

**Inhaltsverzeichnis**

1. Mit Gentle Parser und Compiler für Vorfahren programmieren.....	3
2. Grammatiken A: Mengen von Dezimalzahlen.....	5
3. Grammatiken B: Einfache Sprachen.....	6
4. In Gentle Aktions- und Bedingungsprädikate programmieren.....	8
5. Terme, Grundterme, Grundspezialfälle etc.....	10
6. Vollständige Syntaxprüfung für die Sprache Alg00.....	11
7. Ein Compiler für die Sprache Alg.....	11
7.1 Ausbaustufe Alg11.....	12
7.2 Ausbaustufe Alg12.....	14
7.3 Ausbaustufe Alg13.....	17
7.4 Ausbaustufe Alg14.....	19
8. Eine Typ-1-Grammatik für die Sprache DOPPELT.....	21
9. Eine Typ-0-Grammatik für Vereinbarungen und Anwendungen.....	22

**Regeln für das Lösen dieser Aufgaben**

1. Bilden Sie eine **Arbeitsgruppen**, zu der (im Normalfall) **2 StudentInnen** gehören. Bearbeiten Sie die folgenden Aufgaben in dieser Gruppe und reden Sie dabei möglichst viel miteinander (möglichst unter Verwendung von Fachbegriffen). Gruppen mit mehr als 2 Mitgliedern können in Ausnahmefällen vom Betreuer Ihres Übungstermins genehmigt werden. Gruppen mit weniger als 2 Mitgliedern ("Einzelkämpfer" und leere Gruppen :-)) sind **nicht zulässig**.
2. Ihre Gruppe muss für **jede Aufgabe** eine **Lösung** entwickeln und dem Betreuer Ihrer Übungsgruppe vorführen, sonst dürfen Sie am Ende des Semesters und am Anfang des nächsten Semesters nicht an der Klausur bzw. Nachklausur teilnehmen.
3. Beim Vorführen einer Lösung müssen **alle Gruppenmitglieder** **persönlich anwesend** sein. Alle Gruppenmitglieder müssen **bereit** und in der Lage sein, ihre Lösung zu erläutern und Fragen dazu zu beantworten.

## 1. Mit Gentle Parser und Compiler für Vorfahren programmieren

**Teil 1:** Schreiben Sie eine (kontextfreie, Typ-2-) Grammatik für die folgende formale Sprache A1:

A1 = {mutter, vater, grossmutter, grossvater, urgrossmutter, urgrossvater, ururgrossmutter, ..., urururururgrossmutter, ... urururururururgrossvater, ...}

Übersetzen Sie Ihre Grammatik dann in entsprechende `nonterm`-Prädikate eines Gentle-Programms namens `ahnen01`. Wenn man dieses Gentle-Programm um ein geeignetes `root`-Prädikat ergänzt, wird es zu einem Parser für die Sprache A1. Das `root`-Prädikat eines Gentle-Programms hat Ähnlichkeit mit dem `main`-Unterprogramm eines C-Programms.

**Teil 2:** Erweitern Sie den Parser `ahnen01` zu einem Compiler namens `ahnen02`, der die Worte der formalen Sprache A1 in natürliche Zahlen (1, 2, 3, ...) übersetzt. Diese Zahlen sollen angeben, wie viele Generationen die betreffenden Ahnen von uns entfernt sind (mutter und vater: Eine Generation, grossmutter und grossvater: 2 Generationen, urgrossmutter und urgrossvater: 3 Generationen etc.).

**Teil 3:** Erweitern Sie den Parser `ahnen01` zu einem Compiler namens `ahnen03`, der die Worte der formalen Sprache A1 in eine "Zwischendarstellung" übersetzt, und zwar in Werte des folgenden Gentle-Typs:

```
1 'type' ZAHNE      -- Zwischendarstellung fuer Ahnen
2   mu             -- Zwischendarstellung fuer mutter
3   va             -- Zwischendarstellung fuer vater
4   ur(ZAHNE)     -- Zwischendarstellung fuer alle anderen Ahnen
```

Hier noch ein paar Worte der Sprache A1 und die Zwischendarstellungen, in die sie übersetzt werden sollen:

Wort aus A1	Zwischendarstellung
mutter	mu
vater	va
grossmutter	ur(mu)
grossvater	ur(va)
urgrossmutter	ur(ur(mu))
urgrossvater	ur(ur(va))
ururgrossmutter	ur(ur(ur(mu)))
ururgrossvater	ur(ur(ur(va)))
...	

**Teil 4:** Erweitern Sie den Parser `ahnen01` zu einem Compiler namens `ahnen04`, der die Worte der formalen Sprache A1 direkt (ohne Verwendung einer Zwischendarstellung) in die entsprechenden englischen Worte (mother, father, grandmother, grandfather, greatgrandmother, greatgrandfather, ..., greatgreatgreatgrandmother, ...) übersetzt. Dabei dürfen Sie das folgende Gentle-Prädikat benutzen:

```
5 'action' conc2(Arg1: STRING, Arg2: STRING -> Erg: STRING)
6 -- Erg ist die Konkatenation von Arg1 und Arg2. Dieses
7 -- Prädikat wurde in der Sprache C programmiert und befindet sich
8 -- in der Datei str_hand.c
```

**Teil 5:** Erweitern Sie den Compiler `ahnen03` zu einem Compiler namens `ahnen05`, der die Worte der formalen Sprache A1 in eine Zwischendarstellung (siehe Aufgabe 3) und die Zwischendarstellung in die entsprechenden englischen Worte (siehe Aufgabe 4) übersetzt.

Der letzte Teil dieser Aufgabe (Teil 6) ist eine Arte "Wiederholung und Zusammenfassung" der vorhergehenden Teile.

**Teil 6:** Schreiben Sie einen Compiler namens `ahnen06` mit folgender Quell- und Zielsprache:

Worte der Quellsprache A2:	Entsprechende Worte der Zielsprache:
Die Mutter von Maria	The mother of Mary
Der Vater von Maria	The father of Mary
Die Mutter von Johann	The mother of John
Der Vater von Johann	The father of John
Die Mutter des Vaters von Maria	A grandmother of Mary
Der Vater der Mutter von Johann	A grandfather of John
Die Mutter der Mutter der Mutter von Maria	A greatgrandmother of Mary
Die Mutter des Vaters des Vaters von Johann	A greatgrandmother of John
...	...

Die Worte der Quellsprache A2 beschreiben alle Vorfahren von zwei Personen namens Maria und Johann.

Gehen Sie beim Lösen dieser Aufgabe wie folgt vor:

**Schritt 1:** Entwickeln Sie (mit Papier und Bleistift, ohne Rechner) eine *Grammatik für die Quellsprache A2* und zeigen Sie die Grammatik dem Betreuer Ihrer Übungsgruppe. Der wird prüfen, ob die Grammatik sich als Basis eines Compilers eignet. Falls er es sinnvoll findet, wird er bestimmte Vereinfachungen der Grammatik mit Ihnen besprechen (damit die folgenden Schritte nicht zu schwierig werden).

**Schritt 2:** Schreiben Sie in Gentle einen Parser P für die Quellsprache (d.h. überführen Sie Ihre Grammatik in die Form eines Gentle-Programms). Lassen Sie vorläufig alle Attribute ("alles was in runden Klammern steht") weg (ähnlich wie beim Parser `ahnen01` im Teil 1 dieser Aufgabe). Testen Sie Ihren Parser!

**Schritt 3:** Entwickeln Sie einen Gentle-Typ namens `Z_AHNE`, dessen Werte sich als Zwischendarstellungen für die Quellprogramme eignen. Dieser Typ sollte dem Typ `ZAHNE` im Teil 3 dieser Aufgabe entsprechen. Welche Informationen, die z.B. in dem Quellwort `Die Mutter des Vaters des Vaters von Maria` drin stecken, sind wichtig, wenn man dieses Quell-Wort in das Ziel-Wort `A greatgrandmother of Mary` übersetzen will? Wie kann man diese "wichtigen Informationen" kompakt als Werte eines Gentle-Typs darstellen?

**Anmerkung:** Der Gentle-Typ `ZAHNE` in Teilaufgabe 3 hat z.B. den "Generationsabstand 17" durch einen ziemlich langen Term der Form `ur(ur(ur(...)))` dargestellt. Könnte man den Abstand 17 nicht etwas einfacher und kompakter darstellen? Und welche weiteren Informationen (zusätzlich zum Generationsabstand) müssen die Werte des neuen Typs `Z_AHNE` enthalten? Erinnern Sie sich noch, wie man in Gentle einen Verbundtyp (engl. record type) vereinbart?

