Stichworte und Notizen

Die Lehrveranstaltung **Wahlpflichtfach Compilerbau (TB5-CPB)**
im Studiengang **Technische Informatik Bachelor**
im **WS11/12** an der Beuth-Hochschule

Im Laufe des WS11/12 können die TeilnehmerInnen **in dieser Datei** nach jeder Vorlesung ein paar Stichworte zum Inhalt der Vorlesung finden.

Übersicht über Termine und Tests

Die Lehrveranstaltung TB5-CPB besteht aus
einem Block **seminaristischem Unterricht** (freitags 1. Block, 8.00 - 9.30 Uhr, Raum D211)
einem Block **Übung** (freitags 2. Block, 10.00 - 11.30 Uhr, Raum D E 16b)

Insgesamt sind die folgenden Termine geplant (alle an einem Freitag Ende 2011 / Anfang 2012):

30.09.	04.11. Test05	09.12. Test10	13.01. Test14
07.10. Test01	11.11. Test06	16.12. Test11	20.01. Test15
14.10. Test02	18.11. Test07	23.12. Test12	27.01.
21.10. Test03	25.11. Test08		03.02.
28.10. Test04	02.12. Test09	06.01. Test13	10.02.

Die Klausur wird am **Fr 03.02. 2012** ab **8 Uhr** im **Raum D211** geschrieben und spätestens am **Fr 10.02.2012** zurückgegeben.

Am Anfang von 15 SUs (um 8 Uhr im Raum D211) wird ein kleiner Test (ca. 10 Minuten) geschrieben. Bei jedem Test kann man 10 Punkte bekommen. Mit 5 oder mehr Punkten hat man den Test bestanden, mit 4 oder weniger Punkten hat man ihn nicht bestanden. Wenn Sie an einem Test nicht teilnehmen, spielt es ***keine Rolle***, aus welchem Grund (keine Lust, Krankheit, S-Bahn eingefroren, ...).

Wie bekommt man eine Note für dieses Fach?

Durch eine Klausur am Ende des Semesters.
Um an der Klausur (oder an der Nachklausur) teilzunehmen, müssen Sie vorher zwei Bedingungen erfüllen:

1. Mindestens 12 Tests bestehen.
2. Alle in den Übungen gestellten Aufgaben (vermutlich 12 Stück) spätestens bis Fr 27.01.2012, 11.30 Uhr fertig stellen und vorführen.

1. SU Fr 08.04.11

27 Studenten des Studiengangs TB und 2 des Studiengangs MB haben am Anfang des WS11/12 das Wahlpflichtfach Compilerbau (online) belegt. Erschienen sind etwa 20.

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.

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.

Übersicht über Termine, Tests, Aufgaben und Noten im WS11/12

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 Zeichen oder eine (endliche, nicht-leere) Zeichenfolge.
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).***

Anmerkungen: 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.

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:

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 ...}
S03 soll die Sprache aller 0-1-Folgen sein. Zwischen S02 und S03 gibt es einen subtilen Unterschied. 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 ebenfalls eine formale Sprache. 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").

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 nur "unwichtige Verzierungen" der Regeln, die man auch weglassen kann.

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` entweder 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 den acht Regeln der Grammatik G04 kommen die folgenden vier Zwischensymbole vor:
`{Zahl Vorzeich ZiffFo Ziff}`. Sie bilden das ***Alphabet der Zwischensymbole*** von G04.

Außerdem kommen in den Regeln von G04 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. 60 Millionen [Euro pro Jahr].
An der Beuth Hochschule studieren ca. 10 Tausend StudentInnen.
Also kostet das Studieren an der Beuth Hochschule ca. 6000 [Euro pro StudentIn und Jahr].
Ein Bachelor-Studium (6 Semester, 3 Jahre) kostet also ca. 18 Tausend [Euros pro StudentIn].

Aufgabe-02: Geben Sie eine Grammatik G05 an, die die Sprache S05 aller 0-1-Folgen beschreibt:
S03: {`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-03: 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-04: 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-05: 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-06: Geben Sie eine Grammatik G09 an, die die folgende Sprache S09 beschreibt:
S09: {`mutter vater grossmutter grossvater urgrossmutter urgrossvater` ... `ururururgrossmutter` ... `urururururgrossvater` ...}
Verwenden Sie `ahne` als Startsymbol von G09.

Lösung-02:

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

Lösung-03:

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-04:

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

Lösung-05:

```
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-06:

```
 1   ahne : ahne1
 2   ahne : ahne2
 3   ahne : ahne3

 4   ahne1: "mutter"
 5   ahne1: "vater"

 6   ahne2: "gross" ahne1

 7   ahne3: "ur" ahne2
 8   ahne3: "ur" ahne3
```

2. SU Fr 07.10.11

Test 1

Verschiedene Arten von Grammatiken: Chomsky-Grammatiken

Def.: Eine Chomsky-Grammatik G=(Z, E, S, P) besteht aus
1. einer Menge Z von **Zwischen-Symbolen** (non terminal symbols)
2. einer Menge E von **End-Symbolen** (terminal symbols)
3. einem **Startsymbol** S (einem Element der Menge Z) und
4. einer Menge P von **Produktions-Regeln**, P={R_1, R_2, ... , R_n}

Def.: Eine **Satzform** ist eine Folge von Zwischen- und/oder End-Symbolen (sie kann also nur Zwischen-Symbolen, nur End-Symbole oder ein Mischung aus beiden enthalten). Die ***leere Satzform*** (die 0 Symbole enthält) wird häufig mit dem griechischen Buchstaben ε ("epsilon") bezeichnet.

Für eine Chomski-Grammatik muß gelten:

1. Die Mengen Z und E müssen beide ***nicht-leer*** und ***endlich*** sein und dürfen ***keine gemeinsamen Elemente*** besitzen, d.h. es muß Z ∩ E = {} gelten.
2. Jede Regel R besteht aus einer ***linken Seite*** LS, einem ***Trennzeichen***, z.B. "→" und einer ***rechten Seite*** RS, etwa so:
R: LS → RS
3. LS ist irgendeine Satzform, die mindestens ein Zwischensymbol enthält. RS ist irgendeine Satzform.

Ende der Definition von Chomsky-Grammatiken.

Vereinbarung: Um Grammatiken möglichst kurz beschreiben zu können sei vereinbart:

1. **Große Buchstaben** (A, B, ..., Z) sind **Zwischen-Symbole**.
2. **Kleine Buchstaben** (a, b, ... , z) und **Sonderzeichen** (z.B. + - * / . , etc.) sind **End-Symbole**.
3. Das erste Zwischen-Symbol auf der linken Seite der ersten Regel einer Grammatik ist das **Startsymbol** der Grammatik.

Mit dieser Vereinbarung genügt zur Beschreibung einer Grammatik die Angabe ihrer Regeln. Nur wenn von dieser Vereinbarung abgewichen wird, werden in den folgenden Beispielen die End-Symbole, die Zwischen-Symbole und das Startsymbol einer Grammatik explizit angegeben.

Spezielle Grammatikformen

Je strengeren Einschränkungen man die Regeln einer Chomsky-Grammatik unterwirft, desto ***weniger Sprachen*** kann man damit noch beschreiben, aber desto leichter ist es, ***allgemeine Eigenschaften*** solcher Grammatiken und Sprachen zu beweisen. Deshalb hat man sich intensiv mit **eingeschränkten Formen von Chomsky-Grammatiken** befasst, vor allem mit den folgenden 4 Formen (Typ 3 bis Typ 0).

Typ 3 Grammatiken (lineare, reguläre Grammatiken)

Definition: Eine Regel R : LS → RS heißt **abschliessend**, wenn LS (nur) ein Zwischen-Symbol ist und RS kein Zwischen-Symbol enthält (d.h. RS besteht nur aus End-Symbolen oder ist gleich ε).

Beispiele:
R1: A → bcd
R2: A → a
R3: A → ε
Gegenbeispiele:
R4: A → Bcd -- RS enthält ein Zwischen-Symbol
R5: AB → cd -- LS besteht aus mehr als einem Symbol
R6: Aa → bc -- LS besteht aus mehr als einem Symbol

Def.: Eine Regel R : LS → RS heißt **links-linear**, wenn LS (nur) ein Zwischen-Symbol ist und RS (außer End-Symbolen) genau ***ein*** Zwischen-Symbol enthält und dieses am Anfang von RS (ganz **links**) steht.

Beispiele:

R1: A → Bcd
R2: A → B
R3: A → Abc

Gegenbeispiele:

R4: AB → C -- LS besteht aus mehr als einem Symbol
R5: Ab → C -- LS besteht aus mehr als einem Symbol
R6 A → BcdE -- RS enthält mehr als ein Zwischen-Symbol
R7: A → bcDe -- das Zwischen-Symbol D steht nicht am Anfang von RS

Def.: Eine Regel R : LS → RS heißt **rechts-linear**, wenn LS (nur) ein Zwischen-Symbol ist und RS (außer End-Symbolen) genau ***ein*** Zwischen-Symbol enthält und dieses am Ende von RS (ganz **rechts**) steht.

Beispiele:

R1: A → bcD
R2: A → B
R3: A → abC

Gegenbeispiele:

R4: AB → C -- LS besteht aus mehr als einem Symbol
R5: Ab → C -- LS besteht aus mehr als einem Symbol
R6 A → BcdE -- RS enthält mehr als ein Zwischen-Symbol
R7: A → bcDe -- das Zwischen-Symbol D steht nicht am Ende von RS

Def.: Eine Grammatik ist vom **Typ 3** (man sagt auch: sie ist **linear**, oder: sie ist **regulär**) wenn
entweder gilt: Alle Regeln sind abschließend oder links-linear.
oder wenn gilt: Alle Regeln sind abschließend oder rechts-linear.

Beispiel für eine Typ 3 Grammatik, G6:

R1: S → 0 R3: S → S0
R2: S → 1 R4: S → S1

Aufgabe 2-1: Begründen Sie kurz, warum folgende Grammatik G7 nicht vom Typ 3 ist:

R1: A → aB
R2: B → Ab
R3: A → ab

Aufgabe 2-2: Geben Sie eine Typ 3 Grammatik G8 an für die Menge aller in ihrer Lieblings-Programmiersprache erlaubten Bezeichner (identifier).

Aufgabe 2-3: Geben Sie eine Typ 3 Grammatik G9 an für die Menge aller Binärzahlen (Ganzzahlen wie 1011 und Brüche wie 10.010, wenn ein Punkt vorhanden ist, muss davor und dahinter mindestens eine Binärziffer stehen, führende und nachfolgende Nullen wie bei den Zahlen 00.000 und 00101.10100 etc. sollen erlaubt sein).

Aufgabe 2-4: Geben Sie eine Typ 3 Grammatik G9A an für die Menge aller Binärzahlen (Ganzzahlen wie 1011 und Brüche wie 10.01, wenn ein Punkt vorhanden ist, muss davor und dahinter mindestens eine Binärziffer stehen, führende und nachfolgende Nullen wie bei den Zahlen 00.000 und 00101.10100 etc. sollen **nicht** erlaubt sein).

Typ 2 Grammatiken (kontextfreie Grammatiken)

Für jede Regel R : LS → RS muß gelten: LS ist (nur) **ein** Zwischen-Symbol.

Beispiele:

R1: A → BCD
R2: A → bcd
R3: A → AbCdeFG
R4: A → ε

Gegenbeispiele:

R5: AB → cd -- LS besteht aus mehr als einem Symbol
R6: Ab → cd -- LS besteht aus mehr als einem Symbol

Typ 1 Grammatiken (kontextsensitive Grammatiken)

Für jede Regel R : LS → RS muß gelten: entweder enthält RS mindestens soviele Symbole wie LS ("R ist eine nicht-verkürzende Regel") oder R hat die Form S → ε ("aus dem Startsymbol kann man die leere Symbolfolge ε ableiten", kurz: "Startsymbol geht nach ε").

Beispiele:

R1: AbCd → eFgH
R2: ABC → DEFG
R3: AB → cde
R4: S → ε

Gegenbeispiele:

R5: AbCd → eFg -- RS ist kürzer als LS
R6: ABC → DE -- RS ist kürzer als LS
R7: A → ε -- nur das Startsymbol darf "nach ε gehen"

Typ 0 Grammatiken

Keine Einschränkung der Regeln.

Terminale und nicht-terminale Symbole einer Grammatik, das Startsymbol

Zu jeder Chomski-Grammtik gehören 2 Alphabete:
Das Alphabet der ***nicht-terminalen Symbole*** (oder: der ***Zwischensymbole***)
Das Alphabet der ***terminalen Symbole*** (oder: der ***Endsymbole***)

Beispiel: Die Grammatik G04 :

