von Ulrich Grude

Inhaltsverzeichnis

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 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 einen Parser namens `ahnen_v1`, der die Worte der folgenden formalen Sprache `A1` (wie Ahnen 1) akzeptiert:

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

Teil 2: Erweitern Sie den Parser `ahnen_v1` zu einem Compiler namens `ahnen_v2`, der die Worte der formalen Sprache `A1` in natürliche Zahlen (1, 2, 3, ...) übersetzt. Diese Zahlen sollen angeben, wieviele 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 `ahnen_v1` zu einem Compiler namens `ahnen_v3`, 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:

```
grossmutter:       ur(mu)
urgrossmutter:     ur(ur(mu))
ururgrossmutter:   ur(ur(ur(mu)))
...
grossvater:        ur(va)
urgrossvater:      ur(ur(va))
ururgrossvater:    ur(ur(ur(va)))
...
```

Teil 4: Erweitern Sie den Parser `ahnen_v1` zu einem Compiler namens `ahnen_v4`, 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 `ahnen_v3` zu einem Compiler namens `ahnen_v5`, 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) folgt auf der nächsten Seite. Er ist eine Arte "Wiederholung und Zusammenfassung" der vorhergehenden Teile.

Teil 6: Schreiben Sie einen Compiler namens `ahnen_v6` mit folgender Quell- und Zielsprache:

Worte der Quellsprache:	*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 beschreiben alle Vorfahren von zwei Personen namens Maria und Johann.

Gehen Sie beim Lösen dieser Aufgabe wie folgt vor:

1. Entwickeln Sie (mit Papier und Bleistift, ohne Rechner) eine *Grammatik für die Quellsprache* und zeigen Sie sie 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).

2. Schreiben Sie in Gentle einen Parser 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 im Teil 1 dieser Aufgabe). Testen Sie den Parser!

3. Entwickeln Sie einen Gentle-Typ, dessen Werte sich als Zwischendarstellungen für die Quellprogramme eignen.

4. Ergänzen Sie den Parser dann um Attribute zu einem Compiler C1, der aus dem Quellprogramm, welches er einliest, eine entsprechende Zwischendarstellung erzeugt (ähnlich wie 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.

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` benützen, 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'
```

3. Grammatiken B: Einfache Sprachen

Teil 1: Entwickeln Sie (mit Papier und Bleistift, *ohne* Rechner) für jede der formalen Sprachen FS1 bis FS15 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) benützen, die durch folgende 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 erwähnten werde. (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`, `dddeeefff`, `Ddeeefff`, 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,Qark` 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`, `ansNegElem`, `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 ------------------------------------------------------------------------
```

```
'action' posElem(L: LISTE -> LP: LISTE)
   -- Berechnet eine Liste LP aller positiven Zahlen in L
-----------------------------------------------------------------------
'action' negElem(L: LISTE -> LN: LISTE)
   -- Berechnet eine Liste LN aller negativen Zahlen in L
-----------------------------------------------------------------------
'action' posNegElem(L: LISTE -> LP: LISTE, LN: LISTE)
   -- Berechnet eine Liste LP aller positiven und eine Liste LN aller
   -- negativen Zahlen in L
-----------------------------------------------------------------------
'action' geradeElem(L: LISTE -> LG: LISTE)
   -- Berechnet eine Liste LG aller geraden Zahlen in L
-----------------------------------------------------------------------
'action' anzDoppelte(L: LISTE -> Anzahl: INT)
   -- Wie oft stehen in der Liste L zwei gleiche Zahlen nebeneinander?
   -- Dieses Praedikat berechnet die Antwort.
   -- Achtung: Die "Zweiergruppen" sollen sich nicht ueberlappen. D.h.
   -- Wenn drei gleiche Zahlen nebeneinander stehen, dann zaehlen sie
   -- nur als EINE Zweiergruppe (und eine einzelne Zahl). Erst vier
   -- gleiche Zahlen nebeneinander sollen als ZWEI Zweiergruppen
   -- gezaehlt werden.
-----------------------------------------------------------------------
'action' anzDreimalige(L: LISTE -> Anzahl: INT)
   -- Wie oft stehen in der Liste L drei gleiche Zahlen nebeneinander?
   -- Dieses Praedikat berechnet die Antwort.
   -- Achtung: Die "Dreiergruppen" sollen sich nicht ueberlappen. D.h.
   -- Wenn vier gleiche Zahlen nebeneinander stehen, dann zaehlen sie
   -- nur als EINE Dreiergruppe (und eine einzelne Zahl). Erst sechs
   -- gleiche Zahlen nebeneinander sollen als ZWEI Dreiergruppen
   -- gezaehlt werden.
-----------------------------------------------------------------------
'condition' sindGleich(L1: LISTE, L2: LISTE)
   -- Gelingt, wenn die Listen L1 und L2 ("Element fuer Element")
   -- gleich sind
-----------------------------------------------------------------------
'condition' enthaeltElem(L: LISTE, Elem: INT)
   -- Gelingt genau dann, wenn die Liste L die Zahl Elem enthaelt.
-----------------------------------------------------------------------
'condition' enthaeltListe(L1: LISTE, L2: LISTE)
   -- Gelingt genau dann, wenn die Liste L1 die Liste L2 enthaelt
   -- (d.h. wenn jedes Element von L2 auch in L1 vorkommt)
-----------------------------------------------------------------------
'condition' habenGleicheElemente(L1: LISTE, L2: LISTE)
   -- Gelingt genau dann, wenn
   -- 1. jedes Element von L1 auch in L2 vorkommt und
   -- 2. jedes Element von L2 auch in L1 vorkommt.
-----------------------------------------------------------------------
'condition' enthaeltDoppelgaenger(L: LISTE)
   -- Gelingt genau dann, wenn mindenstens eine Zahl mehr als einmal
   -- in L vorkommt (egal, an welcher Stelle).
-----------------------------------------------------------------------
'condition' istAufSortiert(L: LISTE)
   -- Gelingt genau dann, wenn die Liste L aufsteigend sortiert ist
-----------------------------------------------------------------------
```

