Lösungen zu den Aufgaben im Skript "Ada95":

Lösung 4.1.1.:
1. Erzeuge eine Variable namens **G1** vom Untertyp **integer** mit dem Anfangswert 17.
2. Erzeuge eine Variable namens **G2** vom Untertyp **integer**.

Lösung 4.1.2.:
1. Erzeuge eine Variable namens **CLAUDIA** vom Untertyp **character** mit dem Anfangswert **'A'**.
2. Erzeuge eine Variable namens **CLAUS** vom Untertyp **character**.

Lösung 4.2.1.:
1. Berechne den Wert des Ausdrucks **G1 + G2 + 8**. Ergebnis: Der Wert 20.
2. Berechne den Wert des Ausdrucks **10 + G1 + 3**. Ergebnis: Der Wert 30.

Lösung 4.2.2.:
1. Erzeuge eine Variable namens **G3** vom Untertyp **integer** mit dem Anfangswert **+3**.
2. Berechne den Wert des Ausdrucks **G3 * 5** und erzeuge eine Variable namens **G4** vom Untertyp **integer** mit diesem Wert als Anfangswert.
3. Erzeuge eine Variable namens **G5** vom Untertyp **boolean**.
4. Erzeuge eine Variable namens **B2** vom Untertyp **boolean** mit dem Anfangswert **false**.
5. Berechne den Wert des Ausdrucks **not B2** und erzeuge eine Variable namens **B3** vom Untertyp **boolean** mit diesem Wert als Anfangswert.
6. Erzeuge eine Variable namens **B4** vom Untertyp **boolean**.
7. Berechne den Wert des Ausdrucks **(G3 < G4)** und erzeuge eine Variablen namens **B5** vom Untertyp **boolean** mit diesem Wert als Anfangswert.

Lösung 4.3.1.: Ein **goto**-Befehl verändert den sogenannten **Befehlszähler** (program counter). Das ist eine "interne Variable des Ausführers", mit der er sich merkt, wo in einem Programm er gerade Befehle ausführt.

Lösung 6.5.1.: Siehe Datei **HALLO_02.adb**.

Lösung 6.5.2.:
Zeile 01: Binde das Paket **ada.text_io** in jedes Programm ein, in das du auch die folgende Programmeinheit (d.h. die Prozedur **HALLO_02**) einbindest.
Zeile 02 bis 16: Erzeuge eine Prozedur namens **HALLO_02**, die aus den folgenden sieben Befehlen besteht:
Zeile 08: Erzeuge eine Variable namens ZEICHEN1 vom Untertyp character.
Zeile 09: Erzeuge eine Variable namens ZEICHEN2 vom Untertyp character.
Zeile 11: Führe die Prozedur **ada.text_io.put** mit der Zeichenkette **"Bitte geben Sie zwei Zeichen ein: "** als **item**-Parameter aus.
Zeile 12: Führe die Prozedur **ada.text_io.get** mit der Variablen **ZEICHEN1** als **item**-Parameter aus.
Zeile 13: Führe die Prozedur **ada.text_io.get** mit der Variablen **ZEICHEN2** als **item**-Parameter aus.
Zeile 14: Berechne den Wert des Ausdrucks **ZEICHEN2 & ZEICHEN1 & ZEICHEN2 & ZEICHEN1** und führe die Prozedur **ada.text_io.put** mit diesem Wert als **item**-Parameter aus.
Zeile 15: Führe die Prozedur **ada.text_io.new_line** aus.

Aufgabe 6.5.3.: Keine Musterlösung.
Lösung 6.7.1.: Das Programm **HALLO_10** gibt folgende Zeilen zum Bildschirm aus:
Hier ist HALLO_10, Stelle 1
Hier ist HALLO_11!
Hier ist HALLO_10, Stelle 2
Hier ist HALLO_11!
Hier ist HALLO_10, Stelle 3
Hier ist HALLO_12!
Hier ist HALLO_10, Stelle 4

Lösung 6.7.2.: Siehe die drei Datei **HALLO_20.adb**, **HALLO_21.adb** und **HALLO_22.adb**.

Lösung 7.1.1.: **Vorteil**: Der Programmierer weiß von jedem Ganzzahltyp genau, welche Zahlen dazugehören und kann somit **portable** Programme schreiben. **Nachteil**: Einige Ausführer müssen mit Hardwareformaten rechnen, mit denen sie nicht besonders gut umgehen können.

Lösung 7.1.2.: **Nachteil**: Der Programmierer weiß von den einzelnen Ganzzahltypen nicht, welche Zahlen dazugehören und kann somit **keine portablen** Programme schreiben. **Vorteil**: Jeder Ausführer muß nur mit Hardwareformaten rechnen, mit denen er besonders gut umgehen kann.

Lösung 7.1.3.: Billige Chips werden möglicherweise in **entsprechend größeren Stückzahlen** verkauft und eingesetzt. Heute bekommt man für eine Million Dollar z.B. 1000 Chips, die je 1000 Dollar kosten. In Zukunft bekommt man für eine Million Ecu vielleicht eine Million Chips, die je einen Ecu kosten. Aber wenn man durch geschickte Programmierung jeweils die Anzahl der benötigten Chips **halbieren** kann (von 1000 auf 500 bzw. von 1_000_000 auf 500_000) spart man ungefähr den gleichen Betrag. Möglicherweise wird Effizienz bei massenhaft eingesetzten Chips auch in Zukunft noch eine wichtige Rolle spielen.

Lösung 7.3.1.:

Eingaben für GANZT_01	**Ergebnisse** von GANZT_01
`5  3`	8
`-17  +34`	17
`+500_000_000  +500_000_000`	1000000000
`-200_000_000  -300_000_000`	-500000000
`0  1_000_000_000`	1000000000
`1  1_000_000_000`	constraint_error
`-5_000_000  -1`	constraint_error
`-5_000_001`	data_error
`+1_000_000_001`	data_error
`ABC`	data_error
`17_`	data_error
`_17`	data_error
`1___7`	data_error
`18  0`	18
`2#10010#  0`	18
`3#200#  0`	18
`4#102#  0`	18
`7#24#  0`	18
`8#22#  0`	18
`10#18#  0`	18
`12#16#  0`	18
`16#12#  0`	18
`17#11#  0`	data_error
`1#00#  0`	data_error

Lösung 7.3.2.: Viele Ausführer (z.B. der ObjektAda-Compiler Version 7.0 von Aonix) erlauben für Ganzzahltypen maximal den Bereich **-2*31..+2**31-1**. Andere Ausführer (z.B. der Gnat-Compiler Version 3.10p für Windows95 und NT) erlauben für Ganzzahltypen maximal den Bereich **-2**61..+2**61-1**.

Aufgabe 7.3.3.: Keine Musterlösung.
Lösung 7.3.4.: Siehe Datei **GANZT_02.adb**.
Lösung 8.3.1.: Siehe Datei **GANZT_05.adb**.
Lösung 8.3.2.: Siehe Datei **GANZT_06.adb**.

Lösung 9.2.1.: Zum Untertyp **OTTO'base'base** gehören die gleichen Werte wie zum Untertyp **OTTO'base**. Entsprechendes gilt auch für die Untertypen **OTTO'base'base'base**, **OTTO'base'base'base'base** etc..

Lösung 9.2.2.: Siehe Datei **GANZT_05.adb**.

Lösung 9.6.1.: Die Ausdrücke **0 rem G1** und **0 mod G1** haben den Wert **0** ("0 geteilt durch 17 ist gleich 0 Rest 0"). Die Ausdrücke **G1 rem 0** und **G1 mod 0** lösen die Ausnahme **constraint_error** aus ("weil man nicht durch 0 teilen darf").

Lösung 9.6.2.: Siehe Datei **GANZT_11.adb**.
Lösung 10.1.1.: Siehe Datei **AUFZT_02.adb**.

Lösung 10.2.1.: Zum Untertyp **ABC1** gehören drei Werte, die mit den **Zeichenliteralen** 'A', 'B' und 'C' bezeichnet werden können. Zum Untertyp **ABC2** gehören drei Werte, die mit den normalen **Namen** A, B und C bezeichnet werden können. Der Untertyp **ABC1** ist ein **Zeichenuntertyp**, der Untertyp **ABC2** ist ein **gewöhnlicher Aufzählungsuntertyp**.

Lösung 10.3.1.: Wenn **B1** den Wert **true**, **B2** den Wert **true** und **B3** den Wert **false** hat, dann haben die folgenden Ausdrücke die folgenden Werte:

Ausdruck	Wert
B1 and B2 and B3	false
not B1 or B2	true
not (B1 or B2)	false
not B1 or not B2	false

Ausdruck	Wert
B1 or B2 or not B3	true
B1 xor B2 xor B3	false
not B2 xor not B3	true
(B1or B2) xor (B2 and B3)	true

Lösung 10.3.2.:

Wert der Variablen B1	Wert der Variablen B2	Wert des Ausdrucks B1 /= B2	Wert des Ausdrucks B1 xor B2
true	true	**false**	**false**
true	false	**true**	**true**
false	true	**true**	**true**
false	false	**false**	**false**

Diese Aufgabe soll deutlich machen, daß **"/="** und **"xor"** verschiedene Namen für dieselbe Funktion sind.

Lösung 10.3.3.: Nach Ausführung aller Zuweisungen haben die die folgenden Variablen die folgenden Werte: G1: 10, G2: 10, G3: 240, B1: false, B2: true und B3: true.

Lösung 10.4.1.:
1. Die Variable **default_width** gehört zum Untertyp **field**.
2. Der Ganzzahluntertyp **field** ist am Anfang des Pakets **ada.text_io** als Untertyp des Typs !integer vereinbart. Sein erster Wert (**field'first**) ist gleich 0, **field'last** ist von Ada-Ausführer zu Ada-Ausführer verschieden.
3. Die Variable **default_setting** gehört zum Untertyp **type_set**.
4. Der Aufzählungsuntertyp **type_set** ist am Anfang des Pakets **ada.text_io** vereinbart mit den beiden Werten **lower_case** und **upper_case**.

Lösung 10.4.2.: Siehe Datei **AUFZT_05.adb**.

Lösung 12.1.1.:
10 Verändert wird die aktuelle Ausgabe (d.h. der Bildschirmbehälter).
11 Verändert werden die aktuelle Eingabe (d.h. der Tastaturbehälter) und die Variable NACH.
12 Verändert wird die aktuelle Ausgabe (der Teil des Bilschirmbehälters, in dem die Zeilen- und Spaltennummer des Bildschirmzeigers stehen).

Lösung 12.3.1.:

Bei der **if-then**-Varianten	wird **höchstens**	eine Anweisungsfolge ausgeführt.
Bei der **if-then-else**-Varianten	wird **genau**	eine Anweisungsfolge ausgeführt.
Bei der **if-then-elsif**-Varianten	wird **höchstens**	eine Anweisungsfolge ausgeführt.
Bei der **if-then-elsif-else**-Varianten	wird **genau**	eine Anweisungsfolge ausgeführt.

Lösung 12.3.2.: Siehe Datei **IFANW_03.adb**.
Lösung 12.3.3.: Siehe Datei **IFANW_04.adb**.
Lösung 12.3.4.: Siehe Datei **IFANW_06.adb**.

Lösung 12.3.5.:
Übersetzung von 12.3.1.: Berechne den Wert des Ausdrucks **G1 < G2**. Wenn dieser Wert gleich **true** ist, dann führe die Anweisungsfolge in Zeile 11 bis 12 aus.
Übersetzung von 12.3.2.: Berechne den Wert des Ausdrucks **G1 < G2**. Wenn dieser Wert gleich **true** ist, dann führe die Anweisungsfolge in Zeile 21 bis 22 aus. Sonst führe die Anweisung in Zeile 24 aus.
Übersetzung von 12.3.3.: Berechne den Wert des Ausdrucks **G1 < G2**. Wenn dieser Wert gleich **true** ist, dann führe die Anweisungsfolge in Zeile 31 bis 32 aus.
Sonst berechne den Wert des Ausdrucks **G1 = G2**. Wenn der Wert dieses Ausdrucks gleich **true** ist, führe die Anweusingsfolge in Zeile 34 bis 35 aus.
Sonst berechne den Wert des Ausdrucks **G3 > G1 + G2**. Wenn der Wert dieses Ausdrucks gleich **true** ist, führe die Anweisung in Zeile 37 aus.

Lösung 12.4.1.:
Die case-Anweisung in Zeile 20 bis 25 ist **mehrdeutig** (für **ROT** gibt es zwei Alternativen).
Die case-Anweisung in Zeile 27 bis 32 ist **unvollständig** (für **ROT** gibt es keine Alternative).
Die case-Anweisung in Zeile 34 bis 39 ist falsch, weil die **Wahleinträge** in Zeile 35 und 37 **nicht statisch** sind (sie enthalten die **Variable** FARBE2, das ist nicht erlaubt).

Lösung 12.4.2.: Übersetzung einer **case**-Anweisung in eine **if**-Anweisung:

```
10 if FARBE1 = SCHWARZ then
11    ada.text_io.put(item => "Zu dunktel!");
12    FARBE1 := GELB;
13 elsif FARBE1 = ROT or FARBE1 = GRUEN then
14    ada.text_io.put(item => "Nicht modisch genug!");
15 elsif BLAU <= FARBE1 and FARBE1 <= WEISS then
16    ada.text_io.put(item => "Schon besser!");
17    G1 := G2 + G3;
18 end if;
```

Lösung 12.4.3.: Siehe Datei **CASE_02.adb**.

Lösung 12.4.4: Übersetzungen von **case**-Anweisungen ins Deutsche

Übersetzung des Beispiels 12.4.1.: Berechne den Wert des Ausdrucks **FARBE1**.
Wenn der Wert dieses Ausdrucks gleich **SCHWARZ** ist, dann führe die Anweisungsfolge in Zeile 12 bis 13 aus.
Wenn der Wert des Ausdrucks gleich **ROT** oder **GRUEN** ist, dann führe die Anweisung in Zeile 15 aus.
Wenn der Wert des Ausdrucks im Bereich **BLAU** bis **WEISS** liegt, führe die Anweisungsfolge in Zeile 17 bis 18 aus.