```
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"
```

beruht auf folgenden Alphabeten:

Nicht-terminale Symbole: {Zahl, Vorzeich, ZiffFo, Ziff}
Terminale Symbole: {+ - 0 1}

In der Grammatik muss man terminale und nicht-terminale Symbole irgendwie deutlich unterscheiden, z.B. mit Anführungszeichen, mit Unterstreichen oder Überstreichen, mit Farben, ...

Eines der nicht-terminalen Symbole muss als ***Startsymbol*** ausgezeichnet werden (hat Ähnlichkeit mit der main-Funktion in einem C-Programm).

Bessere Bezeichung:

Die Bezeichnung ***kontextfreie Grammatik*** klingt so, als wäre ***kontextfrei*** eine positive Eigenschaft (etwa so wie "frei von einem lästigen oder schädlichen Kontext"). Tatsächlich soll die Bezeichnung eigentlich darauf hindeuten, dass man mit solchen Grammatiken sog. Kontextbedingungen ***nicht beschreiben kann*** (obwohl man das eigentlich gern täte). Eine bessere Bezeichnung wäre also ***kontextunfähige Grammatik***. Da die Bezeichnung ***kontextfrei*** aber allgemein verbreitet ist, werden wir sie auch verwenden. Man sollte aber versuchen, sich durch den falschen positiven Klang nicht verwirren zu lassen. Die Kontextfreiheit einer Grammatik ist eine Schwäche, keine Stärke.

Zur Entspannung: **Ein unlösbares Problem**

Eine D-Gleichung hat folgende Form:

```
Polynom(x1, x2, ..., xn) = 0   // n ∈ {1, 2, 3, ...}
```

Eine Lösung für eine solche D-Gleichung besteht aus n ***ganzen*** Zahlen `(z1, z2, ..., zn)`, auf die die Gleichung zutrifft.

Beispiele: (statt `x1, x2, ...` benutzen wir hier `x, y, ...` als Variablen)

```
1    x^2 + 2y^2      = 0  // Genau eine Lösung: x=0, y=0
2    x^2 - 4         = 0  // Genau zwei Lösungen: x=2 und x=-2
3    x^1 + 5y^1 - 8  = 0  // unendlich viele Lösungen, z.B. x=3, y=1
4    x^2 - y^2 - z^2 = 0  // unendlich viele Lösungen, z.B. x=-5, y=4, z=3
5    x^2 + 5         = 0  // keine Lösung (leicht zu sehen)
6    x^4 - y^4 - z^4 = 0  // keine Lösung (schwer zu beweisen)
```

Satz: Es gibt keinen Algorithmus, der von jeder D-Gleichung korrekt feststellen kann, ob sie lösbar ist (d.h. mindestens eine Lösung hat) oder nicht.

Statt "Es gibt keinen Algorithmus, der ..." kann man hier auch sagen:
"Es ist unmöglich, ein Computerprogrammm zu schreiben, welches ...".

"D-Gleichungen" werden üblicherweise als "diophantische Gleichungen" bezeichnet (nach dem griechischen Mathematiker Diophant von Alexandrien, der irgendwann zwischen 100 v.Chr. und 350 n.Chr., vermutlich um 250 n.Chr., lebte und 13 Bücher über Arithmetik veröffentlichte, von denen 10 bis heute erhalten geblieben sind).

3. SU Fr 14.10.11

Test02

Syntaxbäume

Mit den Regeln einer Grammatik kann man **Syntaxbäume** konstruieren. Die Regeln der Grammatik legen genau fest, welche Syntaxbäume man konstruieren kann und welche nicht.

Als Beispiel betrachten wir wieder die folgende Grammatik G04:

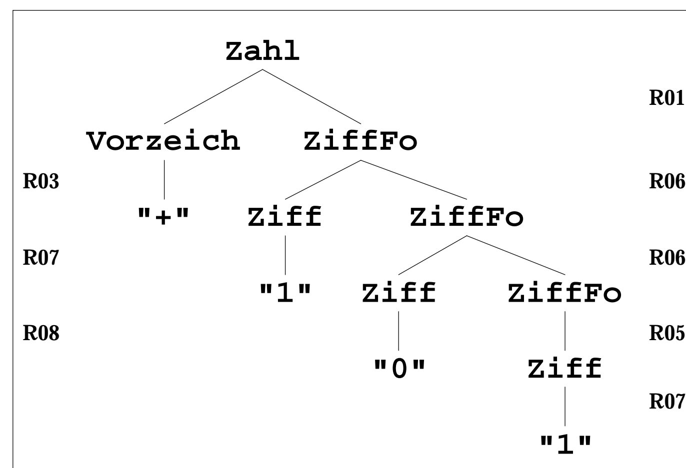
Beispiel: Die Grammatik G04 :

```
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"
```

Def.: Ein **Baum** ist entweder **leer**, oder er bestehat aus einem **Knoten**, an dem 0 oder mehr Bäume hängen.

Jeder nicht-leere Baum besitzt einen obersten Knoten, den man auch als **Wurzel** (-Knoten) des Baumes bezeichnet. Jeder Knoten außer dem Wurzelknoten hat genau einen **Elternknoten**, (oder: Mutterknoten, Vaterknoten, Vorgängerknoten) an dem er hängt. Knoten, an denen 0 weitere Bäume hängen, bezeichnet man auch als **Blätter**, alle anderen Knoten als **innere Knoten** des betreffenden Baumes.

Die ***Knoten*** eines Syntaxbaumes sind (Zwischen- bzw. End-) ***Symbole*** einer Grammatik. Die Wurzel (mit der der Baum "oben anfängt") muss gleich dem ***Startsymbol*** der Grammatik sein (darum heißt das Startsymbol "Startsymbol"). Der folgende Syntaxbaum **SB1** wurde mit den Regeln der Grammatik G04 konstruiert und beginnt deshalb mit dem Startsymbol `Zahl`:



Die Regelnummern (R03, R07, ...) am linken und rechte Rand des Diagramms sind nur Kommentare die andeuten sollen, mit welchen Regeln die einzelnen Teile des Baums konstruiert wurden. Z.B. wurde mit der Regel R03 aus dem Symbol `Vorzeich` das Symbol `"+"` abgeleitet. Mit der Regel R01 wurde aus dem Symbol `Zahl` die Symbolfolge `Vorzeich ZiffFo` abgeleitet etc. Normalerweise werden Syntaxbäume ***ohne*** solche Regelnummern notiert.

Alle **Blätter** eines Syntaxbaums müssen **Endsymbole** (wie `"+"`, `"1"` etc.) sein, alle **inneren Knoten** müssen **Zwischensymbole** (wie `Zahl`, `Vorzeich`, `ZiffFo` etc.) sein.

Wenn man beim Syntaxbaum SB1 alle Endsymbole (d.h. Blätter) zusammenfaßt (und alle doppelten Anführungszeichen wegläßt), erhält man das Wort `+101`. Man sagt auch: "SB1 ist ein Syntaxbaum für das Wort `+101`" oder "Der Syntaxbaum SB1 stellt das Wort `+101` dar".

Aufgabe-01: Konstruieren Sie mit den Regeln der Grammatik G04 einen Syntaxbaum für das Wort `110`.

Die Grammatik **G04** beschreibt die formale Sprache **S04**: {+0, -0, 0, +1, -1, 1, +01, -01, 01, +10, -10, 10, +11, -11, 11, ..., +101, ..., -0010110, ..., 110100111, ...}, denn es gilt:
- Für jedes Wort der Sprache S04 kann man mit den Regeln von G04 einen Syntaxbaum konstruieren.
- Jeder mit den Regeln von G04 konstruierte Syntaxbaum stellt ein Wort der Sprache S04 dar.

Das Beschreiben von formalen Sprachen durch Grammatiken ist etwa so schwer oder leicht wie das Programmieren in einer einfachen, aber ziemlich mächtigen Programmiersprache. Am Besten lernt man es, indem man möglichst viele Aufgaben löst.

Wichtige Grundregel: Immer wenn man für eine formale Sprache FS eine Grammatik geschrieben hat, sollte man die Grammatik testen, indem man versucht, für ein paar Worte aus FS und für ein paar Worte, die nicht zu FS gehören, ***Syntaxbäume zu kontruieren***. Dadurch kann man Fehler in der Grammatik entdecken (das Fehlen von Regeln, die man benötigt oder die Anwesenheit von falschen Regeln, mit denen man unerlaubte Worte ableiten kann).

Zur Entspannung: **Grundlagen für das Verständnis englischer Fachbücher**

Um angloamerikanische Fachbücher verstehen zu können, sollte man sich als Grundlage (unter anderem) mit folgenden Büchern vertraut machen:

1. "**Winnie-the-Pooh**" von A. A. Milne, illustrated von E. H. Shepard.

Z. B. geht der Fachbegriff "a Pooh problem" auf Piglets Frage "Do bees sting?" und Poohs Antwort: "Some do, and some don't!" zurück.

2. "**Alice's Adventures in Wonderland and Through the Looking Glass**" von Lewis Carroll.

Als Beispiel ein wichtiges Zitat: "When ***I*** use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean - neither more nor less.". "The question is," said Alice, "whether you ***can*** make words mean so many different things."

Siehe auch: "**The Annotated Alic**" von Marin Gardner, W. W. Norton & Company

3. "**The Hitch Hikers Guide to the Galaxy**" von Douglas Adams (und die übrigen vier Bände dieser Trilogie).

Zahlreiche englische Artikel und Vorträge beginnen mit der (oder erwähnen die) Antwort **42** und erklären dann die dazugehörige Frage.

4. SU Fr 21.10.11

Test03

Syntaxbäume, eine Ergänzung

Syntaxbäume kann man nur aus Typ-2- und -3-Grammatiken konstruieren, aber nicht aus Typ-1- oder -0-Grammatiken. Sobald auf der linken Seite einer Regel mehr als ein Symbol steht, klappt es nicht mehr mit den Bäumen.

Parser für formale Sprachen

Def. 1: Ein Parser P für eine formale Sprache S ist ein (Unter-) Programm, welches eine ***Zeichenfolge*** F einliest und prüft, ob F ein Wort der Sprache S ist oder nicht.

Viele Parser lesen keine ***Zeichenfolge*** ein, sondern eine ***Symbolfolge,*** und überlassen es einem ***Lexer*** (oder: ***Scanner***), aus der Zeichenfolge eine Symbolfolge zu machen.

Def. 2: Ein Parser P für eine formale Sprache S ist ein (Unter-) Programm, welches eine ***Symbolfolge*** F (oder: Tokenfolge) einliest und prüft, ob F ein Wort der Sprache S ist oder nicht.

Eine geniale Idee: Statt einen Parser für eine Sprache S "von Hand zu programmieren", kann man ihn aus einer Grammatik für S ***automatisch erzeugen lassen*** (von einem sog. Parser-Generator).

Kurze Geschichte der Parser-Generatoren:

In den 1960-er-Jahren war es große Mode, Parser-Generatoren (für Typ-2-Grammatiken) zu schreiben (und "aus dem Überschwang der Zeit heraus" als ***Compiler-Compiler*** zu bezeichnen).

1973 veröffentlichte Stephen C. Johnson den bis heute berühmtesten Parser-Generator namens **yacc** (yet another compiler compiler, auf Deutsch etwa: Noooch so nen Compiler-Compiler). Eine neuere Variante davon heißt **bison**.

Die Parser-Generatoren yacc und bison können ***nicht für alle*** Grammatiken Parser erzeugen, sondern nur für **LALR-Grammatiken**. Dabei ist **LALR** eine schwierige Eigenschaft, die man kaum "von Hand" feststellen kann, nur mit Hilfe eines Programms. Wenn eine Grammatik nicht LALR ist, melden yacc und bison sog. `shift-reduce-` oder `reduce-reduce-`Konflikte.

Der Parser-Generator **accent** (**a** **c**ompiler-compiler for the **ent**ire class of contextfree grammers) kann für ***jede*** kontextfreie Grammatik einen Parser erzeugen.

Wenn der yacc (oder der bison) einen Parser erzeugen (statt Konflikte zu melden), dann ist das ein "schneller Parser".

Die vom Generator accent erzeugten Parser sind heute in vielen Fällen "schnell genug" aber in in einigen Fällen ein bisschen langsam.

Die Paraser-Generatoren yacc, bison und accent haben wichtige Gemeinsamkeiten:
1. ihre Grammatik-Spezifikationen (die Formate ihrer Eingabedaten) sind gleich
2. sie setzen einen ***Lexer*** voraus, der z.B. mit dem Lexer-Generator **lex** (oder **flex**) erzeugt werden kann.

Zur Entspannung: **Ist der folgende Satz wahr oder nicht?**

Dieser Satz enthält trei Fähler!. Stimmt das oder nicht? Offenbar enthält der Satz zwei syntaktische Fehler (***trei*** statt ***drei*** und ***Fähler*** statt ***Fehler***) und einen semantischen Fehler (drei statt zwei). Damit ist er also wahr. Aber wenn er wahr ist, enthält er keinen semantischen Fehler und ist somit falsch. Aber wenn er falsch ist, enthält er einen semantischen Fehler und ist somit wahr. Aber wenn er ... .

Das Typ-System der Sprache Gentle

In Gentle gibt es genau ***drei vordefinierte Typen***: `INT`, `STRING` und `POS`. Die Typen `INT` und `STRING` entsprechen weitgehend den Typen `int` und `char *` des verwendeten C-Compilers. Jeder Wert des Typs `POS` beschreibt "eine bestimmte Position innerhalb einer Quelldatei" (eine Datei-Nr., eine Zeilen-Nr. und eine Spalten-Nr.).

Der Programmierer kann weitere Typen vereinbaren. Hier ein paar Beispiele für Typ-Vereinbarungen:

```
 1 'type' TAG mo di mi di do fr sa so -- Ein Aufzählungstyp (mit 7 Werten)
 2 'type' TERMIN                      -- Ein Verbundtyp (record-, struct-type)
 3    ter(Wochentag: TAG, Stunde: INT)
```

Die Namen `Wochentag:` und `Stunden:` haben den Character von ***Kommentaren*** und können auch weggelassen werden. etwa so:

```
 4 'type' TERMIN                      -- Ein Verbundtyp (record-, struct-type)
 5    ter(TAG, INT)
 6
 7 'type' KFZ_INFO                    -- Ein varianter Verbundtyp (union type)
 8    pkw(Kennzeichen: STRING, Sitze: INT)                // Variante 1
 9    lkw(Kennzeichen: STRING, Gewicht: INT, Achsen: INT); // Variante 2
10
11 'type' LISTE0                      -- Ein rekursiver Typ
12    leer                            -- Die leere Liste (Laenge 0)
13    liste(TERMIN, Rest: LISTE0)     -- Nicht-leere Listen (von TERMINen)
14
15 'type' LISTE1                      -- Ein rekursiver Typ
16    letzt(TERMIN)                   -- Eine Liste der Laenge 1
17    liste(TERMIN, Rest: LISTE1)     -- Laengere Listen (von TERMINen)
18
19 'type' BAUM0                       -- Ein rekursiver Typ
20    leer
21    knoten(TERMIN, BAUM0, BAUM0)
22
23 'type' BAUM1                       -- Ein rekursiver Typ
24    blatt(TERMIN)
25    knoten(TERMIN, BAUM1, BAUM2)
```

Die Typen, die man in Gentle vereinbaren kann, werden auch als ***algebraische Typen*** bezeichnet. Die Bestandteile `mo`, `di`, ..., `so`, `ter`, `pkw`, `lkw`, `leer`, `liste` `knoten`, `blatt`, ... der Typvereinbarungen werden als ***Funktoren*** oder auch als ***Konstruktoren*** bezeichnet.

Die Werte eines algebraischen Typs sind ***Terme*** (genauer: ***Grundterme***, d.h. Terme, in denen ***keine Variablen*** vorkommen). Zum Typ `TAG` gehören nur die 7 Grundterme `mo`, `di`, ..., `so`.

Zum Typ `TERMIN` gehören Grundterme wie
`ter(mo, 10)`, `ter(di, 0)`, `ter(so, 17)`, ...

Zum Typ `KFZ_INFO` gehören Grundterme wie
`pkw("B", 5)`, `pkw("WOB", 6)`, `lkw("B", 24, 3)`, ...

Aufgabe: Geben Sie je zwei Werte (Grundterme) der folgenden Typen an:
`LISTE0`, `LISTE1`, `BAUM0`, `BAUM1`

Achtung: Die in Gentle vordefinierten Typen `INT`, `STRING` und `POS` sind ***keine algebraischen*** Typen.

Das Papier **Terme in Gentle** (11 Seiten, 3 Blätter) austeilen und besprechen.

5. SU Fr 28.10.11

Test04

Besprechung des Papiers "Terme in Gentle" fortsetzen

Def.-04: Variablenbelegung (S. 2, ca. Zeile 14)

Zimmerleute haben manchmal eine Art "Kombi-Werkzeug":
Vorn ein Hammer (zum Einschlagen von Nägeln)
Hinten ein Haken (zum Rausziehen von Nägeln)

Terme kann man auch auf zwei Weisen benutzen: Als ***Ausdrücke*** und als ***Muster***

Auswertung	(Ausdruck	, Variablenbelegung)	ergibt einen	Wert und
Musterabgleich	(Muster	, Wert)	ergibt eine	Variablenbelegung

Das Auswerten von Ausdrücken gibt es in fast allen Programmiersprachen.
Das Abgleichen von Mustern gibt es z.B. in Prolog, Haskell, Scala und Gentle.

Beispiel-05: Eine Auswertung (S. 2, unten)
Beispiel-06: Ein Musterabgleich (S. 2, unten)

Auswertungen gelingen immer, ***Musterabgleiche*** können gelingen oder ***misslingen.***

Beispiel-07: Ein Musterabgleich der misslingt (S. 3 oben)

Aufgabe-04: Musterabgleiche ausführen

Druckfehler in Teilaufgabe 4.6.: Das Muster sollte gleich `b(c, m, leer, leer)` sein
(nicht `b(c, y, leer, leer)`)

Verschiedene Arten von Prädikaten in Gentle (S. 4 oben)

Unterschied zwischen `action-` und `condition-Prädikaten:`

Aufrufe von `action-Prädikaten` sollten ***immer gelingen*** (sonst: Fehler des Programmierers)
Aufrufe von `condition-Prädikaten` dürfen ***gelingen oder misslingen***

Zusammenhang von Auswertungen und Musterabgleichen mit Prädikaten in Gentle

Die Ausfürung eines Prädikat-Aufrufs besteht aus einer Folge

Auswertung, Musterabgleich, Auswertung, Musterabgleich, ..., Auswertung, Musterabgleich

Als erstes Beispiel betrachten wird das Prädikat `Vertausche` (S. 4, Zeilen 8-10)

Die Regeln sind besonders einfach und haben keine rechten Seiten.

Was muss man ***auf der linken Seite einer Regel*** als Parameter angeben? (S. 5, Zeile2 11-14)
Was muss man ***beim Aufruf eines Prädikats*** als Parameter angeben? (S. 5, Zeile 16-18)

Was passiert bei der Ausführung des Prädikataufrufs? (S. 6, Grafik mit viel Text darin)

Verwaltung einer Symboltabelle in einem Gentle-Programm (Aufgabe)

```
 1 -- Die Symboltabelle ordnet jedem Quellbezeichner einen Zielbezeichner
 2 -- zu.  Die Quellbezeichner sind Variablennamen (vom Typ STRING). Die
 3 -- Zielbezeichner sind Ganzzahlen (1, 2, 3, ...), weil in einem Bytecode-
 4 -- Programm (d.h. in einer .class-Datei) die lokalen Variablen der
 5 -- main-Methode mit diesen Ganzzahlen bezeichnet werden.
 6 -- Die Symboltabelle wird als Variable des folgenden Typs realisiert
 7 -- (und GlobSymTab ist die einzige Variable dieses Typs):
 8
 9 'type' SYMTAB
10    empty
11    list(QuellBezeich:STRING, ZielBezeich:INT, Rest: SYMTAB)
12
13 -- Die Symboltabelle wird als globale Variable realisiert. Eine
14 -- weitere globale Variablen enthaelt den naechsten Zielbezeichner
15 -- (eine Ganzzahl). Diese beiden Variablen muessen "aus technischen
16 -- Gruenden" mit Hilfe eines pseudo-nonterm-Praedikats initialisiert
17 -- werden (siehe weiter unten das Praedikat Init):
18 'var' GlobSymTab:              SYMTAB
19 'var' NaechsterZielbezeichner: INT
20 ----------------------------------------------------------------------
21 -- Praedikate zur Verwaltung der Symboltabelle:
22
23 'condition' HasMeaningS(QuellBezeichner: STRING -> ZielBezeichner: INT)
24 -- Gelingt und liefert den entsprechenden ZielBezeichner wenn der
25 -- Quell-Bezeichner schon in der Symboltabelle definiert ist.
26 -- Hinweis zur Implementierung: Dieses Praedikat dient nur dazu,
27 -- das Hilfspraedikat HasMeaning2 aufzurufen. Das Hilfspraedikat
28 -- ist rekursiv und "erledigt die eigentliche Arbeit".
29 -- MUSS ERGAENZT WERDEN!!!
30
31 'condition' HasMeaning2(QuellBezeichner: STRING, SYMTAB -> ZielBez: INT)
32 -- Rekursives Hilfspraedikat fuer HasMeaningS (erledigt die eigentliche
33 -- Arbeit, d.h. such in der Symboltabelle nach der Bedeutung (meaning)
34 -- des als Parameter uebergebenen QuellBezeichners):
35 -- MUSS ERGAENZT WERDEN!!!
36
37 'action'    Definiere(QuellBezeichner: STRING -> Fehler: INT)
38 -- Wenn der QuellBezeichner noch nicht in der globalen Symboltabelle
39 -- eingetragen ist, wird er eingetragen und 0 als Fehleranzahl
40 -- geliefert. Sonst wird eine Fehlermeldung ausgegeben und 1 als
41 -- Fehleranzahl geliefert.
42 -- MUSS ERGAENZT WERDEN!!!
43
44 'action'    DefMeaningS(QuellBezeichner: STRING, ZielBezeichner: INT)
45 -- Der QuellBezeichner wird (zusammen mit dem ZielBezeichner) in die
46 -- globale Symboltabelle GlobSymTab eingetragen.
47 -- MUSS ERGAENZT WERDEN!!!
48
49 'action'    MeldeWennUndefiniert(QuellBez: STRING->Fehler: INT)
50 -- Gibt eine kleine Fehlermeldung zur Standardausgabe aus, wenn der
51 -- QuellBezeichner noch nicht in der Symboltabelle eingetragen ist:
52 -- MUSS ERGAENZT WERDEN!!!
53 ----------------------------------------------------------------------
54 -- Kontextfreie Grammatik (Parser) fuer die Sprache Alg00
55 -- plus Pruefung der Kontextbedingungen KB1 und KB2:
56
57 -- Ein pseudo-nonterm-Praedikat zum Initialisieren aller globalen
58 -- Variablen:
59 'nonterm' Init
60    'rule' Init:
61       GlobSymTab              <- empty
62       NaechsterZielbezeichner <- 1
```

Empfehlung: Programmieren Sie die Prädikate in folgender Reihenfolge: HasMeaningS, HasMeaning2, DefMeaningS, Definiere, MeldeWennUndefiniert

Lösung der vorangehenden Aufgabe:

```
 1 -----------------------------------------------------------------------
 2 -- Praedikate zur Verwaltung der Symboltabelle:
 3
 4 'condition' HasMeaningS(QuellBezeichner: STRING -> ZielBezeichner: INT)
 5 -- Gelingt und liefert den entsprechenden ZielBezeichner wenn der
 6 -- Quell-Bezeichner schon in der Symboltabelle definiert ist:
 7    'rule'   HasMeaningS(QB -> ZB):
 8       GlobSymTab -> LokSymTab
 9       HasMeaning2(QB, LokSymTab -> ZB)
10
11 'condition' HasMeaning2(QuellBezichner: STRING, SYMTAB -> ZielBez: INT)
12 -- Rekursives Hilfspraedikat fuer HasMeaningS:
13    'rule'   HasMeaning2(QB, list(QB1, ZB1, Rest) -> ZB1): eq(QB, QB1)
14    'rule'   HasMeaning2(QB, list( _ ,  _ , Rest) -> ZB ):
15       HasMeaning2(QB, Rest -> ZB)
16
17 'action'    Definiere(QuellBezeichner: STRING -> Fehler: INT)
18 -- Wenn der QuellBezeichner noch nicht in der globalen Symboltabelle
19 -- eingetragen ist, wird er eingetragen und 0 als Fehleranzahl
20 -- geliefert. Sonst wird eine Fehlermeldung ausgegeben und 1 als
21 -- Fehleranzahl geliefert.
22    'rule'   Definiere(QB -> 1):
23       HasMeaningS(QB -> ZB)
24       PutS("Alg00: Der Bezeichner ")
25       PutS(QB)
26       PutS(" wird mehrfach definiert!") Nl
27    'rule'   Definiere(QB -> 0):
28       NaechsterZielbezeichner -> ZB
29       NaechsterZielbezeichner <- ZB+1
30       DefMeaningsS(QB, ZB)
31
32 'action'    DefMeaningsS(QuellBezeichner: STRING, ZielBezeichner: INT)
33 -- Der QuellBezeichner wird (zusammen mit dem ZielBezeichner) in die
34 -- globale Symboltabelle GlobSymTab eingetragen.
35    'rule'   DefMeaningsS(QB, ZB):
36       GlobSymTab              -> LokSymTab
37       GlobSymTab              <- list(QB, ZB, LokSymTab)
38
39 'action'    MeldeWennUndefiniert(QuellBez: STRING->Fehler: INT)
40 -- Gibt eine kleine Fehlermeldung zur Standardausgabe aus, wenn der
41 -- QuellBezeichner noch nicht in der Symboltabelle eingetragen ist:
42    'rule'   MeldeWennUndefiniert(QB->0):
43       HasMeaningS(QB -> ZB) -- Alles o.k.
44    'rule'   MeldeWennUndefiniert(QB->1):
45       PutS("Alg00: Der Bezeichner ") PutS(QB)
46       PutS(" wird benutzt, ist aber nicht definiert!") Nl
47 -----------------------------------------------------------------------
```

Zur Entspannung: **Nicht-transitive Würfel**

Viele "Vergleichs-Relationen" wie ***istSchneller, istHöher, istGrößer*** etc. sind ***transitiv***, d.h.
wenn A schneller ist als B und B schneller ist als C dann gilt auch: A ist schneller als C.

Betrachten wir noch eine andere ***istBesser***-Relation: Zwei Personen würfeln mehrmals mit zwei Würfeln W1 und W2 gegeneinander. Wer die höhere Augenzahl würfelt, hat den Wurf gewonnen und bekommt einen Punkt. Die Würfel W1 und W2 können unterschiedliche Zahlen auf ihren sechs Seiten haben und somit besser oder schlechter sein. Z.B. ist ein Würfel, der auf allen sechs Seiten eine 6 hat offensichtlich besser als einer, der auf allen sechs Seiten eine 1 hat. Aber was ist mit den folgenden 4 Würfeln?

W1: 0 0 4 4 4 4
W2: 3 3 3 3 3 3
W3: 2 2 2 2 6 6
W4: 1 1 1 5 5 5

In einem Kampf W1 gegen W2 wird W1 im Durchschnitt
2/3 aller Spiele gewinnen (mit 4 zu 3) und
1/3 verlieren (mit 0 zu 3).
W1 ist also besser als W2.

Aus ganz ähnlichen Gründen gilt außerdem:

W2 ist besser als W3.
W3 ist besser als W4.
Und erstaunlicherweise auch: W4 ist besser als W1.

Bei Fußball-Turnieren und ähnlichen Wettkämpfen glauben vermutlich viele Fans, dass die Sieger-Mannschaft besser ist, als alle anderen teilnehmenden Mannschaften. Aber wieso sollte das für so komplizierte Gebielde wie Fußballmannschaften gelten, wenn es schon für so einfache Dinge wie Würfel nicht gilt?

Quelle: Martin Gardner, "The Colossal Book of Mathematics",
W. W. Norton & Company, ca. 700 Seiten, 35,- US$

6. SU Fr 04.11.11

Test05 (Musterabgleich), Fragestunde zu Prädikaten, dann Wiederholung von **Test04** (Prädikate)

Fragestunde zu Prädikaten

Was steht in einem C-Programm "in der ersten Zeile" einer Funktionsdefinition?
(der Rückgabetyp, der Name der Funktion, die Namen und Typen der Parameter)

Was steht in einem Gentle-Programm "in der ersten Zeile" ein Prädikatsdefinition?
(die Art des Prädikats, der Name des Prädikats, die Typen der Ein- und Ausgabe-Parameter)

In den Regeln eines Prädikats kommen (fast) ***nie*** Typnamen vor wie `INT`, `STRING`, `LISTE` etc., sondern nur ***Namen von Prädikaten*** und ***Namen von Variablen*** wie `N`, `ERG`, `QB`, `ZB`, ...

Wenn eine Liste als Eingabeparameter bearbeitet werden soll:

Zuerst Regeln für "ganz einfache Listen"
Dann Regeln für "kompliziertere Listen"

