Vorgehensweise beim Nachweis der Korrektheit:

Schreibe zwischen je zwei Anweisungen eine prädikatenlogische Formel und versuche mit Hilfe der Hoareschen Regeln zu bewiesen, dass jeweils {A} c {B} gilt.

Da die Hoareschen Regeln sich nur auf die Anweisungen "leere Anweisung", "Wertzuweisung", "Hintereinanderausführung von Anweisungen", "Alternative" und "while-Schleife" beziehen, können wir dieses Schema bisher auch nur auf Programme anwenden, die höchstens aus diesen Bestandteilen bestehen. Will man beliebige Ada-Programme untersuchen, so muss man weitere Regeln einführen.
(Solche weiteren Regeln kann man aufstellen. Wir tun dies hier nicht, weil es uns ja nur um das Prinzip geht und weil diese Regeln rasch recht kompliziert werden.)

Kann man dieses Vorgehen automatisieren, d.h., kann man ein Programm schreiben, das
- ein Programm einliest,
- die Spezifikation einliest,
- schrittweise die erforderlichen Zusicherungen ermittelt und
- den Beweis der Korrektheit führt?

Nein, denn (Sie ahnen es schon) auch dieses Problem, die Korrektheit eines Programms zu beweisen, ist algorithmisch nicht lösbar.

(Anschaulich ist diese Aussage klar: Denn wer die Korrektheit nachweisen will, muss auch beweisen können, ob ein Programm überhaupt anhält; und damit ist dieses Korrektheitsproblem mindestens so schwer wie das Halteproblem.)

Aber: Man kann natürlich ein interaktives Programm, also ein "Unterstützungssystem" bauen, welches
- ein Programm einliest,
- die Spezifikation einliest,
- diese Spezifikation als letzte Zusicherung verwendet und versucht, die vor der letzten Anweisung einzufügende Zusicherung zu konstruieren oder den Benutzer aufzufordern, einen Vorschlag einzugeben,
- einen Beweis für die Korrektheit dieses einen Schrittes zu führen oder den Benutzer aufzufordern, einen solchen Beweis einzugeben und diesen nachzuvollziehen,
- das Gleiche für die Anweisung davor zu wiederholen usw.

Auf diese Weise könnte ein Beweis der Korrektheit ermöglicht und teilweise sogar automatisiert werden können.

Wie muss man hierbei vorgehen?
Der wichtigste Schritt ist:

Ausgehend von einer Zusicherung und der davor stehenden Anweisung muss man versuchen die Zusicherung zu konstruieren, die vor der letzten Anweisung einzufügen ist.

Also: Finde zu der Situation

c {B}

eine Zusicherung A, so dass

{A} **c** {B}

gilt.

(Das geht in vielen Fällen tatsächlich mittels folgender "wp".)

Triviales Beispiel: Gegeben sei

```
x := x +1;
```
{x > 7}

Dann wird eine Zusicherung vor der Wertzuweisung gesucht:

{x > 6}
```
x := x +1;
```
{x > 7}

Dies entspricht der Hoareschen Regel: {A'} **x:=ß** {A}, wobei A' aus A durch Ersetzen von x durch ß hervorgeht.

Man hätte auch die Zusicherung {x > 20} nehmen können:

{x > 20}
```
x := x +1;
```
{x > 7}

Welche Zusicherung soll man nehmen?

Man denke an die Konsequenzregel. Es gilt:

{x > 20} ⇒ {x > 6}
```
x := x +1;
```
{x > 7}

Die Zusicherung {x>6} ist "schwächer" als die Zusicherung {x>20}. "Schwächer" bedeutet: Die Zusicherung {x>6} trifft auf mehr Werte zu als die Zusicherung {x>20}.

{x>0} wäre eine noch schwächere Bedingung. Leider gilt aber die Formel

{x > 0}
```
x := x +1;
```
falsch
{x > 7}

nicht mehr, da x ja einen der Werte 1, 2, 3, 4, 5 oder 6 haben könnte. Gibt es zu **c** und B vielleicht eine "schwächste" Zusicherung A, für die {A} **c** {B} zutreffend ist? **Ja!**

Bezeichnungen: Wir müssen nun in der Formel

{A} **c** {B}

zwischen den Zusicherungen A und B unterscheiden. Statt "Zusicherung" sagt man oft auch "Bedingung", und daher nennt man

A die Vorbedingung zur Anweisung **c**

(englisch: *precondition*)

und

B die Nachbedingung zur Anweisung **c**

(englisch: *postcondition*).

