

Stichworte und Notizen

1. VL Fr 26.09.2008

13 Studenten des Studiengangs TB haben am Anfang des WS08/09 das Wahlpflichtfach Compilerbau (online) belegt.

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 und Aufgaben im WS08/09

Die Lehrveranstaltung wird jeweils am Freitag im 2. und 3. Block stattfinden (wegen der übersichtlichen Teilnehmerzahl kann der Übungstermin im 1. Block wegfallen). Die Vorlesung (im 3. Block) findet wenn möglich im Raum E17 und sonst im Raum B325 statt, die Übung im 2. Block im Raum E16a.

Das WS08/09 ist 19 Wochen lang. In 18 davon findet CB-WP statt (ein Termin am Fr 03.09.08 fällt aus). Am Anfang der ersten *15 Termine* wird jeweils ein kleiner *Test* (ca. 10 Minuten) geschrieben, bei dem Sie ein paar Wiederholungsfragen beantworten sollen. Wenn Sie sich zu Hause mit dem Stoff der vorhergehenden Woche befasst haben, sollte das Beantworten der Fragen leicht fallen.

Die Tests werde ich ("boolesch") mit m.E. bzw. o.E. benoten (mit Erfolg bzw. ohne Erfolg). Sie müssen an *mindestens 12 Tests* mit Erfolg teilgenommen haben, um an der Klausur am Ende des Semesters teilnehmen zu dürfen. Falls Sie mal fehlen spielt es keine Rolle, warum Sie fehlen (Krankheit oder keine Lust oder dringender Arbeitstermin ...). Es empfiehlt sich also, erst mal an allen Tests teilzunehmen, falls man später im Semester mal krank wird oder aus anderem Grund fehlt.

Außerdem müssen Sie im Laufe des Semesters ca. 12 Aufgaben lösen (am besten in Gruppen, die aus 2 Personen bestehen und eine gemeinsame Lösung abgeben bzw. vorzeigen). Bis wann die einzelnen Lösungen fertig sein sollen, wird in den Übungen bekannt gegeben. **Alle Aufgaben müssen bis spätestens am 23.01.2009 fertig sein** (sonst dürfen Sie nicht an der Klausur teilnehmen).

Wie bekommt man eine Note für dieses Fach?

Durch eine Klausur am Ende des Semesters (am 06.02.2009, im 3. Block im Raum D E 15, nahe dem SWE-Labor). Um an der Klausur teilzunehmen, müssen Sie die oben erläuterten zwei Bedingungen erfüllen:

1. Mindestens 12 Tests mit Note m.E. schreiben
2. Alle 12 Aufgaben vor dem 23.01.09 lösen.

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 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 (endliche oder unendliche) Menge von Worten (oder: von Sätzen).

Beispiel-01: Symbole:

0 oder A oder [oder class oder if 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 ...}

Zwischen den Sprachen S02 und S03 gibt es einen subtilen Unterschied. Welche Worte sollen wohl zu S03 gehören, aber nicht zu S02? (Zu S02 sollen keine Worte mit "unnötigen führenden Nullen" gehören. Zu S03 sollen alle 0-1-Folgen gehören).

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 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.

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 ihrerseits ...))).

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 Grammatik-Autor 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.

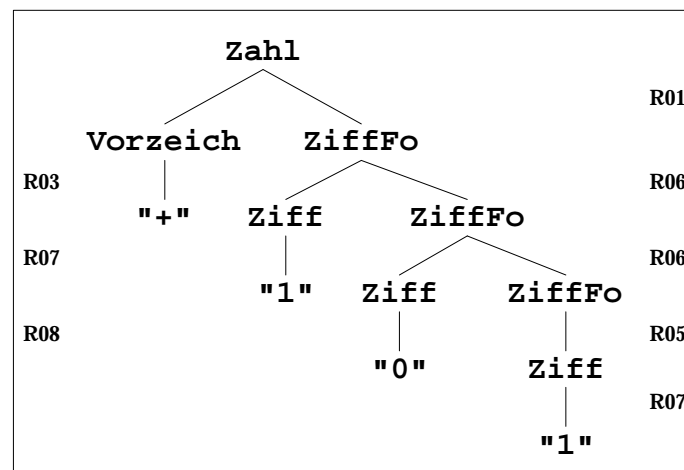
Anmerkung: Dass in der Grammatik G04 die Anzahl der **Zwischensymbole** (4) *gleich* der Anzahl der **Endsymbole** ist und es zu jedem **Zwischensymbol** *genau 2 Regeln* gibt, die mit ihm anfangen, ist reiner Zufall (und bei den meisten Grammatiken anders).

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

Def.: Ein Baum ist entweder leer, oder er besteht 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 rechten 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 *Vorzeichen* das Symbol "+" abgeleitet. Mit der Regel R01 wurde aus dem Symbol *Zahl* die Symbolfolge *Vorzeichen Ziffer* 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*, *Vorzeichen*, *Ziffer* 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 eine 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.

Aufgabe-02: Geben Sie eine Grammatik G_{05} an, die die Sprache S_{05} aller 0-1-Folgen beschreibt:

$S_{03}: \{0\ 1\ 00\ 01\ 10\ 11\ 000\ 001\ 010\ 011\ 100\ 101\ 110\ 111\ 0000\ 0001\ \dots\}$

Verwenden Sie B als Startsymbol von G_{05} (damit man verschiedene Lösungen dieser Aufgabe leichter miteinander vergleichen kann).

Aufgabe-03: Geben Sie eine Grammatik G_{06} an, die die folgende Sprache S_{06} beschreibt:

$S_{06}: \{0\ 1\ 10\ 11\ 100\ 101\ 110\ 111\ 1000\ \dots\}$

S_{06} soll außer dem Wort 0 alle mit 1 beginnenden 0-1-Folgen enthalten.

Verwenden Sie V als Startsymbol von G_{06} .

Aufgabe-04: Geben Sie eine Grammatik G_{07} an, die die folgende Sprache S_{07} beschreibt:

$S_{07}: \{0\ 1\ 01\ 11\ 001\ 011\ 101\ 111\ 0001\ \dots\}$

S_{07} soll außer dem Wort 0 alle mit 1 endenden 0-1-Folgen enthalten.

Verwenden Sie N als Startsymbol von G_{07} .

Aufgabe-05: Geben Sie eine Grammatik G_{08} an, die die Sprache S_{08} 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 G_{08} .

Aufgabe-06: Geben Sie eine Grammatik G_{09} an, die die folgende Sprache S_{09} beschreibt:

$S_{09}: \{\text{mutter vater grossmutter grossvater urgrossmutter urgrossvater ... ururururgrossmutter ... urururururgrossvater ...}\}$

Verwenden Sie $ahne$ als Startsymbol von G_{09} .

Eine Lösung zu Aufgabe-02 (es gibt noch mehrere andere korrekte Lösungen):

R01: B : "0"
R02: B : "1"
R03: B : "0" B
R04: B : "1" B

Eine Lösung zu Aufgabe-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 R01, sondern setzen die Nummerierung mit R05, R06, ... etc. fort.

R05: V : "0"
R06: V : "1"
R07: V : "1" B

Eine Lösung zu Aufgabe-04:

R08: N : "0"
R09: N : "1"
R10: N : B "1"

Eine Lösung zu Aufgabe-05:

R11: 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 hinschreiben").

Eine Lösung zu Aufgabe-06 (es gibt noch viele andere korrekte Lösungen):

R01: ahne : ahne1
R02: ahne : ahne2
R03: ahne : ahne3
R04: ahne1 : "mutter"
R05: ahne1 : "vater"
R06: ahne2 : "gross" ahne1
R07: ahne3 : "ur" ahne2
R08: ahne3 : "ur" ahne3

2. VL Fr 10.10.08

A. Test 1

B. Organisation

In der letzten Lehrveranstaltung haben wir schon die **Teile 1 und 2 der Aufgabe 1** bearbeitet.

Teil 1: Wir haben eine *Grammatik* für die Sprache {mutter, vater, grossmutter, grossvater, urgrossmutter, ...} entwickelt, sie in der Sprache Gentle notiert und daraus (mit dem Gentle-System) einen *Parser* erzeugt.

Teil 2: Wir haben den (in Gentle geschriebenen) Parser zu einem *Compiler* erweitert, der die Worte seiner Quellsprache in natürliche Zahlen {1, 2, 3, ...} übersetzt.

Heute wollen wir die **Teile 3 bis 5 der Aufgabe 1** bearbeiten. All diese Teile haben Ähnlichkeit mit **Teil 2:** Der (in Gentle geschriebene) Parser aus **Teil 1** soll auf verschiedene Weise zu verschiedenen Compilern ergänzt werden.

Außerdem soll das Typsystem der Sprache Gentle besprochen werden.

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.).

