Stichworte zu Compilerbau

für die Lehrveranstaltlung **Wahlpflichtfach Compilerbau (TB5-CPB)**
im Studiengang **Technische Informatik Bachelor**
im **WS13/14** bei Ulrich Grude

Im Laufe des Semesters können die TeilnehmerInnen **in dieser Datei** nach jeder Vorlesung ein paar Stichworte zum Inhalt der Vorlesung (und manchmal auch Korrekturen und Ergänzungen etc.) finden.

Übersicht über Termine und Tests

Die Lehrveranstaltung TB5-CPB besteht aus 2 Blöcken pro Woche, jeweils am **Freitag**
Block 1 (08.00-09.30 Uhr) seminaristischer Unterricht (SU) im Raum D211
Block 2 (10.00-11.30 Uhr) Übung im SWE-Labor, Raum DE16b

Insgesamt sind folgende Termine geplant (jeweils freitags):

28.09.13	15.11.13 Test05	20.12.13 Test10	31.01.14 Test14
18.10.13 Test01	22.11.13 Test06	Feiertage	07.02.14 Klausur
25.10.13 Test02	29.11.13 Test07	10.01.14 Test11	14.02.14 Rückgabe
01.11.13 Test03	06.12.13 Test08	17.01.14 Test12	
08.11.13 Test04	13.12.13 Test09	24.01.14 Test13	

Die **Hauptklausur** findet statt:

am **Fr 07.02.2014, 08.00-09.30** Uhr, Raum **DE37**
Rückgabe: 14.02.2014, 10.00 Uhr, Raum DE17

Die **Nachklausur** findet statt

am Fr 28.03.2014, 10.00-11.30 Uhr, Raum ??
Rückgabe: Mo 31.03.2014, 10.00 Uhr, Raum DE17

Wie bekommt man eine Note für den Übungsteil dieser LV?

Am Anfang von 14 SUs (um 8.00 Uhr im Raum D211) wird ein kleiner Test (ca. 10 bis 15 Minuten) geschrieben. Bei jedem Test kann man 20 Punkte bekommen. Mit 10 oder mehr Punkten hat man den Test bestanden, mit 9 oder weniger Punkten hat man ihn nicht bestanden. Wenn Sie an einem Test nicht teilnehmen, spielt der Grund dafür (hatte keine Lust, war krank, die S-Bahn war eingefroren, musste einen hohen Lottogewinn persönlich abholen, ...) *keine Rolle*.

Um für den Übungsteil dieser LV die Note **m.E.** (mit Erfolg) zu bekommen, müssen Sie zwei Bedingungen erfüllen:

Bedingung 1: Sie müssen mindestens 11 der 14 Tests bestehen.

Bedingung 2: Sie müssen zu einer Arbeitsgruppe gehören (die aus 2 oder höchstens 3 Personen besteht) und diese Arbeitsgruppe muss im Laufe des Semesters einen lauffähigen Compiler für die Sprache **alg3** entwickeln und dem Betreuer Ihrer Übungsgruppe vorführen. Dieser Compiler wird in Aufgabe **7. Ein Compiler für die Sprache alg3** in der Datei **Aufgaben.pdf** beschrieben.

Anmerkung: Sie können die Note **m.E.** also auch dann erreichen, wenn Sie bis zu 3 Tests nicht bestehen (z.B. weil Sie nicht daran teilgenommen haben). Sparen Sie sich diese 3 "Joker" so lange auf wie möglich. Seien Sie nicht überrascht, wenn z.B. die S-Bahnen im Januar nur unregelmäßig verkehren oder ein naher Verwandter Ihre Pflege benötigt, weil er sich auf einem rutschigen Weg verletzt hat, etc.

Zitat (von Wilhelm Busch): "Stets findet Überraschung statt / Da wo man's nicht erwartet hat!"

Wie bekommt man eine Modulnote für diese LV?

Wenn Sie für den Übungsteil dieser LV die Note **m.E.** bekommen haben, dürfen Sie an den Klausuren für diese LV teilnehmen (an der Hauptklausur am Ende des WS13/14 oder an der Nachklausur kurz vor dem SS14).

Die in der Klausur erreichte Note ist dann die Modulnote.

Wenn Sie keine der beiden Klausuren bestehen (weil Sie nicht teilnehmen durften oder aus anderen Gründen) ist die Modulnote gleich 5,0.

Unterlagen für Tests und Klausuren

Zu jedem der *Tests* dürfen Sie als Unterlage mitbringen:
1 Blatt, max. DIN A 4, **beliebig beschriftet** (von Hand oder mit Drucker/Kopierer, nur auf der Rückseite oder nur auf der Vorderseite oder beidseitig, schwarz-weiß oder farbig, ordentlich oder chaotisch, ...).

Zur *Klausur* (und zur *Nachklausur*) dürfen Sie als Unterlagen mitbringen:
5 Blätter, max. DIN A 4, **beliebig beschriftet** (siehe oben).

Hinweis: 10 einseitig beschriebene Blätter sind (wenn man genau nachzählt) 10 Blätter
und nicht 5 Blätter! Somit sind 10 Blätter nicht erlaubt.

1. SU Fr 11.10.13

42 Studenten des Studiengangs TI-B haben am Anfang des WS13/14 das Wahlpflichtfach Compilerbau belegt (per OnlineBelegung).
Davon sind 26 zum ersten Termin erschienen.

An diesem Wahlpflichtfach sollten Sie nur teilnehmen, wenn Sie fest vorhaben,
1. regelmäßig und pünktlich zu den Übungen und Vorlesungen zu kommen und
2. zusätzlich mindestens einmal pro Woche zu Hause den behandelten Stoff zu wiederholen und die Aufgaben zu bearbeiten.

Wenn Sie diese organisatorischen Voraussetzungen erfüllen, haben Sie eine gute Chance, auch die inhaltlichen Anforderungen dieser Lehrveranstaltung zu erfüllen und am Ende eine gute bis sehr gute Note zu bekommen.

Falls zu viele StudentInnen an dieser LV teilnehmen wollen: **Verlosung** durchführen.
War nicht nötig.

Übersicht über Termine, Tests, Aufgaben und Noten

Siehe oben S. 1

Hat jemand noch Fragen zur Organisation dieser Lehrveranstaltung?

Jetzt geht es los mit dem Stoff!

1. Grundbegriffe

Alphabet: Eine (endliche, nicht-leere) Menge von *Symbolen.*
Symbol: Ein einzelnes (nicht-transparentes) *Zeichen* oder
eine (endliche, nicht-leere) *Folge solcher Zeichen*.
Wort: Eine (endliche, leere oder nicht-leere) *Folge von Symbolen.*
Satz: Bedeutet im Zusammenhang mit *formalen Sprachen* genau das Gleiche wie "Wort".
Formale Sprache: Eine (leere, endliche oder unendliche) Menge von *Worten* (oder: von *Sätzen).*

Anmerkung 1: In diesen Definitionen geht es um *Mengen* ("ohne Reihenfolge") und *Folgen* ("mit Reihenfolge") und darum, ob die Menge/Folge *leer*, *endlich* oder *unendlich* sein kann.

Anmerkung 2: Transparente Zeichen (engl. whitespace characters) sind Blanks (Leerzeichen), Tab-Zeichen und Zeilenwechsel-Zeichen (engl. carriage return \r and newline \n).

Beispiel-01: Symbole:

`0` oder `A` oder `[` oder `if` oder `class` etc.

Beispiel-02: Alphabete:

A01: {0 1}
A02: {`A B C a b c 0 1 2 3 [ ] ( )`}
A03: {`class if while switch ( ) { } [ ] ; , \ A B ... Z ...`}

Das Alphabet A03 ist hier nur *angedeutet*. Auf diesem Alphabet beruht die formale Sprache Java.

Beispiel-03: Formale Sprachen über dem Alphabet A01:

S01: {0 1 10 11 101}
Zu dieser Sprache gehören die Binärzahlen von eins bis fünf. Diese Sprache ist endlich.

S02: {0 1 10 11 101 110 111 1000 ...}
Zu dieser Sprache sollen alle Binärzahlen gehören. Diese Sprache ist unendlich.

S03: {0 1 00 01 10 11 000 001 010 011 100 101 110 111 100 ...}
Zu dieser Sprache sollen alle 0-1-Folgen gehören. Zwischen S02 und S03 gibt es einen subtilen Unterschied. **Frage**: Welche Worte sollen wohl zu S03 gehören, aber nicht zu S02?
(Worte mit "unnötigen führenden Nullen", z.B. 01 oder 0001011. Die Zahl 0 besteht zwar aus einer führenden Null, aber die ist nötig).

Die *Menge aller Java-Programme* ist eine formale Sprache über dem Alphabaet A03. Jedes Java-Programm ist ein Wort (oder: ein Satz) dieser Sprache. Entsprechendes gilt auch für andere Programmiersprachen.

Anmerkung: Sprachen wie *Deutsch* oder *Englisch* oder *Türkisch* etc. bezeichnet man (im Gegensatz zu den *formalen Sprachen*, um die es hier geht) als *natürliche Sprachen*. Natürliche Sprachen sind grundsätzlich sehr viel kompliziertere und geheimnisvollere Gebilde als formale Sprachen.

Endliche formale Sprachen (wie z.B. S01) kann man beschreiben, indem man alle Worte der Sprache angibt. Bei *unendlichen* Sprachen ist eine solche "Beschreibung durch eine vollständige Wortliste" grundsätzlich nicht möglich.

Problem: Wie kann man auch *unendliche* formale Sprachen exakt beschreiben?

Eine Lösung des Problems: Mit **formalen Grammatiken**.

Es gibt verschiedene *Arten* von formalen Grammatiken. Die in der Praxis bei weitem am häufigsten verwendeten Grammatiken sind so genannte **Typ-2-Grammatiken**. Die werden häufig auch als **kontextfreie Grammatiken** bezeichnet. Eine bessere Bezeichnung wäre: **kontextunfähige Grammatiken** (weil diese Grammatiken "nicht frei von einem störenden Kontext" sind, sondern "unfähig, einen gewünschten oder notwendigen Kontext zu beschreiben").

Das folgende Arbeitsblatt austeilen und besprechen.

Arbeitsblatt für den 1. SU, Fr 11.10.2013

Eine Typ-2-Grammatik besteht (im Kern und hauptsächlich) aus *Regeln*.

Beispiel-04: Eine Typ-2-Grammatik G04, die aus acht Regeln besteht:

```
R01: Zahl     : Vorzeich ZiffFo    // Zahl ist das Startsymbol von G04
R02: Zahl     : ZiffFo
R03: Vorzeich : "+"
R04: Vorzeich : "-"
R05: ZiffFo   : Ziff
R06: ZiffFo   : Ziff ZiffFo
R07: Ziff     : "0"
R08: Ziff     : "1"
```

Jede Regel besteht aus einem **Trennzeichen** (hier: `:`), welches eine **linke Seite** (z.B. `Zahl`) von einer **rechten Seite** (z.B. `Vorzeich ZiffFo`) trennt.

Außerdem wurde hier jede Regel mit einem *Namen* versehen (R01:, R02:, ... etc.), damit man leichter über sie reden kann. Diese Namen sind aber kein wesentlicher Bestandteil der Regeln und können auch weggelassen werden.

Aufgabe-01: Geben Sie eine Grammatik G05 an, die die Sprache S05 aller 0-1-Folgen beschreibt:
S05: {`0 1 00 01 10 11 000 001 010 011 100 101 110 111 0000 0001` ...}
Verwenden Sie `B` als Startsymbol von G05 (damit man verschiedene Lösungen dieser Aufgabe leichter miteinander vergleichen kann).

Aufgabe-02: Geben Sie eine Grammatik G06 an, die die folgende Sprache S06 beschreibt:
S06: {`0 1 10 11 100 101 110 111 1000` ...}
S06 soll außer dem Wort `0` alle mit `1` beginnenden 0-1-Folgen enthalten.
Verwenden Sie `V` als Startsymbol von G06.

Aufgabe-03: Geben Sie eine Grammatik G07 an, die die folgende Sprache S07 beschreibt:
S07: {`0 1 01 11 001 011 101 111 0001` ...}
S07 soll außer dem Wort `0` alle mit `1` endenden 0-1-Folgen enthalten.
Verwenden Sie `N` als Startsymbol von G07.

