Hinweis: Zuerst löse ich die Aufgaben aus dem Anhang, um einen Einstieg in die Materie zu finden.

Anhang I

Da nur drei Variablen auftauchen, erspare ich mir die Mühe, den Beweis über logische Schlussfolgerungen zu führen, sondern erstelle Wertetabellen. Die Grundlagen dafür bildet (es stehe 0 für falsch und 1 für wahr):

A	B	$\neg$ A	A $\wedge$ B	A $\vee$ B	A $\rightarrow$ B	A $\leftrightarrow$ B
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Die nachfolgenden Ausdrücke werden schrittweise aufgelöst, der Wert des Gesamtausdruckes steht in der grauen Spalte. Sollte er allgemeingültig sein, so dürfen nur Einsen auftauchen, wenn sich nur eine einzige Null findet, so ist er nicht allgemeingültig (rot markiert).
Um Platz zu sparen, habe ich in vorherigen Aufgaben bereits ermittelte Terme teilweise wiederverwendet.

a) $((A \rightarrow C) \wedge (B \rightarrow C)) \rightarrow ((A \wedge B) \rightarrow C)$ ist allgemeingültig:

A	B	C	A→C	B→C	(A→C)∧ (B→C)	A∧B	(A∧B)→C	((A→C)∧ (B→C))→ ((A∧B)→C)
0	0	0	1	1	1	0	1	1
0	0	1	1	1	1	0	1	1
0	1	0	1	0	0	0	1	1
0	1	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1	1
1	0	1	1	1	1	0	1	1
1	1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	1	1

b) $((A \wedge B) \rightarrow C) \rightarrow ((A \rightarrow C) \wedge (B \rightarrow C))$ ist nicht allgemeingültig:

A	B	C	(A∧B)→C	(A→C)∧ (B→C)	((A∧B)→C)→ ((A→C)∧ (B→C))
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	1	1	1

c) $(A \rightarrow B) \rightarrow \neg(B \rightarrow A)$ ist nicht allgemeingültig:

A	B	A→B	B→A	¬ (B→A)	(A→B) → ¬ (B→A)
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	0	1
1	1	1	1	0	0

d) $(A \to B) \leftrightarrow (\neg B \to \neg A)$ ist allgemeingültig:

A	B	A→B	¬B → ¬A	(A→B) ↔ (¬B → ¬A)
0	0	1	1	1
0	1	1	1	1
1	0	0	0	1
1	1	1	1	1

e) $(A \to B) \vee (B \to A)$

A	B	A→B	B → A	(A→B) ∨ (B → A)
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	1	1	1

Anhang II

Um zu zeigen, dass eine Aussage *nicht* allgemeingültig ist, reicht es aus, wenn man nur ein einziges Gegenbeispiel findet.

a) Zu untersuchende Aussage: $\forall X \exists Y : X < Y \to \exists Y \forall X : X < Y$

Eine Implikation ist genau dann falsch, wenn man aus einer wahren Aussage eine falsche schlussfolgern kann. Für A könnte man $X<Y$ einsetzen, dann ist $\forall X \exists Y$: $X<Y$ immerwahr. Die Folgerung $\exists Y \forall X$: $X<Y$ lässt sich hingegen nicht immer erfüllen, da es keine größte Zahl Y gibt. Gäbe es sie, dann müsste sie auch von $\forall X$ überdeckt werden, wodurch sich die Aussage $X<Y$ wiederum nicht erfüllen ließe.

b) Zu untersuchende Aussage: $(\forall X : A \to B) \to \forall X : (A \to B)$

Ein mögliches Gegenbeispiel besteht darin, dass die Aussage A dem booleschen Term $X>2$ entspricht, wohingegen B z.B. $X>1$ ist. Alle Zahlen, die A erfüllen, ergeben für B automatisch eine wahre Aussage.

Aufwärm-Aufgabe

a) Da man an jedem Tag arbeiten kann oder auch nicht, existieren theoretisch $2^7 = 128$ Wochenabläufe.

b) Ich kürze die Wochentage durch die ersten beiden Buchstaben ab:

Marias Arbeitsregel	Boolescher Ausdruck
Wenn ich am Samstag nicht arbeite, dann arbeite ich am Freitag	$\neg Sa \rightarrow Fr$
Wenn ich am Dienstag nicht, wohl aber am Freitag arbeite, dann arbeite ich am Donnerstag.	$(\neg Di \wedge Fr) \rightarrow Do$
Wenn es zutrifft, dass, wenn ich am Dienstag nicht arbeite, ich zwar am Montag nicht, jedoch am Donnerstag arbeite, dann arbeite ich am Samstag.	$(\neg Di \rightarrow (\neg Mo \wedge Do)) \rightarrow Sa$
Wenn ich am Dienstag arbeite, dann arbeite ich am Mittwoch nicht.	$Di \rightarrow \neg Mi$
Wenn ich am Montag arbeite, dann arbeite ich am Freitag nicht.	$Mo \rightarrow \neg Fr$
Wenn ich am Samstag arbeite, dann arbeite ich am Donnerstag nicht, wohl aber am Freitag.	$Sa \rightarrow \neg Do \wedge Fr$

c) Ein möglicher Wochenablauf ist (0,1,0,0,1,1,1), d.h. Maria arbeitet am Dienstag, Freitag, Samstag und Sonntag.

