Lehrerfortbildungsveranstaltung „Fachdidaktische Aspekte der theoretischen Informatik“ am 26.10.07 im FB Informatik der Hochschule Zittau/Görlitz in Görlitz

Material für TeilnehmerInnen: Vorschläge konkreter Unterrichtsplanungen (schriftlich), AtoCC und Dateien

Die folgenden Unterrichtshilfen sind als Anregung für Lehrende zu verstehen, die vor der (schwierigen) Aufgabe stehen, den Wahl-Lernbereich 8 A „Theoretische Informatik - Theoretische Grundlagen von Programmiersprachen“ in der Jahrgangsstufe 11/12 an einem Gymnasium Sachsens zu unterrichten. Ab dem Schuljahr 2008/9 ist dies auf der Basis des reformierten Lehrplans möglich.

Hierbei möchten wir Sie mit unserer Lehrerfortbildungsveranstaltung unterstützen. Das von uns entwickelte System AtoCC wird Ihnen bei der Behandlung der Themengebiete von Automaten bis hin zur automatisierten Compilergenerierung hilfreich zur Seite stehen. AtoCC wurde auf bedeutenden nationalen und internationalen Konferenzen vorgestellt und für die Lehre an Hochschulen und Gymnasien didaktisch erschlossen.

Die in diesem Material formulierten Ziele und Unterrichtsziele sind nicht in der üblichen Form nach Wissens- und Könnenszielen bzw. Kompetenzbereichen gegliedert. Gemäß Lehrplanforderung sollen *Einblicke in die genannten Gebiete* gegeben werden und *einige Wissensziele* erreicht werden. Könnensziele werden nicht formuliert.

In die vorliegende Planung sind Erfahrungen aus langjähriger Lehrtätigkeit in der theoretischen Informatik eingeflossen. Obwohl diese mit deutlicher Mehrheit im Hochschulbereich gesammelt wurden, lassen sich konkrete Empfehlungen für den Schulunterricht ableiten – keinesfalls kopieren.

Die große Herausforderung besteht darin, die eher abstrakten Inhalte der theoretischen Informatik in den praktischen Kontext des Compilerbaus zu stellen und dies alles innerhalb der vom Lehrplan vorgegebenen Zeit (sieben Doppelstunden) zu vermitteln.

Prof. Dr. Christian Wagenknecht
Dipl.-Inf.(FH) Michael Hielscher

1.+2. Stunde:

Ziele und Inhalte:

- ☐ Funktionsweise eines Compilers für eine Robotersprache (Motivation durch Grafik vs. Rechnen o. ä., s. Logo turtle geometry) erschließen
- ☐ Propädeutische Einführung der Begriffe: Compiler, Interpreter, Syntax, Semantik, Quellsprache, Zielsprache, Übersetzung

Vorbereitungen:

- ☐ Installieren von AtoCC (TDiag)
- ☐ Ghostscript installieren für PS2PDF
- ☐ Acrobat Reader ab Version 6.0 oder Foxit Reader installieren
- ☐ Beispielprojekt: Zeichenroboter → PDF bereitstellen
- ☐ Zettel mit Magneten/Pins für Tafel vorbereiten

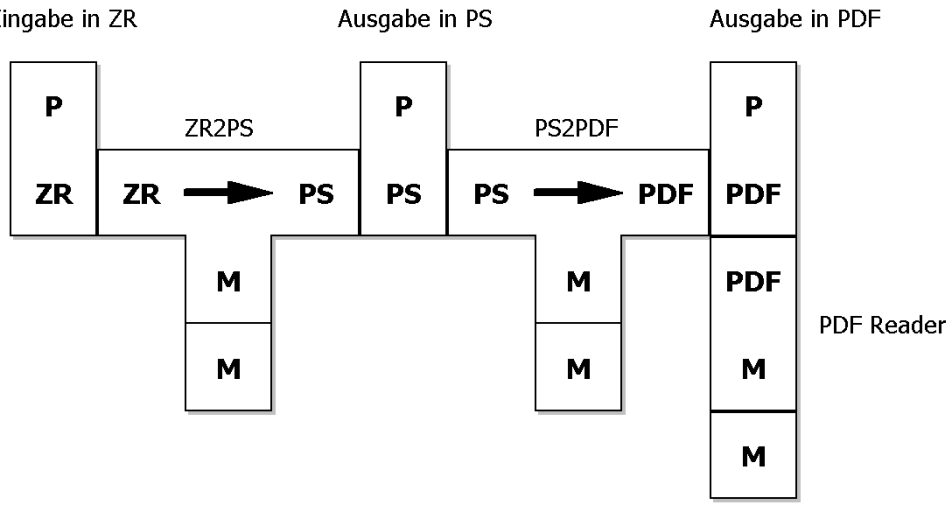
Unterrichtsablauf:

- ➢ *Einleitung*
  - ☐ Wenn wir bislang in unser Programmiersprache (z.B.: Delphi) gearbeitet haben, hat Delphi aus dem Quelltext, den wir geschrieben haben, ein ausführbares Programm erstellt. Delphi war also ein Übersetzer für die Sprache Object Pascal in die für den Computer verständliche Maschinensprache. Müssten wir in Maschinensprache programmieren, wäre das eine furchtbare Arbeit und wir müssten für einfachste Dinge lange menschenunwürdige Quelltexte schreiben.
  - ☐ Wir wollen nun herausfinden wie ein solcher Sprachübersetzer arbeitet und danach selbst einen solchen entwickeln. Dazu schauen wir uns zunächst ein Beispielprojekt an.
- ➢ *Zeichenrobotersprache wird in PS (PDF) übersetzt*
  - ☐ Wir schauen uns PDF's an, die erstellt wurden und vergleichen diese mit den Quelltexten geschrieben in ZR.
  - ☐ Dabei sollen die Schüler die Eingabesprache verstehen lernen, ohne theoretisches Grundwissen zu besitzen.
  - ☐ Vergleich des Aufbaus der natürlichen Sprache mit „Artikel Subjekt Adjektiv…" mit der Sprache des Zeichenroboters.
  - ☐ Experimente: Schüler verändern/erstellen Eingabewörter für den Zeichenroboter, übersetzen diese mit Hilfe des Compilers (Batch-Datei verwenden):
    > robo in1.txt out1.pdf
    Dabei entstehen semantische Fehler (andere Zeichnung als gewünscht) und syntaktische Fehler.
  - ☐ An der Tafel sollen die Schüler nun vorgefertigte Zettel verwenden, um den Aufbau eines Zeichenroboter-Programms zu beschreiben:
    Zettel: 2x PROGRAMM , 3x ZAHL , RE , VW , WH , [,] ;
    Fragen die dabei gestellt werden:
    Wie kann ein Programm beginnen? = RE , VW , WH
    Welche Bausteine müssen folgen? = ZAHL bzw. ZAHL [PROGRAMM]
    Was kann zwischen den zwei Klammern [ ] stehen? = wieder ein (Teil-)Programm

- ☐ Schüler betrachten einige Eingabewörter und prüfen, ob diese Eingabewörter zu Sprache gehören z.B.: **VW 100 10 RE** ; **WH [RE 10 VW 20]**. Begründung!
- ☐ Erweiterung der Roboter-Sprache um zwei Sprachelemente: STIFT, FARBE.
  Hieran soll nun der Unterschied zwischen Semantik und Syntax thematisiert werden.
  Ein Beispiel für ein Roboterprogramm mit STIFT und FARBE:
  WH 32 [STIFT 1 FARBE blau RE 90 VW 80
  WH 4 [FARBE rot STIFT 3 VW 40 RE 90] RE 10]
  Neue Sprachelemente: STIFT Zahl (Stiftbreite) und FARBE
  (rot|blau|gruen|gelb|schwarz) (Stiftfarbe) an die Tafel bringen (Zettel oder Kreide).
  Frage: Ist die Reihenfolge von STIFT und FARBE relevant? = nein
  Frage: Ist die Reihenfolge von VW und RE relevant? = ja, da anderes Bild entsteht
  Frage: Würde uns der Sprachübersetzer einen Fehler anzeigen wenn wir VW und RE vertauschen? = nein, da für ihn die Semantik keine Rolle spielt!