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 Grundspeziallfälle von T9 an.
5.05. Geben Sie alle Grundspeziallfälle von T4 an.
5.06. Geben Sie alle Grundspeziallfälle von T1 an.
5.07. Wieviele Grundspezialfälle hat der Term T6?
5.08. Wieviele Grundspezialfälle hat der Term T11?
5.09. Wieviele Grundspezialfälle hat der Term T12?
5.10. Wieviele Grundspezialfälle hat der Term T5?
5.11. Wieviele Grundspezialfälle hat der Term T7?
5.12. Wieviele 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 `alg00str.g.g`. Erstellen Sie eine Kopie namens `alg00str.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 `alg00ide.g.g`. Ergänzen Sie diese Quelldatei, so dass sie richtig funktioniert (und benennen Sie sie um in `alg00ide.g`). Diese Teilaufgabe stimmt weitgehend mit der vorigen überein und sollte deshalb mit sehr wenig Arbeit lösbar sein.

7. Ein Compiler für die Sprache Alg, Ausbaustufe Alg01

Sie sollen einen Compiler schreiben, der Quellprogramme einer Sprache Alg in Zielprogramme der Sprache Jasmin (Java-Assemblercode) 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 Alg01 implementieren und dann immer größere Teilmengen (Alg02, Alg03, ...). Es folgt hier eine kontextfreie Grammatik der Teilsprache Alg01 von Alg (schon in Gentle notiert):

Alg01: Nur Vereinbarungen von `int`-Variablen, `read`- und `write`-Anweisung:

```
 1 'nonterm' Alg01Prog
 2    'rule' Alg01Prog: VereinbarungsFolge AnweisungsFolge
 3
 4 'nonterm' VereinbarungsFolge
 5    'rule' VereinbarungsFolge: .       -- Kann leer sein
 6    'rule' VereinbarungsFolge: Vereinbarung VereinbarungsFolge
 7
 8 'nonterm' Vereinbarung
 9    'rule' Vereinbarung: "int" Bezeich ";"
10
11 'nonterm' AnweisungsFolge
12    'rule' AnweisungsFolge: Anweisung  -- Mind. eine Anweisung
13    'rule' AnweisungsFolge: Anweisung AnweisungsFolge
14
15 'nonterm' Anweisung
16    'rule' Anweisung: EinfacheAnweisung         ";"
17 -- 'rule' Anweisung: ZusammengesetzteAnweisung ";" -- Fuer spaetere Er-
18                                                    -- weiterungen
19 'nonterm' EinfacheAnweisung
20    'rule' EinfacheAnweisung: "read"  "(" Bezeich       ")"
21    'rule' EinfacheAnweisung: "write" "(" Bezeich       ")"
22    'rule' EinfacheAnweisung: "write" "(" StringLiteral ")"
```

Tipp 1: Legen Sie für jede Ausbaustufe Ihres Compilers *ein eigenes Verzeichnis* (`alg01`, `alg02`, `alg03`, ...) an und kopieren Sie alle benötigten Dateien da hinein. Im Falle einer Katastrophe (Quelldatei gelöscht oder sonstwie unbrauchbar gemacht) können Sie auf die vorige Ausbaustufe zurückgreifen.