Ein Gentle-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)
```

Eine Grammatik GM mit einer unangenehmen Eigenschaft:

R01: EXP : VAR "-" VAR

R02: VAR : "x"

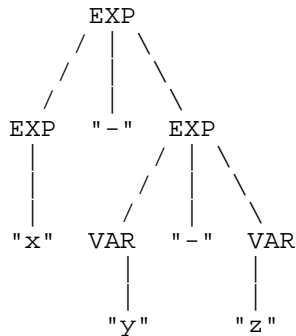
R03: VAR : "y"

R04: VAR : "z"

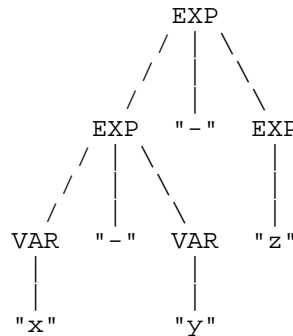
Wie viele Worte enthält die Sprache, die durch diese Grammatik beschrieben wird? (Unendlich viele)

Zu der durch GM beschriebenen Sprache gehört u.a. das Wort $x - y - z$. Leider kann man für dieses Wort mit GM nicht nur *einen*, sondern sogar *zwei* (verschiedene) Syntaxbäume konstruieren, nämlich die folgenden:

Syntaxbaum 1:



Syntaxbaum 2:



Die Grammatik GM ist **mehrdeutig**, weil man z.B. für das Wort $x - y - z$ mehr als einen Syntaxbaum konstruieren kann.

Anmerkung: Für länger Worte wie z.B. $x - x - y - y - z - x$ kann man noch viel mehr als zwei Syntaxbäume konstruieren.

Wichtig: Eine Grammatik beschreibt nicht nur, welche Worte zu einer bestimmten Sprache gehören (und welche nicht), sondern ordnet jedem Wort, welches zur Sprache gehört, eine *syntaktische Struktur* (konkret: einen *Syntaxbaum*) zu. Wenn eine Grammatik *mehrdeutig* ist (d.h. bestimmten Worten mehr als eine syntaktische Struktur zuordnet) ist sie für den Compilerbau (etwas vereinfacht gesagt) unbrauchbar.

Beispiele für syntaktische Strukturen:

Der **Syntaxbaum 1** drückt aus: Das Wort $x - y - z$ besteht aus den drei Teilen "x", "-" und " $y - z$ " und der dritte Teil besteht aus den Teilen "y", "-" und "z".

Der **Syntaxbaum 2** drückt aus: Das Wort $x - y - z$ besteht aus den drei Teilen " $x - y$ ", "-", und "z" und der erste Teil besteht aus den Teilen "x", "-" und "y".

Der **Syntaxbaum 2** entspricht der Konvention, dass der Operator "-" *linksassoziativ* ist (d.h. dass in einem Ausdruck wie $x - y - z$ das y (welches *zwischen zwei Minus-Operatoren* steht) sich zum *linken* Operator "assoziert", was man durch Klammern auch so ausdrücken kann: $(x - y) - z$.

Der **Syntaxbaum 1** beschreibt den Operator "-" als *rechtsassoziativen* Operator und widerspricht damit einer weit verbreiteten Konvention (aber nur einer *Konvention*, keinem *mathematischen Gesetz*).

Die Grammatik GM ist *schlecht*, weil sie *nicht* beschreibt, ob der Operator "-" linksassoziativ oder rechtsassoziativ ist. Die folgende Grammatik GML ist besser, weil sie eindeutig ist (für jedes Wort kann man höchstens *einen* Syntaxbaum konstruieren) und den Operator "-" als linksassoziativen Operator beschreibt:

Die Grammatik GML:

R01: EXP : EXP "-" VAR

R02: VAR : "x"

R03: VAR : "y"

R04: VAR : "z"

Die Regel R01 dieser Grammatik bezeichnet man als *linksrekursiv*:

rekursiv, weil EXP auf der linken und auf der rechten Seite der Regel vorkommt, und *links(-rekursiv)*, weil EXP auf der rechten Seite ganz *links* steht.

Die folgende Grammatik GPR beschreibt einen *rechtsassoziativen* Potenzierungsoperator " * * ":

Die Grammatik GPR:

R05: EXP : VAR " * * " EXP

R06: VAR : "x"

R07: VAR : "y"

R08: VAR : "z"

Die Regel R05 dieser Grammatik bezeichnet man als *rechtsrekursiv*:

rekursiv, weil EXP auf der linken und auf der rechten Seite der Regel vorkommt, und *rechts(-rekursiv)*, weil EXP auf der rechten Seite ganz *rechts* steht.

Die beiden Grammatiken GML und GPR sollen auch veranschaulichen:

Einen *linksassoziativen* Operator wie "-" kann man durch eine *linksrekursive* Regel (wie R01) beschreiben.

Einen *rechtsassoziativen* Operator wie " * * " kann man durch eine *rechtsrekursive* Regel (wie R05) beschreiben.

3. VL Fr 17.10.08

A. Test 2

B. Organisation

Grammatiken für 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
R02: AA -> AA - AA1
R03: AA -> AA1

R04: AA1 -> AA1 * AA2
R05: AA1 -> AA1 / AA2
R06: AA1 -> AA1 % AA2
R07: AA1 -> AA2

R08: AA2 -> AA3 ** AA2
R09: AA2 -> AA3

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

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

Fragen zur Grammatik GAA1:

Welche der Regeln sind	<i>linksrekursiv?</i>	(R01, R02, R04, R05, R06)
Welche Regel ist	<i>rechtsrekursiv?</i>	(R08).
Welche Operatoren sind	<i>linksassoziativ?</i>	(+ - / %).
Welcher Operator ist	<i>rechtsassoziativ?</i>	(**).

Aufgabe-01: Geben Sie eine Grammatik GBA für *boolesche Ausdrücke* an. Verwenden Sie BA als *Startsymbol* und BA1, BA2 etc. als weitere *Zwischensymbole*. Verwenden Sie die folgenden Zeichenfolgen bzw. einzelnen Zeichen als *Endsymbole*:

true false && || ! < <= = >= > ()

Dabei soll && den Und-Operator, || den Oder-Operator und ! den Nicht-Operator bezeichnen. Die sechs Vergleichsoperatoren < <= = >= > sollen nur auf *arithmetische Ausdrücke* (und nicht auf boolesche Ausdrücke) anwendbar sein. Jeder mit ihnen gebildete Ausdruck (wie z.B. $a < b$ oder $x \geq 123$ etc.) ist aber ein *boolescher Ausdruck* (kein arithmetischer Ausdruck).

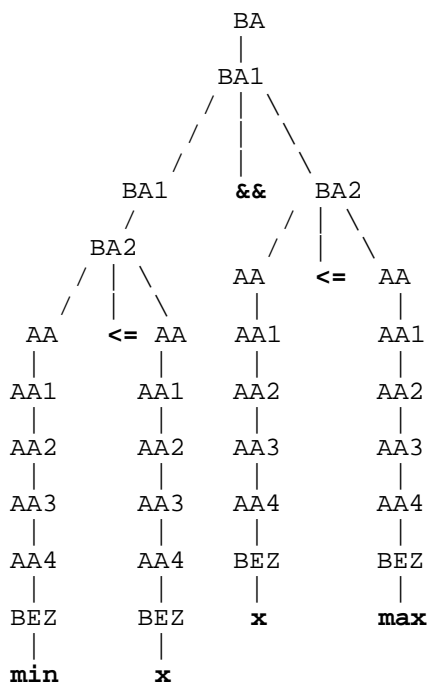
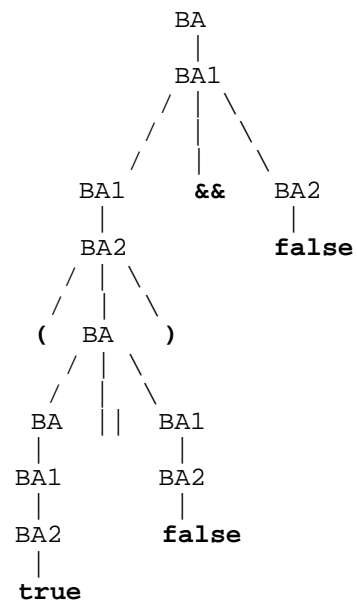
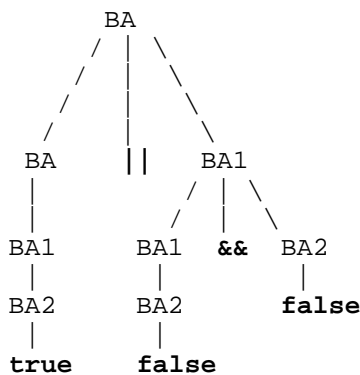
Aufgabe-02: Konstruieren Sie mit Ihrer Grammatik GBA Syntaxbäume für die folgenden booleschen Ausdrücke:

```
true || false && false
( true || false ) && false
min <= x && x <= max
```

Lösung-01: Eine Grammatik GBA für boolesche Ausdrücke:

- 1 BA: BA || BA1
- 2 BA: BA1
- 3
- 4 BA1: BA1 && BA2
- 5 BA1: BA2
- 6
- 7 BA2: true
- 8 BA2: false
- 9 BA2: (BA)
- 10 BA2: ! BA2
- 11
- 12 BA2: AA < AA
- 13 BA2: AA <= AA
- 14 BA2: AA = AA
- 15 BA2: AA >= AA
- 16 BA2: AA > AA

Lösung-02: Syntaxbäume für drei boolesche Ausdrücke



4. VL Fr 24.10.08

A. Test 3

B. Organisation

Der Sinn einer Zwischendarstellung

Im Teil 5 von Aufgabe 1 wird ein Typ `ZAHNE` (wie "Zwischendarstellung für Ahnen") definiert und verwendet. Ein Wort der Quellsprache ("das Quellprogramm") wird vom Compiler zuerst in einen entsprechenden Wert des Typs `ZAHNE` ("in eine Zwischendarstellung") übersetzt. Danach wird diese Zwischendarstellung in ein Wort der Zielsprache ("in ein Zielprogramm") übersetzt und ausgegeben. Warum macht man das (meistens) so?

1. Die Zwischendarstellung ist eine kompakte Darstellung des Quellprogramms. Sie enthält nur noch die wichtigen Informationen über das Quellprogramm, aber keine unwichtigen Einzelheiten. Die Zwischendarstellung wird häufig intensiv bearbeitet. Das geht dann deutlich schneller, als wenn man den Text des Quellprogramms entsprechend bearbeiten würde.

2. Häufig will man mehrere Varianten eines Compilers entwickeln, die sich nur durch die *Zielsprache* unterscheiden. Alle Teile des Compilers, die mit dem Parsen des Quellprogramms, mit dem Aufbau der Zwischendarstellung und mit der Bearbeitung der Zwischendarstellung zu tun haben, können in allen Varianten exakt gleich sein. Nur die Codeerzeugung (Übersetzung der Zwischendarstellung in die Zielsprache und Ausgabe) muss für jede Variante neu geschrieben werden.

Im Teil 5 der Aufgabe 1 wurde die Codeerzeugung durch das Prädikat `aus` erledigt, etwa so:

```
1 'action' aus(ZAHNE)
2 'rule' aus(mu):      PutS("mother") Nl      -- Regel 1
3 'rule' aus(va):      PutS("father") Nl      -- Regel 2
4 'rule' aus(ur(ur(A))): PutS("great")  aus(ur(A)) -- Regel 3
5 'rule' aus(ur(A)):   PutS("grand")   aus(A)   -- Regel 4
```

Die Reihenfolge der Regeln ist wichtig: Eine Regel wird nur dann angewendet, wenn alle davor stehenden Regeln nicht anwendbar sind. Deshalb wird hier nur das letzte `ur` (mit Regel 4) als "grand" und alle vorhergehenden `urs` (mit Regel 3) als "great" ausgegeben.

Aufgabe-01: Vereinbaren Sie ein Prädikat namens `ausF`, welches die Werte des Typs `ZAHNE` auf *Französisch* ausgibt (statt auf Englisch). Hier ein paar französische Vokabeln:

Mutter	mère	Großmutter	grand-mère	Urgroßmutter	arrière-grand-mère
Vater	père	Großvater	grand-père	Urgroßvater	arrière-grand-père

Es ist vermutlich kein besonders übliches und gutes Französisch, aber für französische Muttersprachler doch einigermaßen verständlich, wenn man z.B. Ururgroßmutter mit *arrière-arrière-grand-mère* übersetzt.

Aufgabe-02: Vereinbaren Sie ein Prädikat namens `ausS`, welches die Werte des Typs `ZAHNE` auf *Spanisch* ausgibt (statt auf Englisch oder Französisch). Hier ein paar spanische Vokabeln:

Mutter	madre	Großmutter	abuela	Urgroßmutter	bisabuela
Mutter	padre	Großvater	abuelo	Urgroßvater	bisabuelo

Es ist vermutlich kein besonders übliches und gutes Spanisch, aber für spanische Muttersprachler doch einigermaßen verständlich, wenn man z.B. Ururgroßmutter mit *bisbisabuela* übersetzt.

Parser-Generatoren

Ein *Parser* ist ein Programm (oder Unterprogramm), welches eine Zeichenfolge einliest und prüft, ob sie sich aus einer bestimmten Grammatik ableiten läßt oder nicht.

Parser werden (schon seit den 1970-er Jahren) in aller Regel nicht "von Hand" programmiert, sondern mit Hilfe von sogenannten *Parser-Generator* automatisch erzeugt. Ein Parsergenerator liest eine Grammatik ein und erzeugt daraus einen entsprechenden Parser. *Parser-Generatoren* wurden früher auch als *Compiler-Compiler* bezeichnet (obwohl sie nur Parser generieren, und nicht vollständige Compiler), und diese (etwas übertriebene) Bezeichnung wird auch heute noch häufig verwendet.

Der mit Abstand am weitesten verbreitete Parsergenerator heißt `yacc` (Abkürzung für "Yet Another Compiler-Compiler", zu deutsch etwa: "Noch so ein Compiler-Compiler"). Praktische jede Unix-Distribution (insbesondere jede Linux-Distribution) enthält diesen Parsergenerator. Eine Variante namens `bison` gibt es für Unix und für Windows. Der ursprüngliche `yacc` erzeugt C-Code. Es gibt aber auch Varianten, die Code in anderen Sprachen erzeugen (darunter Fortran, Ada und Java).

Der Parsergenerator (oder: Compiler-Compiler) `yacc` bzw. `bison` hat folgende Eigenschaften:

1. Er kann *nicht* für jede kontextfreie Grammatik einen Parser generieren, sondern nur für sogenannte *LR-Grammatiken*. Dabei bezeichnet LR eine Eigenschaft, die schwer zu erklären und zu erkennen ist.
2. Der `yacc` prüft die Grammatik, die man ihm eingibt. Wenn sie nicht LR ist, gibt er entsprechende Fehlermeldungen aus. Konkret unterscheidet er zwischen Fehlern der Art *shift-reduce conflict* und *reduce-reduce conflict*, aber beide Arten bedeuten: Der erzeugte Parser ist (in aller Regel) nicht korrekt.
3. Eine nicht-LR-Grammatik in eine LR-Grammatik "umzubauen" kann schwer, sehr schwer oder sogar unmöglich sein.
4. Wenn der `yacc` einen Parser erzeugt (und keine Konflikte meldet), dann ist der Parser *sehr schnell*.

Der ursprüngliche `yacc` wurde von S. C. Johnson geschrieben und 1975 veröffentlicht. Seitdem sind Computer um einen Faktor von etwa 10^6 schneller geworden. Konkreter: Was ein Computer von 1975 in etwa 280 Stunden schaffte, schafft ein Computer von 2008 in etwa einer Sekunde. Die Notwendigkeit, besonders schnelle Parser zu erzeugen, ist deutlich geringer geworden. Dass der `yacc` besonders schnelle Parser erzeugt, ist heute also ein geringerer Vorteil als 1975.

Schon 1970 veröffentlichte Jay Earley einen universellen Parser-Algorithmus. "Universell" bedeutet dabei: Dem Algorithmus kann man als ersten Parameter *eine beliebige kontextfreie Grammatik* übergeben (nicht nur eine LR-Grammatik). Als zweiten Parameter übergibt man die Symbolfolge, die geprüft werden soll ("ist sie ableitbar oder nicht?"). 1970 war der Earley-Algorithmus nur von theoretischem Interesse, weil er für praktische Zwecke offenbar viel zu langsam war.

Erst allmählich spricht es sich herum (z.B. an der TFH Berlin :-), dass der Earley-Algorithmus (wenn er von einem modernen PC ausgeführt wird) schnell genug und durchaus für praktische Zwecke geeignet ist. Dass er (immer noch) erheblich langsamer ist als andere Parser, bedeutet in wichtigen Fällen nur, dass er z.B. 100 Millisekunden braucht statt nur 10 Microsekunden.

Anmerkung zum Unterschied zwischen einem *Parser-Generator* und einem *universellen Parser*. Einem Parser-Generator gibt man eine Grammatik ein und er gibt ein (Unter-) Programm aus. Dann läßt man das (Unter-) Programm ausführen und übergibt ihm die zu prüfende Symbolfolge. Einem universellen Parser gibt man gleich die Grammatik und die zu prüfende Symbolfolge ein. Dieser technische Unterschied ist nicht besonders wichtig.

`Accent` (A Compiler Compiler for the Entire Class of Contextfree Grammars) ist ein Parser-Generator, der auf dem Earley-Algorithmus beruht. Man kann mit dem `Accent` also für jede kontextfreie Grammatik einen Parser erzeugen.

Das Gentle-System arbeitet normalerweise mit dem Parser-Generator `yacc` zusammen. Genauer: Der Gentle-Compiler sammelt alle 'nonterm'-Prädikate ein, formt sie ein bißchen um und übergibt sie dem `yacc` bzw. `bison`. Der erzeugt daraus einen Parser (in einer Datei namens `gen.tab.c`). Deshalb gelten für 'nonterm'-Prädikate auch eine paar spezielle Einschränkungen (z.B. dürfen sie nur Ausgabe-Parameter haben, weil der `yacc` mit Eingabe-Parametern nicht klar kommen würde).

Es gibt aber auch eine Gentle-Version, die mit dem `Accent` zusammenarbeitet.

Das Problem der Mehrdeutigkeit

Der Parser-Generator `Accent` ist deutlich bequemer in der Anwendung als der `yacc`: Man braucht keine Angst vor Fehlermeldungen (`shift-reduce conflict` oder `reduce-reduce conflict`) zu haben.

Es bleibt aber noch das Problem der Mehrdeutigkeit. Darüber haben Theoretiker herausgefunden:

Satz-01: Die Eigenschaft der Mehrdeutigkeit einer kontextfreien Grammatik ist *nicht entscheidbar*.

Das bedeutet: Es ist unmöglich *ein* Programm zu schreiben, welches *von jeder kontextfreien Grammatik* feststellen kann, ob sie mehrdeutig ist oder nicht.

Andererseits weiß man:

Satz-02: Die Eigenschaft LR einer kontextfreien Grammatik ist *entscheidbar*.

Z.B. ist der `yacc` ein Programm, welches von jeder kontextfreien Grammatik feststellen kann, ob sie LR ist oder nicht.

Satz-03: Wenn eine Grammatik mehrdeutig ist, dann ist sie nicht LR.

Der `yacc` meldet bei mehrdeutigen Grammatiken also immer Konflikte und kann dann keinen korrekten Parser erzeugen.

Der `Accent` verhält sich wie folgt:

1. Standardmäßig (engl. *by default*) werden Mehrdeutigkeiten nach einem festen Schema entschieden.
2. Mit den Optionen `%nodefault` und `%default` kann man das Verhalten des `Accent` für die gesamte Grammatik oder für bestimmte Abschnitte ändern. Wenn das nicht-Standard-Verhalten aktiviert ist und der `Accent` beim Parsen einer Eingabe eine Mehrdeutigkeit entdeckt, gibt er einem umfangreiche Informationen aus und eine genau Anleitung, wie man die Mehrdeutigkeit beseitigen kann (durch Einfügen von Optionen `%prio`, `%short`, `%long` an bestimmten Stellen der Grammatik).

Theoretiker haben zwar bewiesen, dass kein Programm von allen kontextfreien Grammatiken herausfinden kann, ob sie mehrdeutig sind oder nicht, aber das Programm `amber` (Ambiguity Checker) kann dieses Problem für viele praktisch interessante Grammatiken lösen. Dazu leitet `amber` auf systematische (und besonders effiziente) Weise eine Satzform nach der anderen aus der Grammatik ab und prüft von jeder, ob es mehr als einen Syntaxbaum für sie gibt. Bei der Schnelligkeit heutiger Rechner kann `amber` an einem Wochenende viele Millionen Satzformen erzeugen und prüfen und findet so viele Mehrdeutigkeiten.

Parser und Lexer (oder: Scanner)

Ein Parser, der von einem Generator wie `yacc`, `bison` oder `accent` erzeugt wurde, benützt zum Einlesen des Quellprogramms einen so genannten *Lexer* (wird manchmal auch *Scanner* genannt). Das folgende Diagramm soll die Aufgabenverteilung zwischen Lexer und Parser veranschaulichen:



Der Lexer sucht im Quellprogramm nach *Lexemen*. Ein Lexem ist ein einzelnes Zeichen oder eine Folge von Zeichen, "die eng zusammengehören".

Beispiele für Lexeme:

Schlüsselworte wie `if` `while` `class` etc.

Namen, die der Programmierer erfunden hat, wie `summe` `x_27` `MeineSammlung` etc.

Literale wie `123` `78.56e+3` `'A'` `"Hallo"` etc.

Einzelne Zeichen wie `(` `)` `[` `]` `,` `;` `.` `+` `*` `/` `%` etc.