**Schritt 4:** Ergänzen Sie den Parser P (aus Schritt 2) dann um Attribute zu einem Compiler C1, der aus dem Quellprogramm, welches er einliest, eine entsprechende Zwischendarstellung erzeugt (ähnlich wie der Compiler `ahnen03` im Teil 3 dieser Aufgabe). Lassen Sie die Zwischendarstellung mit dem Gentle-Befehl `print` zum Bildschirm ausgeben und testen Sie, ob sie korrekt erzeugt wurde.

**Schritt 5:** Ergänzen Sie den Compiler C1 zu einem Compiler C2, der die Zwischendarstellung in das entsprechende Wort der Zielsprache übersetzt (ähnlich wie im Teil 5 dieser Aufgabe). Testen Sie den Compiler C2.

## 2. Grammatiken A: Mengen von Dezimalzahlen

**Teil 1:** Entwickeln Sie (mit Papier und Bleistift, *ohne* Rechner) für jede der formalen Sprachen FS1 bis FS4 eine (kontextfreie, Typ2-) Grammatik.

**FS1:** Die Dezimalzahlen von 0 bis 99 (Startsymbol: Zahlen0Bis99)

**FS2:** Die Dezimalzahlen von 0 bis 100 (Startsymbol: Zahlen0Bis100)

**FS3:** Die Dezimalzahlen von 0 bis 299 (Startsymbol: Zahlen0Bis299)

**FS4:** Die Dezimalzahlen von 0 bis 255 (Startsymbol: Zahlen0Bis255)

**Teil 2:** Erzeugen Sie (*mit* einem Rechner) aus der Grammatik für die Sprache FS4 mit dem Parsergenerator Accent einen Parser.

### Anmerkungen und Tips:

Die Sprachen FS1 bis FS4 sind Mengen von Dezimalzahlen. Die Dezimalzahlen sollen keine *unnötigen führenden Nullen* haben, d.h. Zeichenketten wie 01 oder 000035 sollen *nicht* ableitbar sein, wohl aber die Zeichenketten 1, 35 und 0 (die einzelne Ziffer 0 ist zwar eine führende Null, aber nicht *unnötig*, weil sie zur Darstellung der Zahl null benötigt wird).

In Ihren Grammatiken dürfen Sie die Zwischensymbole Ziff0Bis9, Ziff1Bis9, Ziff0Bis5 und Ziff0Bis4 benutzen, die durch die folgenden (eher "langweiligen" und deshalb vorgegebenen) Regeln definiert werden:

```
R01: Ziff0Bis4 -> '0'
R02: Ziff0Bis4 -> Ziff1Bis4
R03: Ziff0Bis5 -> Ziff0Bis4
R04: Ziff0Bis5 -> '5'
R05: Ziff0Bis9 -> '0'
R06: Ziff0Bis9 -> Ziff1Bis9
R07: Ziff1Bis4 -> '1'
R08: Ziff1Bis4 -> '2'
R09: Ziff1Bis4 -> '3'
R10: Ziff1Bis4 -> '4'
R11: Ziff1Bis9 -> Ziff1Bis4
R12: Ziff1Bis9 -> '5'
R13: Ziff1Bis9 -> '6'
R14: Ziff1Bis9 -> '7'
R15: Ziff1Bis9 -> '8'
R16: Ziff1Bis9 -> '9'
```

Beginnen Sie die Nummerierung Ihrer Regeln entsprechend mit R17.

### 3. Grammatiken B: Einfache Sprachen

**Teil 1:** Entwickeln Sie (mit Papier und Bleistift, *ohne* Rechner) für jede der formalen Sprachen FS1 bis FS12 eine (kontextfreie, Typ2-) Grammatik an. Die Sprachen werden unten kurz beschrieben. Als Teil der Beschreibung werden zu jeder Sprache einige positive und einige negative Beispiele angegeben (Worte, die zu der Sprache gehören bzw. nicht dazu gehören, nach *ja* bzw. *nein* in den runden Klammern).

In Ihren Grammatiken dürfen Sie die Zwischensymbole Gb (wie Großbuchstabe), Kb (wie Kleinbuchstabe), Ziff (wie Dezimalziffer) und SoZe (wie Sonderzeichen) benutzen, die durch die folgenden (eher "langweiligen" und deshalb vorgegebenen) Regeln definiert werden:

R01: Gb -> 'A'	R27: Kb -> 'a'	R53: Ziff -> '0'	R63: SoZe -> '.'
R02: GB -> 'B'	R28: Kb -> 'b'	R54: Ziff -> '1'	R64: SoZe -> '!'
...	...	...	R65: SoZe -> '?'
R26: Gb -> 'Z'	R52: Kb -> 'z'	R62: Ziff -> '9'	R66: SoZe -> '#'

Außerdem dürfen Sie in jeder Grammatik alle Zwischensymbole benutzen, die Sie selbst in vorhergehenden Grammatiken definiert haben.

**FS01:** Alle nicht-leeren Ziffernfolgen (*ja*: 123, 0, 0007654, *nein*: abc, a123, 123BC, äöüß). Startsymbol: **ZiffFo**.

**FS02:** Alle nicht-leeren Zeichenfolgen. Als Zeichen sollen genau die 66 Zeichen erlaubt sein, die in den oben vorgegebenen Regeln erwähnt werden (*ja*: abc, 123, ABC, ? . !, aB3!, !!CCaa##007, a, B, 0, #, *nein*: abc\$, §123, ( ), ] ] ], a&b, 1+2, sum\*35). Startsymbol: **ZeichFo**.

**Achtung:** Bevor Sie weitermachen, sollten Sie Ihre Grammatik für die Sprache FS02 dem Betreuer Ihrer Übungsgruppe zeigen und mit ihm kurz besprechen! Möglicherweise ist Ihre Grammatik richtig, ähnelt Ihrer Grammatik für die Sprache FS01 aber weniger als es wünschenswert ist.

**FS03:** Alle nicht-leeren Buchstabenfolgen. Als Buchstaben sollen genau die 52 Zeichen erlaubt sein, die in den Regeln R01 bis R52 erwähnt wurden (*ja*: abcd, ABCD, aBcD, aBCd, aAABbb, Hallo *nein*: a1, 1A, Hallo!). Startsymbol: **BuFo**.

**FS04:** Alle nicht-leeren Buchstabenfolgen, die mit einem großen oder kleinen Buchstaben beginnen, ansonsten aber nur aus (0 oder mehr) kleinen Buchstaben bestehen (*ja*: abc, Abc, dddeeffff, Dddeeffff, a, A, *nein*: AB, endSumme, xY). Startsymbol: **GK\_BuFo**.

**FS05:** Alle nicht-leeren Ziffernfolgen ohne unnötige führende Nullen. Dazu gehören alle Ziffernfolgen, die mit einer von 0 verschiedenen Ziffer beginnen und außerdem die Ziffernfolge 0 (weil eine einzelne 0 keine *unnötige* führende Null, sondern eine zur Darstellung der Zahl 0 *nötige* führende Null ist) (*ja*: 0, 1, 7, 123, 1000, 999888777000, *nein*: 00, 07, 007, 00000, +15, -36). Startsymbol: **OFN\_ZiffFo**.

**FS06:** Alle nicht-leeren Folgen von Worten der Sprache FS04, von denen jedes mit einem Semikolon ';' abgeschlossen wurde (*ja*: Butter;Eier;Quark; oder abc;Abc;aabbcc; oder abc; oder Def; oder a;b;C;D; *nein*: abc oder abc;; oder ;abc oder abc;;def). Startsymbol: **SemAbg**.

**FS07:** Alle nicht-leeren Folgen von Worten der Sprache FS04, die durch Kommas ', ' voneinander getrennt sind (*ja*: Butter,Eier,Quark oder abc,Abc,aabbcc oder abc oder Def oder a,b,C,D *nein*: abc,, oder ,abc oder abc,,def). Zur Verdeutlichung: Ein Komma darf also nur zwischen zwei Worten der Sprache FS04 stehen. Startsymbol: **KomGetr**.

**FS08:** Alle nicht-leeren Folgen von Worten der Sprache FS01, die durch Nummernzeichen '#' voneinander getrennt sind (**ja:** 123#45#6789 oder 007#008#9 oder 007 oder 000 oder 9#8#7#6 **nein:** 123#456# oder #123 oder 123##456). Zur Verdeutlichung: Ein Nummernzeichen darf und muss also nur *zwischen* zwei Worten der Sprache FS01 stehen, aber nicht nach dem letzten (oder vor dem ersten) Wort. Startsymbol: NZ\_Getr.

**FS09:** Alle nicht-leeren Folgen von Worten der Sprache FS01, von denen jedes mit einem Nummernzeichen '#' abgeschlossen wurde (**ja:** 123#45#6789# oder 007#008#9# oder 007# oder 000# oder 9#8#7#6# **nein:** 123## oder #123 oder 123##456). Startsymbol: NZ\_Abg.

