

Einführung in die Informatik

- Teil XVIII –

Reguläre Sprachen

Inhalt:

1. Satzgliederungssprachen
2. Die Backus-Naur-Form BNF
3. Linksreguläre Sprachen
 - 3.1 Generative Grammatiken
 - 3.2 Analysierende Grammatiken
 - 3.3 Die Äquivalenz von erkennenden Automaten und linksregulären Grammatiken
 - 3.4 Beschränkungen regulärer Sprachen
4. Aufgaben
5. Projektvorschlag: Funktionsplotter
 - 5.1 Die Aufgabenstellung
 - 5.2 Die Sprache der rationalen Funktionsterme
 - 5.3 Eine Grammatik für rationale Funktionsterme
 - 5.4 Ein Parser für rationale Funktionsterme
 - 5.5 Ein Rechner für rationale Funktionsterme
 - 5.6 Ein Applet für den Funktionsplotter
 - 5.7 Aufgaben

Literaturhinweise:

- Vossen/Witt: Grundlagen der Theoretischen Informatik mit Anwendungen, Vieweg
- E. Modrow: Theoretische Informatik mit Delphi, www.emu-online.de
- Krüger, Guido: Handbuch der Java-Programmierung, www.javabuch.de oder Addison Wesley 2002

1. Satzgliederungssprachen

Neben den Automaten sind Grammatiken ein effizientes Werkzeug der theoretischen Informatik. Sie sind deshalb so hilfreich, weil sich der Struktur der Grammatiken – speziell der Ersetzungsregeln – direkt ansehen lässt, ob überhaupt, und wenn, mit welchem Automatentyp die durch die Grammatik beschriebene Sprache analysierbar ist. Da hierfür keine mathematischen Vorkenntnisse erforderlich sind, lassen sich durch Grammatiken beschriebene *formale Sprachen* im Informatikunterricht der Schulen an jeder geeigneten Stelle einsetzen.

Ein Klassifizierungsschema für formale Sprachen wurde 1959 von Noam Chomsky aufgestellt, das so genannte *Satzgliederungssprachen* beschreibt. In diesem System werden *Worte*¹ einer Sprache aus Symbolen gebildet, die in dem für die Sprache gültigen Alphabet festgelegt sind. Da diese Symbole „endgültig“ sind, also nicht mehr verändert werden können, spricht man von der Menge **T** der *Terminalsymbole*. Die Worte werden (meist) gebildet, indem man ein einzelnes *Startsymbol S* durch eine Symbolfolge ersetzt, die aus Terminalsymbolen sowie anderen Symbolen bestehen kann, die wiederum ersetzt werden können. Diese „anderen“ – ersetzbaren – Symbole bilden die Menge der *Nichtterminalsymbole N*. Solange das neu gebildete Wort also Nichtterminalsymbole enthält, ist es noch nicht „fertig“. Erst, wenn es ganz aus Terminalsymbolen besteht, ist der Ersetzungsprozess abgeschlossen.

Die Ersetzungen erfolgen nach Regeln, die insgesamt ein *Regelsystem R* bilden. Regeln können in der Form

$$\text{Nichtterminalsymbol} \rightarrow \text{Symbolfolge}$$

aufgeschrieben werden. Dabei ist der Pfeil \rightarrow zu lesen als „wird ersetzt durch“. Die Grammatik **G** einer Satzgliederungssprache wird damit nach Chomsky festgelegt durch das Tupel

$$\mathbf{G} = (\mathbf{T}, \mathbf{N}, \mathbf{S}, \mathbf{R}).$$

Das Ganze hört sich komplizierter an, als es ist. Verdeutlichen wir den Prozess anhand eines einfachen Beispiels: Beschrieben werden sollen die natürlichen Zahlen. Diese bestehen nur aus Ziffern ohne Vorzeichen. Damit haben wir schon die Terminalsymbole gefunden:

$$\mathbf{T} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Jetzt müssen wir natürliche Zahlen durch Ersetzungen des Startsymbols **S** erzeugen. Damit gehört **S** schon mal zu den Nichtterminalsymbolen:

$$\mathbf{N} = \{\mathbf{S}, \dots\}.$$

Wie findet man natürliche Zahlen? Im einfachsten Fall bestehen sie nur aus einer Ziffer. Wir ersetzen also das Startsymbol durch eine solche:

$$\begin{aligned} \mathbf{S} &\rightarrow 0 \\ \mathbf{S} &\rightarrow 1 \\ \mathbf{S} &\rightarrow 2 \\ \mathbf{S} &\rightarrow 3 \\ \mathbf{S} &\rightarrow 4 \\ \mathbf{S} &\rightarrow \dots \end{aligned}$$

Das lässt sich etwas kürzer schreiben, wenn wir das *ODER-Symbol* „|“ benutzen:

$$\mathbf{S} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

¹ Der Begriff „Wort“ ist dabei ziemlich weit aufzufassen: Sind die Grundsymbole einzelne Zeichen, dann entsprechen die „Worte“ der umgangssprachlichen Bedeutung. Sind die Grundsymbole aber selbst schon Worte, dann sind die gebildeten „Worte“ umgangssprachlich eher „Sätze“.

Etwas längere natürliche Zahlen lassen sich erzeugen, wenn weitere Ziffern vor (oder nach) den schon erzeugten eingefügt werden können. Dafür lassen wir zu, dass das Startsymbol auch auf der rechten Seite von Ersetzungsregeln auftaucht. Wir erhalten eine *rekursive Regel*, mit deren Hilfe sich beliebig lange Ziffernfolgen bilden lassen.

$$S \rightarrow S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8 \mid S9$$

Weitere Nichtterminalsymbole benötigen wir nicht. (Aber wir könnten sie trotzdem einführen, wenn wir entsprechende Regeln einführen.)

Die Sprache der natürlichen Zahlen lässt sich damit durch die folgende Grammatik beschreiben.

$G_{\text{natürliche Zahlen}} = (T, N, S, R)$ mit

$$T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$N = \{S\}$$

$$R: S \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$S \rightarrow S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8 \mid S9$$

Die Zahl 123 z. B. entsteht durch die folgende Ableitung (Ersetzungsfolge):

$$S \rightarrow S3 \rightarrow S23 \rightarrow 123$$

Wir wollen hier schon festhalten, dass sich eine Sprache durchaus durch unterschiedliche Grammatiken beschreiben lässt. Wir hätten z. B. das Startsymbol auch rechts an die Ziffer anfügen können, so dass die Zahlen von links nach rechts statt von rechts nach links erzeugt würden.

Die Satzgliederungssprachen lassen sich nun anhand des formalen Aufbaus der Produktionsregeln in unterschiedliche Sprachklassen einordnen:

- Für *Chomsky-Typ 0-Sprachen* gelten keinerlei Einschränkungen.
- *Chomsky-Typ 1-Sprachen* verfügen über *kontextsensitive Regeln*, d. h. Ersetzungen dürfen nur vorgenommen werden, wenn das Nichtterminalsymbol in einem bestimmten Kontext auftaucht. Formal lässt sich das so schreiben:

$$\alpha A \beta \rightarrow \alpha \dots \beta, \quad \text{wobei } \alpha \text{ und } \beta \text{ den Kontext bilden, in dem die Ersetzung vorgenommen werden darf.}$$

- *Chomsky-Typ 2-Sprachen* sind *kontextunabhängig*, d. h. Ersetzungen dürfen ohne Berücksichtigung des Umfeldes vorgenommen werden. Für die Symbolfolge, durch die das Nichtterminalsymbol ersetzt wird, gelten keine weiteren Einschränkungen.

$$A \rightarrow \dots,$$

Zu den kontextfreien Sprachen gehören alle gängigen Programmiersprachen. Entsprechend groß ist die Bedeutung dieser Sprachklasse in der Informatik.

- *Chomsky-Typ 3-Sprachen* sind *regulär*, d. h. für Ersetzungen gelten sehr starke Einschränkungen. In allen Regeln darf das Nichtterminalsymbol entweder *durch ein einziges Terminalsymbol* oder durch *eine Folge aus einem Nichtterminal- und einem Terminalsymbol* ersetzt werden. Steht das Nichtterminalsymbol immer links, dann heißt die Sprache *linksregulär*, steht es immer rechts, dann *rechtsregulär*. Da sich alle regulären Sprachen auch linksregulär schreiben, beschränken wir uns auf diesen Sprachtyp. Alle Regeln sind also von einer der folgenden Formen:

$$A \rightarrow a \quad \text{oder} \quad A \rightarrow Ba \quad \text{mit} \quad A, B \in N \quad \text{und} \quad a \in T$$

Anhand des formalen Aufbaus der Regeln werden Sprachen also klassifiziert. Die Bedeutung dieser Regeln (ihre *Semantik*) ist dafür ohne Bedeutung. Alleine aus der *Syntax* der Grammatik ergeben sich für die Sprachklassen Aussagen zur Analysierbarkeit, also zum *Wortproblem*, das lautet: Gibt es einen Algorithmus, der feststellt, ob ein beliebiges Wort zu einer bestimmten Sprache gehört – oder nicht?

2. Die Backus-Naur-Form BNF

Als alternative Möglichkeit, die Produktionsregeln einer Grammatik aufzuschreiben, bietet sich die etwas ältere BNF-Form an, die schon benutzt wurde, als es keine grafischen Ausgabegeräte für Computer gab. Entsprechend beschränkt sich deren Zeichenvorrat auf die Symbole eines Fernschreibers. Der Vorteil der BNF-Form ist, dass sich die so aufgeschriebenen Regeln sehr leicht durch ein Programm analysieren lassen.

Für die BNF-Form gelten die folgenden Regeln:

1. Terminalsymbole werden ohne (*a*, *b*, *c*, *D*, ...), Nichtterminalsymbole mit spitzen Klammern (*<A>*, ...) geschrieben.
2. Produktionsregeln sind von der Form *<A> ::= b* usw.
3. Mehrere Ersetzungsmöglichkeiten werden durch Oder (|) getrennt: *<A> ::= a | b | ...*
4. Wiederholte Teile werden in geschweifte Klammern gesetzt. Dabei sind 0 Wiederholungen möglich. Wenn der Teil mindestens einmal auftritt, dann wird er einmal vorangestellt: *<A> ::= a{a}*

Die Grammatik für unsere natürlichen Zahlen können wir damit entweder z. B. so

$$\langle S \rangle ::= S0/S1/S2/S3/S4/S5/S6/S7/S8/S9/0/1/2/3/4/5/6/7/8/9$$

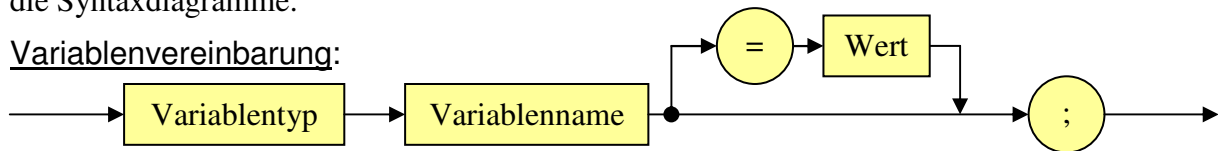
oder so formulieren:

$$\begin{aligned} \langle S \rangle &::= \langle A \rangle \{ \langle B \rangle \} \\ \langle A \rangle &::= 1/2/3/4/5/6/7/8/9 \\ \langle B \rangle &::= 0/1/2/3/4/5/6/7/8/9 \end{aligned}$$

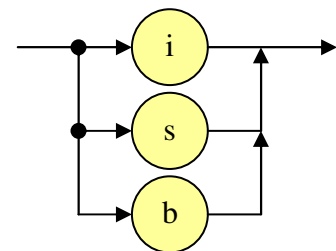
Damit haben wir zusätzlich die führende Null ausgeschlossen.