➢ *Begriffe erläutern und T-Diagramm erstellen:*

- ☐ Begriffe: Programm, Interpreter, Compiler, T-Diagramm einführen
- ☐ Entwickeln eines T-Diagramms für den Zeichenroboter-Compiler, PS2PDF Compiler (Ghostscript muss installiert sein!) und Arcobat Reader (Foxit) als Interpreter von PDF Dateien.



- ☐ Ausgabesprache PostScript betrachten und Schlussfolgerungen ziehen → Zeichen-robotersprache kurz, Ausgabe in PS lang (Vergleich zur bekannten Programmier-sprache und Maschinencodeübersetzung)
  Motivation für 2-Pass-Compilation: ZR->PDF-Compiler will niemand schreiben – PDF ist für den Menschen nicht lesbar, wohl aber PS (ist eine Programmiersprache)
- ☐ Zusammenfassung

3.+4. Stunde:

Ziele und Inhalte:

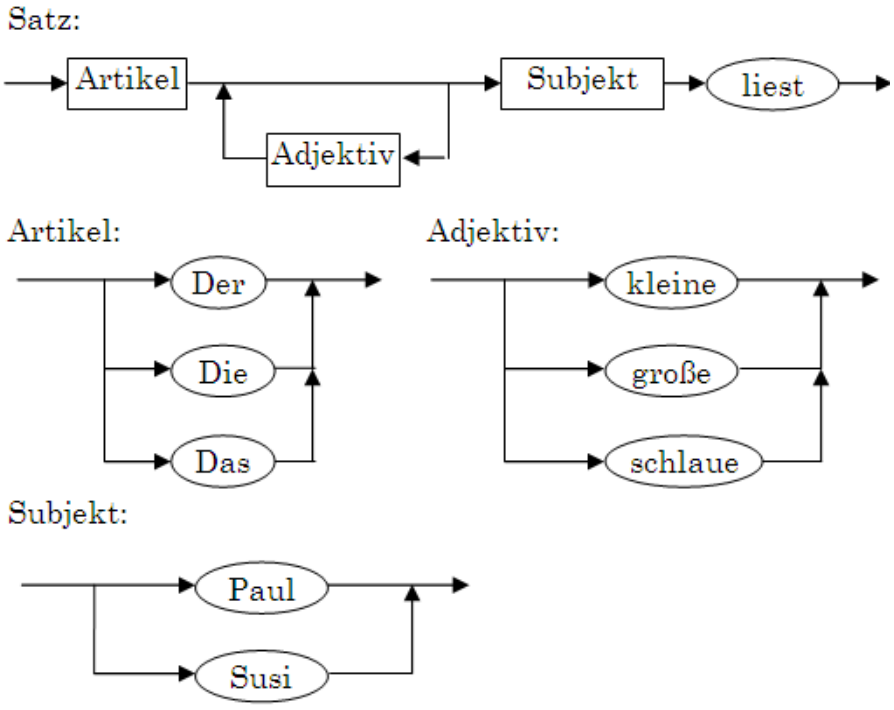
- ☐ Syntaxbeschreibung mit Syntaxdiagrammen
- ☐ Formale Grammatiken
- ☐ Ableitungsbegriff und Anwendung auf Wörter (Ableitungsbaum)

Vorbereitung:

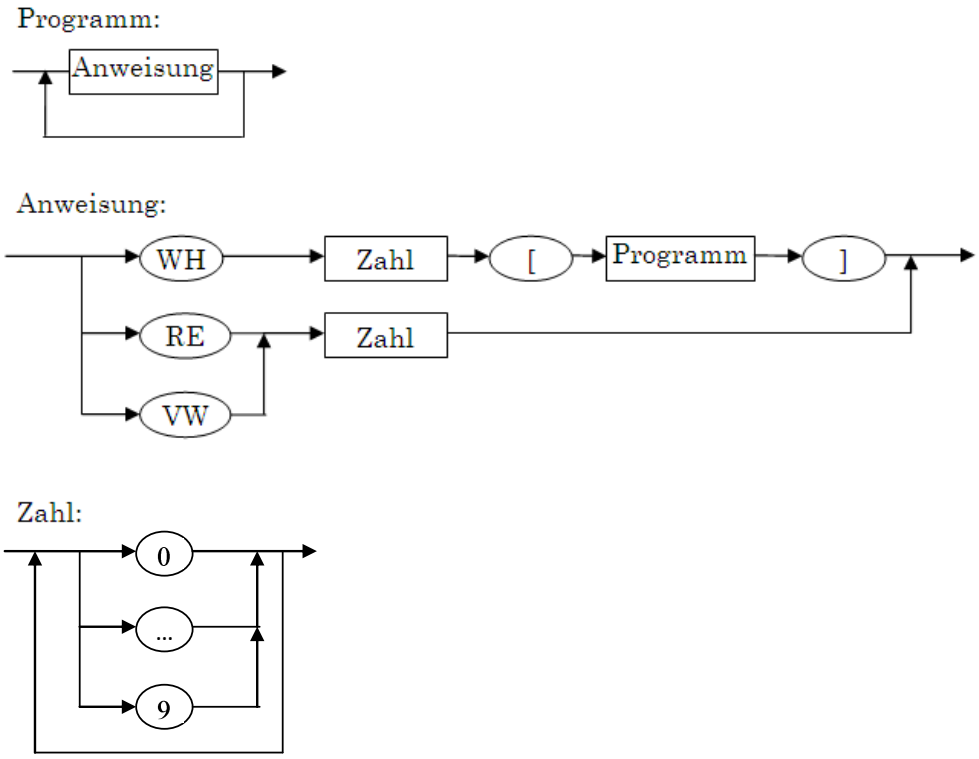
- ☐ Arbeitsblatt A2 - Syntaxdiagramm ausdrucken
- ☐ AtoCC-Grammatik-Editor

Unterrichtsablauf:

- ➢ *Einleitung*
  - ☐ Wir konnten in der letzten Stunde feststellen, dass Programmiersprachen eine ganze bestimmte Syntax aufweisen. Wir müssen die Syntax einer Sprache durch eine geeignete Darstellung bzw. Schreibweise definieren, damit auch andere sie verstehen können. Hierfür verwenden wir Syntaxdiagramme und später Grammatiken.
  - ☐ Die Schüler bekommen das Arbeitsblatt A2 und wir schauen uns zunächst die Bestandteile eines Syntaxdiagramms zur (grafischen) Beschreibung von Sprachen an.
- ➢ *Beispiel Syntaxdiagramm*



