soll die Anzahl der Schließfächer eines Kunden liefern.
- $countEuros : item\,list \to string \to int$, soll die Summe des Geldes aller Konten eines Kunden liefern.

(c) Die Daten der Bank ($itemlist$) wandern auf unbekannten Wegen zum Finanzamt, wo sie von Steuerfahndern mit den Steuerdaten abgeglichen werden. Ein Steuerfahnder bekommt eine Prozedur $steuerdaten : string \to int$, die zu einer beliebigen Person den versteuerten Betrag liefert.

Deklarieren Sie eine Prozedur $steuerfahnder : (string \to int) \to itemlist \to stringlist$. Die Prozedur bekommt $steuerdaten : string \to int$ und die Bankdaten übergeben und liefert eine Liste aller Personen, die mehr Geld auf dem Konto haben als in den Steuerdaten angegeben. Sie können davon ausgehen, dass jede Person nur ein Konto hat. Wenn ein Kunde mehr als 10.000€ hinterzogen hat, werfen sie $exception\,Verhaften\,of\,string$ mit dem Namen des Kunden.

(d) Die Bank ist nicht erfreut, dass das Finanzamt die Daten ihrer Kunden nutzt und möchte deshalb ihren Datenbestand umstellen. Die Daten sollten künftig ähnlich zu einer Umgebung aufgebaut sein:

```
exception Unbound
type bankdaten = string -> item list
```

Zu jedem Kunden wird die Liste seiner Schließfächer und Konten gespeichert, falls der Kunde nicht existiert wird *Unbound* geworfen. Schreiben Sie nun die Prozeduren aus Teil *b* erneut, verwenden Sie jedoch *bankdaten* statt *item list*.

(e) Schreiben Sie eine Prozedur $convert : item\ list \to bankdaten$, die die Daten der Bank ins neue Format konvertiert.

(f) Warum ist es nicht ohne weiteres möglich, nun eine analoge Prozedur $steuerfahnder$ zu schreiben?

Aufgabe CH.7 (*Übungen mit Umgebungen und arithmetischen Ausdrücken*)

(a) Deklarieren Sie zunächst eine Prozedur $addenv : env \to var \to env$, sodass $addenv\, e\, v$ eine Umgebung zurückgibt, in welcher die Variable v an den Wert 1 gebunden ist und sonst alle Einträge aus e übernommen werden. Beispiel: Für die Umgebung $[x := 3, y := 5]$ und die Variable x soll die Prozedur $[x := 1, y := 5]$ liefern.

(b) Schreiben Sie eine Prozedur $getenv : exp \to env$, die für einen arithmetischen Ausdruck (wie in Kapitel 6.4 definiert) eine Umgebung liefert, in der alle Variablen, die in dem Ausdruck vorkommen, den Wert 1 haben. Beispiel: für den Ausdruck $(2x + y)(x + 3)$ soll *getenv* die Umgebung $[x := 1, y := 1]$ liefern. (*Hinweis*: nutzen Sie Teil a) und die Adjunktion von zwei Umgebungen, siehe Buch S. 124)

(c) Wir stellen logische Aussagen mit folgendem Datentyp dar:

```
datatype bexp = C of bool
       | V of var
       | AND of bexp * bexp
       | OR of bexp * bexp
       | NOT of bexp
```

Damit können wir die Formel $a \wedge ((\neg b) \vee true$ darstellen als `AND ( V "a", OR ( NOT ( V "b") , C true ))`. Schreiben Sie eine Prozedur $beval : bexp \to env \to bool$, die einen bool'schen Ausdruck in einer bestimmten Umgebung auswertet. (Schauen Sie sich dabei möglichst nicht das eval für arithmetische Ausdrücke im Buch an)

(d) Wir erweitern den Datentyp aus Teil c) um den Konstruktor ASN:

```
datatype bexp = C of bool
       | V of var
       | AND of bexp * bexp
       | OR of bexp * bexp
       | NOT of bexp
       | ASN of var * bool * bexp
```

Dabei bedeutet $ASN(v, b, e)$, dass die Variable v im Ausdruck e an den Wert b gebunden wird. Wir können damit also neue Variablenbindungen in einem Teilausdruck einführen. Beispiel: `AND ( V "a", ASN ( "b", false , NOT ( V "b") ))` bedeutet, dass b im Teilausdruck `NOT ( V "b")` den Wert false hat (somit ist der Ausdruck dann (a und (nicht false))).

Ihre Aufgabe ist es nun *beval* für ASN anzupassen.