Der Lexer kennt alle erlaubten Lexeme. Wenn er ein Zeichen entdeckt, welches zu keinem erlaubten Lexem gehört, gibt er eine Fehlermeldung aus. Wenn er ein erlaubtes Lexem erkennt, erzeugt er ein entsprechendes *Token*. Ein Token ist eine kleine Datenstruktur, die der Parser leichter und schneller bearbeiten kann als Zeichenketten unterschiedlicher Länge. Typischerweise besteht ein Token nur aus einem `int`-Wert oder aus einem `int`-Wert und einer Adresse.

Bestimmte Lexeme erkennt der Lexer, überliert sie aber nur, ohne ein Token daraus zu machen, z.B. Kommentare und transparente Zeichen (engl. `white space`) wie Blanks und Tab-Zeichen, die zwischen anderen Lexemen erlaubt sind.

Die Token, die der Lexer erzeugt, entsprechen genau den *Endsymbolen* der Quellsprache des Compilers. Der Parser sieht und verarbeitet also keine Zeichenfolge, sondern eine Token-Folge, d.h. eine Folge von Endsymbolen.

Der Parser versucht, aus der Folge von Endsymbolen (Token-Folge) einen Syntaxbaum zu konstruieren. Wenn ihm das nicht gelingt, meldet er entsprechende Syntaxfehler.

Auch Lexer werden heute nicht mehr von Hand geschrieben, sondern mit einem Lexer-Generator aus einer relativ abstrakten Spezifikation erzeugt. Am weitesten verbreitet ist der Lexer-Generator `lex` (bzw. eine Variante namens `flex`). Die vom `yacc/bison` erzeugten Parser können gut mit einem vom `lex/flex` erzeugten Lexer zusammenarbeiten.

Lexen und Parsen bei Gentle

Zum Lexen und Parsen benützt das Gentle-System bewährte Werkzeuge. Der Gentle-Compiler sammelt alle `'nonterm'`-Prädikate eines Gentle-Programms, formt sie ein bisschen um und übergibt sie einem Parsergenerator. Bei Gentle-97 ist das der `yacc` bzw. der `bison`. Gentle-2000 kann wahlweise mit dem `yacc/bison` oder mit dem `accent` zusammenarbeiten.

Ausserdem sammelt der Gentle-Compiler alle String-Literale wie `"mutter"` oder `"John"` und alle `'token'`-Prädikate zusammen und übergibt sie dem Lexer-Generator `lex` bzw. `flex` (unter Linux bzw. Windows).

5. VL Fr 31.10.08

A. Test 4

B. Organisation

Verschiedene Arten von Grammatiken: Die Chomsky-Hierarchie

Konvention (für die Notation von Übungsgrammatiken): Jeder *Großbuchstabe* ist ein *Zwischensymbol*, jeder *Kleinbuchstabe*, jede *Ziffer* und einige *Sonderzeichen* sind *Endsymbole*.

Def. Satzform: Eine Satzform ist eine Folge von Zeichensymbolen und/oder Endsymbolen.

Eine Satzform kann also ganz aus Zeichensymbolen, ganz aus Endsymbolen oder aus einer Mischung von Zwischen- und Endsymbolen bestehen. Eine Satzform kann auch leer sein (d.h. aus 0 Symbolen bestehen). Die leere Satzform wird häufig mit dem Namen epsilon (oder dem griechischen Buchstaben ϵ) bezeichnet (soll an das englische Wort empty erinnern).

Beispiele für Satzformen: AbCdEf (3 Zwischen- und 3 Endsymbole, ABCdef (ebenso), AAA (drei Zeichensymbole), aaa (drei Endsymbole), A (ein Zeichensymbol), a (ein Endsymbol), epsilon (die leere Satzform).

Def. Regel einer allgemeinen Chomsky-Grammatik: Eine Regel besteht aus zwei Satzformen (die man auch als linke Seite und rechte Seite der Regel bezeichnet), die durch ein Trennzeichen (z.B. ein Doppelpunkt : oder ein Pfeil \rightarrow oder ...) voneinander getrennt sind. Die linke Satzform muss mindestens ein Zeichensymbol enthalten.

Beispiele für eine allgemeine Chomsky-Regeln:

R01: AbbCd \rightarrow bbbAbB
 R02: AB \rightarrow ab
 R03: A \rightarrow bcd
 R04: abAa \rightarrow epsilon

Gegenbeispiele für allgemeine Chomsky-Regeln:

R05: abc \rightarrow bbbAbB
 R06: epsilon \rightarrow Ab

Def. Abschließende Regel: Die linke Seite besteht aus genau *einem* Zeichensymbol. Die rechte Seite enthält *keine* Zeichensymbole.

Beispiele für abschließende Regeln:

R10: A \rightarrow abc
 R11: B \rightarrow a
 R12: C \rightarrow epsilon

Gegenbeispiele für abschließende Regeln:

R13: Ab \rightarrow abc
 R14: CD \rightarrow abc
 R15: A \rightarrow bcD

Def.: Linkslinere Regel: Die linke Seite besteht aus genau *einem* Zeichensymbol. Die rechte Seite *beginnt* mit einem Zeichensymbol und enthält keine weiteren Zeichensymbole.

Beispiele für linkslinere Regeln:

R20: A \rightarrow Cabc
 R21: B \rightarrow Babc
 R22: C \rightarrow D

Gegenbeispiele für linkslinere Regeln:

R23: Ab \rightarrow Aabc

R24: BC -> Aabc
 R25: D -> aBc
 R26: E -> bcA
 R27: F -> AbBcd
 R28: G -> AB
 R29: H -> ab

Def.: Rechtslineare Regel: Die linke Seite besteht aus genau *einem* Zwischensymbol. Die rechte Seite *endet* mit einem Zwischensymbol und enthält keine weiteren Zwischensymbole.

Beispiele für rechtslineare Regeln:

R30: A -> abcC
 R31: B -> abcB
 R32: C -> D

Gegenbeispiele für rechtslineare Regeln:

R33: Ab -> abcA
 R34: BC -> abcA
 R35: D -> aBc
 R36: E -> Abc
 R37: F -> bBcdA
 R38: G -> AB
 R39: H -> ab

Def. Kontextfreie Regel: Die linke Seite besteht aus genau *einem* Zwischensymbol (die rechte Seite ist beliebig).

Beispiel für kontextfreie Regeln:

R40: A -> ABcdEFgh
 R41: B -> CCCddd
 R42: C -> D
 R43: D -> a
 R44: E -> epsilon

Gegenbeispiele für kontextfreie Regeln:

R45: Ab -> ABcdEFgh
 R46: CD -> ABcdEFgh

Anmerkung: Die Bezeichnung "kontextfrei" klingt *irreführend positiv* ("frei vom Zwang eines Kontextes" oder so ähnlich). Besser wäre: "kontextunfähig", weil man mit solchen Regeln keine Kontextbedingungen beschreiben kann (obwohl man das gern tun würde). Kontextbedingungen werden später behandelt).

Def. Kontextsensitive Regel: Die rechte Seite darf nicht kürzer sein als die linke (gemessen in "Anzahl der Zwischen- und End-Symbole", z.B. hat die Satzform $AbcDEf$ die Länge 6).

Beispiele für kontextsensitive Regeln:

R50: AbCd -> eFgHiJ
 R51: BBcc -> cBBc
 R52: C -> D
 R53: D -> a

Gegenbeispiele für kontextsensitive Regeln:

R54: AbCd -> eFg
 R55: C -> epsilon

Def. Typ-3-Grammatik (reguläre Grammatik):

Entweder gilt: Alle Regeln sind rechtslinear oder abschließend,
 oder es gilt: Alle Regeln sind linkslinear oder abschließend.

Def. Typ-2-Grammatiken (kontextfreie Grammatiken, eigentlich: kontextunfähige Grammatiken):
Alle Regeln sind kontextfrei.

Def. Typ-1-Grammatiken (kontextsensitive Grammatiken)
Alle Regeln sind kontextsensitiv.

Def. Typ-0-Grammatiken (allgemeine Chomsky-Grammatiken)
Keine Einschränkungen, allgemeine Chomsky-Regeln sind erlaubt.

Typ-0-Grammatiken sind "am stärksten" und Typ-3-Grammatiken sind "am schwächsten". Damit ist Folgendes gemeint:

Sei Typ-0-Sprachen die Menge aller Sprachen, die man mit einer Typ-0-Grammatik beschreiben kann.

Sei Typ-1-Sprachen die Menge aller Sprachen, die man mit einer Typ-1-Grammatik beschreiben kann.

Sei Typ-2-Sprachen die Menge aller Sprachen, die man mit einer Typ-2-Grammatik beschreiben kann.

Sei Typ-3-Sprachen die Menge aller Sprachen, die man mit einer Typ-3-Grammatik beschreiben kann.

Dann gilt:

Typ-0-Sprachen \supseteq Typ-1-Sprachen \supseteq Typ-2-Sprachen \supseteq Typ-3-Sprachen

Jede der vier Sprachmengen enthält abzählbar unendlich viele Elemente (d.h. Sprachen), aber anschaulich gesprochen ist die Menge **Typ-0-Sprachen** am größten und die Menge **Typ-3-Sprachen** am kleinsten. Das soll jetzt anhand einiger konkreter Sprachen illustriert werden.

Als Hilfsmittel brauchen wir aber noch eine spezielle Notation zur Beschreibung von bestimmten formalen Sprachen (deren Worte nur aus Folgen von a's, b's und c's bestehen):

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

Hier soll a^n einfach eine Folge von genau n vielen a's bedeuten und b^m eine Folge von m vielen b's. Die Sprache $L1$ kann man etwas ungenauer, aber "konkreter und direkter" so beschreiben:

$$L1 = \{ ab, abb, abbb, abbbb..., aab, aabb, aabbb, aabbbb \dots, aaab, aaabb, aaabbb, aaabbbb \dots \}$$

Die folgende Sprache $L2$ ist eine Teilmenge von $L1$:

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

Ungenauer, aber konkreter und direkter beschrieben:

$$L2 = \{ ab, aabb, aaabbb, aaaabbbb, \dots \}$$

Die Worte der Sprache $L3$ enthalten nicht nur a's und b's, sondern auch noch c's:

$$L3 = \{ a^n b^n c^n \mid n \geq 1 \}$$

Ungenauer, aber konkreter und direkter beschrieben:

$$L3 = \{ abc, aabccc, aaabbbccc, aaaabbbbcccc, \dots \}$$

Die folgenden Sprachen $L4$ und $L5$ könnte man (mit viel Aufwand) präzise beschreiben, sie werden hier aber nur kurz angedeutet:

$$L4 = \{ \text{Alle haltenden Java-Programme} \}$$

$$L5 = \{ \text{Alle nicht-haltenden Java-Programme} \}$$

Es folgen hier ein paar mathematische Sätze, einige mit, die anderen ohne Beweis.

Satz 1: Die Sprache $L1$ ist so einfach, dass man sie durch eine **Typ-3-Grammatik** (eine reguläre Grammatik) beschreiben kann.

Beweis: Die folgende Typ-3-Grammatik $G1$ (mit 3 rechtslinearen und einer abschließenden Regel), beschreibt die Sprache $L1$:

R01: $A \rightarrow aA$
R02: $A \rightarrow aB$
R03: $B \rightarrow bB$
R04: $B \rightarrow b$

Mit den Regeln R01 und R02 kann man alle Satzformen der Form a^nB ableiten. Mit den Regeln R03 und R04 kann man aus dem B beliebig viele b's (aber nicht weniger als eins) ableiten.

Genau so gut wie die rechts-lineare Grammatik G1 hätte man auch eine links-lineare Grammatik angeben können (die nur links-lineare und abschließende Regeln enthält).

Satz 2: Die Sprache L2 ist schon so kompliziert, dass man sie durch *keine* Typ-3-Grammatik beschreiben kann.