Aufgabe-04: Geben Sie eine Grammatik G08 an, die die Sprache S08 aller Binärbrüche beschreibt. Ein Binärbruch besteht aus einem (Binär-) Punkt `"."`. Vor diesem Punkt dürfen beliebig viele Binärziffern stehen (aber nicht weniger als eine). Nach dem Punkt dürfen ebenso beliebig viele Binärziffern stehen (aber nicht weniger als eine). Vor dem Punkt sollen aber keine "unnötige führende Nullen" stehen. Entsprechend sollen nach dem Punkt keine "unnötigen nachhinkenden Nullen" stehen.

Beispiel für Binärbrüche: `0.0 1.0 0.1 1.1 1000.101 1011.00001`
Die folgenden Worte sind *keine* Binärbrüche:

```
10.      // Nach dem Punkt steht keine Ziffer
.01      // Vor dem Punkt steht keine Ziffer
00.0     // Vor dem Punkt steht eine unnötige führende Null
1.010    // Nach dem Punkt steht eine unnötige nachhinkende Null
```

Verwenden Sie `F` (wie fraction) als Startsymbol von G08.

Aufgabe-05: Geben Sie eine Grammatik GA1 für die *ancestor language* A1 an:
A1 = {`mother, father, grandmother, grandfather, greatgrandmother, greatgrandfather, greatgreatgrandmother, ... greatgreatgreatgreatgrandfather, ...` }
Verwenden Sie `ancestor` als Startsymbol von GA1.

Eine Typ-2-Grammatik besteht (im Kern und hauptsächlich) aus *Regeln*.

Beispiel-04: Eine Typ-2-Grammatik G04, die aus acht Regeln besteht:

```
R01: Zahl     : Vorzeich ZiffFo    // Zahl ist das Startsymbol von G04
R02: Zahl     : ZiffFo
R03: Vorzeich : "+"
R04: Vorzeich : "-"
R05: ZiffFo   : Ziff
R06: ZiffFo   : Ziff ZiffFo
R07: Ziff     : "0"
R08: Ziff     : "1"
```

Jede Regel besteht aus einem **Trennzeichen** (hier: `:`), welches eine **linke Seite** (z.B. `Zahl`) von einer **rechten Seite** (z.B. `Vorzeich ZiffFo`) trennt.

Außerdem wurde hier jede Regel mit einem *Namen* versehen (R01:, R02:, ... etc.), damit man leichter über sie reden kann. Diese Namen sind aber kein wesentlicher Bestandteil der Regeln und können auch weggelassen werden.

Die ersten beiden Regeln (R01 und R02) besagen in etwa, dass eine `Zahl` entweder aus einem `Vorzeich` gefolgt von einer `ZiffFo` oder nur aus einer `ZiffFo` besteht. Die Regeln R03 und R04 besagen, dass ein `Vorzeich` entweder ein Pluszeichen `"+"` oder ein Minuszeichen `"-"` ist.

Die Regeln R05 und R06 sind besonders "mächtig": Sie besagen zusammen, dass eine `ZiffFo` nur aus einer `Ziff` besteht oder aus einer `Ziff` gefolgt von einer `ZiffFo` (die ihrerseits nur aus einer `Ziff` besteht oder aus einer `Ziff` gefolgt von einer `ZiffFo` (die ihrerseits nur aus einer `Ziff` besteht oder aus einer `Ziff` gefolgt von einer `ZiffFo` (die ihrerseist ...))).

Von der Regel R06 sagt man auch, sie sei *rekursiv* (weil `ZiffFo` sowohl *vor* als auch *nach* dem Trennzeichen `:` vorkommt).

In den Regeln einer Grammatik unterscheidet man zwei Arten von Symbolen: **Zwischensymbole** (engl. non-terminal symbols) und **Endsymbole** (engl. terminal symbols).

In G04 kommen die folgenden vier Zwischensymbole vor:
`{Zahl Vorzeich ZiffFo Ziff}`. Sie bilden das *Alphabet der Zwischensymbole* von G04.

In G04 kommen die folgenden vier Endsymbole vor:
`{+ - 0 1}`. Sie bilden das *Alphabet der Endsymbole* von G04.

Eines der Zwischensymbole muss (irgendwie) als **Startsymbol** der Grammatik kenntlich gemacht werden. Im obigen Beispiel geschah das durch den Kommentar hinter der Regel R01.

Konvention: Wenn der Autor einer Grammatik nicht ausdrücklich etwas anderes festlegt, ist das Zwischensymbol *am Anfang der ersten Regel* (im obigen Beispiel also das Symbol `Zahl`) das Startsymbol.

Wenn man eine Grammatik schreibt, muss man irgendwie deutlich machen, welche Symbole *Zwischen-* und welche *Endsymbole* sind. Im obigen Beispiel wurden dazu die Endsymbole in *doppelte Anführungszeichen* eingeschlossen. Man kann aber auch *einfache Anführungszeichen* oder *Farben* oder *Unterstreichungen* etc. verwenden oder zuerst die beiden Alphabete und erst dann die Regeln angeben.
Hauptsache: Die Leser der Grammatik können Zwischen- und Endsymbole klar unterscheiden.

Anmerkung: Dass in der Grammatik G04 die Anzahl der Zwischensymbole (4) *gleich* der Anzahl der Endsymbole ist und es zu jedem Zwischensymbol *genau 2 Regeln* gibt, die mit ihm anfangen, ist reiner Zufall (und bei den meisten Grammatiken anders).

Zur Entspannung: **Was kostet ein Studium an der Beuth Hochschule?**

Haushalt der Beuth Hochschule ca. 80 Millionen [Euro pro Jahr].
An der Beuth Hochschule studieren ca. 10 Tausend StudentInnen.
Also kostet das Studieren an der Beuth Hochschule ca. 8000 [Euro pro StudentIn und Jahr].
Ein Bachelor-Studium (6 Semester, 3 Jahre) kostet also ca. 24 Tausend [Euros pro StudentIn].

Lösungen zum Arbeitsblatt für den 1. SU, Fr 28.09.12

Lösung-01:

```
1   B : "0"
2   B : "1"
3   B : B "0"
4   B : B "1"
```

Lösung-02:

Der entscheidende Trick bei dieser Lösung besteht darin, dass wir die vorige Grammatik (mit dem Startsymbol `B`) voraussetzen und benutzen. Deshalb beginnen wir bei den Regelnummern nicht wieder bei `1`, sondern setzen die Nummerierung mit `5`, `6`, ... etc. fort.

```
5   V : "0"
6   V : "1"
7   V : "1" B
```

Lösung-03:

```
8   N : "0"
9   N : "1"
10  N : B "1"
```

Lösung-04:

```
11  F : V "." N
```

Grammatiken haben Ähnlichkeit mit *Unterprogrammen* (oder: Methoden, Funktionen): Wenn man ein Unterprogramm geschrieben hat, kann man es in anderen Unterprogrammen aufrufen (statt "alles noch mal zu programmieren"). Wenn man eine Grammatik geschrieben hat, kann man ihr Startsymbol in anderen Grammatiken benutzen (statt "alle Regeln noch mal hinzuschreiben").

Lösung-05, Grammatik GA1 für die ancestor language:

```
1   ancestor : ancestor1
2   ancestor : ancestor2
3   ancestor : ancestor3
4   ancestor1: "mother"
5   ancestor1: "father"
6   ancestor2: "grand" ancestor1
7   ancestor3: "great" ancestor2
8   ancestor3: "great" ancestor3
```

1. Übung am Fr 11.09.13

1. Im SWE-Labor einloggen.

2. Arbeitsgruppen bilden (normalerweise besteht eine Gruppe aus 2 Personen).

3. Jede Gruppe sollte sich einen Stick mit dem Ordner `cpb` darauf besorgen. Wer einen Stick vom Betreuer der Übung akzeptiert und an der Hauptklausur teilnimmt, darf den Stick behalten. Wer nicht an der Hauptklausur teilnimmt, muss den Stick (vor der Klausur) zurückgeben.

~~4. Das Archiv `cyan.zip` aus dem Internet laden (von der Adresse http://cyan.compilertools.net/) und in den Ordner `cpb` auf dem Stick entpacken (so dass ein Unterordner `cpb\cyan` entsteht).~~

5. Die Grammatik GA1 für die ancestor language in ein Gentle-Programm umschreiben und in einer Datei `cpb\projects\ancester01\spec.g` abspeichern. Das Gentle-Programm compilieren und mit verschiedenen Eingaben testen.

6. Das 4-seitige Papier **Vorlagen für ancestor-Compiler** wird ausgeteilt. Auf Seite 1 wird ("von allen gemeinsam") der Parser für die ancestor language zu einem Compiler ergänzt, der Quell-Worte wie `father, mother, grandfather` etc. in Zahlen wie `1` oder `2` etc. übersetzt. Der Compiler wird in der Datei `cpb\projects\ancester02\spec.g` abgespeichert, compiliert und getestet.

Ganz entsprechend die Seiten 2, 3, 4 bearbeiten (so weit wir kommen).

2. SU Fr 18.10.13

Heute **Test 1**

Das Papier "Vorlagen für ancestor-Compiler" weiter bearbeiten

ancestor03 (S. 2 des Papiers)

Motivation: Wir wollen unseren Parser für die ancestor language zu einem Compiler erweitern, der beim Einlesen eines Quell-Wortes daraus eine *Zwischendarstellung* erzeugt, die man "leichter weiterverarbeiten kann" als das Quellwort selbst.

Den Gentle-Typ `AS_anc3` besprechen

Aufgabe: Schreiben Sie 5 Werte des Typs `AS_anc3` auf ein Papier.

Wir versehen jedes Zwischensymbol (`ancestor`, `ancestor1`, `ancestor2`, `ancestor3`) mit einem out-Parameter vom Typ `AS_anc3`, etwa so: `(-> AS_anc3)`.

Welche out-Parameter müssen die Regeln von `ancestor1` liefern? (`mo()` bzw. `fa()`)

Bei `ancestor2` arbeiten wir wieder "von rechts nach links":
Rechts geben wir dem von `ancestor1` geliefert Wert einen Namen (z.B. `(-> A)`).
Dann sollte auf der linkenk Seite `ancestor2` folgendes liefern: `(-> g(A))`.

Die übrigen Einträge sollten jetzt in den Arbeitsgruppen durchgeführt werden.

ancestor04 überspringen wir erstmal.

ancestor05 (S. 4 des Papiers)

Hier ist alles wie bei `ancestor03`, aber wir ergänzen ein Prädikat `trout` (translate and output), welches eine Zwischendarstellung (d.h. einen Wert des Typs `AS_anc3`) ins Deutsche übersetzt und ausgibt. Untypischerweise ist es bei diesem Prädikat günstig, erst die "komplizierten" (langen) Zwischendarstellungen zu bearbeiten und dann die einfacheren (kürzeren).

```
 1 proc trout(AS_anc3)
 2    rule trout(g(g(AS))):
 3       "ur"
 4       trout(g(AS))
 5    rule trout(g(AS)):
 6       "gross"
 7       trout(AS)
 8    rule trout(mo())
 9       "mutter\n"
10    rule trout(fa())
11       "vater\n"
```

Wir haben letztes Mal und heute 4 der 6 Teile von Aufgabe 1 gemeinsam bearbeitet. Die übrigen beiden Teilaufgaben (**ancestor04** und **ancestor06**) und evtl. eine *freiwillige* weitere Teilaufgabe (**ancestor07**) sollen Sie selbständig in Ihren Arbeitsgruppen lösen.

Jetzt stehen **zwei wichtige Punkte** auf dem Lernplan, die uns länger beschäftigen werden:

1. Sie werden lernen, *Prädikate zu programmieren*.
2. Sie werden lernen, (kontextfreie) *Grammatiken zu schreiben*.

Heute befassen wir uns zuerst mit Typen in Gentle und fangen dann damit an, Prädikate zu programmieren (indem wir gemeinsam die **Aufgabe 4** in der Datei **Aufgaben.pdf** bearbeiten).

