Compilerbau

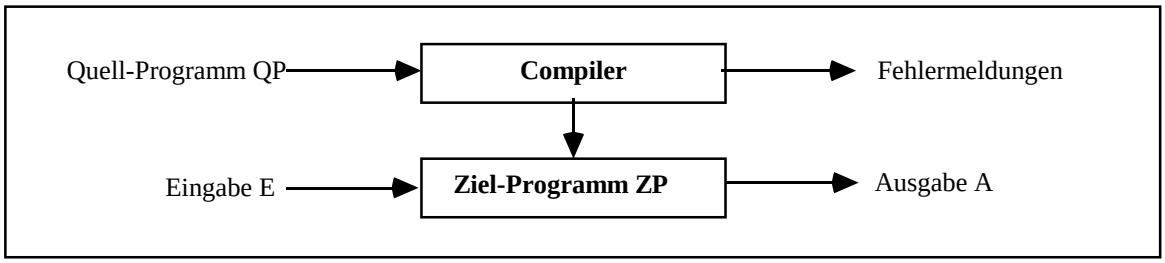
Skript für Lehrveranstaltungen am
Fachbereich Informatik (FB VI)
der Technischen Fachhochschule Berlin
Version 2.6, August 2001
Von
Ulrich Grude,
Christoph Knabe
und Andreas Solymosi

Inhaltsverzeichnis:

1. Einführung

1.1. Was ist ein Compiler?

Ein Compiler ist ein Programm, welches Quell-Programme in Ziel-Programme übersetzt. Die Quell-Programme müssen in der Quell-Sprache des Compilers geschrieben sein. Der Compiler erzeugt die Ziel-Programme in seiner Ziel-Sprache.



Beispiele für Quell-Sprachen von Compilern:
Fortran, Cobol, Basic, C, Algol60, Pascal, Modula, Ada, Eiffel, Lisp, Prolog.

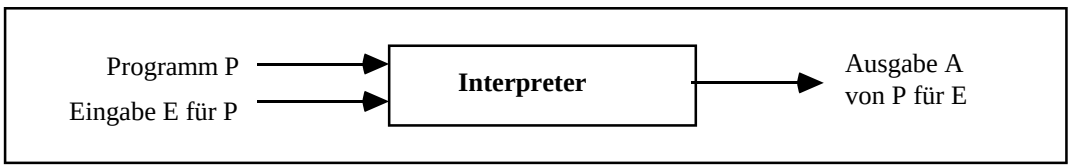
Die Zielsprache eines Compilers ist meistens eine Maschinen-Sprache, oder eine Assembler-Sprache oder eine Assembler-ähnliche Sprache wie z.B. C.

Compiler werden häufig nach ihrer Quell-Sprache und der Ziel-Maschine benannt, z.B. "Pascal-Compiler für die VAX", "Ada-Compiler für den IBM-PC" etc.

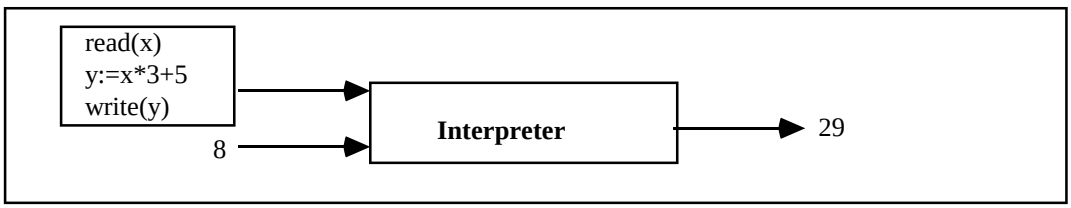
1.2. Was ist ein Interpreter?

Ein Interpreter ist ein Programm, welches Programme ausführt. Dazu gibt man dem Interpreter-Programm I zwei Eingaben:
1. Ein Programm P (welches interpretiert, d.h. ausgeführt, werden soll)
2. Eine Eingabe E für das Programm P (falls P eine Eingabe erwartet).

Der Interpreter I führt das Programm P mit der Eingabe E aus, und liefert als Ergebnis die Ausgabe A (des Programms P für die Eingabe E).



Beispiel:



Wenn man ein Programm P (welches in einer höheren Programmiersprache geschrieben ist) ausführen lassen will, kann man zwischen folgenden "Extremen" wählen:

1: Man lässt P von einem Compiler in ein Maschinenprogramm MP übersetzen und lässt MP von einer "Hardware-Maschine" ausführen.

