Typen in Gentle

1. Vordefinierte Typen

In Gentle gibt es zwei *vordefinierte Typen*: `int` und `string`. Die entsprechen weitgehend den Typen `int` und `String` in Java und anderen Sprachen.

Beispiel-01: Literale der Typen `int` und `string`

`int`-Literale: `0, 1, 123`
`string`-Literale: `"Hallo!", "abc\n", "x", ""`

2. Vom Programmierer definierte Typen

Der Programmierer kann weitere Typen vereinbaren. Hier ein paar Beispiele :

Beispiel-02: Zwei **Aufzählungstypen** (enumeration types):

```
1 type Tag mo() di() mi() do() fr() sa() so()
2 type Op2 add() sub() mul() div()
```

Zum Typ `Op2` gehören 4 Werte. Jeder Wert wird von einem der *Konstruktoren* `add`, `sub`, `mul`, `div` repräsentiert. Bei einem Aufzählungstyp haben alle Konstruktoren null (d.h. keine) Parameter. Konstruktoren werden in Gentle *immer* mit einem Paar runder Klammern `()` dahinter notiert, auch wenn sie keine Parameter haben.

Beispiel-03: Ein **Verbundtyp** (record type, struct type)

```
3 type Person
4    p(NachName:string, VorName:string, JahresEinkommenInDollar:int)
```

Der Konstruktor `p` hat hier 3 Parameter. `NachName`, `VorName` und `JahresEinkommen` sind optionale *Bezeichner* für die Parameter. Sie haben den Charakter von Kommentaren und keine weitere Bedeutung, müssen aber den Regeln für Identifier entsprechen: `[A-Za-z][A-Za-z0-9_]*`.

Beispiel-04: Ein **Verbundtyp mit Varianten** (variant record type, union type)

```
5 type Eintrag
6    e1(VarName:string, ZielBezeichner:string)
7    e2(KonstName:string, Wert:int)
```

Für jede Variante muss man einen eigenen Konstruktor festlegen (hier: `e1` und `e2`).

Beispiel-05: Ein **rekursiver Typ**:

```
 8 type Ausdruck
 9    plus2 (Ausdruck, Ausdruck)
10    minus2(Ausdruck, Ausdruck)
11    minus1(Ausdruck)
12    i(int)
```

Der Typ `Ausdruck` ist *rekursiv*, denn er kommt in seiner eigenen Definition vor (in den Varianten mit den Konstruktoren `plus2`, `minus2` und `minus1`).

In einer rekursiven Typdefinition muss es mindestens *eine* Variante geben, die *nicht rekursiv* ist. Beim Typ `Ausdruck` ist das die Variante mit dem Funktor `i`, die übrigen 3 Varianten sind *rekursiv*.

Regel: Ein Konstruktor (wie z.B. `add` oder `minus2`) darf in höchstens *einer* Typ-Definition und dort nur *einmal* vorkommen.

Deshalb wurden in der Definition des Typs `Ausdruck` die Konstruktoren für das einstellige und das zweistellige Minus `minus1` und `minus2` genannt und der Konstruktor für die Addition heißt `plus2` und nicht `add`, weil `add` schon ein Konstruktor des Typs `Op2` ist.

Aufgabe-01: Geben Sie von jedem der folgenden Typen 3 Werte an:
`int, string, Tag, Op2, Eintrag, Ausdruck.`

Aufgabe-02: Die Werte des Typs `Ausdruck` sind (oder: repräsentieren) *arithmetische Ausdrücke*, die aus `int`-Literalen, einer einstelligen Minus-Operation, einer zweistelligen Minus-Operation und einer zweistelligen Plus-Operation bestehen. Vereinbaren Sie einen Typ `Ausdruck2`, dessen Werte arithmetische Ausdrücke sind (oder: repräsentieren), die aus `int`-Literalen und zweistelligen Operatoren für die Addition, Subtraktion, Multiplikation und Division bestehen und bei denen die Operatoren durch Werte des Typs `Op2` dargestellt werden.