d+e) Ich halte es für notwendig, dass ich die Booleschen Ausdrücke aus Aufgabenteil b) etwas umforme (jetzt nur alleinige Verwendung von $\wedge$, $\vee$ und $\neg$):

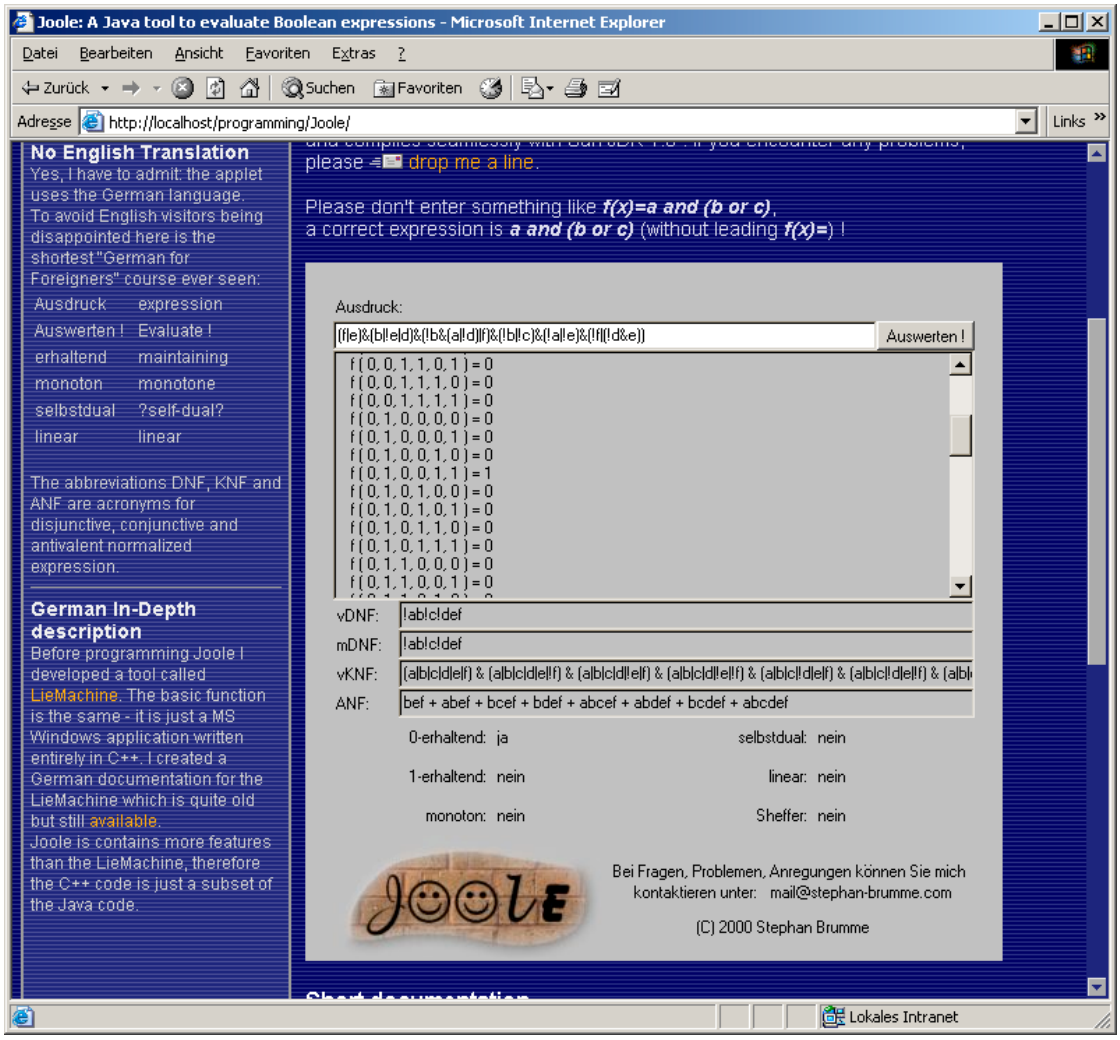
Boolescher Ausdruck	Umformung
$\neg Sa \rightarrow Fr$	$Sa \vee Fr$
$(\neg Di \wedge Fr) \rightarrow Do$	$Di \vee \neg Fr \vee Do$
$(\neg Di \rightarrow (\neg Mo \wedge Do)) \rightarrow Sa$	$\neg Di \wedge (Mo \vee \neg Do) \vee Sa$
$Di \rightarrow \neg Mi$	$\neg Di \vee \neg Mi$
$Mo \rightarrow \neg Fr$	$\neg Mo \vee \neg Fr$
$Sa \rightarrow \neg Do \wedge Fr$	$\neg Sa \vee (\neg Do \wedge Fr)$

Damit ein gültiger Wochenplan entsteht, müssen alle Aussagen erfüllt sein.
Mit Hilfe von Joole (http://www.stephan-brumme.com/programming/Joole/), einem Programm zur Auswertung logischer Ausdrücke, ist das Ergebnis recht einfach zu ermitteln. Vorher benenne alle Wochentagsnamen in die Buchstaben a bis g um (da Joole nur mit dieses arbeiten kann), sodass die Gesamtaussage in Jooles Syntax lautet:

```
(f or e) and (b or !e or d) and (!b and (a or !d) or f) and (!b or !c) and (!a or !e) and
(!f or (!d and e))
```

kam ich zu dem Ergebnis, dass genau 2 verschiedene Wochenpläne existieren.