Als weiteres Beispiel wollen wir wieder die reduzierten Variablenvereinbarungen von Java wählen, die wir schon von den endlichen Automaten her kennen. Zur Erinnerung noch einmal die Syntaxdiagramme:

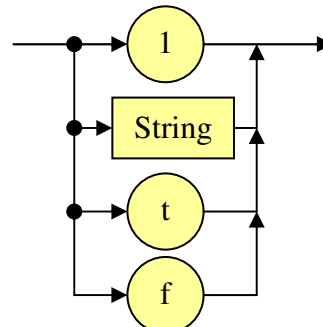
Variablenvereinbarung:



Variablentyp:



Wert:





Für die BNF-Form können wir aus Syntaxdiagramm fast abschreiben:

$$\begin{aligned} \langle \text{Variablenvereinbarung} \rangle &::= \langle \text{Variablentyp} \rangle \langle \text{Variablenname} \rangle ; | \\ &\langle \text{Variablentyp} \rangle \langle \text{Variablenname} \rangle = \langle \text{Wert} \rangle ; \end{aligned}$$

Jetzt werden Typen und Werte aufgezählt:

$$\begin{aligned} \langle \text{Variablentyp} \rangle &::= i | s | b \\ \langle \text{Wert} \rangle &::= 1 | \langle \text{String} \rangle | t | f \end{aligned}$$

Da Namen und Strings aus wenigstens einem Zeichen bestehen, müssen wir die in BNF ziemlich häufig auftauchenden Wiederholungen mit vorangestelltem „Einmalfall“ verwenden:

$$\begin{aligned} \langle \text{Variablenname} \rangle &::= a\{a\} \\ \langle \text{String} \rangle &::= 'x\{x\}' \end{aligned}$$

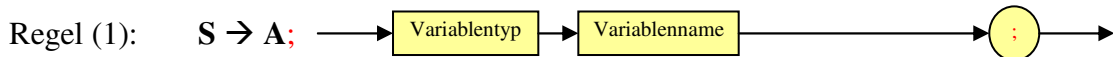
Da die Schreibweise für die Klassifizierung der Sprache nach Chomsky irrelevant ist, sehen wir den Regeln an, dass es sich um eine kontextfreie Grammatik handelt.

3. Linksreguläre Sprachen

3.1 Generative Grammatiken

Setzen wir eine Grammatik ein, Worte einer Sprache zu erzeugen, dann spricht man von einer *generativen Grammatik*. Für eine gegebene Grammatik kann dieser Einsatz dazu dienen, sich eine Übersicht darüber zu verschaffen, welche Art von Konstrukten die Grammatik produziert. Folglich wird man meist die als nächstes anzuwendende Regel zufällig auswählen lassen (falls es mehrere davon gibt). Zuerst einmal muss man aber geeignete Regeln finden.

Wir wollen wieder die bekannten vereinfachten Java-Variablenvereinbarungen beschreiben. Da sich in den Regeln linksregulärer Sprachen das Nichtterminalsymbol jeweils auf der linken Seite befindet, können auch nur dort weitere Zeichen angefügt werden. Auf diese Weise entstehen Worte der Sprache „von rechts nach links“, also „von hinten“². Wir müssen zuerst das letzte Symbol – hier ein Semikolon – erzeugen und uns weitere Ersetzungen offen halten – allerdings keine weiteren Semikolons. Wir beginnen deshalb mit der



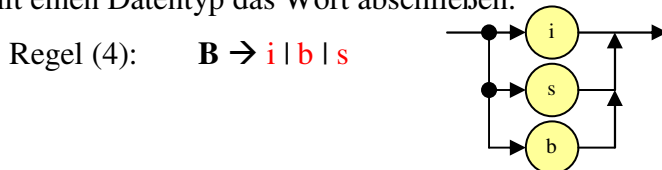
Danach können wir – für die „einfache“ Version ohne Wert – den Namen erzeugen,



der aus mehreren a's bestehen kann



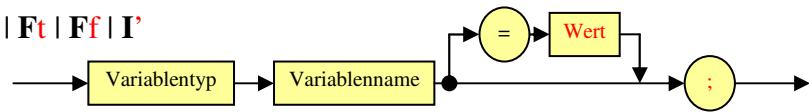
oder mit einem Datentyp das Wort abschließen.



² Das ist etwas gewöhnungsbedürftig. Da wir Grammatiken aber meist nicht generativ, sondern zur Analyse von Computersprachkonstrukten einsetzen, arbeiten wir dann „richtig herum“, also „von links nach rechts“.

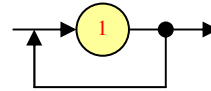
Jetzt müssen wir vor dem Semikolon in die „Werteabteilung“ verzweigen. Dafür muss eines der Zeichen t (true), f (false), ' (Hochkomma) oder 1 (Ziffer) erzeugt werden.

Regel (5): $A \rightarrow C1 | Ft | Ff | I'$



Die Nichtterminalsymbole davor müssen unterschiedlich gewählt werden, da je nach Typ unterschiedliche weitere Zeichen folgen. Erzeugen wir also zuerst einmal vollständige Zahlen

Regel (6): $C \rightarrow C1 | D=$



und dann den Rest dieses Zweiges.

Regel (7): $D \rightarrow Ea$

Regel (8): $E \rightarrow Ea | i$

Weil ein Zahlenzweig mit dem Typ i (integer) enden muss, dürfen wir nicht direkt zur Regel 3 gehen, weil dann die Information, dass wir gerade Zahlen erzeugen, verloren geht.

Ebenso können wir den Zweig für Wahrheitswerte fertig stellen,

Regel (9): $F \rightarrow G=$

Regel (10): $G \rightarrow Ha$

Regel (11): $H \rightarrow Ha | b$

und den für Strings.

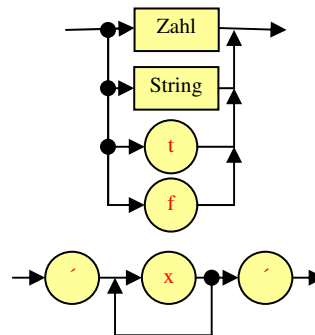
Regel (12): $I \rightarrow Jx$

Regel (13): $J \rightarrow Jx | K'$

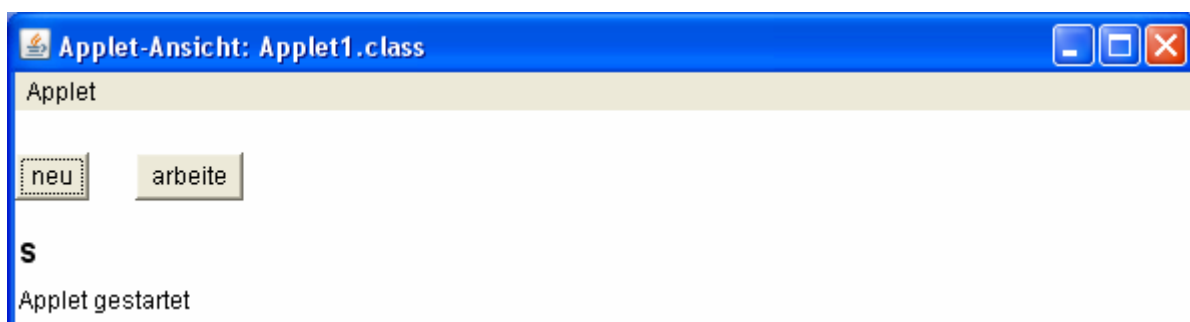
Regel (14): $K \rightarrow L=$

Regel (15): $L \rightarrow Ma$

Regel (16): $M \rightarrow Ma | s$



Jetzt wollen wir diese Grammatik zur Erzeugung von vereinfachten Java-Variablenvereinbarungen einsetzen. Dazu erzeugen wir eine geeignete Applet-Oberfläche.



```
private java.awt.Button bNeu; //ein Button für den Neustart
private java.awt.Button bArbeite; //und einer für Ersetzungsvorgänge
private java.awt.Label lAnzeige; //zur Anzeige der erzeugten Worte
```

Zuerst sehen wir uns die Arbeitsweise der beiden Buttons an.

Wir erzeugen die Komponenten im Designer, benennen und platzieren sie. Dann staten wir sie mit Eventhandler-Methoden aus: Der `neu`-Button setzt die Ausgabe auf einen String zurück, der nur das Startsymbol `S` enthält, der `arbeite`-Button ruft eine Methode zur Zeichenersetzung auf.

```
private void bNeuActionPerformed(java.awt.event.ActionEvent evt)
{
    lAnzeige.setText("S");
}

private void bArbeiteActionPerformed(java.awt.event.ActionEvent evt)
{
    arbeite();
}
```