2: Man lässt P nicht übersetzen sondern gleich von einer "Software-Maschine" ausführen (d.h. man lässt P von einem Interpreter interpretieren).

Zwischen diesen beiden Extremen Vorgehensweisen gibt es viele Zwischenlösungen:

1,5: Man lässt P von einem Compiler "ein bisschen" in ein Programm ZP übersetzen und lässt dann ZP von einer Software-Maschine ausführen.

Die Worte "interpretieren" und "ausführen" bedeuten mehr oder weniger das gleiche.

1.3. Vor- und Nachteile von höheren Sprachen bzw. Maschinen-Sprachen

Höhere Programmiersprachen (wie z.B. Pascal, Modula, Ada etc.) versuchen, die Bedürfnisse, Fähigkeiten und Schwächen von Menschen zu berücksichtigen. Eine Maschinen-Sprache umfaßt die Befehle, die eine Maschine direkt (ohne die Hilfe eines Compilers oder Interpreters) ausführen kann.

1.4. Vorteile von Interpretern gegenüber Compilern

Sofortige Ausführung von Programmen (ohne vorhergehende Compilation), Änderungen an einem Programm können während einer Ausführung des Programms leicht vorgenommen werden, nach einer Änderung des Programms kann die Ausführung fortgesetzt werden, die Ausführung ist leichter "verstehbar und nachvollziehbar" weil es nur das vom Programmierer (in einer höheren Sprache) geschriebene Programm gibt (und kein von einem Compiler erzeugtes, schwer verständliches Maschinen-Programm).

1.5. Vorteile von Compilern gegenüber Interpretern

Die Ausführung eines compilierten Programms ist im allgemeinen effizienter (d.h. braucht weniger Zeit und Speicherplatz) als die Interpretation des entsprechenden Quell-Programms.

Ideal für die Praxis: ein Interpreter für die Programm-Entwicklung und ein Compiler für die "Produktions-Läufe". Beispiel: der sog. Check Out Compiler-und-Interpreter für die Sprache PL1 von der Firma IBM.

Von einem solchen Interpreter/Compiler-Paar wird man verlangen: egal, ob man ein Programm interpretiert oder ob man es erst compiliert und dann ausführt, es soll immer "das gleiche herauskommen". Es ist nicht ganz einfach, solche miteinander verträglichen Interpreter und Compiler zu schreiben.

Ein weiteres Problem, welches (heute leider noch) mit dem Einsatz von Interpretern in der Praxis verbunden ist: Umfangreiche "Programmbibliotheken" liegen meist in Form von schon compilierten Objekten vor (und nicht in Form von Quell-Programmen). Leider sind die meisten Interpreter (noch?) nicht in der Lage, bei der Interpretation eines Programms solche Objekt-Bibliotheken mit zu benutzen.

1.6. Warum sollte man sich mit Compilerbau befassen?

- Weil man häufig Compiler benutzt. Das Fach CB hilft einem, Compiler "besser zu verstehen".
- Weil das Fach CB beispielhaft zeigt, wie man sehr theoretische Erkenntnisse zur Lösung von sehr praktischen Problemen verwenden kann.
- Weil man eine Benutzeroberfläche, eine Eingabe-Sprache oder eine Kontroll-Sprache entwickeln möchte. Dabei sind Kenntnisse in Fach CB nützlich.
- Weil man selbst einen Compiler entwickeln möchte.

1.7. Historisches

- Erste Compiler wurden in den frühen 50-er Jahren für die Sprache Fortran entwickelt, Ende der 50-er Jahre für Cobol.
- Damals war der Bau eines Compilers sehr schwierig und aufwendig, z.B. 15-30 Personen-Jahre (das sind nach Preisen von 1990 ca. 3 bis 6 Millionen DM).
- Anfang der 60-er Jahre starker Einfluss von theoretischen Studien (über Syntax, Chomsky-Grammatiken, Algol60, Backus-Naur-Form etc.) auf den praktischen Compiler-Bau.
- Ab da haben sich die Entwicklung von Programmier-Sprachen und die Entwicklung von Compilern gegenseitig stark beeinflusst.
- Heute ist das Fach Compilerbau eines der am besten entwickelten Gebiete der Informatik, mit umfassenden theoretischen Ergebnissen und sehr brauchbaren praktischen Anwendungen. Gute Werkzeuge für den Compilerbau sind vorhanden. Auf dem Gebiet des Compilerbaus wird weiterhin viel geforscht und entwickelt.