3. Algebraische und nicht-algebraische Typen

In Gentle sind die beiden vordefinierten Typen `int` und `string` *nicht-algebraische* Typen. Dagegen sind alle vom Programmierer vereinbarten Typen *algebraische* Typen.

Jeder Wert eines algebraischen Typs besteht aus *Konstruktoren* und ihren *Parametern* und man kann den Wert *auf eindeutige Weise* in seine Konstruktoren und ihre Parameter zerlegen.

Beispiel-06: Der Wert `plus2(i(5), minus1(i(3)))` des Typs `Ausdruck` besteht aus dem Konstruktor `plus2` und seinen Parametern `i(5)` und `minus1(i(3))`. Und diese Parameter kann man ebenso in Konstruktoren und ihre Parameter zerlegen, und diese Parameter ebenso, ... etc., bis die Parameter vom Typ `int` oder vom Typ `string` sind oder ein Konstruktor keine Parameter hat.

Einen `int`-Wert wie 5 kann man mit verschiedenen Operationen erzeugen, z.B. so: `2+3` oder so: `1+4` oder so `2+4-1` etc. Aber dem fertigen Wert 5 sieht man nicht mehr an, wie er erzeugt wurde und man kann ihn nicht eindeutig in Konstruktoren und ihre Parameter zerlegen. Deshalb ist der Typ `int` nicht-algebraisch. Ganz Entsprechendes gilt für den Typ `string`.

4. Variablen vereinbaren und initialisieren

Im `root`-Prädikat und auf den rechten Seiten der Regeln anderer Prädikate kann man Variablen vereinbaren und initialisieren, z.B. wie folgt:

```
                                    // Typ der Variablen?
13 otto <- 17                       //
14 emil <- otto + 4                 //
15 paul <- p("Meyer", "Paul", 37500) //
16 erna <- e1("x3", "y3")           //
17 egon <- e2("c5", 25)             //
18 anna <- plus2(i(3), i(otto))     //
```

Tragen Sie hinter jeder Vereinbarung den Typ der Variablen als Kommentar ein.

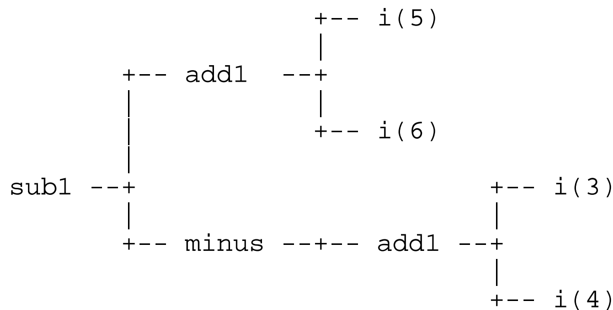
Merke: Den Wert einer solchen Variablen kann man *nicht* verändern.

5. Musterabgleiche (engl. pattern matching)

Musterabgleich	**gelingt?**	**Variablen-Belegung**
`otto -> 18`	nein	
`otto -> 17`	ja	`{}`
`paul -> X`	ja	`{X=p("Meyer", "Paul", 37500)}`
`paul -> p("Schulz", VN, DPJ)`	nein	
`paul -> p("Meyer", VN, DPJ)`	ja	`{VN="Paul", DPJ=37500}`
`paul -> p(NN, VN, _)`	ja	`{NN="Meyer", VN="Paul"}`
`anna -> minus2(A, B)`	nein	
`anna -> plus2(A, B)`	ja	`{A=i(3), B=i(17)}`

Die Werte der beiden Typen `Ausdruck` und `Ausdruck2` sind arithmetische Ausdrücke, Operatoren werden aber auf unterschiedliche Weisen dargestellt: Beim Typ `Ausdruck` durch *Konstruktoren* (`add1`, `sub1`, `minus`) des Typs `Ausdruck` , beim Typ `Ausdruck2` dagegen durch *Werte* eines anderen Typs `Op2`. Beide Darstellungen haben Vor- und Nachteile.