Typen in Gentle

1. Es gibt zwei vordefinierte Typen `int` und `string`.
2. Ein Gentle-Programmierer kann weitere Typen vereinbaren (wie z.B. in **ancestor03**).
3. Zu jedem Typ `T` gibt es einen Typ `T[]` (lies: *Liste von T-Werten*, kurz: Liste von T, engl. list of T).

In Gentle gibt es auch für die Werte von Listen-Typen *Literale*.

Beispiel-01: Literale des Typs `int[]`

```
1 int[]              // Eine leere Liste von int-Werten
2 int[17, 25, 12]    // Eine Liste mit 3 Elementen
```

Anhand von Beispielen (aus dem Papier **TypenInGentle.pdf** auf meiner Netzseite) werden folgende Arten von Typen besprochen:

Aufzählungstypen (enum types, enumeration types)
Verbundtypen (record types, struct types)
Variante Verbundtypen (variant record types, union types)
Rekursive Typen

Anmerkung: In C werden rekursive Typen mit Hilfe von *Zeigern* (engl. pointer) realisiert.
Gentle vermeidet dieses gefähliche, fehleranfällige Konstrukt.

2. Übung am Fr 18.10.13

1. Die "alte Gentle-Distribution" (mit der wir in der 1. Übung gearbeitet haben), gegen die "neuste Gentle-Distribution" Cyan-Pre-0-3 austauschen.

2. Die im SU besprochenen Compiler `ancester03` und `ancester05` eintippen und testen.

3. Wer dann noch Zeit hat: Im Ordner `projects\pred02` die Datei `spec.g` bearbeiten wie folgt:

- Compilieren und ausführen. Es sollten 3 Fehlermeldungen erscheinen
  (die besagen "das Prädikat `length` liefert immer den fehlerhaften Wert -999")
- Das Prädikat `length` in der Datei `spec.g` verbessern, bis keine Fehlermeldungen mehr ausgegeben werden (sondern: "`Nr. of tests: 3, Nr. of errors: 0`").

4. Die drei Aufruf des Prädikats `test_sum` aktivieren ("entkommentarisieren"),
   den Rumpf des Prädikats `sum` verbessern, bis die Datei `spec.g` sich ohne Fehlermeldungen compilieren und ausführen läßt.

Und so weiter. Möglichst viele (in der Datei `spec.g` beschriebene) Prädikate auf diese Weise schreiben und testen.

Zwischendurch, in kleinen Gruppen: "Blitzeinführung: Listen-Typen in Gentle". Wird im nächsten SU nochmal für alle und gründlich gemacht.

3. SU Fr 25.10 13

Heute **Test 2**

Listen-Typen in Gentle

Zu jedem Gentle-Typ `T` gibt es einen weiteren Typ `T[]` ("Liste von T")

Diese Regel gilt für alle Typen, also auch für Listen-Typen.

Beispiele für Typen und Listen-Typen:

```
 1  type Tag mo() di() mi() do() fr() sa() so()
 2  type Student st(Name:string, MatrikelNr:int)
 3
 4  Zu diesen Typen gibt es automatisch folgende Listen-Typen:
 5  Tag[], Tag[][], ..., Person[], Person[][], ...
 6
 7  Zu den vordefinierten Typen int und string gibt es die folgenden Listen-Typen:
 8  int[], int[][], ..., string[], string[][], ...
```

Beispiel für Listen-Literale:

```
 9  Tag[di(), fr(), so()],   Tag[mo(), mo()], Tag[]
10  Student[st("Arno Schmitz", 763541),   st("Berta Belau", 773465)], Student[]
11  int[10, 20, 30],   int[]
12  int[][int[10, 20, 30], int[]],   int[][int[10, 20], int[30, 40], int[50, 60]]
```

Muster (engl. pattern)

Was ein Muster ist, wird später noch genauer behandelt.

Merke-01: Ein Muster kann *Variablen* enthalten
Merke-02: Ein Muster beschreibt eine *Menge von Werten*

Beispiel-02: *Muster* des Typs `int[]` (beschreiben *Mengen* von Werten des Typs `int[]`)

```
 1 Muster                Welche Menge von int-Listen beschreibt das Muster?
 2 int[]                 {int[]} Diese Menge enthält nur 1 Liste
 3 int[17,25,12]         {int[17, 25, 12]} Auch nur 1 Liste
 4 int[E]                Alle int-Listen, die genau ein Element enthalten
 5 int[E1,E2,E3]         Alle int-Listen, die genau drei Elemente enthalten
 6 int[E1,25,E2]         Alle int-Listen, die genau drei Elemente enthalten,
 7                       von denen das zweite gleich 25 ist.
 8 int[E::R]             Alle int-Listen, die aus einem ersten Element E und
 9                       einer Restliste R bestehen. Die Restliste R kann
10                       beliebig viele Elemente (0 oder mehr) enthalten.
11 int[E1,E2::R]         Alle int-Listen, die aus zwei Elementen E1 und E2 und
12                       einer Restliste R bestehen. Die Restliste R kann
13                       beliebig viele Elemente (0 oder mehr) enthalten.
14 int[E1,25::R]         Alle int-Listen, die aus zwei Elementen E1 und 25 und
15                       einer Restliste R bestehen. Die Restliste R kann
16                       beliebig viele Elemente (0 oder mehr) enthalten.
```

Frage: Welche `int`-Liste gehört *nicht* zu der vom Muster `int[E::R]` beschriebenen Menge?
(Die leere `int`-Liste `int[]`)

Solche Muster werden z.B. zum Lösen der **Aufgabe 4** benötigt.

Verschiedene Arten von Gentle-Prädikaten

`phrase`-Prädikate und `token`-Prädikate
dienen dazu, die Syntax einer Quellsprache zu beschreiben

`proc`-Prädikate und `condition`-Prädikate
dienen dazu, alle anderen Arbeiten zu erledigen

`sweep`-Prädikate
dienen dazu, das Traversieren komplexer Strukturen zu vereinfachen (werden wir nicht benutzen)

Für alle Arten von Prädikaten gilt:

Ein Aufruf eines Prädikats kann *gelingen* oder *misslingen* (engl. suceed or fail).

Wenn der Ausführer beim Ausführen eines Prädikataufrufs alle Regeln des Prädikats "durchprobiert hat" und keine davon vollständig ausgeführt werden konnte, dann *misslingt* der Aufruf. Wenn der Ausführer eine der Regeln vollständig ausführen kann, dann *gelingt* der Aufruf.

Unterschied zwischen proc- und condition-Prädikaten

Wenn ein Aufruf eines **proc**-Prädikats *misslingt*, dann ist das ein Fehler des *Programmierers* (und das betreffende Gentle-Programm wird sofort mit einer Fehlermeldung abgebrochen). In einem korrekten Gentle-Programm müssen alle Aufrufe aller proc-Prädikate *gelingen*.

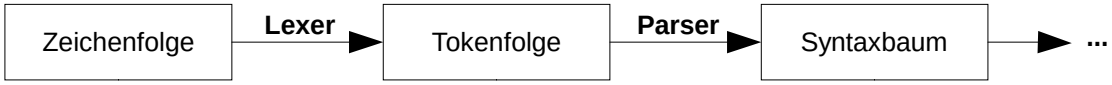
Zum "Wesen eines **condition**-Prädikates" gehört es, dass einige seiner Aufrufe *gelingen* und andere Aufrufe *misslingen* (ganz ähnlich wie die Bedingung in einem `if`-Befehl manchmal den Wert `true` und manchmal den Wert `false` haben kann).

Wozu token-Prädikate?

Die Grammatiken, die wir bisher kennengelernt haben, heißen offiziell *kontextfrei Grammatiken* (kurz: KFG) oder Typ-2-Grammatiken (engl. contextfree grammars, abbr. CFG). Theoretisch kann man z.B. eine Programmiersprache wie Algol60 oder Java vollständig durch eine solche KFG beschreiben. Und aus einer solchen Grammatik könnte man einen vollständigen Parser für die betreffende Sprache erzeugen

Früher (vor etwa 40 bis 50 Jahren) gab es in diesem Zusammenhang aber ein praktisches Problem: So ein Parser wäre damals *viel zu langsam* gewesen. Deshalb erfand man sogenannte **Lexer** (oder: **Scanner**). Die nehmen dem Parser viele "einfache Prüfungen" ab und können diese Prüfungen viel schneller erledigen, als der Parser es könnte.

Zusammenspiel von Lexer und Parser. Wie wird ein Quellprogramm zuerst bearbeitet:



Heutige PCs sind so schnell, dass diese Arbeitsteilung zwischen Lexer und Parser nicht mehr unbedingt nötig wäre (und es gibt ein paar Compiler ohne Lexer, nur mit Parser). Aber mit dieser Arbeitsteilung zwischen Lexer und Parser kann man auch das "*Problem der transparenten Zeichen*" (engl. white space problem) einfach und übersichtlich lösen: Wo in einem Quellprogramm dürfen transparente Zeichen wie Blank (Leerzeichen), Tab-Zeichen oder Zeilenwechsel stehen? Diese Frage mit den Regeln einer Grammatik zu beantworten ist kompliziert und würde die Grammatik unübersichtlich machen. Statt dessen erwähnt man transparente Zeichen in der Grammatik überhaupt nicht und läßt den Lexer alle transparenten Zeichen prüfen und entfernen, so dass der Parser sie nicht mehr bearbeiten muss.

Ein *Lexer* liest eine *Zeichenfolge* ein, zerlegt sie in eine Folge von Lexemen und übersetzt jedes Lexem in ein sog. *Token*. Der Parser liest eine *Folge von Token* ein und konstruiert daraus einen *Syntaxbaum*. Ein *Token* ist ein (Vorkommen von einem) *Zwischensymbol* (engl. nonterminal symbol) der Quellsprachen-Grammatik.

Beispiel: Das Quellprogramm

```
print(12*(++otto - 321));
```

wird vom Lexer in die folgenden 10 Lexeme zerlegt:

| **print** | **(** | **12** | ***** | **(** | **++** | **otto** | **-** | **321** | **)** | **)** | **;** |

und jedes Lexem wird in ein Token übersetzt, etwa so:

```
BEZ  K1A  LIT  OP2  K1A  OP3  BEZ  OP1  LIT  K1Z  K1Z  SEM
```

Dabei sollen die Token-Namen an folgendes erinnern:

`IDE` Bezeichner,
`K1A` Klammer 1 auf
`K1Z` Klammer 1 zu
`LIT` Literal
`OP1` Operator Bindungstärke 1
`OP2` Operator Bindungstärke 2
`OP3` Operator Bindungstärke 3
`SEM` Semikolon

Wichtig: Für den Parser ist es unwichtig, *welches* (Ganzzahl-) Literal oder *welcher* Bezeichner an einer bestimmten Stelle steht, er will erstmal nur wissen, ob da ein *Literal* oder ein *Bezeichner* steht. Darum können die Lexeme `12` und `321` auf gleiche `LIT`-Token und die Lexeme `print` und `otto` auf gleiche `IDE`-Token abgebildet werden.

Konkret kann ein `IDE`-Token aus einem `int`-Wert und einem Zeiger bestehen. Der `int`-Wert besagt, dass es sich um ein `IDE`-Token handelt, und der Zeiger zeigt auf einen String (z.B. auf `"print"` oder `"otto"`). Der Zeiger eines `LIT`-Tokens zeigt auf den Wert des Literals (z.B. `12` oder `321`). Die übrigen Token können durch je einen `int`-Wert realisiert werden und brauchen *keinen* zusätzlichen Zeiger.

In Gentle gilt:
Aus den `token`-Prädikaten und den `string`-Literalen in `phrase`-Prädikaten wird ein *Lexer* generiert. Aus den `phrase`-Prädikaten wird ein *Parser* generiert.

Die **Aufgabe 4** (In Gentle `proc`- und `condition`-Prädikate programmieren) in kleinen Gruppen bearbeiten (siehe auch die Datei `spec.g` im Ordner `projects\pred02`).

Zur Entspannung: **Christian Morgenstern** (1871-1914, München-Meran)

Lyriker, Redakteur, Übersetzer (Ibsen, Strindberg). In seinen Gedichten kommen auch mathematische Ideen vor. Im folgenden Gedicht geht es um die Evolution großer Tiere und Zahlen:

Anto-Logie

Im Anfang lebte, wie bekannt, als größter Säuger der Gig-ant. ...

