Inhaltsverzeichnis

Regeln für das Lösen dieser Aufgaben

1. Bilden Sie eine **Arbeitsgruppe**, 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 3 Mitgliedern können in Ausnahmefällen vom Betreuer Ihres Übungstermins genehmigt werden. Gruppen mit weniger als 2 Mitgliedern ("Einzelkämpfer" und leere Gruppen :-) sind **nicht zuläßig**.

2. Besonders wichtig (und umfangreich) ist die Aufgabe **7. Ein Compiler für die Sprache alg3**. Aber die anderen Aufgaben sind auch wichtig, als Vorübugen und Ergänzungen zur Aufgabe 7.

1. Parser und Compiler für Ancestor Languages

ancestor01: Schreiben Sie eine (kontextfreie, Typ-2-) **Grammatik** für die *ancestor language* A1:

```
A1 = {mother, father, grandmother, grandfather, greatgrandmother, greatgrandfather,
greatgreatgrandmother, ..., greatgreatgreatgreatgrandfather, ...}
```

Übersetzen Sie Ihre Grammatik dann in entsprechende `phrase`-Prädikate eines Gentle-Programms. 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 der `main`-Funktion eines C-Programms.

ancestor02: Erweitern Sie den Parser aus **ancester01** zu einem Compiler, der Worte der formalen Sprache `A1` in natürliche Zahlen (1, 2, 3, ...) übersetzt. Diese Zahlen sollen angeben, wie viele Generationen die betreffenden ancestors von uns entfernt sind (`mother` und `father`: 1 Generation, `grandmother` und `grandfather`: 2 Generationen, `greatgrandmother` und `greatgrandfather`: 3 Generationen etc.).

ancestor03: Erweitern Sie den Parser aus **ancestor01** zu einem Compiler, der Worte der formalen Sprache `A1` in eine *Zwischendarstellung* (oder: in *abstracte Syntax*) übersetzt, und zwar in Werte des folgenden Gentle-Typs:

```
1 type AS_ancestor    // Abstract syntax for
2    mo()             // mother
3    fa()             // father
4    g(AS_ancestor)   // all the other ancestors
```

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

Wort aus A1	**Zwischendarstellung**
mother	mo()
father	fa()
grandmother	g(mo())
grandfather	g(fa())
greatgrandmother	g(g(mo()))
greatgrandfather	g(g(fa()))
...	
greatgreatgreatgrandmother	g(g(g(g(mo()))))
greatgreatgreatgrandfather	g(g(g(g(fa()))))
...	

ancestor04: Erweitern Sie den Parser aus **ancestor01** zu einem Compiler, der ein Wort der ancestor language `A1` in eine entsprechende Listen von Strings (aus der Menge {`mutter`, `vater`, `gross`, `ur`}) übersetzt und diese Liste dann als ein Wort der Ahnensprache A2 ausgibt, mit A2 = {`mutter`, `vater`, `grossmutter`, `grossvater`, `urgrossmutter`, `urgrossvater`, `ururgrossmutter`, ...}.

Diese Aufgabe kann (sollte?) weggelassen werden!

Beispiel: Das Wort `greatgreatgrandfather` aus A1 soll in die Liste (in Gentle-Notation) `string["ur", "ur", "gross", "vater"]` übersetzt werden und diese Liste soll als das Wort `ururgrossvater` aus A2 zum Bildschirm ausgegeben werden.

ancestor05: Erweitern Sie den Compiler aus **ancestor03** zu einem Compiler, der Worte der ancestor language `A1` in eine Zwischendarstellung und die Zwischendarstellung dann in ein Wort der Ahnensprache A2 übersetzt (siehe vorige Aufgabe **ancestor04**) und ausgibt.

Dieser Compiler soll also das Gleiche leisten wie der in **ancestor04**, aber auf eine etwas andere Weise.

ancestor06: Schreiben Sie einen Compiler der Worte der **ancestor language A3** in entsprechende Worte der **Ahnensprache A4** übersetzt.

Ancestor language A3 (source language)	**Ahnensprache A4 (Zielsprache)**
The mother of Mary	Die mutter von Maria
The father of Mary	Der vater von Maria
The mother of John	Die mutter von Johann
The father of John	Der vater von Johann
The mother of the mother of Mary	Eine grossmutter von Maria
The mother of the father of Mary	Eine grossmutter von Maria
The father of the mother of Mary	Ein grossvater von Maria
The father of the father of Mary	Ein grossvater von Maria
The mother of the mother of John	Eine grossmutter von Johann
The mother of the father of John	Eine grossmutter von Johann
The father of the mother of John	Ein grossvater von Johann
The father of the father of John	Ein grossvater von Johann
...	...
The mother of the mother of the father of the mother of John	Eine ururgrossmutter von Johann
...	...