Beachten Sie dabei: Mit ASN wird eventuell der Wert einer Variablen in der Umgebung überschrieben. Wenn man z.B. den Ausdruck oben in der Umgebung [a := true, b := true] auswertet, so wird der Teilausdruck `NOT ( V "b")` in der Umgebung [a := true, b := false] ausgewertet und alles andere in der Umgebung [a := true, b := true]. Sie sollten also bei Ihrer Regel für ASN im rekursiven Aufruf die Umgebung entsprechend anpassen. (*Tipp*: Bauen Sie auf Ihre frühere Arbeit auf)

Aufgabe CH.8 (*Balancierte Bäume*)
Eine der wichtigsten Anwendungen von Bäumen in der Informatik ist die Verwendung als Suchbaum (http://de.wikipedia.org/wiki/Suchbaum). Dabei hängt die Laufzeit von den einzelnen Operationen (Suchen von Elementen, Einfügen, ...) linear von der Tiefe des Baumes ab.
Eine Möglichkeit um eine bestimmte Tiefe zu garantieren, ist die folgende Forderung: Ein Baum ist balanciert, wenn **für jeden Knoten die Tiefe des linken Unterbaums höchstens um eins von der Tiefe des rechten Unterbaums abweicht**.

(a) Zum Aufwärmen: Schreiben Sie eine Prozedur $isbinary :' altr- > bool$, die überprüft, ob ein (markierter) Baum ein Binärbaum ist.

Wie könnten Sie eine Datenstruktur $BinaryTree$ deklarieren, die von vornherein nur Binärbäume akzeptiert?

(b) Deklarieren Sie nun eine Prozedur $isBalanced :' altr- > bool$, die überprüft ob ein Binärbaum balanciert ist. (Ist der angegebene Baum kein Binärbaum, so soll ebenfalls $false$ geliefert werden.)

(c) Verallgemeinern Sie die obige Definition für Bäume eines beliebigen Grades, sodass sich die Tiefen aller Unterbäume jeweils um höchstens eins unterscheiden dürfen.

(d) AVL-Bäume sind eine Beispiel-Datenstruktur für binäre Suchbäume. Für diese muss zusätzlich zu der obigen Definition der Balanciertheit die *Binäre-Suchbaum-Eigenschaft* erfüllt sein, d.h.: Die Marke eines Wurzels ist $\leq$ der Marken aller Teilbäume in seinem linken Unterbaum. Die Marke eines Wurzels ist $\geq$ der Marken aller Teilbäume in seinem rechten Unterbaum.

Schreiben Sie eine Prozedur $isBinarySearchTree$, die überprüft ob die *Binäre-Suchbaum-Eigenschaft* erfüllt ist.

(e) Führen Sie nun zusammen, was Sie wissen, und überprüfen Sie, ob ein beliebiger Baum ein AVL-Baum (d.h. binär, balanciert und ein Suchbaum) ist. Ihre Prozedur sollte keine Fehler für ungültige Bäume werfen, sondern nur $false$ liefern. Nutzen Sie dazu die Auswertungsreihenfolge von *andalso*!

Aufgabe CH.9 (*Strings und Konstrukturen*)
Die meisten Texteditoren bieten einen weichen Zeilenumbruch an. Das heißt, dass ein Texteditor eine Zeile Text umbricht, sobald ein Wort nicht mehr komplett in eine Zeile passt. Das tut er aber intelligent, denn er reißt dabei keine Wörter auseinander, sondern schreibt ein zu langes Wort komplett in die nächste Zeile.
Es sei ein Datentyp

```
datatype word = W of string * int
```

gegeben.
Der String enthält das Wort (ohne Leerzeichen) und das int enthält die Länge des Strings.
Sie können annehmen, dass die Wörter nicht länger sind als die Anzahl Zeichen, die maximal in einer Zeile erlaubt sind.

(a) Schreiben Sie zunächst eine Prozedur $findWords : string \to word\ list$, die zu einem String die Liste seiner Wörter zurck gibt. Wörter werden hier nur durch Leerzeichen voneinander getrennt. "Dieter Schlau hat Probleme." wird z.B. zur Liste:

```
[W ("Dieter", 6), W ("Schlau", 6), W ("hat", 3), W ("Probleme.", 9)]
```

Tipp: Benutzen Sie eine Hilfsprozedur und arbeiten Sie mit $char\ list$ statt strings. Als Fallunterscheidung für die Prozedur bietet sich unter anderem $(\#" " :: s)$ und $(c :: s)$ bei dem Argument, das den abzuarbeitenden String enthält.