3. Übung am Fr 25.10.13

1. Möglichst viele der Prädikate in der Datei `spec.g` im Ordner `projects\pred02` eingeben und testen.

2. In der Datei `spec.g` im Ordner `projects\alg30` die `token`-Prädikate ansehen und verstehen.

4. SU Fr 01.11.13

Heute **Test 3**

Das Papier **Chomsky-Grammatiken.pdf** (4 Seiten) besprechen.

Zur Entspannung: **Anders Hejlsberg** (geb. 1960, Dänemark)

Wichtiger Software-Architekt.

1980: Turbo-Pascal
1991: Bei Borland: Delphi
1996: Bei Microsoft: VBA, Mitarbeit an .NET, Chefentwickler von C#
2012: Bei Microsoft: TypeScript (vorgestellt am 01.10.2012)

TypeScript ist eine Erweiterung von JavaScript um Typen, Klassen und Module. Der Compiler tsc übersetzt TypeScript nach JavaScript und ist Open-Source. Auf einer "Spielwiese"
(siehe http://www.typescriptlang.org/Playground/) kann man den Compiler ausprobieren.

Hejlsberg hat in Kopenhagen Engineering studiert, sein Studium aber nicht abgeschlossen.

4. Übung am Fr 01.11.13

1. In der Datei `spec.g` im Ordner `projects\alg30` die `token`-Prädikate ansehen und verstehen.

2. Die Aufgabe **2 GrammatikenA** bearbeiten (erst mit Papier und Bleistift, dann mit Rechner im Ordner projects\GrammatikenA)

3. Die Aufgabe **3 GrammatikenB** anfangen?

5. SU Fr 08.11.13

Heute **Test 4**

Was bedeutet "kontextsensitiv" bei Grammatiken? Erklärungen 1

Beispiel für eine typische kontextsensitive Regeln:

`a B X c D -> a B Y z c D`

"Im Kern" sagt diese Regel, dass man **X** ableiten kann nach **Y z**.
Das ist aber *nicht immer* erlaubt, sondern nur, wenn das **X** in einem bestimmten Kontext (`a B` und `c D`) steht. Dieser Kontext muss vorhanden sein, wird durch eine Anwendung der Regel aber *nicht* verändert.

Was bedeutet "kontextsensitiv" bei Grammatiken? Erklärungen 2

Programme aller höheren Programmiersprachen müssen bestimmte **Kontextbedingungen** erfüllen um um formal korrekt zu sein, z.B. die folgenden:

KB1: Die Benutzung einer Variablen V ist nur erlaubt, wenn V auch *vereinbart* wurde.

KB2: Die Benutzung eines Unterprogramms U ist nur erlaubt, wenn U auch *vereinbart* wurde.

KB3: Wurde ein Unterprogramm U mit n *formalen Parametern* vereinbart, dann müssen in jedem Aufruf von U genau n *aktuelle Parameter* angegeben werden.

Die Benutzung einer Variablen (z.B. in einem Ausdruck) ist also nur "im Kontext einer entsprechenden Vereinbarung" erlaubt. Für Unterprogramme entsprechend.

Theorem: Es ist *nicht möglich*, solche Kontextbedingungen mit einer Typ-2-Grammatik (einer kontextfreien oder "kontextunfähigen" Grammatik) zu beschreiben.

Daraus folgt z.B. für die Sprache C: Die *kontextfreie Grammatik von C* beschreibt *nicht* die Sprache C, sondern eine "etwas größere Sprache". Zu der gehören alle formal korrekten C-Programme, aber auch solche Zeichenketten, die ähnlich aussehen wie C-Programme, aber bestimmte Kontextbedingungen nicht einhalten. Für Pascal, Ada, Java, ... ganz entsprechend.

Grammatiken und Sprachen

Def.: Eine Typ-3-Sprache ist eine Sprache, die man durch eine Typ-3-Grammatik beschreiben kann.

Für die Typen 2, 1, 0 entsprechend. Man gibt immer eine möglichst große Typ-Nr an.

Beispiel-Sprachen, von denen man schon bewiesen hat, von welchem Typ sie sind:

L1 = $\{a^n b^m \mid n \geq 1, m \geq 1\}$ ≥

Diese Sprache ist vom **Typ 3**. Hier eine rechts-lineare Grammatik G10 für L1:

```
R1: A → aA      R3: B → bB
R2: A → aB      R4: B → b
```

L2 = $\{a^n b^n \mid n \geq 1\}$

Diese Sprache ist vom **Typ 2** (aber nicht vom Typ 3!). Hier eine kontextfreie Grammatik G11 für L2:

```
R1: S → aSb
R2: S → ab
```

L3 = $\{w\, w^{-1} \mid$ w ist eine nicht-leere Folge von a´s und b´s$\}$

w^{-1} besteht aus den gleichen Symbolen wie w, aber in umgekehrter Reihenfolge, z.B.

w = abb, w^{-1} = bba.
Die Sprache L3 ist vom **Typ 2** (aber nicht vom Typ 3!). Hier eine kontextfreie Grammatik G12 für L3:

```
R1: S → aSa     R3: S → aa
R2: S → bSb     R4: S → bb
```

L4 = {$a^n b^m c^m d^n$ | n≥1, m≥1}
Diese Sprache ist vom **Typ 2** (aber nicht vom Typ 3). Hier eine kontextfreie Grammatik G13 für L3:

```
R1: S → aSd      R3: A → bAc
R2: S → aAd      R4: A → bc
```

L5 = {$a^n b^m c^n d^m$ | n≥1, m≥1}
Diese Sprache ist vom **Typ 1** (nicht vom Typ 2!). Hier eine kontextsensitive Grammatik G14 für L5:

```
R1: S → XY       R5: Y → Bd
R2: X → aXc      R6: cB → Bc
R3: X → ac       R7  aB → ab
R4: Y → BYd      R8: bB → bb
```

L6 = {$a^n b^n c^n$ | n≥1}
Diese Sprache ist vom **Typ 1** (nicht vom Typ 2!). Hier eine kontextsensitive Grammatik G15 für L6:

```
R1: S → aSBc     R3: bB → bb
R2: S → abc      R4: cB → Bc
```

L7 = {P | P ist ein haltendes Java-Programm}
Diese Sprache ist vom **Typ 0** (nicht vom Typ 1!). Es ist möglich, aber ziemlich schwierig, eine Typ 0 Grammatik für L7 anzugeben.

L8 = {P | P ist ein nicht-haltendes Java-Programm}
Diese Sprache kann noch nicht einmal durch eine Typ 0 Grammatik beschrieben werden. Der Beweis dieser Tatsache gehört zu den wichtigsten theoretischen Grundlagen der Informatik.

Zur **Aufgabe-03** auf dem nachfolgendenArbeitsblatt:

Aufgabe-03: Geben Sie eine Typ-1-Grammatik (kontextsensitive Grammatik) an für folgende Sprache:

L6 = {$a^n b^n c^n$ | n≥1}

Diese Aufgabe kann man lösen, indem man 3 Gruppen von Regeln entwickelt

Gruppe 1: Regeln zum Ableiten von *rohen Satzformen*

```
R01: S -> aBc
R02: S -> aSBc
```

Mit diesen beiden Regeln kann man folgende *rohen Satzformen* ableiten:

`aBc, aaBcBc, aaaBcBcBc, aaaaBcBcBcBc, ...`

Jede dieser Satzformen enthält schon "die richtige Anzahl von jedem Buchstaben", aber die Bs sind noch groß (Zwischensymbole) statt klein (Endsymbole) und die meisten Bs stehen noch nicht da, wo sie stehen sollten (deshalb bezeichnen wir diese Satzformen als *roh*).

Gruppe 2: Regeln, mit denen man die Buchstaben *in die richtige Reihenfolge* bringen kann:

```
R03: cB -> Bc
```

In diesem einfachen Fall genügt diese eine Regel. Mit 0 oder mehr Anwendungen von R03 kann man (aus den mit R01 und R02 erzeugten Satzformen) die folgenden Satzformen erzeugen:

`aBc, aaBBcc, aaaBBBccc, aaaaBBBBcccc., ....`

Jetzt stehen alle Buchstaben an den richtigen Stellen, aber die Bs sind noch groß statt klein.

Gruppe 3: Regeln, mit denen man aus Zwischensymbolen (großen Bs) Endsymbole (kleine bs) machen kann, *aber nur, wenn sie am richtigen Platz stehen*:

```
R04: aB -> ab
R05: bB -> bb
```

Mit einer oder mehr Anwendungen dieser Regeln kann man (aus den mit den R01 bis R03 erzeugten Satzformen) folgende Satzformen erzeugen:

`abc, aabbcc, aaabbbccc, aaaabbbbcccc, ...`

Der **Trick** dieser Lösung: Auch ein böswilliger Ableiter kann die Zwischensymbole `B` (mit den Regeln der Gruppe 3) nur dann in Endsymbole `b` ableiten, wenn er vorher mit den Regeln der Gruppe 2 alle Symbole "in die richtige Reihenfolge" gebracht hat.

Zur Entspannung: **Eigenschaften von Qubits** (Quanten-Bits)

1. Mit n "normalen Bits" kann man *eine* Zahl (zwischen 0 und 2^n-1) darstellen. Mit n Qubits kann man gleichzeitig bis zu 2^n Zahlen (zwischen 0 und 2^n-1) darstellen und mit *einer* Operation kann man alle diese Zahlen "gleichzeitig bearbeiten".

Angenommen wir haben einen Speicher S, der aus 20 Qubits besteht. Dann können wir darin etwa 1 Million Zahlen (zwischen 0 und etwa 1 Million) speichern und mit *einer* Operation all diese Zahlen auf einmal verändern. Wenn S aus 30 Qubits besteht, gilt alles entsprechend, aber mit 1 Milliarde Zahlen, und bei 40 Qubits mit 1 Billion Zahlen etc.

2. Wenn man unseren Speicher S "ansieht und ausliest", bekommt man allerdings nur *eine* der Zahlen, die sich im Speicher befinden. Die übrigen Zahlen gehen dabei unvermeidbar und unwiderruflich verloren. Während mit unserem Speicher S gerechnet wird, muss der Speicher deshalb sorgfältig von der Umwelt isoliert werden, denn fast jede Interaktion des Speichers mit der Umwelt zählt als "ansehen und auslesen".

3. Es ist nicht möglich, ein Qubit (mit all seinen Werten) zu *kopieren*. Man kann höchstens *einen* seiner Werte kopieren (und die übrigen Werte gehen dabei verloren).

4. Auf Qubits kann man nur *umkehrbare Verknüpfungen* anwenden.

Zur Zeit (die Daten stammen aus 2008) erforschen mehrere Tausend Physiker, Informatiker und Ingenieure in mehr als 100 Forschungsgruppen etwa ein Dutzend Möglichkeiten, Qbits zu realisieren (durch ion traps, quantum dots, linear optics, ...).

Eine interessante Einführung in die Quantenmechanik von einer berliner Schülerin:

Silvia Arroyo Camejo: "Skurrile Quantenwelt", Springer 2006, Fischer 2007

Arbeitsblatt für den 5. SU am Fr 08.11.13

Aufgabe-01: Geben Sie eine Typ-3-Grammatik (reguläre Grammatik) an für folgende Sprache:
L1 = {$a^n b^m$ | n≥1, m≥1}

Aufgabe-02: Geben Sie eine Typ-2-Grammatik (kontextfreie Grammatik) an für folgende Sprache:
L2 = {$a^n b^n$ | n≥1}

Aufgabe-03: Geben Sie eine Typ-1-Grammatik (kontextsensitive Grammatik) an für folgende Sprache:
L6 = {$a^n b^n c^n$ | n≥1}
Diese Aufgabe lösen wir alle gemeinsam.

Aufgabe-04: Geben Sie eine Typ-1-Grammatik (kontextsensitive Grammatik) an für folgende Sprache:
DOPPELT = {w:w | w ∈ W }
wobei W die Menge aller möglichen a-b-Folgen sein soll:
W = {a, b, aa, ab, ba, bb, aaa, aba, aab, ..., aababbbbaabbbbab, ...}
Beispiele für Worte der Sprache **DOPPELT**:
a:a, b:b, aab:aab, bab:bab, ababab:ababab, aaa:aaa
Gegenbeispiele: aabaab (der Doppelpunkt : fehlt), aab:abb (aab ist ungleich abb)

Tipp: Versuchen Sie, die Aufgabe-04 ganz ähnlich zu lösen wie Aufgabe-03 (mit drei Gruppen von Regeln etc.)