```
 1 'type' LISTE                     -- Kann auch leer sein
 2    leer
 3    k(Element: INT,    Rest: LISTE)
 4    s(Element: STRING, Rest: LISTE)
 5
 6 'action'  anzPosElem(L: LISTE -> AnzahlDerPositivenElemente: INT)
 7    -- Ermittelt die Anzahl der positiven Zahlen in der Liste L.
 8                                         -- Regel fuer
 9    'rule' anzPosElem(leer -> 0)         -- leere Liste
10    'rule' anzPosElem(k(N, RL) -> 1 + ARL): -- 1. Elem ist positive Zahl
11       gt(N, 0)
12       anzPosElem(RL -> APRL)
13    'rule' anzPosElem(k(N, RL) -> APRL)     -- 1. Elem ist Zahl, nicht positiv
14       anzPosElem(RL -> APRL)
15    'rule' anzPosElem(s(T, RL) -> APRL):    -- 1. Elem ist Text
16       anzPosElem(Rl -> APRL)
17
18 'action'  listePosElem(L: LISTE -> ListeDerPositivenElemente: LISTE)
19    -- Ermittelt die Liste der positiven Zahlen aus der Liste L.
20    'rule' listePosElem(leer -> leer)
21    'rule' listePosElem(k(N, RL) -> k(N, PERL)):
22       gt(N, 0)
23       anzPosElem(RL -> PERL)
24    'rule' listePosElem(k(N, RL) -> PERL):
25       anzPosElem(RL -> PERL)
26    'rule' listePosElem(s(T, RL) -> PERL):
27       anzPosElem(RL -> PERL)
```

Test04. Jeder bekommt eine mit seinem Namen gekennzeichnete Test-Version.

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. SU Fr 11.11.11

Test06

Falsche Email-Adresse <`steffenrot31@aol.com`> von **Steffen Roth**?

Papier "Terme, Ausdrücke und Muster in Gentle", Besprechung fortsetzen

Terme sind diese "Dinger, in denen Variablen vorkommen können".
Einen Term kann man als ***Ausdruck*** oder als ***Muster*** benutzen ("Kombiwerkzeug").
Was ist der ***Unterschied*** zwischen einem ***Ausdruck*** und einem ***Muster?***

Für die Variablen in einem Ausdruck ***sollten Werte festgelegt sein***
(damit man den Ausdruck auswerten kann).

Für die Variablen in einem Muster ***sollten keine Werte festgelegt sein***
(damit man die Werte der Variablen durch einen Musterabgleich ermitteln kann)

Wenn man Prädikate programmiert, muss man nach einem festen Muster "***immer abwechselnd***" ***Ausdrücke*** und ***Muster*** hinschreiben. Das soll anhand eines Beispiels genauer erläutert werden.

S. 4, Mitte: Ein Typ namens `LISTE`
Hat jemand gegähnt? Sehr gut, wir haben diesen und ähnliche Typen ja schon sehr oft gesehen :-).

Kurz darunter: Ein simples Prädikat namens `Vertausche`
Es hat nur 2 Regeln. Keine der beiden Regeln hat eine rechte Seite.

Wie viele Parameter hat dieses Prädikat?
(1 Eingabeparameter, 1 Ausgabeparameter)

Auf was für Listen als Eingabeparameter kann die erste Regel angewendet werden?
(Auf Listen, die mindestens 2 Elementen enthalten)
Was mach die erste Regel?
(Sie vertauscht die ersten beiden Elemente der Eingabeliste)
Auf was für Listen als Eingabeparameter wird die zweite Regel angewendet?
(Auf Listen mit weniger als 2 Elementen)

Regel 1: Auf der linken Seite einer Regel (im Kopf der Regel) muss man
für jeden ***Eingabeparameter*** ein ***Muster*** und für jeden ***Ausgabeparameter*** einen ***Ausdruck*** angeben.

Regel 2: In einem Aufruf eines Prädikats muss man
für jeden ***Eingabeparameter*** einen ***Ausdruck*** und für jeden ***Ausgabeparameter*** ein ***Muster*** angeben.

S. 6, Die kleine Graphik mit den Pfeilen

In der ersten Zeile der Graphik:
Was ist `list(25, TL1)` für ein Ding? (ein Ausdruck)
Von welchem Typ ist die Variable `TL1`? (Vom Typ `LISTE`)
Woher wissen wir, dass `TL1` vom Typ `LISTE` ist? (Aus Zeile 3 und 8 auf S. 4)

Was ist `list(ERG1, ERG2)` für ein Ding? (Ein Muster)
Von welchen Typ ist die Variable `ERG1`? (`INT`) Und die Variable `ERG2`? (`LISTE`)
Woher wissen wir das? (Aus Zeile 3 auf S. 4)

In der letzten Zeile der Grafik:
Was ist `list(E1, list(E2, R2))` für ein Ding? (Ein Muster)
Von welchenTypen sind die Variablen `E1, E2, R2? (INT, INT, LISTE)`
Was ist `list(E2, list(E1, R2))` für ein Ding? (Ein Ausdruck)

S. 6, Aufgabe-05

Token-Prädikate in .t-Dateien definieren

In einem Gentle-Programm kann ein Token-Prädikat z.B. wie folgt ***deklariert*** werden:

```
'token' GanzLiteral(-> INT)
```

Definiert wir so ein Token-Prädikat in einer `.t`-Datei, etwa so:

```
 1  /* ------------------------------------------------------------------- */
 2  /* IlitPas.t: beschreibt Ganzzahl-Literale, die (nur) aus dezimalen     */
 3  /* Ziffern bestehen duerfen. Vorzeichen, Unterstriche  etc. sind NICHT  */
 4  /* erlaubt. Liefert den Wert des Literals als Ganzzahl vom Typ int.     */
 5  /* ------------------------------------------------------------------- */
 6 [0-9]+ {
 7    yylval.attr[1] = atoi (yytext);
 8    yysetpos();
 9    return GanzLiteral;       /* Dem Parser ein Token GanzLiteral liefern */
10 }
11  /* ------------------------------------------------------------------- */
```

In Zeile 6 steht ein regulärer Ausdruck, der Ganzzahlliterale beschreibt.
Dahinter steht eine C-Blockanweisung (Zeile 6 bis 10)
In der Variablen `yytext` steht die vom Lexer im Quellprogramm erkannte Zeichenkette.
In die Reihung `yylval.attr` müssen Werte für die Ausgabeparameter des Token-Prädikats `GanzLiteral` geschrieben werden. Im Beispiel hat das Token-Prädikat nur ***einen*** Ausgabeparameter vom Gentle-Typ `INT`, der entspricht dem C-Typ `int`).

Token-Prädikate dürfen/können keine Eingabeparameter haben.

Die Methode yysetpos verschiebt die ***Position des Lexers*** (in der Quelldatei) hinter die gerade erkannte und bearbeitete Zeichenkette (die in `yytext` steht). Die ***Position*** besteht aus einer Datei-Nr, einer Zeilen-Nr. und einer Spalten-Nr.

Damit die Datei `IlitPas.t` als Definition des Token-Prädikats `GanzLiteral` gefunden wird, muss man im `build`-Skript das Programm `reflex` wie folgt aufrufen:

```
reflex.exe GanzLiteral=ILitPas.t
```

Das Token-Prädikat `GanzLiteral` kann auch durch die folgende `.t`-Datei definiert werden:

```
 1  /* ----------------------------------------------------------------- */
 2  /* IlitAda.t: beschreibt Ganzzahl-Literale, die aus dezimalen Ziffern  */
 3  /* und Unterstrichen bestehen duerfen (aehnlich wie in Ada).           */
 4  /* Ein Unterstrich darf aber nur zwischen zwei Ziffern stehen.         */
 5  /* Liefert den Wert des Literals als Ganzzahl vom Typ INT.             */
 6  /* ----------------------------------------------------------------- */
 7 [0-9](_[0-9]|[0-9])* {
 8    char *p  = yytext;
 9    int   d = 0;         /* Distanz, um die Zeichen verschoben werden    */
10    while (*p != '\0') { /* Unterstriche "_" aus yytext entfernen        */
11       if  (*p == '_')
12          d++;
13       else
14          *(p-d) = *p;   /* Zeichen um d Stellen nach links kopieren     */
15       p++;
16    }
17    *(p-d) = '\0';       /* abschliessende Null nach yytext              */
18
19    yylval.attr[1] = atoi (yytext);
20    yysetpos();
21    return GanzLiteral;      /* Dem Parser ein Token GanzLiteral liefern */
22 }
23  /* ----------------------------------------------------------------- */
```

Das Token-Literal

```
'token' Bezeich(-> STRING)
```

kann z.B. durch die folgned `.t`-Datei definiert werden:

```
 1  /* ------------------------------------------------------------------ */
 2  /* IdntCehStr.t: Bezeichner (identifier) wie in C                     */
 3  /* Liefert den Bezeichner als Zeichenkette vom Typ STRING             */
 4  /* ------------------------------------------------------------------ */
 5 [A-Za-z_][A-Za-z0-9_]* {
 6    yylval.attr[1] = strdup(yytext);
 7    yysetpos();
 8    return Bezeich;               /* Dem Parser ein Token Bezeich liefern */
 9 }
10  /* ------------------------------------------------------------------ */
```

Zu Gentle gehört auch eine vordefinierte Symboltabellen-Verwaltung. Sie erlaubt es, ***Bezeichner*** zusammen mit ihrer ***Bedeutung*** in eine Symboltabelle einzutragen. Die Bedeutung darf zu einem beliebigen Gentle-Typ `MEANING` gehören. Die öffentliche Schnittstelle dieser Symboltabellen-Verwaltung besteht aus den folgenden vier Methoden:

```
 1 'action' id_to_string (IDENT  -> STRING)
 2 'action' string_to_id (STRING -> IDENT)
 3
 4 'condition' HasMeaning(QuellBezeichner: IDENT -> Bedeutung: MEANING)
 5 -- Gelingt und liefert die entsprechenden Bedeutung wenn der
 6 -- Quell- Bezeichner schon in der Symboltabelle definiert ist:
 7
 8 'action'    DefMeaning(QuellBezeichner: IDENT, Bedeutung: MEANING)
 9 -- Der QuellBezeichner wird (zusammen mit seiner Bedeutung) in die
10 -- Symboltabelle eingetragen.
```

Diese Symboltabellen-Verwaltung verwaltet Quellbezeichner vom Typ `IDENT`. Wenn Quellbezeichner als Strings vorliegen, müssen sie mit `string_to_id` umgewandelt werden.

Um diese Symboltabellen-Verwaltung benutzen zu können, sollte man das Token-Prädikat `Bezeich` nicht durch obige Datei `IdntCehStr.t` definieren, sondern z.B. durch die folgende `.t`-Datei:

```
 1  /* ------------------------------------------------------------------ */
 2  /* IdntCehIde.t: Bezeichner (identifier) wie in C.                    */
 3  /* Liefert einen Wert vom Typ IDENT (d.h. einen Zeiger)               */
 4  /* ------------------------------------------------------------------ */
 5 [A-Za-z_][A-Za-z0-9_]* {
 6    long id;
 7    string_to_id (yytext, &id);
 8    yylval.attr[1] = id;        /* Wert vom Typ IDENT nach yyval.attr[1]  */
 9    yysetpos();
10    return Bezeich;             /* Dem Parser ein Token Bezeich liefern   */
11 }
12  /* ------------------------------------------------------------------ */
```

Natürlich kann man anstelle des regulären Ausdrucks in Zeile 5 auch einen anderen angeben, z.B. einen der folgenden:

```
[A-Za-z][A-Za-z0-9]*                  // Bezeichner wie in Pascal oder Modula
[A-Za-z](_[A-Za-z0-9]|[A-Za-z0-9])*   // Bezeichner wie in Ada
[A-Za-z][A-Za-z0-9_]*                 // Bezeichner wie in Eiffel
```

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

Der Werwolf

Ein Werwolf eines Nachts entwich ...

Merkblatt: ***Reguläre Ausdrücke*** für die Lexer-Generatoren lex und flex

<table>
<tr><th>Der Reguläre Ausdruck</th><th>paßt auf ...</th></tr>
<tr><td>.</td><td>ein beliebiges Zeichen (ausser auf ein Zeilenwechselzeichen \n)</td></tr>
<tr><td>*</td><td>0 Mal oder häufiger das, was unmittelbar davor steht.</td></tr>
<tr><td>abc*</td><td>ab, abc, abcc, abccc, ...</td></tr>
<tr><td>a(bc)*</td><td>a, abc, abcbc, abcbcbc, ...
(die runden Klammern (...) dienen zum Zusammenfassen)</td></tr>
<tr><td>+</td><td>1 Mal oder häufiger das, was unmittelbar davor steht.</td></tr>
<tr><td>abc+</td><td>abc, abcc, abccc, ...</td></tr>
<tr><td>a(bc)+</td><td>abc, abcbc, abcbcbc, ...</td></tr>
<tr><td>?</td><td>0 Mal oder 1 Mal das, was unmittelbar davor steht.</td></tr>
<tr><td>abc?</td><td>ab, abc</td></tr>
<tr><td>a(bc)?</td><td>a, abc</td></tr>
<tr><td>{2,5}</td><td>mindestens 2 und höchstens 5 Mal das, was unmittelbar davor steht</td></tr>
<tr><td>abc{2,5}</td><td>abcc, abccc, abcccc, abccccc</td></tr>
<tr><td>a(bc){2,5}</td><td>abcbc, abcbcbc, abcbcbcbc, abcbcbcbcbc</td></tr>
<tr><td>[a3?]</td><td>eines der Zeichen in den eckigen Klammern</td></tr>
<tr><td>[a-z0-9]</td><td>einen kleinen Buchstaben oder eine Ziffer</td></tr>
<tr><td>[^a3?]</td><td>ein Zeichen, welches nicht in den eckigen Klammern steht</td></tr>
<tr><td>[^a-z0-9]</td><td>ein Zeichen, welches kein kleiner Buchstabe und keine Ziffer ist.</td></tr>
<tr><td>^
(als erstes Zeichen eines regulären Ausdruck)</td><td>den Anfang einer Zeile</td></tr>
<tr><td>^abc</td><td>abc, wenn es ganz am Anfang einer Zeile steht</td></tr>
<tr><td>$
(als letztes Zeichen eines regulären Ausdruck)</td><td>das Ende einer Zeile</td></tr>
<tr><td>abc$</td><td>abc, wenn es ganz am Ende einer Zeile steht</td></tr>
<tr><td>^abc$</td><td>abc, wenn es allein auf einer Zeile steht</td></tr>
<tr><td>Anna|Bert|Carl</td><td>Anna, Bert, Carl</td></tr>
<tr><td>an/halten</td><td>an, aber nur wenn halten dahinter steht</td></tr>
<tr><td>\.</td><td>auf einen Punkt .
(der Abwärtsschrägstrich \ nimmt dem Punkt seine besondere Bedeutung)</td></tr>
<tr><td>*</td><td>auf einen Stern *</td></tr>
<tr><td>"1.5+3*4"</td><td>auf 1.5+3*4
(die Anführungszeichen " ... " nehmen allen Zeichen dazwischen ihre besondere Bedeutung)</td></tr>
</table>

8. SU Fr 18.11.11

Test07

Blitz-Einführung in die Java Virtual Machine (JVM)

Die JVM ist so etwas wie ein ***Prozessor***. Es gibt einige Ähnlichkeiten zwischen der JVM und konventionellen Prozessoren (wie z.B. ***Intel i386***, ***i486***, ***AMD Athlon***, ***Intel Pentium***, ***Motorola 68000*** oder ***Sun SPARC***), aber auch wichtige Unterschiede (behandeln wir später genauer).

Papier **Die virtuelle Java Maschine** austeilen.

Architektur: Die JVM hat einen ***Speicher*** (engl. storage, memory) und einen ***Stapel*** (engl. stack). Sie hat ***keine Register***.

S. 3, oben, Eine Java-Klasse namens `Hallo11`

S. 5, Ein Assembler-Programm für die JVM

Kommentare beginnen mit einem Semikolon ;

Bestimmte ***globale Informationen*** beginnen mit einem Punkt und einem Schlüsselwort, z.B. `.source`, `.class`, `.super`

Informationen wie z.B. `.line 14` sollen einen Zusammenhang zum Java-Quellprogramm herstellen, aus dem die Java-Assemblerdatei entstanden ist. Im Beispiel ist das die Datei `Hallo11.java` (siehe S. 3 oben)

Konstruktoren werden als statische Methoden namens <init> mit dem Ergebnistyp `V` (wie `void`) dargestellt (siehe S.5, Zeile 11).

Statische Initialisierer (die in einem Java-Quellprogramm so aussehen: `static { ... }`) werden als statische Methoden namens <cinit> mit dem Ergebnistyp `V` (wie `void`) dargestellt (siehe S. 6, Zeile 47).

Zur Erinnerung: In einem Java-Quellprogramm haben Konstruktoren ***keinen*** Ergebnistyp!

Die 8 primitiven Typen von Java und der Typ `void` haben je ***einen Großbuchstaben*** als Namen:

Typ	byte	character	double	float	int	long	short	void	boolean
Buchst.	**B**	**C**	**D**	**F**	**I**	**J**	**S**	**V**	**Z**

Die Namen von ***Refernztypen*** müssen voll qualifiziert angegeben werden (d.h. "mit allen Paketnamen davor") und zwischen die Zeichen Ell `L` und Semikolon `;` eingeschlossen werden, z.B. so:

`Ljava/lang/String;` oder `Ljava/lang/Object` ; oder `LHallo11;` etc.

Anmerkung: Die Klasse `Hallo11` gehört zum namenlosen Paket.

In einem ***Java-Assembler-Propgramm*** (Jasmin-Programm) wird das Ziel eines Sprungbefehls (wie z.B. `goto` oder `ifeq` oder `if_icmpeq` etc.) durch sog. ***Labels*** angegeben. Diese Labels gelten nur innerhalb einer Methode (oder innerhalb eines Konstruktors).

In einem ***Java-Maschinen-Programm*** (Bytecode-Programm) enthalten Sprungbefehle ***keine Labels***, sondern eine Zahl, die angibt, wie viele Bytes (vom Anfang des Sprungbefehls aus gerechnet) nach vorn oder hinten gesprungen werden soll, z.B. so: Eine +5 bedeutet: 5 Bytes nach vorn springen, -7 bedeutet: 7 Bytes zurück springen etc.

Anmerkung: Das Java-Assembler-Programm auf S. 5 enthält mehrere Labels namens `Label10` und `Label11`, aber leider keinen Sprungbefehl.

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. Im September 2009 entschuldigte sich der britische Premierminister Gordon Brown dafür, wie Turing behandelt worden ist.

Die Befehle der JVM

Ab S. 6 im ausgeteilten Papier.

Was bedeuten folgende Worte:

`load` (vom Speicher auf den Stapel kopieren)
`store` (vom Stapel in den Speicher kopieren)
`pop` (vom Stapel entfernen)
`access` (Referenz, Adresse, Zeiger)

Viele Befehle sind nur 1 Byte lang. Andere Befehle belegen 2, 3 oder mehr Bytes.

Das alphabetische Verzeichnis aller Befehle der JVM

Ab S. 8 auf dem ausgeteilten Papier

Notation für die Bedeutung der folgenden Befehle (`iadd`, `isub`, ..., `ladd`, `lsub`, ...) besprechen:

i l d f	add sub mul div rem

`invokestatic` Aufruf einer ***statischen*** Methode (Klassenmethode)
`invokevirtual` Aufruf einer ***nicht-statischen*** Methode (Objektmethode)

9. SU Fr 25.11.11

Test08

Klausur: Fr 03.02.2012, 8 Uhr (leider), **Raum B507**

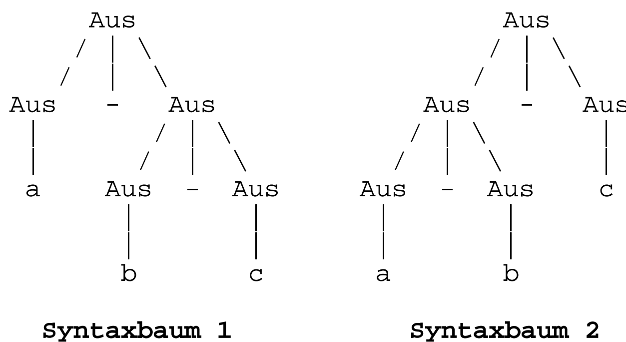
Ein Problem bei (Typ-2-) Grammatiken

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: Wenn eine Grammatik ***mehrdeutig*** ist, dann ist sie automatisch auch ***nicht LR***, d.h. der yacc kann für eine solche Grammatiken keinen Parser generieren.

Anmerkung 2: Es ist (beweisbar) unmöglich, ein Programm zu schreiben, welches von jeder beliebigen Grammatik korrekt herausfinden kann, ob sie mehrdeutig ist oder nicht.

Anmerkung 3: 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 kein Wort mit mehreren Syntaxbäumen findet ist nicht garantiert, dass es kein solches Wort gibt.

Wiederholung: Was ist eine Satzform?

Grammatiken für Ausdrücke (z.B. arithmetische Ausdrücke)

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

Beispiel-01: Eine besonders schlechte Grammatik GAA0 für arithmetische Ausdrücke:

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

Hier sollen `AA`, `LITERAL` und `BEZEICHNER` *Zwischensymbole* sein und jedes Sonderzeichen wie `+ - *` ... etc. sowie das doppelte Sonderzeichen `**` soll ein *Endsymbol* sein.

Die Regeln für `LITERAL` und `BEZEICHNER` sind hier nicht angegeben, wir gehen aber davon aus, dass man aus `LITERAL` `int`-Literale wie z.B. `123` oder `987654321` und aus `BEZEICHNER` Bezeichner von `int`-Variablen wie z.B. `a` oder `x` oder `summe` oder `y17` etc. ableiten kann.

Die Grammatik GAA0 ist in hohem Maße *mehrdeutig*. Z.B. kann man für einen Ausdruck wie `a + b + c + d + e` zahlreiche (verschiedene) Syntaxbäume konstruieren.

Diese Mehrdeutigkeit hängt damit zusammen, dass die Grammatik nichts über die *Bindungsstärke* und die *Assoziativität* der Operatoren `+ - * /` etc. aussagt.

Die folgende Grammatik GAA1 ist deutlich komplizierter als GAA0, aber auch besser ("informativer und eindeutig").

Beispiel-02: Eine Grammatik GAA1 für arithmetische Ausdrücke

```
R01: AA  -> AA  + AA1  -- Der Operator + ist linksassoziativ
R02: AA  -> AA  - AA1  -- Der Operator - ist linksassoziativ
R03: AA  -> AA1

R04: AA1 -> AA1 * AA2  -- Der Operator * ist linksassoziativ
R05: AA1 -> AA1 / AA2  -- Der Operator / ist linksassoziativ
R06: AA1 -> AA1 % AA2  -- Der Operator % ist linksassoziativ
R07: AA1 -> AA2