(b) Schreiben Sie eine Prozedur $softWrap : int \to string \to string\ list$, so dass $softWrap\ n\ s$ den String s so umbricht, dass in jeder Zeile maximal n sichtbare Zeichen stehen (das Leerzeichen darf also das n+1-te Zeichen sein. Die Zeilen sollen in einer Liste zurückgegeben werden.

(c) Jetzt muss der Text nur noch ausgegeben werden. Schreiben Sie eine Prozedur $printText : int \to string \to unit$, die beim Aufruf $printText\ n\ s$ den String s auf die Konsole schreibt, und dabei jede Zeile maximal n Zeichen enthält.

Tipp: Benutzen Sie die Prozedur $print : string \to unit$.

Aufgabe CH.10 (*fold-ception*)
Im Zusammenhang mit unmarkierten Bäumen wurde in der Vorlesung $fold$ behandelt. Faltung für markierte Bäume lässt sich analog definieren:

```
fun foldltr f (L(a,xs)) = f(a, List.map (foldltr f) xs)
val foldltr : ('a * 'b list -> 'b) -> 'a ltr -> 'b
```

(a) Schauen Sie sich diese Prozedur ganz genau an und versuchen Sie, sie zu verstehen.

(b) Schreiben Sie mit $foldltr$ eine Prozedur $sum : int\ ltr \to int$, welche die Summe aller Werte eines $int\ ltr$ berechnet.

(c) Schreiben Sie eine Prozedur $mirror : stringltr \to stringltr$, die einen $stringltr$ spiegelt. Auch die Markierungen, also die Strings, sollen dabei gespiegelt werden. Gro- und Kleinschreibung darf vernachlässigt werden.

Tipp: Erinnern Sie sich zunächst an die Spiegelung von Strings.

(d) Schreiben Sie mit $foldltr$ die Prozeduren $preorder$ und $postorder : \alpha\ ltr \to \alpha\ list$, die die Markierungen das Baumes in der entsprechenden Reihenfolge ausgeben.

Tipp: $preorder$ und $postorder$ sind semantisch äquivalent zur Prä- und Postprojektion $prep$ und pop aus dem Buch.

(e) Schreiben Sie mit $foldltr$ die Prozedur $fold' : (\alpha\ list \to \alpha) \to \beta\ ltr \to \alpha$, die analog zu $fold$ auf markierten Bumen arbeitet, die Markierungen an sich aber nicht benötigt. Sind $fold$ und $fold'$ semantisch äquivalent?

(f) Schreiben Sie mit $foldltr$ die Prozedur $isheap : int\ ltr \to bool$, die überprüft, ob ein gegebener $int\ ltr$ die Heap-Eigenschaft erfüllt.

Ein Baum erfüllt genau dann eine Heap-Eigenschaft, wenn für alle seine Teilbäume gilt: Die Markierung der Wurzel ist größer oder gleich aller Markierungen seiner Unterbäume.

Sie können davon ausgehen, dass sämtliche Markierungen positiv sind.

Tipp: Tupel und Faltung

Aufgabe CH.11 (*Sortieren und Umgebungen*)
Der Nikolaus befindet sich schon seit Jahren in einem bitter-bösen Kampf mit Knecht Ruprecht, weil dieser ihm jedes Jahr Geschenke stibitzt. Da der Nikolaus schlau ist, sortiert er dieses Jahr seine Geschenke nach Wert, die teuren Geschenke sollen zuerst verteilt werden. Wenn zwei Geschenke den gleichen Wert haben, sollen die Kinder, die in dem Jahr besonders artig waren, ihre Geschenke zuerst bekommen. Um herauszufinden, wer artig war, hat der Nikolaus ein goldenes Buch, in dem zu jedem Kind vermerkt ist ob es artig war oder nicht.
Gegeben sind die folgenden Datentypen:

```
datatype artig = Artig | Boese
datatype geschenk = G of string*int
type goldenes_buch = string -> artig
```

Helfen Sie nun dem Nikolaus, seine Geschenke zu sortieren. Gehen Sie dabei wie folgt vor:

(a) Schreiben Sie eine Prozedur $gCompare : goldenes\_buch \to geschenk * geschenk \to order$, die zwei Geschenke nach den oben gegebenen Kriterien vergleicht.

(b) Der Nikolaus braucht noch zusätzlich eine Prozedur $findMostExpensive : (geschenk * geschenk \to goldenes\_buch \to order) \to goldenes\_buch \to geschenk\ list \to geschenk$, die das teuerste Geschenk in einer Liste von Geschenken findet.