Die `arbeite`-Methode muss nun erzeugt werden. Hier gestalten sich die Ersetzungsvorgänge etwas komplizierter, wenn wir die schon erzeugte Zeichenfolge weiterhin anzeigen wollen, z. B. als $S \rightarrow A; \rightarrow C1$. Dafür trennen wir den letzten Teil der Zeichenkette, die das schon erzeugte Teilwort enthält, ab und speichern es in der Variablen `h`. Sollten keine Ersetzungen möglich sein, dann soll der Ursprungstext bleiben. Den speichern wir dazu in `h2`. Danach betrachten wir das erste Zeichen von `h` und wenden eine der möglichen Regeln an, die wir durch Zufallszahlen auswählen.

```
private void arbeite()
{
    char c;
    int i;
    String h, h1, h2;

    h1 = lAnzeige.getText(); //bisherige Zeichenkette holen
    h2 = h1;
    h = "";
    i = h1.length() - 1; //letzten Teil abspalten
    do
    {
        c = h1.charAt(i);
        h = "" + c + h;
        i--;
    } while ((i >= 0) && !(c == '>'));
    if (h.charAt(0) == '>')
    {
        h = h.substring(1);
    }

    h1 = h1 + "-->"; //Ableitungsfolge anzeigen

    c = h.charAt(0); //erstes Zeichen abspalten
    h = h.substring(1);

    switch (c) //geeignete Regel anwenden
    {
        case 'S':
        {
            h = h1 + "A;" + h;
            break;
        }
    }
}
```

```
case 'A':
{
    i = (int) Math.round(5 * Math.random()); //je nach Zufallszahl handeln
    if (i == 0) h = h1 + "Ba" + h;
    else if (i == 1) h = h1 + "Cl" + h;
        else if (i == 3) h = h1 + "Ft" + h;
            else h = h1 + "I'" + h;
    break;
}

case 'B':
{
    i = (int) Math.round(4 * Math.random());
    if (i == 0) h = h1 + "Ba" + h;
    else if (i == 1) h = h1 + "i" + h;
        else if (i == 2) h = h1 + "b" + h;
            else h = h1 + "s" + h;
    break;
}

case 'C':
{
    i = (int) Math.round(2 * Math.random());
    if (i == 0) h = h1 + "Cl" + h;
    else h = h1 + "D=" + h;
    break;
}

case 'D':
{
    h = h1 + "Ea" + h;
    break;
}

case 'E':
{
    i = (int) Math.round(2 * Math.random());
    if (i == 0) h = h1 + "Ea" + h;
    else h = h1 + "i" + h;
    break;
}

case 'F':
{
    h = h1 + "G=" + h;
    break;
}

case 'G':
{
    h = h1 + "Ha" + h;
    break;
}

case 'H':
{
    i = (int) Math.round(2 * Math.random());
    if (i == 0) h = h1 + "Ha" + h;
    else h = h1 + "b" + h;
    break;
}
```

```
case 'I':
{
    h = h1 + "Jx" + h;
    break;
}

case 'J':
{
    i = (int) Math.round(2 * Math.random());
    if (i == 0) h = h1 + "Jx" + h;
    else h = h1 + "K'" + h;
    break;
}

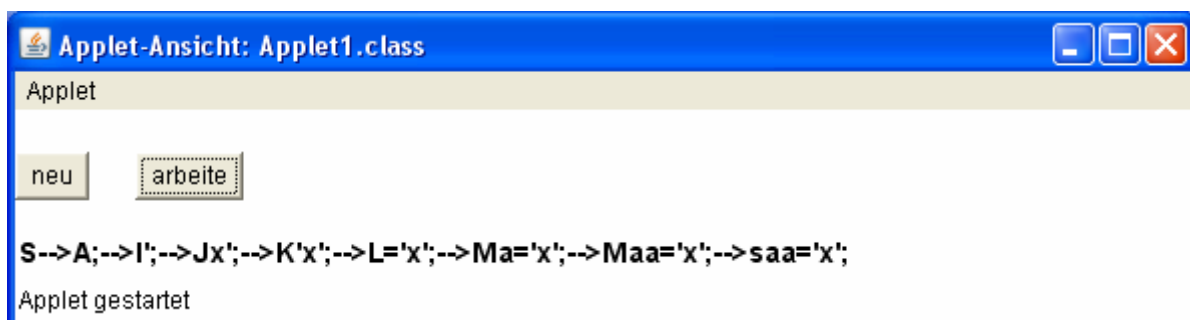
case 'K':
{
    h = h1 + "L=" + h;
    break;
}

case 'L':
{
    h = h1 + "Ma" + h;
    break;
}

case 'M':
{
    i = (int) Math.round(2 * Math.random());
    if (i == 0) h = h1 + "Ma" + h;
    else h = h1 + "s" + h;
    break;
}

default:
    h = h2; //es gab nichts zu ersetzen
}
lAnzeige.setText(h); //Ergebnis anzeigen
}
```

Das Ergebnis kann z. B. so aussehen:



3.2 Analysierende Grammatiken

Für unsere Zwecke sind generative Grammatiken eher Spielereien. Sie eignen sich ganz gut, um Zufallsgedichte u. Ä. zu erzeugen. Für die Praxis ist aber die „umgekehrte“ Anwendung wichtiger: mithilfe der Grammatik wird festgestellt, ob ein „Wort“ (ein Programmtext, ...) zu einer Sprache (z. B. zu Java) gehört, also entsprechend den Regeln der Programmiersprache zusammengesetzt wurde. Wir benutzen also die Grammatik zu Analysezwecken innerhalb eines Parsers (eines „Syntaxprüfers“).

Probieren wir das gleich mal wieder am bekannten Beispiel:

Statt aus dem Startsymbol ein Wort abzuleiten, wird versucht, das gegebene Wort auf das Startsymbol zu reduzieren. Das geschieht bei linksregulären Grammatiken, indem – von links beginnend – Zeichen für Zeichen anhand einer der Regeln der Grammatik ersetzt wird. Dabei werden jeweils die Zeichen, die auf der rechten Seite einer Regel auftauchen, durch das Nichtterminalsymbol auf der linken Seite der Regel ersetzt. Es entsteht eine baumartige – hier noch recht einfache – Struktur.

Versuchen wir es einmal mit der bekannten, etwas kürzer geschriebenen Grammatik aus dem vorigen Abschnitt:

- | | |
|--|---|
| (1): $S \rightarrow A;$ | (2): $A \rightarrow Ba \mid C1 \mid Ft \mid Ff \mid I'$ |
| (3): $B \rightarrow Ba \mid i \mid b \mid s$ | (4): $C \rightarrow C1 \mid D=$ |
| (5): $D \rightarrow Ea$ | (6): $E \rightarrow Ea \mid i$ |
| (7): $F \rightarrow G=$ | (8): $G \rightarrow Ha$ |
| (9): $H \rightarrow Ha \mid b$ | (10): $I \rightarrow Jx$ |
| (11): $J \rightarrow Jx \mid K'$ | (12): $K \rightarrow L=$ |
| (13): $L \rightarrow Ma$ | (14): $M \rightarrow Ma \mid s$ |

Analysiert werden soll das Wort (die Variablenvereinbarung) **ia=1;**

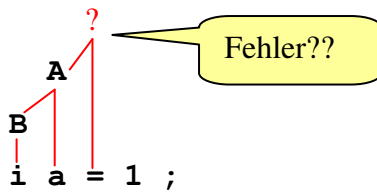
Schon mit dem ersten Zeichen gibt es Probleme: das Zeichen **i** tritt auf der rechten Seite der Regeln (3) und (6) auf. Welche nehmen? Probieren wir es zunächst mit Regel (3): Wir ersetzen das **i** durch das Symbol **B**. Zur Anschauung schreiben wir das **B** über das **i** und verbinden die beiden durch eine Linie.

$$\begin{array}{c} \mathbf{B} \\ | \\ \mathbf{i} \end{array} \mathbf{a} = \mathbf{1} ;$$

Jetzt lautet unser Wort **Ba=1;** - enthält also links ein Nichtterminalsymbol. Da bei linksregulären Sprachen auf den rechten Seiten der Grammatik entweder einzelne Terminalsymbole oder Folgen aus einem Nichtterminalsymbol und einem Terminalsymbol auftauchen, betrachten wir jetzt die nächsten beiden Symbole: **Ba**. Diese tauchen wiederum in zwei Regeln auf: in (2) und (3). Versuchen wir es also zunächst mit (2). Wir schreiben das gefundene Nichtterminalsymbol **A** wieder etwas höher und zeichnen zwei Linien zu den „verarbeiteten“ Symbolen **B** und **a**.

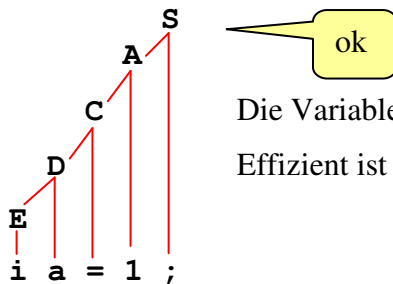
$$\begin{array}{c} \mathbf{A} \\ / \quad \backslash \\ \mathbf{B} \quad \mathbf{a} \\ | \quad | \\ \mathbf{i} \quad \mathbf{a} \end{array} = \mathbf{1} ;$$

Jetzt steht die Symbolfolge **A=** links im Wort. Die finden wir aber nirgends im Regelsystem.



Was heißt das? Auf *diesem* Weg lässt sich das gegebene Wort nicht auf das Startsymbol reduzieren. Wir müssen noch *alle anderen* Wege durchprobieren, solange, bis entweder ein Weg zum Startsymbol gefunden wurde, oder bis keine neuen Regelfolgen mehr übrig sind. Erst dann können wir das Wortproblem für diesen Fall entscheiden.

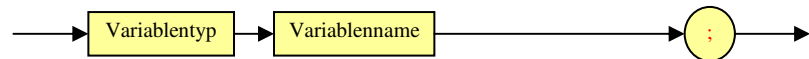
Nach langer Suche finden wir den richtigen *Syntaxbaum*:



Die Variablenvereinbarung ist also syntaktisch korrekt.
Effizient ist dieser Weg aber nicht!

Die bisher gefundene Grammatik eignet sich also ganz gut als generative Grammatik, aber sehr schlecht als analysierende. Da es für eine Sprache meist mehr als eine Grammatik gibt, konstruieren wir uns eine neue.

Wir beginnen diesmal von links und fangen mit einem einfachen Durchgang für ganze Zahlen ohne Wertzuweisung an:



Zuerst ersetzen wir das **i** durch ein Nichtterminalsymbol (hier: **A**).

$$(1) A \rightarrow i$$

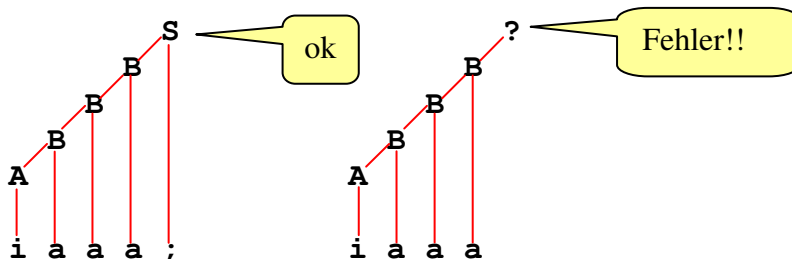
Danach halten wir uns offen, wie es nach den **a**'s des Namens weiter gehen soll. Wir geben nur eine Regel für Namen an.

$$(2) B \rightarrow Aa \mid Ba$$

Zum Schluss kann nur noch das Semikolon kommen: fertig!

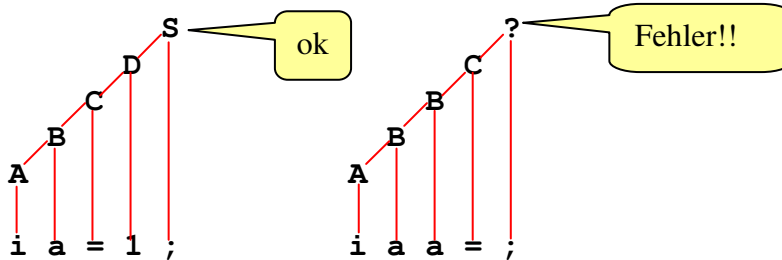
$$(3) S \rightarrow B;$$

Da jetzt die rechten Seiten eindeutig sind, kann jeweils beim Lesen des nächsten Zeichens entschieden werden, welche Regel anwendbar ist – es gibt ja nur eine (oder keine).



Weil das so schön geklappt hat, basteln wir jetzt den Wertezweig in das Regelsystem, indem wir ebenfalls dafür sorgen, dass die rechten Seiten eindeutig sind.

