Leserbrief zu „Ein Ausflug in den Compilerbau“ von Torsten Brandes (LOG IN Heft Nr. 156 (2009), S. 59-64)

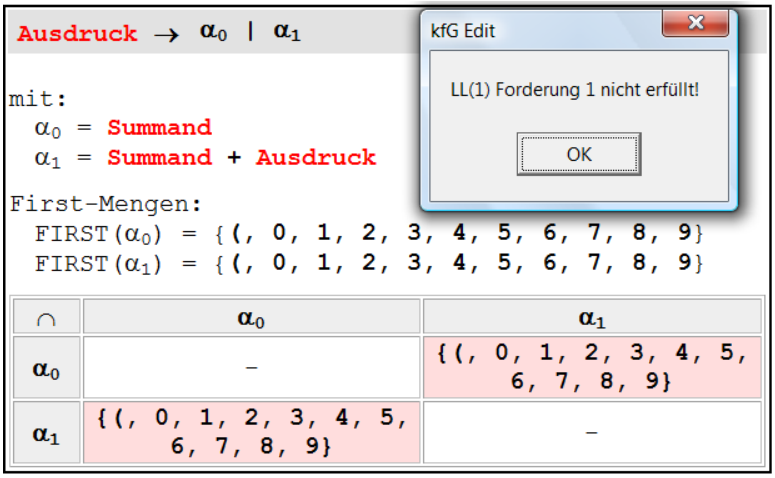
Herr Brandes hat einen Beitrag vorgelegt, der die Motivation für Theoretische Informatik (TI), insbesondere formale Sprachen, aus dem Compilerbau gewinnt. Dies entspricht auch der Sicht von Wagenknecht und Hielscher, die in /1/ dieses Potenzial für den Informatikunterricht herausgestellt haben. In /2/ wird ein Kurs zur Einführung formaler Sprachen und abstrakter Automaten mit Anwendungen im automatisierten Compilerbau beschrieben. Dabei wird eine Lern- und Arbeitsumgebung AtoCC (http://www.atocc.de) einbezogen.

Neben der Vermittlung von TI-Inhalten verfolgt Herr Brandes weitere Lehrziele, die die Programmierung mit Java betreffen. Für die betrachteten Syntaxanalyseverfahren und Berechnungen werden folglich Java-Programme entwickelt. Dieses Vorgehen birgt offensichtlich die Gefahr, dass Modellierungsaspekte zugunsten der Implementierungsaufgabe in den Hintergrund treten. Auch im vorliegenden Beitrag findet man solche Stellen, deren konzeptionelle Betrachtung durch „geschickte Implementierung“ überschattet wird. Wir möchten hier zur Verwendung von AtoCC anregen, um den Modellierungsaspekt deutlicher hervorzuheben. Die von Brandes erarbeiteten Java-Programme werden dabei einbezogen. Weiter unten findet sich außerdem eine fast Java-freie Lösung.

Der Autor des o.g. Basisartikels geht in von folgender Grammatik aus.

```
G = (N, T, P, s)
N = {Ausdruck, Summand, Faktor, Ziffer}
T = {+, *, (, ), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
P = { Ausdruck -> Summand | Summand + Ausdruck | Summand
      Summand  -> Faktor | Faktor * Summand
      Faktor   -> Ziffer | ( Ausdruck )
      Ziffer   -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  }
s = Ausdruck
```

Der Text von Brandes suggeriert, dass man die Syntaxanalyse für G mit einem Parser erledigen kann, der nach dem Prinzip des rekursiven Abstiegs arbeitet. Das ist falsch, da G die erforderliche Voraussetzung, nämlich eine LL(1)-Grammatik zu sein, nicht erfüllt.

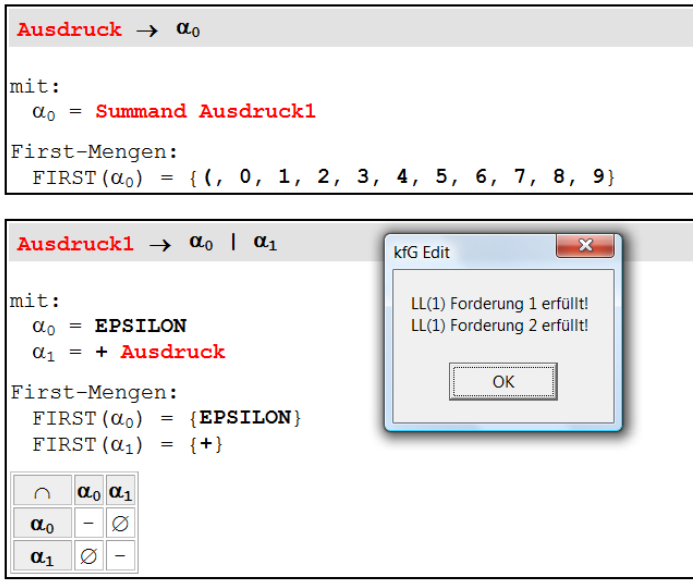


∩	α_0	α_1
α_0	–	{(, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
α_1	{(, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}	–