2. Formale Sprachen und Grammatiken

2.1. Formale Sprachen, Definition

Eine formale Sprache ist eine Menge von Symbolketten. Die Elemente einer formalen Sprache (d.h. die Symbolketten) werden häufig als Worte oder als Sätze (der Sprache) bezeichnet. Die Worte Wort und Satz haben also, im Zusammenhang mit formalen Sprachen, die gleiche Bedeutung.

Beispiele:

1. Die Menge {rauf, runter, halt}.
2. Die Menge aller (ganzen, nicht-negativen) Binärzahlen {0, 1, 10, 11, 100, 101, 110, 111 ...}.
3. Die Menge aller geraden Binärzahlen {0, 10, 100, 110, 1000, 1010, 1100, 1110, ...}.

4. Die Menge aller Pascal-Programme.
5. Die Menge aller in Pascal erlaubten Bezeichner (identifier).
6. Die Menge aller in Pascal erlaubten if-Befehle.
7. Die leere Menge {}.

Anmerkung: Natürliche Sprachen (wie z.B. Griechisch, Türkisch, Deutsch, Latein, Englisch etc.) sind sehr viel kompliziertere und geheimnisvollere Gebilde als formale Sprachen. Die Worte und Sätze von natürlichen Sprachen haben Bedeutungen, die kein einzelner Mensch willkürlich verändern kann, sie sind mit den Gefühlen und Gewohnheiten von Menschen verknüpft, aus ihnen bestehen Gedichte und Romane, in natürlichen Sprachen kann man die Wahrheit sagen, Witze erzählen und lügen etc. Eine natürliche Sprache kann man nicht definieren und man kann sie wohl kaum vollständig beschreiben.

2.2. Wie beschreibt man formale Sprachen?

Formale Sprachen, die relativ wenig Worte umfassen, kann man am einfachsten dadurch beschreiben, dass man alle Worte der Sprache in Form einer Liste oder Tabelle angibt. Allerdings umfassen viele interessante formale Sprachen sehr viele oder sogar unendlich viele Worte. Wie kann man solche Sprachen präzise beschreiben???

Beispiel: Wie kann man all die Symbolketten beschreiben, die als if-Anweisungen in Ada-Programmen zulässig sind? Die Beschreibung sollte so genau sein, dass man von jeder konkreten Zeichenkette möglichst einfach und eindeutig feststellen kann, ob sie als if-Befehl zulässig ist oder nicht.

Dieses Problem ist ein Spezialfall des allgemeineren Problems "Wie kann man mit endlichen Mitteln (sehr große und auch) unendlich große Mengen beschreiben?".

Formale Grammatiken dienen dazu, formale Sprachen zu beschreiben. Eine formale Grammatik ist immer endlich groß. Die (durch eine formale Grammatik) beschriebene Sprache kann endlich oder unendlich viele Worte umfassen.

Es gibt verschiedene Arten und Unterarten von formalen Grammatiken. Im folgenden befassen wir uns fast ausschließlich mit sog. Chomsky-Grammatiken. Zwei-Stufen-Grammatiken und attributierte Grammatiken sind andere Arten von Grammatiken.

Anmerkung: Auch zu natürlichen Sprachen (wie Griechisch, Türkisch, Deutsch etc.) gibt es Grammatiken. Eine Grammatik z.B. der deutschen Sprache soll einem dabei helfen, "richtige deutsche Sätze" von falschen Sätzen zu unterscheiden. Die üblichen Schulgrammatiken sind allerdings in sehr starkem Maße unvollständig und ungenau. Unter Linguisten wird die Frage diskutiert, ob es überhaupt im Prinzip möglich ist, für eine natürliche Sprache eine vollständige und präzise Grammatik anzugeben (und in welchem Sinne diese Grammatik vollständig und präzise ist). Ansonsten versucht man (mit einigem Erfolg), Grammatiken für "interessante Teile von natürlichen Sprachen" zu entwickeln.

2.3. Formale Grammatiken, ein einführendes Beispiel

Eine Grammatik G1, die alle (ganzen, vorzeichenlosen) Binärzahlen beschreibt:

R1: Eine Binärzahl ist (entweder) eine 0
R2: (oder) eine Binärzahl ist eine 1
R3: (oder) eine Binärzahl ist eine 0 gefolgt von einer Binärzahl
R4: (oder) eine Binärzahl ist eine 1 gefolgt von einer Binärzahl.