R08: AA2 -> AA3 ** AA2 -- Der Operator ** ist rechtsassoziativ
R09: AA2 -> AA3

R10: AA3 -> + AA4
R11: AA3 -> - AA4
R12: AA3 -> AA4

R13: AA4 -> ( AA  )
R14: AA4 -> LITERAL
R15: AA4 -> BEZEICHNER

R16-R18: BEZEICHNER -> a | b | c
```

Was für Satzformen kann man allein mit den Regeln R01 bis R03 aus AA ableiten? (`AA1`, `AA1+AA1`, `AA1+AA1+AA1`, `AA1+AA1-AA1`, `AA1-AA1-AA1`, ...)

Was für Satzformen kann man allein mit den Regeln R04 bis R07 aus einem AA1 ableiten? (`AA2`, `AA2*AA2`, `AA2/AA2`, `AA2%AA2`, `AA2*AA2/AA2%AA2*AA2`, ...)

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

Tertius Gaudens ("Da freut sich der Dritte")

10. SU Fr 02.12.11

Test09

Wiederholung: Wichtige Unterschiede zwischen der JVM und älteren Prozessoren

Im Test08 haben alle nur ***relativ unwichtige*** Unterschiede genannt. ***Wichtige*** Unterschiede sind:
- Die JVM ist ***getypt*** (das ist revolutionär!)
- Die JVM ist ***objektorientiert***
- Die JVM ***prüft*** jede Bytecode-Datei, bevor sie sie ausführt (mit einem ByteCodeVerifier)

Die Struktur des Alg-Compilers

In den Übungen sollen Sie einen Compiler für die Sprache `Alg` entwickeln. Eine minimale Version des Compilers ist (in der Datei `Alg11.g`) vorgegeben. Diese minimale Version kann nur Befehlsfolgen übersetzen, bei denen alle Befehle die Form `writeln("ABC");` haben (wobei anstelle von `"ABC"` ein beliebiges String-Literal stehen darf).

Die vorgegebene Datei `Alg11.g` sieht auf den ersten Blick möglicherweise ein bisschen verwirrend aus (zumindest, wenn man die vorher behandelten Gentle-Programme `ahnen01.g`, `ahnen02.g`, ..., `ahnen06.g` und `Alg00.g` (in zwei Varianten: `Str` und `Ide`) nicht alle auswendig gelernt hat :-).

Was sind "die großen Abschnitte" der Datei `Alg11.g`?

1. Abstrakte Syntax

2. Konkrete Syntax

3. Übersetzung und Ausgabe

Was wird in den einzelnen Abschnitten vereinbart und beschrieben?

1. Abstrakte Syntax

Ein paar Gentle-Typen werden vereinbart. Wie heißen diese Typen?

`AS_BefehlsFolge`, `AS_Befehl`, `AS_Ausdruck`, `AS_Wert`, ...

Wozu werden der AS-Typen vereinbart? Wozu dienen sie? Was kann man mit ihnen anfangen?

Mit den Werten der AS-Typen kann man die Teile eines Alg-Programms (Befehlsfolgen, Befehle, Ausdrücke, ...) darstellen, und zwar als Terme/Bäume.

Beispiel-01: Ein `string`-Literal, dargestellt durch eine Wert des Typs `AS_Ausdruck`

```
1 lit(stringW("ABC"))
```

Beispiel-02: Ein `writeln`-Befehl, dargestellt durch einen Wert des Typs `AS_Befehl`:

```
2 writeln(lit(stringW("ABC")), POS)
```

Beispiel-03: Eine Befehlsfolge, dargestellt durch einen Wert des Typs `AS_BefehlsFolge`:

```
3 list(writeln(lit(stringW("ABC")), POS), leer)
```

Die AS-Typen sind anfangs nur "unvollständig definiert". Wenn der Compiler weitere Ausdrücke oder Befehle übersetzen soll, müssen wir diese Typen entsprechend erweitern.

Angenommen, wir wollen auch ***Zuweisungen*** intern darstellen. Welchen Typ müssen wir dann erweitern?

Beispiel-04: Den Typ `AS_Befehl` um Zuweisungen erweitern:

```
4 type AS_Befehl
5    writeln(AS_Ausdruck, POS)
6    ...
7    assign(IDENT, AS_Ausdruck, POS)
```

Achtung: Innerhalb der runden Klammern nach einem Konstruktor wie `writeln` oder `assign` darf man nur ***Namen von Gentle-Typen*** angeben. Namen von ***Prädikaten*** sind hier fehl am Platz.

Anstelle von `IDENT` darf man also nicht etwa `Bezeich` angeben, weil `Bezeich` ein (Token-) Prädikat ist.

Hinweis: Wenn man die AS_Typen "zu sehr erweitert" (d.h. zu viele neue Konstruktoren einführt) dann ist das häßlich, aber im Allgemeinen ***kein Fehler***, den der Gentle-Compiler entdecken und ablehnen würde. Natürlich sollte man nur Konstruktoren einführen, die man braucht (und unbebrauchte wieder entfernen).

2. Konkrete Syntax

Welche Arten von Prädikaten stehen in diesem Abschnitt? `action`-Prädikate? `condition`-Prädikate?

(Nein, sondern `nonterm`- und `token`-Prädikate)

Was leisten diese `nonterm`- und `token`-Prädikate? Was macht der Gentle-Compiler mit ihnen?

(Diese Prädikate beschreiben die Quellsprache des Compilers.
Der Gentle-Compiler erzeugt daraus einen Parser (und "als Assistent des Parsers" einen Lexer))

Was sollen die ***Ausgabe-Parameter*** bewirken, die wir (in runden Klammern) an die `nonterm`-Prädikate schreiben?

Beispiel-05:

```
8    'rule' KS_Ausdruck(-> lit(stringW(S))):
9       StringLiteral(->S)
```

Diese Ausgabe-Parameter sollen
die konkrete Syntax (d.h. Teile eines Quellprogramms)
in abstrakte Syntax (in eine Zwischendarstellung) übersetzen.

Was für Teile eines Quellprogramms können mit der Regel aus **Beispiel-05** erkannt werden?

(String-Literale wie z.B. "ABC")

In was für eine Zwischendarstellung (abstrakte Syntax) wird das String-Literal `"ABC"`
durch die Regel in **Beispiel-05** übersetzt?

(In die Zwischendarstellung `lit(stringW("ABC"))`)

Die Ausgabe-Parameter der `token`-Prädikate sollten schon festliegen, wenn wir die `nonterm`-Prädikate programmierer.

Typischerweise ruft ein `nonterm`-Prädikat ein oder mehrere `token`-Prädikate auf und die Ausgabe des `nonterm`-Prädikats wird dann aus den Ausgaben der `token`-Prädikate zusammengesetzt.

In den Regeln eines `nonterm`-Prädikats verhält sich ein String-Literal (wie z.B. `"int"` oder `"true"`) wie ein `token`-Prädikat ohne Ausgabeparameter. Das `token`-Prädikat `"int"` gelingt, wenn der Lexer im Quellprogramm das Wort `"int"` erkennt.

3. Übersetzung und Ausgabe

Welche Arten von Prädikaten stehen in diesem Abschnitt? `nonterm`-Prädikate? `token`-Prädikate?

(Nein, `action`- und `condition`-Prädikate)

Erinnert sich jemand an Namen von `action`-Prädikaten in diesem Abschnitt?

(`ausBefehlsFolge`, `ausBefehl`, `ausAusdruck`, ...)

Was für Parameter hat `ausBefehlsFolge`? (einen Ein-Parameter vom Typ `AS_BefehlsFolge`)
Was für Parameter hat `ausBefehl`? (einen Ein-Parameter vom Typ `AS_Befehl`)
Was für Parameter hat `ausAusdruck`? (einen Ein-Parameter vom Typ `AS_Ausdruck`)

Beism Schreiben des Compilers (insbesondere beim Übersetzen von Ausdrücken) muss man bestimmte ***Zeiten*** untercheiden, und von verschiedenen Befehlen klar im Kopf haben, zu welcher ***Zeit*** sie ausgeführt werden. Als konkretes Beispiel betrachten wir folgende Situation:

Mit dem Gentle-Compiler `gentle.exe` etc. übersetzen wir
die Datei `Alg11.g` in einen Alg-Compiler `Alg11.exe`.

Mit dem Alg-Compiler `Alg11.exe` etc. übersetzen wird
die Alg-Quelldatei `t11_01` in ein Bytecode-Programm `t11_01.class`.

Vom Java-Interpreter `gij.exe` lassen wir das Bytecode-Programm `t11_01.class` ausführen.
Dieses Programm gibt `"Hallo Welt!"` aus.

Laufzeit von `gentle.exe`		
Compilezeit von `Alg11` (`Alg11.g -> Alg11.exe`)	**Laufzeit** von `Alg11.exe`	
	Compilezeit von `t11_01` (`t11_01 -> t11_01.class`)	**Laufzeit** von `t11_01.class`

Wir müssen also 3 Laufzeiten und 2 Compilezeiten unterscheiden.

Es folgen ein paar Fragen dazu:

Wann wird aus den nonterm-Prädikaten unseres Compilers ein Parser erzeugt?
(Zur Laufzeit von `gentle.exe` gleich Compilezeit von `Alg11`)

Wann wird das `root`-Prädikat unseres Compilers ausgeführt?
(Zur Laufzeit von `Alg11.exe` gleich Compilezeit von `t11_01`)

Wann wird `"Hallo Welt!"` ausgegeben?
(Zur Laufzeit von `t11_01.class`)

Wann werden C-Befehle in eine Datei ausgegeben?
(Zur Laufzeit von `gentle.exe` gleich Compilezeit von `Alg11`)

Wann werden diese C-Befehle ausgeführt?
(Zur Laufzeit von `gentle.exe` gleich Compilezeit von `Alg11`)

Wann werden Jasmin-Befehle in eine Datei ausgegeben?
(Zur Laufzeit von `Alg11.exe` gleich Compilezeit von `t11_01`)

Wann werden diese Jasmin-Befehle ausgeführt?
(Zur Laufzeit von `t11_01.class`)

Zur Entspannung: **Charles Babbage** (1791-1871)

Forschte auf verschiedenen Gebieten. Unternahm mehrere Versuche, mechanische Rechenmaschinen zu bauen. Keine davon wurde funktionstüchtig, aber seine Pläne und Überlegungen dazu waren wichtige Stationen auf dem Weg zu Computern.

1823 Difference Engine no. 1 (in die Entwicklung floßen 17000 Pfund, was etwa dem Preis von zwei Schlachtschiffen entsprach)
1833 Analytical Engine (die sollte sogar schon programmierbar werden)
1847 Difference Engine no. 2 (wurde ab 1985 im Science Museum in London genau nach den Plänen von Babbage gebaut und funktioniert)

Ada Byron, Lady Lovelace (1815-1852), Tochter des berühmten Dichters Lord Byron, arbeitete mit an der Analytical Engine und gilt seitdem als erste Programmiererin. Nach ihr ist die Sprache Ada benannt (ANSI/MIL-STD-1815, nach ihrem Geburtsjahr).

Weitere Erfindungen von Charles Babbage: Kuhfänger (für Lokomotiven, z. B. im Wilden Westen). Er knackte als erster eine Vignére-Verschlüsselung (die vorher als sicher galt). Schrieb das Buch ***Economy of machinery and manufactures***, eine Analyse des Frühkapitalismus und wichtige Quelle für Karl Marx.

11. SU Fr 09.12.11

Test10

Die Struktur des Alg-Compilers (Fortsetzung)

Erinnert sich jemand an Namen von `action`-Prädikaten im 3. Teil des Compilers ("Übersetzen und ausgeben")?

(`ausBefehlsFolge`, `ausBefehl`, `ausAusdruck`, ...)

Was für Parameter hat `ausBefehlsFolge`? (einen Ein-Parameter vom Typ `AS_BefehlsFolge`)
Was für Parameter hat `ausBefehl`? (einen Ein-Parameter vom Typ `AS_Befehl`)
Was für Parameter hat `ausAusdruck`? (einen Ein-Parameter vom Typ `AS_Ausdruck`)

Was macht das Prädikat `ausBefehlsfolge` mit seinem `AS_BefehlsFolge`-Parameter?
(Es übersetzt ihn in Jasmin-Befehle und gibt die aus)

Was macht das Prädikat `ausBefehl` mit seinem `AS_Befehl`-Parameter?
(Es übersetzt ihn in Jasmin-Befehle und gibt die aus)

Was macht das Prädikat `ausAusdruck` mit seinem `AS_Ausdruck`-Parameter?
(Es übersetzt ihn in Jasmin-Befehle und gibt die aus)

Erinnert sich jemand an Namen von `condition`-Prädikaten in diesem Abschnitt?

`(istInt, istBool, istString)`

Was für Parameter haben diese Prädikate?

(Einen Ein-Parameter vom Typ `AS_Ausdruck`)

Wo/wozu werden diese Prädikate benötigt?

An verschiedenen Stellen des Compilers:

Was muss das Prädikat `ausBefehl` machen, wenn ihm der folgende Parameter
(vom Typ `AS_Befehl`) übergeben wird:

```
write(AUS, POS)
```

wobei `AUS` eine Variable vom Typ `AS_Ausdruck` und `POS` eine Variable vom Typ `POS` ist?

Es muss prüfen, von welchem Typ der durch `AUS` dargestellte Ausdruck ist und eine entsprechende Ausgabe-Methode in `ALS3` aufrufen `(pInt, pBool, pString)`

Was muss das Prädikat `ausBefehl` machen, wenn ihm der folgende Parameter (vom Typ `AS_Befehl`) übergeben wird:

```
read(QB)
```

wobei `QB` eine Variable vom Typ `IDENT` ist.

Es muss prüfen, von welchem Typ der Bezeichner `QB` ist und eine entsprechende Eingabe-Methode in `ALS3` aufrufen `(liesInt, liesBool, liesString)`

Merke: Etwas vereinfacht gesagt gilt allgemein:
Mit einer kontextfreien Grammatik kann man die ***Typen von Ausdrücken*** nicht beschreiben und auseinanderhalten. Deshalb beschreibt man alle Ausdrücke (`int`-Ausdrücke, `bool`-Ausdrücke, `string`-Ausdrücke, ...) zusammen, d.h. man stellt sie intern all durch Werte eines einzigen Gentle-Typs (z.B. des Typs `AS_Ausdruck`) dar.

Beim Übersetzen eines Ausdrucks (d.h. eines Wertes des Typs `AS_Ausdruck`) muss man aber unterscheiden, ob es sich um einen `int-`, `bool-` oder `string-Ausdruck` handelt. Deshalb gibt es die `condition`-Prädikate `istInt`, `istBool`, `istString`.

Zur Erinnerung: Wie werden die `bool`-Werte `false` und `true` im Alg-Compiler intern dargestellt? (Durch die `int`-Werte `0` bzw. `1`)

Der Compiler `Alg11.g` (erste Ausbaustufe) kann nur mit sehr ***einfachen Ausdrücken*** umgehen. Was für Ausdrücke sind das?
(Literale der Typen `int`, `bool` und `string` und
Variablen der Typen `int`, `bool` und `string`)

Der Compiler `Alg12.g` (zweite Ausbaustufe) soll auch mit ***komplizierteren Ausdrücken*** umgehen können, d.h. mit Ausdrücken in denen Operatoren wie `or`, `and`, `not`, `<`, `<=`, `==`, ..., `+`, `-`, `*` ... vorkommen.

Damit der Compiler solche komplizierteren Ausdrücke bearbeiten kann, sollten wir als Erstes eine geeignete abstrakte Syntax (eine Zwischendarstellung) für sie festlegen, d.h den Typ `AS_Ausdruck` entsprechend erweitern.

Beispiel-01: Abstrakte Syntax für `or`-Ausdrücke