Unsere Aufgabe lautet also: Suche zu der Situation

c {B}

eine Vorbedingung A, so dass

{A} **c** {B}

gilt. Wenn es mehrere solche Vorbedingungen gibt, dann suche die "schwächste" Vorbedingung, d.h., suche ein A mit:

1. {A} **c** {B}
2. wenn für irgendeine Zusicherung A' gilt: {A'} **c** {B}, dann ist A eine Folgerung aus A', d.h., dann gilt A' ⇒ A.

Definition:

Eine solche Zusicherung A heißt "schwächste Vorbedingung" zu **c** und B (englisch: *weakest precondition*, abgekürzt wp).

Man schreibt dann: **A = wp(c,B)**.

Als erstes müssen wir fragen, ob unsere Definition in sich stimmig ist (man sagt auch, ob sie "wohldefiniert" ist). Es könnte ja sein, dass es zu **c** und B in der Regel keine schwächste Vorbedingung gibt oder dass es mehrere gibt.

Ohne Beweis notieren wir hier:
Es gibt stets eine schwächste Vorbedingung zu **c** und B.

Dass diese stets eindeutig ist, ist leicht zu sehen: Wenn es zwei solche schwächsten Vorbedingungen A und A' gäbe, dann müssen gleichzeitig $A' \Rightarrow A$ und $A \Rightarrow A'$ gelten. Zwei solche Zusicherungen sind aber gleichwertig, d.h., es gilt dann $A' \Leftrightarrow A$.

Beispiel 1.7:
c ≡ `if (X mod 2 = 1) then X:=X+3; end if;`
B ≡ X mod 6 = 0
Gesucht ist wp(**c**,B).

Betrachte hierzu alle Belegungen der Variablen X, für die nach Durchführung der Anweisung **c** die Zusicherung B gilt:
{a∈ • | falls a ungerade ist, dann muss a+3 durch 6 teilbar sein;
falls a gerade ist, dann muss a durch 6 teilbar sein}
= {a∈ • | a ist durch 3 teilbar}.

Eine Zusicherung, die genau für die Werte dieser Menge erfüllt ist, muss zur schwächsten Vorbedingung gehören (also zur größten Menge, die die Zusicherung erfüllt). Also gilt hier:

wp(**c**,B) ≡ X mod 3 = 0

Wenn **c** eine Wertzuweisung ist, dann ist wp(**c**,B) in der Regel leicht zu berechnen und dies kann auch automatisiert werden. Die Hintereinanderausführung und die Alternative sind auch noch gut zu handhaben. Als Beispiel betrachte man:

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte zunächst den then-Zweig:
$\mathbf{c_1}$ ≡ **X := -X + 1;**
B ≡ X > 5
wp($\mathbf{c_1}$, B) ≡ X < -4

Dies muss man noch koppeln mit der Bedingung b ≡ X < 0 für den then-Zweig. Dies ergibt die Zusicherung
(X < -4) ∧ (X < 0) , also X < -4.

c ≡ if X<0 then X:=-X+1; else X:=X-1; end if;
B ≡ X > 5

Betrachte nun den else-Zweig:
$\mathbf{c_2}$ ≡ **X := X - 1;**
B ≡ X > 5
wp($\mathbf{c_2}$, B) ≡ X > 6

Dies muss man koppeln mit der Bedingung not(b) ≡ X ≥ 0. So erhält man die Zusicherung
(X > 6) ∧ (X ≥ 0) , also X > 6 für den else-Zweig.

Aus beiden Vorbedingungen erhält man die schwächste Vorbedingung der gesamten Alternative als Disjunktion:
wp(**c**, B) ≡ (X < -4) ∨ (X > 6).

Wesentlich schwieriger ist die Behandlung der while-Schleife. Als Beispiel betrachte man:

c ≡ **`while (X /= 0) loop X := X-2; end loop;`**
B ≡ X mod 3 = 0
wp(**c**, B) ≡ ?

"X mod 3 = 0" ist keine Schleifeninvariante, da die Formel
{(X mod 3 = 0) ∧ (X /= 0)} **`X := X-2;`** {X mod 3 = 0}
nicht erfüllt ist (vgl. Hoaresche Regel 5, sie steht auch auf der nächsten Folie).

Dagegen sind sowohl "X mod 2 = 0" als auch "X mod 2 = 1" Schleifeninvarianten.

Diese Information nützt uns aber nichts. Betrachten wir stattdessen noch mal die 5. Hoaresche Regel:

$$\frac{\{A \wedge b\}\ \mathbf{c}\ \{A\}}{\{A\}\ \mathbf{while\ b\ loop\ c\ end\ loop}\ \{A \wedge \text{not}(b)\}}$$

In unserem Fall ist not(b) ≡ (X = 0), so dass B ≡ X mod 3 = 0 stets erfüllt ist, egal mit welchem Wert von X die Schleife begonnen wurde. Die Frage, ob die Schleife terminiert, spielt nach der Definition von {A} **c** {B} keine Rolle: Denn es soll B nach der Ausführung von **c** erfüllt sein (vgl. Folie 31); wenn jedoch die Schleife nicht endet, "dann ist danach alles erfüllt".

