Terme, Ausdrücke und Muster in Gentle

Zwei Typen (als Grundlage für Beispiele)

```
1  type FARBE c() m() y()
2  type BAUM
3     leer()
4     b(Vordergrund:FARBE, Hintergrund:FARBE, links:BAUM, rechts:BAUM)
```

`c`, `m`, `y`, `leer` und `b` sind *Konstruktoren* (oder: Funktoren) der Typen `FARBE` und `BAUM`. Sie werden in Gentle immer mit runden Klammern dahinter notiert.

In einem Gentle-Programm folgt der *Typ* einer Variablen immer eindeutig aus dem *Zusammenhang*, in dem die Variable verwendet wird. Damit die folgenden Beispiele auch *ohne* einen solchen Zusammenhang verständlich sind, sei abgemacht:

`F, F1, F2,` ... sind Variablen vom Typ `FARBE`
`B, B1, B2,` ... sind Variablen vom Typ `BAUM`

Terme

Statt einer formalen Definition hier ein paar Beispiele für *Terme* des Typs `BAUM`:

```
T01: leer()
T02: b(m(),c(),leer(),leer())
T03: b(y(),y(),b(m(),y(),leer(),leer()),leer())
T04: b(c(),m(),b(y(),c(),leer(),leer()),b(m(),y(),leer(),leer()))
T05: B
T06: b(c(),y(),B,leer())
T07: b(F,y(),leer(),B)
T08: b(F1,F2,B1,B2)
T09: b(m(),m(),b(F1,F2,leer(),B1),b(y(),F3,B2,B3))
```

Der Term `T05` soll deutlich machen, dass jede *Variable* ("allein für sich") auch ein *Term* ist.

Def.-01: Ein **Grundterm** ist ein Term, in dem *keine Variablen* vorkommen (siehe T01 bis T04).

Def.-02: Einen **Spezialfall** eines Terms T erhält man, indem man in T
null oder mehr Variablen durch je einen *Term* ersetzt.

Beispiel-01: Ersetzt man im Term `b(F,c(),leer(),B)`
die Variable `F` durch den Term `m()` und
die Variable `B` durch den Term `b(F,c(),leer(),B)`, so erhält man
den Spezialfall `b(m(),c(),leer(),b(F,c(),leer(),B))`

Beispiel-02: Ersetzt man im Term `b(F,c(),leer(),B)`
null Variablen durch Terme, so erhält man
den Spezialfall `b(F,c(),leer(),B)`
(d.h. jeder Term ist ein Spezialfall von sich selbst).

Def.-03: Einen **Grundspezialfall** eines Terms T erhält man, indem man in T
jede Variable durch einen *Grundterm* ersetzt.

Beispiel-03: Ersetzt man im Term `b(F,c(),leer(),B)`
die Variable `F` durch den Grundterm `m()` und
die Variable `B` durch den Grundterm `b(c(),y(),leer(),leer())`, so erhält man
den Grundspezialfall `b(m(),c(),leer(),b(c(),y(),leer(),leer()))`

Beispiel-04: Da der Term `b(c(),c(),leer(),leer())` keine Variablen enthält, ist er ein *Grundspezialfall von sich selbst* (und der *einzige* Grundspezialfall von sich selbst).

Aufgabe-01: Welche der obigen Terme (T01 bis T09) sind *Spezialfälle* des Terms `b(F1,F2,B1,leer())`? Welche der Terme sind *Grundspezialfälle* dieses Terms?

Aufgabe-02: Geben Sie für jeden der folgenden Terme an, *wie viele* Grundspezialfälle er hat:
1.) `b(F,c(),leer(),leer())` **2.)** `b(F1,F2,leer(),leer())`
3.) `b(m(),m(),B,leer())` **4.)** `F` **5.)** `B` **6.)** `leer()` **7.)** `b(F1,F2,B1,B2)`.

Merke: Ein Term T beschreibt die *Menge* seiner *Grundspezialfälle.* Kurz: T beschreibt GT(T).