```
 1 'type' AS_Ausdruck
 2    lit(AS_Wert)
 3    var(IDENT)
 4    or(AS_Ausdruck, AS_Ausdruck, POS)
 5    ...
```

Das Prädikat `ausAusdruck` muss jeden Ausdruck A (d.h. jeden Wert A des Typs `AS_Ausdruck`) in eine geeignete Folge von Jasmin-Befehlen übersetzen. Zu welcher Zeit werden diese Jasmin-Befehle ausgeführt? (Zur Laufzeit eines Alg-Programms, z.B. `t12_01.class`)

Was sollen die Jasmin-Befehle, in die der Ausdruck A übersetzt wurde, machen? (Sie sollen den Wert von A oben auf den Stapel der JVM legen)

Wenn man diese Regel immer klar im Kopf hat, ist das Übersetzen von Ausdrücken (d.h. das Schreiben des Prädikats `ausAusdruck`) gar nicht so schwer.

Beispiel-02: Ausdrücke der Form `or(AS_Ausdruck)` übersetzen und ausgeben

```
 6 'action' ausAusdruck(AS_Ausdruck)
 7    ...
 8    'rule' ausAusdruck(or(AUS1, AUS2, POS)):
 9       aSkom("bool or")
10       ausAusdruck(AUS1)
11       ausAusdruck(AUS2)
12       aS   ("   ior")
13    ...
```

Zu welcher Zeit wird diese Regel ausgeführt?
(Zur ***Laufzeit*** des Compilers `Alg12.exe` gleich ***Compilezeit*** eines Alg-Programms, z.B. `t12_01`)

Was macht diese Regel, ganz allgemein und einfach gesagt?
(Sie gibt eine Folge von Jasmin-Befehlen aus)

Zu welcher Zeit werden diese Jasmin-Befehle ausgeführt?
(Zur Laufzeit des Alg-Programms `t12_01.class`)

Was bewirken die Jasmin-Befehle, die durch den Befehl in Zeile 10 ausgegeben werden?
(Sie legen den Wert des Ausdrucks `AUS1` oben auf den Stapel)

Was bewirken die Jasmin-Befehle, die durch den Befehl in Zeile 11 ausgegeben werden?
(Sie legen den Wert des Ausdrucks `AUS2` oben auf den Stapel)

Was bewirkt der `ior`-Befehl, der durch den Befehl in Zeile 12 ausgegeben wird?
(`xy -> (x or y)`, `x` und `y` müssen `int`-Werte sein, sie werden bitweise mit ***oder*** verknüpft)

Alle anderen komplizierten Ausdrücke werden ebenso einfach übersetzt (nur die einfachen Ausdrücke ***Literal*** und ***Bezeichner*** werden ein bisschen anders übersetzt).

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)

12. SU Fr 16.12.11

Heute Test11

Wer kann in Zukunft ein Notebook oder einen Labtop etc. mitbringen, so dass wir auch im 1. Block an dem Alg-Compiler arbeiten können?

Ein universeller Parser für Typ-1-Grammatiken

Wie dürfen Regeln einer Typ-1-Grammatik aussehen? Welche Einschränkung müssen sie erfüllen? (linke Seite: Eine Satzform, die mind. ein Zwischensymbol enthält, rechte Seite: irgendeine Satzform, aber: Die linke Seite darf nicht kürzer sein als die rechte Seite)

Gegeben eine Typ-2-Grammatik G und ein Wort W. Wie könnten wir herausfinden, ob W zur Sprache von G gehört oder nicht?

Universelle Parser

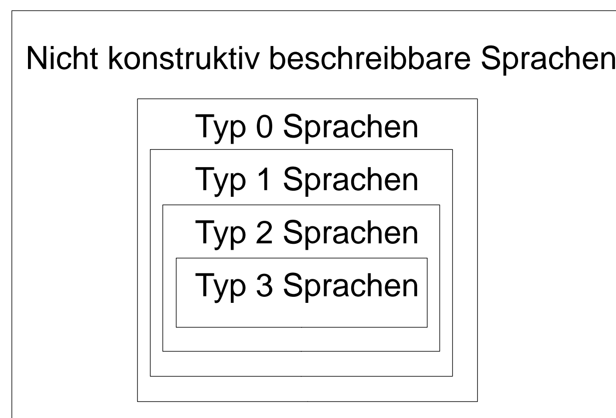
für Typ-3-Grammatiken: Praktisch möglich, sehr schnell
für Typ-2-Grammatiken: Praktisch möglich, häufig schnell genug
für Typ-1-Grammatiken: Theoretisch möglich, praktisch zu langsam
für Typ-0-Grammatiken: Theoretisch (d.h. beweisbar) unmöglich

Grammatiken und Sprachen

Eine formale Sprache heißt **Typ i Sprache** (für i = 0, 1, 2, 3), wenn sie durch eine **Typ i Grammatik** beschrieben werden kann. Häufig meint man zusätzlich: sie kann durch keine Typ i+1 Grammatik (d.h. durch keine ***schwächere*** Form von Grammatik) beschrieben werden.

Eine formale Sprache heißt ***linear*** oder ***regulär***, ***kontextfrei*** bzw. ***kontextsensitiv***, wenn sie durch eine ***lineare*** oder ***reguläre***, ***kontextfreie*** bzw. ***kontextsensitive*** Grammatik (und keine schwächere Form von Grammatik) beschrieben werden kann.

Jede Typ-3-Grammatik ist auch eine Typ-2-Grammatik. Das folgt aus den Definitionen der Einschränkungen für Typ-2-bzw. Typ-3-Grammatiken. Entsprechend gilt: Jede Typ-1-Grammatik ist auch eine Typ-0-Grammatik. Es gibt aber Typ-2-Grammatiken, die **keine** Typ-1-Grammatiken sind (z.B. eine Typ-2-Grammatik mit der der Regel $A \to \varepsilon$, wobei A nicht das Startsymbol ist). Man kann aber beweisen, dass "auf der Ebene der Sprachen" gilt:
Die Menge aller Typ-2-Sprachen ist (echt) in
der Menge aller Typ-1-Sprachen enthalten.
Das folgende Diagramm zeigt, wie die 4 Sprachmengen ineinander enthalten sind:



Um zu zeigen, dass eine bestimmte Sprache S
eine Typ-i-Sprache ist, genügt es,
eine Typ-i-Grammatik für S anzugeben.
Um zu zeigen, dass S **keine** Typ-i-Sprache ist, genügt es ***nicht***, eine bestimmte Zeit nach einer Typ-i-Grammatik für S zu suchen und keine zu finden.
Man muss vielmehr **beweisen**, dass S durch **keine** Typ-i-Grammatik beschrieben werden kann (und da es unendlich viele Typ-i-Grammatiken gibt, ist ein solcher Beweis meistens nicht ganz einfach).

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

L1 = $\{a^n b^m \mid n \geq 1, m \geq 1\}$ L1 ist vom **Typ 3**.

Aufgabe L1: Geben Sie eine Typ-3- Grammatik für die Sprache L1 an.

L2 = $\{a^n b^n \mid n \geq 1\}$ L2 ist vom **Typ 2** (aber nicht vom Typ 3!).

Aufgabe L2: Geben Sie eine Typ-2-Grammatik für die Sprache L2 an.

L3 = $\{w\, w^{-1}$ | w ist eine nicht-leere Folge von a´s und b´s} L3 ist vom **Typ 2** (aber nicht vom Typ 3!).
w^{-1} soll "w rumgedreht" bedeuten. Wenn z.B. w gleich **abb** ist, dann ist w^{-1} gleich **bba**.

Aufgabe L3: Geben Sie eine Typ-2-Grammatik an für die Sprache L3.

L4 = $\{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$ L4 ist vom **Typ 2** (aber nicht vom Typ 3).

Aufgabe L4: Geben Sie eine Typ-2-Grammatik an für die Sprache L4.

L5 = $\{a^n b^n c^n \mid n \geq 1\}$ L5 ist vom **Typ 1** (nicht vom Typ 2!).

L6 = $\{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$ L6 ist vom **Typ 1** (nicht vom Typ 2!).

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 Entspannung: **Kurt Gödel** (1906-1978, Österreich-Ungarn, Princeton, USA)

Studierte Physik und Mathe in Wien, frühe Begabung für Mathe.

1931: "Über formal unentscheidbare Sätze der Principia Mathematica und verwandte Sätze". Die Principia Mathematika (3 Bände, erschienen 1910-1913) war ein philosophisch-mathematisch wichtiges Werk von Bertrand Russell (1872-1970) und Alfred North Whitehead (1861-1947).

Mit diesem Papier zerstöre Gödel die Hoffnung des Mathematikers David Hilbert (1862-1943), alle mathematischen Sätze rein formal aus ***einer*** Basis von Axiomen abzuleiten.

1932: Habilitation in Wien.
1933: Hitler kam an die Macht.
1934: Vorlesungen in Princeton.
1938: "Anschluss" Österreichs an Deutschland.
1940: Auswanderung in die USA, bis 1978 in Princeton, Freund von Einstein. "Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory".

Einer der bedeutendsten Mathematiker des 20. Jahrhunderts. Starb in einer Nervenheilanstalt an Unterernährung, weil er Angst vor einer Vergiftung hatte.

13. SU Fr 23.12.2011

Heute Test12

Eine Typ-1-Grammatik entwickeln

Eine der "einfachsten Typ-1-Sprachen" ist die folgende Sprache L%:

L5 = {$a^n b^n c^n$ | n≥1} L6 ist vom **Typ 1** (nicht vom Typ 2!).

Um für diese Sprache eine (Typ-1-) Grammatik zu entwickeln, kann man wie folgt vorgehen:

1. Für jedes Endsymbol erfinden wir ein entsprechendes Zwischensymbol. Naheliegend sind folgende Zwischensymbole: A, B, C (für die Endsymbole a, b, c).
Wir verbinden diese Zwischensymbole erstmal mit der (etwas vagen) Vorstellung, dass sie "am Ende einer Ableitung durch die entsprechenen Endsymbole ersetzt werden sollen" (d.h. A soll durch a, B durch b und C durch c ersetzt werden).

2. Wir entwickeln Regeln, mit denen man geeignetes "Rohmterial" ableiten kann. Folgende Satzformen ist ein Beispiele für "geeignetes Rohmaterial": `A B c A B c`

Diese Satzform unterscheidet sich von
dem richtigen Wort `a a b b c c` durch folgende Eigenschaften:

- Einige der Symbole sind noch ***Zwischensymbole*** (große Buchstaben) statt ***Endsymbole*** (kleine Buch.)
- Die ***Reihenfolge*** der Symbole ist noch nicht richtig

Deshalb bezeichnen wir solche Satzformen hier als "***Roh***material".

Aufgabe-01: Geben Sie 2 oder 3 weitere Beispiele für ***Rohmaterial-Satzformen*** an

Gruppe 1 von Regeln: Ableitung von Rohmaterial

```
1  S : A B c
2  S : A B c S
```

3. Dann entwickeln wir Regeln, "mit denen ein gutwilliger Ableiter" die Symbole in die richtige Reihenfolge bringen ***kann*** (erst alle a's, dann alle b's, dann alle c's, unabhängig von Groß- und Kleinschreibung).

Gruppe 2 von Regeln: Umordnen ermöglichen

```
3  c A : A c
4  c B : B c
5  B A : A B
```

Schließlich entwickeln wir Regeln, mit denen ein Ableiter die Zwischensymbole (A und B) durch die entsprechenden Endsymbole (a bzw. b) ersetzen kann. Jetzt kommt "der zentrale Trick" dieser Grammatik: Wir formulieren diese Regeln so, dass der Ableiter sie nur dann anwenden kann, wenn er die Symbole vorher in die richtige Reihenfolge gebracht hat (mit den Regeln der Gruppe 2):

Gruppe 3 von Regeln: Kleinmachen ermöglichen, aber nur wenn ...