  - ☐ Diskussion: Welche Sätze können gebildet werden? Die Schüler sollen dabei die Wege entlang der Pfeile „ablaufen“ und das Diagramm schließlich verlassen.
    Wäre „Der große kleine große Paul liest“ ein Satz? = Ja, auch wenn er semantisch keinen Sinn macht.
  - ☐ Die Schüler versuchen nun für unsere Robotersprache ein Syntaxdiagramm zu zeichnen. Ein Beispiel:



➢ *Formale Grammatiken*

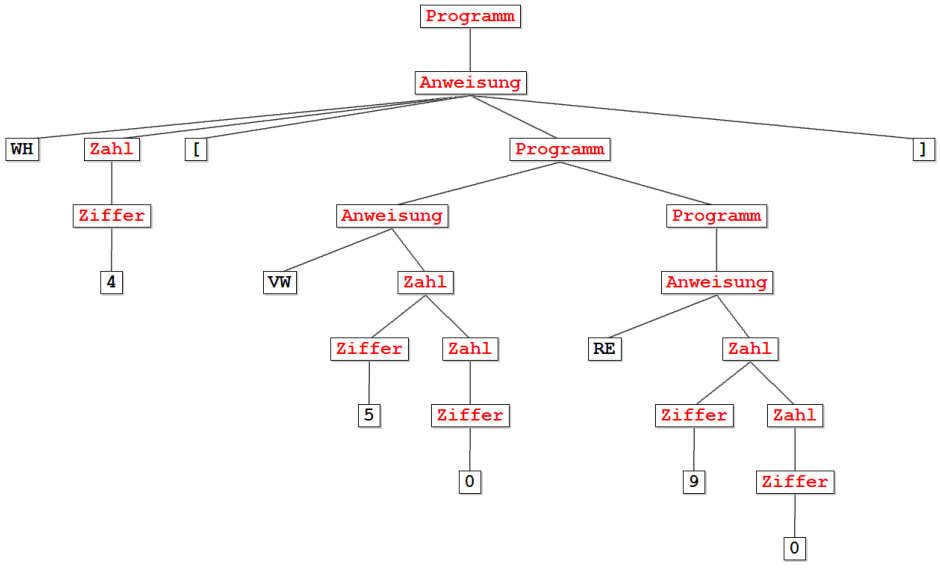
- ☐ Wir wollen uns nun einer weiteren Definitionsform von Sprachen zuwenden, den Grammatiken. Syntaxdiagramme können wir als Menschen gut ablesen und schnell verstehen, aber der Computer selbst kann mit unseren „Bildern“ nicht viel anfangen. Wir benötigen also eine verständliche Textbeschreibung um eine Sprache sowohl für uns als auch für den Computer verständlich zu machen.
- ☐ Die Terminale und Nichtterminale (wie wir sie im Syntaxdiagramm bereits verwendet haben) werden in der Grammatik als Mengen T und N angegeben.
  Für das Aufschreiben der Syntaxregeln verwenden wir eine eigene Sprache die BNF.
  Eine Wiederholung wie etwa bei Zahl wird durch eine Regel der Form:
  A → Ziffer A | Ziffer
  ausgedrückt. Der „|“ wird als ODER gelesen und entspricht der Alternative.
- ☐ Grammatik für Zeichenrobotersprache:

  G = (N,T,P,s) mit:
  N = { Programm, Anweisung, Zahl, Ziffer }
  T = { WH, VW, RE, [,], 0 , 1 … 9 }
  P = {

Programm	→	Anweisung | Anweisung Programm
Anweisung	→	WH Zahl [Programm] | RE Zahl | VW Zahl
Zahl	→	Ziffer Zahl | Ziffer
Ziffer	→	0 | 1 | … | 9

  }
  s = Programm
- ☐ Wir können nun die Grammatik um die Befehle FARBE und STIFT erweitern.
- ☐ Man kann nun auch für die BNF selbst eine Grammatik angeben (andeuten wie diese aussehen könnte). Schreib-, Sprechweise für die Regeln.

- ☐ Anhand solch einer Grammatik können wir, aber auch der Rechner, prüfen, ob ein Eingabewort syntaktisch korrekt ist, dass heißt zur Sprache gehört, die durch diese Grammatik beschrieben wird.
  Um herauszufinden, ob ein konkretes Wort zur Sprache gehört, müssen wir diese Ableiten. Hierfür werden wir uns Ableitungsbäume ansehen.
- ☐ Betrachten wir ein Eingabewort für unseren Zeichenroboter:
  WH 4 [VW 50 RE 90]
  In der Grammatik ist das Spitzensymbol „Programm“ angegeben. Wir beginnen also mit diesem Nichtterminal die Ableitung:
  Programm => **Anweisung** => WH **Zahl** [Programm] => WH **Ziffer** [Programm] => WH 4 [**Programm**] => WH 4 [**Anweisung** Programm] => WH 4 [VW **Zahl** Programm] => WH 4 [VW **Ziffer** Zahl Programm] => WH 4 [VW 5 **Zahl** Programm] => WH 4 [VW 5 **Ziffer** Programm] => WH 4 [VW 5 0 **Programm**] => WH 4 [VW 5 0 **Anweisung**] => WH 4 [VW 5 0 RE **Zahl**] => WH 4 [VW 5 0 RE **Ziffer** Zahl] => WH 4 [VW 5 0 RE 9 **Zahl**] => WH 4 [VW 5 0 RE 9 **Ziffer**] => WH 4 [VW 5 0 RE 9 0]
- ☐ In jedem Schritt haben wir das jeweils am weitesten links stehende Nichtterminal durch eine zugehörige rechte Regelseite ersetzt, bis schließlich nur noch Terminale vorhanden waren. Dies kann in der Tat auch in Sackgassen führen.
  → Verhalten eines PKW-Fahrers in Sackgassen ansprechen.
  Die Schüler sollten nun selbst versuchen, einige Wörter (auf Papier) abzuleiten, wie: **VW 10, VW RE 10** (Syntaxfehler –kein Wort aus ZR) oder **WH 8 [RE 45 VW 20]**
- ☐ Erklären der unterschiedlichen Ableitungsverfahren Top-Down und Buttom-Up.
- ☐ Die Ableitung eines Wortes kann auch in Form von Ableitungsbäumen dargestellt werden. Die Schüler sollen dazu die Grammatik in den Grammatik-Editor von AtoCC übertragen und einige Eingabewörter ableiten.
  Hinweise für den Lehrenden:
  1. Ableitungsbäume erhält man nur für kfS.
  2. Der Grammatik-Editor verwendet eine Eingabesprache, die etwas von der BNF abweicht: Terminale werden in Anführungszeichen eingeklammert.