Somit erhalten wir als die schwächste Vorbedingung obiger Schleife die Zusicherung: X∈ •, d.h. die Nachbedingung ist für jede ganze Zahl, mit der man startet, erfüllt.

Im Allgemeinen lässt sich die schwächste Vorbedingung für eine Schleife jedoch algorithmisch nicht berechnen. Diejenigen, die das Programm entwickelt haben, kennen in der Regel aber mindestens eine Schleifeninvariante, nämlich die, die sie bei der Formulierung der Schleife im Sinn hatten. Mit einem interaktiven System kann diese Zusicherung eingegeben werden und das System kann versuchen zu beweisen, dass sie tatsächlich eine Schleifeninvariante ist.

Hier brechen wir die Erläuterung der Korrektheit mit Hilfe der axiomatischen Semantik ab. Es sollten nur die Ideen vorgestellt und ein mögliches Unterstützungssystem angedeutet werden. (Vertiefungen hierzu erfolgen in Vorlesungen über Semantik und z.T. in Vorlesungen über Interaktive Systeme, Wissensverarbeitung sowie Programmiersprachen/Übersetzerbau.)

Historischer Hinweis:
Das Problem, die Richtigkeit von Programmen nachzuweisen, besteht seit dem Beginn der praktisch verwendbaren Programmierung, also seit den 1950er Jahren. Dies führte rasch auf die Frage nach der Bedeutung ("Semantik") eines Programms und dessen präziser Formulierung in Kalkülen.
Die ersten Arbeiten hierzu stammen von R. Floyd (Einführung von festen Stellen im Programm, an denen die Bedeutung ermittelt wird), C.A.R. Hoare (Aufstellung eines Regelsystems) und W. Dijkstra (Einführung der weakest precondition) in den 1960er Jahren. Ab 1970 entwickelt sich eine Fülle von Arbeiten zu diesem Gebiet (denotationale Semantik, Semantik nebenläufiger Systeme, Entwicklung von Kalkülen und Beweissystemen, konkrete Methoden wie model checking usw.).

1.4. Komplexität von Algorithmen

Die Idee: Komplexität ist der Aufwand an Zeit $t_A(w)$ oder an Platz $s_A(w)$, den ein Algorithmus A bei der Eingabe w benötigt, um seine Berechnung durchzuführen. (t kommt von "time complexity" und s von "space complexity".)

Genauer: Wenn ein Algorithmus A eine Eingabe w erhält, dann führt er eine Berechnung durch. Eine Berechnung ist eine Folge von elementaren Anweisungen (leere Anweisung, Wertzuweisung) und Abfragen, die durch die aktuelle Eingabe w festgelegt ist.

$t_A(w)$ = Anzahl der elementaren Anweisungen und Abfragen, die A bei Eingabe w durchläuft, bis er anhält,

$s_A(w)$ = Anzahl der Speicherplätze, auf die A bei Eingabe von w zugreift, bis er anhält.

In der Praxis interessiert man sich nicht für die einzelnen Eingabefolgen w, sondern nur für deren Länge. Daher definiert man:

Etwas konkretere Idee:
Es sei A ein Algorithmus, w sind Eingabefolgen.

$t_A(n)$ = Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,

$s_A(n)$ = Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.

Man beachte: Diese Definition setzt indirekt voraus, dass der Algorithmus A für alle Eingaben anhält; denn falls der Algorithmus A für eine Eingabefolge w der Länge n *nicht* anhält, dann wären $t_A(n)$ [und eventuell auch $s_A(n)$] unendlich groß und somit für Anwendungen uninteressant.

Definition:

Es sei A ein Algorithmus, der für alle Eingaben w anhält. Dann definieren wir:

$t_A(n)$ = Max $\{t_A(w) \mid$ die Länge von w ist n$\}$,

$s_A(n)$ = Max $\{s_A(w) \mid$ die Länge von w ist n$\}$.

t_A und s_A sind also Abbildungen von $\bullet_0$ nach $\bullet_0$.

Hinweis: Legt man ein Maschinenmodell zugrunde, z.B. eine Turing- oder eine Registermaschine, dann lässt sich diese Definition exakt definieren als die Anzahl der durchlaufenen Konfigurationen oder als die Anzahl der besuchten Speicherplätze. Siehe Vorlesung "Theoretische Informatik II", Kapitel Komplexität.

Obige Definition ist aber für unsere Zwecke ausreichend.