Joole gibt zwar nur eine Lösung auf, aber in den Gleichungen taucht nirgends der Sonntag auf, er ist daher vollkommen unabhängig und kann sowohl als Arbeitstag, als auch als Ruhetag fungieren.



In der nachfolgenden Tabelle sind Arbeitstage grau hinterlegt, freie Tage (es gibt Studenten, die so etwas haben ?) bleiben weiß:

Nr.	Mo	Di	Mi	Do	Fr	Sa	So
1							
2							

Man erkennt, dass Maria nie am Montag, Mittwoch oder Donnerstag arbeitet, sich dafür aber jeden Dienstag, Freitag und Sonnabend mächtig ins Zeug legt. Am Sonntag arbeitet sie ab und zu. Das finde ich total unvernünftig, aber ich habe mir die Aufgabenstellung ja auch nicht ausgedacht.

Aufgabe 1

Es lassen sich alle drei Spezifikation mit dem Hoare-Kalkül beweisen.

a) Die Nachbedingung ist $c=9$. Nun sind alle Vorkommen von c in $x:=y$ mit 9 zu ersetzen. Da c aber überhaupt auftaucht, ist die Vorbedingung gleich der Nachbedingung:

$$
\begin{aligned}
S &:= "x := y" \\
Q &:= "c = 9" \\
P\{S\}Q &:= P\{x := y\}c = 9 \\
P &:= "c = 9" \\
&\Downarrow \\
c = 9\{x := y\}c &= 9
\end{aligned}
$$

b) Aufgrund der Nachbedingung $x=y$ sind alle Vorkommen von x in $x:=a+b$ durch y zu ersetzen:

$$
\begin{aligned}
S &:= "x := a + b" \\
Q &:= "x = y" \\
P\{S\}Q &:= P\{x := a + b\}x = y \\
P &:= "y = a + b" \\
&\Downarrow \\
y = a + b\{x := a + b\}x &= y
\end{aligned}
$$

c) Sieht man sich die Spezifikation genauer an, so stellt man fest, dass sie sich vereinfachen lässt:

$$TRUE\{z := 0\}z < 8$$

Diese Aussage ist stets wahr, da immer $0<8$ gilt.

Aufgabe 2

Die Inferenzregel für WHILE-Schleifen (ohne Beachtung der totalen Korrektheit) wurde in der Vorlesung hergeleitet:

$$\frac{P \wedge B\{S\}P}{P\{\textit{while } B \textit{ do } S \textit{ end}\}P \wedge \neg B}$$

wobei B die Bedingung ist, die bei jedem Schleifendurchlauf überprüft wird, S der auszuführende Algorithmus und P eine Zusicherung der Umgebung darstellt.

Die DO-WHILE-Schleife zeichnet sich dadurch aus, dass sie nicht abweisend ist, d.h. mindestens ein Schleifendurchlauf garantiert wird. Sie wird so oft wiederholt, wie die Bedingung B erfüllt ist. Hier steckt auch ein kleiner Unterschied zur REPEAT-UNTIL-Schleife aus Pascal, da dort solange durchlaufen wird, bis B *nicht* mehr erfüllt ist.

Für den ersten Schleifendurchlauf gilt:

$$P\{S\}Q$$

Die Nachbedingung *Q* ist lediglich temporärer Natur. Ich nutze sie in den nachfolgenden Durchläufen, die zusätzlich von der Bedingung B abhängen:

$$Q \wedge B\{S\}Q$$

Zusammengefasst lässt sie die DO-WHILE-Schleife in Hoares Notation beschreiben als:

$$\frac{\begin{array}{c}P\{S\}Q \\ Q \wedge B\{S\}Q\end{array}}{P\{\textit{do } S \textit{ while } B\}Q \wedge \neg B}$$

Aufgabe 3

Unmittelbares Einsetzen führt hier zum Ziel:

a) $x + y = 10\ \{z = x + y\}\ z = 10$

b) $f(x) = u\ \{x = f(x)\}\ x = u$

c) $b \geq 0\ \{a = 0\}\ a = 0 \wedge b \geq 0$

Aufgabe 4

a) Die Schleifeninvariante ist vollkommen unabhängig von der Vererbung, da objektorientiertes Programmieren in keiner Weise die Funktionsweise von Schleifen verändert. Etwas anders sieht es bei Klasseninvarianten aus: sie werden durch Vererbung verschärft, da die geerbte Invariante um zusätzliche, neu eingeführte Bedingungen konkretisiert wird.