Aufgabe-03: Geben Sie von jedem der folgenden Terme T die Menge GT(T) seiner Grundspezialfälle an:
1.) F **2.)** b(c(), `F, leer(), leer())` **3.)** `b(m(), y(), leer(), leer())`

Aufgabe-04: Geben Sie Terme an, die die folgenden *Mengen von Grundtermen* beschreiben:
1. Die Menge aller nicht-leer()-en Bäume?
2. Die Menge aller Bäume?
3. Die Menge aller vollen Bäume der Tiefe 2 (d.h. jeder Ast muss die Länge 2 haben)?

Def.-04: Eine **Variablenbelegung** ist eine Menge von Zweitupeln der Form `V=W`,
wobei `V` eine *Variable* und `W` ein Wert (für diese Variable) ist.
Natürlich muss der Wert zum selben *Typ* gehören wie die Variable.

Die *Werte* in einer Variablenbelegung werden durch *Grundterme* dargestellt.

Beispiel-05: Eine Variablenbelegung:

`VB1` gleich `{F1=m(),F2=m(),B1=leer(),B2=b(c(),y(),leer(),leer())}`

Die Variablenbelegung `VB1` belegt die Variablen `F1, F2, B1` und `B2` mit Werten.

Ausdrücke und Muster

Einen *Term* kann man auf zwei verschiedene Weisen benutzen: Als *Ausdruck* oder als *Muster.*

Einen Ausdruck kann man *auswerten* (Grundoperation: *Auswertung*)
Ein Muster kann man *abgleichen* (Grundoperation: *Musterabgleich*)

Grundoperation	**Was geht rein?**		**Was kommt raus?**
Auswertung	Ausdruck	Variablenbelegung	Wert
Musterabgleich	Muster	Wert	Variablenbelegung

Regel-A: Bei einer Auswertung eines Ausdrucks A mit einer Variablenbelegung VB muss VB für jede in A vorkommende Variable eine Belegung enthalten.

Zur Erinnerung: In Gentle darf man eine Variable nur initialisieren, ihren Wert danach aber nicht mehr verändern.

Daraus folgt:

Regel-M: Bei einem Musterabgleich mit einem Muster M darf keine der in M vorkommenden Variablen schon mit einem Wert belegt sein ("alle Variablen müssen ganz neu sein", denn sie werden durch den Musterabgleich vereinbart und initialisiert).

Die Grundoperation *Auswertung* (evaluation of expessions) gibt es in praktisch allen verbreiteten höheren Programmiersprachen. *Musterabgleiche* (pattern matching) als Grundoperation gibt es nur in wenigen Sprachen (z.B. in Prolog, Haskell, Gentle, ...).

Beispiel-01: Eine *Auswertung:*
Ausdruck T1: `b(m(),F,B,leer())`
Variablenbelegung VB1: `{F=c(),B=b(y(),y(),leer(),leer())}`
Wert W1: `b(m(),c(),b(y(),y(),leer(),leer()),leer())`

Beispiel-02: Ein *Musterabgleich:*
Muster T2: `b(m(),F,B,leer())`

Wert W2: `b(m(),y(),leer(),leer())`
Variablenbelegung VB2: `{F=y(),B=leer()}`

Man beachte dass in diesen beiden Beispielen derselbe Term `b(m(),F,B,leer())`
einmal als *Ausdruck* und einmal als *Muster* verwendet wird.

Die Programme vieler Programmiersprachen müssen eine *Kontextbedingung* der folgenden Art erfüllen: Bevor eine Variable in einem Ausdruck *benutzt* werden darf, muss sie *vereinbart* und *initialisiert* ("mit einem Anfangswert versehen") werden. Auch der Gentle-Compiler akzeptiert Programme nur dann, wenn sie diese Kontextbedingung erfüllen. Damit ist dann garantiert, dass eine *Auswertung* (eines Ausdrucks) immer "klappt" und als Ergebnis einen Wert liefert.