Diese Tabelle soll verdeutlichen, dass in vielen Fällen unterschiedliche Worte aus A3 in das gleiche Wort aus A4 zu übersetzen sind.

Gehen Sie beim Lösen dieser Aufgabe wie folgt vor:

Schritt 1: Entwickeln Sie (mit Papier und Bleistift, ohne Rechner) eine *Grammatik für die Quellsprache A3* 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).

Tipp zu Schritt 1: Beim Schreiben der Grammatik für `ancestor01` haben wir die zu beschreibende *Sprache* in 3 Teile eingeteilt (`ancestor1`: 1. Generation, `ancestor2`: 2. Generation, `ancestor3`: alle anderen Generationen). Beim Schreiben der Grammatik für `ancestor06` ist folgende Strukturierung empfehlenswert: Man denkt sich jedes *Wort* der zu beschreibenden Sprache (nicht die Sprache) in drei Teile eingeteilt.

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 in **ancestor01**). Testen Sie Ihren Parser!

Schritt 3: Machen Sie sich klar, dass bei der Übersetzung von A3 nach A4 "Information weggeworfen wird". Z.B. werden beide Großmütter von Mary, für die es in A3 unterschiedliche Worte gibt, auf das selbe Wort in A4 ("Eine Großmutter von Maria") abgebildet. Ähnliches gilt auch für die 4 Urgroßväter von John und in vielen weiteren Fällen.

Entwickeln Sie einen Gentle-Typ namens `AS_ancestor06`. Jeder Wert dieses Typs soll so viel Informationen enthalten, dass man daraus genau ein bestimmtes Wort der Zielsprache A4 erzeugen kann. Welche Informationen braucht man dazu? Um das herauszufinden können Sie zu zweit (A und B) folgendes Spiel spielen:

A wählt heimlich ein Wort aus A3 (z.B. "The father of the mother of John"). Dann darf B 3 Fragen nach Eigenschaften des ancestors aus A3 stellen z.B. "Ist es ein Mann oder eine Frau?" und so ähnlich. B muss die Fragen wahrheitsgemäß beantworten. Mit Hilfe der Antworten muss A dann die richtige Übersetzung für das von A gewählte A3-Wort herausfinden (im Beispiel ist das das A4-Wort: "Ein grossvater von Johann"). Auf diese Weise können Sie herausfinden, welche Informationen in jedem Wert des Typs `AS_ancestor06` enthalten sein muss.

Tipp zu Schritt 3: Bevor Sie den Typ `AS_ancestor06` vereinbaren, sollten Sie zwei Hilfstypen vereinbaren, mit deren Werten man 2 der 3 Fragen (die B dem A gestellt hat) beantworten kann. Für die Frage: "Ist es ein Mann oder eine Frau?" könnte das z.B. ein Aufzählungstyp mit zwei Werten sein.

Anmerkung zu Schritt 3: Der Gentle-Typ `AS_ancestor03` in **ancestor03** hat z.B. den "Generationsabstand 17" durch einen ziemlich langen Term der Form `g(g(g(...)))` dargestellt. Könnte man den Abstand 17 nicht etwas einfacher und kompakter darstellen?

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 **ancestor03**). 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 durch Ausgabeprädikate zu einem **Compiler C2**. Die Ausgabeprädikate sollen eine Zwischendarstellung in die Zielsprache A4 übersetzen und ausgeben (ähnlich wie in **ancestor05**). Testen Sie den Compiler C2.

ancestor07: Diese Teilaufgabe dürfen Sie bearbeiten, müssen es aber nicht. Empfehlenswert ist eine Bearbeitung, wenn Ihnen die vorige Teilaufgabe (**ancestor06**) noch ziemlich schwer vorkam und Ihnen die Einzelschritte noch nicht geläufig und selbstverständlich sind.

Schreiben Sie einen Compiler der Worte der **Ahnensprache A5** in entsprechende Worte der **ancestor language A6** übersetzt.

Ahnensprache A5 (Quellsprache)	**Ancestor language A6 (Target language)**
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 der Mutter von Maria	A grandmother of Mary
Die Mutter des Vaters von Maria	A grandmother of Mary
Der Vater der Mutter von Maria	A grandfather of Mary
Der Vater des Vaters von Maria	A grandfather of Mary
Die Mutter der Mutter von Johann	A grandmother of John
Die Mutter des Vaters von Johann	A grandmother of John
Der Vater der Mutter von Johann	A grandfather of John
Der Vater des Vaters von Johann	A grandfather of John
...	...
Die Mutter der Mutter des Vaters der Mutter von Johann	A greatgreatgrandmother of John
...	...