Beispiel 1.8: Wir betrachten erneut Beispiel 1.5:

```
x, y, z: Natural;
Get (x, y);
z:=0;
while y > 0 loop
     if (y mod 2 = 0) then
          y:=y div 2;
          x:=x+x;
     else
          y:=y-1;
          z:=z+x;
     end if;
end loop;
Put(z);
```

Dieses Programmstück berechnet die Multiplikation zweier natürlicher Zahlen. Wie groß ist der Aufwand?

```
Get (x, y);                    2 Schritte
z:=0;                          1 Schritt
while y > 0 loop               1 Schritt                    m+1 mal
     if (y mod 2 = 0) then     1 Schritt
          y:=y div 2;
          x:=x+x;              2 Schritte                   m mal
     else
          y:=y-1;              oder:
          z:=z+x;              2 Schritte
     end if;
end loop;
Put(z);                        1 Schritt
```

Zusammen: 4m + 5 Schritte. Aber was ist m?

Die Schleife wird durchlaufen, bis y = 0 ist. Spätestens in jedem zweiten Schritt wird y halbiert. Also wird die Schleife mindestens log(y) mal und höchstens 2 log(y) mal durchlaufen. Also gilt $\log(y) \leq m \leq 2\log(y)$.

Die Eingabewerte seien die natürlichen Zahlen a und b. Die Eingabelänge ist dann n = log(a) + log(b) + 2. (Das "+2" kommt daher, dass man ein Trennzeichen zwischen a und b und ein Zeichen für das Ende der Eingabe benötigt.)
m kann daher durch n nach oben abgeschätzt werden.

Für unseren Multiplikationsalgorithmus A gilt also wegen $m \leq 2 \cdot \log(y) \leq 2 \cdot n$:
$\mathbf{t_A(n)} = \text{Max}\ \{t_A(w) \mid \text{die Länge von w ist n}\} \leq 8n + 5$.

Man braucht drei Speicherplätze für x, y und z, folglich gilt:
$\mathbf{s_A(n)} = \text{Max}\ \{s_A(w) \mid \text{die Länge von w ist n}\} = 3$

Aber: Ist dies eine "richtige" Aufwandsberechnung?
Warum geht zum Beispiel nur die Länge der zweiten Zahl b, die y zugeordnet wird, und nicht auch die der ersten Zahl a, die in x steht, in die Komplexität ein??

..... weil wir jede Wertzuweisung und jede Abfrage so behandeln, als ob sie **in einem Schritt** ausgeführt werden kann.

In der Praxis würde die Anweisung x:=x+x nicht einen Schritt, sondern so viele Schritte benötigen, wie der Wert von x lang ist (ungefähr log(x)); denn so lange dauert die ziffernweise Addition, wenn man sie wie in der Schule erlernt durchführt.

Auch die Frage, ob y durch 2 teilbar ist (y mod 2 = 0), oder die Division durch 2 benötigen in der Regel einen Aufwand proportional zur Länge der jeweiligen Zahl (das hängt aber von der Zahldarstellung ab; bei der Basis 2 oder 10 braucht man nur die letzte Ziffer zu prüfen).

Also noch mal rechnen:

```
Get (x, y);        log(a) + log(b) = n  Schritte
z:=0;              1 Schritt
while y > 0 loop   ≤ log(b) Schritte   2 log(b)+1 mal
     if (y mod 2 = 0) then ≤ log(b)  Schritte
          y:=y div 2;  ≤ log(b) Schritte
          x:=x+x;      log(a)+log(b) Schritte
     else                                          2 log(b) mal
          y:=y-1;      log(b) Schritte
          z:=z+x;      log(a)+log(b) Schritte
     end if;
end loop;
Put(z);   log(a) + log(b) = n  Schritte
```

Zusammen: ≤ (2log(b)+1) (4log(b)+log(a))+2n+1 ≤ ca. 8n(n+1)+1 Schritte.

Bezeichnung:
Berechnet man die Komplexität $t_A(n)$ bzw. $s_A(n)$ so, dass jede elementare Anweisung und jede Abfrage genau einen Schritt dauern, bzw. dass jede Zahl genau einen Speicherplatz belegt, so spricht man von der **uniformen Komplexität**.
Berechnet man die Komplexitäten so, dass jede Operation so viel Zeit kostet, wie die zeichenweise Ausführung erfordert, bzw. dass Werte so viel Platz belegen, wie sie Zeichen haben, so spricht man von der **logarithmischen Komplexität**.

In der Praxis berechnet man stets zunächst die uniforme Komplexität: Sie gibt meist eine hinreichend genaue Orientierung über den zu erwartenden Aufwand. Erst für eine genaue Abschätzung bzw. bei der Programmierung betrachtet man die logarithmische Komplexität, die den tatsächlichen Aufwand wesentlich genauer wiedergibt.