Im Gegensatz dazu kann ein Musterabgleich *gelingen* oder *misslingen.*

Beispiel-03: Ein *Musterabgleich* der misslingt:
Muster T3: `b(m(),F,B,leer())`
Wert W3: `b(m(),y(),leer(),b(y(),y(),leer(),leer()))`

Der Teilterm `leer()` im Muster T3 kann nicht mit dem entsprechenden
Teilterm `b(y(),y(),leer(),leer())` im Wert W3 *abgeglichen* werden.
Deshalb *misslingt* dieser Musterabgleich und liefert somit auch *keine Variablenbelegung* als Ergebnis.

Aufgabe-05: Führen Sie folgende *Musterabgleiche* durch. Geben Sie für jeden gelingenden Abgleich die sich ergebende *Variablenbelegung* an.

5.1. Muster T41: `b(F, y(),leer(),B)`
Wert W41: `b(c(),y(),leer(),leer())`
5.2. Muster T42: `b(F, y(),leer(),B)`
Wert W42: `b(c(),y(),leer(),b(c(),y(),leer(),leer()))`
5.3. Muster T43: `b(F, y(),leer(),B)`
Wert W43: `b(c(),y(),b(c(),y(),leer(),leer()),leer())`
5.4. Muster T44: `B5`
Wert W44: `b(c(),y(),leer(),leer())`
5.5. Muster T45: `b(c(),y(),leer(),leer())`
Wert W45: `b(c(),y(),leer(),leer())`
5.6. Muster T46: `b(c(),m(),leer(),leer())`
Wert W46: `b(c(),y(),leer(),leer())`

Terme sind Bäume

Ein Term ist eine "lineare Darstellung" eines *Baumes.* Das gilt für die Terme *aller* algebraischen Typen (nicht nur für Terme des Typs `BAUM`).

Beispiel-04: Die Baumstruktur des Terms `T09` durch Einrückung hervorgehoben:

```
b(
   m(),
   m(),
   b(
      F1,
      F2,
      leer(),
      B1
   ),
   b(
      y(),
      F3,
      B2,
      B3
   )
)
```

Lösungen zu den Aufgaben:

Lösung zu Aufgabe-01:

Die Terme T2, T3 und T6 sind *Spezialfälle* des Terms `b(F1, F2, B1, leer())`.
Die Terme T2 und T3 sind sogar *Grundspezialfälle* dieses Terms.

Lösung zu Aufgabe-02:

1.) `b(F,c(),leer(),leer())` hat 3 Grundspezialfälle,
2.) `b(F1,F2,leer(),leer())` hat 9 Grundspezialfälle,
3.) `b(m(),m(),B,leer())` hat unendlich viele Grundspezialfälle,
4.) `F` (eine Variable vom Typ `FARBE`) hat 3 Grundspezialfälle,
5.) `B` (eine Variable vom Typ `BAUM`) hat unendlich viele Grundspezialfälle,
6.) `leer()` hat einen Grundspezialfall (nämlich den Grundspezialfall `leer()`),
7.) `b(F1,F2,B1,B2)` hat unendlich viele Grundspezialfälle.

Lösung zu Aufgabe-03:

```
GT(F)                              = {
   c(), m(), y()
}
GT(b(c(), F,   leer(), leer())) = {
   b(c(), c(), leer(), leer())
   b(c(), m(), leer(), leer())
   b(c(), y(), leer(), leer())
}
GT(b(m(), y(), leer(), leer())) = {
   b(m(), y(), leer(), leer())
}
```

Lösung zu Aufgabe-04:

Die Menge aller nicht-leer()-en Bäume: `b(F1, F2, B1, B2)`
Die Menge aller Bäume: `B`
Die Menge aller vollen Bäume der Tiefe 2:
`b(F1,F2,b(F3,F4,leer(),leer()),b(F5,F6,leer(),leer()))`

Lösung zu Aufgabe-05:

4.1. glückt, VB41 = `{F=c(),B=leer()}`
4.2. glückt, VB42 = `{F=c(),B=b(c(),y(),leer(),leer())}`
4.3. misslingt
4.4. glückt, VB44 = `{B5=b(c(),y(),leer(),leer())}`
4.5. glückt, VB45 = `{}` // Das ist die *leere* Variablenbelegung
4.6. misslingt

Grundlegendes über Prädikate in Gentle

Ein *Proc-Prädikat* in Gentle hat eine gewisse Ähnlichkeit mit einer *Prozedur* in konventionellen Sprachen wie Pascal, Ada oder C:

1. Ein Prädikat muss *vereinbart* werden und kann dann beliebig oft *aufgerufen* werden.

2. Ein Prädikat hat (null oder mehr) *formale in-Parameter* und (null oder mehr) *formale out-Parameter*. Jeder Parameter gehört zu einem bestimmten *Typ*.

3. Wenn man ein Prädikat aufruft, dann muss man als *aktuelle Parameter* für jeden *in-Parameter* einen *Ausdruck* des betreffenden Typs und für jeden *out-Parameter* eine *Variable* (oder ein komplizierteres *Muster*) des betreffenden Typs angeben.

4. Häufig (aber nicht immer) ruft man ein Prädikat auf, damit aus bestimmten Werten für die *in-Parameter* die entsprechenden Werte der *out-Parameter* berechnet werden.

Gleichzeitig hat ein Prädikat auch eine gewisse Ähnlichkeit mit einer *Funktion* in konventionellen Sprachen wie Pascal, Ada oder C, und zwar mit einer Funktion, die einen *Wahrheitswert* (true oder false) als Ergebnis liefert. Man sagt auch: Ein *Aufruf eines Prädikates* kann *gelingen* (das entspricht einem Funktionsergebnis *true*) bzw. *misslingen* (das entspricht einem Funktionsergebnis *false*).

Englisch: An invocation of a predicate either *succeeds* or *fails.*

Manchmal ruft man ein Prädikat nur auf, um zu sehen, ob der Aufruf gelingt oder misslingt (und nicht, um aus den Werten der in-Parameter entsprechende Werte der out-Parameter berechnen zu lassen). Das Gelingen bzw. Misslingen eines Prädikataufrufes hängt (normalerweise) von seinen aktuellen in-Parametern ab.

In Gentle unterscheidet man 5 **Arten von Prädikaten**:

```
1 Phrasen    -Prädikate (nonterm   predicates)
2 Token      -Prädikate (token     predicates)
3 Proc       -Prädikate (proc      predicates) <-- werden hier behandelt
4 Bedingungs -Prädikate (condition predicates) <-- werden hier behandlet
5 Feger      -Prädikate (sweep     predicates)
```

In diesem Abschnitt werden nur Proc- und Bedingungs-Prädikate behandelt.

Die Aufrufe eines *Proc-Prädikates* sollten immer *gelingen.* Wenn ein Aufruf misslingt, gilt das als *Fehler des Programmierers* und das betreffende Gentle-Programm wird sofort mit einer entsprechenden Fehlermeldung *abgebrochen*.

Dagegen ist das Misslingen eines Aufrufs eines *Bedingungs-Prädikates* ganz normal und entspricht in etwa der Situation, dass die Bedingung einer `if`-Anweisung den Wert `false` hat.

Indem der Programmierer ein Prädikat P als *Proc-Prädikat* vereinbart, drückt er also aus, dass (seiner Ansicht und Absicht nach) *alle* Aufrufe von P *gelingen werden*. Vereinbart er P dagegen als *Bedingungs-Prädikat*, dann gehört es zum Sinn und Zweck von P, in gewissen Fällen zu *misslingen*.

Als Beispiele werden im folgenden einige Prädikate vereinbart, mit denen man *Listen von Ganzzahlen* (Listen vom Typ `int[]`) bearbeiten kann.