6. SU Fr 15.11.13

Heute **Test 5**

Grammatiken und Sprachen (Fortsetzung und Ende)

Letztes mal haben wir Beispiele für Typ-3-, Typ-2- und Typ-1-Sprachen besprochen (z.B. a^nb^m, a^nb^n, $a^nb^nc^n$), siehe Arbeitsblatt auf S. 19 in diesem Papier.

Beispiel für eine **Typ-0-Sprache**: siehe oben S. 17
Beispiel für eine **nicht-mit-einer-Grammatik-beschreibbare Sprache**: siehe oben S. 17

Das Papier TermeInGentle.pdf besprechen

1. Zwei Typen `FARBE` und `BAUM`

Wie viele Werte gehören zum Typ `FARBE`? (3)
Wie viele Werte gehören zum Typ `BAUM` (unendlich viele)

2. Konvention für Variablen-Namen (`F` ist vom Typ `FARBE`, `B` ist vom Typ `BAUM` etc.)

3. Neun Beispiele für Terme des Typs `BAUM`

4. Grundterme enthalten keine Variablen

5. Spezialfälle von Termen, **Beispiel-01**, **Beispiel-02**

6. Grundspezialfälle von Termen, **Beispiel-03**, **Beispiel-04**

Aufgabe-01: Welche Beispielterme sind Spezialfälle/Grundspezialfälle von `b(F1,F2,B1,leer())`?

Aufgabe-02: *Wie viele* Grunspezialfälle haben die Terme ...

Aufgabe-03: Geben Sie die *Menge* der Grundspezialfälle der Terme ... an.

Aufgabe-04: Geben Sie Terme an, die folgende Mengen von Grundtermen ... beschreiben.

Merke: Die Worte "Grundterm" und "Wert" haben dieselbe Bedeutung

7. Einen Term kann man auf 2 Weisen benutzen: Als *Ausdruck* und als *Muster*

Einen Ausdruck kann man *auswerten* (*Auswertungen* das gibt es auch in C und Java und ...)
Ein Muster kann man *abgleichen* (oder: Man kann damit einen *Musterabgleich* durchführen)

Zur Entspannung: **Der EPR-Effekt**

Albert Einstein mochte die Quantenmechanik nicht. Er dachte sich mehrere Gedankenexperimente aus, die zeigen sollten, dass sie fehlerhaft ist oder zumindest "keine vollständige Beschreibung der Natur" liefert. Nils Bohr vertrat die Quantenmechanik und fühlte sich zuständig für ihre Verteidigung. Mehrmals gelang es ihm, in einem Gedankenexperiment von Einstein einen subtilen Fehler zu entdecken und die Argumente von Einstein dadurch zu entkräften. Bei *einem* der Gedankenexperimente gelang ihm das aber nicht.

Einstein mochte die Quantenmechanik nicht, kannte sie aber offenbar so genau, dass er auch einige ihrer entfernten Konsequenzen erkennen konnte. Den Effekt, der heute als *EPR-Effekt* bezeichnet wird (nach Einstein, seinem Physiker-Kollegen Podolski und einem Studenten Rosen) ist eine Konsequenz der Quantenmechanik, die Einstein "so unsinnig und unglaubhaft" vorkam, dass er sie als *Argument gegen die Quantenmechanik* veröffentlichte. Inzwischen hat man diesen merkwürdigen Effekt experimentell nachgewiesen und benutzt ihn zur *abhörsicheren Übertragung von Daten*.

6. Übung: Aufgabe **6. Vollständige Syntaxprüfung für die Sprache alg20**
Siehe dazu auch Ordner `projects\alg20`

7. SU Fr 22.11.13

Heute **Test 6**

Papier TermeInGentle.pdf, Fortsetzung und Ende

Beispiel-01 (Eine Auswertung), **Beispiel-02** (ein Musterabgleich)

8. Eine Auswertung "klappt immer". Ein Musterabgleich kann *gelingen* oder *misslingen*.

Beispiel-03 (Ein Musterabgleich der misslingt)

Aufgabe-05: Führen Sie Musterabgleiche durch

9. Terme sind Bäume

Token-Prädikate in Gentle

Wenn man mit `phrase`-Prädikaten bestimmte Worte wie `"mother"` oder `"vater"` oder `"class"` oder `"while"` etc. einlesen will, kann man einfach das entsprechende String-Literal hinschreiben.

Wenn man aber "irgendein Ganzzahl-Literal" oder "irgendeinen Bezeichner" etc. einlesen will, sollte man dazu ein entsprechendes *Token-Prädikat* verwenden.

Es gibt (zur Zeit) 3 vordefinerte Token-Prädikate:

```
token IDENT(-> string)    // <<<[A-Za-z][A-Za-z0-9_]*>>>
token INTEGER(-> int)     // <<<[0-9]+>>>
token STRING(-> string)   // <<<\".*\">>>
```

Wenn man das vordefinierte Prädikat `IDENT` in einem Gentle-Programm benutzen will, muss man es (nicht vollständig *definieren* sondern nur) *deklarieren* wie folgt:

```
token IDENT(-> string)
```

Für `INTEGER` und `STRING` gilt natürlich ganz entsprechendes.

Beispiel für vollständige *Definitionen* von `token`-Prädikaten findet man in der Datei `projects\alg30\spec.g`

Achtung: Solche Token-Prädikate "dürfen sich nicht überlappen" (d.h. keine Zeichenfolge in einem Quellprogramm darf zu zwei verschiedenen Token-Prädikaten "passen").

Papier **LexerUndParser.pdf** besprechen.

Zur Entspannung: **Hilberts Hotel**

Denken Sie sich ein Hotel mit unendlich vielen, nummerierten Zimmern: 1, 2, 3, ... Alle Zimmer sind belegt. Dieses Hotel wurde nach dem Mathematiker David Hilbert (1862-1943) benannt.

1. Wie kann man *einen* weiteren Gast unterbringen? (`ZrNr := ZrNr + 1`)
2. Wie kann man *hundert* weitere Gäste unterbringen? (`ZrNr := ZrNr + 100`)
3. Wie kann man *unendlich* viele weitere Gäste unterbringen? (`ZrNr := ZrNr * 2`)

danach sind alle Zimmer mit ungeraden Nrn. (1, 3, 5, ...) frei.

7. Übung: Aufgabe **6. Vollständige Syntaxprüfung für die Sprache alg20**
Siehe dazu auch Ordner `projects\alg20`

8. SU Fr 29.11.13

Heute **Test 7**

Papier **Generatoren.pdf** besprechen.

Schon in den 1960-er-Jahren begann man, Parser (für Compiler und andere Programme) nicht mehr "von Hand" zu schreiben, sondern aus Grammatiken *generieren* zu lassen durch sog. Compiler-compiler, heute: Parsergeneratoren.

yacc (yet another compiler-compiler) erzeugt sehr schnelle Parser, aber nur für bestimmte Grammatiken (LR-Grammatiken). **bison** ist eine Weiterentwicklung des yacc (auch für Windows).

accent (a compiler-compiler for the entire class of contextfree grammars) kann für jede kontextfreie Grammatik einen Parser generieren (der im allgemeinen langsamer ist als ein LR-Parser).

S. 2: Beispiel für eine yacc/accent-Spezifikation eines Parsers

Die von yacc/bison/accent generierten Parser lesen keine Zeichen ein, sondern Symbole (die aus einem oder mehreren Zeichen bestehen können). Das Einlesen von Zeichen und Zusammenfassen zu Symbolen erledigt ein sog. Lexer (oder: Scanner). Auch Lexer werden heute in aller Regel generiert (und nicht "von Hand" geschrieben).

Der Lexer-Generator **lex**, Weiterentwicklung **flex** (auch für Windows).

S. 2: Beispiel für eine lex/flex-Spezifikation eines Lexers (passend zum obigen Parser).

Der Gentle-Compiler verteilt Arbeit wie folgt:

Alle `token`-Prädikate und alle in `phrase`-Prädikaten benutzten String-Literale werden einem Lexer-Generator (**flex**) übergeben.

Alle `phrase`-Prädikate werden einem Parser-Generator übergeben (**accent** oder **yacc**).

Papier JvmUndJasmin.pdf besprechen

Mit vielen höheren Programmiersprachen (angefangen mit Cobol und Fortran) war die Hoffnung verbunden, dass die in diesen Sprachen geschriebenen Programme auf vielen verschiedenen Computern (oder sogar auf allen Computern) ausgeführt werden können. Erst mit Java wurden diese Hoffnungen zum ersten Mal erfüllt.

Ein wichtiger Grund für den Erfolg von Java: Die virtuelle Java Maschine (JVM).

Beispiel für ein Java-Quellprogramm und ein entsprechendes Bytecode-Programm (**S. 3**)

Die Namen von Klassen, Methoden und Attributen sind im Bytecode enthalten

Nur lokale Variablen von Methoden haben im Bytecode Zahlen (0, 1, 2, ...) als Namen.

Java-Assembler Jasmin (nicht von Sun, aber verbreitet)

Beispiel Jasmin-Programm **S. 5** (Wir werden Jasmin als Zielsprache eines Compilers benutzen)

Ein paar Maschinenbefehle / Assemblerbefehle

```
aload_0, aload 7
astore_0, astore 7
bipush, sipush, iconst_3, iconst_m1
i2d, d2i
getstatic, putstatic
getField, putField
if_icmpge label
invokestatic, invokevirtual
```

Das Verzeichnis aller Maschinenbefehle / Assemblerbefehle

```
iadd = 96 60 int add (xy -> (x+y))
```

Die Notation, mit der die Wirkung der Rechenbefehle beschrieben wird, besprechen.

Struktur der JVM

Mit *Speicher* ist im Folgenden der Bereich in einem Stapel-Rahmen gemeint, in dem die Parameter und anderen lokalen Variablen einer Methode "leben".

`load, get`	vom Speicher zum Stapel
`store, put`	vom Stapel zum Speicher
`push`	auf den Stapel
`pop`	vom Stapel weg

Zur Entspannung: **G-vorn oder K-vorn** (big endian or little endian)

Jonathan Swift (1667-1745, Irland, damals unter englischer Herrschaft) veröffentlichte 1726 einen Roman mit dem Titel "Travels Into Several Remote Nations of The World", im Wesentlichen eine Satire gegen die politischen Verhältnisse in Irland und England. Der Roman erschien unter dem Titel "Gullivers Reisen" auch in Deutschland und wurde lange als Kinderbuch (miss-) verstanden.

Damals waren die wichtigsten politischen Parteien die Tories und die Whigs. In seinem Roman beschrieb Swift zwei Parteien ("in einem fernen Land"), die sich nur dadurch unterschieden, auf welcher Seite sie ihre Frühstückseier aufschlugen: Die Big Endians an der stumpferen Seite und die Little Endians an der spitzeren Seite.

Allgemein kann man bei vielen Daten zwei Formen unterscheiden: G-vorn (das Große vorn, big endian) und K-vorn (das Kleine vorn, little endian). Beispiele:

Ein Datum 17.12.2005	K-vorn
Neues Datum 2005.12.17	G-vorn
Ein Pfadname `d:\bsp\java\Hallo.java`	G-vorn
Eine Netzadresse `beuth-hochschule.de`	K-vorn
Eine arabische Zahl im Deutschen 12345	G-vorn
Eine arabische Zahl im Arabischen 12345	K-vorn

Man unterscheidet auch zwischen K-vorn Prozessoren (little endian processors) und G-vorn-Prozessoren (big endian p.), die Zahlen mit den niedrigwertigen (bzw. hochwertigen) Ziffern vorn darstellen. Keine der beiden Formen ist prinzipiell überlegen, beide haben Vor- und Nachteile.

x86-Prozessoren von AMD oder Intel sind	K-vorn (little endian)
m68-Prozessoren von Motorola	G-vorn (big endian)
PowerPCs von IBM sind	G-vorn (big endian)

9. SU 06.12.13

Heute **Test 8**

Papier JvmUndJasmin.pdf besprechen (Fortsetzung)

JVM und klassische Prozessoren wie I386 (Intel) oder M68 (Motorola) oder Sparc (Sun)

Ähnlichkeiten:
- JVM hat einen Befehlssatz von ca. 200 Maschinenbefehlen
- Typische Befehle laden einen Wert auf den Stapel oder addieren zwei Zahlen etc.