- ☐ Zusammenfassung

5.+6. Stunde:

Ziele und Inhalte:

- ☐ Abstrakter Automat
- ☐ Kellerautomat als Akzeptor

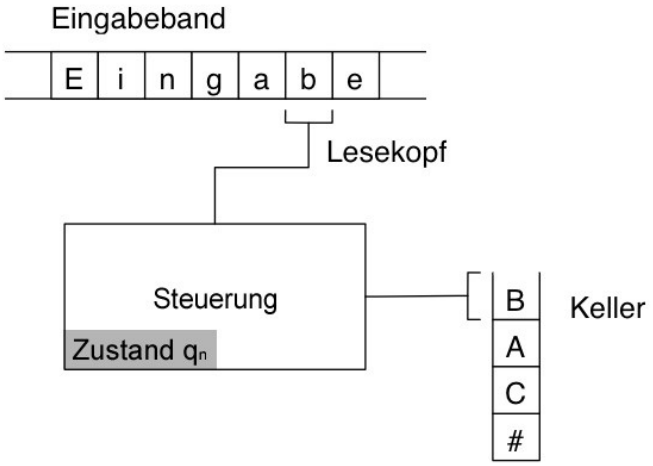
Vorbereitung:

- ☐ AtoCC-AutoEdit

Unterrichtsablauf:

- ➢ *Einleitung*
  - ☐ In der letzten Stunde haben wir über Sprachen und Grammatiken (die Sprachen beschreiben) gesprochen. Wir haben gesehen, dass man Wörter ableiten kann, um zu prüfen ob diese zur Sprache gehören oder nicht. Diese Prüfung haben wir auf Papier und auch mit dem Werkzeug Grammatik Editor durchgeführt.
    Wir wollen nun eine alternative Beschreibungsform formaler Sprachen kennenlernen – den abstrakten Automaten. Motivation: Ableitungsprozess abstrahieren.
  - ☐ Es gibt verschiedene Typen von Automaten die jeweils verschiede Typen von Sprachen beschreiben können. Wir wollen uns zunächst den Kellerautomaten (oder Stack-Automaten) ansehen, da er auch unsere Zeichenrobotersprache beschreiben kann. Motivation: rekursive Produktionsregeln → mit Kellerspeicher realisieren
  - ☐ Arbeitsweise eines Kellerautomat:
    Im nachfolgenden Diagramm sind die Bestandteile eines Kellerautomaten dargestellt. Das Eingabeband ist potentiell unendlich und ist in kleine Felder unterteilt in die jeweils genau ein Zeichen geschrieben wird. Die erlaubten Zeichen auf dem Band werden durch das Eingabealphabet (einer Menge von Zeichen) definiert. Obwohl Kellerautomaten abstrakte Objekte unseres Geistes sind, können wir uns deren Arbeitsweise wie die eines kleinen Roboters vorstellen, der das Eingabeband entlang fährt und immer genau ein Eingabezeichen anschauen kann. Außerdem besitzt der Automat noch einen Hilfsspeicher in Form eines Kellers oder Stapels. Der Automat liest und entfernt hier immer genau das oberste Zeichen des Stapels. Der Kellerautomat besitzt n Zustände und befindet sich zu jedem Zeitpunkt immer in genau einem Zustand q_n. Aus den 3 Informationen: aktuelles Eingabezeichen am Lesekopf, aktuelles Kellerzeichen (TopOfStack) und den aktuellen Zustand q_n wird entschieden, in welchen Folgezustand q_m der Automat wechselt und welches Wort er auf den Stapel schreibt.

    Der Automat besitzt eine endliche Menge von Zuständen Q. Die Menge E der Endzustände ist eine Teilmenge von Q. Befindet sich der Automat in einem Enzustand wenn das Eingabewort vollständig gelesen wurde, dann gehört das Eingabewort zur Sprache S, die dieser Automat beschreibt. Der am Ende der Bearbeitung verbleibende Kellerinhalt spielt keine Rolle. Stoppt der betrachtete Kellerautomat in einem Nichtendzustand, gehört das Wort nicht zu S.

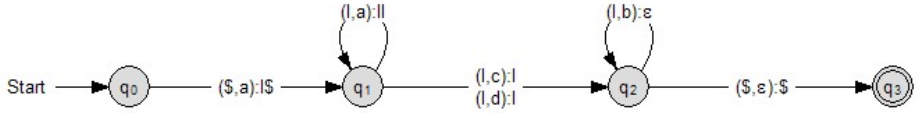


☐ Beispiel-Kellerautomat für die durch folgende Grammatik bestimmte dkfS.

G = (N, T,P,s), N = {S}, T = {a, b, c}, P = {S → a S b | c | d}

Vorgehensweise:

1. Grammatik „ausschroten“, d.h. Wörter / Nichtwörter bilden, ...
2. Plan entwerfen: umgangssprachlich formulieren
3. Automatentyp festlegen (DKA)
4. Alphabete eingeben (Eingabe, Keller)
5. Überführungsfunktion definieren (Tabelle/Graph)
6. Simulation für ausgewählte Eingabewörter



$M = (\{q_0,q_1,q_2,q_3\},\{a,b,c,d\},\{\$,I\},\delta,q_0,\$,\{q_3\})$

$\delta(q_0,a,\$) = (q_1,I\$)$
$\delta(q_1,a,I) = (q_1,II)$
$\delta(q_1,c,I) = (q_2,I)$
$\delta(q_1,d,I) = (q_2,I)$
$\delta(q_2,b,I) = (q_2,\varepsilon)$
$\delta(q_2,\varepsilon,\$) = (q_3,\$)$

DKA01.XML

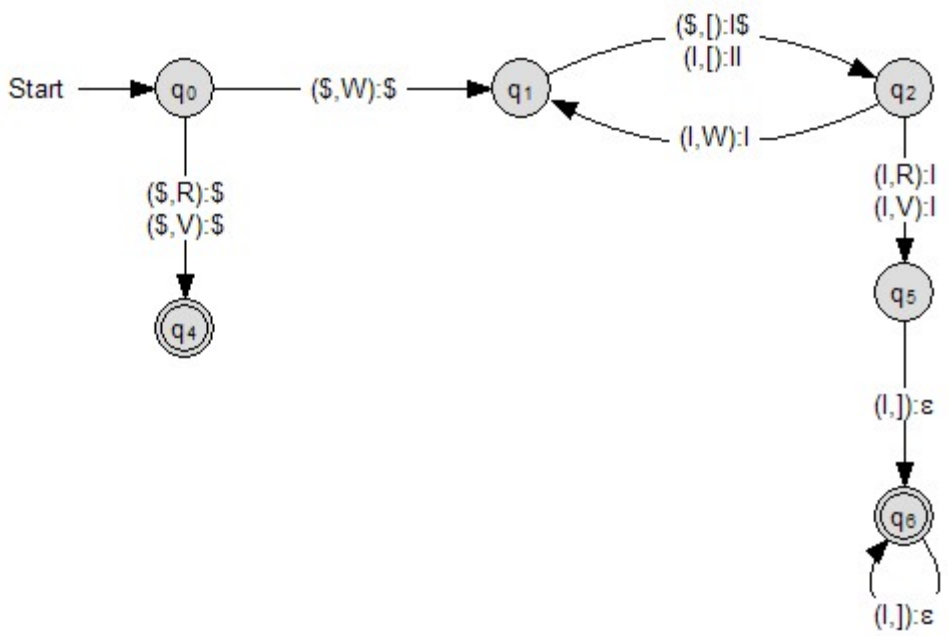
ÜA: Entwerfen Sie einen Kellerautomaten für die folgenden Sprachen:

$L_1 = \{a^n b^n, n>0\}$ und L_2={w | w ... Palindrom über {a,b} mit durch x markierter Wortmitte}, z.B. abbxbba und baxab

ÜA: Obige Grammatik leicht erweitern in Richtung auf ZR.