Dass es mit den statischen Java-Methoden dennoch funktioniert, liegt daran, dass die Implementierung nicht die oben genannte Grammatik umsetzt, sondern die folgende.

```
G = (N, T, P, s)
N = {Ausdruck, Ausdruck1, Summand, Summand1, Faktor, Ziffer}
T = {+, *, (, ), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
P = { Ausdruck  -> Summand Ausdruck1
      Ausdruck1 -> + Ausdruck | EPSILON
      Summand   -> Faktor Summand1
      Summand1  -> * Summand | EPSILON
      Faktor    -> ( Ausdruck ) | Ziffer
      Ziffer    -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
s = Ausdruck
```

Diese Grammatik G (auf Indizierung zur Unterscheidung der Grammatiken wollen wir hier verzichten) erfüllt in der Tat die LL(1)-Forderungen und entsteht aus obiger Grammatik durch die Transformation Linksfaktorisierung („Ausklammern des jeweils längsten gemeinsamen Präfix'"). Die Prüfung der beiden LL(1)-Forderungen erledigt AtoCC für uns:



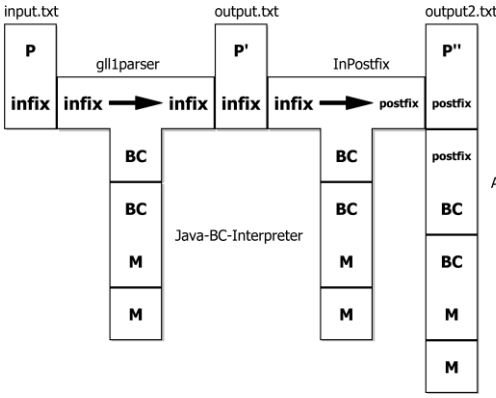
Weder die erste noch die zweite Grammatikdefinition lässt jedoch klar erkennen, welche Ausdrücke damit beschrieben werden. Mag sein, dass die in dieser Form vorgegebenen Produktionen (durch Ableitungsübungen) von den Schülerinnen und Schülern verifiziert und akzeptiert, wohl aber nicht selbstständig entwickelt und verantwortungsbewusst vertreten werden können. Ein wesentlich besserer Vorschlag, der die Ausdrücke strukturadäquat beschreibt, ist folgender.

```
G = (N, T, P, s)
N = {Ausdruck, Summe, Produkt, Zahl, Ziffer}
T = {(, ), +, *, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
P = { Ausdruck -> ( Ausdruck ) | Produkt | Zahl | Summe
      Summe    -> Ausdruck + Ausdruck
      Produkt  -> Ausdruck * Ausdruck
      Zahl     -> Ziffer | Ziffer Zahl
      Ziffer   -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  }
s = Ausdruck
```

Zu beachten ist nebenbei, dass hier mit beliebig langen Zahlwörtern gearbeitet wird, denn das vorgeschlagene Rechnen mit Ziffern (um der Implementierung den unhandlichen Umgang mit trennenden Leerzeichen zu ersparen) dürfte Schülerinnen und Schüler nicht allzu praxisorientiert anmuten.

Tatsächlich liefert dieser Entwurf keine LL(1)-Grammatik und kann folglich nicht zur (automatisierten) Herstellung eines rekursiven-Abstiegs-Parsers herangezogen werden. Da es zur Erreichung der hier angestrebten Lernziele von Bedeutung ist, einen Parser nach dieser bequemen Entwurfsmethode (je Nichtterminal eine Methode) herzustellen, müssen also Grammatiktransformationen, wie oben angewandt, unbedingt thematisiert werden.

Für das weitere Vorgehen verwendet man also (nach gezielter Transformation) die oben als erste angegebene LL(1)-Grammatik und kann nun mittels AtoCC automatisch einen Parser (in der Abb.: gll1parser) generieren. Die Modellierung des Verarbeitungsprozesses, der von Brandes dargelegt wurde, wird mit AtoCC visualisiert und ausgeführt. Hierzu wurden die im Beitrag entwickelten Java-Programme zur Infix-Postfix-Konvertierung (InPostfix) und zur Auswertung des jeweils erzeugten Postfix-Ausdrucks durch sehr wenige Programmzeilen (Lesen aus / Schreiben in Dateien) erweitert und verwendet, d.h. an den entsprechenden Stellen im T-Diagramm einbezogen.




```
1 ;Starting batch file that will execute the diagram.
2 ;=====================================================
3 Starting: java.EXE -cp "." gll1parser input.txt output.txt
4 ...
5
6 Done with: java.EXE -cp "." gll1parser input.txt output.txt
7 ;=====================================================
8 Starting: java -classpath "." InPostfix output.txt output2.txt
9 ...
10
11 Done with: java -classpath "." InPostfix output.txt output2.txt
12 ;=====================================================
13 Starting: java -classpath "." Auswertung output2.txt
14 ...
15
16  127*+ ergibt 15
17
18 Done with: java -classpath "." Auswertung output2.txt
19 ;=====================================================
20 ;Diagram successfuly executed.
21
```