Ein Beweis für diesen Satz wird hier nicht angegeben.

Satz 3: Die Sprache L2 ist aber immerhin noch so einfach, dass man sie durch eine Typ-2-Grammatik (eine kontextfreie Grammatik) beschreiben kann.

Beweis: Hier eine Typ-2-Grammatik G2 für die Sprache L2:

R01: $S \rightarrow ab$
R02: $S \rightarrow aSb$

Beweis: Um das Wort $a^n b^n$ abzuleiten, muss man $n-1$ Mal die Regel R02 und dann einmal die Regel R01 anwenden.

Satz 4: Die Sprache L3 ist so kompliziert, dass man sie durch *keine* Typ-2-Grammatik beschreiben kann.

Ein Beweis für diesen Satz wird hier nicht angegeben (er ist auch nicht ganz einfach und wird meist so geführt, dass man zuerst einen Hilfssatz namens *Pumping-Lemma* beweist).

Satz 5: Die Sprache L3 ist aber immerhin noch so einfach, dass man sie durch eine Typ-1-Grammatik (eine kontextsensitive Grammatik) beschreiben kann.

Ein solche Typ-1-Grammatik werden wir später in diesem Semester entwickeln.

Zur Abrundung des Bildes hier noch zwei Sätze ohne Angabe eines Beweises:

Satz 6: Die Sprache L4 *kann man* mit einer Typ-0-Grammatik beschreiben.

Satz 6a: Man kann ein Java-Programm schreiben, welches der Reihe nach alle Sätze der Sprache L4 ausgibt. Dieses Programm wird natürlich nie fertig sein, da es unendlich viele haltende Java-Programme gibt.

Satz 7: Die Sprache L5 kann man mit *keiner* Typ-0-Grammatik beschreiben (und in einem bestimmten Sinne auch nicht mit irgendeinem anderen "konstruktiven Formalismus").

Satz 7a: Man kann *kein* Java-Programm schreiben, welches der Reihe nach alle Sätze der Sprache L5 ausgibt.

Geschichtliche Anmerkung: Die ersten Computer wurden in den 1940-er Jahren (während des zweiten Weltkriegs) gebaut. Aber schon etwa 20 bis 30 Jahre vorher haben Mathematiker so ähnlich Sätze wie Satz 7a bewiesen. Seitdem weiß man, dass es Probleme gibt, die man mit keinem Computer lösen kann (egal wie groß und schnell der Computer ist).

6. VL Fr 07.11.08

A. Test 5

B. Organisation

Eine Typ-1-Grammatik für die Sprache $L3 = \{ a^n b^n c^n \mid n \geq 1 \}$

Zur Erinnerung: Von der Sprache $L3$ kann man beweisen, dass sie durch keine Typ-2-Grammatik beschrieben werden kann. Die folgende Typ-1-Grammatik soll deutlich machen, was und warum Typ-1-Grammatiken "mehr können" als Typ-2-Grammatiken.

Die Regeln der Grammatik $G3$ (für die Sprache $L3$) werden hier in 3 Gruppen aufgeteilt und mit ein paar informellen Kommentaren versehen:

Gruppe 1: Rohmaterial ableiten

R01: $S \rightarrow A B c$

R02: $S \rightarrow A B S c$ -- Typische Satzform: $A B A B A B c c c$

Gruppe 2: Umordnen der Symbole ermöglichen

R03: $B A \rightarrow A B$ -- Typische Satzform: $A A A B B B c c c$

Gruppe 3: Aus Gross mach Klein (aber nicht immer!)

R04: $B c \rightarrow b c$

R05: $B b \rightarrow b b$

R06: $A b \rightarrow a b$

R07: $A a \rightarrow a a$ -- Typische Satzform: $a a a b b b c c c$

Anmerkung: *Syntaxbäume* kann man nur aus Typ-2- und Typ-3-Grammatiken konstruieren, aber nicht aus Typ-1- und Typ-0-Grammatiken. Regeln mit mehr als einem Symbol auf der linken Seite (wie z.B. R03 bis R07) passen nicht zu Bäumen. Aus allen Grammatiken kann man aber Worte *ableiten*, wie das folgende Beispiel zeigen soll.

Beispiel: Eine Ableitung für das Wort $a a a b b b c c c$ aus der Grammatik $G3$:

1	<u>S</u>	-- R02
2	A B <u>S</u> c	-- R02
3	A B A B <u>S</u> c c	-- R01
4	A B A B A B <u>S</u> c c c	-- R03
5	A <u>B A</u> A B B c c c	-- R03
6	A A <u>B A</u> A B c c c	-- R03
7	A A A B B <u>B c</u> c c	-- R04
8	A A A B <u>B b</u> c c c	-- R05
9	A A A <u>B b</u> b c c c	-- R05
10	A A <u>A b</u> b b c c c	-- R06
11	A <u>A a</u> b b b c c c	-- R07
12	<u>A a</u> a b b b c c c	-- R07
13	a a a b b b c c c	

Evtl. Papier *Unlösbarkeit des Halte-Problems (für Java-Prozeduren, Version 2)* verteilen und besprechen.

Lösen Sie jetzt die Aufgabe 12 Eine Typ-1-Grammatik für die Sprache DOPPELT.

7. VL Fr 14.11.08

A. Test 6

B. Organisation: Alle Gruppen sollten die Aufgaben 1 bis 3 und 12 fertig (und einen Gruppen-Namen für sich gewählt) haben.

Universelle Parser für bestimmte Typen von Grammatiken

Grammatik-Typ	Universeller Parser möglich?	Geschwindigkeit
Typ-3 (reguläre G.)	ja	sehr schnell
Typ-2 (kontextfreie G.)	ja	(meistens) schnell
Typ-1 (kontextsensitive G.)	ja	(meistens) sehr langsam
Typ-0 (allgemeine G.)	nein	---

Wie funktioniert ein universeller Parser für Typ-1-Grammatiken?

Gegeben ein Typ-1-Grammatik G erzeugt dieser Parser den *Baum aller möglichen Ableitungen* "Schicht für Schicht" (oder: Breite zuerst, nicht Tiefe zuerst, engl. breadth first, not depth first). Jeder Ast dieses Baums ist eine Ableitung. Da dieser Baum im allgemeinen unendlich tief ist, kann man ihn nicht vollständig erzeugen. Aber man kann ihn (zumindest theoretisch) bis zu jeder endlichen Tiefe t erzeugen (z.B. bis zu einer Tiefe von t gleich 100 oder t gleich 850 etc.).

Wenn man in diesem Baum einem Ast folgt (d.h. einer Ableitung folgt), was gilt dann sicherlich für die Satzformen, an denen man vorbeikommt? (Sie werden nie kürzer, sondern bleiben mindestens gleich lang, und wenn man weit genug geht, werden sie länger).

Zur Erinnerung: Eine Ableitung ist eine Folge von Satzformen (die bestimmte Bedingungen erfüllen muss). Man entwickelt jede Ableitung so weit, bis die Satzform länger ist als das zu parsende Wort.

Worher kommt die Bezeichnung "kontextsensitiv"

1. Zu den meisten Programmiersprachen gehören nur Programme, die bestimmte Kontextbedingungen erfüllen (z.B. "Jede Variable, die benutzt wird, muss auch vereinbart werden" oder "Jedes Unterprogramm, welches aufgerufen wird, muss auch vereinbart werden" oder "Jedes Unterprogramm, welches mit n formalen Parametern vereinbart wurde, muss auch mit n aktuellen Parametern aufgerufen werden" etc.).

Solche Kontextbedingungen kann man mit kontextfreien Grammatiken *nicht* beschreiben. Einige dieser Regeln kann man mit kontextsensitiven Grammatiken beschreiben.

2. Eine typische kontextsensitive Regel:

KS1: $a B C d e F \rightarrow a B X Y z e F$

Diese Regel erlaubt es, aus der Satzform $C d$ die Satzform $X Y z$ abzuleiten, aber nur im Kontext von $a B$ (links) und $e F$ (rechts). Dieser Kontext muss vorhanden sein, wird durch eine Anwendung der Regel aber nicht verändert.

Grundbegriffe der Programmierung

Es gibt drei Arten von Befehlen (die ein Programmierer in ein Programm schreiben kann):

BA1: **Vereinbarungen** ("Erzeuge ... ", z.B. eine Variable, ein Unterprogramm, eine Klasse ...).

BA2: **Ausdrücke** ("Berechne den Wert des Ausdrucks ...", z.B. $x + 1$ oder $\sin(y) * \text{PI}$...).

BA3: **Anweisungen** ("Tue die Werte ... in die Wertebehälter ...", z.B. $x = x + 1$);

Es gibt zwei Arten von Unterprogrammen:

UA1: **Funktionen** (die einen Wert berechnen und als Ergebnis liefern).

UA2: **Prozeduren** (die die Inhalte von Wertebehältern verändern).

Jeder Aufruf einer *Funktion* ist ein *Ausdruck*.

Jeder Aufruf einer *Prozedur* ist eine *Anweisung*.

Verschiedene Arten von Programmiersprachen

Art von Progr.-Sprachen	Vereinbarungen	Ausdrücke	Anweisungen	Beispiele für solche Sprachen
Prozedurale PS	ja	ja	ja	Fortran, Cobol, Pascal, C, C++, Java
Funktionale PS	ja	ja	nein	Lisp, Opal, Miranda, Scheme, Haskell
Deklarative PS	ja	nein	nein	Prolog, Gentle

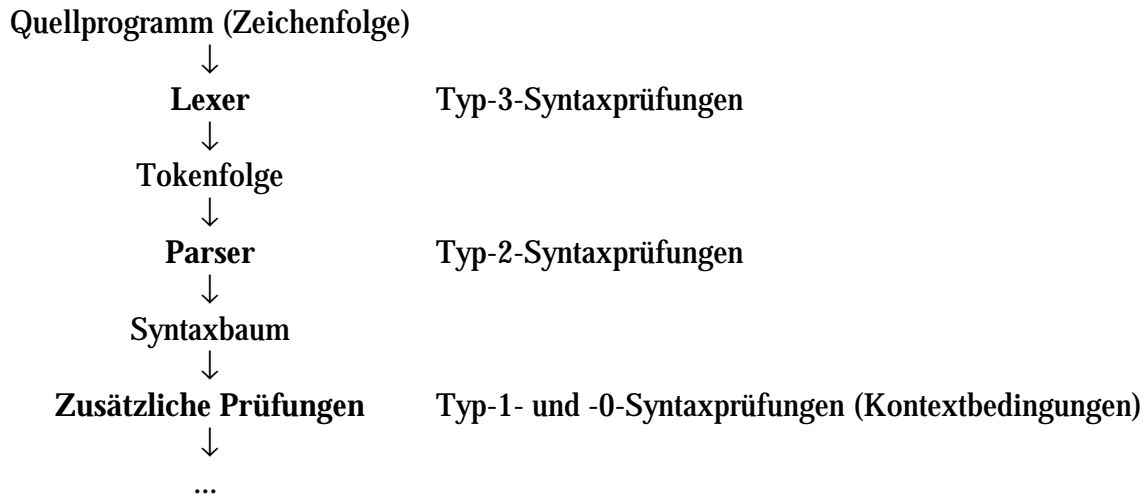
Mit den verschiedenen Arten von Programmiersprachen sind auch verschiedene Arten oder Stile zu programmieren verbunden. Die Sprache C# wurde bereits und Java wird gerade so erweitert, dass ein funktionaler bzw. deklarativer Programmierstil unterstützt wird.

Wer hat Aufgabe 4 schon (weitgehend) fertig?

Entweder Aufgabe 4 fertig machen, oder im mit der Besprechung des Papiers *TermeInGentle* anfangen.

8. VL Fr 21.11.08**A. Test 7**

B. Organisation: Sind alle mit Aufgabe 4 und Aufgabe 12 fertig?!!

Syntaxprüfung durch verschiedene Teile eines Compilers**Die Symboltabelle eines Compilers**

In diese Tabelle werden alle vom Programmierer erfundenen Namen eingetragen, und Hinweise zu ihrer Bedeutung, etwa so:

Name	Bedeutung
summe	Variable, Typ <code>int</code>
add	Funktion, Rückgabotyp <code>long</code> , 2 Parameter vom Typ <code>long</code> , ...
print	Prozedur, 1 Parameter vom Typ <code>String</code> , ...
...	...

In Gentle enthält eine Symboltabelle zu jedem Namen einen 32-Bit-Wert, z.B. einen `int`-Wert oder die Adresse eines Verbundes (a pointer to a record), der Informationen zu dem Namen enthält.

In Aufgabe 5 werden wir alle drei Arten von Syntaxprüfungen durchführen und eine Symboltabelle aufbauen.

Unveränderbare und veränderbare Variablen in Gentle

Alle Variablen, die wir bisher beim Programmieren von Prädikaten verwendet haben, sind *unveränderbar* und *Regel-lokal* (also nur innerhalb "ihrer Regel" sichtbar).

Zur Lösung bestimmter Probleme gibt es aber auch in Gentle *veränderbare* und *globale* Variablen. Auf solche globalen Variablen darf man allerdings nur zwei Befehle anwenden: Den Wert der globalen Variablen lesen (->) und einen Wert in die globale Variable schreiben (<-).

Beispiel: Vereinbarung und Benutzung einer globalen Variablen