Beispiel-01: Grundterme (oder: Werte) des Typs `int[]`:

```
1                                      // Eine Liste mit
2 int[]                                // null  Elementen
3 int[10]                              // einem Element
4 int[10, 20]                          // zwei  Elementen
5 int[10, 20, 30] // drei  Elementen   // drei  Elementen
6 int[10, 20, 30::int[40, 50::int[]]] // fünf  Elementen
```

Die Begriffe *Ausdruck* und *Muster* wurden im vorigen Abschnitt eingeführt. Jetzt soll damit erläutert werden, was ein Prädikat ist und was bei der Ausführung eines Prädikat-Aufrufs passiert.

Beispiel-02: Definitionen von drei Proc-Prädikaten

```
1 proc remove1(L1:int[] -> L2:int[])
```

```
 2    rule remove1(int[H::T] -> T):
 3    rule remove1(L -> L):
 4
 5 proc remove2(L1:int[] -> L2:int[])
 6    rule remove2(int[E1, E2::T] -> T):
 7    rule remove2(L -> L):
 8
 9 proc remove3(L1:int[] -> L2:int[])
10    rule remove3(L30 -> L33):
11       remove1(L30 -> L31)
12       remove2(L31 -> L33)
13    rule remove3(L -> L):
```

Die Definition eines Prädikats besteht aus
einem (Prädikaten-) *Kopf* gefolgt von einem (Prädikaten-) *Rumpf*.

Im Kopf wird der *Name* des Prädikats festgelegt, sowie die *Anzahl* der Parameter und von jedem Parameter seine *Art* (in oder out) und sein *Typ*.

Optional kann man jedem Parameter einen *Namen* geben (im **Beispiel-02**: `L1` und `L2`). Das ist besonders dann empfehlenswert, wenn es mehrere Parameter der gleich Art und des gleich Typs gibt (z.B. 3 in-Parameter vom Typ `int` oder 2 out-Parameter vom Typ `string`). Außerdem erleichtern solche Namen das Formulieren von Kommentaren. Die Parameter-Namen haben keine formale Bedeutung und somit ähneln sie Kommentaren.

Hinweis: Auf die Frage "Wie viele Parameter hat das Prädikat xxx?" wird als Antwort nicht nur *eine* Zahl erwartet, sondern eine Aussage der Form "n in-Parameter und m out-Parameter".

Der Rumpf eines Prädikats besteht normalerweise aus *Regeln*.

Eine Regel besteht aus einem (Regel-) *Kopf* und einem (Regel-) *Rumpf*.

Der Kopf muss mit `rule` beginnen und kann (muss aber nicht) mit einem Doppelpunkt `:` abgeschlossen werden.

Hinweis: Die Doppelpunkt-Regel ist vor allem für ältere Gentle-Programmierer gedacht, denen es schwerfällt, sich an die neue und einfachere Syntax von Gentle zu gewöhnen :-).

Der Rumpf einer Regel kann leer sein (wie z.B. bei beiden Regeln des Prädikats `remove1`). Oder er kann z.B. aus einer Folge von Prädikat-Aufrufen bestehen (wie bei der ersten Regel des Prädikats `remove3`). Weitere Möglichkeiten werden später behandelt.

Ausdrücke und *Muster* können sowohl im Kopf als auch im Rumpf einer Regel vorkommen. Das folgende Beispiel soll illustrieren, wo genau Ausdrücke und wo Muster stehen.

Beispiel-03: Die erste Regel des Prädikats `remove3` mit Hinweisen auf Ausdrücke und Muster

```
14            //   Muster    Ausdruck
15    rule remove3(  L30   ->  L33  ):
16            // Ausdruck    Muster
17       remove1(  L30  ->    L31 )
18       remove2(  L31  ->    L33 )
```

In Zeil 15 steht der Kopf der Regel (offenbar von einem älteren Menschen geschrieben),
in den Zeilen 17-18 steht der Regel-Rumpf. Der Rumpf enthält zwei Prädikataufrufe.