b) Ähnlich wie in Aufgabenteil a) gilt auch hierbei, dass Schleifeninvarianten unabhängig von der Schnittstellendefinition (d.h. den Zusicherungen einer Klasse nach außen) sind. Sie können aus diesem Grunde beliebig verändert werden, Verallgemeinerungen sind also auch erlaubt.

c) *(LÖSUNG IST NOCH NICHT FERTIG UND WIRD ÜBERARBEITET !!!)*

Als erstes untersuche ich den Fall, dass die Schleife sich abweisend verhält, also *x* nicht-positiv ist:

$$\begin{aligned} anzahlSummanden &= 0 \\ anzahlSummanden^2 \le x &< (anzahlSummanden+1)^2 \\ 0 \le x &< 1 \end{aligned}$$

Leider zeigt sich bereits hier, dass die Nachbedingung nicht immer erfüllt ist, da als Vorbedingung lediglich für x eine ganze Zahl gefordert ist. Die Einschränkung, nur natürliche Zahlen zu nehmen, ist hingegen vernünftig, mit ihr gilt nach obigen Gleichungen die Nachbedingung. Für den Codeausschnitt bedeutet das, dass man die Signatur ändern muss:

```
int ganzzahligeWurzel(unsigned int x)
```

Ich halte es für sinnvoll, zusätzlich auch den Rückgabewert als `unsigned int` zu definieren.

Der nächste Schritt besteht in der Ermittlung der Schleifeninvariante. Nach kurzer Überlegung findet man:

$$\begin{aligned} anzahlSummanden_0 &= 0 \\ anzahlSummanden_n &= anzahlSummanden_{n-1} + 1 \\ &= n \end{aligned}$$

$$\begin{aligned} ungerade_0 &= 1 \\ ungerade_n &= ungerade_{n-1} + 2 \\ &= 2n + ungerade_0 \\ &= 2n+1 \end{aligned}$$

$$\begin{aligned} summe_0 &= 1 \\ summe_n &= summe_{n-1} + ungerade_n \\ &= summe_{n-1} + 2n + 1 \\ &= (n+1)^2 \end{aligned}$$

All diese Aussagen lassen sich per vollständiger Induktion beweisen, aus Zeitgründen verzichte ich jedoch darauf, da die Herleitungen jeweils zu primitiv sind, um damit wertvolles Papier aus unseren Regenwälder zu beschmieren. Die Bäume sollte man viel besser zum Bau von finnischen Saunen verwenden.

Sobald die Abbruchbedingung erfüllt ist, gilt:

$$summe_{n-1} \le x < summe_n$$
$$n^2 \le x < (n+1)^2$$
$$anzahlSummanden^2 \le x < (anzahlSummanden+1)^2$$

Ich habe somit gezeigt, dass die Bedingung $anzahlSummanden^2 \le x < (anzahlSummanden+1)^2$ stets erfüllt ist, sie also bei jedem nur erdenklichen Parameter *x* (mit Ausnahme negativer Zahlen, siehe oben) als Nachbedingung gültig ist.

Ich kürze die Variablennamen ab, um die Tabelle nicht zu groß werden zu lassen (anzahlSummanden → anzSum, ungerade → ung):

Nr.	Vorbedingung	Anweisung	Nachbedingung	Axiom
1	TRUE ⇒ anzSum²≤x<(anzSum+1)²			Lemma

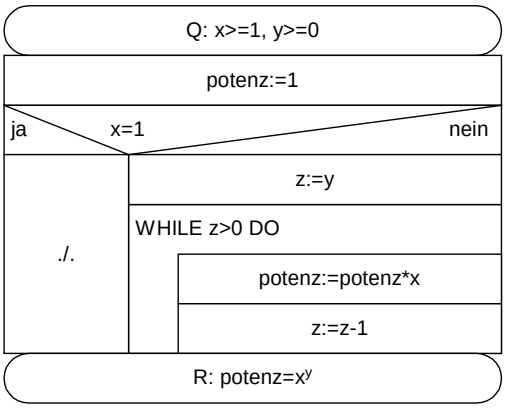
2	anzSum²≤x<(anzSum+1)²	`summe := 1,`
		`anzSum := 0,`
		`ung := 1`
			anzSum²≤x<(anzSum+1)²	A0, A3

3		`while (summe <= x)`		
4		`ungerade += 2`		
		`summe += ungerade`		
		`anzSum++`		

d) Totale Korrektheit erweitert die partielle Korrektheit um die Forderung nach der Terminierung des Algorithmus.

e) Die totale Korrektheit lässt sich in diesem Fall beweisen. Dabei ist lediglich zu zeigen, dass die `while`-Schleife terminiert, falls sie betreten wird. In diesem Fall wäre `x>0` und konstant. Da `summe` streng monoton wächst (weil ungerade streng monoton wächst), ist die Differenz `x-summe` eine Nullfolge. Im Bereich der ganzen Zahlen heißt dies, dass die Null definitiv in endlicher Zeit erreicht wird, die Schleife somit abbricht und der Algorithmus terminiert.