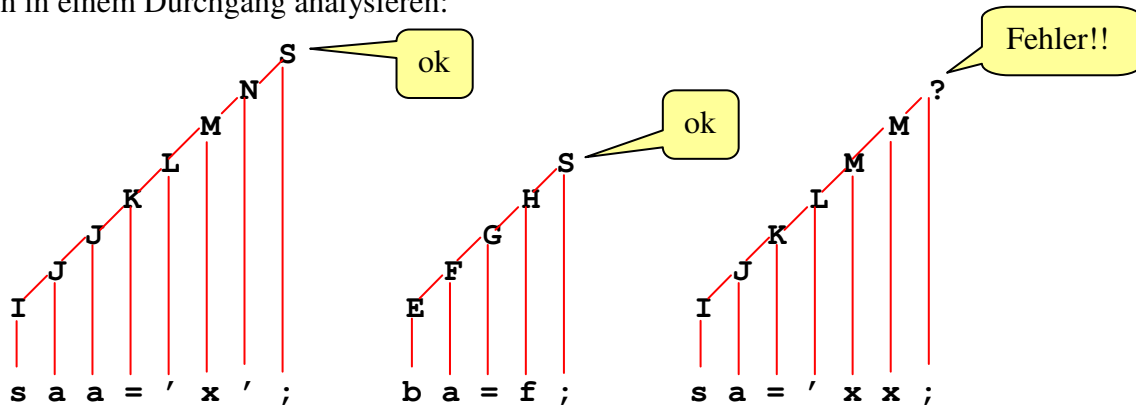
- (4) $C \rightarrow B=$ da wird verzweigt!
- (5) $D \rightarrow C1 \mid D1$ jetzt kommt die Zahl
- (6) $S \rightarrow D;$ fertig!



Für die anderen Datentypen können wir genauso vorgehen:

- | | | |
|--|--------------------------------|---------------------------------|
| (1) $A \rightarrow i$ | (6) $E \rightarrow b$ | (10) $I \rightarrow s$ |
| (2) $B \rightarrow Aa \mid Ba$ | (7) $F \rightarrow Ea \mid Fa$ | (11) $J \rightarrow Ia \mid Ja$ |
| (3) $S \rightarrow B; \mid D; \mid F; \mid H; \mid J; \mid N;$ | (8) $G \rightarrow F=$ | (12) $K \rightarrow J=$ |
| (4) $C \rightarrow B=$ | (9) $H \rightarrow Gt \mid Gf$ | (13) $L \rightarrow K'$ |
| (5) $D \rightarrow C1 \mid D1$ | | (14) $M \rightarrow Lx \mid Mx$ |
| | | (15) $N \rightarrow M'$ |

Mit dieser neuen Grammatik können wir jetzt die vereinfachten Java-Variablenvereinbarungen in einem Durchgang analysieren:



Folglich können wir diese Grammatik auch in einem Programm einsetzen, das solche Sprachkonstrukte analysiert.

Zuerst erzeugen wir wieder im Designer die Oberfläche:



Danach arbeiten wir den eingegebenen String zeichenweise von links her durch. Wir trennen jeweils das erste Zeichen ab. Ist dieses ein richtiges Terminalzeichen (**i**, **b** oder **s**), dann wenden wir die entsprechende Regel an, indem wir es durch das richtige Nichtterminalsymbol ersetzen. Ist es ein Nichtterminalzeichen, dann trennen wir auch das zweite Zeichen ab und versuchen, eine Regel anzuwenden.

```
private void arbeite()
{
    char c,d; //das sind die ersten beiden Zeichen
    String h,h1;

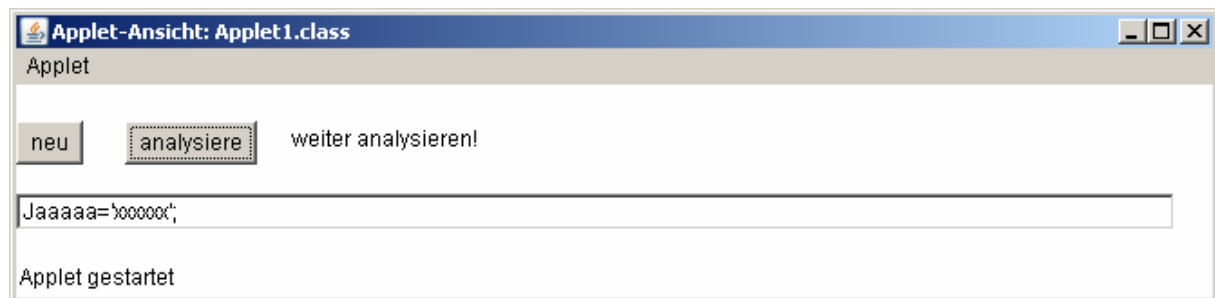
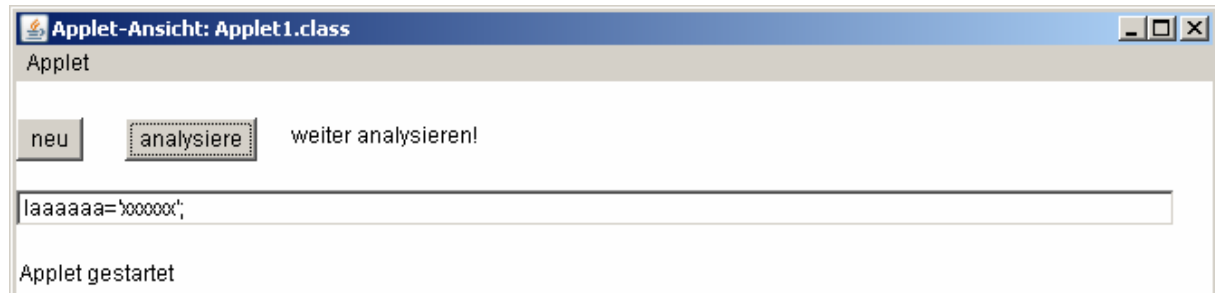
    h = tfEingabe.getText(); //Eingabe abholen
    h1 = h;
    if(!h.equals("S")) //dann wären wir ja fertig!
        if(h.length() == 0) h = "FEHLER!";
        else
        {
            c = h.charAt(0); //erstes Zeichen abtrennen
            h = h.substring(1);
            if(h.length()==0) h = "FEHLER! --> " + h1;
            else
            {
                d =h.charAt(0); //zweites Zeichen vorsorglich speichern

                switch(c)
                {
                    case 'i': {h = "A" + h; break;}
                    case 'b': {h = "E" + h; break;}
                    case 's': {h = "I" + h; break;}
                    case 'A':
                    {
                        h = h.substring(1); //jetzt kann das zweite Zeichen auch weg
                        if(d=='a') h = "B" + h;
                        else h = "FEHLER! --> " + h1;
                        break;
                    }
                    case 'B':
                    {
                        h = h.substring(1); //jetzt kann das zweite Zeichen auch weg
                        if(d=='a') h = "B" + h;
                        else if(d==';') h = "S" + h;
                            else if(d=='=') h = "C" + h;
                                else h = "FEHLER! --> " + h1;
                        break;
                    }
                    case 'C':
                    {
                        h = h.substring(1); //ebenso
                        if(d=='l') h = "D" + h;
                        else h = "FEHLER! --> " + h1;
                        break;
                    }
                    case 'D':
                    {
                        h = h.substring(1); //ebenso
                        if(d=='l') h = "D" + h;
                        else if(d==';') h = "S" + h;
                            else h = "FEHLER! --> " + h1;
                        break;
                    }
                    case 'E':
                    {
                        h = h.substring(1); //ebenso
                        if(d=='a') h = "F" + h;
                        else h = "FEHLER! --> " + h1;
                        break;
                    }
                }
            }
        }
}
```

```
case 'F':
{
  h = h.substring(1); //ebenso
  if(d=='a') h = "F" + h;
  else if(d==';') h = "S" + h;
    else if(d=='=') h = "G" + h;
      else h = "FEHLER! --> " + h1;
  break;
}
case 'G':
{
  h = h.substring(1); //ebenso
  if((d=='t')||(d=='f')) h = "H" + h;
  else h = "FEHLER! --> " + h1;
  break;
}
case 'H':
{
  h = h.substring(1); //ebenso
  if(d==';') h = "S" + h;
  else h = "FEHLER! --> " + h1;
  break;
}
case 'I':
{
  h = h.substring(1); //ebenso
  if(d=='a') h = "J" + h;
  else h = "FEHLER! --> " + h1;
  break;
}
case 'J':
{
  h = h.substring(1); //jetzt kann das zweite Zeichen auch weg
  if(d=='a') h = "J" + h;
  else if(d==';') h = "S" + h;
    else if(d=='=') h = "K" + h;
      else h = "FEHLER! --> " + h1;
  break;
}
case 'K':
{
  h = h.substring(1); //ebenso
  if((int)d==39) h = "L" + h; //das ist das Hochkomma
  else h = "FEHLER! --> " + h1;
  break;
}
case 'L':
{
  h = h.substring(1); //ebenso
  if(d=='x') h = "M" + h;
  else h = "FEHLER! --> " + h1;
  break;
}
case 'M':
{
  h = h.substring(1); //ebenso
  if(d=='x') h = "M" + h;
  else if((int)d==39) h = "N" + h; //das ist das Hochkomma
  else h = "FEHLER! --> " + h1;
  break;
}
```

```
        case 'N':  
        {  
            h = h.substring(1); //ebenso  
            if(d==';') h = "S" + h;  
            else h = "FEHLER! --> " + h1;  
            break;  
        }  
        default: h = "FEHLER! --> " + h1;  
    }  
    }  
    }  
    tfEingabe.setText(h);  
    if(h.equals("S")) lAnzeige.setText("ok");  
    else lAnzeige.setText("weiter analysieren!");  
    }  
}
```

Nach Eingabe einer Zeichenkette kann jetzt mehrmals der „analysiere“-Button gedrückt werden. Bei jedem Klick wird entweder eine Ersetzung durchgeführt oder ein Fehler angezeigt.



usw.

3.3 Die Äquivalenz von erkennenden Automaten und linksregulären Grammatiken

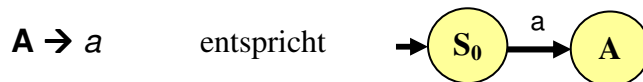
Haben wir einen endlichen Automaten ohne Ausgabe, dann bildet die Menge der Konstrukte aus Eingabezeichen, die den Automaten in einen Endzustand überführt, die Sprache, die dieser Automat akzeptiert. Diese Sprache kann immer durch eine linksreguläre Sprache beschrieben und somit auch analysiert werden. Umgekehrt gibt es zu jeder linksregulären Sprache einen endlichen Automaten, der die Sprache akzeptiert.