Unterschiede:
- JVM nicht nur *Prozessor*, sondern vollständige *Plattform*
- JVM ist *stark getypt*
- JVM ist *objektorientiert* (`new` ist *ein* Maschinenbefehl etc.)
- JVM beruht auf einem *Sicherheitskonzept* (Bytecode Verifier etc.)

Bytecode Verifier: Bevor eine `.class`-Datei in die JVM geladen wird, wird sie genau geprüft. Wenn der Bytecode nicht bestimmten, sehr strikten Regeln entspricht, wird er abgelehnt und nicht geladen.
Beispiel: Wenn eine Methode einen Überlauf ihres Operanden-Stapels verursachen könnte, wird die betreffende `.class`-Datei abgelehnt.

Anmerkung: Ein großer Teil aller heute verbreiteten, gefährlichen Software-Fehler haben mit Schwächen der Sprache C zu tun (ein C-Programm kann einen Stapel-Überlauf provozieren oder z.B. auf eine Reihungskomponente `r[500]` zugreifen, obwohl die Reihung `r` nur mit 10 Komponenten vereinbart wurde). So etwas ist in der JVM grundsätzlich *nicht* möglich.

Speicherbereiche der JVM

Einmal pro JVM:
Eine *Halde* (engl. heap). Da "wohnen" alle Objekte
Einmal pro Faden (engl. thread):
Ein *Programmzähler* (engl. program counter register, pc register)
Ein (Faden-) *Stapel* (engl. JVM stack). Der enthält Stapel-Rahmen (engl. stack frames)

Stapel-Rahmen:
Wenn ein Faden einen Methoden-Aufruf ausführt,
wird ein Rahmen für diese Methode auf den Faden-Stapel gelegt.
Wenn die Methode fertig ausgeführt ist (oder abgebrochen wird),
wird der Rahmen wieder vom Stapel entfernt.

Der Stapel-Rahmen enthält:
alle *Parameter* der Methode
alle (anderen) *lokalen Variablen* der Methode
einen Operanden-*Stapel* (Op-Stapel, engl. operand stack)

Wenn in der Beschreibung eines Jasmin-Befehls ein *Stapel* erwähnt wird, dann ist immer der *Op-Stapel* in einem Stapel-Rahmen gemeint (nicht der *Faden-Stapel*!).

Anmerkung: Da bei der Ausführung eines Java-Programms im allgemeinen mehrere oder *viele* Faden-Stapel gebraucht werden (für jeden Faden einer) und einige davon dynamisch "entstehen und vergehen", werden die einzelnen Faden-Stapel meistens nicht mehr als *Speicherbereiche fester Länge* implementiert, sondern als *verkettete Listen von Stapel-Rahmen*.

Wie lang die einzelnen *Operanden-Stapel* (in den Stapel-Rahmen) sein müssen, ermittelt der Bytecode Verifyer, bevor er die betreffende Klasse "zuläßt".

Zur Entspannung: **Christian Morgenstern** (1871 - 1914):
Der Werwolf "Ein Werwolf eines Nachts entwich, ..."

Sprungbefehle

Unterschiede zwischen *Assemblerbefehlen* und *Maschinenbefehlen*

Eine Java-`main`-Methode:

```
 1    static public void main(String[] sonja) {
 2       if (sonja.length>=1) {
 3          pln("Then-Teil");
 4       } else {
 5          pln("Else-Teil");
 6       }
 7       pln("Vers02: Das war's erstmal!");
 8    } // main
```

Die Übersetzung dieser Methode in Assembler:

```
 1 .method public static main([Ljava/lang/String;)V
 2 .limit stack  2
 3 .limit locals 1
 4 .var 0 is arg0 [Ljava/lang/String; from Label2 to Label3
 5
 6 Label2:
 7 .line 9
 8    aload_0                                              ; 0000|2A
 9    arraylength                                          ; 0001|BE
10    iconst_1                                             ; 0002|04
11    if_icmplt Label0                                     ; 0003|A1 00 0B
12 .line 10
13    ldc "Then-Teil"                                      ; 0006|12 02
14    invokestatic Vers02/pln(Ljava/lang/Object;)V         ; 0008|B8 00 03
15    goto Label1                                          ; 000b|A7 00 08
16 Label0:
17 .line 12
18    ldc "Else-Teil"                                      ; 000e|12 04
19    invokestatic Vers02/pln(Ljava/lang/Object;)V         ; 0010|B8 00 03
20 Label1:
21 .line 14
22    ldc "Vers02: Das war's erstmal!"                     ; 0013|12 05
23    invokestatic Vers02/pln(Ljava/lang/Object;)V         ; 0015|B8 00 03
24 Label3:
25 .line 15
26    return                                               ; 0018|B1
27
28 .end method ; main
```

Zeile 15: Der Assemblerbefehl `goto Label1` springt zum `Label1` in Zeile 20
Der Maschinenbefehl `A7 00 08` springt ("von seinem Anfang aus") 8 Byte nach vorn
(er beginnt bei der Adresse `000b` und er springt zur Adresse `000b + 8` gleich `0013`) .

Aufgabe: Erklären Sie ganz entsprechend den Sprung-Befehl in Zeile 11.

Sprungziele werden dargestellt

in einem *Jasmin-Programm* durch *Label* (für Menschen leicht lesbar)
in einem *Bytecode-Programm* durch *relative Byte-Nummern*
z.B. "8 Byte nach vorn" oder "20 Byte zurück" (für Maschinen leicht ausführbar)

10. SU 13.12.13

Heute **Test 9**

Das Papier JvmUndJasmin.pdf besprechen (Fortsetzung)

Die JVM wurde in *Software* realisiert (für viele Plattformen) und in *Hardware*.

Erläuterungen zur **Notation in Jasmin-Programmen** (**S. 6**)

Besondere Gentle-Befehle: Alternativen-Anweisung (alternatives statement)

Die *Regeln* eines Prädikats sind mit **oder** verbunden
Damit ein Prädikataufruf gelingt, muss die 1. Regel gelingen **oder** die 2. Regel **oder** ...

Die *Prädikataufrufe* (PA) auf der rechten Seite einer Regel sind mit **und** verbunden
Damit die Regel gelingt, muss der 1. PA gelingen **und** der 2. PA **und** ...

Theoretisch genügt das, um alle Programmierprobleme zu lösen. Allerdings kann man mit diesen Konstrukten bestimmte Probleme nur ineffizient lösen. Darum gibt es in Gentle zusätzlich ein Konstrukt, welches if- oder switch-Anweisungen in anderen Sprachen ähnelt: Die Alternativen-Anweisung.

Papier **GentleBeispiele.txt**, Beispiel **alternatives01.g**

Angenommen, `expensive` ist ein Prädikat, dessen Ausführung teuer ist ("viel Zeit kostet").

Das Prädikat `processA` ruft `expensive` auf.

Der Befehl `RES -> 1` ist ein *Musterabgleich*. Er gelingt, wenn der Wert des Variablen `RES` sich abgleichen läßt mit dem Muster `1`. Das ist nur dann der Fall, wenn `RES` den Wert `1` hat. In C oder Java hätte man also `if (RES == 1)` geschrieben.

Wenn `processA` mit bestimmten `int`-Parametern aufgerufen wird, wird `expensive` dreimal (mit demselben in-Parameter) aufgerufen. Das ist ineffizient.

Das Prädikat `processB` macht genau das Gleiche wie `processA`, aber effizienter: `expensive` wird immer nur *einmal* aufgerufen.

Vergleich:
`processA` unterscheidet drei Fälle mit Hilfe von *drei Regeln*.
`processB` unterscheidet die drei Fälle mit einer *Alternativen-Anweisung* in nur *einer* Regel.

Papier **GentleBeispiele.txt**, Beispiel **alternatives02.g**

Dieses Beispiel zeigt, wie ein Gentle-Programmierer sich mit einer Alternativen-Anweisung *Schreibarbeit sparen* kann (hier wird also die Effizienz des Programmierers verbessert, nicht die Effizienz des Ausführers).

Aufgabe: Schreiben Sie ein paar Werte des Typs `Exp` auf (mindestens 3)

```
lit(17), exp(add(), lit(17), lit(4)), exp(mul(), exp(add(), lit(17), lit(4)), lit(3))
```

Die Prädikate `evalA`, `evalB` und `evalC` leisten alle das Gleiche: Sie erwarten einen Wert des Typs `Exp` (einen Ausdruck) als in-Parameter und liefern den *Wert* des Ausdrucks als out-Parameter.

Das Prädikat `evalA` enthält viele "Wiederholungen". Die Prädikate `evalB` und `evalC` vermeiden das mit einer Alternativen-Anweisung (sie unterscheiden sich voneinander nur durch das Layout).

Wichtig: Nur weil die Variable `V` in *allen* (vier) Alternativen definiert wird, kann sie auch als out-Parameter verwendet werden.

Papier **GentleBeispiele.txt**, Beispiel **alternatives03.g**

Alternativen-Anweisungen können natürlich auch geschachtelt werden. Dieses Beispiel zeigt einen Vorschlag, wie man durch ein bestimmtes Layout eine akzeptable Lesbarkeit erreichen kann.

Funktionale und prozedurale Programmiersprachen

Ein *Wertebehälter* ist entweder eine *Variable*, deren Wert man beliebig oft verändern kann, oder ein *E/A-Gerät* wie z.B. ein Bildschirm, eine Tastatur, ein Drucker, eine Datei auf einer Festplatte etc.

In vielen Sprachen (z.B. in C und in Java) ist es sinnvoll, zwei Arten von Unterprogrammen (Methoden, Subroutines etc.) zu unterscheiden:
Funktionen dienen dazu, einen Wert zu berechnen (und als Ergebnis zu liefern). Solche Funktionen sollten normalerweise keine Wertebehälter verändern (es gibt aber Ausnahmen).
Prozeduren dienen dazu, Wertebehälter zu verändern (und liefern kein Ergebnis)

C-Programmierer haben eigene Sprachgewohnheiten:

Normale Bezeichnung	Unterprogramm	Funktion	Prozedur
C-speak	Funktion	non-void-Funktion	void-Funktion

Eine **funktionale Sprache** ist eine Sprache, in der alle Unterprogramme *Funktionen* sind. Solche Sprachen betonen sehr stark unveränderbare Variablen und behandeln Wertebehälter als Ausnahmen.

Eine **prozedurale Sprache** ist eine Sprache, in der es (außer Funktionen auch) *Prozeduren* gibt. Solche Sprachen betonen Wertebehälter (deren Inhalt man beliebig oft verändern kann) und behandeln unveränderbare Variablen als Ausnahmen.

In Gentle kann man fast alle Probleme mit "normalen, unveränderbaren Variablen" lösen. Für die seltenen Fälle, in denen unveränderbare Variablen "sehr unbequem" wären, gibt es **QUP**s (Query Update Pairs). Papier **GentleBeispiele.txt**, Beispiel **qups02.g**

Zur Entspannung: **Alan Mathison Turing** (1912-1954), einer der Begründer der Informatik

Bevor man die ersten elektronischen Computer baute, konzipierte und untersuchte der Mathematiker Turing eine Rechenmaschine, die so einfach war, dass niemand an ihrer prinzipiellen Realisierbarkeit zweifelte. Eine solche Turing-Maschine besteht aus einem unbegrenzt langen Band, welches in kleine Abschnitte eingeteilt ist, von denen jeder genau ein Zeichen eines endlichen Alphabets aufnehmen kann. Ein Schreib-Lese-Kopf über dem Band kann bei jedem Arbeitsschritt der Maschine das Zeichen auf dem aktuellen Abschnitt lesen und in Abhängigkeit davon ein bestimmtes Zeichen auf den aktuellen Abschnitt schreiben und einen Abschnitt nach links oder rechts weiterrücken. Ein Programm für eine solche Maschine besteht aus einer endlichen Menge von Befehlen der folgenden Form:

"Wenn das aktuelle Zeichen gleich X ist, dann schreibe Y und gehe einen Abschnitt nach links bzw. nach rechts bzw. bleib wo du bist" (wobei X und Y beliebige Zeichen des Alphabets sind).