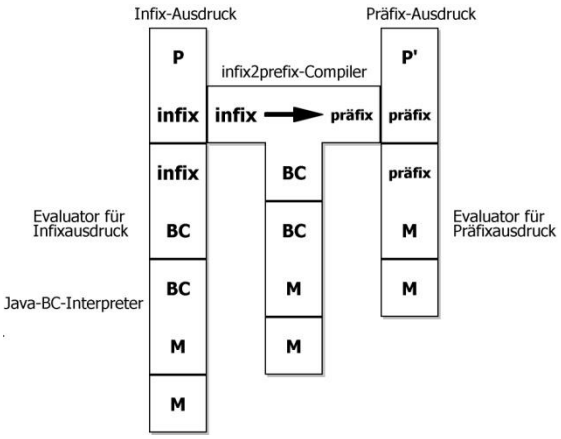
Der im T-Diagramm verwendete infix−>infix-Compiler schreibt den Infix-Ausdruck aus der Eingabedatei unverändert in die Ausgabedatei. Hier ist also nur das Parsing von Bedeutung. Anschließend wird der syntaktisch korrekte Infix-Ausdruck (Inhalt von output.txt) in einen entsprechenden Postfix-Ausdruck übersetzt und vom Java-Programm für die Auswertung von Postfix-Ausdrücken der hier betrachteten Form evaluiert (Interpreter-Baustein).

Auf diese Weise können die Lernziele des von Brandes betrachteten Themas u.E. besser erreicht und irreführende Sichtweisen vermieden werden. Weitere Erläuterungen zur AtoCC-Nutzung und zugehörige Theoriebezüge findet man in /2/.

Leserinnen und Leser, die Interesse an der von Brandes beschriebenen Aufgabenstellung haben, dabei jedoch weniger Java-Integration wünschen, können dies komplett mit AtoCC erledigen. Dabei verwenden wir die folgende Grammatik, die die gewünschten Operator-Präzedenzen bereits berücksichtigt. Außerdem soll das Rechnen mit natürlichen Zahlen zulässig sein.

```
G = (N, T, P, s)
N = {S, Ausdruck, Term, Faktor}
T = {+, *, (, ), Zahl}
P = {S        -> Ausdruck
     Ausdruck -> Term + Ausdruck | Term
     Term     -> Faktor * Term | Faktor
     Faktor   -> ( Ausdruck ) | Zahl }
s = S
```

G ist keine LL(1)-Grammatik. Die Methode des rekursiven Abstiegs kann im gesuchten Parser für Infix-Ausdrücke nicht angewandt werden. AtoCC unterstützt uns jedoch bei der Herstellung eines entsprechenden LR-Parsers für diese Sprache. Dieser Parser kann durch Angabe einiger weniger Attribute zu einem Interpreter, genau: einem Evaluator für Infix-Ausdrücke, ausgebaut werden.




```
;Starting batch file that will execute the diagram.
;=================================================
Starting: java -classpath "." infix input.txt
...
6252

Done with: java -classpath "." infix input.txt
;=================================================
Starting: java.EXE -cp "." infix2prefix input.txt output.txt
...

Done with: java.EXE -cp "." infix2prefix input.txt output.txt
;=================================================
Starting: petite --script output.txt
...
6252
Done with: petite --script output.txt
;=================================================
;Diagram successfuly executed.

```

Wie man der Abb. entnimmt, haben wir exemplarisch einen Infix->Präfix-Compiler entwickelt. Der von diesem Compiler erzeugte Präfix-Ausdruck kann von einem entsprechenden Evaluator interpretiert werden. Dafür ist kein weiteres Programm zu schreiben, da man beispielsweise auf einen vorhandenen Scheme-Interpreter zurückgreifen kann.

Im Beispiel wurde der Ausdruck `((82+4)*4*(3+1+2*7)+5*12)` = 6252 verwendet.

Interessierte Leserinnen und Leser können sich mit Fragen und Anmerkungen gern per Mail c.wagenknecht@hs-zigr.de an den Autor dieses Briefes wenden. Wir unterstützen gern, wenn es um Vorhaben zur Vermittlung von Inhalten aus der TI in der Schule geht.

Literatur:

/1/ Wagenknecht, Chr.; Hielscher, M.: Ein „LP-schweres“ Problem – Theoretische Informatik im Unterricht. In: LOG IN, 28. Jg. (2008), Heft 150/151, S. 69–73.

/2/ Wagenknecht, Chr.; Hielscher, M.: Formale Sprachen, abstrakte Automaten und Compiler Lehr- und Arbeitsbuch für Grundstudium und Fortbildung, Vieweg+Teubner, 2009. XII, 243 S. Mit 95 Abb. Br.

Prof. Dr. Christian Wagenknecht
M.Sc. Michael Hielscher