Der Beweis erfolgt durch Konstruktion:

Die Reduktion eines Wortes auf das Startsymbol der linksregulären Grammatik entspricht der Überführung des Automaten in den Endzustand. Also



Eine Ersetzung durch ein Terminalsymbol entspricht einem Übergang aus dem Anfangszustand:



Sonstige Ersetzungen entsprechen „normalen“ Übergängen im Automaten (wobei die Reihenfolge der Symbole „umgekehrt“ ist).



Als Beispiel wollen wir einfache Bezeichner einer Programmiersprache zuerst durch eine Grammatik beschreiben und dann den zugehörigen Automaten konstruieren. Das $\langle + \rangle$ -Zeichen steht für ein „unzulässiges“ Symbol, F für einen Fehler:

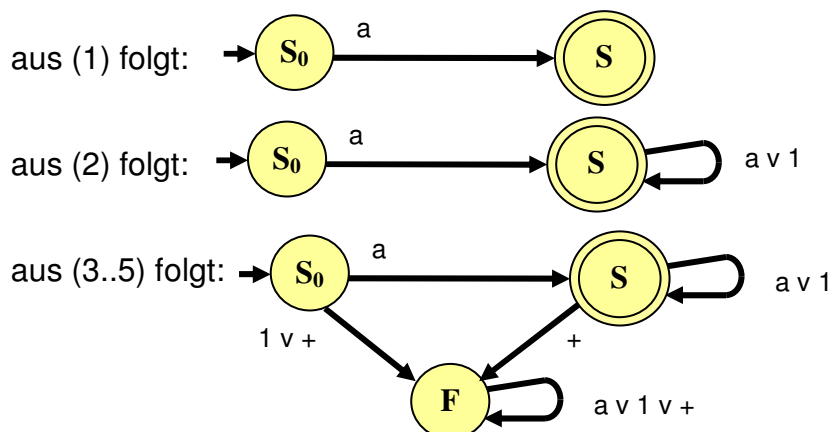
Beispiel: Bezeichner

$T = \{a, 1, +\}$ $N = \{S, F\}$ Startsymbol: S

Regeln:

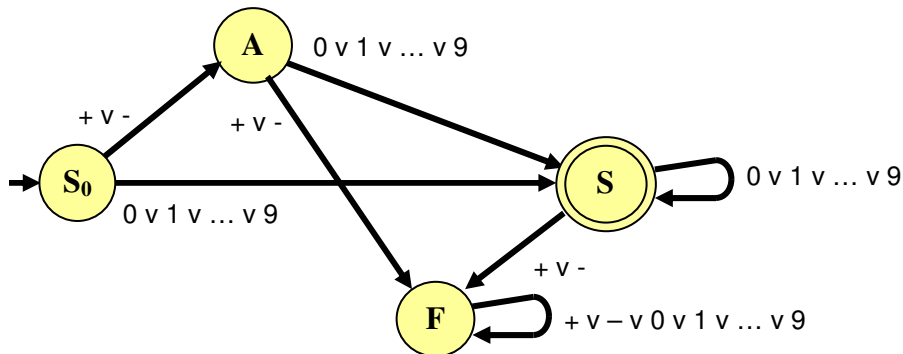
- (1) $S \rightarrow a$
- (2) $S \rightarrow Sa \mid S1$
- (3) $F \rightarrow 1 \mid +$
- (4) $F \rightarrow S+$
- (5) $F \rightarrow Fa \mid F1 \mid F+$

Der zugehörige Automat:



Beispiel: ganze Zahlen mit Vorzeichen

Diesmal wollen wir den Automaten vorgeben und die Grammatik daraus ableiten:



Die Menge der Terminalsymbole der Grammatik entspricht dem Eingabealphabet des Automaten:

$$\mathbf{T} = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Die Nichtterminalsymbole ergeben sich aus den Zuständen (wobei der Anfangszustand weggelassen wird):

$$\mathbf{N} = \{S, F, A\}$$

Das Startsymbol entspricht dem Endzustand:

Startsymbol: **S**

Jetzt kommen die Regeln:

Übergänge aus dem Anfangszustand entsprechen Regeln mit nur einem Terminalsymbol auf der rechten Seite. Dabei sind Übergänge in den Endzustand besonders zu beachten.

$$(1) \mathbf{S} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$(2) \mathbf{A} \rightarrow + \mid -$$

Weitere Übergänge in den Endzustand erfolgen von **A** aus:

$$(3) \mathbf{S} \rightarrow \mathbf{A}0 \mid \mathbf{A}1 \mid \mathbf{A}2 \mid \mathbf{A}3 \mid \mathbf{A}4 \mid \mathbf{A}5 \mid \mathbf{A}6 \mid \mathbf{A}7 \mid \mathbf{A}8 \mid \mathbf{A}9$$

Jetzt fehlen nur noch die Übergänge von **S** nach **S** und die in den Fehlerzustand:

$$(4) \mathbf{S} \rightarrow \mathbf{S}0 \mid \mathbf{S}1 \mid \mathbf{S}2 \mid \mathbf{S}3 \mid \mathbf{S}4 \mid \mathbf{S}5 \mid \mathbf{S}6 \mid \mathbf{S}7 \mid \mathbf{S}8 \mid \mathbf{S}9$$

$$(5) \mathbf{F} \rightarrow \mathbf{A}+ \mid \mathbf{A}- \mid \mathbf{S}+ \mid \mathbf{S}-$$

$$(6) \mathbf{F} \rightarrow \mathbf{F}+ \mid \mathbf{F}- \mid \mathbf{F}0 \mid \mathbf{F}1 \mid \mathbf{F}2 \mid \mathbf{F}3 \mid \mathbf{F}4 \mid \mathbf{F}5 \mid \mathbf{F}6 \mid \mathbf{F}7 \mid \mathbf{F}8 \mid \mathbf{F}9$$

Fertig ist die Grammatik!

3.4 Beschränkungen regulärer Sprachen

Endliche Automaten ohne Ausgabe werden also durch reguläre Sprachkonstrukte in den Endzustand überführt, sie akzeptieren Worte dieser Sprache. Umgekehrt können wir durch Konstruktion die Grammatik der Sprache finden, die ein gegebener Automat akzeptiert. Je nach Aufgabe können wir damit das geeignetere Modell (Grammatik oder Automat) benutzen, um unser Problem zu lösen. Folgerichtig unterliegen endliche Automaten und reguläre Sprachen dann auch den gleichen Beschränkungen.

Ein endlicher Automat mit n Zuständen kann sich vorausgegangene Zeichenfolgen nur „merken“, indem er in einen seiner Zustände übergeht. Kommt es bei den Zeichen auf deren Anzahl („geöffnete Klammern“, ...) oder ihre Struktur („Finde ein bestimmtes Wort“, ...) an, dann muss der Automat nach jedem Zeichen seinen Zustand wechseln. Hat er eine Zeichenfolge aus n Zeichen gelesen, dann muss er spätestens nach dem nächsten Zeichen in einen Zustand übergehen, den er schon vorher eingenommen hatte. Für einen Automaten in diesem Zustand lässt sich damit nicht mehr sagen, wie er in diesen Zustand gelangt ist, wie also die bisher gelesene Zeichenfolge beschaffen war. Bei genügend langen Zeichenfolgen können solche „Zustandsschleifen“ sogar beliebig oft durchlaufen werden. Endliche Automaten mit n Zuständen können damit keine Sprachen analysieren, bei denen für die Analyse die Struktur einer Teilzeichenfolge von Bedeutung ist, die mehr als n Zeichen umfasst. Standardbeispiel dafür sind Klammerstrukturen. Ein endlicher Automat mit n Zuständen kann damit durchaus z. B. eine Klammerstruktur analysieren, die Teilfolgen mit höchstens n öffnenden Klammern umfasst. Zu jedem endlichen Automaten können aber Klammerstrukturen angegeben werden, die nicht mehr analysiert werden können. Damit kann so eine Maschine nicht beliebige Strukturen analysieren, sondern nur einige.³

Für reguläre Sprachen wird dieser Sachverhalt durch das *Pumping-Lemma* sehr anschaulich beschrieben⁴: Hat man eine reguläre Sprache L , dann gibt es eine natürliche Zahl n , so dass alle Worte aus L , die länger als n sind, sich so in drei Teile u , v und w zerlegen lassen, dass gilt:

- (1) $|v| \geq 1$ (v hat mindestens ein Zeichen)
- (2) $|uv| \leq n$ (das ist der „führende“ Teil)
- (3) $uv^i w \in L, i \in \mathbb{N}_0$ (v kann beliebig oft wiederholt werden)

Damit können genügend lange Worte der Sprache durch Wiederholung von Wortteilen „aufgepumpt“ werden und so neue Worte erzeugt werden, die trotzdem weiter zur Sprache gehören. Das ist die direkte Analogie zu den „Zustandsschleifen“ weiter oben.

³ Ältere Kolleginnen und Kollegen werden sich erinnern, dass die ersten „wissenschaftlichen“ Taschenrechner nur eine vorgegebene Schachteltiefe von Klammerstrukturen verarbeiten konnten.

⁴ Vossen/Witt Seite 100ff.

4. Aufgaben

1. Eine neue Sprache

Gegeben ist eine Grammatik mit dem folgenden Regelsystem:

$$(1) \quad S \rightarrow a \mid b \mid 1 \mid 2 \mid Aa \mid Ab \mid A1 \mid A2$$

$$(2) \quad A \rightarrow S^+ \mid S^*$$

- Weshalb ist die Sprache linksregulär?
- Erzeugen Sie mindestens drei unterschiedlich lange Worte der zugehörigen Sprache (mit mindestens jeweils drei Zeichen). Geben Sie dazu jeweils alle Ersetzungen an.
- Prüfen Sie anhand der Syntaxbäume, ob die folgenden Worte zur Sprache gehören:
 $\alpha) a+a+b*2$ $\beta) 2a+1$
- Konstruieren Sie den zugehörigen endlichen Automaten.
- Beschreiben Sie die Grammatik in BNF-Form.
- Schreiben Sie ein Java-Programm, das Worte dieser Sprache zufallsgesteuert erzeugt.
- Analysieren Sie Worte dieser Sprache durch ein Java-Applet.

2. Gleitpunktzahlen Zahlen mit Vorzeichen und Dezimalpunkt als Trennzeichen

- Geben Sie eine linksreguläre Grammatik an, die ganze Zahlen, ggf. mit Vorzeichen beschreibt.
- Behandeln Sie die Aufgabe dann wie in Aufgabe 1.

3. Zufallsgedichte

Gesucht sind Zufallsgedichte aus jeweils drei Strophen zu jeweils vier Zeilen, von denen die ersten drei ähnlich, die vierte gesondert behandelt werden. Beispiel für eine (bessinnliche) Strophe:

*Der Hund bellt leise,
Der Mond steht schweigend,
Der Mann sinnt immer noch,
Aber das macht mir nichts aus!*

- Geben Sie eine linksreguläre Grammatik an, solche Gedichte beschreibt. Ggf. kann auch die Tendenz (fröhlich, deprimiert, tödlich gelangweilt, ...) vorgegeben werden!
- Behandeln Sie die Aufgabe dann – soweit möglich und sinnvoll – wie in Aufgabe 1.