**FS10:** Alle nicht-leeren Folgen gerader Länge von Worten, für die gilt: Die Worte an ungerader Position (d.h. das 1., 3., 5. ... Wort) stammen aus der Sprache FS04 und die Worte an gerader Position (d.h. das 2., 4, 6. ... Wort) aus der Sprache FS01 (**ja:** abc123 oder x1 oder x1x2y1y2 oder Betrag01Betrag03Summe01 **nein:** 123abc oder abc oder 123). Startsymbol: PaarFo.

**FS11:** Alle nicht-leeren Folgen von Worten, die abwechselnd aus FS01 und FS04 stammen. Das erste (und möglicherweise einzige) Wort einer solcher Folge kann wahlweise aus FS01 oder FS04 stammen (**ja:** 123 oder abc oder 123abc oder abc123 oder 12ab34de56 oder 12ab34de56fg oder Hallo01Sonja02wie03gehts04 **nein:** +123 oder Hallo! oder 123,abc oder abc;123). Startsymbol: Alt0104 ("alternierend aus FS01/FS04").

**FS12:** Wie FS11, aber jedes Wort aus FS01 bzw. FS04 soll durch ein Nummernzeichen '#' abgeschlossen sein (**ja:** 123# oder abc# oder 123#abc# oder abc#123# oder 12#ab#34#de#56# oder 12#ab#34#de#56#fg# oder Hallo#01#Sonja#02#wie#03#gehts#04# **nein:** 123 oder #Hallo oder #Hallo# oder 123## oder abc#123). Startsymbol: NZ\_Getr\_0104.

**Teil 2:** Erzeugen Sie aus der Grammatik für die Sprache FS12 mit dem Parsergenerator Accent einen Parser.

#### 4. In Gentle Aktions- und Bedingungsprädikate programmieren

Ergänzen Sie die Deklarationen der Prädikate `anzElem`, `summe`, ... etc. um geeignete Regeln. Für die Prädikate `anzPosElem`, `anzNegElem`, `posElem` und `negElem` soll gelten: 0 ist *weder* eine positive *noch* eine negative Zahl.

```

1 ----- */
2 'type' LISTE                -- Kann auch leer sein
3   leer
4   k(Element: INT, Rest: LISTE)
5 -----
6 'action'  anzElem(L: LISTE -> INT)
7   -- Ermittelt die Anzahl der Zahlen in der Liste L.
8 -----
9 'action'  summe (L: LISTE -> INT)
10  -- Berechnet die Summe aller Zahlen in der Liste L.
11 -----
12 'action'  anzPosElem(L: LISTE -> INT)
13  -- Ermittelt die Anzahl der positiven Zahlen in der Liste L.
14 -----
15 'action'  anzNegElem(L: LISTE -> INT)
16  -- Ermittelt die Anzahl der negativen Zahlen in der Liste L.
17 -----
18 'action'  summePosElem(L: LISTE -> INT)
19  -- Berechnet die Summe aller positiven Zahlen in der Liste L.
20 -----
21 'action'  summeNegElem(L: LISTE -> INT)
22  -- Berechnet die Summe aller negativen Zahlen in der Liste L.
23 -----
24 'action'  maxPosElem(L: LISTE -> INT)
25  -- Ermittelt die groesste positive Zahl in L
26  -- (und 0, falls L keine positive Zahl enthaelt)
27 -----
28 'action'  minNegElem(L: LISTE -> INT)
29  -- Ermittelt die kleinste negative Zahl in L
30  -- (und 0, falls L keine negative Zahl enthaelt)
31 -----
32 'action'  mod(DEND: INT, DOR: INT -> Rest: INT)
33  -- Berechnet den Rest der uebrig bleibt, wenn man
34  -- DEND durch DOR teilt.
35 -----
36 'condition'  istGerade (DEND: INT)
37  -- Gelingt genau dann wenn DEND eine gerade Zahl ist.
38 -----
39 'condition'  istUngerade(DEND: INT)
40  -- Gelingt genau dann wenn N eine ungerade Zahl ist.
41 -----
42 'action'  anzGeradeElem(L: LISTE -> INT)
43  -- Ermittelt die Anzahl der geraden Zahlen in der Liste L.
44 -----
45 'action'  anzUngeradeElem(L: LISTE -> INT)
46  -- Ermittelt die Anzahl der ungeraden Zahlen in der Liste L.
47 -----
48 'action'  anzPosNegElem(L: LISTE -> AnzPos: INT, AnzNeg: INT)
49  -- Liefert die Anzahl der positiven und die Anzahl der negativen
50  -- Zahlen in L.
51 -----
52 'action'  anzPosGeradeElem(L: LISTE -> AnzPos: INT, AnzGerade: INT)
53  -- Ermittelt die Anzahl der positiven und die Anzahl der geraden
54  -- Zahlen in L.
55 -----
56 'action'  posElem(L: LISTE -> LP: LISTE)
57  -- Berechnet eine Liste LP aller positiven Zahlen in L
58 -----
59 'action'  negElem(L: LISTE -> LN: LISTE)
60  -- Berechnet eine Liste LN aller negativen Zahlen in L

```

```
61 -----
62 'action' posNegElem(L: LISTE -> LP: LISTE, LN: LISTE)
63 -- Berechnet eine Liste LP aller positiven und eine Liste LN aller
64 -- negativen Zahlen in L
65 -----
66 'action' geradeElem(L: LISTE -> LG: LISTE)
67 -- Berechnet eine Liste LG aller geraden Zahlen in L
68 -----
69 'action' anzDoppelte(L: LISTE -> Anzahl: INT)
70 -- Wie oft stehen in der Liste L zwei gleiche Zahlen nebeneinander?
71 -- Dieses Praedikat berechnet die Antwort.
72 -- Achtung: Die "Zweiergruppen" sollen sich nicht ueberlappen. D.h.
73 -- Wenn drei gleiche Zahlen nebeneinander stehen, dann zaehlen sie
74 -- nur als EINE Zweiergruppe (und eine einzelne Zahl). Erst vier
75 -- gleiche Zahlen nebeneinander sollen als ZWEI Zweiergruppen
76 -- gezaehlt werden.
77 -----
78 'action' anzDreimalige(L: LISTE -> Anzahl: INT)
79 -- Wie oft stehen in der Liste L drei gleiche Zahlen nebeneinander?
80 -- Dieses Praedikat berechnet die Antwort.
81 -- Achtung: Die "Dreiergruppen" sollen sich nicht ueberlappen. D.h.
82 -- Wenn vier gleiche Zahlen nebeneinander stehen, dann zaehlen sie
83 -- nur als EINE Dreiergruppe (und eine einzelne Zahl). Erst sechs
84 -- gleiche Zahlen nebeneinander sollen als ZWEI Dreiergruppen
85 -- gezaehlt werden.
86 -----
87 'condition' sindGleich(L1: LISTE, L2: LISTE)
88 -- Gelingt, wenn die Listen L1 und L2 ("Element fuer Element")
89 -- gleich sind
90 -----
91 'condition' enthaeltElem(L: LISTE, Elem: INT)
92 -- Gelingt genau dann, wenn die Liste L die Zahl Elem enthaelt.
93 -----
94 'condition' enthaeltListe(L1: LISTE, L2: LISTE)
95 -- Gelingt genau dann, wenn die Liste L1 die Liste L2 enthaelt
96 -- (d.h. wenn jedes Element von L2 auch in L1 vorkommt)
97 -----
98 'condition' habenGleicheElemente(L1: LISTE, L2: LISTE)
99 -- Gelingt genau dann, wenn
100 -- 1. jedes Element von L1 auch in L2 vorkommt und
101 -- 2. jedes Element von L2 auch in L1 vorkommt.
102 -----
103 'condition' enthaeltDoppelgaenger(L: LISTE)
104 -- Gelingt genau dann, wenn mindestens eine Zahl mehr als einmal
105 -- in L vorkommt (egal, an welcher Stelle).
106 -----
107 'condition' istAufsortiert(L: LISTE)
108 -- Gelingt genau dann, wenn die Liste L aufsteigend sortiert ist
109 -----
```

## 5. Terme, Grundterme, Grundspezialfälle etc.

Gehen Sie von folgenden Typ-Vereinbarungen (in der Sprache Gentle) aus:

```
1 'type' FARBE rot blau gruen
2 'type' BAUM
3 leer
4 b(Vorn: FARBE, Hinten: FARBE, Links: BAUM, Rechts: BAUM)
```