```
6  B c : b c
7  B b : b b
8  A b : a b
9  A a : a a
```

Aufgabe-02: Leiten Sie aus diesen 9 Regeln das Wort `a a b b c c` ab.

Aufgabe-03: Zu einer kürzeren Grammatik für dieselbe Sprache L5 kann man kommen, wenn man mit den folgenden Rohmaterial-Regeln beginnt:

Gruppe 1 von Regeln: Ableitung von Rohmaterial

```
1  S : a b c
1  S : a B S c
```

Ergänzen Sie die beiden noch fehlenden Regel-Gruppen (Gruppe 2 und Gruppe 3). Wie viele Regeln braucht man jetzt in der Gruppe 2? Und in der Gruppe 3?

Zur Entspannung: **Ein bewiesenermaßen unlösbares Problem**

Beispiel für eine D-Gleichung:

$+7*x^1 - 3*x2 + 5*y^4 - 2*z^5 = 0$

Das Polynom auf der linken Seite der Gleichung darf Potenzen von ***beliebig vielen Variablen*** `x, y, z, x1, x2, ...` aber nur ***ganzzahlige Koeffizienten*** (im Beispiel: +7, `-3, +5, -2`) enthalten. Bei D-Gleichungen interessiert man sich nur für ***ganzzahlige Lösungen***.

Beispiele: D-Gleichungen mit und ohne Lösungen

```
2    x^2 + 2y^2      = 0  // Genau eine Lösung:             x= 0, y=0
3    x^2 - 4         = 0  // Genau zwei Lösungen:           x= 2 und x=-2
4    x^1 + 5y^1 - 8  = 0  // unendlich viele Lösungen, z.B. x= 3, y=1
5    x^2 - y^2 - z^2 = 0  // unendlich viele Lösungen, z.B. x=-5, y=4, z=3
6    x^2 + 5         = 0  // keine Lösung (leicht zu sehen)
7    x^4 - y^4 - z^4 = 0  // keine Lösung ausser der trivialen (schwer zu beweisen)
```

Schön wäre ein Programm, welches von jeder D-Gleichung feststellen kann, ob sie lösbar ist (d.h. mindestens eine Lösung hat) oder nicht. Leider wurde aber der folgender Satz bewiesen:

Satz: Es gibt keinen Algorithmus, der von jeder D-Gleichung korrekt feststellen kann,
ob sie lösbar ist oder nicht.

"D-Gleichungen" werden üblicherweise als "diophantische Gleichungen" bezeichnet, nach dem griechischen Mathematiker Diophant von Alexandrien, der irgendwann zwischen 100 v.Chr. und 350 n.Chr., vermutlich um 250 n.Chr., lebte und 13 Bücher über Arithmetik veröffentlichte, von denen 10 bis heute erhalten geblieben sind.

Lösung-01: Geben Sie 2 oder 3 weitere Beispiele für Rohmaterial-Satzformen an (z.B. `A B c, A B c A B c A B c, A B c A B c A B c A B c, ...`)

Lösung-02: Leiten Sie aus diesen 9 Regeln das Wort `a a b b c c` ab.

```
 1  S                     Regel 2
 2  A B c S               Regel 1
 3  A B c A B c           Regel 3
 4  A B A c B c           Regel 4
 5  A B A B c c           Regel 5
 6  A A B B c c           Regel 6
 7  A A B b c c           Regel 7
 8  A A b b c c           Regel 8
 9  A a b b c c           Regel 9
10  a a b b c c           FERTIG
```

Aufgabe-03: Zu einer kürzeren Grammatik für dieselbe Sprache L5 kann man kommen, wenn am als Gruppe 1 die folgenden Regeln verwendet:

```
 1  S : a b c
 2  S : a B S c
```

Ergänzen Sie die beiden noch fehlenden Regel-Gruppen (Gruppe 2 und Gruppe 3). Wie viele Regeln braucht man jetzt in der Gruppe 2? Und in der Gruppe 3?

Eine typische Rohmaterial-Satzform, die mit den Regeln dieser Gruppe 1 von Regeln abgeleitet werden kann:
`a B a B a b c c c`

Gruppe 2 von Regeln: Umordnen ermöglichen

```
 3  B a : a B
```

Mit dieser Regel kann man aus dem Rohmaterial `a B a B a b c c c` folgende Satzform ableiten:
`a a a B B b c c c`

Gruppe 3 von Regeln: Kleinmachen ermöglichen, aber nur wenn ...

```
 4  B b : b b
```

Insgesamt besteht diese Grammatik nur aus 2 + 1 + 1 gleich 4 Regeln.

Ableitung des Wortes `a a b b c c` mit diesen 4 Regeln:

```
 1  S                     Regel 2
 2  a B S c               Regel 1
 3  a B a b c c           Regel 3
 4  a a B b c c           Regel 4
 5  a a b b c c           FERTIG
```

Grammatiken für die Sprachen L1 bis L6:

Lösung A1: Eine Typ-3 (rechts-lineare-) Grammatik für **L1**:

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

Lösung A2: Eine Typ-2- (kontextfreie-) Grammatik für **L2**:

R1:	S	→	aSb
R2:	S	→	ab

Lösung A2: Eine Typ-2- (kontextfreie-) Grammatik für **L3**:

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

Lösung A3: Eine Typ-2- (kontextfreie-) Grammatik für L3:

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

Lösung A4: Eine Typ-2- (kontextfreie-) Grammatik für L4:

R1	S	→	abcd
R2:	S	→	abScd

Eine Typ-1 (kontextsensitive-) Grammatik für **L5**:

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

Eine Typ-1- (kontextsensitive-) Grammatik für **L6**:

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

14. SU Fr 06.01.12

Heute Test13

Das Papier **T-Diagramme und einen Compiler bootstrappen** austeilen und besprechen

Zur Entspannung: **Christian Morgenstern** (1871 - 1914)
Anto-Logie
Im Anfang lebte, wie bekannt, / als größter Säuger der Gig-ant. ...

15. SU Fr 13.01.12

Heute Test14

Anmerkung zur Korrektur von Test12: Drei Teilnehmern habe ich vorige Woche mündlich bestätigt, dass ihre Typ-2-Grammatik keinen Fehler enthält, obwohl ich Ihnen Punkte abgezogen habe. Vermutlich war der Punkteabzug doch berechtigt (wegen Problemen mit dem Startsymbol).

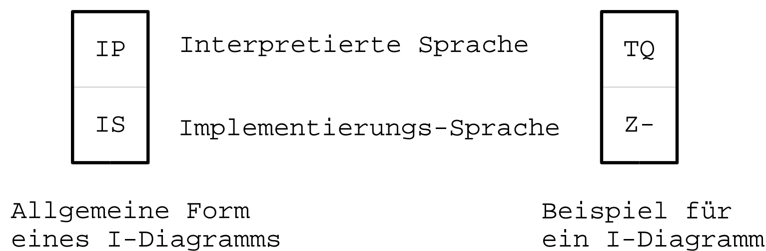
T-Diagramme und einen Compiler bootstrappen, Fortsetzung

Bei einem Compiler sind 3 Sprachen wichtig (QS, ZS, IS).

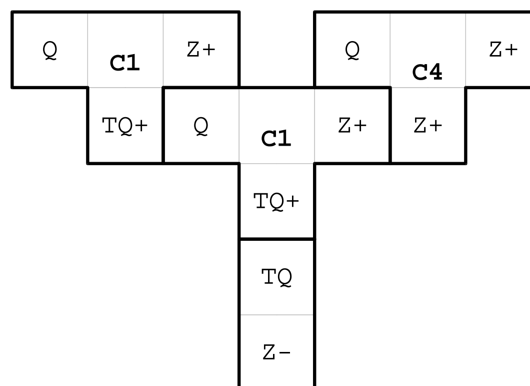
Wie viele Sprachen sind bei einem Interpreter wichtig?
(2 Sprachen:
Die Sprache IP , die vom Intepreter interpretiert wird, und
die Implementierungssprache IS, in der der Interpreter geschrieben ist).

Compiler kann man durch ***T-Diagramme*** beschreiben.
Wie könnten entsprechende ***Diagramme für Interpreter*** aussehen (und heißen)?

I-Diagramme (allgemeine Form und ein Beispiel):



Mit einem TQ-Interpreter auf einer Z-Maschine den Compiler C1 mit sich selbst übersetzen



Ein in der Sprache Z geschriebener TQ-Interpreter ist also eine Alternative zu dem "schnell und schmutzig geschriebenen" Compiler C2, den wir im vorigen SU besprochen haben. Mit dem Interpreter können wir ***in einem Schritt*** den angestrebten Compiler C4 erzeugen.

Zur Entspannung: **Mega-, giga-, und wie geht es dann weiter?**

Ein Blatt mit Vorsilben für kleine und große Zahlen (SI-Präfixe) austeilen.

Man beachte, in welchem Jahr die einzelnen Silben offiziell eingeführt wurden:

Milli- bis mega-:	1795.
Piko- bis tera- :	1960.
Atto- und femto-:	1964
Peta-:	1975
Yocto- bis yotta-:	1991

Mit kontextfreien Grammatiken (ein bisschen) rechnen

Wir entwickeln eine Grammatik für die Sprache aller 2-er-Zahlen, bei denen sich der Rest 0 ergibt wenn man sie durch 3 teilt ("alle durch 2 teilbaren 2-Zahlen").

Zwischensymbole: `RK0`, `RK1`, `RK2` (`RK` soll an "Restklasse" erinnern)
Startsymbol: `RK0`

```
Regeln für alle 2-er-Zahlen, die aus nur einer Ziffer bestehen:

R01  RK0 -> 0
R02  RK1 -> 1

Regeln für alle 2-er-Zahlen, die aus mehr als einer Ziffer bestehen:
(Trick: Wir schreiben erst die rechten Seiten aller Regeln in systematischer
Reihenfolge hin, indem wir jede Restklasse mit jeder 2-er-Ziffer kombinieren.
Danach ermitteln wir die rechten Seiten der Regeln)

R03  RK0 -> RK0 0
R04  RK1 -> RK0 1

R05  RK2 -> RK1 0
R06  RK0 -> RK1 1

R07  RK1 -> RK2 0
R08  RK2 -> RK2 1
```

Was müssen wir an dieser Grammatik ändern, wenn wir alle 2-er-Zahlen beschreiben wollen, bei denen sich ***der Rest 1*** ergibt, wenn man sie durch 3 teilt? (Anstelle von `RK0` müssen wir `RK1` zum Startsymbol machen)

Welche Zahlen kann man bei dieser Grammatik aus dem Zwischensymbol `RK2` ableiten? (Alle 2-er-Zahlen, bei denen sich ***der Rest 2*** ergibt, wenn man sie durch 3 teilt)

Angenommen, wir wollen ganz entsprechend eine Grammatik entwickeln, die
alle 7-er-Zahlen beschreibt, bei denen sich der Rest 0 ergibt, wenn man sie durch 5 teilt.

Wie viele Zwischensymbole brauchen wir und wie nennen wir sie? (5 Stück, `RK0` bis `RK4`)

Wie viele Regeln müssen wir in der ersten Gruppe hinschreiben? (7 Stück)

Wie viele Regeln müssen wir in der zweiten Gruppe hinschreiben? (4 x 7 gleich 28)

Wie viele Regeln hat unsere Grammatik also insgesamt? (7 + 28 gleich 35)

Wie viele Regeln hat eine entsprechende Grammatik, die
alle b-er-Zahlen beschreibt, bei denen sich der Rest 0 ergibt, wenn man sie durch d (wie "Divisor") teilt?
(b x (d + 1))

16. SU Fr 20.01.12

Heute Test 15 (der letzte!)

Warum sollten Informatiker sich mit dem Fach Compilerbau befassen?

1. Das Fach Compilerbau ist ein Beispiel dafür, wie tiefgehende ***theoretische*** (mathematische) ***Erkenntnisse*** die Lösung ***praktischer Probleme*** erleichtern können.

Zu den theoretischen Erlenntnissen gehören solche über
Automaten, formale Sprachen und Grammatiken sowie die Theorie der Berechenbarkeit.

Zu den praktischen Problemen gehören
die Erstellung von effizienten Lexern, Parsern und Codegeneratoren.

2. Bestimmte Compiler demonstrieren die Nützlichkeit von Konzepten, die bei der Erstellung anderer Programme bisher nur zurückhaltend eingesetzt wurden, z.B. Rekursion, funktionale und deklarative Programmierung, allgemein "deklaratives Denken".

3. Mit den Grundtechniken des Compilerbaus kann man (nicht nur Compiler bauern sondern) auch andere Aufgaben erledigen, z.B. die Eingaben eines Programms prüfen, Testdaten erzeugen, Programmtexte analysieren etc.

4. Informatiker haben häufig intensiv mit Compilern zu tun. Deshalb ist es gut, wenn sie deren Grundprinzipien genauer verstehen und "ein Gefühl für die Möglichkeiten und Grenzen" von Compilern entwickeln.

5. Compiler sind besonders wichtige Programme, weil die Korrektheit und Effizienz der meisten Programme von den Compilern abhängt, mit denen sie erzeugt wurden.

6. Moderne Trends bei der Software-Entwicklung (Model Driven Architecture MDA, Generative Software-Entwicklung, Domain Specific Languages DSL) bedeuten: Mehr Compilerbau, weniger "Programmierung von Hand".

Das neue Gentle, ein paar Stichworte

Der neue Compiler wird Gentle-Programme (nicht nur in C-Programme, sondern) wahlweise in
Java-, C# oder C-Programme (möglicherweise auch in weitere Sprachen) übersetzen.

Ein Compiler kann vollständig in der neuen Sprache Gentle geschrieben werden (und muss nicht mehr durch `.t`-, `.b`- und `.c`-Dateien ergänzt werden).

Ein Gentle-PlugIn für Eclipse wird die Entwicklung von DSLs unterstützen (durch die ***automatische Erzeugung*** von syntaxorientierten Editoren, die Calltips, Codevervollständigung, Codefaltung, Syntaxhervorhebung etc. beherrschen).