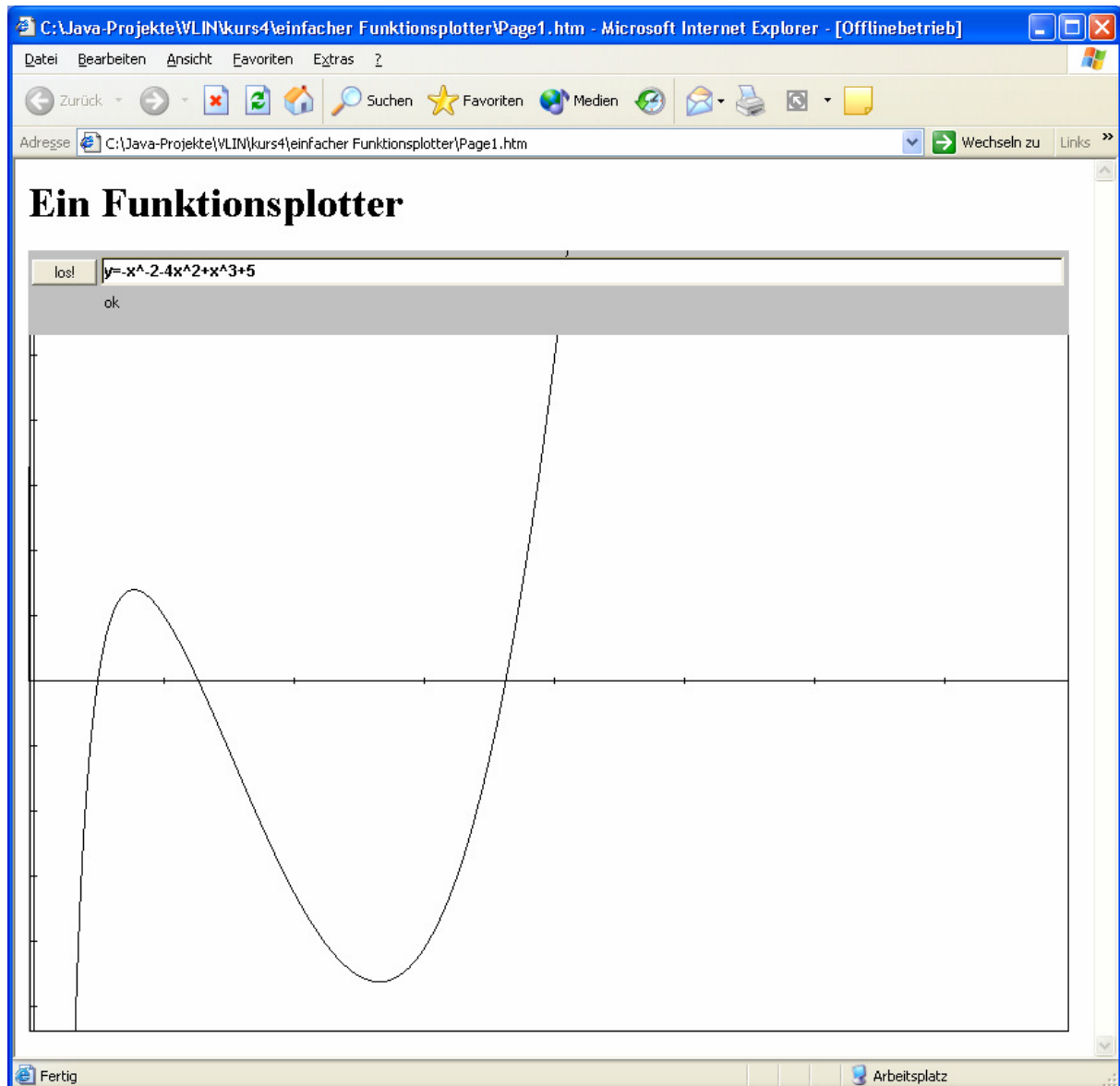
4. Zufällige Turtlebewegungen

- Begründen Sie, weshalb die Turtlebefehle aus dem ersten Teil durch eine linksreguläre Sprache beschrieben werden können.
- Geben Sie eine für eine generative Grammatik geeignete linksreguläre Grammatik an.
- Erzeugen Sie damit zufällig Turtlebefehle, die dann von der Turtle ausgeführt werden.

5. Projektvorschlag: Funktionsplotter

5.1 Die Aufgabenstellung

Wir wollen einige der bis jetzt gelernten Techniken auf eines der Standardthemen der Schul-informatik anwenden: die Darstellung von Funktionsgraphen. Mit den bisherigen Möglichkeiten können wir noch keine geschachtelten Strukturen analysieren, also z. B. keine Klammerstrukturen. Funktionsterme mit Klammern sind deshalb verboten. Stattdessen lassen wir negative Exponenten zu, so dass die auftretenden gebrochen-rationalen Funktionen auch ganz schöne Graphen liefern.

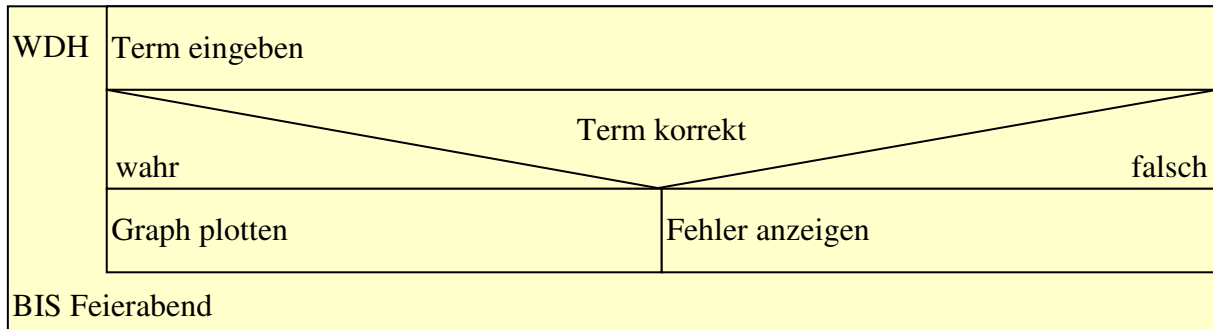


Um unser Projekt zu realisieren, müssen wir einige Fragen klären:

- Wie sind die zugelassenen Funktionsterme definiert?
- Wie können wir deren korrekte Schreibweise überprüfen?
- Wie berechnen wir aus den Termen die Funktionswerte?
- Wie stellen wir das Ergebnis dar?

Die Teilaufgaben sind weitgehend unabhängig voneinander zu bearbeiten, erfordern unterschiedliche Techniken aus unterschiedlichen Themenbereichen und lassen auch unterschiedliche Lösungsansätze mit unterschiedlichen Schwierigkeitsgraden zu. Das Thema eignet sich deshalb gut für arbeitsteiligen Unterricht.

Unsere Funktionsterme sollen, wie oben in der Grafik sichtbar, in ein Textfeld eingegeben, dann analysiert und berechnet werden. Der Arbeitsablauf lässt sich somit leicht im Struktogramm darstellen:

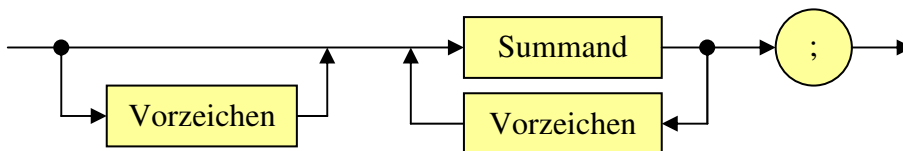


5.2 Die Sprache der rationalen Funktionsterme

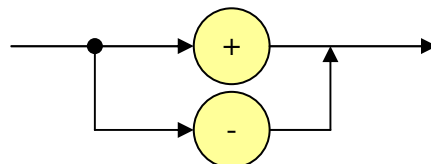
Bei der Eingabe wird dem Programm eine Zeichenkette übergeben, die erstmal keine besondere Bedeutung hat. Die Interpretation, dass es sich um einen Funktionsterm handeln soll, bleibt dem Benutzer überlassen. Für die späteren Berechnungen muss also zuerst einmal festgelegt werden, von welcher Art die Eingaben sein dürfen.

Die für unseren Plotter zugelassenen Funktionsterme wollen wir – wie üblich – über Syntaxdiagramme definieren, denn die Regeln, denen sie gehorchen müssen, bilden die Grammatik einer Sprache. Wir definieren als:

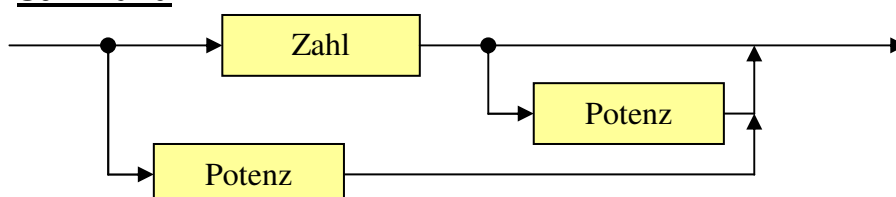
Term:

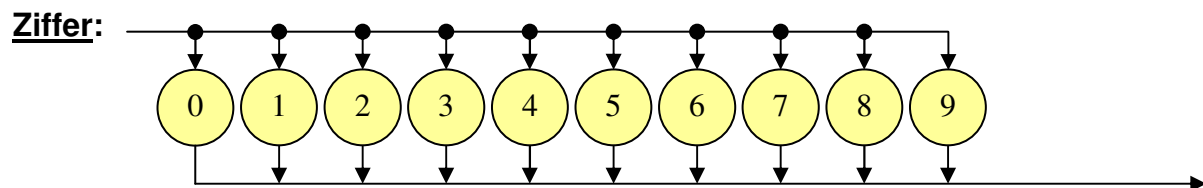
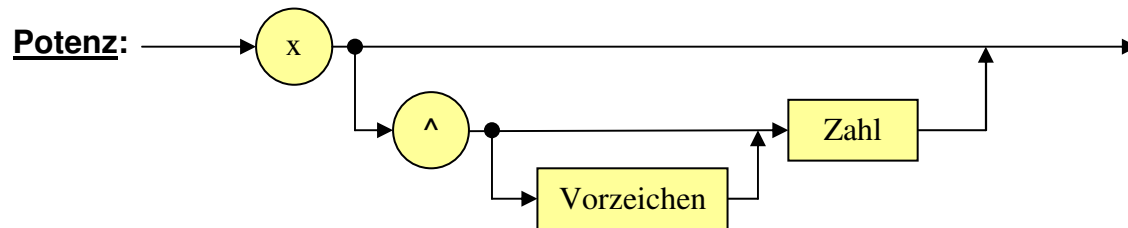
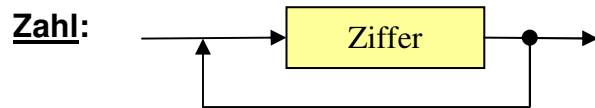


Vorzeichen:



Summand:





Einige Terme, die nach diesen Regeln gebildet werden können, lauten:

- $22;$
- $x;$
- $-3x^2-2x-2;$
- $1-x^2;$

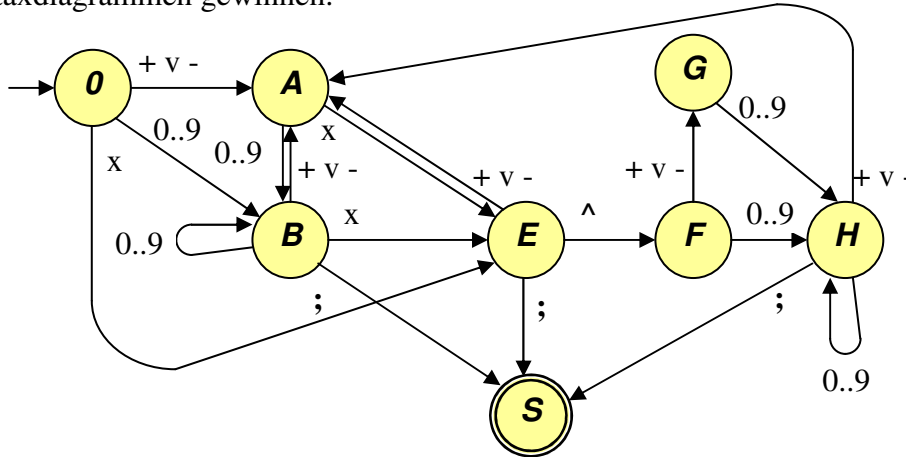
Fehlerhaft sind die folgenden Eingaben:

- $3*x;$
- $1-x$
- $1.5-x^2;$

Der Einfachheit halber wollen wir bei Eingaben, die nicht mit einem Semikolon abgeschlossen wurden, dieses ergänzen.

5.3 Eine Grammatik für rationale Funktionsterme

Das Parsen der Eingaben soll mithilfe einer regulären Grammatik geschehen. Diese können wir einfach gewinnen, wenn wir den Transitionsgraphen eines endlichen Automaten, der die Sprache akzeptiert, in eine Grammatik umsetzen⁵. Der Automat lässt sich direkt aus den Syntaxdiagrammen gewinnen.



Mithilfe der „Übersetzungsregeln“ aus Kapitel 3.3 lässt sich die gesuchte Grammatik direkt entnehmen:

$$T = \{+, -, x, ^, ;, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$N = \{S, A, B, E, F, G, H\}$$

Startsymbol: S

$$(1) S \rightarrow B; \mid E; \mid H;$$

$$(2) A \rightarrow + \mid - \mid B+ \mid B- \mid E+ \mid E- \mid H+ \mid H-$$

$$(3) B \rightarrow 0 \mid 1 \mid \dots \mid 9 \mid B0 \mid B1 \mid \dots \mid B9$$

$$(4) E \rightarrow x \mid Bx \mid Ax$$

$$(5) F \rightarrow E^{\wedge}$$

$$(6) G \rightarrow F+ \mid F-$$

$$(7) H \rightarrow F0 \mid F1 \mid \dots \mid F9 \mid G0 \mid G1 \mid \dots \mid G9 \mid H0 \mid H1 \mid \dots \mid H9$$

⁵ Natürlich könnten wir auch den Automaten als Parser einsetzen – aber das ist nicht das Thema dieses Kapitels!

5.4 Ein Parser für rationale Funktionsterme

Eine Klasse, die mithilfe des Methodenaufrufs *pruefe(String s)* einen Term überprüfen kann, lässt sich in direkter Analogie zu den schon bekannten Parsern schreiben. Sie besteht im Wesentlichen aus einer Fallunterscheidung (*switch...*).

```
class FParser
{
    public boolean pruefe(String s) //s wird überprüft
    {
        char c,d; //die ersten beiden Zeichen
        boolean ok = true; //das Ergebnis
        if(s.length()<2) ok = false;
        while((s.length()>1) && ok)
        {
            c = s.charAt(0); //erstes Zeichen abtrennen
            s = s.substring(1);
            d = s.charAt(0); //zweites Zeichen vorsorglich speichern
            switch(c)
            {
                case '+': {s = "A" + s; break;}
                case '-': {s = "A" + s; break;}
                case '0': {s = "B" + s; break;}
                case '1': {s = "B" + s; break;}
                case '2': {s = "B" + s; break;}
                case '3': {s = "B" + s; break;}
                case '4': {s = "B" + s; break;}
                case '5': {s = "B" + s; break;}
                case '6': {s = "B" + s; break;}
                case '7': {s = "B" + s; break;}
                case '8': {s = "B" + s; break;}
                case '9': {s = "B" + s; break;}
                case 'x': {s = "E" + s; break;}
                case 'A':
                {
                    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
                    if(d=='x') s = "E" + s;
                    else if((d>='0')&&(d<='9')) s = "B" + s;
                    else ok = false;
                    break;
                }
                case 'B':
                {
                    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
                    if((d=='+')||(d=='-')) s = "A" + s;
                    else if(d=='x') s = "E" + s;
                    else if((d>='0')&&(d<='9')) s = "B" + s;
                    else if(d==';') s = "S" + s;
                    else ok = false;

                    break;
                }
                case 'E':
                {
                    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
                    if((d=='+')||(d=='-')) s = "A" + s;
                    else if(d=='^') s = "F" + s;
                    else if(d==';') s = "S" + s;
                    else ok = false;

                    break;
                }
            }
        }
    }
}
```

```
case 'F':
{
    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
    if((d=='+')||(d=='-')) s = "G" + s;
    else if((d>='0')&&(d<='9')) s = "H" + s;
    else if(d=='^') s = "F" + s;
        else ok = false;
    break;
}
case 'G':
{
    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
    if((d>='0')&&(d<='9')) s = "H" + s;
    else ok = false;
    break;
}
case 'H':
{
    s = s.substring(1); //jetzt kann das zweite Zeichen auch weg
    if((d=='+')||(d=='-')) s = "A" + s;
    else if(d==';') s = "S" + s;
        else if((d>='0')&&(d<='9')) s = "H" + s;
            else ok = false;
    break;
}
}
}
return ok && (s.equals("S"));
}
```

5.5 Ein Rechner für rationale Funktionsterme

Die Auswertung der Terme, die vorher auf syntaktische Korrektheit geprüft wurde, ist in einer Methode `berechne(String s, double x)` einfach. Die Zeichenkette `s` wird von links nach rechts gelesen, wobei für jeden Summanden Zeichen für Zeichen die benötigten Informationen (Vorzeichen, Zahlenwerte, Exponenten, ...) zusammengesucht werden. Anschließend wird aus dem übergebenen Wert `x` der Wert des Summanden berechnet (in einem geschützten Block (`try ... catch ...`) und ggf. zum Gesamtergebnis `ergebnis` addiert. Bei auftretenden Fehlern wird der Wert von `fehler` entsprechend gesetzt. Dieser kann vom Benutzer der Klasse abgefragt werden.

```
class FRechner
{
    boolean fehler = false;

    public double berechne(String s, double x)
    {
        char c;
        double y=0, ergebnis = 0;
        int vorzeichen = 1; //Vorzeichen
        double zahl = 1;
        double exponent = 1;

        while((s.length()>1) && !fehler)
        {
            //Zahl zusammensuchen
            c = s.charAt(0); s = s.substring(1); //erstes Zeichen abschneiden
            vorzeichen = 1;
            zahl = 1;

            if(c=='-') //ggf. Vorzeichen bestimmen
            {
                vorzeichen = -1;
                c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
            }
            else if(c=='+')
            {
                c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
            }

            if((c>='0')&&(c<='9')) //kommt ´ne Zahl?
            {
                zahl = (int)c - (int)'0';
                c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
                while((c>='0')&&(c<='9'))
                {
                    zahl = 10*zahl + (int)c - (int)'0';
                    c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
                }
            }
            zahl = vorzeichen*zahl; //das ist doch schon mal was!

            if((c=='+')|| (c=='-') || (c==';')) //jetzt kommt ein neuer Summand
            {
                ergebnis = ergebnis + zahl;
                s = "" + c + s; //Vorzeichen wieder vorne anhängen
            }
        }
    }
}
```

```
if (c=='x') //der Parameter wird benutzt
{
    //Exponent suchen
    c = s.charAt(0); s = s.substring(1); //<x> abschneiden
    if ((c==';' || c=='+' || c=='-')) //x^1
    {
        ergebnis = ergebnis + zahl*x;
        s = "" + c + s;
    }
    else
    {
        c = s.charAt(0); s = s.substring(1); //<^> abschneiden
        vorzeichen = 1;
        exponent = 1;
        if (c=='-')
        {
            vorzeichen = -1;
            c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
        }
        else if (c=='+')
        {
            c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
        }
        if ((c>='0') && (c<='9'))
        {
            exponent = (int)c - (int)'0';
            c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
            while ((c>='0') && (c<='9'))
            {
                exponent = 10*exponent + (int)c - (int)'0';
                c = s.charAt(0); s = s.substring(1); //nächstes Zeichen abschneiden
            }
            exponent = vorzeichen*exponent;
            s = "" + c + s; //Vorzeichen wieder vorne anhängen
            fehler = false;
            try //Wert des Summanden berechnen
            {
                y = zahl*Math.exp(exponent*Math.log(x));
                ergebnis = ergebnis + y;
            }
            catch (Exception ex)
            {
                fehler = true;
            }
        }
    }
}
return ergebnis;
}
```

Aus der Benutzung des Logarithmus ergeben sich drastische Einschränkungen für den Wertebereich von x .
Dafür ist es schön einfach!