Im folgenden sollen F, F1, F2, ... Variablen vom Typ FARBE und B, B1, B2, ... Variablen vom Typ BAUM sein. Betrachten Sie die folgenden Terme:

```
T1: b(rot, gruen, leer, b(gruen, blau, leer, leer))
T2: leer
T3: b(F1, F2, B1, B2)
T4: blau
T5: b(blau, rot, B1, B2)
T6: b(F1, F2, b(rot, rot, leer, leer), leer)
T7: B
T8: b(rot, F1, b(F1, blau, B1, leer), b(F2, F1, leer, B1))
T9: F2
T10: b(F, rot, leer, leer)
T11: b(rot, F1, leer, b(F2, F3, leer, leer))
T12: b(rot, F1, leer, b(F1, F1, leer, leer))
```

Beantworten Sie folgende Fragen:

- 5.01. Welche der Terme T1 bis T12 sind Grundterme?
- 5.02. Welche der Terme T1 bis T12 gehören zum Typ (oder: sind vom Typ) FARBE?
- 5.03. Geben Sie alle Grundspezialfälle von T10 an.
- 5.04. Geben Sie alle Grundspezialfälle von T9 an.
- 5.05. Geben Sie alle Grundspezialfälle von T4 an.
- 5.06. Geben Sie alle Grundspezialfälle von T1 an.
- 5.07. Wie viele Grundspezialfälle hat der Term T6?
- 5.08. Wie viele Grundspezialfälle hat der Term T11?
- 5.09. Wie viele Grundspezialfälle hat der Term T12?
- 5.10. Wie viele Grundspezialfälle hat der Term T5?
- 5.11. Wie viele Grundspezialfälle hat der Term T7?
- 5.12. Wie viele Grundspezialfälle hat der Term T9?
- 5.13. Welche der Terme T1 bis T12 sind Spezialfälle des Terms T3?
- 5.14. Welche der Terme T1 bis T12 sind Grundspezialfälle des Terms T3?
- 5.15. Welche der Terme T1 bis T12 sind Spezialfälle des Terms T9?
- 5.16. Welche der Terme T1 bis T12 sind Grundspezialfälle des Terms T9?
- 5.17. Geben Sie (irgend) einen Grundspezialfall von T11 an.
- 5.18. Geben Sie (irgend) einen Grundspezialfall von T12 an.
- 5.19. Passt das Muster T11 auf den Wert T1?
- 5.20. Entspricht der Wert T1 dem Muster T12?
- 5.21. Passt das Muster T10 auf den Wert T1?
- 5.22. Passt das Muster T4 auf den Wert T4?

## 6. Vollständige Syntaxprüfung für die Sprache Alg00

Bei den Gentle-Beispielen finden Sie im Verzeichnis `alg00Str` eine unvollständige Gentle-Quelldatei namens `Alg00.g.g`. Erstellen Sie eine Kopie namens `Alg00.g` dieser Datei und ergänzen Sie die Kopie, so dass sie richtig funktioniert.

Bei den Gentle-Beispielen finden Sie im Verzeichnis `Alg00Ide` eine unvollständige Gentle-Quelldatei namens `Alg00.g.g`. Erstellen Sie eine Kopie namens `Alg00.g` dieser Datei und ergänzen Sie die Kopie, so dass sie richtig funktioniert. Diese Teilaufgabe stimmt weitgehend mit der vorigen überein und sollte deshalb mit sehr wenig Arbeit lösbar sein.

## 7. Ein Compiler für die Sprache Alg

Sie sollen einen Compiler schreiben, der Quellprogramme einer Sprache Alg in Zielprogramme der Sprache Jasmin (Java-Assembler-Code) übersetzt. "Alg" soll nicht an "Alkohol" erinnern, sondern an die *Algol-Sprachfamilie*, zu der Algol60, Algol-W, Algol68, Pascal, Modula und Ada gehören.

Den Alg-Compiler sollen Sie in *mehreren Stufen* entwickeln, indem Sie zuerst eine kleine Teilsprache Alg11 implementieren und dann immer größere Teilsprachen (Alg12, Alg13, ...).

Für die *erste Ausbaustufe* Ihres Compilers ist ein Ordner `alg11` vorgegeben, der eine Gentle-Quelldatei `Alg11.g.g` und zahlreiche weitere, (hoffentlich) nützliche Dateien enthält. Die Datei `Alg11.g.g` enthält einen funktionstüchtigen Compiler, der allerdings nur sehr einfache Hallo-Programme übersetzen kann. Sie sollen diesen Compiler so erweitern, dass er alle Alg11-Programme übersetzen kann. Die Sprache Alg11 wird am Anfang der Datei `Alg11.g.g` und im vorliegenden Papier im Abschnitt *Übersicht über die Sprache Alg11* beschrieben (siehe weiter unten).

Für die *weiteren Ausbaustufen* sollen Sie selbst je einen Ordner anlegen (`alg12`, `alg13`, ...) und alle benötigten Dateien hineinkopieren.

Gehen Sie bei der Entwicklung des Compilers nach den folgenden strategischen Grundsätzen vor:

**Strategischer Grundsatz 1:** Entwickeln Sie Ihren Alg-Compiler in *kleinen Schritten*. Dabei ist mit einem *Schritt* eine Erweiterung oder Veränderung des Compilers gemeint, die mit einer *Prüfung* abgeschlossen werden kann. Eine solche Prüfung besteht aus Aktionen wie den folgenden:

A1. Der Alg-Compiler wird neu *erzeugt*.

A2. Mit dem neuen Alg-Compiler wird ein Alg-Quellprogramm `t` in eine Datei `t.j` *übersetzt*.

A3. Die Datei `t.j` wird mit dem Jasmin-Assembler in eine Datei `t.class` *übersetzt*

A4. Die Klasse `t.class` wird (mit einem Interpreter wie `gij.exe` oder `java.exe`) *ausgeführt*.

Die Aktion A1 kann man z.B. mit einem `build`-Skript durchführen, die Aktionen A2 bis A4 mit einem `tst`-Skript.

**Anmerkung:** Ein *build-Skript* ist eines der Skripte `build.cmd` oder `build.bsh`.

Ein *tst-Skript* ist eines der Skripte `tst.cmd` oder `tst.bsh` etc.

Jede Prüf-Aktion liefert Ihnen wichtige Informationen: Eine Erfolgsmeldung oder Fehlermeldungen. In aller Regel gilt: Je früher man auf einen Fehler aufmerksam wird, desto weniger Arbeit kostet seine Beseitigung. Mit kleinen Schritten und häufigen Prüf-Aktionen können Sie sich viel Arbeit sparen.

**Strategischer Grundsatz 2:** Beginnen Sie einen Schritt immer damit, das Alg-Quellprogramm `t` für die Prüf-Aktionen A2 bis A4 zu schreiben. Das Quellprogramm `t` sollte *möglichst wenige Befehle* enthalten (z.B. nur *einen* `write`-Befehl oder eine Variablen-Vereinbarung, einen `read`- und einen `writeln`-Befehl etc.). Wenn `t` fertig ist sollten Sie sich fragen: "An welchen Stellen muss ich den Quellcode meines Alg-Compilers erweitern/verändern, damit er `t` bearbeiten kann?"

**Strategischer Grundsatz 3:** Wenn Ihr Alg-Compiler in einem Quellprogramm einen *Fehler* entdeckt (z.B. eine Variable, die *weniger als 1 Mal* oder *mehr als 1 Mal* vereinbart wurde) soll er ihn melden und sich dann *sofort beenden*.

Diese Strategie *Nur den ersten Fehler erkennen und melden* ist deutlich einfacher zu programmieren als *Alle Fehler erkennen und melden*, ermöglicht eine einfachere Struktur des Compilers und vermeidet überflüssige und verwirrende Meldungen über *Folgefehler*.

**Strategischer Grundsatz 4:** Speziell für die Übersetzung von Ausdrücken gilt: Jeder Ausdruck A soll in Jasmin-Befehle übersetzt werden, die den Wert von A zuoberst auf den Stapel (der Java Virtual Machine) legen.

**Strategischer Grundsatz 5:** Wenn das Einhalten der vorangehenden strategischen Grundsätze sich nachteilig auf die Entwicklungsarbeit auswirken würde, sollte man die Grundsätze ignorieren und anders vorgehen.

## 7.1 Ausbaustufe Alg11

Öffnen Sie mit einem Editor die Datei `..\cb\bspGentle\alg11\LiesMich.txt` und führen Sie die dort beschriebenen Ersten Schritte (mit einer cmd-Shell und mit einer bash-Shell) aus.