Auch dieser Compiler muss in vielen Fällen unterschiedliche Worte aus A5 in das gleiche Wort aus A6 übersetzen.

Zum Abschluss folgt hier eine Kurzfassung der empfohlenen Entwicklungsschritte:

Schritt 1: Eine *Grammatik* für die Quellsprache schreiben

Schritt 2: Einen *Parser* für die Quellsprache programmieren (noch ohne Attribute)

Schritt 3: Eine *Zwischendarstellung* (abstrakte Syntax) für die Worte der Quellsprache entwickeln (oder schon vorhandene Gentle-*Typen* dafür wiederverwenden).

Schritt 4: Den Parser so *mit Attributen versehen*, dass er Quell-Worte in die Zwischendarstellung (oder: konkrete Syntax in abstrakte Syntax) übersetzt.

Schritt 5: *Ausgabeprädikate* hinzufügen, die eine Zwischendarstellung in die Zielsprache übersetzen und ausgeben.

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"
```

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 **FS01** 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) benützen, 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 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`, `Dddeeefff`, `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#geht s#04#` **nein**: `123` oder `#Hallo` oder `#Hallo#` oder `123##` oder `abc#123`). Startsymbol: **NZ_Getr_0104**.

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

4. In Gentle proc- und condition-Prädikate programmieren

Ergänzen Sie die folgenden Spezifikationen der Prädikate `length`, `sum`, `nrOfPosElems`, ... etc. um geeignete Regeln. Ihre Regeln können Sie sofort testen, indem Sie sie an der entsprechenden Stelle in die Quelldatei `projects\pred02\prob01.g` eintippen, die Quelldatei compilieren und ausführen lassen.

Hinweis: Im Englischen ist **iff** eine Abkürzung für **if and only if**
(im Deutschen etwa: **genau dann wenn** oder kürzer **wennn**).

```
 1 // ------------------------------------------------------------------- 1
 2 proc length(list:int[] -> length:int)
 3    // Computes the length of list.
 4 // ------------------------------------------------------------------- 2
 5 proc sum(list:int[] -> sum:int)
 6    // Computes the sum of the elements of list.
 7 // ------------------------------------------------------------------- 3
 8 proc nrOfPosElems(list:int[] -> nr:int)
 9    // Computes the number of positive elemenst in list.
10    // Remember: 0 is not positive.
11 // ------------------------------------------------------------------- 4
12 proc nrOfNegElems(list:int[] -> nr:int)
13    // Computes the number of negative elements in list.
14    // Remember: 0 is not negative.
15 // ------------------------------------------------------------------- 5
16 proc sumOfPosElems(list:int[] -> sum:int)
17    // Computes the sum of all positive numbers in list.
18 // ------------------------------------------------------------------- 6
19 proc sumOfNegElems(list:int[] -> sum:int)
20    // Coumputes the sum of all negative numbers in list.
21 // ------------------------------------------------------------------- 7
22 proc maxPosElem(list:int[] -> max:int)
23    // Computes the largest postivie element (max) in list
24    // (and 0 if list does not contain positive numbers).
25 // ------------------------------------------------------------------- 8
26 proc minNegElem(list:int[] -> min:int)
27    // Computes the smallest negative element (min) in list
28    // (and 0 if list does not contain negative numbers).
29 // ------------------------------------------------------------------- 9
30 condition isEven(number:int)
31    // Succeeds iff number is even
32 // ------------------------------------------------------------------ 10
33 condition isOdd(number:int)
34    // Succeeds iff number is odd
35 // ------------------------------------------------------------------ 11
36 proc nrOfEvenElems(list:int[] -> nrEven:int)
37    // Computes the number of even elements (nrEven) in list.
38 // ------------------------------------------------------------------ 12
39 proc nrOfOddElems(list:int[] -> nrOdd:int)
40    // Computes the number of odd elements (nrOdd) in list.
41 // ------------------------------------------------------------------ 13
42 proc nrsOfPosNegElems(list:int[] -> nrPos:int, nrNeg:int)
43    // Computes the number of positive elements (nrPos)
44    // and      the number of negative elements (nrNeg) in list.
45 // ------------------------------------------------------------------ 14
46 proc nrsOfNegEvenElems(list:int[] -> nrPos:int, nrEven:int)
47    // Computes the number of positive elements (nrPos)
48    // and      the number of even     elements (nrEven) in list
49 // ------------------------------------------------------------------ 15
50 proc listOfPosElems(list:int[] -> listPos:int[])
51    // The listPos contains all positive elements of list
52    // Remember: 0 is not positive.
53 // ------------------------------------------------------------------ 16
54 proc listOfNegElems(list:int[] -> listNeg:int[])
55    // The listNeg contains all negative elements of list
56    // Remenber: 0 is not negative.
```