G = (N, T, P, s), N = {P}, T = {W, [,], R, V}, P = {P → W [P] | R | V}

Lösung: s. nächste Seite



$M = (\{q_0,q_1,q_2,q_6,q_4,q_5\},\{[,],R,V,W\},\{\$,I\},\delta,q_0,\$,\{q_6,q_4\})$

$\delta(q_0,W,\$) = (q_1,\$)$
$\delta(q_0,R,\$) = (q_4,\$)$
$\delta(q_0,V,\$) = (q_4,\$)$
$\delta(q_1,[,\$) = (q_2,I\$)$
$\delta(q_1,[,I) = (q_2,II)$
$\delta(q_2,R,I) = (q_5,I)$
$\delta(q_2,V,I) = (q_5,I)$
$\delta(q_2,W,I) = (q_1,I)$
$\delta(q_6,],I) = (q_6,\varepsilon)$
$\delta(q_5,],I) = (q_6,\varepsilon)$

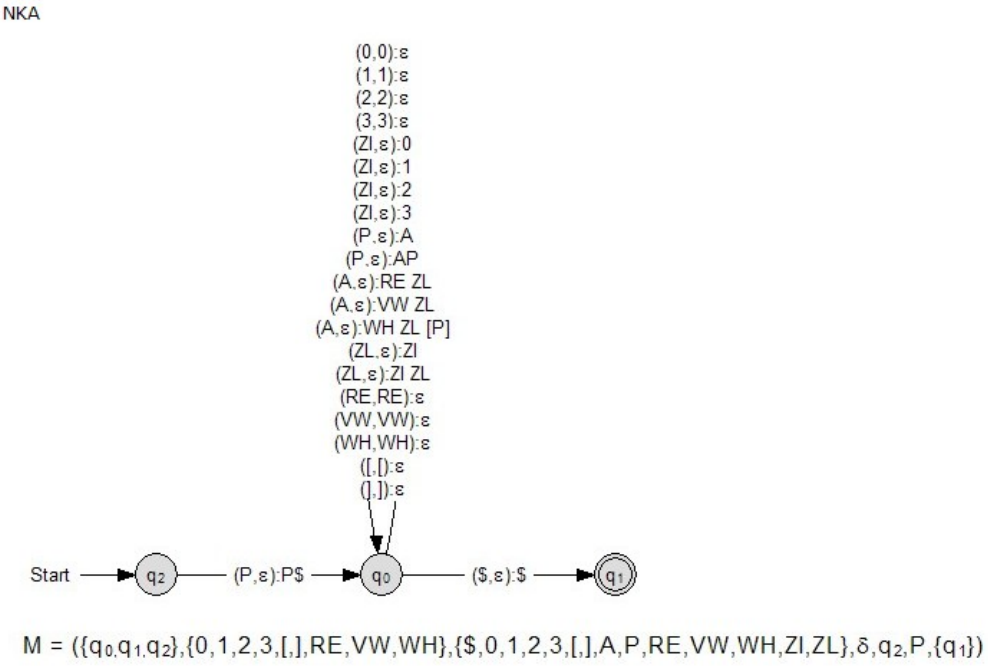
DKA02.XML

Hinweis für Lehrende:

Ein Kellerautomat für die (volle) Sprache ZR wird durch eine Transformation der kfG in den zugehörigen NKA gewonnen. Die nichtdeterministischen Automatentypen werden hier jedoch nicht thematisiert.

Im Folgenden ist ein solcher NKA angegeben. Die Sprache wurde ein wenig eingeschränkt: Es sind nur Zahlen zugelassen, die nur aus beliebig vielen Ziffern 0,1,2,3 bestehen.

Die Simulation z.B. von WH 3 [VW 10 RE 32] ist sehr eindrucksvoll, erfordert aber die Einführung des Konzeptes des Nichtdeterminismus‘.



$M = (\{q_0,q_1,q_2\},\{0,1,2,3,[,],RE,VW,WH\},\{\$,0,1,2,3,[,],A,P,RE,VW,WH,Zl,ZL\},\delta,q_2,P,\{q_1\})$

NKA01.XML

Außerdem ergibt sich eine gute Motivation für ZR mit VCC, da offensichtlich ein recht komplexer Kellerautomat vom Compiler-Generator erzeugt werden muss.

7.+8. Stunde:

Ziele und Inhalte:

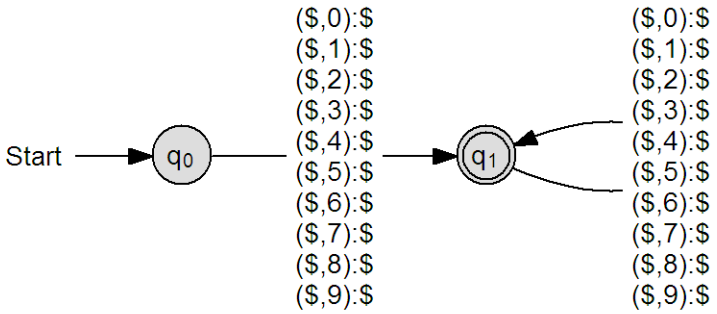
- ☐ Endlicher Automat als Akzeptor

Vorbereitung:

- ☐ AtoCC-AutoEdit

Unterrichtsablauf:

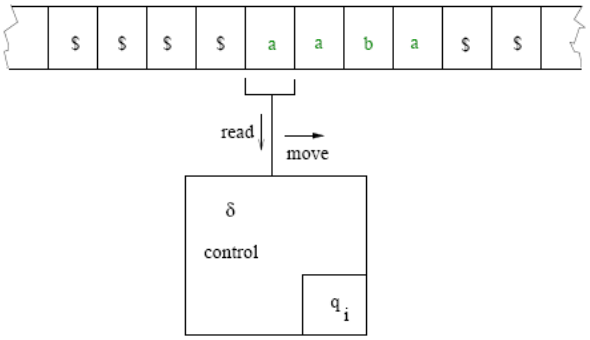
- ➢ *Einleitung*
  - ☐ Motivation: In Sprache ZR des Roboters gibt es ganz einfache Teilsprachen, z.B. für Zahl: Zahl → 0 | 1 | ... | 9 | 0 Zahl | 1 Zahl | ... | 9 Zahl
    Haben sämtliche Produktionen einer Grammatik diese Gestalt, so spricht man von einer regulären Grammatik/Sprache. (evtl. Hinweis auf Transformation der Regeln in diese Gestalt) Schüler geben zunächst einen Kellerautomaten an → Keller unnötig.



Der Zustand q1 ist notwendig, um zu verhindern, dass der Automat das leere Wort akzeptiert.