Übersetzung des Beispiels 12.4.2.: Berechne den Wert des Ausdrucks **G1**.
Wenn dieser Wert im Bereich **OTTO'first** bis **-10_000** liegt oder gleich **-5_000** oder gleich **-2_500** ist, dann führe die Anweisung in Zeile 42 aus.

Wenn der Wert im Bereich **-100** bis **-20** oder im Bereich **-20** bis **+100** liegt, dann führe die Anweisung in Zeile 44 aus.
Wenn der Wert gleich **2** oder **3** oder **5** oder **7** oder **13** oder gleich **17** ist, dann führe die Anweisung in Zeile 46 aus.
Wenn der Ausdruck **G1** einen **anderen** Wert hat (d.h. wenn der Ausdruck einen Wert hat, auf den keine der bisher genannten Bedingungen zutrifft), dann führe die Anweisung in Zeile 48 aus.

Lösung 12.5.1.: Die Anweisungsfolge zwischen **loop** und **exit** (Zeile 11 bis 12) wird mindestens **einmal** ausgeführt.
Die Anweisungsfolge zwischen **exit** und **end loop** (Zeile 14 bis 15) wird mindestens **nullmal** ausgeführt.

Lösung 12.5.2.: Die Schleife im **Beispiel 12.5.2.** besteht aus **fünf** Teilen, nämlich aus **drei** Anweisungsfolgen und **zwei** Ausdrücken.

Erste	**Anweisungsfolge**:	Die **eine** Anweisung in Zeile 21.
Erster	**Ausdruck**:	**G1 = 0** in Zeile 22.
Zweite	**Anweisungsfolge**:	Die **eine** Anweisung in Zeile 23.
Zweiter	**Ausdruck**:	**G2 = 0** in Zeile 24.
Dritte	**Anweisungsfolge**:	Die **eine** Anweisung in Zeile 25.

Übersetzung ins Deutsche:
1. Führe die erste Anweisungsfolge aus.
2. Wenn die erste Bedingung den Wert **true** hat, dann beende die Ausführung der Schleife.
3. Sonst führe die zweite Anweisungsfolge aus.
4. Wenn die zweite Bedingung den Wert **true** hat, dann beende die Ausführung der Schleife.
5. Sonst führe die dritte Anweisungsfolge aus und mache bei 1. weiter.

Lösung 12.5.3.:

Anfangs in G1	Zum Schluß in G2
0	0
1	0
2	1
3	1
4	2

Anfangs in G1	Zum Schluß in G2
5	2
10	3
50	5
64	6
128	7

Anfangs in G1	Zum Schluß in G2
-1	0
-137	0

2. Die **Schleife** im **Beispiel 12.5.3.** (Zeile 32 bis 36) besteht im wesentlichen aus einem **Ausdruck** (in Zeile 33: **G1 <= 1**) und einer **Anweisungsfolge** (in den Zeilen 34 bis 35).
3. Übersetzung der Schleifenanweisung ins Deutsche:
3.1. Wenn der Ausdruck **G1 <= 1** den Wert **true** hat, dann Beende die Ausführung der Schleife.
3.2. Sonst führe die Anweisungsfolge in Zeile 34 bis 35 aus.
3.3. Mache mit 3.1. weiter.

Lösung 12.5.4.:

Anfangs in G1	Zum Schluß in G2
0	1
1	1
2	2
3	2
4	3

Anfangs in G1	Zum Schluß in G2
5	3
10	4
50	6
64	7
128	8

Anfangs in G1	Zum Schluß in G2
-1	1
-137	1

2. Die **Schleife** im **Beispiel 12.5.4.** (Zeile 32 bis 36) besteht im wesentlichen aus einer **Anweisungsfolge** (in den Zeilen 43 bis 44) und aus einem **Ausdruck** (in der Zeile 45:
G1 = 0).
3. Übersetzung der Schleifenanweisung ins Deutsche:
3.1. Führe die Anweisungsfolge in Zeile 43 bis 44 aus.

3.2. Wenn der Ausdruck **G1 = 0** den Wert **true** hat, dann beende die Ausführung der Schleife.
3.3. Sonst mache bei 3.1. weiter.

Lösung 12.7.1.: Ein Pentium-Prozessor mit 200 Megahertz kann den Schleifenrumpf pro Sekunde etwa 1_000_000 mal ausführen. Um den Schleifenrumpf 1_000_000_000 mal auszuführen, braucht er also Zeit in der Größenordnung von 1_000 Sekunden (ca. 15 Minuten).

Lösung 12.7.2.:

```
10 procedure FUEHRE_OFT_AUS is
11    G1 : integer := 0;
12    ------------------------------------------------------------------
13    procedure P1 is
14    begin
15       G1:=G1+1; G1:=G1+1; G1:=G1+1; G1:=G1+1; G1:=G1+1;
16       G1:=G1+1; G1:=G1+1; G1:=G1+1; G1:=G1+1; G1:=G1+1;
17    end P10;
18    ------------------------------------------------------------------
19    procedure P2 is
20    begin
21       P1; P1; P1; P1; P1; P1; P1; P1; P1; P1;
22    end P2;
23    ------------------------------------------------------------------
24    procedure P3 is
25    begin
26       P2; P2; P2; P2; P2; P2; P2; P2; P2; P2;
27    end P3;
28    ------------------------------------------------------------------
...    ...
53    ------------------------------------------------------------------
54    procedure P9 is
55    begin
56       P8; P8; P8; P8; P8; P8; P8; P8; P8; P8;
57    end P9;
58    ------------------------------------------------------------------
59 begin -- FUEHRE_OFT_AUS
60    P9;
61 end FUEHRE_OFT_AUS;
```

Es geht darum, die Zuweisung **G1:=G1+1;** etwa 1_000_000_000 mal auszuführen, **ohne** eine **Schleifenanweisung** zu benützen. Wenn man die Prozedur **P1** aufruft, wird die Zuweisung **10** mal ausgeführt. Wenn man die Prozedur **P2** aufruft, wird die Prozedur **P1** genau **10** mal und somit die Zuweisung **G1:=G1+1;** genau $\mathbf{10^2}$ mal ausgeführt. Wenn man **P3** aufruft, wird die Anweisung $\mathbf{10^3}$ mal ausgeführt. ... Wenn man **P9** aufruft, wird die Zuweisung $\mathbf{10^9}$ mal ausgeführt.

Lösung 13.1.: Das Programm **AUSNA_02**:
Wenn der Benutzer ein **C** eingibt, löst der Ausführer (in Zeile 13) die Ausnahme **DAS_WAR_EIN_C** aus, beendet die Ausführung des normalen Anweisungsteils (Zeile 08 bis 16), geht in den Ausnahmeteil (Zeile 18 bis 22), findet in der zweiten Wahlliste (un Zeile 21) die Ausnahme **DAS_WAR_EIN_C** und führt die Anweisung des zugehörigen Behandlers (in Zeile 22) aus.

Wenn der Benutzer ein **X** eingibt, wird die **others**-Alternative der **case**-Anweisung ausgeführt (die **null**-Anweisung in Zeile 14) und danach die **put**-Anweisung in Zeile 16. Damit ist der Anweisungsteil der Prozedur **AUSNA_02** fertig ausgeführt.

Lösung 13.2.:

Eingabe	Ausgabe
A	Sie haben ein A eingegeben! Versuchen Sie mal ein B oder C!
B	Sie haben B oder C eingegeben!

C	Sie haben B oder C eingegeben!
X	Normalfall, keine Ausnahme!

Lösung 13.3.: Das Programm **AUSNA_03**:
Wenn der Benutzer ein **A** eingibt: Der Aufruf der Prozedur **AUSNA_01** (in Zeile 04) löst die Ausnahme **DAS_WAR_EIN_A** aus. Deshalb wird die Ausführung des normalen Anweisungsteils (Zeile 04 bis 05) abgebrochen und der zuständige Ausnahmebehandler (in Zeile 10 bis 11) ausgeführt.

Wenn der Benutzer ein **X** eingibt: Der Aufruf der Prozedur **AUSNA_01** (in Zeile 04) löst keine Ausnahme aus und die **put_line**-Anweisung in Zeile 05 wird ausgeführt.

Lösung 13.4.:

Eingabe	Ausgabe
A	AUSNA_01 loeste folg. Ausnahme aus: AUSNA_01.DAS_WAR_EIN_A
B	AUSNA_01 loeste constraint_error aus!
C	AUSNA_01 loeste folg. Ausnahme aus: ADA.IO_EXCEPTIONS.DATA_ERROR
X	Normalfall, keine Ausnahme! AUSNA_01 wurde normal beendet!

Lösung 13.5.: Programm **AUSNA_10**:
Wenn der Benutzer ein **A** eingibt: Die Ausnahme **DAS_WAR_EIN_A** wird ausgelöst (in Zeile 12). Die Ausführung des normalen Anweisungsteils (Zeile 11 bis 17) des Blocks **AUSNA_11** wird abgebrochen und der zuständige Ausnahmebehandler (in Zeile 20) wird ausgeführt. Danach wird die **put**-Anweisung in Zeile 24 ausgeführt.

Wenn der Benutzer ein **X** eingibt: Im Block **AUSNA_11** wird keine Ausnahme ausgelöst. Die **put_line**-Anweisung in Zeile 17 und die **put**-Anweisung in Zeile 24 werden ausgeführt.

Lösung 13.6.:

Eingabe	Ausgabe
A	Sie haben ein A eingegeben! AUSNA_11 wurde normal beendet!
B	Sie haben ein B oder C eingegeben! AUSNA_11 wurde normal beendet!
C	Sie haben ein B oder C eingegeben! AUSNA_11 wurde normal beendet!
X	Normalfall, keine Ausnahme! AUSNA_11 wurde normal beendet!

Lösung 13.7.: Siehe Datei **AUSNA_04.adb**.

Lösung 13.8.: Programm **AUSNA_20**:
Wenn der Benutzer ein **A** eingibt: Die Ausnahme **DAS_WAR_EIN_A** wird ausgelöst (in Zeile 13). Der zuständige Ausnahmebehandler (in Zeile 21) wird ausgeführt. Danach werden die **put_line**-Anweisungen in Zeile 23 und 28 ausgeführt.

Wenn der Benutzer ein **X** eingibt: Die **put_line**-Anweisungen in Zeile 18, 23 und 28 werden ausgeführt.

Lösung 13.9.:

Eingabe	Ausgabe
A	Sie haben ein A eingegeben!

	AUSNA_22 wurde normal beendet! AUSNA_21 wurde normal beendet!
B	Sie haben ein B eingegeben! AUSNA_21 wurde normal beendet!
C	Sie haben C eingegeben!
X	Normalfall, keine Ausnahme! AUSNA_22 wurde normal beendet! AUSNA_21 wurde normal beendet!

Lösung 13.10.:

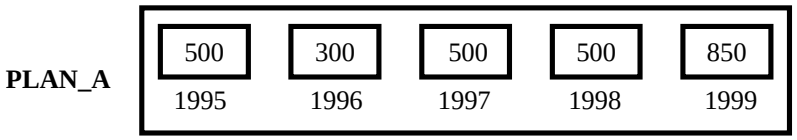
Anweisungsteil	der Prozedur AUSNA_30:	Zeile 34 bis 38
Normaler Anweisungsteil	der Prozedur AUSNA_30:	Zeile 34 bis 35
Ausnahmeteil	der Prozedur AUSNA_30:	Zeile 37 bis 38
Anweisungsteil	der Prozedur AUSNA_31:	Zeile 26 bis 30
Normaler Anweisungsteil	der Prozedur AUSNA_31:	Zeile 26 bis 27
Ausnahmeteil	der Prozedur AUSNA_31:	Zeile 29 bis 30
Anweisungsteil	der Prozedur AUSNA_32:	Zeile 10 bis 21
Normaler Anweisungsteil	der Prozedur AUSNA_32:	Zeile 10 bis 18
Ausnahmeteil	der Prozedur AUSNA_32:	Zeile 20 bis 21

Im Rahmen AUSNA_30 wird die Ausnahme DAS_WAR_EIN_C behandelt.
Im Rahmen AUSNA_31 wird die Ausnahme DAS_WAR_EIN_B behandelt.
Im Rahmen AUSNA_32 wird die Ausnahme DAS_WAR_EIN_A behandelt.

Lösung 13.11.:

Eingabe	Ausgabe
A	Sie haben ein A eingegeben! AUSNA_32 wurde normal beendet! AUSNA_31 wurde normal beendet!
B	Sie haben ein B eingegeben! AUSNA_31 wurde normal beendet!
C	Sie haben C eingegeben!
X	Normalfall, keine Ausnahme! AUSNA_32 wurde normal beendet! AUSNA_31 wurde normal beendet!

Lösung 14.1.1.: Die Reihung PLAN_A:



2. Die Indices sind vom Untertyp **JAHR**.
3. Die Komponenten sind vom Untertyp **BETRAG**.
4. Die Reihung PLAN_A ist vom Untertyp **PLAN**.