```
57 // ------------------------------------------------------------------- 17
58 proc listsOfPosNegElems(list:int[] -> listPos:int[], listNeg:int[])
59    // The listPos contains all positive and listNeg all negative
60    // elements of list.
61 // ------------------------------------------------------------------- 18
62 proc listOfEvenElems(list:int[] -> listEven:int[])
63    // The listEven contains all even elements of list
64 // ------------------------------------------------------------------- 19
65 proc nrOfEqualNeighbors(list:int[] -> pairs:int)
66    // Computes the number of pairs of equal neighbors in list.
67    // Such pairs must not overlap:
68    // int[5, 5, 5]    contains 1 pair  of equal neighbors (and a single 5)
69    // int[5, 5, 5, 5] contains 2 pairs of equal neighbors
70    // int[5, 4, 5, 4] contains 0 pairs of equal neighbors
71 // ------------------------------------------------------------------- 20
72 condition containsElem(list:int[], elem:int)
73    // Succeeds iff the list contains the elem
74 // ------------------------------------------------------------------- 21
75 condition containsList(list1:int[], list2:int[])
76    // Succeeds iff list1 contains list2, i.e.
77    // iff each element of list2 is contained (at least once) in list1.
78    // Note: In the presence of double elements, a shorter list may
79    // contain a longer list (e.g. int[1, 2] contains int[2, 1, 2])
80 // ------------------------------------------------------------------- 22
81 condition containsEqualElems(list:int[])
82    // Succeeds iff list contains at least two elements which are equal.
83 // ------------------------------------------------------------------- 23
84 condition containEqualElems(list1:int[], list2:int[])
85    // Succeeds iff
86    // each element in list1 is also in list2 and
87    // each element in list2 is also in list1.
88    // In the presence of double elements this predicate may succeed even
89    // if list1 and list2 are of different lengths
90    // (e.g. containEqualElems(int[1, 2, 2], int[2, 1]) succeeds).
91 // ------------------------------------------------------------------- 24
92 condition isSortedAscending(list:int[])
93    // Succeeds iff list is sorted in ascending order
94 // ------------------------------------------------------------------- 25
95 condition isSortedDescending(list:int[])
96    // Succeeds iff list is sorted in descending order
```

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 Alg2

Im Verzeichnis `projects\alg20` finden Sie einen unvollständigen Compiler (Quelldatei: `spec.g`). Den kann man compilieren und auf Quelldateien anwenden, er gibt aber immer die Meldung

```
Nr of errors found by check10: 0
```

aus, auch wenn die Quelldatei undeklarierte oder doppelt definierte Identifier enthält.
Ersetzen Sie die (speziell gekennzeichneten) unsinnigen Regeln in der Datei `spec.g` so durch "richtige Regeln", dass der Compiler alle Verstöße gegen Kontextbedingungen entdeckt und meldet.

7. Ein Compiler für die Sprache alg3

Dies ist die mit Abstand umfangreichste und wichtigste Aufgabe, die in diesem Fach gestellt wird:

Sie sollen einen Compiler schreiben, der Quellprogramme einer Sprache **alg3** in Zielprogramme der Sprache **Jasmin** (Java-Assembler) übersetzt. Dabei soll "alg" nicht an "Alkohol" erinnern, sondern an die *Algol-Sprachfamilie*, zu der Algol60, Algol-W, Algol68, Pascal, Modula und Ada gehören.

Den a**lg3**-Compiler sollen Sie in *mehreren Stufen* entwickeln, indem Sie zuerst eine kleine Teilsprache **alg30** implementieren und dann immer größere Teilsprachen (**alg31**, **alg32**, **alg33**). Für jede dieser vier Ausbaustufen ist im Ordner `projects` ein gleichnamiger Ordner vorgegeben (`projects\alg30`, `projects\alg31`, `projects\alg32`, `projects\alg33`).

7.1 alg30

Im Ordner `alg30` finden Sie einen vollständigen und (hoffentlich) funktionierenden Compiler (`spec.g`), der allerdings nur sehr simple Programme übersetzen kann, z.B. das folgende:

```
1  writeln("Hallo Welt!");
2  writeln("-----------");
```

Diese Programme dürfen belibig viele (einen oder mehr) `writeln`-Befehle mit einem `string`-Literal als Parameter enthalten, sonst (noch) nichts.

Machen Sie sich mit diesem vorgegebenen Compiler vertraut. Schauen Sie sich die Datei `spec.g` genauer an und versuchen Sie, die einzelnen Abschnitte zu erkennen:

Wo werden die Typen für die abstrakte Syntax vereinbart?
Wo wird die konkrete Syntax festgelegt?
Wo werden Kontextbedingungen überprüft?
Wo erfolgt die Übersetzung der abstrakten Syntax in ein Zielprogramm und die Ausgabe?

Schreiben Sie ein oder zwei Quellprogramme, übersetzen Sie sie und lassen Sie sie ausführen. Schreiben Sie sich **Skripte**, um diese Arbeitsschritte möglichst einfach durchführen zu können.

Erweitern Sie den vorgegebenen Compiler dann so, dass er auch kompliziertere Programme übersetzen kann, z.B. das folgende:

```
 1  // Variablendeklarationen ohne explizite Initialisierung:
 2  int    anna;
 3  bool   bert;
 4  string carl;
 5
 6  // Variablendeklarationen mit expliziter Initialisierung:
 7  int    dora := 17;
 8  bool   emil := true;
 9  string fany := "Hallo!";
10
11  // Zuweisungen
12  anna := 25;
13  anna := dora;
14  bert := false;
15  bert := emil;
16  carl := "Pickelhering";
17  carl := fany;
18
19  // write und writeln
20  write(1234);   write(true);   write("AB\n");
21  write(anna);   write(bert);   write(carl);
22  writeln(1234); writeln(true); writeln("\nAB\n\n");
23  writeln(anna); writeln(bert); writeln(carl);
24
25  // read (liest einen Wert von der Standardeingabe in eine Variable)
26  read(anna); read(bert); read(carl);
```

Allgemeine Regeln der Sprache alg:

1. Ein Programm besteht aus einer nicht-leeren Folge von Befehlen (engl. commands).
Jeder Befehl wird mit einem Semikolon `;` abgeschlossen.

2. Es gibt genau drei Typen: `int`, `bool` und `string`.

3. Ein Befehl ist eine (Variablen-) Vereinbarung oder eine Anweisung.

4. Variablen, die ohne eine explizite Initialisierung vereinbart wurden, werden (implizit, automatisch) mit `0`, `false` bzw. einem leeren String `""` initialisiert.

5. In der Teilsprache **alg30** ist als *Ausdruck* (rechts von einem Zuweisungszeichen `:=` oder als Parameter von `write` oder `writeln`) nur ein *Literal* oder ein *Variablen-Bezeichner* erlaubt (in **alg31** werden weitere Ausdrücke hinzukommen).

Für einen Abschlusstest Ihres **alg30**-Compilers sind (im Ordner `projects\alg30`) die Skripte `TST1.CMD` und `TST2.CMD` vorgegeben. `TST1.CMD` prüft, ob der Compiler "alle richtigen **alg30**-Programme" akzeptiert und richtig übersetzt. `TST2.CMD` prüft, ob der Compiler "alle falschen **alg30**-Programme" ablehnt.

Wichtige Empfehlungen zur Vorgehensweise

Erweitern Sie den vorgegebenen Compiler in **kleinen Schritten**. Beginnen Sie jeden Schritt damit, ein möglichst kleines **alg30-Quellprogramm** zu schreiben, in dem ein neuer Befehl vorkommt, den der Compiler bisher noch nicht erkennen und übersetzen kann. Das Quellprogramm sollte möglichst nur *einen* `alg30`-Befehl enthalten, höchstens aber 3 solche Befehle. Beendet wird ein Entwicklungsschritt dadurch, dass Sie das `alg30`-Quellprogramm übersetzen und ausführen lassen und sich davon überzeugen, dass es die richtige Ausgabe produziert.

Die folgenden Entwicklungsschritte (und ihre *Reihenfolge*) sind besonders empfehlenswert:

Schritt 1: `write`-Befehle, die den Wert eines Ausdrucks ausgeben. Als Ausdrücke sind erstmal nur `string`-Literale erlaubt (die der Compiler "schon von Anfang an kannte").

Schritt 2: Zusätzlich zu `string`-Literalen sollen (in `write`- und `writeln`-Befehlen) auch `int`-Literale erlaubt sein.

Schritt 3: Zusätzlich zu `string`- und `int`-Literalen sollen auch `bool`-Literale (`true`, `false`) erlaubt sein.
Problem: Die JVM kennt keinen booleschen Typ.
Lösung: Es ist üblich, `bool`-Werte durch `int`-Werte darzustellen (`0` für `false` und `1` für `true`).

Anmerkung: Auch wenn Sie `bool`-Werte intern durch `int`-Werte darstellen, sollten Sie `int` und `bool` als *zwei völlig verschiedene Typen* und getrennt voneinander behandeln.

Schritt 4: Erlauben Sie Vereinbarungen von `int`-Variablen *ohne* explizite Initialisierung, z.B.
`int ilse;`
Solche Variablen sollen ganz am Anfang einer Programmausführung (vor der Ausführung aller anderen Befehlen des `alg`-Programms) mit `0` initialisiert werden. Dafür sollten Sie den Typ `AS_Cmd` um die abstrakte Syntax von *Zuweisungs-Befehlen* erweitern, etwa so:

```
type AS_Cmd
   ...
   assign(QB:string, AS_Exp)
   ...
```

Schritt 5: Erlauben Sie Vereinbarung von `bool`-Variablen *ohne* explizite Initialisierung, z.B.
`bool bert;`
Solche Variablen sollen mit `false` (alias `0`) initialisiert werden.

Schritt 6: Erlauben Sie die Vereinbarung von `string`-Variablen *ohne* explizite Initialisierung, z.B.
`string sonja;`

Solche Variablen sollen mit einem leeren String `""` initialisiert werden.

Schritt 7: Erlauben Sie Vereinbarungen von `int`-Variablen *mit* expliziter Initialisierung, z.B.

```
int ingo := 17;
int inge := ingo;
```

Benutzen Sie als abstrakte Syntax von Variablen-Vereinbarung (*ohne* und *mit* expliziter Initialisierung) folgende Variante des Typs `AS_Cmd`:

```
type AS_Cmd
   ...
   vardec(QB:string, AS_Type, AS_Exp?)
   ...
```

Der Typ `AS_Exp?` ist ein **Options-Typ** (in der Gentle-online-Doku siehe **Values / Option_Types**). Komponenten eines solchen Typs "können auch weggelassen werden". Am besten schreibt man ein kleines Gentle-Programm, in dem man solche Options-Typen "ausprobiert" (es gibt da einiges zu lernen).

Alle in einem `alg30`-Programm vereinbarten Variablen sollen am Anfang einer Programmausführung mit dem Standardwert ihres Typs (`0`, `false`, `""`) initialisiert werden. Falls der `alg`-Programmierer in einer Variablen-Vereinbarung eine explizite Initialisierung angegeben hat, dann soll die erst dann ausgeführt werden, wenn im `alg`-Quellprogramm "alle davor stehenden Befehle" ausgeführt wurden.

Beispiel für ein `alg30`-Programm mit Kommentaren:

```
writeln(anna);  // hier muss 0 ausgegeben werden
...
int anna := 17;
...
writeln(anna);  // hier muss 17 ausgegeben werden
```

Schritt 8: Erlauben Sie auch Vereinbarungen von `string`- und `bool`-Variablen *mit* expliziter Initialisierung, z.B.

```
string siggy := "Hallo!";
string susi  := siggy;
bool   berta := true;
bool   bernd := berta;
```

Schritt 9: Erlauben Sie Zuweisungs-Befehle, z.B.

```
inge :=  42;
ingo := inge;
```

Die abstrakte Syntax für solche Zuweisungen haben Sie ja bereits im **Schritt 4** festgelegt.

7.2 alg31

In **alg30** sind als Ausdrücke nur *Literale* und *Variablen-Bezeichner* erlaubt. In der Ausbaustufe **alg31** soll Ihr Compiler auch Ausdrücke akzeptieren und übersetzen, die folgende Operatoren enthalten:

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

Tabelle 1: Die Operatoren der Sprache Alg3

Gehen Sie in den folgenden Schritten vor, um Ihren Compiler entsprechend zu erweitern:

Schritt 1: Definieren Sie einen Aufzählungs-Typ namens `AS_Op1`, der für jeden *einstelligen Operator* (unären Operator) einen Namen festlegt (z.B. `not()` und `minus()`). Definieren Sie ausserdem einen Aufzählungstyp namens `AS_Op2`, der für jeden *zweistelligen Operator* (binären Operator) einen Namen festlegt (z.B. `conc()`, `lt()`, `le()`, ..., `or()`, ..., `add()`, `sub()`, ...). Erweitern Sie dann die Definition des Typs `AS_Exp` um die abstrakte Syntax von Ausdrücken mit Operatoren darin. Verwenden Sie als neue Konstruktoren möglichst `exp1` (für Ausdrücke mit einstelligen Operatoren) und `exp2` (für Ausdrücke mit zweistelligen Operatoren).

Schritt 2: Schreiben Sie (mit Papier und Bleistift) eine Grammatik für Ausdrücke, in denen die oben beschriebenen Operatoren, runde Klammern, Literale und Variablen-Bezeichner vorkommen können. Besprechen Sie diese Grammatik mit dem Betreuer Ihrer Übungsgruppe (auf diese Weise können Sie kleinere oder größere Fehler entfernen, ehe Sie viel Arbeit in die Implementierung der Grammatik und das Testen Ihrer Implementierung investieren).

Schritt 3: Ersetzen Sie in Ihrem Compiler die erste Zeile des `phrase`-Prädikats `CS_Exp`