- ☐ Für ZR gilt dies nicht (Schüler sollen das versuchen!)
- ☐ Für reguläre Sprachen gibt es abstrakte Automaten, die viel einfacher sind als Kellerautomaten. Sie heißen endliche Automaten. Arbeitsweise, Definition.

Endlicher Automat (*abstrakter* Automat - Bitte nicht nachbauen!)



Start: in q_0

Stopp: in q_i, wenn Eingabewort vollständig gescannt

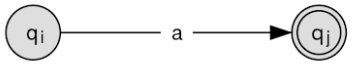
Arbeitstakt: read, Folgezustand, move

$q_i \in E$: Wort wird akzeptiert
$q_i \notin E$: Wort wird nicht akzeptiert

Totale Funktion δ, mit $\delta(q_i, a) = q_j$

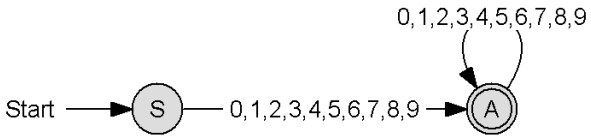
δ	a_0	a_1	...	a_n
q_0	...	q_r	...	
q_1	...			
⋮	⋮	...		
q_m	...			

Jedes Tabellenfeld enthält genau einen Eintrag (Zustand).



$q_j \in E$

- ☐ DEA für Zahl.



- ☐ Hier ist ein weiteres Beispiel für eine reguläre Sprache:
  G = (N,T,P,s) N = {S, A, B}, T = {a, b}, s = S,
  P = {S → a A | b B ; A → a A | a ; B → b B | b}
  Vorgehensweise:
  1. Vertrautmachen mit der Grammatik: Wörter bilden, auch extreme
  2. DEA nicht aus G transformieren, sondern auf der Grundlage der Sprache, da sonst NEA

➢ *Beispiele für reguläre Sprachen (ÜA, s. auch AutoEdit Workbook)*

- ☐ L = {a^nb, n≥0}, L = {a^nb, n>0}, Wörter über {a,b}*, die eine gerade Anzahl von a's und eine gerade Anzahl von b's enthalten (Kunstsprache)
- ☐ Praxis: Variablennamen in Programmiersprachen (echte Sprache – Modellierung!); Sprachen der Telefonnummern, Kfz-Kennzeichen, E-Mail-Adressen
- ☐ Konstruktion einer zugehörigen Grammatik G aus einem gegebenen DEA: Transformation der Regeln q_x –a– > q_y zu X → a Y und falls q_y ε E gilt zusätzlich X → a.

➢ Reguläre *Ausdrücke (RA)* ...
... sind nicht Lehrplaninhalt. Es ist aber möglich, dass Schüler/innen RA aus praktischen Anwendungen kennen. Dann kann kurz darauf hingewiesen werden, dass dies eine weitere alternative Beschreibungsform für reguläre Sprachen ist.

9.+10. Stunde:

Ziele und Inhalte:

- ☐ Token, Scanner, Parser
- ☐ ZR2PDF-Compiler
- ☐ Automatisierte Compilerentwicklung (konzeptionelles Verständnis von Übersetzungsprozessen für Sprachen)

Vorbereitung:

- ☐ AtoCC-VCC
- ☐ ZR2PS.xml bereitstellen

Unterrichtsablauf:

- ☐ Schüler finden reguläre Subsprachen → Token des Compilers
- ☐ Ein Kellerautomat für ZR kann dramatisch vereinfacht werden, wenn Subsprachen als Terminale angesehen werden. So geht man im Compilerbau vor und spricht von Token = Syntaktische Einheiten, die von einem sog. Scanner in einem dem Parsing vorausgehenden Pass erkannt und für den Parser als Terminale angesehen werden.
- ☐ Dadurch entsteht eine reduzierte Grammatik (zunächst ohne STIFT und FARBE).

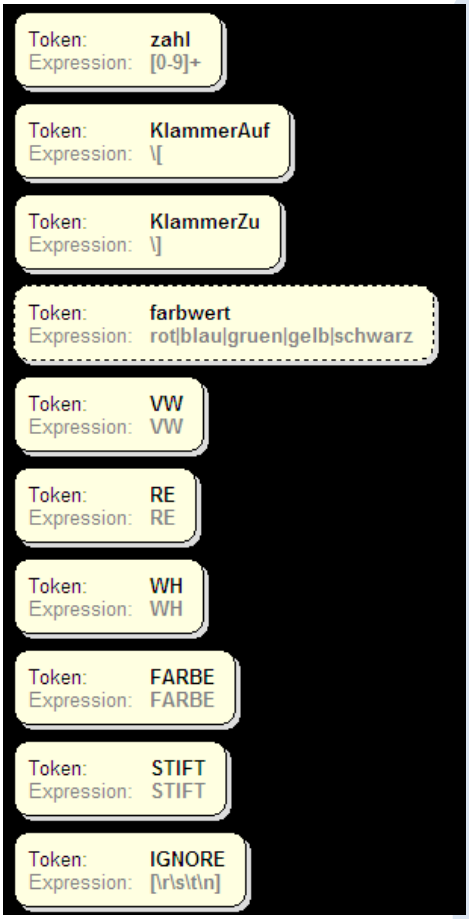
  Quellgrammatik:

  G = (N,T,P,s) mit:
  N = { Programm, Anweisung, Zahl, Ziffer}
  T = { WH, VW, RE, [,], 0 , 1 … 9}
  P = {
  Programm → Anweisung | Anweisung Programm
  Anweisung → WH Zahl [Programm] | RE Zahl | VW Zahl
  Zahl → Ziffer Zahl | Ziffer
  Ziffer → 0 | 1 | … | 9
  }
  s = Programm

  wird zu:
  G = (N,T,P,s) mit:
  N = { Programm, Anweisung }
  T = { WH, VW, RE, [,], **Zahl** }
  P = {
  Programm → Anweisung | Anweisung Programm
  Anweisung → WH **Zahl** [Programm] | RE **Zahl** | VW **Zahl**
  }
  s = Programm
  und
  [0-9]+ → regulärer Ausdruck für **Zahl**

  BNF für G(reduziert) und G1, G2, ... für die regulären Subsprachen

Die Reduktion wird an der Tafel vorgeführt. Anschließend können wir die Grammatik mit FARBE und STIFT verwenden und auch hier eine reguläre Subsprachen für FarbWert → (rot|blau|gruen|gelb|scharz) aufstellen.



➢ *Entwicklung des ZR2PS Compilers in VCC:*

☐ Wir öffnen die Datei ZR2PS.xml in VCC um das Grundgerüst für die PostScript Ausgabe später nicht mehr per Hand schreiben zu müssen.