5.6 Ein Applet für den Funktionsplotter



Unser Applet soll nur über eine spartanische Oberfläche verfügen: Im oberen Bereich befindet sich eine Panel-Komponente *pOben*, die einen Button *bStart*, ein Text-Eingabefeld *tfEingabe* und eine Label-Komponente *lFehler* trägt. Darunter wird direkt auf dem Applet gezeichnet. In diesem Fall nur die beiden rechten Quadranten eines Koordinatensystems, weil aus der Berechnungsart in der Klasse *FRechner* wegen der Logarithmen keine Werte für negative Argumente bestimmt werden. Die Erzeugung dieser Komponenten erfolgt wie immer im Designermodus, Anfangswerte werden am Ende der *init*-Methode gesetzt.

```
import java.awt.*;
public class Applet1 extends java.applet.Applet
{
    private java.awt.Button bStart; //im Designer erzeugt
    private java.awt.Label lFehler;
    private java.awt.Panel pOben;
    private java.awt.TextField tfEingabe;

    int breite, hoehe; //Maße des Applets
    int x0, y0, dx, dy; //bestimmende Größen für das Koordinatensystem
    String eingabe; //das wird der Funktionsterm
    boolean eingabeOK = false;
    boolean ersterPunkt = true;
    FParser p = new FParser(); //der Syntaxchecker
    FRechner r = new FRechner(); // der Rechner
}
```

```
public void init()
{
    ...
    setSize(800, 600); //Appletmaße bestimmen
    breite = getBounds().width;
    hoehe = getBounds().height;
    x0 = 4; //Mittelpunkte des Koordinatensystems und Skalierungsfaktoren festlegen
    y0 = (int) Math.round((hoehe - 60) / 2) + 60;
    dx = 100;
    dy = 50;
}
```

Zuerst einmal muss ein leeres Koordinatensystem gezeichnet werden:

```
private void anfangsbild(Graphics g)
{
    g.setColor(Color.white); //Bild löschen
    g.fillRect(1, 61, breite - 2, hoehe - 62);
    g.setColor(Color.black);
    g.drawRect(1, 61, breite - 2, hoehe - 62);
    g.drawLine(1, y0, breite - 1, y0); //Achsen zeichnen
    g.drawLine(x0, 61, x0, hoehe - 1);
    for (int i = 1; i <= (int) Math.round(breite / dx); i++) //x-Achse skalieren
    {
        g.drawLine(x0 - dx * i, y0 - 2, x0 - dx * i, y0 + 2);
        g.drawLine(x0 + dx * i, y0 - 2, x0 + dx * i, y0 + 2);
    }
    for (int i = 1; i <= (int) Math.round(y0 / dy); i++) //y-Achse skalieren
    {
        g.drawLine(x0 - 2, y0 + dy * i, x0 + 2, y0 + dy * i);
        g.drawLine(x0 - 2, y0 - dy * i, x0 + 2, y0 - dy * i);
    }
}
```

Beim Anklicken des Buttons oder wenn im Eingabefeld die Return-Taste gedrückt wurde soll der Auswertevorgang starten:

```
private void bStartActionPerformed(java.awt.event.ActionEvent evt)
{
    werteAus();
}

private void tfEingabeKeyReleased(java.awt.event.KeyEvent evt)
{
    if (evt.getKeyCode() == 10) werteAus();
}
```

Dann kommt:

- Der Eingabestring wird geholt und ggf. die Zeichenfolge „y=“ vorne abgeschnitten.
- Danach wird bei Bedarf hinten ein Semikolon angehängt (weil man das so gerne vergisst).
- Danach wird die Syntax überprüft.
- Wenn alles richtig war, dann wird der Zeichenvorgang durch den *repaint()*-Aufruf eingeleitet.

```
private void werteAus()
{
    eingabe = tfEingabe.getText();
    boolean allesOK = true;
    try //ggf. "y=" abschneiden
    {
        if (eingabe.substring(0, 2).equals("y="))
            eingabe = eingabe.substring(2);
    } catch (Exception ex)
    {
        lFehler.setText("FEHLER: Eingabe zu kurz!");
        allesOK = false;
    }
    if (eingabe.length() > 0) //ggf. Semikolon anhängen
        if (!(eingabe.charAt(eingabe.length() - 1) == ';'))
            eingabe = eingabe + ";";
    else allesOK = false;
    allesOK = allesOK && p.pruefe(eingabe); //Syntaxcheck

    if (allesOK)
    {
        lFehler.setText("Eingabe ok --> es wird gezeichnet");
        eingabeOK = true;
        repaint(); //Graph zeichnen
        lFehler.setText("ok");
    }
    else
    {
        lFehler.setText("FEHLER");
        eingabeOK = false;
    }
}
```

Und wie zeichnet man den Graphen?

In ein neu gezeichnetes Koordinatensystem wird in einer Schleife für jede horizontale Bildschirmkoordinate der entsprechende x-Wert ermittelt und der zugehörige Funktionswert bestimmt. Danach wird dieser in eine vertikale Bildschirmkoordinate umgerechnet. Wurde schon ein Punkt gezeichnet, dann wird eine Linie zum neuen Punkt gemalt, sonst wird der Punkt nur vermerkt.

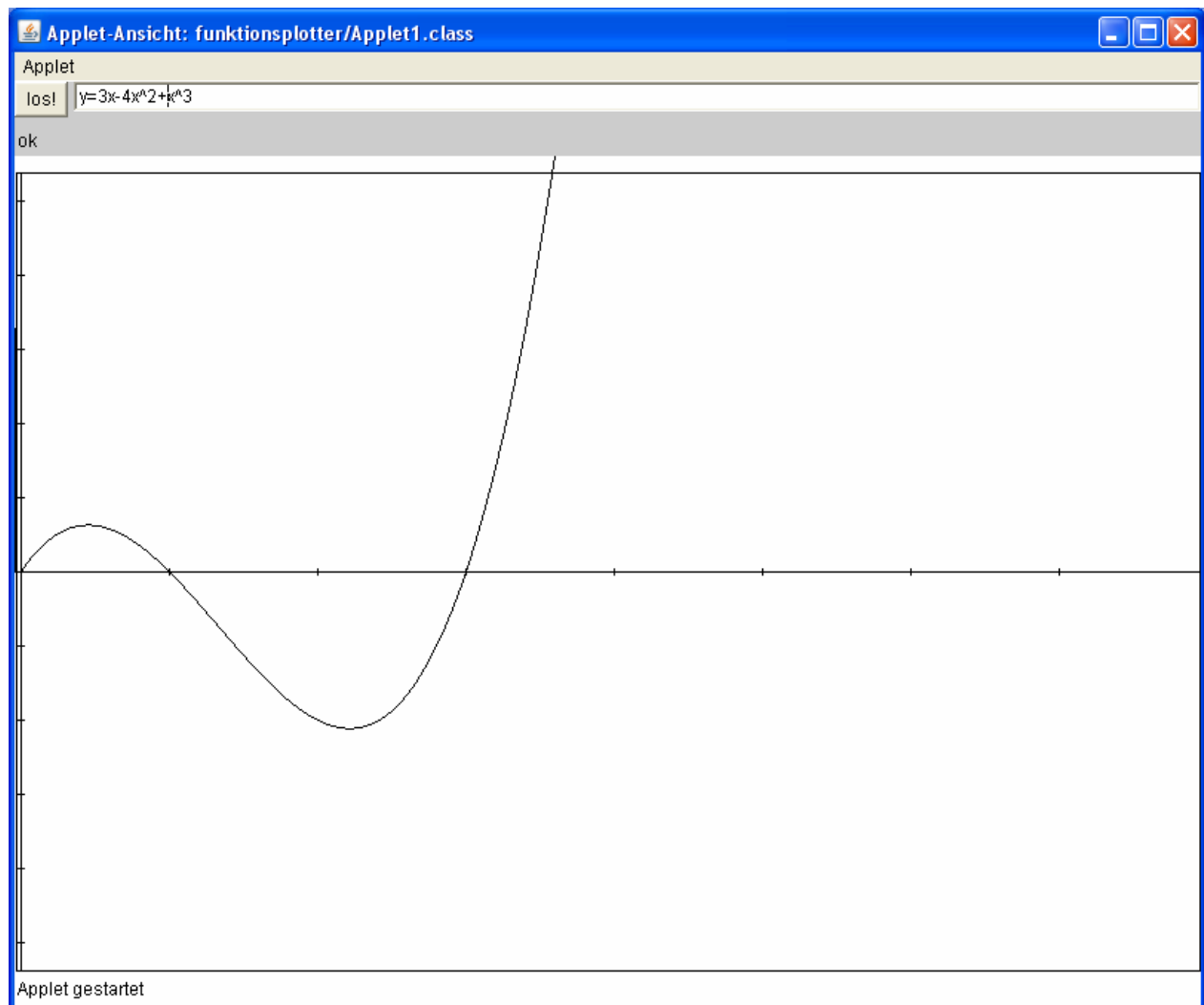
```
public void paint(Graphics g)
{
    int yalt = 0, yneu;
    anfangsbild(g);

    if (eingabeOK)
    {
        double x, y;
        for (int i = 0; i < breite - 1; i++) //mit allen horizontalen Bildschirmkoordinaten
        {
            x = 1.0 * (i - x0) / dx; //x-Wert bestimmen
            y = r.berechne(eingabe, x); //y-Wert berechnen

            if (!r.fehler)
            {
                if (ersterPunkt)
                {
                    yalt = y0 - (int) Math.round(y * dy); //Position merken
                    ersterPunkt = false;
                }
            }
        }
    }
}
```

```
else
{
  yneu = y0 - (int) Math.round(y * dy); //Linie zeichnen
  if ((Math.abs(yalt) < 2*hoehe) && (Math.abs(yneu) < 2*hoehe))
    g.drawLine(i - 1, yalt, i, yneu);
  yalt = yneu;
}
}
else ersterPunkt = true;
}
}
```

Das Ergebnis sieht dann so aus:.



5.7 Aufgaben

1. Erweitern Sie den Funktionsplotter um die Möglichkeit, **Zahlen mit Nachkommateil** einzugeben. Ändern Sie dazu die Syntaxdiagramme und die Grammatik des Parsers.
2. Die Berechnung der Funktionswerte erfolgt im Programm immer mithilfe der Logarithmusfunktion. Dadurch ist der Wertebereich des Arguments unnötig eingeschränkt. Versuchen Sie, mithilfe von Fallunterscheidungen die **Definitionsbereiche** der benutzten Terme zu ermitteln und dann so weit wie möglich Funktionswerte zu bestimmen. Verlegen Sie dann das Koordinatensystem in die Mitte des Bildschirms.
3. Führen Sie Möglichkeiten ein,
 - a: den **Ursprung des Koordinatensystems** zu verändern,
 - b: die **Skalierung der Achsen** zu verändern.
4. Die definierte Sprache akzeptiert nur rationale Funktionsterme. Führen Sie Möglichkeiten ein
 - a: **Trigonometrische Funktionen** zu benutzen,
 - b: **Exponentialfunktionen** zu benutzen,
 - c: **Logarithmusfunktionen** zu benutzen.
5. Der Funktionsplotter zeichnet nur den eigentlichen Funktionsgraph. Diskutieren Sie Möglichkeiten, bei Bedarf auch Graphen der **Ableitungsfunktionen** zu zeichnen. Realisieren Sie Ihre Ideen.
6. Wie kann man einfach **Funktionenscharen** zeichnen lassen? Führen Sie auch diese Möglichkeit ein.
7. Wenn sich mehrere Graphen auf dem Bildschirm befinden, dann sollten diese **farbig** unterschieden werden. Realisieren Sie diese Option.
8. In der beschriebenen Form zeichnet das Applet immer wieder neu in ein leeres Koordinatensystem. Verdeckt man zwischenzeitlich den Zeichenbereich, dann gehen die gezeichneten Bereiche verloren.
 - a: Führen Sie eine Möglichkeit ein, die schon gezeichneten Graphen stehen zu lassen, bis ein **Löschknopf** gedrückt wird.
 - b: Führen Sie einen **Bildschirmpuffer** ein, so dass der Zeichenbereich nicht verloren gehen kann.
9. In der realisierten Form interpretiert das Programm den Eingabestring bei jedem Zeichenvorgang neu. So werden z. B. für jeden gezeichneten Punkt des Graphen die Zahlen im Term neu ermittelt. Diskutieren Sie unterschiedliche Möglichkeiten, hier die Arbeit zu vereinfachen. Realisieren Sie eine davon.