Lösung 14.1.2.: Vereinbarung der Reihung **KARL**:

```
01 declare
02    type  OTTO is range 1..5;
03    type  EMIL is array(OTTO) of boolean;
04    KARL: EMIL := (true, false, true, true, false);
```

Lösung 14.1.3.: Werte für die Komponenten der Reihung **ANZ** einlesen:

```
01 begin
02    for G in GROESSE range ANZ'range loop
03       ada.text_io.put(item => "Bitte eine Anzahl (0..100) eingeben: ");
04       ANZAHL_EA  .get(item => ANZ(G));
05    end loop;
```

Lösung 14.1.4.: Die Komponenten der Reihung **ANZ** summieren:

```
01 begin
02    SUMME := 0;
03    for G in GROESSE range ANZ'range loop
04       SUMME := SUMME + ANZ(G);
05    end loop;
```

Lösung 14.1.5.: Die Komponenten der Reihung **ANZ** bearbeiten:

```
01 begin
02    for G in GROESSE range ANZ'range loop
03       if ANZ(G) mod 2 = 0 then
04          ANZ(G) := ANZ(G) / 2;
05       else
06          ANZ(G) := ANZ(G) - 1;
07       end if;
08    end loop;
```

Lösung 14.1.6.: Sind alle Komponenten von **ANZ** gerade Zahlen?

```
01 begin
02    ALLE_SIND_GERADE := true;
03    for G in GROESSE range ANZ'range loop
04       if ANZ(G) mod 2 /= 0 then
05          ALLE_SIND_GERADE := false;
06          exit;
07       end if;
08    end loop;
```

Lösung 14.1.7.: Ist die Reihung **ANZ** sortiert?

```
01 begin
02    IST_SORTIERT := true;
03    for G in GROESSE range ANZ'first..ANZ'last-1 loop
04       if ANZ(G) > ANZ(G+1) then
05          IST_SORTIERT := false;
06          exit;
07       end if;
08    end loop;
```

Lösung 14.1.8.: Sind alle Werte in **ANZ** verschieden oder gibt es "Doppelgänger"?

```
01 begin
02    KEIN_WERT_DOPPELT := true;
03    for G1 in GROESSE range ANZ'first..ANZ'last-1 loop
04       for G2 in GROESSE range G1+1..ANZ'last loop
05          if ANZ(G1) = ANZ(G2) then
06             KEIN_WERT_DOPPELT := false;
07          end if;
08       end loop;
09    end loop;
```

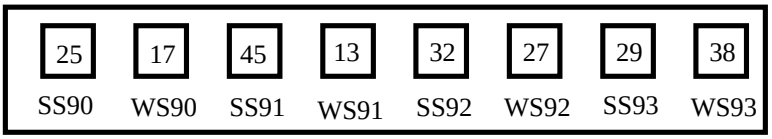
Lösung mit **Bezeichnern** an den Schleifen und **exit**-Anweisung:

```
01 begin
02    KEIN_WERT_DOPPELT := true;
```

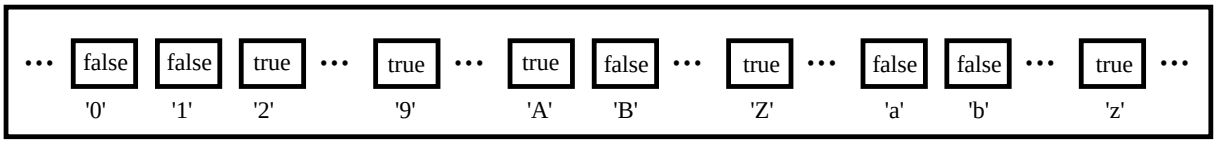
```
03    SCHLEIFE1: for G1 in GROESSE range ANZ'first..ANZ'last-1 loop
04       SCHLEIFE2: for G2 in GROESSE range G1+1..ANZ'last loop
05          if ANZ(G1) = ANH(G2) then
06             KEIN_WERT_DOPPELT := false;
07             exit SCHLEIFE1;
08          end if;
09       end loop SCHLEIFE2;
10    end loop SCHLEIFE1;
```

Lösung 14.1.9.:

1. Der Untertyp **SEMESTER** ist der Indexuntertyp des Reihungsuntertyps TEILNEHMER
2. Der Untertyp **ANZAHL** ist der Komponentenuntertyp des Untertyps TEILNEHMER
3. Jede Reihung vom Untertyp TEILNEHMER hat 8 Komponenten (weil zum Untertyp SEMESTER genau 8 Werte gehören).
4. Bildliche Darstellung einer Reihung des Untertyps TEILNEHMER:



5. Der Untertyp **character** ist der Indexuntertyp des Reihungsuntertyps ZEICHEN_MENGE.
6. Der Untertyp **boolean** ist der Komponentenuntertyp des Untertyps ZEICHEN_MENGE.
7. Jede Reihung vom Untertyp ZEICHEN_MENGE hat 256 Komponenten (weil zum Untertyp character genau 256 Werte gehören, siehe (ARM A.1)).
8. Bildliche Darstellung einer Reihung des Untertyps ZEICHEN_MENGE:



Lösung 14.1.10.: Ein paar Vereinbarungen:

```
01 declare
02    type ISADORA is range 0..100;
03    type KLAUS   is range -10_000..+10_000;
04    type REIMUND is array(ISADORA) of KLAUS;
05    RV1: REIMUND := (0 => -500, 1..3 => +250, others => -370);
06    RV2: REIMUND := (others => 0);
```

Lösung 14.1.11.: Werte einiger Attribute:

Attribut	Wert
BESTAND'first	34
BESTAND'last	48
BESTAND'range	34..48
BESTAND'length	15

Attribut	Wert
ZEICHEN_MENGE'first	character'first
ZEICHEN_MENGE'last	character'last
ZEICHEN_MENGE'range	character
ZEICHEN_MENGE'length	256

Lösung 14.1.12.:

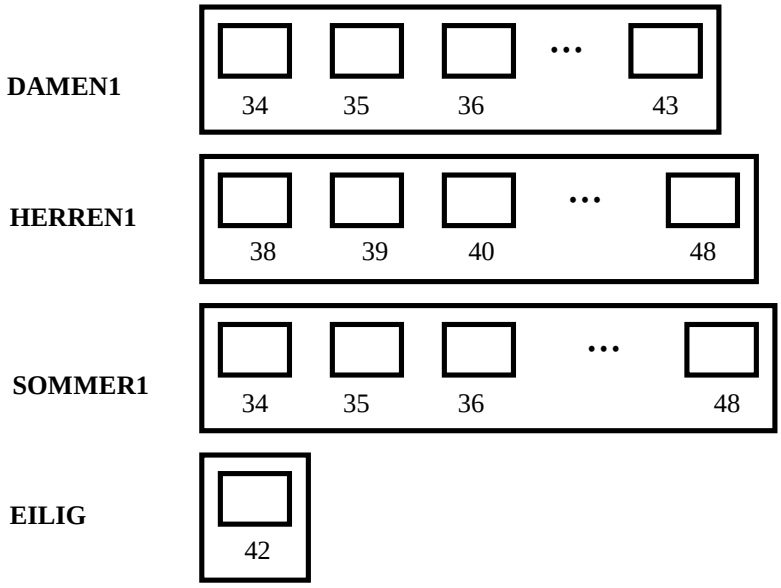
PLAN'last	ist der letzte **Index**	des Untertyps PLAN.
ANZAHL'last	ist der letzte **Wert**	des Untertyps ANZAHL.

Lösung 14.1.13.:
Die Menge **M6** enthält die Zeichen '0'..'9'.
Die Menge **M7** enthält alle Zeichen des Typs **character** mit Ausnahme der Ziffern '0'..'9'.
Die Menge **M8** enthält alle Zeichen des Typs **character** mit Ausnahme der ungeraden Ziffern '1', '3', '5', '7' und '9'.

Lösung 14.1.14.: Zeichen in eine Menge einfügen und aus einer Menge entfernen:

```
01 begin
02    M1('7') := true;      -- '7' in M1 einfuegen
03    M1('X') := false;     -- 'X' aus M1 entfernen
```

Lösung 14.2.1.: Ein paar Reihungen bildlich dargestellt:



Lösung 14.2.2.: Falsche Vereinbarungen von Reihungen:
1. Untergrenze **5** gehört nicht zum Indextyp **GROESSE**, Obergrenze fehlt.
2. Obergrenze **50** gehört nicht zum Indextyp **GROESSE**.
3. Das Aggregat ist unvollständig: Für die Komponente mit Index **41** fehlt eine Angabe.
4. Das Aggregat ist mehrdeutig: Mehrere Angaben mit Indices **39** und **40**.
5. Das Aggregat enthält zuviele Komponenten (nämlich 16, maximal erlaubt sind 15).
6. Für die Reihung **HERREN8** wurden keine Indexgrenzen festgelegt.

Lösung 14.2.3.: Bei der Realisierung von **flexiblen Strings begrenzter Länge** mit eine zusätzlichen Ganzzahlvariablen können **alle** Zeichen im String vorkommen (auch beliebig viele **Nullzeichen**) und der Zugriff auf den belegten Teil eines solchen Strings ist in einigen Fällen **etwas schneller**. Als Nachteil ist die **zusätzliche Variable** zu werten (Speicherplatz). **Nullterminierte Strings** kommen ohne zusätzliche Variable aus, können aber nur **ein** Nullzeichen enthalten und in einigen Fällen ist der Zugriff auf solche Strings etwas langsamer.

Lösung 14.2.4.: Siehe Datei **HALLO_30.adb**.
Lösung 14.2.5.: Siehe Datei **REIHE_05.adb**.
Lösung 14.2.6.: Siehe Datei **REIHE_06.adb**.

Lösung 14.2.7.: Eine änderungsfreundliche Zuweisung:

```
07 T1(T1'last-2..T1'last) := T2;
```

Lösung 14.2.8.: Es gibt 256^{10_000} verschiedene Strings der Länge 10_000 (weil zum Komponentenuntertyp **character** genau 256 Werte gehören).

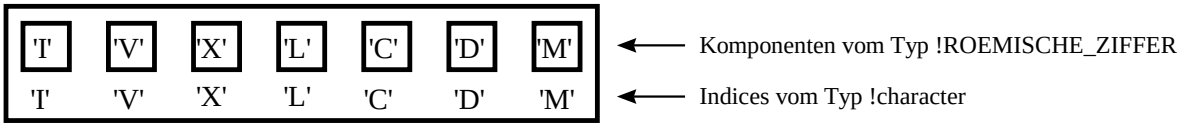
Lösung 14.2.9.: Die Zeichenketten in (aufsteigender) **lexikografischer Reihenfolge**:
"AAA", "BBB", "AAAAAAAAAA", "BA", "AAAAAAAABA", "B", "BAB", "A".

Lösung 14.2.10.: Man vergleicht die beiden Strings S1 und S2 von links nach rechts Komponente für Komponente bis eines der folgenden Ereignisse eintritt:
1. Die beiden verglichenen Komponenten S1(I1) und S2(I2) sind **ungleich**. Der String mit der **kleineren Komponente** an dieser ersten "ungleichen Stelle" muß in einem Lexikon vor dem anderen String stehen.
2. Man erreicht das Ende des einen Strings, während beim anderen String "noch Komponenten übrig sind". In diesem Fall stimmt der **kürzere** String mit einem Anfangsstück des **längeren** Strings überein und muß im Lexikon **vor** dem längeren String stehen (z.B. "BA" vor "BAC").
3. Beide Strings sind gleichzeitig zu Ende. In diesem Fall sind die Strings gleich.

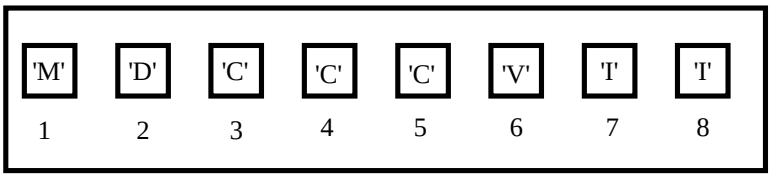
Lösung 14.2.11.: Siehe Datei **REIHU_03.adb**.

Lösung 14.3.1.:

ROEM_TO_CHAR

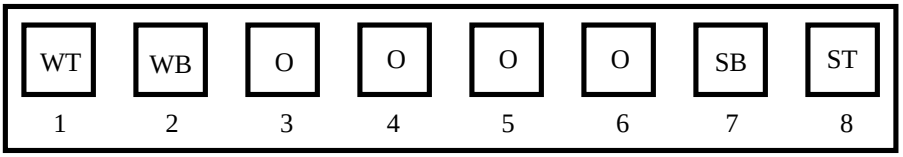


R1

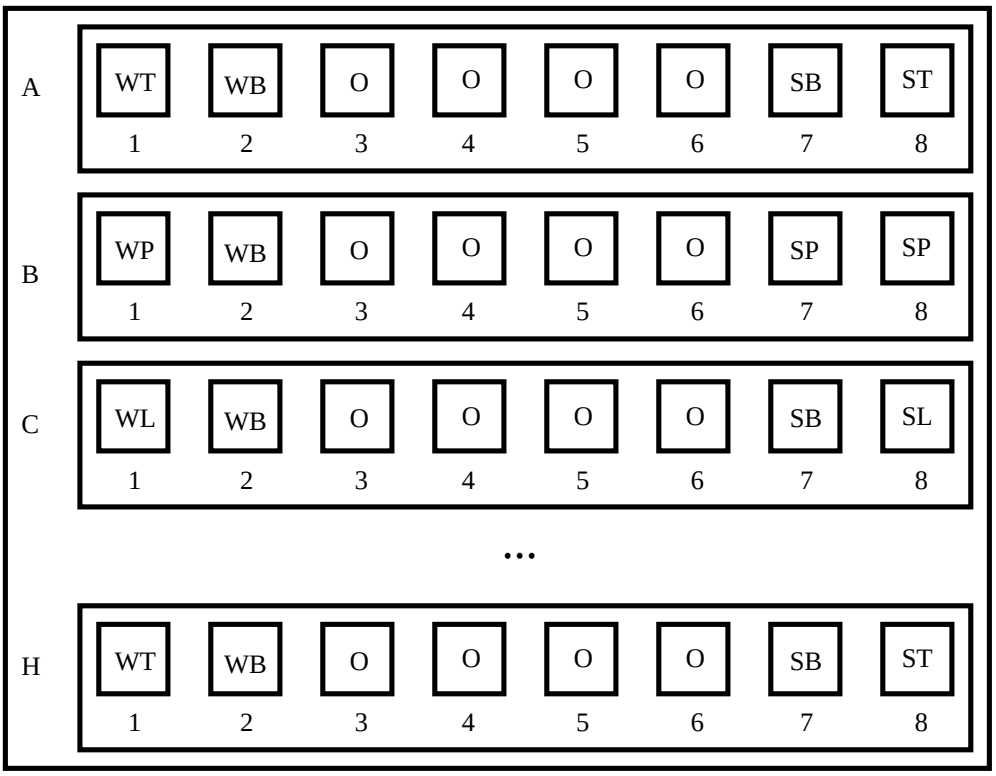


Lösung 14.3.2.: Siehe Datei **REIHU_02.adb**.