Die folgende Tabelle soll das **Beispiel-03** ergänzen:

↓ Ort Parameter-Art →	**in-Parameter**	**out-Parameter**
Im **Kopf einer Regel** (z.B. in Zeile 15)	Muster	Ausdruck
In einem **Prädikat-Aufruf** (z.B. in Zeile 17 oder 18)	Ausdruck	Muster

Im *Kopf einer Regel* muss man also für jeden in-Parameter ein *Muster* und für jeden out-Parameter einen *Ausdruck* angeben. In einem *Prädikat-Aufruf* ist es genau umgekehrt.

Aufgabe-06: Betrachten Sie noch einmal das **Beispiel-02** (oben).
Geben Sie für jede Zeile an, welche *Ausdrücke* und welche *Muster* darin vorkommen.

Es hat Sie hoffentlich nicht überrascht, dass im **Beispiel-02** einige Terme als Ausdrücke *und* als Muster verwendet werden.

Wie wird ein Prädikat-Aufruf ausgeführt?

Die Ausführung eines Prädikat-Aufrufs besteht aus einer alternierenden Folge von *Musterabgleichen* und *Auswertungen*. Das soll anhand eines Beispiels erläutert werden.

Beispiel-01: Ausführung eines Prädikat-Aufrufs

```
 1 proc remove1(L1:int[] -> L2:int[])
 2    rule remove1(int[H::T] -> T):
 3    rule remove1(L -> L):
 4
 5 proc remove2(L1:int[] -> L2:int[])
 6    rule remove2(int[E1, E2::T] -> T):
 7    rule remove2(L -> L):
 8
 9 proc remove3(L1:int[] -> L2:int[])
10    rule remove3(L30 -> L33):
11       remove1(L30 -> L31)
12       remove2(L31 -> L33)
13    rule remove3(L -> L):
14
15 root
16    anna <- int[10, 20]
17    remove3(anna -> bert)
18    ...
```

Es geht hier um den hervorgehobenen Prädikat-Aufruf in Zeile 17.

Schritt-01: Bevor der Ausführer den Aufruf ausführt, hat er in Zeile 16 eine Variable `anna` erzeugt und ihr den Wert `int[10,20]` zugeordnet. Der Ausführer hat also eine aktuelle Variablenbelegung `VB0` gleich `{anna=int[10,20]}`.

Damit beginnt er die Ausführung des Aufrufs in Zeile 17.
Zur Erinnerung: Dort ist der in-Parameter `anna` ein Ausdruck.

Schritt-02: Auswertung des Ausdrucks `anna` mit der Variablenbelegung `VB0`.
Ergebnis: Ein Wert `W0` gleich `int[10,20]`.

Mit `W0` geht der Ausführer zur Vereinbarung von `remove3` und probiert als erstes, ob ihm die Ausführung der ersten Regel (ab Zeile 10) gelingt.
Zur Erinnerung: Im Regel-Kopf in Zeile 10 ist der in-Parameter `L30` ein Muster.

Schritt-03: Musterabgleich des Wertes `W0` gleich `int[10,20]` mit dem Muster `L30.`
Ergebnis: Eine Variablenbelegung `VB1` gleich `{L30=int[10,20]}`.

Mit `VB1` beginnt der Ausführer, den Rumpf der ersten `remove3`-Regel (ab Zeile 11) auszuführen.
Zur Erinnerung: Im Aufruf in Zeile 11 ist der in-Parameter `L30` ein Ausdruck.

Schritt-04: Auswertung des Ausdrucks `L30` mit der Variablenbelegung `VB1.`
Ergebnis: Ein Wert `W1` gleich `int[10,20]`.

Mit `W1` geht der Ausführer zur Vereinbarung von `remove1` und probiert als erstes, ob ihm die Ausführung der ersten Regel (in Zeile 2) gelingt.
Sie erinnern sich: Im Regel-Kopf in Zeile 2 ist der in-Parameter `int[H::T]` ein Muster.

Schritt-05: Musterabgleich des Wertes `W1` gleich `int[10,20]` mit dem Muster `int[H::T]`.

Ergebnis: Eine Variablenbelegung VB2 gleich {H=10, T=int[20]}.