Die Werte der beiden rekursiven Typen sind *Bäume*. Z.B. entspricht der "linear notierte Wert" `sub1(add1(i(5), i(6)), minus(add1(i(3), i(4)))` direkt dem folgenden Baum:



```
                  +-- i(5)
                  |
      +-- add1  --+
      |           |
      |           +-- i(6)
      |
sub1 --+                     +-- i(3)
      |                      |
      +-- minus --+-- add1 --+
                             |
                             +-- i(4)
```

Die vom Programmierer vereinbarten Typen sind sogenannte *algebraische Typen*. Mit ihren Werten kann man *Musterabgleiche* durchführen. Die vordefinierten Typen `int` und `string` sind *keine* algebraischen Typen. Mit ihren Werten sind nur ganz simple Musterabgleiche möglich.

6. Einstufige Listen-Typen

Besonders häufig benötigt man *Listen-Typen*, z.B. Listen von `int`-Werten, Listen von `Eintrag`-Werten etc. Solche Listen-Typen kann man selbst als rekursive Typen definieren, ganz ähnlich wie die Baum-Typen `Ausdruck` und `Ausdruck2` im vorigen Abschnitt. Aber weil sie so häufig benötigt werden, gibt es in Gentle spezielle Notationen, die eine Vereinbarung unnötig machen.

Grundregel: Zu jedem Typ `T` gibt es (automatisch, ohne Vereinbarung) einen Typ `T[]` (gesprochen: *Liste von T-Werten* oder kürzer: *Liste von T*).

Beispiel-01: Namen von Listen-Typen

`int[]`, `string[]`, `Eintrag[]`, `Eintrag2[]`, `Ausdruck[]`

In Gentle kann man die Werte aller Typen durch *Literale* bezeichnen, auch die Werte von Listen-Typen. Im folgenden Beispiel werden Listen-Literale verwendet.

Beispiel-02: Variablen einen Wert eines Listen-Typs zuordnen

```
1  L11 <- int[10, 20, 30]
2  L12 <- int[]
3  L13 <- Eintrag[e("anna", 17), e("bert", 25)]
4  L14 <- Eintrag[]
```

Die Variablen `L11` und `L12` sind vom Typ `int[]`, `L13` und `L14` sind vom Typ `Eintrag[]`.

Merke-01: `T[]` ist nicht nur der Name eines Typs (Liste von `T`-Werten), sondern gleichzeitig auch eine Notation für die *leere Liste* dieses Typs (wie in den Zeilen 2 und 4).

Merke-02: Ein Listen-Literal wie z.B. `int[10,20,30]` besteht aus dem Typ der Elemente (hier: `int`), gefolgt von den Elementen in eckigen Klammern (hier: `[10,20,30]`). Dieser Regel werden wir im nächsten Abschnitt nochmal begegnen.

7. Musterabgleiche (engl. pattern matching)

Auch Listen-Werte kann man mit entsprechenden *Mustern* abgleichen, um die Werte zu prüfen und auf ihre Teile zuzugreifen.

Beispiel-03: Musterabgleiche mit Listen-Mustern

Angenommen, `L15` ist eine Variable vom Typ `int[]`. Dann können wir ihren Inhalt durch Musterabgleiche prüfen und auf Teile des Inhalts zugreifen. Es folgen ein paar Beispiele:

Musterabgleich	Was muss L15 enthalten, damit der Abgleich gelingt?	Nach gelungenem Abgleich: Welche Variable hat welchen Wert?
L15 -> int[]	int[]	--
L15 -> int[10,20,30]	int[10,20,30]	--
L15 -> int[N1,N2]	Eine Liste mit genau 2 Elementen	**N1**: 1. Element von L15 **N2**: 2. Element von L15
L15 -> int[N1::R1]	Eine Liste mit mindestens einem Element	**N1**: 1. Element von L15 **R1**: Rest der Liste L15
L15 -> int[10::R1]	Eine Liste mit 10 als 1. Element	**R1**: Rest der Liste L15
L15 -> int[N1,N2::R2]	Eine Liste mit mindestens 2 Elementen	**N1**: 1. Element von L15 **N2**: 2. Element von L15 **R2**: Rest der Liste L15
L15 -> L ~ int[N1::R1]	Eine Liste mit mindestens einem Element	**L** : Die ganze Liste L15 **N1**: 1. Element von L15 **R1**: Rest der Liste L15

Ausführbare Beispiele für (gelingende und misslingende) Musterabgleiche mit Listen vom Typ `int[]` findet man in der Datei `projects\pred03\patternMatching.g`.