Lösung 14.4.1.: Eine Reihung des Untertyps **REIHE**:



Lösung 14.4.2.: Die Reihung SU:

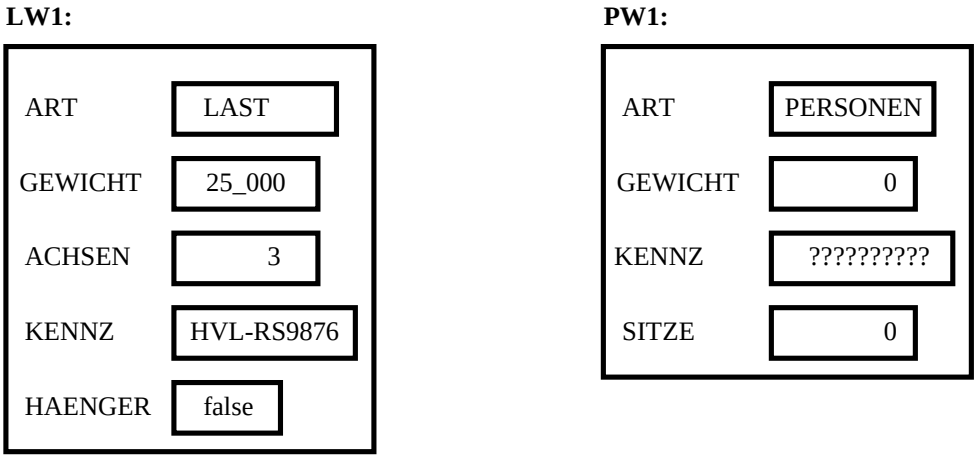


Lösung 14.4.3.: Mit einem weißen Pferd eröffnen:

```
01 begin
02    SU(C)(3) := SU(B)(1);
03    SU(B)(1) := O;
```

Lösung 15.1.1.: Siehe Datei **VERBE_01.adb**.
Lösung 15.1.2.: Siehe Datei **VERBE_02.adb**.

Lösung 15.2.1.: Variablen eines varianten Verbunduntertyps:



Lösung 15.2.2.: Siehe Datei **VERBU_02.adb**.

Lösung 15.2.3.: Für jeden Wert des Untertyps **natural** gibt es eine Variante des Untertyps **FLEX**. Wenn zum Untertyp **natural** z.B. 2^{31} Werte gehören (0.. 2^{31}-1), dann gibt es 2^{31} Varianten des Untertyps FLEX und die S-Komponente eines größten Verbundes vom Untertyp **FLEX** besteht aus 2^{31}-1 Komponenten.

Lösung 15.2.4.: Vom Untertyp **SCHUH** gibt es **20 Varianten** (5 mal 4, weil zum Untertyp **FARBE** 5 Werte gehören und zum Untertyp **ART** 4 Werte).

Lösung 15.2.5.: Siehe Datei **VERBU_03.adb**.
Lösung 15.2.6.: Siehe Datei **VERBU_04.adb**.

Lösung 17.2.1.: Im Vereinbarungsteil der Prozedur HALLO werden 3 Dinge vereinbart: Eine Konstante namens TEXT und zwei Prozeduren namens HALLO1 und HALLO2. Die Prozedur HALLO gibt folgende Zeilen zur aktuellen Ausgabe aus:

```
Alloh alloh!
Hallo!
Hello one!
Alloh alloh!
```

Lösung 17.2.2.:
Übersetzung der Vereinbarung der Prozedur **HALLO1** (Zeile 04 bis 08) ins Deutsche:
Erzeuge eine Prozedur namens **HALLO1**, die aus den folgenden zwei Befehlen besteht:

1. Erzeuge eine Konstante namens **TEXT** vom Untertyp string mit dem Wert "Hello one!".
2. Führe die Prozedur **put_line** aus mit der Konstanten **TEXT** als **item**-Parameter.

Übersetzung der Vereinbarung der Prozedur HALLO (Zeile 01 bis 19) ins Deutsche:
Binde das Paket **ada.text_io** in jedes Programm mit ein, in das du die folgende Prozedur **HALLO** einbindest.
Erzeuge eine Prozedur namens **HALLO**, die aus den folgenden 7 Befehlen (3 Vereinbarungen und 4 Anweisungen) besteht:

1. Erzeuge eine Konstante namens **TEXT** vom Untertyp **string** mit dem Wert "Hallo!".
2. Erzeuge eine Prozedur namens **HALLO1**, die ... (siehe oben).
3. Erzeuge eine Prozedur namens **HALLO2**, die ... (ganz entsprechend wie **HALLO1**).
4. Führe die Prozedur **HALLO2** aus.
5. Führe die Prozedur **put_line** mit der Konstanten **TEXT** als **item**-Parameter aus.
6. Führe die Prozedur **HALLO1** aus.
7. Führe die Prozedur **HALLO2** aus.

Lösung 17.1.1.: Den Aufruf der Prozedur **DOPPEL_LINIE_2** in Zeile 16 des Beispiels 17.1.2. führt der Ausführer wie folgt aus:

1. Er berechnet den Wert des Literals **80** und den Wert des Ausdrucks **character'succ(Z1))**. Als Ergebnis bekommt er die Werte **80** und **'Y'** heraus.
2. Mit diesen Werten erzeugt er die formalen Parameter **LAENGE** und **ZEICHEN**, so als wären sie folgendermaßen vereinbart worden:

```
LAENGE : constant natural   := 80;
ZEICHEN: constant character := 'Y';
```

3. Er führt die Vereinbarung in Zeile 04 der Prozedur **DOPPEL_LINIE_2** aus, d.h. er erzeugt eine Konstante namens **LINIE** vom Untertyp **string** mit den Indices 1..80 und dem Wert **(others => 'Y')**.
4. Dann führt er die beiden **put_line**-Anweisungen (in Zeile 06 und 07) aus.
5. Schließlich zerstört er die formalen Parameter **LAENGE** und **ZEICHEN** und die Konstante **LINIE** wieder und
6. macht hinter der Aufrufstelle (d.h. in Zeile 17) weiter.

Lösung 17.1.2.: Siehe Datei **UPROS_09.adb**.

Lösung 17.1.3.: Den Aufruf der Prozedur **GET_ZIFFER** in Zeile 21 des Beispiels 17.1.3. führt der Ausführer folgendermaßen aus:
1. Er erzeugt die formalen Parameter ITEM und ALLES_OK so, als wären sie folgendermaßen vereinbart worden:

```
ITEM     : character;
ALLES_OK : boolean;
```
2. Er führt die Vereinbarung in Zeile 06 aus, d.h. er erzeugt eine Konstante namens **TEXT**.
3. Er führ die drei Anweisungen in Zeile 08 bis 14 aus.
4. Er weist den **aktuellen** Parametern die Werte der **formalen** Parameter zu.:
```
Z1             := ITEM;
ZIFFER_GELESEN := ALLES_OK;
```
5. Er zerstört die formalen Parameter **ITEM** und **ALLES_OK** und die Konstante **TEXT** und
6. macht hinter der Aufrufstelle (also in Zeile 22) weiter.

Zusammgefaßte Beschreibung des Prozeduraufrufs: Von der aktuellen Eingabe wird ein Zeichen in die Variable **Z1** gelesen. Falls dieses Zeichen eine Ziffer ist, wird der booleschen Variablen **ZIFFER_GELESEN** der Wert **true** zugewiesen, sonst **false**.

Lösung 17.1.4.: Den Aufruf der Prozedur **PLUS3** in Zeile 15 des Beispiels 17.1.4. führt der Ausführer folgendermaßen aus:
1. Er ermittelt den Wert des aktuellen Parameters **AG2**. Wir nehmen hier mal an, daß dieser wert gleich **-123** ist.
2. Mir diesem Wert erzeugt er den formalen Parameter **FG** so, als wäre er folgendermaßen vereinbart worden:
```
FG : GANZ := -123;
```
3. Er führt die Vereinbarung in Zeile 07 aus, d.h. er erzeugt eine Konstante namens **DREI**.
4. Er führt die Anweisung in Zeile 09 aus.
5. Er weist dem aktuellen Parameter den Wert des formalen Parameters zu:
```
AG2 := FG;
```
6. Er zerstört den formalen Parameter **FG** und die Konstante **DREI** und
7. macht hinter der Aufrufstelle (d.h. in Zeile 16) weiter.

Zusammengefaßte Beschreibung des Prozeduraufrufs: Der Wert der Variablen **AG2** wird um 3 erhöht.

Lösung 17.1.4.: Siehe Datei **UPROS_05.adb**.

Lösung 17.2.1.: Den Aufruf der Funktion **ABSOLUT_KLEINER** in Zeile 17 des Beispiels 17.2.1. führ der Ausführer folgendermaßen aus:
1. Er ermittelt den Wert des Literals **17** und den Wert des Ausdrucks **AG1 + AG2**. Hier nehmen wir einmal an, daß er dabei die Werte **17** und **25** ermittelt.
2. Mit diesen Werten erzeugt er die formalen Parameter **FP1** und **FP2**, so als wären sie wie folgt vereinbart worden:
```
FP1 : constant GANZ := 25;
FP2 : constant GANZ := 17;
```
3. Er führt die Vereinbarungen in Zeile 07 und 08 aus, d.h. er erzeugt zwei Konstanten namens **BETRAG1** und **BETRAG2** mit den Werten 25 bzw. 17.
4. Er führt die **return**-Anweisung in Zeile 10 aus. Dazu berechnet er den Wert des Ausdrucks **BETRAG1 < BETRAG2** und merkt sich das Ergebnis (den Wert **false**) als Ergebnis der Funktion **ABSOLUT_KLEINER**. Dann zerstört er die formalen Parameter **FP1** und **FP2** und die beiden Konstanten **BERTRAG1** und **BETRAG2** und kehrt mit dem Ergebnis der Funktion (**false**) zur Aufrufstelle in Zeile 17 zurück. Dort steht jetzt sozusagen **B1 := false;**.

Lösung 17.2.2.: Siehe Datei **UPROS_22.adb**.
Lösung 17.2.3.: Siehe Datei **UPROS_23.adb**.
Lösung 17.2.4.: Siehe Datei **UPROS_24.adb**.
Lösung 17.2.5.: Siehe Datei **UPROS_25.adb**.

Lösung 17.2.6.: Die ausgefüllte Tabelle mit den Ergebnissen der Funktion "&":

G1	G2	G1 / G2	G1 rem G2	G1 & G2
-6	3	-2	0	**-2**
-5	3	-1	-2	**-2**
-4	3	-1	-1	**-2**
-3	3	-1	0	**-1**
-2	3	0	-2	**-1**

Lösung 17.2.7.:
In Zeile 26 wird die in Zeile 11 bis 16 vereinbarte "&"-Funktion aufgerufen.
In Zeile 27 wird die in Zeile 18 bis 26 vereinbarte "&"-Funktion aufgerufen.
In Zeile 28 werden (von links nach rechts) folgende "&"-Funktionen aufgerufen:
1. die in Zeile 18 bis 23 vereinbart, 2. die in Zeile 11 bis 16 vereinbarte und 3. eine vordefinierte "&"-Funktion mit zwei string-Parametern.

Lösung 17.4.1.: In der Funktion **ANZAHL_BUCHSTABEN_1** ist die **for**-Schleife ab Zeile 05 falsch. Der Schleifenindex **I** durchläuft nicht alle Indices des Strings S. Statt
range 1..S'length müßte es **range S'range** oder **range S'first..S'last** heißen.

Lösung 17.4.2.: In der Funktion **ANZAHL_GLEICHER_STELLEN** wird für die beiden Strings **S1** und **S2** derselbe Index **I** verwendet (in Zeile 07: **if S1(I) = S2(I) then**). Das ist falsch, weil die aktuellen Parameter nicht unbedingt die gleichen Indices haben.

Lösung 17.4.3.: Die Funktion ALLE_ZIFFERN:

```
01 function ALLE_ZIFFERN(S: string) return string is
02 -------------------------------------------------
03 -- Liefert einen String, der alle Ziffern aus S
04 -- enthaelt.
05 -------------------------------------------------
06    ERGEBNIS: string(1..S'length);
07    LBI     : natural := ERGEBNIS'first - 1;
08 begin
09    for I in natural range S'range loop
10        if S(I) in '0'..'9' then
11           LBI           := LBI + 1;
12           ERGEBNIS(LBI) := S(I);
13        end if;
14     end loop;
15     return ERGEBNIS(ERGBNIS'first..LBI);
16 end ALLE_ZIFFERN;
```

Lösung 17.4.4.: Siehe Datei **UPROS_28.adb**.

Lösung 17.5.1.: Ergebnisse der rekursiven Funktion **FAKULTAET**:

N	0	1	2	3	4	5	6	
FAKULTAET(N)	**1**	**1**	**2**	**6**	**24**	**120**	**720**	

Lösung 17.5.2.: Ergebnisse der rekursiven Funktion **FIBO**:

ANZ_TAGE =>	0	1	2	3	4	5	
FIBO(1, ANZ_TAGE)	1	2	4	8	16	32	
FIBO(2, ANZ_TAGE)	2	3	4	6	9	13	
FIBO(3, ANZ_TAGE)	3	4	5	6	8	10	
FIBO(4, ANZ_TAGE)	4	5	6	7	8	10	

Lösung 17.5.3.: Ein Funktionsaufruf wie z.B.
FIBO(ANZ_ELTERN => **0**, ANZ_TAGE => 3) löst die Ausnahme **constraint_error** aus, weil er eine Division durch 0 veranlaßt. **Verbesserung** des Beispiels 17.5.2.:

```
02    type    NAT is     range 0..1_000_000_000;            -- Wie bisher.
03    subtype POS is NAT range 1..NAT'last;                 -- Neu!
04    function FIBO(ANZ_ELETERN: POS; ANZ_TAGE: NAT) return NAT is -- Neu!
05    ...                                                   -- Wie bisher.
```

Aufgabe 17.5.4.: Keine Musterlösung.
Lösung 17.5.5.: Siehe Datei **REKUP_03.adb**.