```
phrase  CS_Exp(-> AS_Exp)
```

durch die folgenden 25 Zeilen

```
phrase  CS_Exp (-> AS_Exp)
   rule CS_Exp (-> EXP):
        CS_Exp1(-> EXP)
phrase  CS_Exp1(-> AS_Exp)
   rule CS_Exp1(-> EXP):
        CS_Exp2(-> EXP)
phrase  CS_Exp2(-> AS_Exp)
   rule CS_Exp2(-> EXP):
        CS_Exp3(-> EXP)
phrase  CS_Exp3(-> AS_Exp)
   rule CS_Exp3(-> EXP):
        CS_Exp4(-> EXP)
phrase  CS_Exp4(-> AS_Exp)
   rule CS_Exp4(-> EXP):
        CS_Exp5(-> EXP)
phrase  CS_Exp5(-> AS_Exp)
   rule CS_Exp5(-> EXP):
        CS_Exp6(-> EXP)
phrase  CS_Exp6(-> AS_Exp)
   rule CS_Exp6(-> EXP):
        CS_Exp7(-> EXP)
phrase  CS_Exp7(-> AS_Exp)
   rule CS_Exp7(-> EXP):
        CS_Exp8(-> EXP)
phrase  CS_Exp8(-> AS_Exp)
```

Die 5 Regeln hinter dieser Ersetzung gehörten vorher zum Prädikat `CS_Exp`. Nach der Ersetzung gehören sie zum Prädikat `CS_Exp8`. Ersetzen Sie deshalb in diesen Regeln `CS_Exp` durch `CS_Exp8`.

Damit haben Sie die "trivialen Regeln" einer *Grammatik für Ausdrücke* eingefügt. In den Ausdrücken können Operatoren mit Bindungsstärken von 0 bis 8 vorkommen.

Schritt 4: Ergänzen Sie (in der Definition von `CS_Exp`) die nicht-trivialen Regeln für die Operatoren mit der Bindungsstärke 0. Zufällig ist das hier nur *ein* Operator (der Konkatenationsoperator `conc()`) und somit nur *eine* Regel).

Schritt 5: Erweitern Sie die Prädikate `outExp`, `isString` und `pExp` so, dass sie auch Ausdrücke der Form bearbeiten können, die mit einem Konkatenationsoperator beginnen.

Schritt 6: Testen Sie den erweiterten Compiler mit möglichst simplen **alg31**-Quellprogrammen.

Wiederholen Sie die Schritte 4 bis 6 für jede Gruppe von Operatoren mit gleicher Bindungsstärke.

Für einen Abschlusstest Ihres **alg31**-Compilers sind (im Ordner `projects\alg31`) die Skripte `TST1.CMD` und `TST2.CMD` vorgegeben. `TST1.CMD` prüft, ob der Compiler "alle richtigen **alg31**-Programme" akzeptiert und richtig übersetzt. `TST2.CMD` prüft, ob der Compiler "alle falschen **alg31**-Programme" ablehnt.

7.3 alg32

Der **alg31**-Compiler *akzeptiert* mehr Programme als sein Vorgänger **alg30**.
Der **alg32**-Compiler soll mehr Programme *ablehnen* als sein Vorgänger **alg31**.

Der **alg31**-Compiler akzeptiert und übersetzt auch Befehle, die *Typfehler* enthalten, etwa die folgenden:

```
int    anna := "ABC";
bool   bert := "true"
string carl := 123;
int    dora; dora := "ABC";
write(10 + "ABC");
write(true < 123);
```

Der **alg32**-Compiler soll solche Typfehler erkennen.

Programmieren Sie für die Typprüfungen drei Pädikate mit folgenden Signaturen:

```
proc checkCmds(Cmds:AS_Cmd[])
proc checkCmd (Cmd: AS_Cmd)
proc checkExp (Exp: AS_Exp -> Type:AS_Type)
```

Jedes dieser Prädikate soll seinen Eingabeparameter auf Typfehler prüfen. Der Einfachheit halber soll nur der *erste* erkannte Typfehler gemeldet werden. Danach soll der Compiler sich *sofort beenden*.

Warum das Prädikat `checkExp` (nicht nur einen in-Parameter, sondern auch) einen out-Parameter hat, wird Ihnen wahrscheinlich von allein klar, wenn Sie die drei check-Prädikate programmieren (und sonst können Sie in der Übung danach fragen).

Für einen Abschlusstest Ihres **alg32**-Compilers sind (im Ordner `projects\alg32`) die Skripte `TST1.CMD` und `TST2.CMD` vorgegeben. `TST1.CMD` prüft, ob der Compiler "alle richtigen **alg32**-Programme" akzeptiert und richtig übersetzt. `TST2.CMD` prüft, ob der Compiler "alle falschen **alg32**-Programme" ablehnt.

7.4 alg33

Der **alg33**-Compiler soll seinen Vorgänger um `if`-Anweisungen (ohne und mit `else`), `while-do`-Anweisungen und `do-until`-Anweisungen erweitern.

Die folgenden Beispiele zeigen die konkrete Syntax dieser neuen **alg33**-Anweisungen:

```
if a < b then write("summe: "); writeln(sum); end if;

if a <= b and b <= c then
   writeln("b ist ok!");
else
   write("b: ");
   write(b);
   writeln(" ist nicht ok!");
end if;

while (a >= 0) do
   writeln(a);
   a := a - 1;
end while;

do
   writeln(a);
   a := a - 1;
until a <= 0 end until;
```

Für einen Abschlusstest Ihres **alg33**-Compilers sind (im Ordner `projects\alg33`) die Skripte `TST1.CMD` und `TST2.CMD` vorgegeben. `TST1.CMD` prüft, ob der Compiler "alle richtigen **alg33**-Programme" akzeptiert und richtig übersetzt. `TST2.CMD` prüft, ob der Compiler "alle falschen **alg33**-Programme" ablehnt.

7.5 Fehlermeldungen der alg3-Compiler

Die vorgegebenen Skripte namens `TST2.CMD` prüfen, ob der betreffende Compiler auf bestimmte Fehler "mit der richtigen Fehlermeldung" reagiert. Damit ein Compiler seinen Abschlusstest bestehen kann, müssen seine Fehlermeldung genau mit den hier festgelegten Texten übereinbstimmen.

Fehler der erkannt wurde:	**Text der Fehlermeldung**
Eine Variablenvereinbarung enthält den gleichen Bezeichner wie eine vorhergehende Variablenvereinbarung.	`Double declararation`

Auf der rechten Seite einer Zuweisung oder in einem Ausdruck wird eine nicht vereinbarte Variable angewendet.	`Undeclared variable`
In einer Variablen-Deklaration mit Initialisierung stimmt der Typ der Variablen nicht mit dem Typ des Initialisierungsausdrucks überein.	`Typ error in variable declaration`
In einer Zuweisung stimmt der Typ der Variablen auf der linken Seite nicht mit dem Typ des Ausdrucks auf der rechten Seite überein.	`Type error in assignment`
Der Operand des einstelligen Operators `not()` hat nicht den Typ `bool` oder der Operand des einstelligen Operators `minus()` hat nicht den Typ `int()`.	`Operand of unary operator has wrong type`
Einer der Operanden des zweistelligen Operators `or()` oder des Operators `and()` ist nicht vom Typ `bool()`.	`Operand of binary bool operator is not bool`
Die beiden Operanden eines (zweistelligen) Vergleichsoperators sind nicht vom selben Typ.	`Operands of a comparison are of different types`
Einer der Operanden eines zweistelligen arithmetischen Operators (`add()`, `sub()`, `mul()`, `div()`) ist nicht vom Typ `int()`.	`Operand of binary arithmetic operator is not int`
In einer `if-then`-Anweisung (ohne `else`) ist die Bedingung (der Ausdruck zwischen `if` und `then`) nicht vom Typ `bool()`.	`Condition in if-then-statement is not bool`
In einer `if-then-else`-Anweisung ist die Bedingung (der Ausdruck zwischen `if` und `then`) nicht vom Typ `bool()`.	`Condition in if-then-else-statement is not bool`
In einer `while-do`-Anweisung ist die Bedingung (der Ausdruck zwischen `while` und `do`) nicht vom Typ `bool()`.	`Condition in while-do-statement is not bool`
In einer `do-until`-Anweisung ist die Bedingung (der Ausdruck zwischen `until` und `end`) nicht vom Typ `bool()`.	`Condition in do-until-statement is not bool`

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

Entwickeln Sie (mit Papier und Bleistift) 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 Variablen-Bezeichner (`a`, `b` und `c`) zehn oder hundert Variablen-Bezeichner erlaubt wären?