☐ Entwicklung des Scanners mit Token:
Ein Token in VCC besteht aus Tokenname + Expression (regulärer Ausdruck). Die Terminale der reduzierten Grammatik werden direkt übertragen:

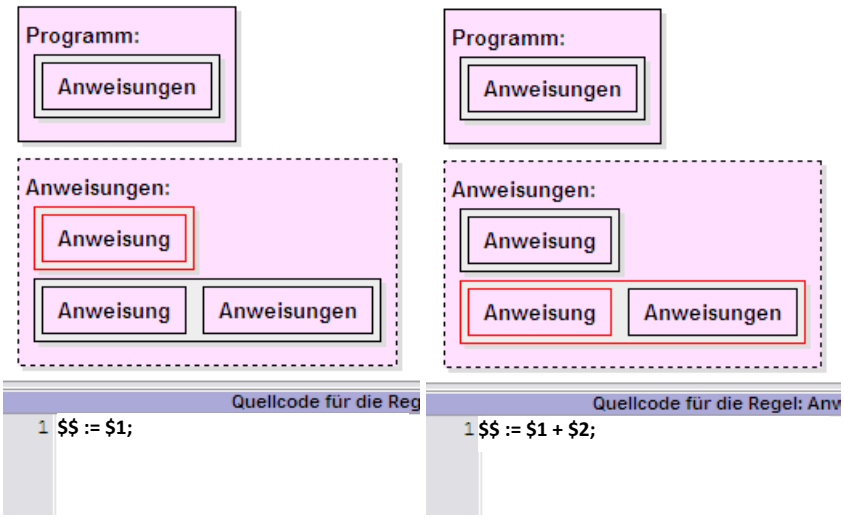
Hinweise:
Metazeichen in regulären Ausdrücken und entsprechendes „escapen“ beachten. Folgende Zeichen benötigen ein vorangestellten „\“ :
( ) [ ] { } + - ? * . $ ^ \ |

Das Token **IGNORE** ist ein Spezialtoken. Alle Zeichen die auf dieses Token passen werden im Eingabestrom ignoriert.
\r = Return, \s = Space, \t = Tab,
\n = Linefeed
Ein IGNORE Token muss nicht verwendet werden, erlaubt aber eine bessere Lesbarkeit von Eingabetexten (durch zusätzliche Formatierungsmöglichkeiten)

☐ Die Parserbeschreibung in VCC entspricht optisch im weitesten Sinne einem Syntaxdiagramm. Die Produktionsregeln der Grammatik können 1:1 übertragen werden. Da wir durch ZR2PS.xml einige Vorgaben für das Spitzensymbol und das Nichtterminal Anweisungen entwickeln wir ab Anweisung weiter.

Anweisung → WH Zahl [Anweisungen]
| RE Zahl
| VW Zahl
| …

☐ Allgemeine Hinweise zur Entwicklung von S-Attribute:
Für jede rechte Regelseite kann ein S-Attribut definiert werden. Das Ergebnis dieser Regel wird im Platzhalter $$ definiert und an die nächst höhere Regel gegeben (Ableitungsbaum).



Die synthetisierten Inhalte der Elemente einer rechten Regelseite werden mit Platzhaltern $1 bis $n definiert und können als Variablen verwendet werden.

Alle $ Variablen (auch $$) sind vom Datentyp String! Soll ein String in eine Zahl umgewandelt werden, ist eine Umwandlungsfunktion wie:
StrToInt($n);** (Delphi) oder **Int32.Parse($n); (C#) oder **Integer.parseInt($n);** (Java) nötig.

Die Zuweisung an $$ erfordert umgekehrt einen String:
$$:= IntToStr(5+10); (Delphi)
$$ = (5+10).ToString(); (C#)
$$ = Integer.toString(5+10); (Java) oder kurz: **$$ = "" +(5+10);** (in C# und Java)

In der Regel werden die Ergebnisse der einzelnen S-Attribute über $$ bis ganz hinauf zum Spitzensymbol gereicht. Es eignet sich deshalb ein Spitzensymbol mit nur einer rechten Regelseite um das finale S-Attribut (welches meist für die Codeausgabe verantwortlich ist) nicht mehrfach definieren zu müssen.

Ausgabe in die Zieldatei erfolgt über die Anweisung:
Output.WriteLine($1); (Delphi und C#) oder **Output.println($1);** (Java)

- *Entwicklung von S-Attribute für ZR2PS Compiler:*
  - Da PostScript eine Stack-Programmiersprache ist, müssen wir auch Parameter den Befehlen voranstellen. Durch die VCC Vorgabe (bzw. PS selbst) sind die Funktionen: **n draw**; **n turn**; **x y goto**; **n setlinewidth**; **r g b setrgbcolor** bereits in PS vordefiniert und können verwendet werden.

    Für eine Regel wie „Anweisung → VW Zahl“ entsteht ein einfaches S-Attribut der Form: $$:= $2+' draw '; Der Wert des Nichtterminals Zahl wird der Funktion draw vorangestellt. Dabei sind die Leerzeichen wichtig, da sonst die Befehle in der Zieldatei direkt aneinander geschrieben würden.

Für die Stiftfarbe Anweisung → FARBE farbwert benötigen wir eine Übersetzung aus den Farbwörtern in RGB Werte wie:

```
if ($2 = 'blau')      then $$ := '0 0 255 setrgbcolor ';
if ($2 = 'rot')       then $$ := '255 0 0 setrgbcolor ';
if ($2 = 'gruen')     then $$ := '0 255 0 setrgbcolor ';
if ($2 = 'gelb')      then $$ := '255 255 0 setrgbcolor ';
if ($2 = 'schwarz')   then $$ := '0 0 0 setrgbcolor ';
```

Die Wiederholung WH wird durch eine einfache FOR Schleife verwirklicht:
Anweisung → WH Zahl [Anweisungen]

```
$$ := '';
for i := 1 to StrToInt($2) do $$ := $$ + $4;
```

(unter Global eine Variable “var i : Integer; “ definieren für die Schleife)

- ☐ Übung: Befehlsnamen modifizieren etc. (Zielstellung: Überprüfung ob das Prinzip von Scanner, Token, Parser, S-Attributen verstanden wurde.)

11.+12. Stunde:

Ziele und Inhalte:

- ☐ Komplexe Übungen

Vorbereitung:

- ☐ Mozilla FireFox oder Adobe SVG Plugin für Internet Explorer für die Betrachtung von SVG Dateien installieren
- ☐ Arbeitsblatt A6a und A6b verteilen
- ☐ BeebImport.txt bzw. SVGGrundgerüst.txt bereitstellen

Unterrichtsablauf:

- ➢ *Einleitung*
  - ☐ Heute sollen die Schüler praktisch Anwenden was sie gelernt haben.
    Dazu soll 2 Projekte umgesetzt werden (Arbeitsblättern sind für C# gestaltet, Lösungen in Delphi liegen aber auch im Lösungsordner in Woche 6 vor):
  - ☐ Projekt 1: Entwickeln eines Interpreters für einfache Tonfolgen (vergleichbar mit alten Handyklingeltönen). In diesem Beispiel werden die Befehle direkt bei Auftreten ausgeführt und nicht erst synthetisiert → NotenInterpreter.
    Das Arbeitsblatt A6a enthält die entsprechenden Informationen für die Schüler.
    Es wird praktisch nur ein S-Attribut benötigt welches entsprechend des Notennamens (C0, E1,…) und Länge (1,2,4,8) einen zugehörigen Beep Befehl ausführt.
    Da VCC stets mit String Variablen arbeitet, müssen Platzhalter explizit in Int32 umgewandelt werden mit etwa: **Beep(264,1000/Int32.Parse($2));**
  - ☐ Projekt 2: Die selbstdefinierte Notensprache soll nun in eine Grafik (Notenzeile) übersetzt werden → NotenCompiler.
    Das Arbeitsblatt A6b enthält die nötigen Informationen für die Schüler über die zu verwendenden SVG Befehle. Das SVG Grundgerüst muss eingefügt werden und sollte gegebenenfalls unter Anleitung erfolgen.
    Die Verwendung einer globalen Variable (X Position der aktuellen Note) muss thematisiert werden (da bislang noch nicht verwendet).
    Die Aufgabe kann schrittweise aufgebaut werden – zunächst nur Noten ohne Beachtung des Notenwerts, anschließend Notenhals für viertel und achtel Noten, dann Fähnchen hinzufügen für achtel Note.
  - ☐ Denkbar ist auch die Entwicklung von Projekt 1 als Hausaufgabe aufzugeben und zu Beginn der Stunde nur mit einem fertigen NotenInterpreter zu arbeiten.