Lösung 17.5.6.: SAEGEZAHN(MAX_L=>8, AKT_L=>8, ZEILEN=>10):

```
********
******
****
**
********
******
****
**
********
******
```

Lösung 17.5.7.: Siehe Datei **REKUP_05.adb**.
Lösung 17.5.8.: Die Prozedur **NICHT_RICHTIG** enthält in Zeile 05 einen rekursiven Aufruf, der zu einer **Endlosrekursion** führt. Maschinelle Ada-Ausführer brechen das Programm mit einer Ausnahme (**storage_error** oder **constraint_error**) ab.
Lösung 17.5.9.: Siehe Datei **REKUP_06.adb** und **REKUP_07.adb**.
Lösung 18.1.: Ausgabe des Programms LESIB_01: **17 Stdn 35 Min 58 Sec**
Lösung 18.2.: Ausgabe des Programms LESIB_02: **BAAAB**

Lösung 18.3.:

Variable	**OTTO-03**	**OTTO-05**	**OTTO-09**	**OTTO-19**
Gültigkeitsbereich	03-24	05-13	09-11	19-21
Sichtbarkeitsbereich	04, 16-18, 23-24	06-08, 13	10-11	20-21

Lösung 18.4.: Ausgabe des Programms LESIB_03: **ABCBDBCBA**
Lösung 18.5.: Ausgabe des Programms LESIB_04: **ABACBADCBA**
Lösung 18.6.: Ausgabe des Programms LESIB_05:
123
35 DM
ECU 17

Lösung 18.7.:
LESIB_06-02: Gültig in Zeile 03-20 und in allen Programmeinheiten, vor denen die **with**-Klausel **with LESIB_06** steht. Direkt sichtbar in Zeile 03-04.
LESIB_06-04: Gültig in Zeile 05-20. Direkt sichtbar in Zeile 05-06 und 19-20.
LESIB_06-06: Gültig in Zeile 07-16. Direkt sichtbar in Zeile 07-16.
OTTO-03: Direkt sichtbar in Zeile 04 und 19-20.
OTTO-05: Direkt sichtbar in Zeile 06 und 14-16.
OTTO-07: Direkt sichtbar in Zeile 08-11.

Lösung 18.8.: Der einfache Name **LESIB_06** bezichnet
in Zeile 20 die Prozedur LESIB_06-**04** und
in Zeile 14 die Prozedur LESIB_06-**06**.
Lösung 18.9.: Ausgabe des Programms LESIB_06: **ACBABA**
Lösung 19.1.1.: Siehe Datei **BEN1P_01.adb**.
Lösung 19.1.2.: Siehe Datei **BEN2P_01.adb** und **BEN3P_01.adb**.
Lösung 19.1.3.: Siehe Datei **BEN4P_01.adb**.

Lösung 19.1.4.:

```
01 package PAKET1 is
```

```
02    V12: string := "Hallo 12!";
03 end PAKET1;

04 package PAKET2 is
05    V23: string := "Hallo 23!";
06 end PAKET2;

07 package PAKET3 is
08    V31: string := "Hallo 31!";
09 end PAKET3;

10 with PAKET1, PAKET2;
11 procedure U1 is ... end U1;

12 with PAKET2, PAKET3;
13 procedure U2 is ... end U1;

14 with PAKET3, PAKET1;
15 procedure U3 is ... end U1;
```

Lösung 19.1.5.: Siehe Datei **PAKET_07.ads** und **PAKET_07.adb**.
Lösung 19.2.1.: Siehe Datei **BEN1P_02.adb**.

Lösung 19.3.1.: Die Prozedur **PUSH** entfernt nicht wirklich irgendwelche Elemente von einem Stapel, sondern vermindert nur den "Füllhöhenanzeiger" des Stapels um 1. Wenn man zwei Stapel **OTTO** und **EMIL** mit der vordefinierten Gleichheitsfunktion **"="** vergleicht kann **false** herauskommen, weil früher verschiedene Zeichen auf **OTTO** und **EMIL** lagen und obwohl die beiden Stapel jetzt eigentlich gleich sind ("bis zu ihrer Füllhöhe"). Indem man **S_TYP** als **limitierten** Typ vereinbart, verhindert man die Benutzung der "fehlerhaft funktionierenden" vordefinierten Gleichheitsfunktion "=".

Lösung 19.3.2.: Der Rumpf einer "richtigen" Gleichheitsfunktion für Stapel:

```
01 package body PAKET_03 is
...
36    function "="(L, R: S_TYP) return boolean is
37    begin
38       return L.STAPEL(L.STAPEL'first..L.LBI) =
39              R.STAPEL(R.STAPEL'first..R.LBI);
40    end "=";
...
```

Lösung 19.4.1.: Siehe Datei **PAKET_04.ads** und **PAKET_04.adb**.
Lösung 20.1.1.: Siehe Datei **SCHAB_02.adb**.

Lösung 20.1.2.: Das Programm **TESTS_02** gibt 16 Zahlen **untereinander** aus. Die Zahlen werden hier **nebeneinander** wiedergegeben:

```
1  1  1  -5  25  -125  625  -3125    1  1  1  -5  25  -125  625  -3125
```

Lösung 20.1.3.: Siehe Datei **SCHAB_03.adb**.
Lösung 20.1.4.: Siehe Datei **TESTS_03.adb**.

Lösung 20.1.5.: Die Schablone namens **enumeration_io** (im Paket **ada.text_io**) sollte besser **discrete_io** heißen, weil man sie nicht nur mit Aufzählungstypen (enumeration types) parametrisieren kann, sondern mit beliebigen **diskreten** Typen.

Lösung 20.1.6.: Siehe Datei **SCHAB_04.ads**, **SCHAB_04.adb** und **TESTS_04.adb**.
Lösung 20.1.7.: Siehe Datei **SCHAB_05.adb**.
Lösung 20.2.1.: Siehe Datei **BEN1S_06.adb**.

Lösung 20.3.1.: Ausgabe des Programms TESTS_06:
```
RBBBBBBL
BBBBBB
```

Lösung 20.3.2.: Siehe Datei **BEN1S_08.adb**.
Lösung 20.3.3.: Siehe Datei **SCHAB_09.ads**, **SCHAB_09.adb** und **TESTS_09.adb**.
Lösung 20.3.4.: Siehe Datei **TESTS_10.adb**.

Lösung 20.4.1.: Zwei verschiedene Stapel vereinbaren:

```
01 with SCHAB_11;
02 procedure KUNIGUNDE is
03    package BOOLEAN_STAPEL is
04       new SCHAB_11(E_TYP => boolean, MAX_ANZAHL => 50);
05    package INTEGER_STAPEL is
06       new SCHAB_11(MAX_ANZAHL => 2000, E_TYP => integer);
07    ...
```

Lösung 20.4.2.: Ausgabe des Programms TESTS_12:
```
KONES(T1) ist gleich Hallo, wie geht es?
PLUS3(15) ist gleich 18
MAL17(-3) ist gleich -51
```

Lösung 20.4.3.: Die Funktionsschablone **SCHAB_12** hat unter anderem einen generisch formalen Parameter namens **K**. Dieser Parameter ist (kein Typ und kein Unterprogramm sondern) ein **Objekt** und hat den Modus **in** (weil kein anderer Modus angegeben wurde). Diesem Parameter muß beim Instanziieren also ein Wert zugewiesen werden. Das wäre aber nicht erlaubt, wenn **K** zu einem limitierten Typ gehören würde. Der Objektparameter K gehört zum Typ **ANY**. Darum darf der formale Typparameter **ANY** nicht als limiterter (privater) Typ vereinbart werden. Siehe dazu auch (ARM 7.5(13)).

Lösung 21.1.: Mein Taschenrechner (Casio fx-82 Super Fraction, Preis ca. 20,- DM) kann die Fakultät von 13 noch exakt berechnen (6_227_020_800) und die Falultät von 69 noch näherungsweise ermitteln ($1.711\_224\_524 * 10^{98}$).

Lösung 21.2.: Mein Taschenrechner kann die Zahl 9.0 insgesamt sechsmal quadrieren. $(((((9.0^2)^2)^2)^2)^2)^2$ ist näherungsweise gleich $1.179\_018\_458 * 10^{61}$. Die Zahl 5.5 kann dieser Taschenrechner siebenmal quadrieren mit dem näherungsweisen Ergebnis $5.840\_153\_408 * 10^{94}$.

Aufgabe 21.3.: Keine Musterlösung.
Aufgabe 21.4.: Keine Musterlösung.

Lösung 21.1.1.: Mit **dezimalen Festpunktzahlen** rechnen (Beispielprogramm **BRUCH_01**)**:** Gibt man Zahlen mit mehr als zwei Nachpunktstellen ein, wird die eingegebene Zahl auf zwei Stellen nach dem Punkt gerundet. Das Ergebnis der Addition und der Subtraktion ist mathematisch exakt. Beim Multiplizieren und Dividieren werden alle Nachpunktstellen ab der dritten (einschließlich) abgeschnitten (es erfolgt also kein Runden). Wenn man Ganzzahlen eingibt, werden die wie Bruchzahlen mit zwei Nullen nach dem Dezimalpunkt eingelesen. Im Beispielprogramm **BRUCH_02** werden Zahlen mit 2 Nachpunktstellen und Zahlen mit 4 Nachpunktstellen eingelesen und miteinander verrechnet.

Lösung 21.2.1.: Mit **gewöhnlichen Festpunktzahlen** rechnen (Beispielprogramm **BRUCH_04**)**:** Für den Typ **GFIX1** wurde eine **delta**-Zahl von 0.1 angegeben. Typische Ada-Compiler wählen daraufhin für **GFIX1** als **small**-Zahl den Wert 0.0625 (gleich 1/16 gleich 2^{-4}, das ist die größte Zweierpotenz, die kleiner oder gleich der **delta**-Zahl 0.1 ist). Werte des Typs **GFIX1** werden also als ganzzahlig Vielfache von 1/16 dargestellt. Im Programm **BRUCH_05** wählen typische Ada-Compiler für den gewöhnlichen Festpunkttyp **GFIX2** als **small**-Zahl den Wert 0.0078125 (gleich 1/128 gleich 2^{-7}, das ist die größte Zweierpotenz, die kleiner oder gleich der **delta**-Zahl 0.01 ist). Werte des Typs **GFIX2** werden intern im Rechner also als ganzzahlig Vielfache von 1/128 dargestellt.

Lösung 21.3.1.: Zwei positive, **dezimale**, normalisierte Gleitpunktzahlen mit gleichem Exponenten können sich höchstens um einen Faktor unterscheiden, der kleiner als 10 ist (nämlich um den Faktor 9.999...). Zwei positive,

binäre, normalisierte Gleitpunktzahlen mit gleichem Exponenten können sich höchstens um einen Faktor unterscheiden, der kleiner als 2 ist.

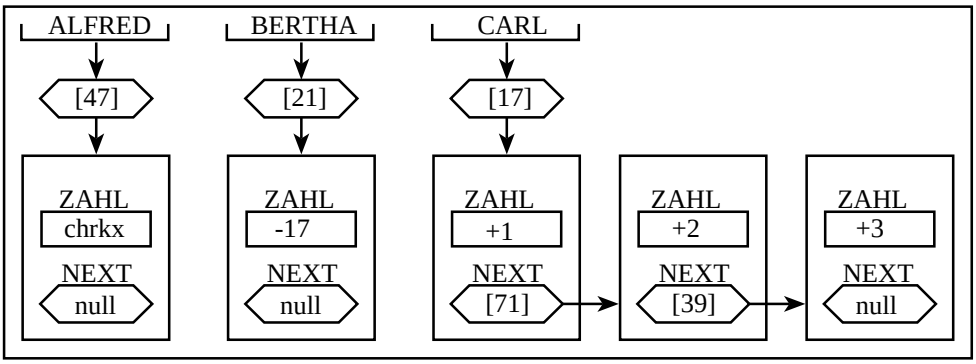
Lösung 21.3.2.: Zwei positive Gleitpunktzahlen vergleichen: Zuerst sollte man die **Exponenten** miteinander vergleichen. Wenn einer der Exponenten größer ist als der andere, ist die zugehörige Gleitpunktzahl größer als die andere, unabhängig davon, welche Mantissen die beiden Gleitpunktzahlen haben. Nur wenn die beiden Exponenen **gleich** sind, muß man auch die **Mantissen** vergleichen.

Lösung 21.3.3.: Seien **M1** und **E1** (bzw. **M2** und **E2**) die Mantisse und der Exponent einer Gleitpunktzahl **G1** (bzw. **G2**). Um G1 und G2 zu **multiplizieren** kann man etwa so vorgehen:
1. Man **addiert** die Exponenten **E1** und **E2**. Sei **E3** das Ergebnis dieser Addition.
2. Man **multipliziert** die Mantissen **M1** und **M2**. Sei **M3** das Ergebnis dieser Multiplikation.
3. Falls die erste Nachpunktstelle von **M3** gleich **0** ist, schiebt man alle Ziffern von **M3** um eine Stelle nach links und subtrahiert 1 von E3. Dadurch **normalisiert** man das Ergebnis **M3**, **E3** der Multiplikation.
4. Eventuell muß man von **M3** ganz rechts noch ein paar Stellen abschneiden. Manche Ausführer runden dabei, andere runden nicht.

Beim Dividieren kann man ganz entsprechend die Exponenten **subtrahieren** und die Mantissen **dividieren**.

Aufgabe 21.3.4.: Keine Musterlösung.
Aufgabe 21.3.5.: Keine Musterlösung.