Aufgabe 5



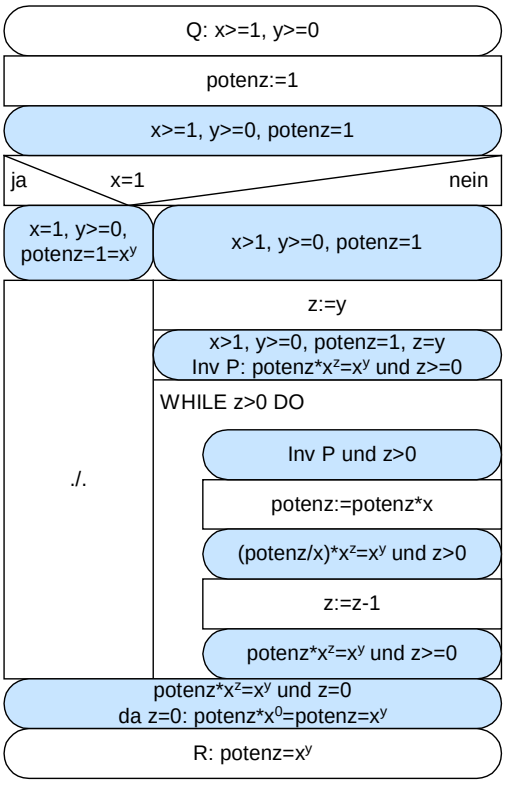
In der innersten Schleife wird *z* von *y* bis auf 0 heruntergezählt. Gleichzeitig multipliziert man *potenz* jedesmal mit *x*. Zu einem beliebigen Moment gilt in dieser Schleife zu Beginn eines Durchlaufes:

$$potenz = x^{y-z}$$

Wenn man das Endresultat einfließen lassen will, dann erhält man:

$$x^y = x^{y-z} \cdot x^{y-(y-z)}$$
$$x^y = potenz \cdot x^z$$

Letzteres ist die gesuchte Schleifeninvariante. Diese überprüfe ich nun, indem ich in das Struktogramm alle Zusicherungen einfüge (blau unterlegt):



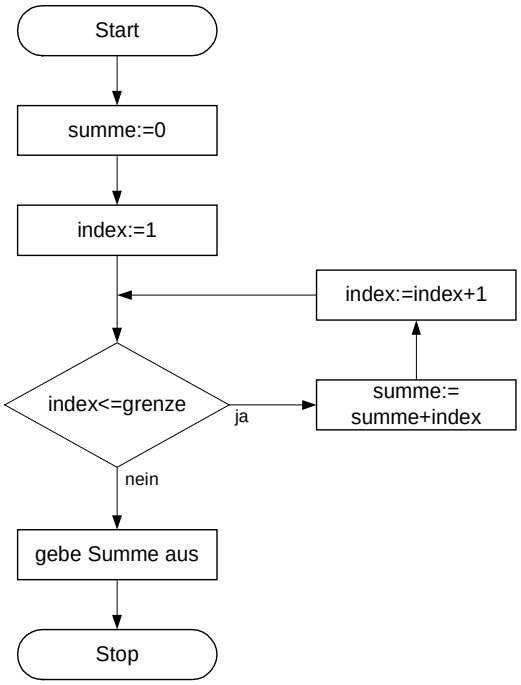
In der Tat stimmt die gefundene Schleifeninvariante mit den Zusicherungen überein.

Aufgabe 6

Dieser Codeausschnitt ist zu untersuchen:

```
 1 public void writeSum(int grenze)
 2 {
 3    int summe = 0;
 4    int index = 1;
 5
 6    while (index <= grenze)
 7    {
 8       summe = summe + index;
 9       index = index + 1;
10    }
11
12    System.out.println(summe);
13 }
```

a) Der Programmablaufplan ist nicht sonderlich aufwändig, da nur eine Schleife auftritt und der Rest sequentiell abgearbeitet wird:



b) Das Programm selbst verlangt lediglich einen Integerwert für `grenze` als Eingangszusicherung. Es wäre jedoch exakter, einen nicht-negativen Wert als Anforderung zu stellen (obwohl das Programm auch damit korrekt arbeitet). Es gibt formal keine Ausgangszusicherung, da kein Rückgabewert existiert, sondern nur eine Bildschirmausgabe getätigt wird.