Wichtige Erkenntnis 1: Es gibt viele (präzise definierte, mathematische) Probleme, die man mit Hilfe einer solchen Turing-Maschine lösen kann (z. B. das Multiplizieren von dreidimensionalen Matrizen).

Wichtige Erkenntnis 2: Es gibt aber auch (präzise definierte, mathematische) Probleme, die man *nicht* mit Hilfe einer solchen Turing-Maschine lösen kann.

Wichtige Vermutung 3: Alle Probleme, die man mit heutigen oder zukünftigen Computern lösen kann, kann man im Prinzip auch mit einer Turing-Maschine lösen.

Im zweiten Weltkrieg arbeitete Turing für die *Government Code and Cypher School* in Bletchley Park (d. h. für den britischen Geheimdienst) und half entscheidend dabei, die Maschine zu durchschauen, mit der die deutsche Marine ihre Funksprüche verschlüsselte (die Enigma), und wichtige Funksprüche zu entschlüsseln. Damit hat er vermutlich einer ganzen Reihe von aliierten Soldaten (Engländern, Amerikanern, Franzosen, Russen) das Leben gerettet.

Weil er homosexuell war, wurde Turing nach dem Krieg zu einer Hormonbehandlung "seiner Krankheit" gezwungen, bekam schwere Depressionen und nahm sich das Leben. Inzwischen wurden die entsprechenden Gesetze in England (und ähnliche Gesetze in anderen Ländern) beseitigt. 2009 entschuldigte sich der britische Premierminister Gordon Brown dafür, wie Turing behandelt worden ist.

11. SU 20.12.13

Heute **Test 10**

Query update pairs

Papier GentleBeispiele.pdf, Beispiel qups02.g

Aufgabe-01: Welche beiden Prädikate werden durch die folgende Vereinbarung definiert:

```
data Word(-> string)
```

Anmerkung: Anfangs stand in der vorigen Zeile `data Word(String)`, ohne Pfeil `->`.
Das war falsch. Jetzt ist die Zeile richtig (mit Pfeil `->`) . **Regel:** In den runden Klammern eines `data`-Befehls muss das Gleiche stehen wie in den runden Klammern eines `Get`-Prädikats.
Die Beispiele im Gentel-Programm `qups02.g` waren schon immer vom Gentle-Compiler geprüft und richtig.

Aufgabe-02: Welche beiden Prädikate werden durch die folgende Vereinbarung definiert:

```
data int2string(int -> string)
```

Optionstypen

Häufig muss ein bestimmter Wert nicht unbedingt "vorhanden sein", sondern kann auch fehlen. Für solche Fällen könnte man bei der Definition des betreffenden Typs einen Konstruktor wie `empty()` oder `leer()` oder `nichtDa()` vorsehen, müsste sich aber jedesmal einen neuen Namen ausdenken (da kein Konstruktor zu zwei verschiedenen Typen gehören darf).

Zur Lösung dieses kleinen, aber lästigen Problems gibt es in Gentle *Option Types*. Sie legen eine einheitliche Syntax für optionale Werte fest.

Papier GentleBeispiele.txt, Beispiel optionTypes01.g

Aufgabe-03: Gegeben sei der folgende Gentle-Typ:

```
type Color c() m() y()
```

Geben Sie alle Werte des Optionstyp `Color?` an

Zur Entspannung: **Christian Morgenstern** (1871 - 1914)

Das Butterbrotpapier

Über einen Seitenarm der Evolution, der wegen eines unglücklichen Zufalls leider nur sehr kurzlebig war.

Lösung-01:

```
Set-Word(string)
Get-Word(-> string)
```

Lösung-02:

```
Set-int2string(int, string)
Get-int2string(int -> string)
```

Lösung-03:

`Color?(c()), Color?(m()), Color?(y()), Color?().`

12. SU Fr 10.01.14

Heute **Test 11**

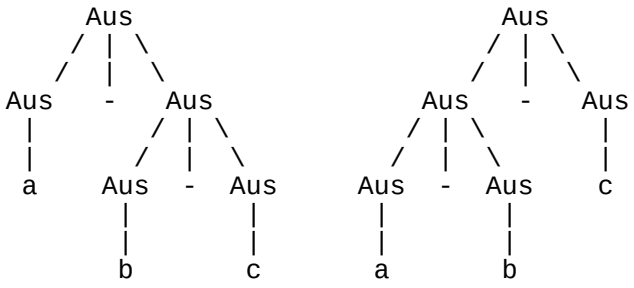
Ein Problem bei (Typ-2-) Grammatiken: Mehrdeutigkeit

Eine kleine Beispiel-Grammatik G15 (1 Zwischensymbol, 4 Regeln):

```
1   Aus -> Aus "-" Aus
2   Aus -> a
3   Aus -> b
4   Aus -> c
```

Kontruieren Sie mit dieser Grammatik einen Syntaxbaum für den Ausdruck `a - b - c`

Problem: Für diesen Ausdruck gibt es *zwei* verschiedene Syntaxbäume:



Syntaxbaum 1 **Syntaxbaum 2**

Sind beide Bäume "gleich gut"? Oder ist einer besser als der andere? Warum?

Der Syntaxbaum 1 entspricht der Klammerung: `a - (b - c)`
Der Syntaxbaum 2 entspricht der Klammerung: `(a - b) - c`

Der Syntaxbaum 2 entspricht der weit verbreiteten Konvention, dass
der Minus-Operator `'-'` *linksassoziativ* ist
(d.h. ein Operand wie `b`, der zwischen zwei Minuszeichen steht, "assoziiert sich nach links")

Aus dieser Konvention folgt auch, dass z.B.

`8 - 4 - 2` gleich `2` ist (und nicht gleich `6`)

Für längere Ausdrücke wie z.B. `a - b - c - c - a - b`
kann man aus der Grammatik G15 nicht nur *zwei*, sondern *ziemlich viele* Syntaxbäume konstruieren.

Def: Eine Grammatik ist *mehrdeutig*, wenn man daraus für mindestens 1 Wort
mehrere Syntaxbäume konstruieren kann

Eine Grammatik, die man zum Bau eines Compilers verwendet, sollte jedem Wort der beschriebenen Sprache nur *einen* Syntaxbaum ("den richtigen") zuordnen. Mehrdeutige Grammatiken sind ein Problem.

Anmerkung 1: Es ist (beweisbar) unmöglich, ein Programm zu schreiben, welches von jeder beliebigen kontextfreien Grammatik korrekt herausfinden kann, ob sie mehrdeutig ist oder nicht.

Das drückt man auch so aus: Die Eigenschaft der *Mehrdeutigkeit* bei kontextfreien Grammatiken ist *nicht entscheidbar* (d.h. nicht per Computerprogramm feststellbar).

Anmerkung 2: Das Programm `amber` liest eine Grammatik ein, leitet systematisch Worte daraus ab und prüft, ob es für dieses Wort mehrere Syntaxbäume gibt.

Praktisch kann `amber` von sehr vielen Grammatiken feststellen, dass sie mehrdeutig sind. Aber wenn `amber` (z.B. nach 10 Stunden Rechenzeit) kein Wort mit mehreren Syntaxbäumen findet, ist nicht garantiert, dass es kein solches Wort gibt.

Die **Assoziativität** eines binären Operators wie z.B. `-` legt fest,
ob `A - B - C`
gleich `((A - B) - C )` oder
gleich `( A - (B - C))` sein soll.

Im ersten Fall ist der Operator linksassoziativ (das B gehört nach links),
im zweiten Fall ist der Operator rechtsassoziativ (das B gehört nach rechts).

Die **Bindungsstärke** von zwei binären Operatoren wie z.B. `+` und `*` legt fest,
ob `A + B * C`
gleich `((A + B) * C )` oder
gleich `( A + (B * C))` sein soll.

Im ersten Fall bindet `+` stärker als `*`,
im zweiten Fall bindet `*` stärker als `+`.

Wiederholung: Was ist eine Satzform?

Grammatiken für Ausdrücke

Wir beginnen mit einem (hoffentlich abschreckenden :-) schlechten Beispiel:

Beispiel-01: Eine besonders schlechte Grammatik **GAus0** für Ausdrücke
(siehe nachfolgendes Arbeitsblatt)

Was ist schlecht an dieser Grammatik?

Sie ist (in hohem Maße) *mehrdeutig*.
Sie sagt nichts über die *Assoziativität* und die *Bindungsstärke* der Operatoren aus.

Beispiel-02: Eine Grammatik **GAus1** für Ausdrücke (siehe nachfolgendes Arbeitsblatt)

Die Aufgaben auf dem Arbeitsblatt sollen dabei helfen, "ein Gefühl für die Grammatik GAus1" zu entwickeln und zu verstehen, warum sie besser ist als GAus0.

Arbeitsblatt für den 12. SU am Fr 10.01.2014

Zwei Grammatiken für Ausdrücke

Grammatik GrAus0:

```
R01: Aus -> Aus +  Aus     -- Addieren
R02: Aus -> Aus –  Aus     -- Subtrahieren
R03: Aus -> Aus *  Aus     -- Multiplizieren
R04: Aus -> Aus /  Aus     -- Dividieren
R05: Aus -> Aus %  Aus     -- Modulo (Rest nach einer Ganzzahldivision)
R05: Aus -> Aus ** Aus     -- Potenzieren
R06: Aus -> + Aus          -- Vorzeichen +
R07: Aus -> - Aus          -- Vorzeichen -
r08: Aus -> ( Aus )        -- Klammern um einen Ausdruck Aus
R09: Aus -> Literal        -- Literale wie 123 oder 0
R10: Aus -> Bezeichner     -- Bezeichner wie Summe oder x17
```

Grammatik GrAus1:

```
R01: Aus1 -> Aus1 + Aus2   -- Der Operator + ist linksassoziativ
R02: Aus1 -> Aus1 – Aus2   -- Der Operator - ist linksassoziativ
R03: Aus1 -> Aus2

R04: Aus2 -> Aus2 * Aus3   -- Der Operator * ist linksassoziativ
R05: Aus2 -> Aus2 / Aus3   -- Der Operator / ist linksassoziativ
R06: Aus2 -> Aus2 % Aus3   -- Der Operator % ist linksassoziativ
R07: Aus2 -> Aus3

R08: Aus3 -> Aus4 ** Aus3  -- Der Operator ** ist rechtsassoziativ
R09: Aus3 -> Aus4

R10: Aus4 -> + Aus5        -- Alternative: Aus4 -> + Aus4
R11: Aus4 -> - Aus5        -- Alternative: Aus4 -> - Aus4
R12: Aus4 -> Aus5

R13: Aus5 -> ( Aus1 )
R14: Aus5 -> Literal
R15: Aus5 -> Bezeichner
```

Die Regeln für die Zwischensymbole `Literal` und `Bezeichner` sind hier unwichtig und deshalb weggelassen.

Die Grammatiken `GrAus0` und `GrAus1` beschreiben dieselbe Sprache, unterscheiden sich aber trotzdem erheblich.

Die folgenden Aufgaben beziehen sich alle auf die Grammatik **GrAus1**.

Aufgabe-01: Was für Satzformen kann man
allein mit den Regeln R01-R03 aus dem Zwischensymbol `Aus1` ableiten?

Aufgabe-02: Wie sieht der Syntaxbaum für die Satzform `Aus2 - Aus2 - Aus2` aus?
Was sagt dieseser Syntaxbaum über die Assoziativität des zweistelligen Minus-Operators `-` ?

Aufgabe-03: Was für Satzformen kann man
allein mit den Regeln R04-R07 aus dem Zwischensymbol `Aus2` ableiten?

Aufgabe-04: Wie sieht der Syntaxbaum für die Satzform `Aus3 / Aus3 / Aus3` aus?
Was sagt dieseser Syntaxbaum über die Assoziativität des (zweistelligen) Divisions-Operators `/` ?