Lösung 22.1.1.: Die Variablen ALFRED, BERTHA und CARL als Bojen:



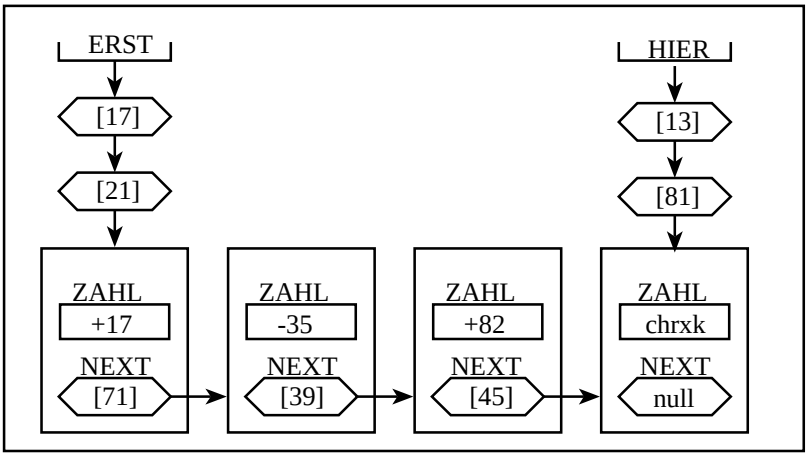
Lösung 22.1.2.: Vereinbarungen der Variablen ARNO, BEATE und CONNY:

```
01 declare
02    ARNO : ZEIGER_AUF_GANZ; -- Koennte auch ZEIGER_AUF_PERLE sein!
03    BEATE: ZEIGER_AUF_GANZ  := new GANZ'(-123);
04    CONNY: ZEIGER_AUF_PERLE := new PERLE'(ZAHL => -273, NEXT =>
05                                 new PERLE'(ZAHL => +123, NEXT => null));
```

Lösung 22.1.3.: Vereinbarungen
1. Der Name ARNO steht für den Zeigerwert **[27]**.
2. Die Variable ARNO hat den Wert **null**.
2a.Der Name BEATE steht für den Zeigerwert **[131]**, die Variable BEATE hat den Wert **[57]**.
2b.Der Name CONNY steht für den Zeigerwert **[43]** und die Variable CONNY hat den Wert [182]
3. Der Zeiger [57] zeigt auf eine namenlose Variable mit dem Wert **-123**.
3a.Der Zeiger [182] zeigt auf eine namenlose Verbundvariable mit dem Wert **(ZAHL => -273, NEXT => [15])**.
3b.Der Zeiger [15] zeigt auf eine namenslose Verbundvariable mit dem Wert **(ZAHL => +123, NEXT => null)**.

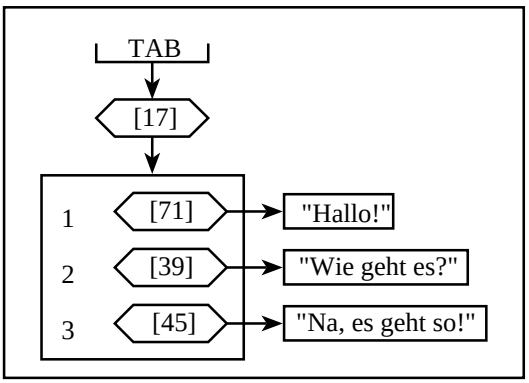
Anmerkung: Ein Aggregat wie in 3a (mit einm Zeiger wie [15] darin) darf **nicht** in einem Ada-Programm verwendet werden.

Lösung 22.1.4.: Die Variablen im Programm ZEIGER_01 nach der Eingabe der vier Ganzzahlen +17, -35, +82 und 0 und bevor der Ausführer die Zahlen in der verketteten Liste wieder ausgibt (siehe Zeile 31):



Aufgabe 22.1.5.: Keine Musterlösung.

Lösung 22.2.1.: Die Reihung TAB als Boje dargestellt:

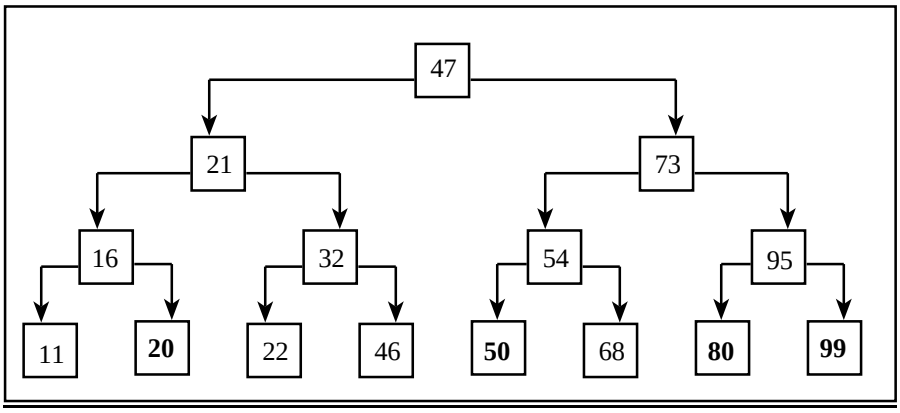


Lösung 22.3.1.: Erlaubte und nicht erlaubt Zuweisungen:

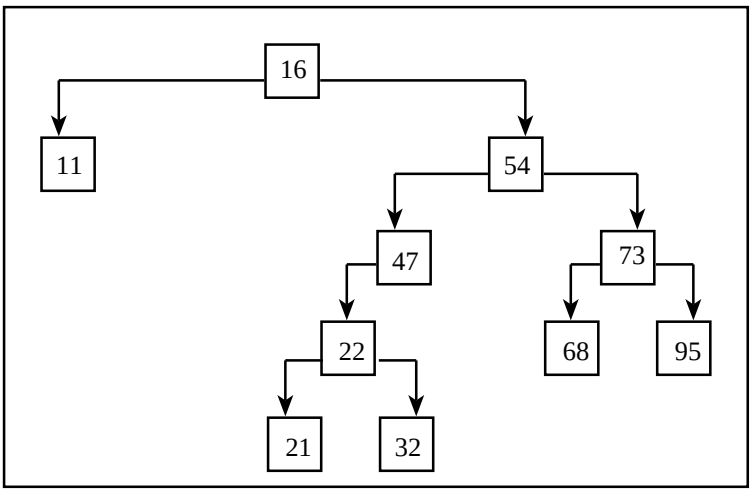
```
01 Z1     := null;                                -- Erlaubt
02 Z1     := Z4;                                  -- Erlaubt
03 Z1.all := "WIE GEHT ES?";                      -- Nicht erlaubt
04 Z2     := null;                                -- Erlaubt
05 Z2     := Z5;                                  -- Erlaubt
06 Z2.all := S1;                                  -- Nicht erlaubt
07 Z3     := null;                                -- Erlaubt
08 Z3     := Z6;                                  -- Erlaubt
09 Z3.all := S2;                                  -- Nicht erlaubt
10 Z4     := null;                                -- Nicht erlaubt
11 Z5     := new string("Mal was anderes!");      -- Nicht erlaubt
12 Z6.all := S2;                                  -- Nicht erlaubt
```

Lösung 22.6.1.: Anfangs zeigen die beiden Zeiger **Z1** und **Z2** auf eine Speicherstelle, an der der String **"Hallo!"** steht. In Zeile 14 wird dieser String an die Speicherverwaltung zurückgegeben und dem Zeiger **Z1** wird der Wert **null** zugewiesen. Aber der Zeiger **Z2** zeigt weiterhin auf die Stelle im Speicher, an der vorher der String **"Hallo!"** gestanden hat. Einige Ada-Ausführer speichern genau an dieser Stelle den neuen String **"hello!"** (siehe Zeile 15), andere Ada-Ausführer verwenden diese Speicherstelle auf andere Weise.

Lösung 22.7.1.: Der binäre Baum nach dem Einfügen von 20, 50, 80 und 99:



Lösung 22.7.2.: Ein sortierter binärer Baum, in den Knoten mit den Schlüsseln 16, 54, 73, 95, 47, 68, 22, 32, 11 und 21 (in dieser Reihenfolge) eingefügt wurden:



Aufgabe 22.7.3.: Keine Musterlösung.

Lösung 22.7.4.: Zur Ebene **n** eines binären Baums können maximal 2^{n-1} Knoten gehören. Zu einem binären Baum mit **n** Ebenen können maximal 2^{n}-1 Knoten gehören.

Lösung 22.7.5.: Um in einer **Liste** von einer Milliarde Knoten einen Knoten mit einem bestimmten Schlüssel zu finden muß man durchschnittlich **eine halbe Milliarde** Suchschritte durchführen. Um in einem sortierten binären **Baum** mit einer Millarde Knoten (die kompakt auf nur 30 Ebenen verteilt sind) einen Knoten mit einem bestimmten Schlüssel zu finden, muß man maximal **30 Suchschritte** durchführen. Allerdings: Wenn die Knoten des Baumes auf viele Ebenen verteilt sind (im Extremfall: auf eine Milliarde Ebenen) braucht man entsprechend mehr Suchschritte.

Lösung 23.1.1.: Primitive Operationen:

```
01 -- Fuer den Typ !BIRNEN:
02 function  MAL2(A: BIRNEN) return BIRNEN;                  -- geerbt von AEPFEL
03 function  MAL5(B: BIRNEN; I: integer) return BIRNEN;
04 procedure LOESCHE(B: in out BIRNEN);

05 -- Fuer den Typ !KIWIS:
06 function  MAL2(A: KIWIS) return KIWIS;                    -- geerbt von BIRNEN
07 function  MAL5(B: KIWIS; I: integer) return KIWIS;        -- geerbt von BIRNEN
```

```
08 procedure LOESCHE(B: in out KIWIS);                        -- geerbt von BIRNEN
09 function  MAL5(K: KIWIS; B: boolean) return KIWIS;
10 procedure LOESCHE(K: in out KIWIS; I: in integer);
```

Lösung 23.2.1.: Siehe Datei **TAG_A_01.ads**.

Lösung 23.2.2.: Primitive Operationen:

```
01 -- Fuer den Typ !PUNKT:
02 function  ABSTAND_U(P   : in     PUNKT)         return GLEIT;
03 procedure SPIEGEL_U(P   : in out PUNKT);
04 procedure PUT      (ITEM: in     PUNKT);
05 -- Fuer den Typ !QUADRAT:
06 function  ABSTAND_U(P   : in     QUADRAT)       return GLEIT;
07 procedure SPIEGEL_U(P   : in out QUADRAT);
08 function  FLAECHE  (Q   : in     QUADRAT)       return GLEIT;
09 function  UMFANG   (Q   : in     QUADRAT)       return GLEIT;
10 procedure PUT      (ITEM: in     QUADRAT);
11 -- Fuer den Typ !RECHTECK:
12 function  ABSTAND_U(P   : in     RECHTECK)      return GLEIT;
13 procedure SPIEGEL_U(P   : in out RECHTECK);
14 function  FLAECHE  (R   : in     RECHTECK)      return GLEIT;
15 function  UMFANG   (R   : in     RECHTECK)      return GLEIT;
16 function  FORMAT   (R   : in     RECHTECK)      return GLEIT;
17 procedure PUT      (ITEM: in     RECHTECK);
18 -- Fuer den Typ !KREIS:
19 function  ABSTAND_U(P   : in     KREIS)         return GLEIT;
20 procedure SPIEGEL_U(P   : in out KREIS);
21 function  FLAECHE  (K   : in     KREIS)         return GLEIT;
22 function  UMFANF   (K   : in     KREIS)         return GLEIT;
23 procedure PUT      (ITEM: in     KREIS);
```

Lösung 23.2.3.: Siehe Datei **TAG_A_01.adb**.
Lösung 23.2.4.: Siehe Datei **TAG_A_02.adb**.

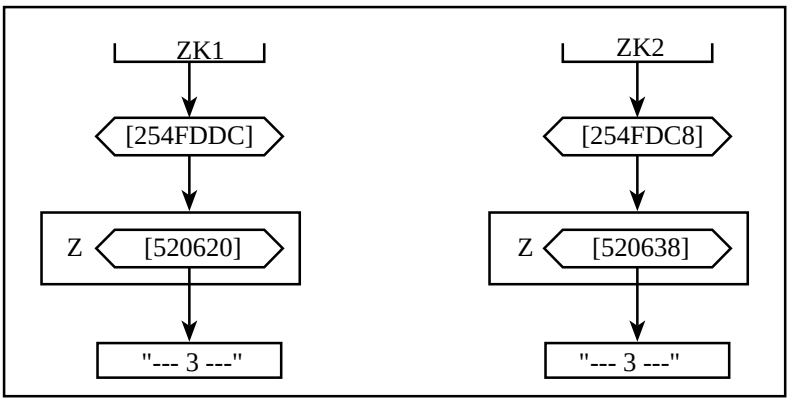
Lösung 23.5.1.: Siehe Datei **TAG_C_02-KIND1.adb**.
In den Prozeduren **MACHE_RECHTECK** und **PUT** (für RECHT_ECKe) kann man sich darauf beschränken, **öffentliche** Unterprogramme des Vatertyps !QUADRAT aufzurufen und man muß **nicht** auf **private** Eigenschaften dieses Vatertyps zugreifen.

Aufgabe 23.5.2.: Keine Musterlösung.
Lösung 23.6.1.: Siehe Dateien **TAG_D_03.ads**, **TAG_D_03.adb**, **TAG_C_04.adb**.

Aufgabe 24.1.1.: Keine Musterlösung.

Lösung 24.1.2.:
Die Variable ZK1 stand bei der Adresse 16#254FDDC#.
Die Variable ZK2 stand bei der Adresse 16#254FDC8#.
Strings wurden bei den Adressen 16#520620# und 16#520638# allokiert.

Wenn der Ausführer beim Ausführen der Prozedur **KON_A_02** bei Zeile 18 angekommen ist, sehen die Variablen **ZK1** und **ZK2** als Bojen dargestellt so aus:



Aufgabe 24.2.1.: Keine Musterlösung.
Aufgabe 24.2.2.: Keine Musterlösung.
Aufgabe 24.3.1.: Keine Musterlösung.
Aufgabe 25.1.1.: Keine Musterlösung.
Lösung 25.1.2.: Siehe Datei **EATXT_02.adb**.
Lösung 25.1.3.: Siehe Datei **EATXT_03.adb**.
Lösung 25.1.4.: Siehe Datei **EATXT_04.adb**.