Hinweis: H ist hier ein Variable des Typs int. T ist dagegen vom Typ int[].
Eine Liste mit einem einzelnen int-Wert darin ist etwas ganz anderes als ein einzelner int-Wert.

Da die Regel in Zeile 2 keinen Rumpf hat, braucht er auch nicht ausgeführt zu werden.
Zur Erinnerung: Im Regel-Kopf in Zeile 2 ist der out-Parameter T ein Ausdruck.

Schritt-06: Auswertung des Ausdrucks T mit der Variablenbelegung VB2.
Ergebnis: Ein Wert W2 gleich int[20].

Damit ist die Ausführung der erste Regel von remove1 (in Zeile 2) gelungen und der Ausführer kehrt mit dem Wert W2 zum Aufruf von remove1 in Zeile 11 zurück.
Zur Erinnerung: Im Aufruf in Zeile 11 ist der out-Parameter L31 ein Muster.

Schritt-07: Musterabgleich des Wertes W2 gleich int[20] mit dem Muster L31.
Ergebnis: Die Variablenbelegung VB1 gleich {L30=int[10,20]} (aus **Schritt-03**) wird erweitert zu einer Variablenbelegung VB3 gleich {L30=int[10,20], L31=int[20]}.

Nachdem die Ausführung des remove1-Aufrufs in Zeile 11 gelungen ist, kann der Ausführer mit der Ausführung des remove2-Aufrufs in Zeile 12 beginnen.
Zur Erinnerung: Im remove2-Aufruf in Zeile 12 ist der in-Parameter L31 ein Ausdruck.

Schritt-08: Auswertung des Ausdrucks L31 mit der Variablenbelegung VB3.
Ergebnis: Ein Wert W3 gleich int[20].

Mit W3 geht der Ausführer zur Vereinbarung von remove2 und probiert als erstes, ob ihm die Ausführung der ersten Regel (in Zeile 6) gelingt.
Sie erinnern sich: Im Regel-Kopf in Zeile 6 ist der in-Parameter int[E1, E2::T] ein Muster.

Schritt-09: Musterabgleich des Wertes W3 gleich int[20] mit dem Muster int[E1, E2::T].
Ergebnis (wenn das das richtige Wort ist): Der Musterabgleich *misslingt*.

Damit ist auch die Ausführung des remove2-Aufrufs in Zeile 12 misslungen und
damit auch die Ausführung der ersten Regel von remove3 (begonnen in **Schritt-03**).

Der Ausführer geht deshalb mit dem Wert W0 gleich int[10,20] zurück zur Vereinbarung von remove3 und probiert, ob ihm die Ausführung der *zweiten* Regel (in Zeile 13) gelingt.
Zur Erinnerung: Im Regel-Kopf in Zeile 13 ist der in-Parameter L ein Muster.

Schritt-10: Musterabgleich des Wertes W0 mit dem Muster L.
Ergebnis: Eine Variablenbelegung VB4 gleich {L=int[10,20]}.

Da die zweite remove3-Regel (in Zeile 13) keinen Rumpf hat, braucht er auch nicht ausgeführt zu werden. Zur Erinnerung: Im Regel-Kopf in Zeile 13 ist der out-Parameter L ein Ausdruck.

Schritt-11: Auswertung des Ausdrucks L mit der Variablenbelegung VB4 gleich {L=int[10,20]}.
Ergebnis: Ein Wert W4 gleich int[10,20].

Damit ist die Ausführung der zweiten Regel von remove3 (in Zeile 13) gelungen und der Ausführer kehrt mit dem Wert W4 zum Aufruf von remove3 in Zeile 17 zurück.
Zur Erinnerung: Im Aufruf in Zeile 17 ist der out-Parameter bert ein Muster.

Schritt-11: Musterabgleich des Wertes W4 gleich int[10,20] mit dem Muster bert.
Ergebnis: Die Variablenbelegung VB0 gleich {anna=int[10,20]} (aus dem **Schritt-01**) wird erweitert zu einer Variablenbelegung VB5 gleich{anna=int[10,20], bert=int[10,20]}.