13.+14. Stunde:

Ziele und Inhalte:

- ☐ Turing-Maschine
- ☐ Chomsky-Hierarchie und Zusammenfassung

Vorbereitung:

- ☐ AtoCC-AutoEdit

Unterrichtsablauf:

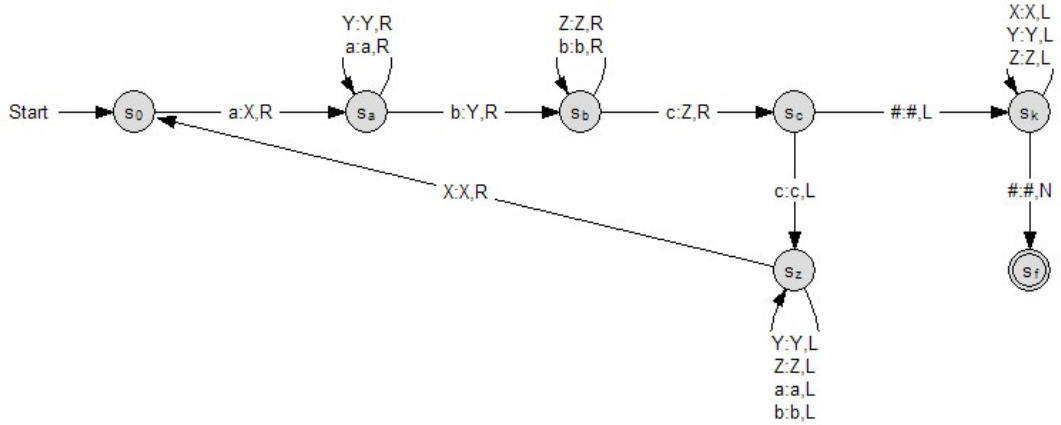
- ☐ Verschiedene Sprachklassen kennengelernt → Systematisierungswunsch → Chomsky-Hierachie: Endliche Automaten, Kellerautomaten.
- ☐ Es gibt sogar Sprachen, die nicht durch Kellerautomaten erkannt werden können: $L = \{a^n b^n c^n, n>0\}$ – diskutieren, begründen
- ☐ Evtl.: Einführung Turing-Maschine, einfaches Beispiel (nur, um Arbeitsweise vorzustellen) – Vorsicht mit TM zur Berechnung von Funktionen; Akzeptor!!!

Definition

Eine **Turingmaschine** (TM) ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \$, E)$.

Q ... endliche Menge von Zuständen
Σ ... Eingabealphabet
Γ ... Bandalphabet, wobei $\Sigma \subseteq \Gamma \setminus \{\$\}$ erlaubt ist
δ ... partielle Überführungsfunktion
q_0 ... Anfangszustand, $q_0 \in Q$
\$ ... Bandvorbelegungszeichen, kurz: Blankzeichen, mit $\$ \in \Gamma$ und $\$ \notin \Sigma$
E ... endliche Menge von Endzuständen $E \subseteq Q$

- ☐ Verwendung der TM als Akzeptor für L



$M = (\{s_0,s_a,s_b,s_c,s_k,s_z,s_f\},\{a,b,c\},\{\#,a,b,c,X,Y,Z\},\delta,s_0,\#,\{s_f\})$

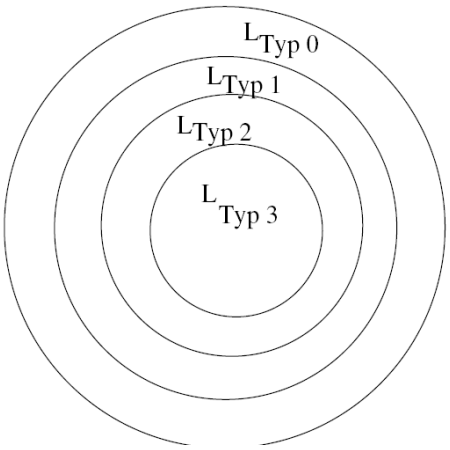
δ	#	a	b	c	X	Y	Z
s_0	-	(s_a,X,R)	-	-	-	-	-
s_a	-	(s_a,a,R)	(s_b,Y,R)	-	-	(s_a,Y,R)	-
s_b	-	-	(s_b,b,R)	(s_c,Z,R)	-	-	(s_b,Z,R)
s_c	$(s_k,\#,L)$	-	-	(s_z,c,L)	-	-	-
s_k	$(s_f,\#,N)$	-	-	-	(s_k,X,L)	(s_k,Y,L)	(s_k,Z,L)
s_z	-	(s_z,a,L)	(s_z,b,L)	-	(s_0,X,R)	(s_z,Y,L)	(s_z,Z,L)
s_f	-	-	-	-	-	-	-

DTM01.XML

Allg. Regelgestalt: $\alpha \rightarrow \beta \mid \alpha \in (N \cup T)^* \setminus T^*$ und $\beta \in (N \cup T)^*$

Typ	Bezeichnung	Regelgestalt
0	unbeschränkt (Phrasenstrukturgrammatik; Semi-Thue-System – Norwegischer Mathematiker A. THUE, 1863-1922)	keine Einschränkung
1	kontextsensitiv	längenmonotone Regeln, d.h. $\|\alpha\| \leq \|\beta\|$; Ausnahme: $s \rightarrow \varepsilon$ darf vorkommen, allerdings nur dann, wenn s in keiner Regel auf der rechten Seite vorkommt
2	kontextfrei	wie Typ 1 aber zusätzlich $\alpha \in N$; Regeln der Form $\alpha \rightarrow \varepsilon$ sind erlaubt
3	regulär	wie Typ 2 aber zusätzlich haben *alle* Regeln *entweder* die Form $\alpha \rightarrow x$ und ggf. $\alpha \rightarrow xA$ (rechtslinear) *oder* $\alpha \rightarrow x$ und ggf. $\alpha \rightarrow Ax$ (linkslinear), wobei $x \in T$ und $A \in N$

☐ Vervollständigung der Chomsky-Hierarchie (Grammatik-Typ und Automatentyp)



Grammatiktyp	Automatentyp
0	Turingmaschine (DTM, NTM)
1	Linear begrenzte NTM
2	NKA (DKA für dkfS in kfS enthalten)
3	EA (DEA und NEA)

☐ Zusammenfassung für die gesamten TI Inhalte.