Lösung 25.2.1.: In jeder der beiden Prozeduren **EASEQ_01** und **EASQ_02** wird ein Typ **!GESCHLECHT** vereinbart, aber in **EASEQ_02** ein bißchen anders als in **EASEQ_01**:

```
09 type GESCHLECHT is (WEIBLICH, MAENNLICH, UNBESTIMMT); -- In EASEQ_02
10 type GESCHLECHT is (MAENNLICH, WEIBLICH, UNBESTIMMT); -- In EASEQ_01
```

Aufgabe 25.2.2.: Keine Musterlösung.
Lösung 25.2.3.: Siehe Datei **EASEQ_05.adb**.
Aufgabe 25.3.1.: Keine Musterlösung.
Aufgabe 25.4.1.: Keine Musterlösung.
Lösung 25.4.2.: Siehe Dateien **EAS_A_07.ads**, **EAS_A_07.adb**, **EAS_A_08.adb**.
Aufgabe 25.4.3.: Keine Musterlösung
Aufgabe 26.1.1.: Keine Musterlösung
Lösung 26.1.2.: Siehe Datei **DARST_02.adb**.
Aufgabe 26.1.3.: Keine Musterlösung
Aufgabe 26.1.4.: Keine Musterlösung
Aufgabe 26.2.1.: Keine Musterlösung
Aufgabe 26.3.1.: Keine Musterlösung
Aufgabe 26.4.1.: Keine Musterlösung
Lösung 26.4.2.: Siehe Datei **BRUCH_15.adb**.
Aufgabe 27.1.1.: Keine Musterlösung
Aufgabe 27.1.2.: Keine Musterlösung
Aufgabe 27.1.3.: Keine Musterlösung
Lösung 27.2.1.: Siehe Datei **TASKS_02.adb** (kommentierte Zeilen).

Lösung 27.2.2.: Der Ausführer beginnt mit der Ausführung der vereinbarten Tasks **ALICE** und **BERTA**, nachdem der alle Vereinbarungen der Prozedur **TASKS_02** abgearbeitet hat, d.h. "wenn der das **begin** in Zeile 53 erreicht".

Aufgabe 27.3.1.: Keine Musterlösung
Lösung 27.4.1.: Siehe Datei **TASKS_13.adb**.
Lösung 27.4.2.: Siehe Datei **TASKS_06.adb**.

Lösung 27.5.1.:
Das Lager bleibt "fast immer" **leer** bei den folgenden Produktions- und Konsumptionszeiten:
```
01 P_ZEIT: array(PRODUKT'range) of duration := (others => 0.5);
```

```
02 K_ZEIT: array(PRODUKT'range) of duration := (others => 0.1);
```

Das Lager ist "fast immer" **voll** bei den folgenden Produktions- und Konsumptionszeiten:

```
01 P_ZEIT: array(PRODUKT'range) of duration := (others => 0.1);
02 K_ZEIT: array(PRODUKT'range) of duration := (others => 0.5);
```

Lösung 27.6.1.: Mein PC (unter **Windows95** mit einem 90-MHz-Pentium und dem **Aonix-Compiler 7.1**) kann in einer Sekunde von 0 bis etwa **6_000_000** oder bis **6_500_000** zählen (bei jedem Aufruf des Programms **TASKS_08** wird ein etwas anderes Ergebnis ausgegeben). Derselbe PC unter **Linux** und mit dem **Gnat-Compiler 3.10p** kann in einer Sekeunde etwa bis **8_500_000** zählen.

Lösung 27.6.2.: Siehe Datei **TASKS_09.adb**.
Aufgabe 27.6.3.: Keine Musterlösung.

Liste der Beispielprogramme, die zum Skript "Ada95" gehören

V = vollständig im Skript wiedergegebens Beispielprogramm, **T** = teilweise im Skript wiedergegeben, **B** = Beispiel (im Skript nicht wiedergegeben, nur besprochen), **L** = Lösung einer Aufgabe

V	HALLO_01	Zwei Zeichen werden eingelesen, Zeichenketten werden ausgegeben.
L	HALLO_02	Zwei Zeichen werden eingelesen und mehrfach wieder ausgegeben
B	HALLO_03	Jedes lexikalische Element steht **allein auf einer Zeile**.
V	HALLO_10	Ruft HALLO_11 und HALLO_12 (mehrmals) auf.
V	HALLO_11	Gibt "Hier ist HALLO_11" aus.
V	HALLO_12	Gibt "Hier ist HALLO_12" aus.
L	HALLO_20	Ruft HALLO_21 und HALLO_22 (mehrmals) auf.
L	HALLO_21	Gibt "----------" aus.
L	HALLO_22	Gibt "| |" aus.
L	HALLO_30	Liest einen String (den Namen des Benutzers, z.B. "R2D2") ein und gibt ihn verziert wieder aus, z.B. so: "Hallo R2D2, wie geht es?.
V	GANZT_01	Ein **signierter Ganzzahltyp** wird vereinbart, zwei Zahlen werden eingelesen und ihre Summe wird ausgegeben.
L	GANZT_02	Variante von GANZT_01, drei Zahlen werden addiert und ausgegeben.
T	GANZT_03	Mehrere **signierte Ganzzahltypen** (AEPFEL, BIRNEN) und Untertypen
T	GANZT_04	Eingaben werden mithilfe eines Untertyps geprüft, eingelesen werden zwei Ganzzahlen. Ihre Summe wird ausgegeben.
L	GANZT_05	Variante von GANZT_04, eine Zahl wird eingelesen, ihr Quadrat wird ausgegeben.
L	GANZT_06	Variante von GANZT_04, eine Zahl wird eingelesen und durch zehn geteilt, das Ergebnis wird ausgegeben.
L	GANZT_07	Der größte und der kleinste Wert eines **Ganzzahltyps** (nicht eines Untertyps!) wird ausgegeben.
B	GANZT_08	Ein **modularer Ganzzahltyp** wird vereinbart, modulare Werte werden eingelesen, addiert, multipliziert etc. und logisch verknüpft mit and, or, xor und not, die Ergebnisse werden ausgegeben.
B	GANZT_09	Werte des Untertyps **integer** werden mit dem Paket **ada.integer_text_io** eingelesen und ausgegeben.
B	GANZT_10	Werte des Untertyps **integer** werden mit einer Instanz der Paketschablone **integer_io** eingelsen und ausgegeben.
L	GANZT_11	Zwei Ganzzahlen G1 und G2 werden eingelesen, die Werte G1 / G2, G1 **rem** G2 und G1 **mod** G2 werden ausgegeben.
T	GANZT_12	**Formatkontrolle** bei der Ein- und Ausgabe von Ganzzahlen (width- und base-Parameter bei OTTO_EA.put und OTTO_EA.get)
T	AUFZT_01	Ein gewöhnlicher **Aufzählungstyp** (WOCHEN_TAG) und Untertypen davon werden vereinbart, Aufzählungswerte werden eingelesen und ausgegeben, der Nachfolger und der Vorgänger eines Wertes wird berechnet.
L	AUFZT_02	Ein Aufzählungstyp FARBE wird vereinbart, ein Wert eingelesen, Vorgänger und Nachfolger ausgegeben.
B	AUFZT_03	Ein **Zeichentyp** STELLUNG wird vereinbart, ein Wert eingelesen, Vorgänger und Nachfolger ausgegeben.
T	AUFZT_04	**Formatkontrolle** bei der Ein- und Ausgabe von Aufzählungswerten (width- und set-Parameter bei FARBE_EA.get und FARBE_EA.put).
L	AUFZT_05	Wochentage auf Deutsch und auf Englisch ausgeben.
T	IFANW_01	Alle Varianten der **if-Anweisung** (if-then, if-then-else, if-then-elsif etc.)
B	IFANW_02	Doppelt **geschachtelte** if-Anweisungen (schwer lesbar!).
L	IFANW_03	Drei Ganzzahlen werden eingelesen, die größte wird wieder ausgegeben.
L	IFANW_04	Fünf Ganzzahlen werden eingelesen, die größte wird wieder ausgegeben.
L	IFANW_05	Drei Zahlen werden eingelesen, die größte wird wieder ausgegeben (**ohne** if-Anweisung)
B	CASEA_01	Einfache **case-Anweisungen**.
L	CASEA_02	Ein Zeichen wird eingelesen und daraufhin untersucht, ob es ein großer/kleiner Vokal/Konsonant oder ein anderes Zeichen ist.

T	LOOPS_01	loop-exit, Zahlen einlesen und summieren.
T	LOOPS_02	loop-exit, den ganzzahligen Logarithmus zur Basis 2 berechnen.
T	LOOPS_03	while-Schleife, läuft eventuell **endlos**.
B	LOOPS_04	for-Schleifen, ohne und mit **reverse**.
T	LOOPS_05	loop-Schleife wird mit **return** beendet.
T	LOOPS_06	loop-Schleife wird mit **Ausnahme** beendet.
B	LOOPS_07	for-Schleifen, **geschachtelte**.
B	LOOPS_08	loop-exit-Schleifen, **geschachtelte**.
V	AUSNA_01	Eine **Ausnahme** wird vereinbart, verschiedene Ausnahmen werden ausgelöst.
V	AUSNA_02	**Drei** Ausnahmen werden vereinbart, ausgelöst und behandelt.
V	AUSNA_03	Ruft AUSNA_01 auf und behandelt die dadurch ausglösten Ausnahmen.
L	AUSNA_04	Versucht solange eine Ganzzahl einzulesen, bis der Benutzer Daten eingibt, die keine Ausnahme **data_error** auslösen.
V	AUSNA_10	Ausnahmen werden in einem **Block** ausgelöst und behandelt.
V	AUSNA_20	Ausnahmen werden in **geschachtelten Blöcken** ausgelöst und behandelt.
V	AUSNA_30	Ausnahmen werden in lokalen **Prozeduren**, die sich gegenseitig aufrufen, ausgelöst und behandelt.
B	AUSNA_40	Ruft AUSNA_41 auf und behandelt die Ausnahme DAS_WAR_EIN_C.
B	AUSNA_41	Ruft AUSNA_42 auf und behandelt die Ausnahme DAS_WAR_EIN_B.
B	AUSNA_42	Löst evtl. eine von drei Ausnahmen aus, behandelt DAS_WAR_EIN_A.
B	REIHE_01	Ein **eingeschränkter Reihungsuntertyp** wird vereinbart, eine Reihung wird ausgegeben.
B	REIHE_02	Eine Reihung von Ganzzahlen wird typisch bearbeitet.
B	REIHE_03	Eingeschränkter Reihungsuntertyp **ZEICHEN_MENGE** wird vereinbart.
B	REIHE_04	Reihungsaggregate, Reihungen im Ganzen bearbeiten, Teilreihungen.
L	REIHE_05	Drei Zeichenketten werden eingelesen und konkateniert ausgegeben.
L	REIHE_06	Die Anzahl der Buchstaben in einer eingelesenen Zeichenkette wird ermittelt.
B	REIHU_01	Ein **uneingeschränkter Reihungsuntertyp** wird vereinbart.
L	REIHU_02	**Römische** Ziffern und Zahlen werden eingelesen und ausgegeben.
T	VERBE_01	Der **eingeschränkte Verbunduntertyp** LAST_WAGEN
T	VERBU_01	Der **uneingeschränkte Verbunduntertyp** KFZ.
L	VERBU_02	Mehrere KFZ-Verbunde einlesen und wieder ausgeben.
B	VERBU_03	Zahlen mit **Einheiten** im Meter-Kilopond-Sekunden-System
B	VERBU_04	Ganz- und Bruchzahlen zu einem (varianten Verbund-) Typ zusammenfassen.
T	UPROS_01	Eine einfache Prozedur DOPPEL_LINIE_1 wird vereinbart und aufgerufen.
T	UPROS_02	Prozedur DOPPEL_LINIE_2 mit zwei in-**Parametern**.
T	UPROS_03	Prozedur GET_ZIFFER mit zwei **out**-Parametern.
T	UPROS_04	Prozedur PLUS3 mit einem **in-out**-Parameter.
L	UPROS_05	Prozedur ADD_MULT mit zwei **in**- und zwei **out**-Parametern.
T	UPROS_06	Werden String-Parameter per **Zeiger** oder per **Kopie** übergeben?
T	UPROS_07	Prozedur PUT_PRIM, wird mit einer **return**-Anweisung verlassen.
T	UPROS_08	Prozedur FUELLE_MIT_01 mit **out**-Parameter vom Untertyp **string**.
T	UPROS_21	Eine einfache Funktion ABSOLUT_KLEINER wird vereinbart und aufgerufen.
L	UPROS_22	Funktion ADD5 mit teilweise vorbesetzten Parametern.
L	UPROS_23	Funktion MAX5 mit vorbesetzten Parametern.
L	UPROS_24	Funktion MAX_TEILER.
L	UPROS_25	Funktion MIN_TEILER.
V	UPROS_26	Operation "&", Ganzzahldivision, die immer abrundet.
V	UPROS_27	Operationen "&", GANZ-Zahlen und Strings zu Strings konkatenieren.
L	UPROS_28	Die Funktionen A_NACH_B, A_NACH_BB und AA_NACH_BBB.
V	UPROS_50	Die Prozedur UPROS_50 ruft die Prozedur UPROS_51 auf.
V	UPROS_51	Die Prozedur UPROS_51 ruft die Prozedur UPROS_52 auf.
V	UPROS_52	Die Prozedur UPROS_52 ruft die Prozedur UPROS_51 auf.
V	REKUP_01	Die **rekursive** Funktion **FAKULTAET**.
V	REKUP_02	Die **rekursive** Funktion **FIBO** (zum Bauen von Robotern)
B	REKUP_03	**Iterative** Version von **FIBO** (zum Bauen von Robotern)
V	REKUP_04	Die rekursive Prozedur **SAEGEZAHN** (Ausgabemuster).