(c) Mit $deleteFromList : geschenk \to geschenk\ list \to geschenk\ list$, die ein Geschenk aus einer Liste von Geschenken löscht, hat der Nikolaus dann alle Prozeduren fertig, die er zum sortieren seiner Geschenke benötigt.

(d) Die Vorgehensweise beim Sortieren der Geschenke ist jetzt die folgende: Da der Nikolaus die teuren Geschenke zuerst verteilen möchte, wird zunächst das teuerste Geschenk aus der Liste gesucht. Dieses soll nachher ganz vorne in der Liste stehen, also wird es, nach dem die Restliste sortiert ist, vorne angehängt. Hat man das teuerste Geschenk gefunden, merkt man sich dieses, und löscht es aus der Geschenkliste. Mit dem selben Vorgehen wird jetzt auch die Restliste sortiert.

(e) Bevor Sie dem Nikolaus jetzt Ihren Sortieralgorithmus vorstellen, denken Sie noch einmal über folgende Dinge nach: Warum terminiert Ihr Sortieralgorithmus? Haben Sie an alle Basisfälle gedacht? Sortiert er so, wie er laut Beschreibung sortieren soll? Wie sieht das Typschema ihres Sortieralgorithmus aus?

Aufgabe CH.12 (*Korrektes Weihnachtsfest*)
An Heiligen Abend muss der Weihnachtsmann jede Menge Geschenke verteilen und Familien besuchen.
Damit der enge Zeitplan eingehalten werden kann, muss das gesamte Vorgehen natürlich korrekt und gut geplant sein.

(a) Ohne die Liste geht natürlich garnichts! Die Liste? Natürlich die Liste der artigen und unartigen Kinder, die in der letzten Aufgabe auch schon erwähnt wurde. Aber da der Weihnachtsmann nicht binär denkt, wird ab jetzt jedem Kind eine natürliche Zahl zugeordnet. Dabei bedeutet eine hohe Zahl, dass das Kind sehr artig war, und eine niedrige als vergleichsweise weniger artig. 0 bedeutet hierbei minimal artig. Folgende mathematische Prozedur berechnet aus einer gegebenen Artigkeit die Anzahl an Geschenken, die das Kind erhält.

```
p 0 = 1

p 1 = 1

p 2 = 2

p 3 = 3

p n = f(n-4) + 2*f(n-3) + f(n-2)
```

Helfen Sie dem Weihnachtsmann, indem Sie die Ergebnisfunktion mit nur 2-facher Rekursion formulieren! Das bedeutet, dass der Rekursionsbaum ein binärer Baum ist.
Kommt Ihnen die Funktion bekannt vor?
Zeigen Sie mithilfe des Korrektheitssatzes, dass die mathematische Prozedur des Weihnachtsmannes diese Funktion berechnet.

(b) Während der Weihnachtsmann fröhlich Geschenke unter den Weihnachtsbaum einer Familie legt, steht plötzlich jemand in der Tür.
Dieter Schlau ist aufgewacht und stellt den Weihnachtsmann zur Rede.
Er sieht 2 Geschenke mit seinem Namen darauf und argumentiert:

Er habe offensichtlich ein erstes Geschenk bekommen. Außerdem habe er ein Zweites bekommen, nachdem er bereits ein Erstes hatte.
Damit könne er per natürlicher Induktion über die Anzahl der Geschenke beweisen, dass ihm beliebig viele (unendlich viele) Geschenke zustehen.

Der Weihnachtsmann, anfangs etwas verwirrt, erinnert sich jedoch kaum noch an sein lange Zeit zurückliegendes Informatikstudium.

Helfen Sie ihm: Wo liegt der Fehler in der Argumentation von Dieter Schlau? Warum kann man hier keine Induktion anwenden? Was wäre das Problem, falls man hier doch mit Induktion argumentieren könnte?

Aufgabe CH.13 (*Induktion*)

(a) Schreiben Sie eine rekursive Prozedur $sum : \mathbb{N} \to \mathbb{N}$, welche zu einer natürlichen Zahl $n \in \mathbb{N}$ die Summe $0 + 1 + 2 + \cdots + n$ berechnet.

(b) Beweisen Sie durch natürliche Induktion, dass die Funktion

$$gauss \in \mathbb{N} \to \mathbb{N}$$
$$gauss\ n = \frac{n}{2}(n+1)$$

die Ergebnisfunktion von *sum* ist.

(c) Schreiben Sie nun eine endrekursive Version von *sum* und passen Sie die Ergebnisfunktion *gauss* entsprechend an.

Beweisen Sie die Korrektheit Ihrer neuen Ergebnisfunktion durch einen induktiven Beweis.