Erweitern Sie dann den Compiler `Alg11.g` so, dass er alle Befehle übersetzen kann, die in seinem Anfangskommentar erwähnt werden. Eine etwas abstraktere Übersicht über die Sprache Alg11 finden Sie im vorliegenden Papier weiter unten. Beachten Sie beim Implementieren der einzelnen Alg11-Befehle die oben erwähnten strategischen Grundsätze und die folgenden Tipps:

**Tipp 1:** Implementieren Sie zuerst `write-` und `writeln-`Befehle mit einem `int`-Literal, einem `bool`-Literal oder einem `string`-Literal als Parameter. Als erster Schritt sind `write-`Befehle mit einem `string`-Literal empfehlenswert (`writeln-`Befehle mit so einem Parameter sind ja schon implementiert). Siehe dazu auch die Methoden `pInt`, `pBool`, `pString`, `plnInt`, `plnBool`, `plnString` im Alg-Laufzeitsystem `ALS3.java` im Ordner `lib`.

**Tipp 2:** Bevor Sie Befehle implementieren, in denen *Variablen-Bezeichner* vorkommen, sollten Sie die Schnittstelle zur Symboltabelle (d.h. die Prädikate `Definiere` und `MeldeWennUndefiniert`) programmieren, so wie Sie es im Zusammenhang mit dem Compiler `Alg00` gelernt haben.

**Tipp 3:** Implementieren Sie dann Variablen-Vereinbarungen, erst für *einen* Typ (z.B. `int`) und dann für die anderen beiden Typen.

**Tipp 4:** Die Reihenfolge, in der Sie dann die restlichen Befehle implementieren, können Sie frei wählen. Die restlichen Befehle sind: `write-` und `writeln-`Befehle mit einem Variablen-Bezeichner als Parameter und `read-`Befehle (die haben immer einen Variablen-Bezeichner als Parameter).

## Übersicht über die Sprache Alg11

Es gibt drei Variablen-Typen: `int`, `bool`, `string`. Werte dieser Typen können von einer Standard-eingabe (Tastatur) eingelesen und zu einer Standardausgabe (Bildschirm) ausgegeben werden.

Ein Programm besteht aus einer (möglicherweise leeren) Folge von Befehlen. Jeder Befehl wird mit einem Semikolon `;` abgeschlossen.

Ein *Befehl* kann sein:

- |                                  |  |
|----------------------------------|--|
| Eine Vereinbarung                | -- vereinbart eine Variable des Typs <code>int</code> , <code>bool</code> oder <code>string</code> . |
| Ein <code>read-</code> Befehl    | -- liest einen Wert in eine Variable   |
| Ein <code>write-</code> Befehl   | -- gibt den Wert eines Ausdrucks aus (ohne Zeilenwechsel)  |
| Ein <code>writeln-</code> Befehl | -- gibt den Wert eines Ausdrucks aus (mit Zeilenwechsel)   |

Eine *Variable* kann *ohne* oder *mit* Initialisierung vereinbart werden. Ohne Initialisierung vereinbarte Variablen werden automatisch mit `0`, `false` bzw. dem String `"nicht initialisiert"` initialisiert.

Ein *Ausdruck* ist entweder ein Literal oder ein Variablen-Bezeichner. Ein Literal bezeichnet einen `int`-Wert, einen `bool`-Wert oder einen `string`-Wert.

Es gibt zwei `bool`-Literale: `false` und `true`. Da die JVM (Java Virtual Machine) keinen Typ `bool` oder `boolean` etc. kennt, werden `bool`-Werte (in übersetzten Alg-Programmen) durch die `int`-Werte 0 (für `false`) und 1 (für `true`) repräsentiert.

Ein Kommentar beginnt mit `--` und reicht bis zum Ende der aktuellen Zeile. Es gibt keine Mehrzeilenkommentare.

**Beispiel für ein Alg11-Programm (mit vielen Kommentaren):**

```
1 int n; -- Vereinbarung ohne Initialisierung
2 write("Eine Ganzzahl n? "); -- Ausgabe ohne Zeilenwechsel
3 read(n); -- Einen int-Wert nach n lesen
4 write("n ist gleich "); -- Ausgabe ohne Zeilenwechsel
5 writeln(n); -- Ausgabe mit Zeilenwechsel
6 int m := 17; -- Vereinbarung mit Initialisierung
7 bool b := true; -- Vereinbarung mit Initialisierung
8 string txt1 := "m ist gleich "; -- Vereinbarung mit Initialisierung
9 string txt2 := "b ist gleich "; -- Vereinbarung mit Initialisierung
10 write(txt1); writeln(m); -- Ausgabe ohne und mit Zeilenwechsel
11 write(txt2); writeln(b); -- Ausgabe ohne und mit Zeilenwechsel
```

## 7.2 Ausbaustufe Alg12

Legen Sie im Ordner `bspGentle` einen Ordner `alg12` an.

Kopieren Sie aus dem Ordner `alg11` alle nützlichen Skripte und die Gentle-Datei `Alg11.g` in den Ordner `alg12`.

Ändern Sie den Namen der Datei `Alg11.g` zu `Alg12.g`.

Ersetzen Sie in allen Skripten den Namen `Alg11` durch `Alg12`

(öffnen Sie dazu alle Skripte mit dem TextPad, drücken Sie F8, tragen Sie die nötigen Strings ein und wählen Sie Bereich: Alle Dokumente).

Der Compiler Alg12 soll zahlreiche neue *Ausdrücke* und außerdem *Zuweisungsbefehle* erkennen und übersetzen können. Beispiele für neue Befehle:

```
write(true or false);  -- Alter write-Befehl      mit neuem Ausdruck
writeln(2*n + 123);   -- Alter writeln-Befehl   mit neuem Ausdruck
n := 17;              -- Neuer Zuweisungsbefehl mit altem Ausdruck
n := n + 4;          -- Neuer Zuweisungsbefehl mit neuem Ausdruck
```

In Alg11 waren als Ausdrücke nur Literale und Variablen-Bezeichner erlaubt und man konnte noch nicht einmal den Wert von `1 + 1` berechnen lassen. In Alg12 sollen Ausdrücke auch die folgenden Operatoren enthalten dürfen:

Nr.	Operator	linker Operand	rechter Operand	Ergebnis	Bindungsstärke
1	<code>or</code>	<code>bool</code>	<code>bool</code>	<code>bool</code>	0
2	<code>and</code>	<code>bool</code>	<code>bool</code>	<code>bool</code>	1
3	<code>not</code>	<code>--</code>	<code>bool</code>	<code>bool</code>	2
4	<code>&lt;</code>	<code>int</code> <code>bool</code> <code>string</code>	<code>int</code> <code>bool</code> <code>string</code>	<code>bool</code>	3
5	<code>&lt;=</code>				
6	<code>=</code>				
7	<code>!=</code>				
8	<code>&gt;=</code>				
9	<code>&gt;</code>				
10	<code>+</code>	<code>int</code>	<code>int</code>	<code>int</code>	4
11	<code>-</code>				
12	<code>*</code>	<code>int</code>	<code>int</code>	<code>int</code>	5
13	<code>/</code>				
14	<code>-</code>	<code>--</code>	<code>int</code>	<code>int</code>	6
15	<code>&amp;</code>	irgendein Typ	irgendein Typ	<code>string</code>	7
16	Literale, Variablen-Bezeichner und Ausdrücke in runden Klammern				8

Tabelle 1: Die Operatoren der Sprache Alg

Bei den Vergleichsoperatoren (4 bis 9) müssen beide Operanden *zum selben Typ* gehören, sonst liegt ein Typfehler vor. Der `bool`-Wert `false` ist kleiner als `true`. Siehe dazu auch die Methoden `lessThan`, `lessEquals`, `equals` etc. in der Datei `ALS3.java` im Ordner `lib`.

Beim Konkatenationsoperator (15) dürfen die Operanden unabhängig voneinander zu irgendeinem der Typen `int`, `bool` oder `string` gehören. `int`- und `bool`-Operanden werden in `string`-Werte umgewandelt und dann konkateniert. Siehe dazu auch die Methode `intToString` und `boolToString` in der Datei `ALS3.java` im Ordner `lib`.

Speziell für die Erweiterung von Alg11 zu Alg12 sind folgende Schritte *sehr empfehlenswert*:

**Schritt 1:** Entwickeln Sie *mit Papier und Bleistift* (und ohne sich von einem eingeschalteten Rechner ablenken zu lassen) eine *Grammatik für Ausdrücke*. In den Ausdrücken sollen natürlich die 15 Operatoren

aus der obigen Tabelle vorkommen dürfen. Außerdem sollen (Variablen-) *Bezeichner*, *Literale* und *Ausdrücke in runden Klammern* erlaubt sein.

Verwenden Sie in der Grammatik (nur) die folgenden Zwischensymbole:

KS\_Ausdruck (als Startsymbol), KS\_Ausdruck1, KS\_Ausdruck2, ..., KS\_Ausdruck8, Bezeich, GanzLiteral und StringLiteral.