L	REKUP_05	Die rekursive Prozedur **DREIECK** (Ausgabemuster).
L	REKUP_06	Anzahl **Plusbäume** berechnen, rekursive Version.
L	REKUP_07	Anzahl **Plusbäume** berechnen, iterative Version.
V	LESIB_01	**Lebensdauer** und **Sichtbarkeit**, Beispiel 1
V	LESIB_02	Sichtbarkeit von gleichnamigen Variablen in **geschachtelten Prozeduren**.
V	LESIB_03	Sichtbarkeit von gleichnamigen Variablen in **geschachtelten Blöcken**
V	LESIB_04	**Direkte** und **indirekte** Sichtbarkeit von Variablen
V	LESIB_05	Ein Prozedurname wird **überladen**.
V	LESIB_06	Geschachtelte **Vereinbarungsbereiche**, insbesondere der Bereich **standard**.
B	LESIB_07	Ein extremer Fall von Sichtbarkeit.
V	PAKET_01	Ein **Stapel** wird in einem Paket als **abstrakte Variable** realisiert.
V	PAKET_02	In einem Paket wird ein konkreter **Stapel-Typ** vereinbart.
T	PAKET_03	In einem Paket wird ein **limitierter**, privater **Stapel-Typ** vereinbart.
T	PAKET_04	In einem Paket werden **Konstanten** eines privaten Ganzzahltyps vereinbart.
T	PAKET_05	Ein Paket stellt den Typ FARBE und das Paket FARBE_EA zur Verfügung.
T	PAKET_06	Paket zum Rechnen mit **rationalen Zahlen**, limitierter priv. Typ RATIONAL.
?	PAKET_07	Paket mit einem Ganzzahluntertyp und 2 Konkatenationsfunktionen **"&"**
V	TESTP_01	Testprogramm für PAKET_01
V	TESTP_02	Testprogramm für PAKET_02
B	TESTP_03	Testprogramm für PAKET_03
B	TESTP_04	Testprogramm für PAKET_04
V	TESTP_05	Testprogramm für PAKET_05
T	TESTP_06	Testprogramm für PAKET_06
B	TESTP_07	Testprogramm für PAKET_07
L	BEN1P_01	Eine Zeichenkette wird eingelesen und rumgedreht wieder ausgegeben.
L	BEN2P_01	Eine Zeichenkette wird eingelesen und rumgedreht wieder ausgegeben, Version 2.
L	BEN3P_01	Hilfsprozedur für BEN2P_01.
L	BEN4P_01	Prueft, ob alle Klammern in einer Zeichenkette "richtig gepaart" sind.
L	BEN1P_02	Eine Zeichenkette wird eingelesen, zweimal rumgedreht und wieder ausgegeben.
V	SCHAB_01	Prozedurschablone zum **Vertauschen** von zwei Variablen.
T	SCHAB_02	Funktionsschablone zum **Potenzieren**
T	SCHAB_03	Paketschablone mit zwei Konkatenationsfunktionen **"&"**
L	SCHAB_04	Funktionsschablone, **zyklische Nachfolgerfunktion** für diskrete Typsn
T	SCHAB_05	Prozedurschablone zum **Sortieren** von Reihungen mit diskreten Komponenten.
T	SCHAB_06	Paketschablone ohne Parameter, **Stapel** wie in PAKET_01.
T	SCHAB_07	Funktionsschablone mit zwei Funktionsparametern, **komponiert** sie.
V	SCHAB_08	Prozedurschablone zum **Sortieren** von Reihungen mit beliebigen Komponenten.
T	SCHAB_09	Paketschablone zum Ein-/Ausgeben von Reihungen.
T	SCHAB_10	Funktionsschablone, **komponiert** 2 Funktionen wann immer das möglich ist.
T	SCHAB_11	Paketschablone, Stapel mit beliebigem **Elementtyp**.
T	SCHAB_12	Funktionsschablone, zweistellige Funktion plus Wert gleich einstellige Funktion.
?	SCHAB_13	Prozedurschablone mit zwei **Paketen** als formale Paramter.
?	SCHAB_14	Prozedurschablone, Alternative und Kontrast zu SCHAB_13
?	SCHAB_15	Paketschablone, **kombiniert** die Schablonen SCHAB_08 und SCHAB_09.
V	TESTS_01	Testprogramm für SCHAB_01
V	TESTS_02	Testprogramm für SCHAB_02
L	TESTS_03	Testprogramm für SCHAB_03
B	TESTS_04	Testprogramm für SCHAB_04
T	TESTS_05	Testprogramm für SCHAB_05
V	TESTS_06	Testprogramm für SCHAB_06
V	TESTS_07	Testprogramm für SCHAB_07
V	TESTS_08	Testprogramm für SCHAB_08
B	TESTS_09	Testprogramm für SCHAB_09
L	TESTS_10	Testprogramm für SCHAB_10
V	TESTS_11	Testprogramm für SCHAB_11
V	TESTS_12	Testprogramm für SCHAB_12

V	TESTS_13	Testprogramm für SCHAB_13
B	TESTS_14	Testprogramm für SCHAB_14
V	TESTS_15	Testprogramm für SCHAB_15
L	BEN1S_06	Eine Zeichenkette wird eingelesen, zweimal rumgedreht und wieder ausgegeben.
L	BEN1S_08	Eine Tabelle absteigend nach Namen und aufsteigend nach Nummern sortieren.
T	BRUCH_01	Ein **dezimaler Festpunkttyp** DFIX1
T	BRUCH_02	Zwei dezimale Festpunkttypen (DFIX1, DFIX2) unterschiedlicher Genauigkeit.
B	BRUCH_03	**Formatkontrolle** beim Ausgeben von dezimalen Fespunktzahlen.
T	BRUCH_04	Ein **gewöhnlicher Festpunkttyp** GFIX1
B	BRUCH_05	Zwei gewöhn. Festpunkttypen (GFIX1, GFIX2) unterschiedlicher Genauigkeit.
B	BRUCH_06	**Formatkontrolle** beim Ausgeben von gewöhnlichen Festpunktzahlen.
T	BRUCH_07	Ein **Gleitpunkttyp** GLEIT1
B	BRUCH_08	Zwei Gleitpunkttypen (GLEIT1, GLEIT2) unterschiedlicher Genauigkeit.
B	BRUCH_09	**Formatkontrolle** beim Ausgeben von Gleitpunktzahlen.
B	BRUCH_10	Fakultätsfunktion, Unterschied zwischen Ganz-, Festpunkt-, Gleitpunktzahlen.
B	BRUCH_11	Demonstriert **Ungenauigkeit von Gleitpunktzahlen**
B	BRUCH_12	**Runden** von dezimalen Festpunktzahlen
B	BRUCH_15	Stellt 32-Bit Gleitpunktzahlen auf dem Bildschirm als Bitketten dar.
B	BRUCH_16	Stellt 64-Bit Gleitpunktzahlen auf dem Bildschirm als Bitketten dar.
T	ZEIGR_01	Zeiger, **verkettete Liste** von Ganzzahlen.
B	ZEIGR_02	Zeiger, verkettete Liste von **Strings**.
T	ZEIGR_03	Zeiger, **access**, **access all** und **access constant**.
B	ZEIGR_04	Zeiger, verkettete Liste von **vereinbarten** (aliased) Strings.
T	ZEIGR_05	Zeiger, Prozedur mit **Zugriffsparameter** (access parameter)
T	ZEIGR_06	Zeiger, auf **Unterprogramme** (Prozeduren und Funktionen)
T	ZEIGR_07	Zeiger, fehlerhafte Speicherbereinigung mit **unchecked_deallocation**
T	ZEIGR_08	Zeiger, ein **binärer Baum**, Knoten **einfügen** und **suchen**.
T	TAG_A_01	Paket mit **etikettierten Typen** PUNKT, QUADRAT, RECHTECK, KREIS.
L	TAG_A_02	Prozedur zum Testen von TAG_A_01.
T	TAG_B_01	Paket, wie TAG_A_01, aber mit **privaten** etikettierten Verbundtypen.
B	TAG_B_02	Prozedur zum Testen von TAG_B_01
T	TAG_C_01	Paket mit privatem etikettierten Verbundtyp **PUNKT**.
V	TAG_C_02	Paket mit priv. etikett. Typ **QUADRAT** (Erweiterung von PUNKT)
T	TAG_C_02.KIND1	**Kindpaket** mit Typ **RECHTECK** (Erweiterung von QUADRAT).
B	TAG_C_03	Prozedur zum Testen von TAG_C_01, TAG_C_02 und TAG_C_02.KIND1.
V	TAG_D_01	Paket mit **abstraktem** Typ SPEICHERBAR.
V	TAG_D_02	Paket mit Typ SPEICHERBAR1 (Erweiterung von SPEICHERBAR).
L	TAG_D_03	Paket mit Typ SPEICHERBAR2 (Erweiterung von SPEICHERBAR).
V	TAG_C_04	Prozedur zum Testen von TAG_D_01, TAG_D_02 und TAG_D_03.
T	KON_A_01	Paket mit kontrolliertem Typ ZEICHEN_KETTE, Ausgabe von Strings.
B	KON_B_01	Paket mit kontrolliertem Typ ZEICHEN_KETTE, Ausgabe von **Adressen**.
T	KON_C_01	Paket mit **limitiertem** kontrollierten Typ ZEICHEN_KETTE
T	KON_D_01	Paket mit kontrolliertem Typ ZEICHEN_KETTE, **Referenzsemantik**.
V	KON_A_02	Prozedur zum Testen von KON_A_01.
B	KON_B_02	Prozedur zum Testen von KON_B_01.
V	KON_C_02	Prozedur zum Testen von KON_C_01.
V	KON_D_02	Prozedur zum Testen von KON_D_01.
B	EATXT_01	E/A in Textform, einfaches Beispiel.
B	EATXT_02	E/A in Textform, die Funktionen set_output, set_input, standard_output etc..
B	EATXT_03	E/A in Textform, der Dateimodus **append_file** zum Verlängern einer Datei.
L	EATXT_04	Anzahl Zeilen und Zeichen einer Textdatei zählen.
B	EASEQ_01	Sequentielle E/A, gibt Verbunde des Typs PERSON in eine Datei aus.
B	EASEQ_02	Sequentielle E/A, liest Verbunde des Typs PERSON aus einer Datei ein.
B	EASEQ_03	Sequentielle E/A, gibt **variante Verbunde** in eine Datei aus.
B	EASEQ_04	Sequentielle E/A, gibt **Strings** unterschiedlicher Länge in eine Datei aus.
L	EASEQ_05	Anzahl Zeilen und Zeichen in einer sequentiellen Datei zählen.

B	EADIR_01	Direkte E/A, einfaches Beispiel.
T	EAS_A_01	Strom E/A, Paket mit neuem **Stromtyp** SCHLANGE (der wird von allen Prozeduren namens EAS_A_xx benützt)
V	EAS_A_02	Prozedur, schreibt/liest **Strings** mit 'write/'read und 'output/'input in/aus Strom.
B	EAS_A_03	Prozedur, schreibt Werte **verschiedener Typen** in einenn Strom.
B	EAS_A_04	Paket mit etikettierten Typen PERSONEN_WAGEN und LAST_WAGEN.
B	EAS_A_05	Prozedur, schreibt PERSONEN_ - und LAST_WAGEN in einen Strom.
B	EAS_A_06	Prozedur, demonstriert **Typenfehler** beim Benützen eines **Stroms**.
L	EAS_A_07	"**Sichere**" Version des Paketes EAS_A_01 (Stromtyp namens SCHLANGE).
L	EAS_A_08	Prozedur zum Testen von EAS_A_07 (ähnlich wie EAS_A_02).
B	EAS_B_02	Prozedur, wie EAS_A_02, aber mit Strom**datei** statt SCHLANGE-Strom.
B	EAS_B_03	Prozedur, wie EAS_A_03, aber mit Stromdatei statt SCHLANGE-Strom.
B	EAS_B_05	Prozedur, wie EAS_A_05, aber mit Stromdatei statt SCHLANGE-Strom.
B	EAS_B_06	Prozedur, wie EAS_A_06, aber mit Stromdatei statt SCHLANGE-Strom.
T	DARST_01	Darstellungsbefehle, gibt das '**size**-Attribut diverser Größen aus.
L	DARST_02	Darstellungsbefehle, gibt das '**size**-Attribut diverser Größen aus.
B	DARST_03	Darstellungsbefehle, Pragma **pack**, viele Zahlen in einer Bittabelle.
V	DARST_04	Darstellungsbefehle, Pragma **pack** für Verbundtyp.
T	DARST_05	Darstellungsbefehle, **Layout** eines Verbundtyps **bitgenau** festlegen.
T	DARST_06	Darstellungsbefehle, die **Adresse** eines Objekts festlegen.
B	DARST_07	Darstellungsbefehle, **unchecked_conversion**, Ganzzahlen in Bittabelle.
V	TASKS_01	Ganz einfache Tasks.
V	TASKS_02	**Rendezvous** zwischen den Tasks ALICE und BERTA.
V	TASKS_03	Rendezvous an einem Eingang mit einem **Parameter**.
B	TASKS_04	Der **requeue**-Befehl und private Eingänge.
V	TASKS_05	Ein **Tasktyp** mit Diskrimnante.
T	TASKS_06	**Zeiger** auf Tasks und beliebig viele Tasks mit **new** erzeugen.
V	TASKS_07	Produzenten/Konsumenten-Problem mit LAGER-**Task** gelöst.
T	TASKS_08	Asynchrone **select**-Anweisung mit **delay**-Anweisung als Auslöser.
L	TASKS_09	Asynchrone **select**-Anweisung mit **delay**-Anweisung als Auslöser.
T	TASKS_10	Asynchrone **select**-Anweisung mit **Eingangsaufruf** als Auslöser.
V	TASKS_11	Produzenten/Konsumenten-Problem mit **geschütztem Objekt** als LAGER.
T	TASKS_12	Ein geschützter **Typ** wird vereinbart.
L	TASKS_13	Variante von **TASKS_05**, gibt eine Endemeldung aus.