Ich vermute jedoch, dass die Aufgabestellung nur auf die Schleife ausgerichtet ist, dort lassen sich beide Zusicherung wesentlich besser beschreiben. Als Eingangszusicherung gelten dann drei (Un-) Gleichungen:

$$grenze > 0 \wedge grenze \geq index \ \wedge summe = \sum_{i=0}^{index-1} i$$

Die Ausgangssicherung sieht ganz ähnlich aus, die Ungleichung hat nur ein geändertes Relationszeichen:

$$grenze > 0 \wedge grenze < index \ \wedge summe = \sum_{i=0}^{index-1} i$$

Diese Bedingungen sind aber nur erfüllt, wenn die Schleife auch wirklich betreten wird, sonst gilt:

$$grenze \leq 0 \ \wedge summe = 1$$

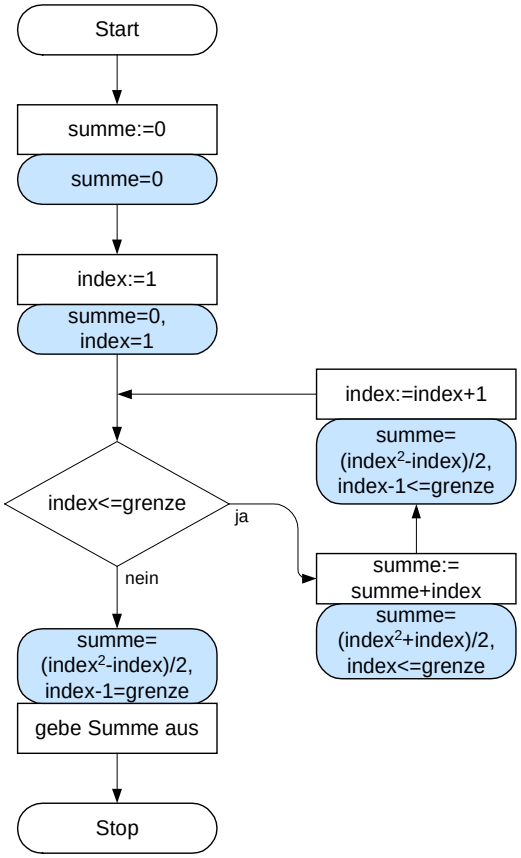
c) Die Schleifeninvariante erhält man sofort, wenn man sich die netten Geschichten von C.F. Gauß zurückerinnert (Addition aller natürlichen Zahlen von 1 bis 100):

$$summe = \frac{index \cdot (index - 1)}{2}$$

Für die weiteren Berechnungen hat es sich als einfacher erwiesen, die Klammer aufzulösen:

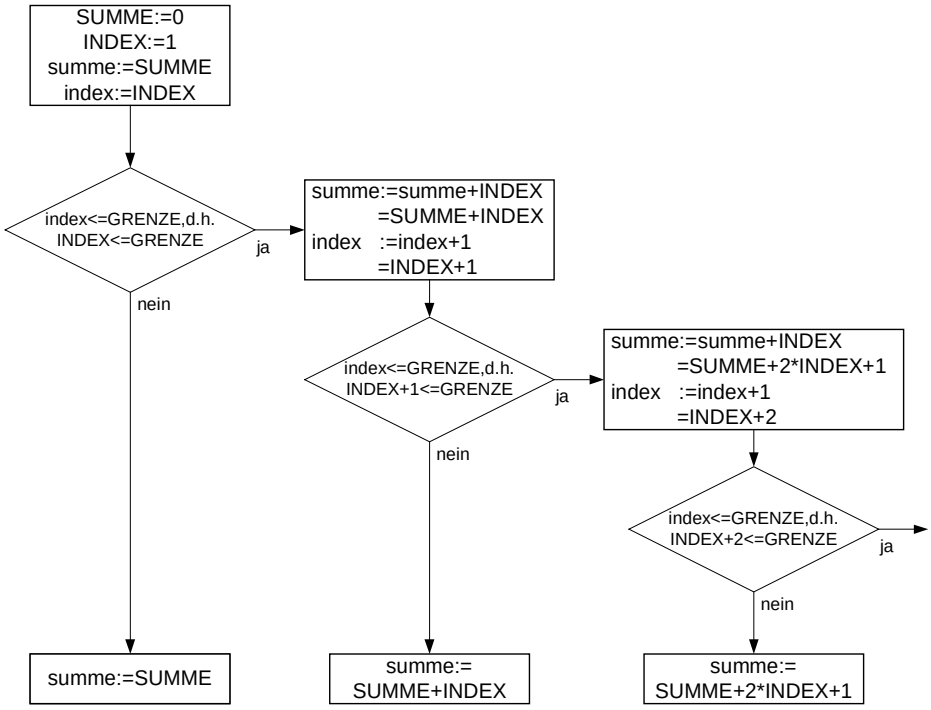
$$summe = \frac{index^2 - index}{2}$$

d) Ich schließe den Fall aus, dass *index* negativ ist:



e) *index* ist als natürliche Zahl streng monoton steigend und wächst gegen die Konstante *grenze*. Aus diesem Grunde ist *grenze* - *index* eine Nullfolge, d.h. der Algorithmus terminiert.

f) Ich beschränke mich auf den Schleifenkörper, der symbolische Wert ist GRENZE. Alle anderen Variablen lassen sich durch konkrete Werte ersetzen:

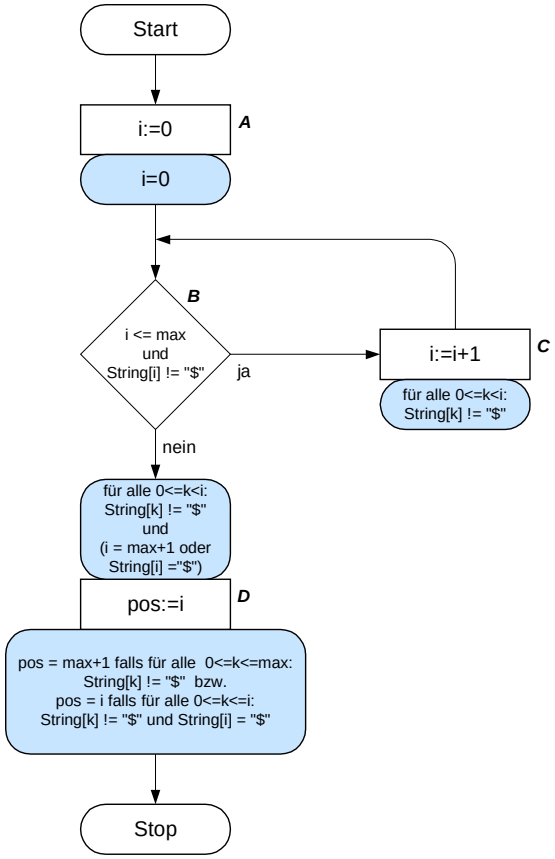


g) Der Computer ist nicht in der Lage, alle natürlichen Zahlen darzustellen. Stattdessen muss er sich auf einen Teilbereich davon beschränken, z.B. von 0 bis 2^{31}-1. Obwohl der oben untersuchte Algorithmus mathematisch korrekt ist, zeigt er nicht auf, ob dieser Wertebereich verlassen wird, was zu einem falschen Ergebnis führt.

Aufgabe 7

Hinweis: Ich definiere, dass *max* stets positiv ist.

a) Der mit Annotationen versehene Programmablaufplan hat die Gestalt:



b) Die Schleifeninvariante wurde bereits in Aufgabenteil a) entdeckt, sie lautet:

$$\forall 0 \le k < i : String[k] != "\$"$$

c) Die Schleife wird höchstens *max*+1 Mal durchlaufen (wenn *String* keinerlei Dollarzeichen enthält). Da es möglich ist, dass in String an jeder nur denkbaren Stelle ein Dollarzeichen auftaucht, also von der Position Null bis *max*+1, entstehen *max*+2, d.h. 4 symbolische Ausführungspfade.

Im oben abgebildeten Programmablaufplan habe ich die einzelnen Anweisungen bzw. Alternativen mit den Buchstaben A bis D versehen. Die potentiell möglichen Pfade und die dazu notwendigen Bedingungen sind:

Pfad	Bedingung	pos
ABD	*String*[0] = „$“	0
ABCD	$k \in \{0\}$: *String*[*k*] != „$“ *String*[1] = „$“	0+1=1
ABCBD	$k \in \{0,1\}$: *String*[*k*] != „$“ *String*[1] = „$“	0+1+1=2
ABCBCD	$k \in \{0,1,2\}$: *String*[*k*] != „$“	0+1+1+1=3

d) Zur Erfüllung der Zweigüberdeckung ist jeder Pfad min. einmal zu durchlaufen, d.h. das gesamte Prädikat muss min. einmal wahr und min. einmal falsch sein. Die Bedingungsüberdeckung ist schärfer definiert, sie verlangt, dass jedes Teil-Prädikat min. einmal wahr und min. einmal falsch ist. Für das Beispiel muss man daher min. einen Testfall finden, der die volle Länge *max* von *String* testet.

Aufgabe 8 (Algebraische Spezifikation)

Die zu verwendenden Funktionen sind laut Aufgabenstellung definiert durch:

Funktion	Abbildung		
neu	ZEICHEN	→	BAUM
erstelle	BAUM × ZEICHEN × BAUM	→	BAUM
links	BAUM	→	BAUM
rechts	BAUM	→	BAUM
wert	BAUM	→	ZEICHEN
istBlatt	BAUM	→	BOOLEAN
knoten	BAUM	→	INTEGER

Und die dazugehörigen Axiome für alle $b,b_1,b_2 \in$ BAUM und $c \in$ ZEICHEN:

Nr.	Axiom
1	*wert*(*neu*(c)) = c
2	*wert*(*erstelle*(b_1, c, b_2)) = c
3	*istBlatt*(*neu*(c)) = true
4	*links*(b) = b_1 if b==*erstelle*(b_1, c, b_2) else *error*
5	*rechts*(b) = b_2 if b==*erstelle*(b_1, c, b_2) else *error*
6	*knoten*(b) = 0 if *istBlatt*(b) else *knoten*(*links*(b)) + *knoten*(*rechts*(b)) + 1

Die Vereinfachung der Terme erfolgt schrittweise, dabei löse ich von innen nach außen auf. Zur besseren Orientierung sind die Verschachtelungstiefen farblich markiert, Terminale bleiben schwarz:

1. Ausgangsterm:
   wert(*rechts*(*erstelle*(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘)), ‚d‘, *links*(*erstelle*(*neu*(‚e‘), ‚f‘, *neu*(‚g‘)))))))

   Axiom 4 entfernt *links*:
   wert(*rechts*(*erstelle*(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘)), ‚d‘, *neu*(‚e‘))))

   Axiom 5 erledigt das gleiche für *rechts*:
   wert(*neu*(‚e‘))

   Axiom 1 führt zum Ziel:
   ‚e‘

2. Ausgangsterm:
   istBlatt(*links*(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘))))

   Axiom 4 ersetzt *links*:
   istBlatt(*neu*(‚a‘))

   Axiom 3 beendet die ganze Angelegenheit:
   true

3. Ausgangsterm:
   knoten(*erstelle*(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘)), ‚d‘, *neu*(‚e‘)))

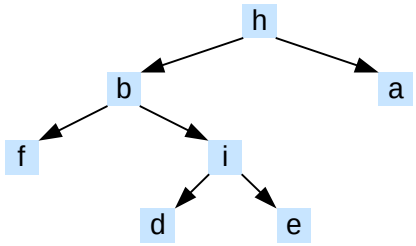
   Diesmal gehe ich von außen nach innen vor, da jeweils nur Axiom 6 zu verwenden ist:
   knoten(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘))) + *knoten*(*neu*(‚e‘)) + 1
   knoten(*erstelle*(*neu*(‚a‘), ‚b‘, *neu*(‚c‘))) + 0 + 1
   knoten(*neu*(‚a‘)) + *knoten*(*neu*(‚c‘)) + 1 + 0 + 1
   0 + 0 + 1 + 0 + 1

   Mit einem Taschenrechner erhält man die Summe:
   2

Die graphische Darstellung des Terms

erstelle(*erstelle*(*neu*(‚f‘), ‚b‘, *rechts*(*erstelle*(*neu*(‚c‘), ‚g‘, *erstelle*(*neu*(‚d‘), ‚i‘, *neu*(‚e‘))))), ‚h‘, *neu*(‚a‘))

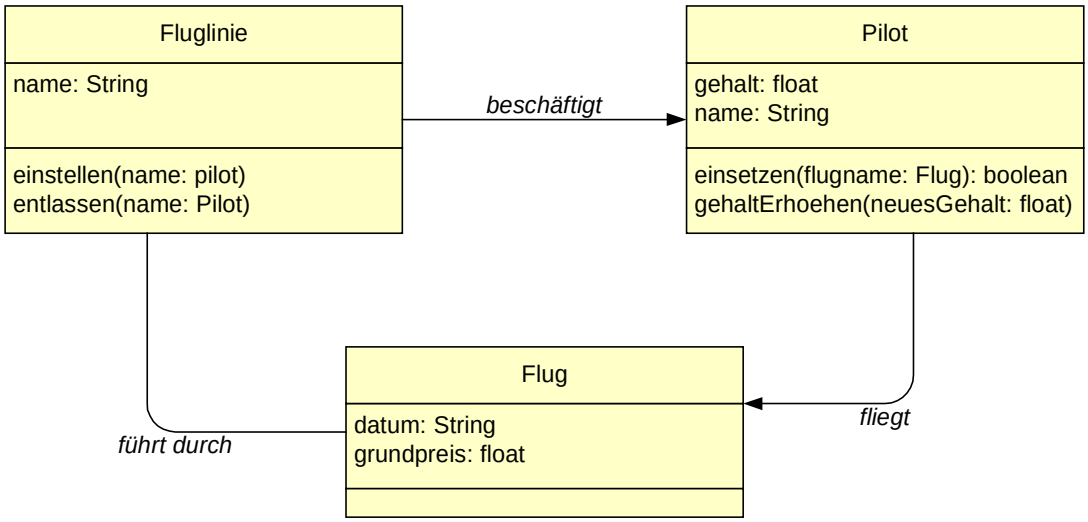
erfolgt am besten mit Hilfe eines Baums:



Man sieht schön, dass die Terminale ‚c‘ und ‚g‘ durch die Funktion *rechts* entfallen.

Aufgabe 9 (Formale Objektorientierte Softwareentwicklung)

Ich hatte leider nicht die Zeit, diese Aufgabe selbstständig zu bearbeiten. Aus diesem Grunde gebe ich nur die Lösung von Jörg Gericke wieder, die leider auch nicht vollständig ist:



Die Vor- / Nachbedingungen (Teil g) lauten:

context `Flug::business() : Real`
pre: true
post: result = 5*grundpreis

context `Flug::economy() : Real`
pre: true
post: result = 2*grundpreis

context `Pilot::gehaltErhoehen(neuesGehalt: Real)`
pre: $0 \leq$ neuesGehalt $\leq$ gehalt*0.16
post: gehalt = gehalt@pre + neuesGehalt

context `Pilot::einsetzen(flug: Flug)`
pre: es gibt kein einsatz mit datum == flug.datum
post: einsatz = einsatz@pre.add(flug)