```

14 'var' GNR: INT
15 ...
16 'nonterm' -- oder 'action' oder 'condition' oder ...
17   'rule' ...
18     ...
19     GNR -> LNR
20     GNR <- LNR + 1
21     ...

```

Nicht erlaubt sind z.B. folgende Benutzungen der globalen Variablen GNR:

```

22 ...
23 'rule' praed01(GNR -> ...)
24 'rule' praed01(... -> GNR)
25 'rule' praed01(... -> list(GNR, ...))

```

Die Quellsprache für Aufgabe 6

Struktur aller Worte der Quellsprache: `prog ... begin ... end`

Beispiel: `prog anna, bert, carl begin bert, anna, bert end`

Die Namen zwischen `prog` und `begin` repräsentieren *Vereinbarungen*, die Namen zwischen `begin` und `end` repräsentieren *Anwendungen* von Variablen.

Gegenbeispiel 1: `prog anna, anna begin end`

Gegenbeispiel 2: `prog anna begin bert end`

Zwei Arten von Vereinbarungen (z.B. in C/C++): Deklarationen und Definitionen

Eine *Definition* ist Befehl, etwas zu erzeugen.

Eine *Deklaration* ist ein Versprechen des Programmierers an den Compiler, eine bestimmte Größe irgendwo im Programm zu definieren.

Token-Prädikate

Werden in einem Gentle-Programm nur *deklariert* und in separaten Dateien *definiert* (durch reguläre Ausdrücke und C-Befehle).

Aufgabe 6 austeilen, besprechen und bearbeiten

Der Typ `SYMTAB`

Die globalen Variablen `GlobSymTab` und `NaechsterZielbezeichner`

Das `nonterm`-Prädikat `init`

Die Prädikate `HasMeaning` und `HasMeaning2`

Das Prädikat `DefMeaning`

9. VL Fr 28.11.08**A. Test 8****B. Organisation****Die virtuelle Java-Maschine (JVM)**

Papier über die JVM austeilen.

Ein Java-Quellprogramm. Was ist ein statischer Initialisierer (oder: ein Klassen-Konstruktor)?

Die Byte-Code-Datei (.class-Datei), in die er Java-Compiler von Sun das Quellprogramm übersetzt hat.

Auffälliger Unterschied zu .exe-Dateien: Die Namen aller Klassen, Methoden und Attribute stehen genau so in der Bytecode-Datei, wie der Programmierer sie erfunden hat. Nur die Namen von lokalen Variablen (Variablen, die in einer Methode, einem Konstruktor oder einem Block vereinbart wurden) werden vom Compiler in Ganzzahlen übersetzt.

Unterschiede zwischen der JVM und älteren Prozessor-Architekturen:

1. Die JVM ist getypt.
2. Die JVM ist objektorientiert (es gibt z.B. einen Maschinenbefehl namens new, der ein Objekt erzeugt).
3. Die JVM prüft jedes Bytecode-Programm, bevor sie es ausführt. Wenn das Prüfprogramm (der Bytecode Verifier) Fehler entdeckt, wird das Bytecode-Programm nicht ausgeführt.

Ein wichtiger Bestandteil eines Bytecode-Programms: Die Konstantentabelle

Die Konstantentabelle ist keine Reihung, da die einzelnen Einträge unterschiedliche Längen haben.

Viele Einträge in der KT enthalten Indizes, die auf andere Einträge in der KT verweisen.

Die KT ist weitgehend frei von Redundanz. Wenn z.B. die Klasse `StringBuilder` an 27 Stellen einer Klasse benutzt wird, steht der Name `java.lang.StringBuilder` trotzdem nur *einmal* in der KT und an den 27 Benutzungstellen steht nur ein entsprechender Index.

10. VL Fr 05.12.08**A. Test 9****B. Organisation****Die Befehle der JVM**

Die Firma Sun hat die Befehle der JVM genau spezifiziert und als Standard verbreitet, aber (leider) keine dazu passende Assembler-Sprache veröffentlicht.

Jasmin (Java Assembler) ist ein relativ weit verbreiteter Assembler für die JVM.

Übersicht über die JVM: Sie hat einen *Speicher* und einen *Stapel*, aber keine Register. Sie kennt 201 Maschinenbefehle. Sie kann mit Werten der Typen `int`, `long`, `float` und `double` rechnen (aber nicht mit Werten der Typen `char`, `byte`, `short`). Sie kennt keinen Typ `boolean` (hat aber Befehle `iand`, `ior`, `ixor`, `land`, `lor`, `lxor` zum zum bitweisen logischen Verknüpfen von `int`- bzw. `long`-Werten).

Ein paar Befehle der JVM besprechen (S. 6 im Papier "Die virtuelle Java-Maschine").

Namen in einem Bytecode-Programm:

Klassen, **Schnittstellen**, **Methoden** und **Attribute** (engl. fields) haben ganz ähnliche Namen wie in einem Java-Programm. Beispiele:

```
Java:      java.lang.String
Bytecode:  Ljava/lang/String;

Java:      java.lang.String[]
Bytecode   [Ljava/lang/String;

Java:      void Hallo03.druckeVerzierungszeile(int i, char c)
Bytecode:  Hallo03/druckeVerzierungszeile(IC)V
```

Die *Parameter* und *lokalen Variablen* einer Methoden haben keine Namen, sondern Nummern (mit 0 beginnend).

Die Befehle sind möglichst platzsparend organisiert: Die am häufigsten gebrauchten Befehle sind nur 1 Byte lang, weniger häufig gebrauchte sind 2 Byte lang und noch seltener gebrauchte 4 Byte oder länger.

Beispiel-01: Die `aload`-Befehle (access value load) laden eine Referenz aus einer lokalen Variablen auf den Stapel. Es gibt folgende Varianten:

Befehl	bearbeitet die Variable	Länge des Befehls
<code>aload_0</code>	0	1
<code>aload_3</code>	3	1
<code>aload 0</code>	0	2
<code>aload 255</code>	255	2
<code>wide aload 0</code>	0	4
<code>wide aload 65535</code>	65535	4

Die Befehlsgruppen `iload`, `lload`, `fload`, `dload` und `astore`, `istore`, `lstore`, `fstore`, `dstore` kann man in ganz entsprechenden Tabellen darstellen (insgesamt sind das 5 x 2 x 6 gleich 60 Befehle).

Aufgabe-01: Wie viele `iconst`-Befehle gibt es? Welche `int`-Werte kann man damit auf den Stapel laden? Wie lang sind die Befehle?

Anmerkung: Wer die Maschinenbefehle der JVM immer noch nicht auswendig gelernt hat :-), darf in den beiden Verzeichnissen ab S. 8 des ausgeteilten JVM-Papiers nachsehen.

Lösung-01: Es gibt **sieben** `iconst`-Befehle. Mit ihnen kann man die `int`-Werte von -1 bis 5 auf den Stapel laden. Diese Befehle sind nur 1 Byte lang.

Beispiel-02: Befehle zum laden bestimmter `int`-Werte auf den Stapel

Befehl	Werte für op bzw. ind	Länge des Befehls
<code>bipush op</code>	-128 .. +127	2
<code>sipush op</code>	-32 769 .. +32 767	3
<code>ldc ind</code>	0 .. 255	2
<code>ldc_w ind</code>	0 .. 65 535	3

Dabei bezeichnet `op` den Wert, der auf den Stapel geladen wird. Dagegen bezeichnet `ind` einen Index. An der entsprechenden Stelle der Konstantentabelle muss der zu ladende (4-Byte-lange) Wert stehen.

Unterschied zwischen dem Assembler-Befehl `ldc` und dem Maschinenbefehl `ldc`

In einem Jasmin-Programm (Java-Assembler-Programm) sind Befehle wie z.B. der folgende erlaubt:

```
ldc 1234567890
```

Der Assembler trägt die Zahl `1234567890` in die Konstantentabelle ein, z.B. an die Stelle mit dem Index `0A`, und übersetzt den Assembler-Befehl in den folgenden Maschinenbefehl:

```
12 0A
```

Dabei ist `12` der Operationscode des Assembler-Befehls `ldc`.

Notation für die Wirkung von Stapel-Befehlen

Im Verzeichnis der Maschinenbefehle wird z.B. die Wirkung des Befehls `iadd` (`int add`) so beschrieben: (`xy -> x+y`)

Das soll bedeuten: Damit der Befehl ausführbar ist, müssen oben auf dem Stapel 2 `int`-Werte `x` und `y` liegen. Die werden (entfernt und) *ersetzt* durch den einen Wert `x+y` (die Summe von `x` und `y`).

Aufgabe: Was bewirkt der Befehl `dmult`?

Befehle zum Aufrufen von Unterprogrammen

```
invokestatic    für Klassen-Methoden
invokevirtual   für Objekt-Methoden
invokespecial   für Objekt-Methoden (in der Superklasse, private etc.)
invokeinterface für Methoden einer Schnittstelle
```

Sprungbefehle und Sprungziele

Im Maschinenprogramm werden Sprungziele immer relativ zum Ort des Sprungbefehls angegeben (z.B. "17 Bytes nach vorn" oder "122 Bytes zurück"). Für Menschen ist das ziemlich schwer lesbar :-).

In einem Jasmin-Programm werden Sprungziele durch *Label* bezeichnet (siehe S. 5, Zeilen 16, 20, 30, ...). *Label* sind Methoden-lokal und man kann natürlich nicht aus *einer* Methode zu einem *Label* in einer *anderen* Methode springen.

Der Sprungbefehl `if_icmpge label` (S. 7).

Definitionen für die Begriffe Literal, Konstante und Wert

Literal: Ein Name für einen Wert. Literale sind syntaktische Größen, d.h. sie kommen in Quellprogrammen vor. Der Wert ist durch die betreffende Programmiersprache festgelegt.

Beispiele: In C/C++ und Java bezeichnet jedes der vier Literale `10`, `012`, `0xA` und `0XA` den `int`-Wert zehn. Das Literal `0.1` bezeichnet einen `double`-Wert, der etwas größer ist als ein Zehntel. Das Literal `0.1F` bezeichnet einen `float`-Wert, der etwas größer ist als der vom Literal `0.1` bezeichnete Wert. Das Literal `"H\u0061ll\u0061o!"` bezeichnet (eine Referenz die auf) ein String-Objekt "Hallo!" (zeigt).

Konstante: Ein Name für einen Wert. Konstanten sind syntaktische Größen, d.h. sie kommen in Quellprogrammen vor. Der Wert einer Konstanten wird vom Programmierer festgelegt.

Beispiele: In C/C++ kann der Programmierer z.B. wie folgt eine Konstante definieren:

```
#define MWST 19.0
```

Eine Adress-Konstante namens `r` kann man in C/C++ z.B. so definieren:

```
int r[] = {17, 35, 24};
```

Der *Wert* von `r` ist die Adresse der ersten Reihungskomponenten (d.h. der Stelle an der `17` steht).

Wert: Werte sind semantische Größen, d.h. sie werden (nur, erst) während der Ausführung eines Programms vom Ausführer berechnet, in Variablen abgelegt, miteinander verglichen etc. In einem Quellprogramm kommen keine Werte vor, nur Namen für Werte (z.B. Literale und Konstanten). Ein Programmierer sieht eigentlich nie einen Wert, nur Namen für Werte (z.B. Literale und Konstanten).

Konstanten und unveränderbare Variablen: Eine Konstante besteht nur aus einem Namen und einem Wert, hat also keine Adresse (siehe oben die `#define`-Konstante `MWST`). Eine unveränderbare Variable besteht mindestens aus einer Adresse und einem Wert (und kann zusätzlich auch noch einen Namen haben). Häufig wird nicht klar zwischen Konstanten und unveränderbaren Variablen unterschieden (und nur selten führt das zu Problemen).

11. VL Fr 12.12.08**A. Test 10****B. Organisation**

Heute fand im Vorlesungsblock anstelle einer Vorlesung eine zweite Übung statt, damit alle Teilnehmer noch vor den Weihnachtsferien die Aufgabe 7 (Alg01) und möglichst auch die Anfänge von Aufgabe 8 (Alg02) fertig bekommen. Im SWE-Labor gab es genug freie Arbeitsplätze, um diese zweite Übung durchzuführen.

12. VL Fr 19.12.08

A. Test 11

B. Organisation

Am Test (am letzten Termin vor Weihnachten) nahmen heute 10 Studenten teil, 3 fehlten.

Im Vorlesungsblock fand nur eine kurze Vorlesung statt (ca. 30 Minuten). Die verbleibende Zeit wurde für die Fortsetzung der Übung (Aufgabe 7 und 8, Alg01 und Alg02) verwendet.

In der Kurzvorlesung wurde das vor 3 Wochen ausgeteilte Papier Terme, Ausdrücke und Muster in Gentle besprochen (Seite 1 bis einschliesslich Anfang von Seite 3). Als "Rohmaterial" für die Erläuterungen wurden 2 Gentle-Typen (Farbe und Baum) eingeführt. Dann wurden die Begriffe Term, Grundterm, Spezialfall, Grundspezialfall, Ausdruck, Muster, Variablenbelegung und die Operationen

Auswertung (Ausdruck \times Variablenbelegung ergibt einen Wert) und
Musterabgleich (Muster \times Wert ergibt eine Variablenbelegung)
besprochen.

13. VL Fr 09.01.09**A. Test 12****B. Organisation: Klausur am Fr 06.02.09, im 3. Block (12-14 Uhr) im Raum D E15.**

Unterlagen: 5 Blätter, maximal DIN A4, beliebig beschriftet.

Die erste Klausur-AufgabeGeben Sie eine Grammatik an für die Menge aller natürlichen Zahlen im b -er-System, die man (ohne Rest) durch t teilen kann.Anmerkung: b soll an "Basis eines Zahlensystems" und t an "Teiler" erinnern.Wie löst man diese Aufgabe z.B. mit b gleich 2 und t gleich 3? Wie "konstruiert man" die einzelnen Regeln der gesuchten Grammatik?

// Zahlen, die aus nur einer Ziffer bestehen:

R01: RK0 -> 0

R02: RK1 -> 1

// Zahlen, die aus 2 oder mehr Ziffern bestehen

// Wir schreiben erst die rechten Seiten hin

// (kombinieren dort jede Restklasse mit jeder Ziffer)

// und überlegen uns dann die richtige linke Seite

R03: RK0 -> RK0 0

R04: RK1 -> RK0 1

R05: RK2 -> RK1 0

R06: RK0 -> RK1 1

R07: RK1 -> RK2 0

R08: RK2 -> RK2 1

Wie viele Regeln brauchen wir insgesamt (für beliebige Zahlen b und t)? $b + b \cdot t$ viele Regeln.**Papier "Terme, Ausdrücke und Muster in Gentle", S. 3, Aufgabe 4**

Haben wir diese Aufgabe letztes Mal schon bearbeitet?

Ein einfaches Prädikat: Vertausche**S. 4: Definition des action-Prädikats Vertausche durch 2 Regeln.****S. 5: Ein Aufruf des Prädikats Vertausche (mit einer Variablenbelegung VB0)****S. 6: Ausführung des Aufrufs, grafisch dargestellt****S. 7: Zusammenfassung: Wo muss man ein Muster und wo einen Ausdruck angeben?**

14. VL Fr 16.01.09

A. Test13

B. Organisation

Ein komplizierteres Prädikat: Rest3

Auf dem ausgeteilten Papier "Terme, Ausdrücke und Muster in Gentle"

S. 6: **Aufgabe 6** (ganz unten). Können Sie diese Aufgabe in unter 3 Minuten lösen?

Das Papier "Was passiert bei der Ausführung eines Prädikataufrufs?" (ohne Lösungen 4 Seiten) aus-
teilen und bearbeiten (zuerst gemeinsam, dann jeder für sich oder in kleinen Gruppen)

Aufgabe-01

Gemeinsam die ersten (nach unten gerichteten) Pfeile beschriften:

Pfeil 1 mit AW0 VB0. Das bedeutet: Wenn der Ausführer damit beginnt, einen Prädikataufruf auszufüh-
ren, muss er eine Variablenbelegung (VB0) haben. Damit und dem Eingabeparameter A0 des Aufrufs
führt er eine Auswertung AW0 durch. Das Ergebnis ist ein Wert W0.

Pfeil 2 mit MA0 W0. Das bedeutet: Mit dem Wert W0 und dem Eingabeparameter M0 der linken Seite
einer Regel führt er einen Musterabgleich MA0 durch. Das Ergebnis ist eine Variablenbelegung VB1.

Pfeil 3 mit AW1 VB1. Das bedeutet: Mit der Variablen-Belegung VB1 und dem Eingabeparameter A1
des ersten Prädikataufrufs auf der rechten Seite der Regel führt er eine Auswertung AW1 durch. Das Er-
gebnis ist ein Wert W1.

u.s.w.

Aufgabe-02

Die ersten 2 oder 3 Zeilen gemeinsam entsprechend der Musterlösung ausfüllen. Dann jeder für sich oder
in kleinen Gruppen weitermachen.

Aufgabe-03

Die ersten 2 oder 3 Zeilen gemeinsam entsprechend der Musterlösung ausfüllen. Dann jeder für sich oder
in kleinen Gruppen weitermachen.

Aufgabe-04

Diese Aufgabe sollte zu Hause bearbeitet werden.

15. VL Fr 23.01.09

A. Test 14

B. Organisation

Eine neue Version des Papiers "Was passiert bei der Ausführung eines Prädikataufrufs?" (mit nur kleinen Änderungen im Vergleich zur Vorversion) wurde ausgeteilt und kurz besprochen.

Die übrige Zeit des Vorlesungsblocks wurde zum Bearbeiten der Aufgaben 7 bis 11 verwendet.

16. VL Fr 30.01.09

A. Test 15 (entfällt, da alle Teilnehmer bereits an mindestens 13 Tests teilgenommen haben)

B. Organisation: Klausur am kommenden Freitag, 06.02.09, im 3. Block
(ab 12.15 Uhr) im Raum D E 15 (neben den Räumen des SWE-labors).

Und/Oder-Bäume, tiefes und flaches Wiederaufsetzen

Die Ausführung eines Prädikataufrufes kann man sich als ein Durchsuchen eines sogenannten *Und/Oder-Baumes* vorstellen. An der Wurzel eines solchen Baumes steht der auszuführende Prädikataufruf. In der "Schicht unterhalb der Wurzel" entspricht jeder Knoten einer Regel (des aufgerufenen Prädikates). Diese Regel-Knoten sollte man sich durch Oder verknüpft denken, denn der Prädikataufruf ist erfolgreich ausgeführt, wenn eine der Regeln erfolgreich ausgeführt wurde (Regel 1 oder Regel 2 oder ... oder Regel n):

```
Aufruf -|- Regel 1
        |- Regel 2
        ...
        |- Regel n
```

Unter jeder Regel R stehen auf der nächst tieferen Schicht die Prädikataufrufe (und ähnliche Konstrukte), die die rechte Seite von R bilden. Diese Prädikataufrufe sollte man sich durch Und verknüpft denken, denn die Regel R ist erfolgreich ausgeführt, wenn alle Prädikataufrufe auf ihrer rechten Seite erfolgreich ausgeführt wurden (Aufruf 1 und Aufruf 2 und ... und Aufruf m):