Als Prüfung dieses Schrittes sollten Sie die Grammatik möglichst dem Betreuer Ihrer Übungsgruppe zeigen und mit ihm diskutieren. Falls Sie schon dadurch Fehler feststellen, kann Ihnen das viel Arbeit ersparen.

**Schritt 2:** In der Datei Alg12.g die Definition des Typs AS\_Ausdruck so erweitern, dass man (mit Werten dieses Typs) auch *or*-Ausdrücke (z.B. *false or true* oder *b1 or false* etc.) intern darstellen kann. Dazu ist nur *eine* neue Zeile nötig! Nach dem Einfügen dieser Zeile sollte man als Prüfung ein *build*-Skript aufrufen.

**Schritt 3:** Wenn der Schritt 2 für den *or*-Operator geklappt hat, kann man den Typ AS\_Ausdruck (unter Missachtung des Strategischen Grundsatzes 1 und nach dem Strategischen Grundsatz 5 :-)) *in einem Schritt* für alle anderen Operatoren (*and*, *not*, *<*, *...*, *&*) erweitern. Dazu sind ungefähr 14 Zeilen nötig. Nach dem Einfügen dieser Zeilen sollte man aber unbedingt wieder eine Prüfung durchführen, indem man ein *build*-Skript aufruft.

**Schritt 4:** Ersetzen Sie (in der Datei Alg12.g) die erste Zeile der Definition des *nonterm*-Prädikates KS\_Ausdruck:

```
1 'nonterm' KS_Ausdruck(-> AS_Ausdruck)
```

durch die folgenden 19 Zeilen (d.h. durch die trivialen Regeln einer Ausdrucks-Grammatik mit Bindungsstärken von 0 bis 8):

```
1 'nonterm' KS_Ausdruck(-> AS_Ausdruck)
2   'rule' KS_Ausdruck(-> AUS): KS_Ausdruck1(-> AUS)
3
4 'nonterm' KS_Ausdruck1(-> AS_Ausdruck)
5   'rule' KS_Ausdruck1(-> AUS): KS_Ausdruck2(-> AUS)
6 'nonterm' KS_Ausdruck2(-> AS_Ausdruck)
7   'rule' KS_Ausdruck2(-> AUS): KS_Ausdruck3(-> AUS)
8 'nonterm' KS_Ausdruck3(-> AS_Ausdruck)
9   'rule' KS_Ausdruck3(-> AUS): KS_Ausdruck4(-> AUS)
10 'nonterm' KS_Ausdruck4(-> AS_Ausdruck)
11   'rule' KS_Ausdruck4(-> AUS): KS_Ausdruck5(-> AUS)
12 'nonterm' KS_Ausdruck5(-> AS_Ausdruck)
13   'rule' KS_Ausdruck5(-> AUS): KS_Ausdruck6(-> AUS)
14 'nonterm' KS_Ausdruck6(-> AS_Ausdruck)
15   'rule' KS_Ausdruck6(-> AUS): KS_Ausdruck7(-> AUS)
16 'nonterm' KS_Ausdruck7(-> AS_Ausdruck)
17   'rule' KS_Ausdruck7(-> AUS): KS_Ausdruck8(-> AUS)
18
19 'nonterm' KS_Ausdruck8(-> AS_Ausdruck)
```

Da durch diese Ersetzung *die* Regeln, die vorher zum Prädikat KS\_Ausdruck gehörten (ca. 6 Stück), jetzt zum Prädikat KS\_Ausdruck8 gehören, muss man sie entsprechend anpassen. Diese Regeln beschreiben die konkrete Syntax von Literalen, Variablen und Ausdrücken-in-runden-Klammern.

Rufen Sie nach diesem Schritt als Prüfung ein *build*-Skript auf.

Die vorangehenden Schritte 1 bis 4 waren Vorbereitungen. Jetzt sollen Sie die 15 Operatoren (siehe oben Tabelle 1) erst einzeln, dann evtl. in kleinen Gruppen, implementieren. Hier wird vorgeschlagen, als erstes den Operator *or* zu implementieren, etwa so:

**Schritt 5:** Schreiben Sie ein möglichst einfaches Alg12-Quellprogramm namens t12\_01, in dem ein *oder-Ausdruck* vorkommt. Die Datei t12\_01 kann z.B. folgende 2 Zeilen enthalten:

```
-- Alg-Quellprogramm t12_01
writeln(false or true);
```

Korrekt übersetzt sollte dieses Programm den Wert `true` ausgeben.

**Schritt 6:** Fügen Sie in die Definition des `nonterm`-Prädikates `KS_Ausdruck` eine weitere Regel für Ausdrücke ein, in denen der Operator `or` vorkommt. Der `or`-Operator hat die kleinste Bindungsstärke (nämlich 0) und gehört deshalb zu `KS_Ausdruck` (und nicht zu `KS_Ausdruck1` oder `KS_Ausdruck5` etc.).

**Schritt 7:** Erweitern Sie die Definition des `condition`-Prädikates `istBool` so, dass es auch auf oder-Ausdrücke zutrifft. Als Prüfung sollten Sie wieder ein `build`-Skript aufrufen.

**Schritt 8:** Erweitern Sie die Definition des `action`-Prädikates `ausAusdruck` um eine Regel, die einen oder-Ausdruck in Jasmin-Befehle übersetzt. Diese Befehle sollen den Wert des oder-Ausdrucks zuoberst auf den Stapel legen (nach dem **Strategischen Grundsatz 4**).

Rufen Sie auch nach diesem Schritt als Prüfung ein `build`-Skript auf.

**Schritt 9:** Lassen Sie das Alg-Quellprogramm `t12_01` (siehe Schritt 1) mit Hilfe eines `tst`-Skripts übersetzen und ausführen. Wenn die Ausgabe `true` auf dem Bildschirm erscheint, ist alles gut gelaufen. Sonst müssen Sie versuchen, die Fehlermeldungen auf dem Bildschirm zu verstehen und zu beseitigen.

**Schritt 10:** Erweitern Sie das Alg-Quellprogramm `t12_01` so, dass durch seine Ausführung der Operator `or` gründlicher getestet wird und das Gelingen oder Misslingen des Tests am Bildschirm leichter zu erkennen ist, etwa so:

```

1 -- Alg-Quellprogramm t12_02
2
3 writeln("");
4 writeln("-----+-----");
5 writeln("  soll  |  ist");
6 writeln("-----+-----");
7 write  ("01 false | "); writeln(false or  false);
8 write  ("02 true  | "); writeln(false or  true );
9 write  ("03 true  | "); writeln( true  or  false);
10 write ("04 true  | "); writeln( true  or  true );
11 writeln("-----+-----");

```

Wiederholen Sie mit diesem aufgebohrten Quell-Programm den Schritt 9.

Mit den Schritten 5 bis 10 haben Sie den Operator `or` implementiert. Ganz entsprechend können Sie jetzt auch alle anderen Operatoren implementieren, und zwar *in beliebiger Reihenfolge*. Wenn Sie gut drauf sind und sich ein bisschen mutig fühlen, können Sie auch mehrere Operatoren auf einmal implementieren (unter sanfter Missachtung des **Strategischen Grundsatzes 1** :-).

Es folgt hier eine *weiche* Empfehlung zur Reihenfolge der Implementierungen (d.h. es wird nicht garantiert, dass Sie sofort in die Hölle kommen, wenn Sie in einer anderen Reihenfolge vorgehen):

Zuerst die booleschen Operatoren `and` (zweistellig) und `not` (einstellig).

Dann die zweistelligen arithmetischen Operatoren `+`, `-`, `*`, `/` und den einstelligen Operator `-`.

Dann den Konkatenationsoperator `&` (zweistellig) für alle Kombinationen von Operanden.

Schließlich die Vergleichsoperatoren (alle zweistellig) für alle erlaubten Kombinationen von Operanden.

**Vor** dem Implementieren der Vergleichsoperatoren sollten Sie die folgenden beiden Tipps dazu zur Kenntnis nehmen.

**Tipp 1 zur Implementierung der Vergleichsoperatoren:**

Behandeln Sie die Vergleichsoperatoren mit Respekt. Implementieren Sie zuerst nur *einen* dieser Operatoren allein (z.B. den kleiner-Operator `<`), nicht gleich mehrere auf einmal.

Da `bool`-Werte intern durch `int`-Werte dargestellt werden, kann man das Vergleichen von `bool`-Werten und das Vergleichen von `int`-Werten "gemeinsam erledigen". Ein einfaches `condition`-Prädikat namens `istIntOderbool` (zusätzlich zu den Prädikaten `istInt` und `istBool`) kann dabei nützlich sein.