Aufgabe-05: Was für Satzformen kann man
allein mit den Regeln R08-R09 aus dem Zwischensymbol `Aus3` ableiten?

Aufgabe-06: Wie sieht der Syntaxbaum für die Satzform `Aus4 ** Aus4 ** Aus4` aus?
Was sagt dieseser Syntaxbaum über die Assoziativität des (zweistelligen) Potenzierungs-Operators `**` ?

Lösungen zum Arbeitsblatt für den 12. SU am Fr 10.01.2014

Lösung-01:

```
Aus2
Aus2+Aus2,
Aus2+Aus2+Aus2
...
Aus2-Aus2+Aus2-Aus2-Aus2+Aus2+Aus2-Aus2
...
```

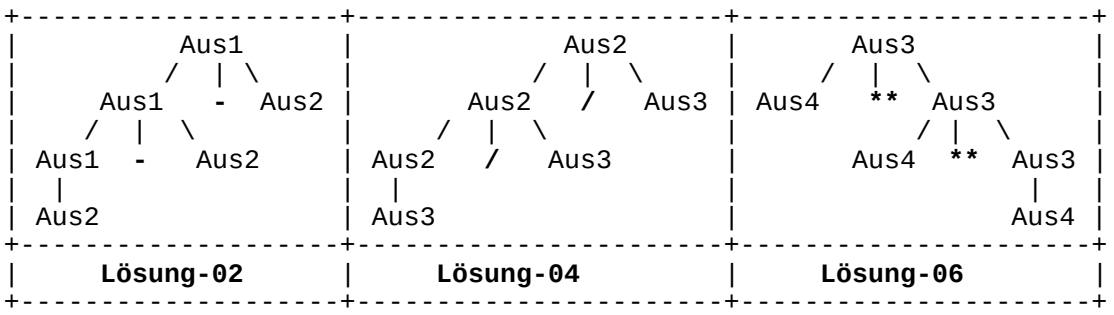
Lösung-03:

```
Aus3
Aus3*Aus3
Aus3*Aus3*Aus3
...
Aus3/Aus3*Aus3/Aus3/Aus3*Aus3*Aus3/Aus3
...
```

Lösung-05:

```
Aus4
Aus4**Aus4
Aus4**Aus4**Aus4
...
Aus4**Aus4**Aus4**Aus4**Aus4**Aus4**Aus4**Aus4
...
```

Lösung-02, -04, -06:



Zur Entspannung: Kostenlose online-Kurse von Spitzenunis (**MOOCs**: massive open online courses)

Seit Anfang 2012 (oder etwas früher) haben verschiedene Unis, vor allem amerikansiche Spitzenunis wie Stanford, Harvard, MIT, Yale, .., damit angefangen, zahlreiche Vorlesungen (tausende) aus vielen Fachgebieten im Internet zu veröffentlichen. Typischerweise gibt es zu jeder solchen Vorlesung umfangreiche Unterlagen: Videos mit den eigentlichen Vorlesungen, dazu .pdf-Dateien mit Folien, Aufgaben, Quizzes (d.h. multiple choice tests, die automatisch korrigiert und bewertet werden) etc. Viele dieser Kurse wurden bereits von vielen hundert TeilnehmerInnen gleichzeitig absolviert, die populärsten mit 200 Tausend TeilnehmerInnen oder noch mehr. Die meisten Kurse sind auf Englisch, einige auch auf Deutsch. Hier ein paar Adressen, unter denen man solche Kurse und weitere Informationen findet:

https://www.coursera.org/ (Stanford, Caltech, University of London, ...)
https://www.edx.org/ (Harvard, MIT, ...)
http://oyc.yale.edu/ (Yale)
https://www.canvas.net/
http://www.udacity.com/
http://mooc13.wordpress.com/geschichte-und-beispiele/deutschsprachige-moocs/

13. SU 17.01.14

Heute **Test 12**

Aufgabe 7.2 bearbeiten (eine Grammatik für Ausdrücke entwickeln, die man in den alg-Compiler einbauen kann).

Den Compiler `alg30` fertig machen und dann `alg31` entwickeln.

14. SU 24.01.14

Heute **Test 13**

Die erste Aufgabe in der Klausur (am Fr 07.02.2014, ab 8 Uhr im Raim DE37)

Geben Sie eine (kontextfreie, Typ-2-) Grammatik an für die Menge aller natürlichen Zahlen (0, 1, 2, ...) im **m**-er-System, die (ohne Rest) durch **n** teilbar sind. Startsymbol **RK0** (wie "Restklasse 0").

Statt **m** und **n** stehen in der Klausur dann zwei positive Ganzzahlen, die größer als 1 sind, z.B. m = 5 und n = 2 ("aller 5-er-Zahlen die durch 2 teilbar sind").

Heute besprechen wir ein Methode, wie man Aufgaben dieser Art lösen kann und nehmen dabei m = 5 und n = 2 als Beispiel:

Da n = 2 ist, benutzen wir die 2 Zwischensymbole RK0, RK1

Aus **RK0** soll man alle (nicht-negativen) 5-er-Zahlen ableiten können, bei denen der **Rest 0** übrig bleibt, wenn man sie durch 5 teilt.

Aus **RK1** soll man alle (nicht-negativen) 5-er-Zahlen ableiten können, bei denen der **Rest 1** übrig bleibt, wenn man sie durch 5 teilt.

Frage: Wie viele und welche Ziffern gibt es im 5-er-System? (5 Ziffern: 0, 1, 2, 3, 4)

1. Gruppe von Regeln: Einziffrige 5-er-Zahlen

```
R01: RK0 -> 0
R02: RK1 -> 1
R03: RK0 -> 2
R04: RK1 -> 3
R05: RK0 -> 4
```

2. Gruppe von Regeln: Mehrziffrige 5-er-Zahlen
Zuerst schreiben wir ganz mechanisch nur die *rechten Seiten* der Regeln in einer besonders systematischen Reihenfolge hin: Jedes Zwischensymbol gefolgt von jeder Ziffer:

```
R05:     -> RK0 0
R06:     -> RK0 1
R07:     -> RK0 2
R08:     -> RK0 3
r09:     -> RK0 4

R10:     -> RK1 0
R11:     -> RK1 1
R12:     -> RK1 2
R13:     -> RK1 3
R14:     -> RK1 4
```

Dann rechnen wir uns für jede Regel die richtige *linke Seite* aus:

```
R05: RK0 -> RK0 0
R06: RK1 -> RK0 1
R07: RK0 -> RK0 2
R08: RK1 -> RK0 3
r09: RK0 -> RK0 4

R10: RK1 -> RK1 0
R11: RK0 -> RK1 1
R12: RK1 -> RK1 2
R13: RK0 -> RK1 3
R14: RK1 -> RK1 4
```

Aufgabe: Lösen Sie die Aufgabe für m=2 und n=5 (aus dem Startsymbol Ihrer Grammatik soll man alle 2-er-Zahlen ableiten können, die ohne Rest durch 5 teilbar sind).

Zur Entspannung: **Al-Khwarizmi** (ca. 780-850)

Abu Ja'far Muhammad ibn Musa Al-Khwarizmi war ein islamischer Mathematiker, der in Bagdad lebte, über Indisch-Arabische Zahlen schrieb und zu den ersten gehörte, die die Ziffer 0 benutzten. Aus seinem Namen entstand (durch Übertragung ins Latein und dann in andere Sprachen) das Wort **Algorithmus**. Eine seiner Abhandlungen heißt *Hisab al-jabr w'al-muqabala* (auf Deutsch etwa: "Über das Hinüberbringen", von einer Seite einer Gleichung auf die andere). Daraus entstand unser Wort **Algebra**.

15. SU 31.01.14

Heute **Test 14** (der letzte Test in diesem Semester)

Anmerkung: Da von den Tests 11 und 12 nur der jeweils "bessere" gilt, haben wir insgesamt nur 13 gültige Tests geschrieben (von denen man 10 bestanden haben muss).

Von T-Diagrammen zu TI-Diagrammen

Ein **Compiler** hat 3 charakteristische Sprachen (Quell-, Ziel, Implementierungs-Sprache)

Wie viele und welche charakteristischen Sprachen hat ein **Interpreter**?
(2,
die *Ausführungsprache*, die der Interpreter ausführen kann, und
die *Implementierungssprache*, in der er geschrieben ist)

Einen Interpreter kann man durch ein I-Diagramm darstellen

Aufgabe-01: Wenn man folgendes hat:
1. einen Interpreter `I1` mit Ausführungssprache A und Implementierungssprache B und
2. einen Compiler `C1` mit Implementierungssprache A und
3. eine B-Maschine

was kann man dann machen?

Lösung-02: Man kann den Compiler `C1` und den Interpreter `I1` kombinieren zu einem Compiler `C2` mit Implementierungssprache A.

Aufgabe-02: Wenn man folgendes hat:

1. einen Interpreter `I1` mit Ausführungssprache A und Implementierungssprache B und
2. einen Interpreter `I2` mit Ausführungssprache B und Implementierungssprache C und
3. eine C-Mascheine

was kann man dann machen?

Lösung-02: Man kann `I1` und `I2` kombinieren zu einem Interpreter mit Ausführungssprache A und Implementierungssprache C.

Aufgabe-03: Wenn man folgendes hat

1. einen Interpreter `I1` mit Ausführungssprache A und Implementierungssprache B und
2. einen Compiler `C1` mit Quellsprache B, Zielsprache C und Implementierungssprache D und
3. eine D-Maschine

was kann man dann machen?

Lösung-03: Man kann `I1` mit `C1` in einen Interpreter `I2` übersetzen, der die Ausführungssprache A und die Implementierungssprache C hat.

Vorbesprechung der Klausur (am 07.02.14 ab 8 Uhr im Raum DE37)

1. Die Klausur besteht aus 4 Aufgaben.

2. Sie haben 90 Minuten Zeit zum Lösen der Aufgaben.

3. Für die Aufgaben gibt es 20, 20, 40 und 20 Punkte, insgesamt also 100 Punkte.

4. Die 1. Aufgabe haben wir ja schon im vorigen SU besprochen.

5. Als Lösung für die 3. Aufgabe (die mit 40 Punkten) sollen Sie (in der Sprache Gentle) 3 Prädikate programmieren.

Dabei ist es erlaubt, zusätzliche Hilfsprädikate zu vereinbaren (wenn Ihnen das sinnvoll erscheint).

Voraussetzen dürfen Sie nur die (wenigen) in Gentle vorderfinierten Prädikate (Equal, Unequal, ...)

6. Ansonsten sollten Sie natürlich den gesamten Stoff beherrschen, den wir in diesem Semester behandelt haben, insonbesondere *kontextfreie Grammatiken* und *TI-Diagramme.*

7. Sie dürfen mit *Bleistift* zeichnen und schreiben.

8. Bringen Sie DIN A 4 Papier für Ihre Lösungen mit (möglichst kariertes, gut radierbares Papier, kein Umweltpapier). Lösungsblätter mit stark *ausgefranstem Rand* kosten 10 Minuspunkte.

9. Schreiben Sie jede Ihrer Lösungen auf die *Vorderseite eines neuen Blattes* und lassen Sie die *Rückseiten* Ihrer Lösungsblätter grundsätzlich *leer*.

Kleine Wiederholung einiger Fachbegriffe:

Operatoren können 1-stellig sein, oder 2-stellig oder 3-stellig oder ...

Beispiele für 1-stellige Operatoren? Für 2-stellige Operatoren? Für 3-stellige Operatoren?

1-stellige Operatoren: `+  -  ++  --  !  ...`

2-stellige Operatoren: `+  -  *  /  %  ...`

3-stellige Operatoren: `? :` (der Ausdrucks-if-Operator)

Welche Operatoren können **infix** notiert werden? (2-stellige Operatoren)

Wie kann man 1-stellige-Operatoren notieren? (**präfix** oder **postfix**, d.h *vor* ihrem Operanden oder *nach* ihrem Operanden)

3 und mehrstellige Operatoren werden im Allgemeinen **mixfix** notiert (d.h. Operand 1, ein Teil des Operators, Operand 2, ein Teil des Operators, Operand3, ...)

10. **Rückgabe** der Klausur: **Fr 14.02.2014, 10 Uhr im SWE-Labor (Raum DE16a)**