```
Regel -|- Aufruf 1
        |- Aufruf 2
        ...
        |- Aufruf m
```

Unter jedem Aufruf A stehen auf der nächst tieferen Schicht wieder die Regeln des aufgerufenen Prädikates und diese Regeln sind mit Oder verknüpft und unter jeder Regel stehen auf der nächst tieferen Schicht die Aufrufe, die die rechte Seite der Regel bilden und diese Aufrufe sind mit Und verknüpft u.s.w.

Ein Blatt eines solchen Und/Oder-Baumes ist entweder eine Regel, deren rechte Seite leer ist oder ein Aufruf eines "primitiven Prädikates", dessen Bedeutung nicht durch Regeln (sondern z. B. durch eine C-Routine) definiert ist.

Die folgenden Gentle-Typen und -Prädikate sollen dazu dienen, einige Und/Oder-Bäume als konkrete Beispiele zu konstruieren:

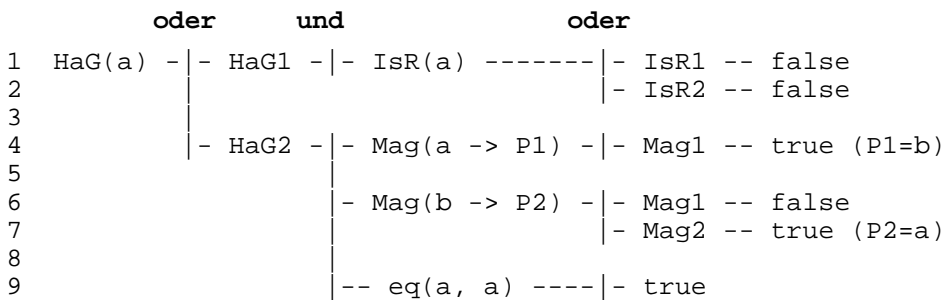
```
1 'type' PERSON a b c d e           -- Anton, Berta, Claus, Doris und Ernst
2 'condition' Mag(PERSON -> PERSON) -- Wer mag wen?
3 'rule' Mag(a -> b): .              -- Regel Mag1
4 'rule' Mag(b -> a): .              -- Regel Mag2
5 'rule' Mag(d -> c): .              -- Regel Mag3
6 'rule' Mag(d -> e): .              -- Regel Mag4
7 'rule' Mag(e -> d): .              -- Regel Mag5
8 'condition' IsR(PERSON)           -- Ist reich
9 'rule' IsR(b): .                  -- Regel IsR1
10 'rule' IsR(c): .                 -- Regel IsR2
11 'condition' HaG(PERSON)          -- Hat Glück
12 'rule' HaG(P0): .                -- Regel HaG1
13   IsR(P0).                       -- Aufruf HaG1.1
14 'rule' HaG(P0): .                -- Regel HaG2
15   Mag(P0 -> P1)                   -- Aufruf HaG2.1
16   Mag(P1 -> P2)                   -- Aufruf HaG2.2
17   eq(P0, P2).                     -- Aufruf HaG2.3
```

Das Prädikat Mag soll beschreiben, welche Person welche andere Person (bzw. welche anderen Personen, Mehrzahl) mag. Man beachte, dass Doris sowohl den Claus als auch den Ernst mag (siehe Regel Mag3 und Mag4). Anhand der Regeln Mag3 und Mag4 soll der Unterschied zwischen zwei wichtigen

Suchstrategien in Und/Oder-Bäumen illustriert werden (der Unterschied zwischen "tiefem Wiederaufsetzen" und "flachem Wiederaufsetzen", the difference between deep and shallow backtracking). Dieser Unterschied ist gleichzeitig ein wichtiger Unterschied zwischen der Sprache Prolog und der Sprache Gentle. In einem Gentle-Programm wäre die Regel `Mag4` sinnlos, in einem Prolog-Programm könnte eine entsprechende Regel dagegen durchaus sinnvoll sein. Dieser Unterschied zwischen Gentle und Prolog (d.h. der Unterschied zwischen flachem und tiefen Wiederaufsetzen) wird im folgenden genauer behandelt.

Das Prädikat `IsR` ("Ist reich") besagt, dass Berta und Claus reich sind (und die anderen drei Personen nicht reich sind). Zum Prädikat `HaG` ("Hat Glück") gehören zwei Regeln. Die erste (`HaG1`) besagt, dass eine Person Glück hat, wenn sie reich ist. Die zweite Regel (`HaG2`) drückt aus, dass eine Person `P0` Glück hat, wenn `P0` eine Person `P1` mag und `P1` wiederum `P0` mag.

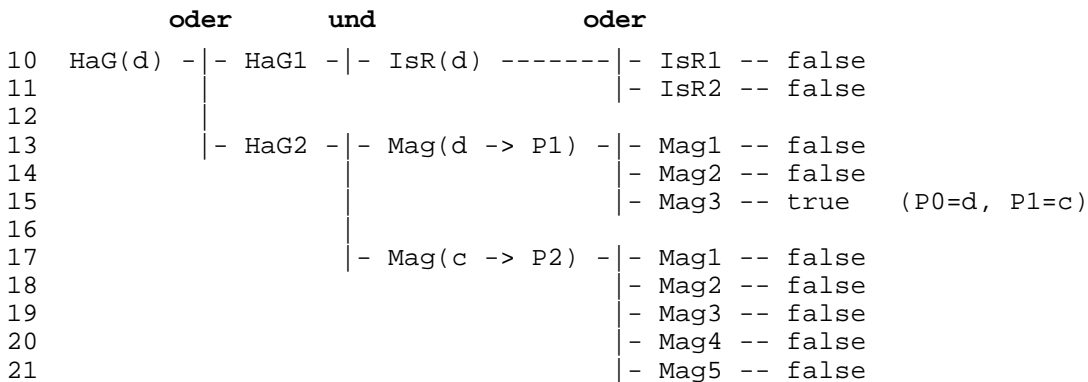
Als erstes Beispiel soll der Prädikataufruf `HaG(a)` ("Hat Anton Glück?") betrachtet werden. Die Ausführung dieses Aufrufs kann durch den folgenden Und/Oder-Baum dargestellt werden:



Der Aufruf `HaG(a)` kann erfolgreich ausgeführt werden, wenn die Regel `HaG1` oder die Regel `HaG2` erfolgreich ausgeführt werden kann. Die Regel `HaG1` kann nur dann erfolgreich ausgeführt werden, wenn der Aufruf `IsR(a)` erfolgreich ausgeführt werden kann. Der Aufruf `IsR(a)` kann erfolgreich ausgeführt werden, wenn die Regel `IsR1` oder die Regel `IsR2` erfolgreich ausgeführt werden kann. Das ist aber nicht der Fall, also können der Aufruf `IsR(a)` und damit die Regel `HaG1` nicht erfolgreich ausgeführt werden. Die Regel `HaG2` kann nur dann erfolgreich ausgeführt werden, wenn der Aufruf `Mag(a -> P1)` und der Aufruf `Mag(P1 -> P2)` und der Aufruf `eq(a, P2)` erfolgreich ausgeführt werden können u.s.w. Als Endergebnis ergibt sich, dass Anton tatsächlich Glück hat (weil er Berta mag und Berta ihn auch mag).

Aufgabe 21.1.: Beschreiben Sie ganz entsprechend die Ausführung des Prädikataufrufs `HaG(e)` ("Hat Ernst Glück?") durch einen Und/Oder-Baum und in Worten.

Das hier besonders wichtige Beispiel ist der Prädikataufruf `HaG(d)` ("Hat Doris Glück?"). Der folgende Und/Oder-Baum stellt die Ausführung dieses Aufrufs nach der Strategie *flaches Wiederaufsetzen* (shallow backtracking) dar. Diese Strategie wird in Gentle (für die Ausführung von Aktions- und Bedingungs-Prädikaten) verwendet:



Man erkennt hier: Die Regel HaG1 kann nicht ausgeführt werden ("Doris ist nicht reich"). Deshalb wird versucht, die Regel HaG2 auszuführen (siehe Zeile 13). Der Aufruf $\text{Mag}(d \rightarrow P1)$ kann aufgrund der Regel Mag3 ausgeführt werden und belegt die Variable $P1$ mit dem Wert Carl ("Doris mag Carl", siehe Zeile 15). Mit dieser Belegung scheitert dann aber die Ausführung des nächsten Aufrufs $\text{Mag}(c \rightarrow P2)$, denn "Claus mag niemanden" (siehe Zeile 17 bis 21). Damit ist die Regel HaG2 gescheitert. Da es für das Prädikat HaG keine weiteren Regeln (HaG3 , HaG4 etc.) gibt, die man jetzt noch probieren könnte, ist damit die Ausführung des Aufrufs $\text{HaG}(d)$ an der Wurzel des Und/Oder-Baumes endgültig gescheitert ("Doris hat kein Glück").

Dieses Scheitern des Aufrufs $\text{HaG}(d)$ ist eine Konsequenz der verwendeten Strategie *flaches Wiederaufsetzen*. Bei dieser Strategie betrachtet man von jedem Aufruf nur "das *erste* Gelingen" (falls die Ausführung des Aufrufs überhaupt gelingen kann). Der Aufruf $\text{Mag}(d \rightarrow P1)$ könnte aber nicht nur aufgrund der Regel Mag3 gelingen ("Doris mag Claus"), sondern auch aufgrund der Regel Mag4 ("Doris mag Ernst"). Bei der Strategie *tiefes Wiederaufsetzen* (deep backtracking) betrachtet man von jedem Aufruf nicht nur die erste gelungene Ausführung, sondern *alle* möglichen gelungenen Ausführungen. Mit dieser Strategie hat Doris doch noch Glück, denn sie mag ja (außer Carl auch noch) den Ernst und er mag sie. Der folgende Und/Oder-Baum stellt die Ausführung des Aufrufs $\text{HaG}(d)$ nach der Strategie tiefes Wiederaufsetzen (deep backtracking) dar. Diese Strategie wird in Prolog verwendet:

```

      oder      und      oder
22  HaG(d) - | - HaG1 - | - IsR(d) ----- | - IsR1 -- false
23          |         |         |         | - IsR2 -- false
24          |         |         |         |
25          | - HaG2 - | - Mag(d -> P1) - | - Mag1 -- false
26          |         |         |         | - Mag2 -- false
27          |         |         |         | - Mag3 -- true   (P0=d, P1=c)
28          |         |         |         |
29          |         | - Mag(c -> P2) - | - Mag1 -- false
30          |         | z         |         | - Mag2 -- false
31          |         | u         |         | - Mag3 -- false
32          |         | r         |         | - Mag4 -- false
33          |         | ü         |         | - Mag5 -- false
34          |         | c         |         |
35          |         | k - Mag(d -> P1) - | - Mag4 -- true   (P1=e)
36          |         |         |         |
37          |         | - Mag(e -> P2) - | - Mag1 -- false
38          |         |         |         | - Mag2 -- false
39          |         |         |         | - Mag3 -- false
40          |         |         |         | - Mag4 -- false
41          |         |         |         | - Mag5 -- true   (P2=d)
42          |         |         |         |
43          |         | - eq(d, d) ----- | - true

```

Wenn die Ausführung des Aufrufs $\text{Mag}(c \rightarrow P2)$ in Zeile 33 scheitert, dann wird sozusagen zur Zeile 27 zurückgegangen. Dort wurde mit der Regel Mag3 die erste Möglichkeit gefunden, den Aufruf $\text{Mag}(d \rightarrow P1)$ erfolgreich auszuführen. Jetzt wird (in Zeile 35) die nächste Möglichkeit (Mag4 : "Doris mag Ernst") genommen und damit "weitergemacht".

Den Unterschied zwischen flachem Wiederaufsetzen (wie in Gentle) und tiefem Wiederaufsetzen (wie in Prolog) kann man auch so zusammenfassen: Beim flachen Wiederaufsetzen stellt jedes Prädikat eine *Funktion* dar, während beim tiefen Wiederaufsetzen ein Prädikat im allgemeinen eine *Relation* beschreibt. Wenn Mag eine *Funktion* ist, dann kann Doris nur *einen* mögen. Ist Mag dagegen eine *Relation*, dann kann Doris *mehrere* Personen mögen.

Eine Funktion ist eine linkstotale, rechtseindeutige Relation. Eine nullstellige Funktion ist ein Wert. Eine nullstellige Relation ist eine Menge von Werten. Z. B. stellt das folgende Prädikat in Gentle den Wert b dar. Ein entsprechendes Prädikat in Prolog würde dagegen die Menge $\{b, c\}$ beschreiben:

```
1 'condition' ReM(-> PERSON)      -- Reiche Menschen
2 'rule' ReM(-> b): .             -- Berta ist ein reicher Mensch
3 'rule' ReM(-> c): .             -- Claus ist ein reicher Mensch
```