8. Mehrstufige Listen

Die Grundregel, dass es zu jedem Typ `T` einen Typ `T[]` gibt, ist *rekursiv* gemeint:
Zum Typ `T[]` gibt es einen Typ `T[][]` (Liste von Listen von T)
und zu `T[][]` gibt es den Typ `T[][][]` (Liste von Listen von Listen von T) etc.

Beispiel-01: Namen von 2-stufigen Listen-Typen

`int[][]`, `string[][]`, `Eintrag[][]`, `Eintrag2[][]`, `Ausdruck[][]`

Auch 2-stufige Listen kann man durch Literale bezeichnen:

Beispiel-02: Variablen einen Wert eines 2-stufigen Listen-Typs zuordnen

```
1  L21 <- int[][int[10, 20, 30], int[40], int[50, 60]]
2  L22 <- int[][]
3  L23 <- Eintrag[][Eintrag[e("anna", 17), e("bert", 25)], Eintrag[]]
4  L24 <- Eintrag[][]
```

Merke-01: `T[][]` ist nicht nur der Name eines Typs (Liste von Listen von von `T`-Werten), sondern gleichzeitig auch eine Notation für die *leere Liste* dieses Typs (wie in den Zeilen 2 und 4).

Merke-02: Ein Listen-Literal wie z.B. `int[][int[10,20],int[30,40]]` besteht aus dem Typ der Elemente (hier: `int[]`), gefolgt von den Elementen in eckigen Klammern (hier: `[int[10,20],int[30,40]]`). Dieser Regel sind wir im vorigen Abschnitt schon mal begegnet

Natürlich kann man auch mehrstufige Listen-Werte mit entsprechenden Mustern abgleichen, um die Werte zu prüfen und auf ihre Teile zuzugreifen.

Beispiel-03: Musterabgleiche mit 2-stufigen Listen-Mustern

Angenommen, `L25` ist eine Variable vom Typ `int[][]`. Dann können wir ihren Inhalt unter anderem durch die folgenden Musterabgleiche prüfen und auf Teile des Inhalts zugreifen:

Musterabgleich	Was muss L25 enthalten, damit der Abgleich gelingt?	Nach gelungenem Abgleich: Welche Variable hat welchen Wert?
L25 -> int[][]	int[][]	--
L25 -> int[][int[10,20], int[30,40,50]]	int[][int[10,20], int[30,40,50]]	--
L25 -> int[][L1,L2]	Eine Liste mit genau 2 Elementen	**L1**: 1. Element von L25 **L2**: 2. Element von L25
L25 -> int[][L1::R1]	Eine Liste mit mindestens einem Element	**N1**: 1. Element von L25 **R1**: Rest der Liste L25
L25 -> int[][int[10,20]::R1]	Eine Liste mit int[10,20] als 1. Element	**R1**: Rest der Liste L25
L25 -> int[][L1,L2::R2]	Eine Liste mit mindestens 2 Elementen	**L1**: 1. Element von L25 **L2**: 2. Element von L25 **R2**: Rest der Liste L25
L25 -> L ~ int[][L1::R1]	Eine Liste mit mindestens einem Element	**L** : Die ganze Liste L25 **L1**: 1. Element von L25 **R1**: Rest der Liste L25

Die Elemente von `L25` sind hier natürlich Listen vom Typ `int[]` und der jeweilige Rest von `L25` ist vom Typ `int[][]`.

Ausführbare Beispiele für (gelingende und misslingende) Musterabgleiche mit Listen vom Typ `int[][]` findet man in der Datei `projects\pred03\patternMatching.g`.