Damit hat der Ausführer den Prädikat-Aufruf remove3(anna -> bert) in Zeile 17 erfolgreich ausgeführt und kann (mit der Variablenbelegung VB5) in Zeile 18 weitermachen.

Operationen in Ausdrücken, nicht in Mustern

Neben der Möglichkeit, ganz normale Prädikate zu vereinbaren, gibt es in Gentle auch ein paar sehr merkwürdige und exotische Gebilde namens `+`, `-`, `*` und `/`, die man auch als *vordefinierte Operationen* bezeichnet. Diese Operationen kann man (ähnlich wie Prädikate) *aufrufen.* Allerdings muss man dazu eine sehr unnatürliche Notation verwenden, die aus dem Mittelalter stammt und als *Infixnotation* bezeichnet wird. Beispiele für diese Infixnotation: `1 + 1`, `A - B - C`, `A + 17 * B / 5` etc.

Der exotische Charakter dieser sogenannten Operationen wird noch dadurch verschärft, dass der Name `-` (minus) zwei verschiedene Operationen bezeichnet: Eine einstellige Operation ("negatives Vorzeichen") und eine zweistellige Operation ("Subtraktion zweier Ganzzahlen"). Ganz entsprechendes gilt für den Namen `+` (plus).

Die sechs sogenannten Operationen mit den vier Namen `+`, `-`, `*` und `/` sind gedacht als Ersatz für die (ganz normalen und leicht verständlichen) Prädikate

```
1 proc Plus (Summand1: int, Summand2:   int -> Summe:     int)
2 proc Minus(Minunend: int, Subtrahend: int -> Differenz: int)
3 proc Mal  (Faktor1:  int, Faktor2:    int -> Produkt:   int)
4 proc Durch(Dividend: int, Divisor:    int -> Quotient:  int)
5 proc Pos  (Argument: int                  -> Ergebnis:  int)
6 proc Neg  (Argument: int                  -> Ergebnis:  int)
```

mit denen man zwei Ganzzahlen vom Typ `int` addieren, subtrahieren, multiplizieren, dividieren bzw. das Vorzeichen einer Ganzzahl unverändert lassen (`Pos`) oder umdrehen (`Neg`) kann.

Anmerkung: Es soll Programmierer geben, die *Operationen* mit ihren zweideutigen Namen und ihrer schwer lesbaren Infixnotation für einfacher und natürlicher halten, als *Prädikate* (mit ihren eindeutigen Namen und ihrer leicht lesbaren Präfixnotation). Vermutlich ist eine solche Vorliebe für Operationen auf eine harte Jugend zurückzuführen, in der nur Fortran, C oder Java gesprochen wurde.

Operationssaufrufe sind nur in Termen der vordefinierten Typen `int` und `string` möglich. Terme mit Operationsaufrufen darin dürfen nur als *Ausdrücke* verwendet werden, aber *nicht* als *Muster*.

Beispiel: Operationsaufrufe sind nur in Ausdrücken erlaubt, aber nicht in Mustern:

```
 1 proc  Minus1(int -> int)
 2    rule Minus1(N + 1 -> N    ):    // falsch:  Operationsaufruf in Muster
 3    rule Minus1(N     -> N - 1):    // erlaubt: Operationsaufruf in Ausdruck
 4
 5 proc  Wurzel(int -> int)
 6    rule Wurzel(N * N -> N):        // falsch: Operationsaufruf in Muster
 7
 8 proc  Quadrat(int -> int)
 9    rule Quadrat(N -> N * N):       // erlaubt: Operationsaufruf in Ausdruck
10
11 proc  MachWasWenn17(int ->)
12    rule MachWasWenn17(17  ->):     // erlaubt: Literal als Muster
13       print "Param ist 17!"
14
15 proc  MachWasWennXY(string ->)
16    rule MachWasWennXY("XY" ->):    // erlaubt: Literal als Muster
17       print "Param ist XY!"
```