Dagegen muss man das Vergleichen von `string`-Werten separat behandeln. Dabei kann und sollte man die Methoden `lessThan`, `lessEquals`, `equals`, ... etc. im Alg-Laufzeitsystem `ALS3.java` (im Ordner `lib`) zu Hilfe nehmen.

Einen Ausdruck mit einem Vergleichsoperator darin übersetzt man u.a. in *Sprungbefehle*. Im Jasmin-Assembler gibt man das *Ziel* eines Sprungbefehls immer als ein *Label* an, etwa so:

```

12 ...
13 label13:           ; Definition des Labels label13
14 ...
15     ifeq karlHeinz17 ; Bedingter Sprung zum Label karlHeinz17
16 ...
17     goto label13    ; Unbedingter Sprung zum Label label13
18 ...
19 karlHeinz17:      ; Definition des Labels karlHeinz17
20 ...

```

Ein Label muss *eindeutig* sein, d.h. es darf (innerhalb einer Methode) nur *einmal* definiert werden. Wenn ein Alg12-Quellprogramm *mehrere* Vergleichsausdrücke enthält, müssen sich die Labels in den Übersetzungen dieser Ausdrücke voneinander *unterscheiden*.

Um eindeutige Labels zu erzeugen, verwenden wir eine globale (veränderbare) `int`-Variable, die üblicherweise `Dollar` heißt. Beim Übersetzen eines Vergleichsausdrucks lesen wir diese Variable und erhöhen sie um 1. Den gelesenen Wert hängen wir an jedes (bei dieser Übersetzung) erzeugte Label hinten dran. Ein eindeutiges Label kann man etwa so anspringen und definieren:

```

21     ...
22     Dollar -> NR           ; NR bekommt z.B. den Wert 37
23     Dollar <- NR+1
24     ...
25     aSI (" if_icmplt kleiner", NR) ; z.B. zum Label kleiner37 springen
26     ...
27     aSIDp("kleiner", NR)       ; z.B. das Label kleiner37 definieren
28     ...

```

Vereinbaren Sie die Variable `Dollar`, am besten gleich unterhalb der Variablen `NaechsterZielbezeichner`, und initialisieren Sie sie im `nonterm`-Prädikat `init` mit 1.

### Tipp 2 zur Implementierung der Vergleichsoperatoren:

Wenn Sie sehr in Zeitnot sind und nicht vorhaben, Ihren fertigen Alg-Compiler zu verkaufen, genügt es, wenn Sie nur zwei der sechs Vergleichsoperatoren implementieren: den kleiner-Operator `<` und den gleich-Operator `=`. Die übrigen Vergleichsoperatoren kann man in Alg-Programmen notfalls durch kompliziertere Ausdrücke und die Operatoren `not` und `and` ersetzen.

Nach der Implementierung aller Operatoren haben Sie eine Pause und eine großzügige Portion Ihres Lieblingsgetränks (Kaffee? Tee? Ahoi-Brause? ...) verdient.

Zum Abschluss dieser Ausbaustufe sollten Sie noch den *Zuweisungsbefehl* implementieren. Das ist etwas einfacher als die Implementierung der vielen Operatoren :-)

**Schritt 1:** Ein Alg12-Quellprogramm schreiben, z.B. eines namens `t12_37`, mit (einer Variablen-Vereinbarung und) einem Zuweisungsbefehl darin.

**Schritt 2:** Den Typ `AS_Befehl` erweitern. Dann wieder ein `build`-Skript aufrufen.

**Schritt 3:** Das `nonterm`-Prädikat `KS_Befehl` erweitern. Dann wieder ein `build`-Skript aufrufen.

**Schritt 4:** Das `action`-Prädikat `ausBefehl` erweitern. Dann wieder ein `build`-Skript aufrufen.

**Schritt 5:** Mit einem `tst`-Skript das Quellprogramm `t12_37` übersetzen und ausführen lassen.

## 7.3 Ausbaustufe Alg13

Ihr Compiler Alg12 kann erheblich mehr Quellprogramme *übersetzen* als Alg11.

Ihr Compiler Alg13 soll mehr Quellprogramme (mit einer Fehlermeldung) *ablehnen* als Alg12.

Für die Compiler Alg11 und Alg12 gilt:

Der (hauptsächlich aus den `token`-Prädikaten erzeugte) *Lexer* erkennt alle Typ-3-Syntaxfehler.

Der (aus den `nonterm`-Prädikaten erzeugte) *Parser* erkennt alle Typ-2-Syntaxfehler.

Mit Hilfe der Symboltabelle haben Sie dafür gesorgt, dass auch bestimmte Typ-1/0-Syntaxfehler erkannt werden (Variablen, die *weniger als 1 Mal* oder *mehr als 1 Mal* vereinbarte wurden).

Es gibt aber noch weitere Typ-1/0-Syntaxfehler, nämlich *Typfehler*. Die können weder vom *Lexer* noch vom *Parser* erkannt werden. Sie machten sich bisher gar nicht, oder durch freundliche Fehlermeldungen wie "Unerlaubter `bool`-Wert: 3" oder durch schwer zu verstehende Ausnahmen des Typs `java.lang.VerifyError` (gemeldet vom Java-Interpreter) bemerkbar. Jeder der folgenden Alg-Befehle enthält einen solchen Typfehler:

```
1 int    n1 := true;
2 int    n2 := "123";
3 bool   b1 := "false";
4 bool   b2 := 0;
5 string s1 := 123;
6 string s2 := true;
7
8 write(17 + true);
9 write(true and "false");
10 write("Endsumme: " + 123);
```

Ihr Compiler Alg13 soll all diese Typfehler erkennen, und zwar mit Hilfe der folgenden Prüf-Prädikate:

```
'action' prfBefehlsFolge(AS_BefehlsFolge)
'action' prfBefehl      (AS_Befehl)
'action' prfAusdruck   (AS_Ausdruck -> AS_Typ)
```

Wenn eines dieser `prf`-Prädikate einen Typfehler entdeckt, soll es (dem Strategischen Grundsatz 3 folgend, siehe oben) eine möglichst informative Meldung ausgeben und dann den Compiler *sofort beenden*. Dazu soll das Prädikat

```
'action' Error(STRING, POS)
```

aufgerufen werden. Es gibt seine beiden Parameter aus und beendet dann das umgebende Programm (z.B. Ihren Compiler `Alg13.exe`) mit dem C-Befehl `exit(1);`.

Warum das Prädikat `prfAusdruck` (nicht nur einen Eingabe-, sondern auch) einen *Ausgabe*-Parameter hat, werden Sie sicher schnell herausfinden, wenn Sie die Prüfung von Ausdrücken wie etwa `or(AUS1, AUS2)` oder `add(AUS1, AUS2)` programmieren.

### Tipp zur Typ-Prüfung von *Ausdrücken*

Ausdrücke wie `add(AUS1, AUS2)`, `sub(AUS1, AUS2)`, ... ähneln sich sehr, und deshalb ähneln sich auch die Regeln für ihre Typ-Prüfung durch das Prädikat `prfAusdruck`. Lästige Wiederholungen kann man vermeiden, wenn man zuerst folgende Hilfsprädikate programmiert:

```
1 -- Fuer Operatoren mit 2 int-Parametern und int-Ergebnis:
2 'action' prfIntInt_Int(
3   AS_Ausdruck, AS_Ausdruck, Opr: STRING, POS -> ErgebnisTyp: AS_Typ)
4
5 -- Fuer Operatoren mit 1 int-Parameter und int-Ergebnis:
6 'action' prfInt_Int(
7   AS_Ausdruck, Opr: STRING, POS -> ErgebnisTyp: AS_Typ)
8
9
10 -- Fuer Operatoren mit 2 bool-Parametern und bool-Ergebnis:
11 'action' prfBoolBool_Boolean(
12   AS_Ausdruck, AS_Ausdruck, Opr: STRING, POS -> ErgebnisTyp: AS_Typ)
13
14 -- Fuer Operatoren mit 1 bool-Parameter und bool-Ergebnis:
15 'action' prfBool_Boolean(
16   AS_Ausdruck, Opr: STRING, POS -> ErgebnisTyp: AS_Typ)
17
18
```

```

19 -- Fuer Operatoren mit 2 Parametern von irgend einem Typ (aber beide
20 -- vom selben Typ) und einem bool-Ergebnis (z.B. Vergleichsoperatoren)
21 'action' prfIrgendIrgend_Bool(
22   AS_Ausdruck, AS_Ausdruck, Opr: STRING, POS -> ErgebnisTyp: AS_Typ)

```

Der Eingabeparameter `Opr` (vom Typ `STRING`) soll den *Namen* des betreffenden Operators enthalten (z.B. `or` oder `+` oder `<` etc.). Er soll es möglich machen, eine "informative Fehlermeldung" auszugeben, wenn bei einer Prüfung ein Typ-Fehler entdeckt wird.