Tipp 2: Bei den Gentle-Beispielen finden Sie im Verzeichnis `alg01` einige für die erste Ausbaustufe interessante Dateien, insbesondere eine unvollständige Quelldatei namens `Alg01.g.g` (mit zweimal `.g`), von der Sie eine Kopie namens `Alog01.g` (einmal `.g`) erstellen und vervollständigen sollen.

8. Ein Compiler für die Sprache Alg, Ausbaustufe Alg02

In dieser Ausbaustufe fügen wir *Zuweisungen* und *Ausdrücke* hinzu. Es folgt hier eine Ergänzung der kontextfreien Grammatik:

```
23    'rule' EinfacheAnweisung:
24       Bezeich ":=" ArithmetischerAusdruck
25
26 'nonterm' ArithmetischerAusdruck
27    'rule' ArithmetischerAusdruck:
28       ArithmetischerAusdruck "+" ArithmetischerAusdruck1
29    'rule' ArithmetischerAusdruck:
30       ArithmetischerAusdruck "-" ArithmetischerAusdruck1
31    'rule' ArithmetischerAusdruck:
32       ArithmetischerAusdruck1
33
34 'nonterm' ArithmetischerAusdruck1
35    'rule' ArithmetischerAusdruck1:
36       ArithmetischerAusdruck1 "*" ArithmetischerAusdruck2
37    'rule' ArithmetischerAusdruck1:
38       ArithmetischerAusdruck1 "/" ArithmetischerAusdruck2
39    'rule' ArithmetischerAusdruck1:
40       ArithmetischerAusdruck1 "%" ArithmetischerAusdruck2
41    'rule' ArithmetischerAusdruck1:
42       ArithmetischerAusdruck2
43
44 'nonterm' ArithmetischerAusdruck2
45    'rule' ArithmetischerAusdruck2:
46       GanzLiteral
47    'rule' ArithmetischerAusdruck2:
48       Bezeich
49       MeldeWennUndefiniert
50    'rule' ArithmetischerAusdruck2:
51       "(" ArithmetischerAusdruck ")"
```

9. Ein Compiler für die Sprache Alg, Ausbaustufe Alg03

In dieser Ausbaustufe fügen wir `if`-Anweisungen und boolesche Ausdrücke hinzu. Es folgt hier eine Ergänzung der kontextfreien Grammatik:

```
52    'rule' Anweisung: ZusammengesetzteAnweisung ";"
53
54 'nonterm' ZusammengesetzteAnweisung
55    'rule' ZusammengesetzteAnweisung:
56       "if"   BoolescherAusdruck
57       "then" AnweisungsFolge
58       "else" AnweisungsFolge
59       "end" "if"
60
61 'nonterm' BoolescherAusdruck
62    'rule' BoolescherAusdruck:
63       BoolescherAusdruck "||" BoolescherAusdruck1
64    'rule' BoolescherAusdruck:
65       BoolescherAusdruck1
66
67 'nonterm' BoolescherAusdruck1 ... etc. etc.
68 'nonterm' BoolescherAusdruck2 ... etc. etc.
```

10. Ein Compiler für die Sprache Alg, Ausbaustufe Alg04

In dieser Ausbaustufe fügen wir `while-do-` und `do-until-`Schleifen hinzu. Es folgt hier eine Ergänzung der kontextfreien Grammatik:

```
69    'rule' ZusammengesetzteAnweisung:
70       "while" BoolescherAusdruck
71       "do"    AnweisungsFolge
72       "end" "while"
73    'rule' ZusammengesetzteAnweisung:
74       "do"    AnweisungsFolge
75       "until" BoolescherAusdruck
76       "end" "until"
```

11. Ein Compiler für die Sprache Alg, Ausbaustufe Alg05

Erweitern Sie die Sprache Alg um folgende Möglichkeiten:

1. Bein Vereinbaren einer `int`-Variablen soll der Alg-Programmierer einen arithmetischen Ausdruck zum Initialisieren der Variable angeben können.

2. Als Parameter des `write`-Befehls soll der Alg-Programmierer einen arithmetischen Ausdruck oder einen String-Ausdruck angeben können, z. B. so:

```
77  write(2*otto + 5);
78  write("Erg1: " & zahl1 & ", Erg2: " & zahl2 & "\n") ).
```

Wenn der Konkatenationsoperator `&` zwischen einem `String`- und einem `int`-Ausdruck steht, wird der Wert des `int`-Ausdrucks automatisch in einen `String` umgewandelt und dann mit dem anderen String konkateniert (ähnlich wie `+` in Java).

12. 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".

13. 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 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 genauso wichtig wie die Regeln selbst!

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