Mit diesen Hilfsprädikaten kann man das Prädikat `prfAusdruck` dann deutlich einfacher programmieren als ohne sie.

Um Ihren Compiler Alg13 zu testen, müssen Sie ziemlich viele kleine Alg-Quellprogramme schreiben, von denen jedes nur *einen* Typ-Fehler enthält (denn der Compiler beendet sich ja, wenn er diesen Fehler erkannt und gemeldet hat). Das ist eine Kehrseite der einfachen Strategie *Nur den ersten Fehler erkennen und melden*. Die `tstc`-Skripte (`tstc.bsh` und `tstc.cmd`) sollen es Ihnen erleichtern, diese vielen kleinen Alg-Quellprogramme zu schreiben (ohne für jedes davon eine eigene Datei anzulegen).

## 7.4 Ausbaustufe Alg14

Ihr Compiler Alg14 soll folgende neuen Befehle erkennen und übersetzen können:

if-Befehle ohne `else`  
 if-Befehle mit `else`  
 loop-break-Schleifen

Es folgen hier konkrete Beispiele für solche Befehle. Darin wird vorausgesetzt, dass `int`-Variablen `x`, `y` und `z` bereits vereinbart wurden:

```

1 if x < 0 then x := -1; end ;
2
3 if x < y then
4   z := x;
5   write(y);
6 else
7   z := y;
8   write(x);
9 end ;
10
11 loop
12   write("Eine positive Ganzzahl? ");
13   read(x);
14   break when x < 0;
15   writeln("2*x ist " & (2*x));
16   break when 2*x > 100;
17   writeln("Jetzt nochmal!");
18 end ;

```

In den `if`-Befehlen steht die Bedingung (ein Ausdruck vom Typ `bool`) zwischen `if` und `then` und muss deshalb *nicht* in runde Klammern eingeschlossen werden.

Ein `loop`-Befehl muss mindestens *ein* `break`-Konstrukt enthalten, darf aber auch *mehrere* davon enthalten. Ein `break`-Konstrukt besteht aus den Schlüsselworten `break` und `when`, einem `bool`-Ausdruck und einem Semikolon `;`. Wenn der `bool`-Ausdruck den Wert `true` hat, wird ans Ende des `loop`-Befehls gesprungen (oder genauer: zum ersten Befehl nach dem `loop`-Befehl). Vor dem ersten und nach jedem `break`-Konstrukt darf eine beliebige Folge von Befehlen (einschließlich einer leeren Folge) stehen.

**Tipp:** Betrachten Sie das `break`-Konstrukt *nicht* als einen neuen Befehl (obwohl das nahe liegt), sondern als "ein spezielles Konstrukt, welches nur in einem `loop`-Befehl vorkommen darf".

Entwickeln Sie mit dieser Sichtweise (und Papier und Bleistift) eine Grammatik für `loop`-Befehle mit dem Startsymbol `KS_Befehl`. Benutzen Sie dabei zusätzlich zu den schon eingeführten Zwischensym-

holen `KS_BefehlsFolge` und `KS_Ausdruck` als neue Zwischensymbole nur `KS_Break` und `KS_BreakFolge`.

Definieren Sie dann zwei Gentle-Typen namens `AS_Break` und `AS_BreakFolge`, deren Werte dazu geeignet sind, aus `KS_Break` und `KS_BreakFolge` ableitbare Konstrukte intern darzustellen (so, wie Werte des Typs `AS_Befehl` dazu geeignet sind, aus `KS_Befehl` ableitbare Konstrukte intern darzustellen).

Erweitern Sie dann Ihren Alg14-Compiler so, dass er die neuen `if`- und `loop`-Befehle erkennen und übersetzen kann. Vergessen Sie dabei nicht, auch das Prädikat `prfBefehl` so zu erweitern, dass es auch in den neuen Befehlen und ihren Bestandteilen alle Typfehler erkennen kann und den ersten erkannten Fehler meldet.

## 8. Eine Typ-1-Grammatik für die Sprache DOPPELT

Entwickeln (mit Papier und Bleistift) Sie eine Typ-1-Grammatik (d.h. eine kontextsensitive Grammatik) für die Sprache DOPPELT. Jedes Wort dieser Sprache besteht aus zwei gleichen (nicht-leeren) Zeichenfolgen, die durch ein Trennzeichen  $:$  voneinander getrennt und zusammen in Klammern  $($  und  $)$  eingeschlossen sind. Die Zeichenfolgen sollen nur aus den Buchstaben  $a$  und  $b$  bestehen (der Einfachheit halber).

**Beispiele:** Zu der Sprache DOPPELT sollen unter anderem die folgenden sechs Worte gehören:

$(aab:aab)$

$(abaab:abaab)$

$(a:a)$

$(b:b)$

$(aaaaa:aaaaa)$

$(abababababab:abababababab)$

**Gegenbeispiele:** Die folgenden acht Worte sollen *nicht* zur Sprache DOPPELT gehören:

$(abab)$  weil das Trennzeichen  $:$  fehlt,

$(ab:aa)$  weil  $ab$  nicht gleich  $aa$  ist,

$(aab:aa)$  weil  $aab$  nicht gleich  $aa$  ist,

$(abc:abc)$  weil  $c$  nicht zulässig ist,

$((aab:aab))$  weil doppelte Klammern nicht zulässig sind,

$(aab:aab$  weil die schliessende Klammer fehlt,

$)aab:aab($  weil die beiden Klammern an falschen Stellen stehen,

$(aab: )aab$  weil die beiden Klammern an falschen Stellen stehen

Kommentieren Sie Ihre Grammatik, indem Sie für jede "Gruppe von zusammengehörigen Regeln" angeben, "was man mit diesen Regeln machen kann" oder "wozu diese Regeln da sind".

## 9. Eine Typ-0-Grammatik für Vereinbarungen und Anwendungen

Geben Sie eine Typ-0-Grammatik an, aus der man unter anderem die folgenden 6 Worte ableiten kann (die Zeilen-Nummern am Anfang gehören nicht zu den Worten):

1. begin dec-b; dec-c; middle app-c; app-c; app-b; app-c; app-b; app-b; end
2. begin dec-a; dec-c; dec-b; middle app-c; app-c; app-c; app-a; app-b; end
3. begin dec-b; dec-a; dec-c; middle app-a; app-b; app-c; app-c; app-b; app-a; end
4. begin dec-a; dec-b; dec-c; middle app-b; app-c; end
5. begin dec-a; middle end
6. begin middle end

Allgemein soll gelten: Jedes ableitbare Wort soll mit `begin` anfangen, mit `end` enden und irgendwo dazwischen ein `middle` haben.

Zwischen `begin` und `middle` sollen null bis drei Variablen-Deklarationen stehen. Die Deklaration einer Variablen namens `a` sieht so aus: `dec-a;`. Als Variablen-Namen sind nur `a`, `b` und `c` erlaubt. Ein Variablen-Namen darf höchstens in *einer* Deklaration erscheinen, aber die Reihenfolge der Deklarationen ist beliebig.

Zwischen `middle` und `end` sollen beliebig viele Variablen-Anwendungen stehen (z.B. null Variablen-Anwendungen oder eine oder zwei oder siebzehn oder ...). Die Anwendung einer Variablen namens `a` sieht so aus: `app-a;`.

Die ableitbaren Zeichenketten sollen folgende Kontextbedingung erfüllen: nach `middle` dürfen nur Anwendungen der Variablen stehen, die vor `middle` vereinbart wurden. Vereinbarte Variablen dürfen, müssen aber nicht angewendet werden (siehe oben das 4. und 5. Beispiel).

Hier ein paar Zeichenketten, die nicht ableitbar sein sollen:

```
begin dec-a; dec-a; middle end    -- a mehr als einmal vereinbart
begin dec-a; middle app-b; end   -- b angewendet, aber nicht vereinbart
begin dec-d; middle app-d; end   -- d ist kein erlaubter Name
middle ap-a end decc-a; begin    -- falsche Schlüsselworte
```

**Kommentieren** Sie Ihre Grammatik, indem Sie für jede "Gruppe von zusammengehörigen Regeln" angeben, "was man mit diesen Regeln machen kann" oder "wozu diese Regeln da sind". Diese Kommentare sind mindestens so wichtig wie die Regeln selbst!

Zusatzfrage (freiwillig): Würde Ihr Lösungsansatz auch dann noch „vernünftig funktionieren“, wenn anstelle der drei Variablennamen (`a`, `b` und `c`) zehn oder hundert Variablennamen erlaubt wären?