Die gleichee Grammatik G1, in einer unter Fachleuten üblichen Form:

R1:	B	→	0	(sprich z.B. so: "Be geht nach Null")
R2:	B	→	1	("Be geht nach Eins")
R3:	B	→	0B	("Be geht nach Null Be")
R4:	B	→	1B	("Be geht nach Eins Be")

Diese Grammatik G1 besteht nur aus endlich vielen (genauer: aus 4) Regeln. Man kann aber mit dieser Grammatik unendlich viele Symbolketten ableiten, z.B. die Symbolkette 1001.

Beispiel für eine Ableitung (noch ohne Über- und Unterstreichungen):

B
R4:
1B
R3:
10B
R3
100B
R2:
1001

Aufgabe 2.3.1.: Geben Sie mit obiger Grammatik G1 je eine Ableitung an für die drei Worte 101, 001 und 0.

Aufgabe 2.3.2.: Begründen Sie kurz, warum man mit der Grammatik G1 das Wort 1051 (welches keine Binärzahl ist) nicht ableiten kann.

Aufgabe 2.3.3.: Geben Sie eine Grammatik G2 an, mit der man nur solche Binärzahlen ableiten kann, die mit einer 1 beginnen. Andererseits soll man mit G2 natürlich alle solche Binärzahlen ableiten können. Leiten Sie ("als Test") ein paar Worte mit Ihrer Grammatik G2 ab, z.B. die Worte 1, 10, 11, 100 und 111.

Aufgabe 2.3.4.: Geben Sie eine Grammatik G3 für die Sprache aller geraden Binärzahlen an. Leiten Sie mit Ihrer Grammatik G3 ein paar Worte ab, z.B. die Worte 0 und 110.

Aufgabe 2.3.5.: Geben Sie eine Grammatik G4 für alle (vorzeichenlosen) Binärzahlen an. Eine Binärzahl kann (muss aber nicht) einen Binärpunkt "." enthalten (entspricht dem Komma "," in üblichen Dezimalzahlen). Wenn eine Binärzahl einen Binärpunkt "." enthält, dann muss vor und hinter diesem Punkt mindestens je eine Binärziffer stehen (d.h. die Symbolketten .1, .0011, 0. und 101. sollen nicht erlaubt sein). Eine Binärzahl (so sei für diese Aufgabe festgelegt) darf vorn und hinten keine "überflüssigen" Nullen enthalten (d.h. die Symbolketten 011, 0.10 und 000.000 sind nicht erlaubt, statt dessen sollten die Symbolketten 11, 0.1 bzw. 0.0 genommen werden). Zu dieser Sprache der Binärzahlen gehören z.B. die Worte 0, 1, 100100, 1.0, 1010.101, 0.1 und 0.00001!

Leiten Sie aus Ihrer Grammatik G4 mindestens 5 Worte ab (besser: leiten Sie 10 oder 20 Worte ab).

Aufgabe 2.3.6.: Lesen Sie Ihre Grammatik G4 laut vor, so dass Ihr Nachbar die Grammatik nach Ihrem Diktat aufschreiben könnte.

Aufgabe 2.3.7.: Geben Sie eine Grammatik G5 an für die Sprache aller Ausdrücke, die man aus den Variablen-Namen x, y und z, den Operations-Namen + und * und aus runden Klammern "(" und ")" nach den üblichen Regeln zusammensetzen kann. Zu dieser Sprache sollen z.B. die Worte x, y, z, x+y, z*z, x+y+y+z+x, x*y+z*z, (x+y)*z, ((x+x)*(y+y)) gehören. Dagegen sollen z.B. die Worte a+1, (x+y, (),)x+y(, xx+z und y**z nicht zu dieser Sprache gehören. Leiten Sie aus der Grammatik G5 Ihres Nachbarn ein paar Worte ab. Versuchen Sie insbesondere, aus seiner Grammatik ein "unerlaubtes Wort" abzuleiten. Wenn das möglich ist, ist die Grammatik nicht korrekt und sollte verbessert werden.

2.4. Chomsky-Grammatiken, Definition

Eine Chomsky-Grammatik G=(Z, E, S, P) besteht aus

1. einer Menge Z von Zwischen-Symbolen (non terminal symbols)
2. einer Menge E von End-Symbolen (terminal symbols)
3. einem Startsymbol S (einem Element der Menge Z) und
4. einer Menge P von Produktions-Regeln, P={R_1, R_2, ... , R_n}

Dabei muss gelten:

1. Die Mengen Z und E müssen beide nicht-leer und endlich sein und dürfen keine gemeinsamen Elemente besitzen, d.h. es muss $Z \cap E = \{\}$ gelten.

2. Jede Regel R besteht aus
einer linken Seite LS,
einem Trennzeichen "$\rightarrow$" und
einer rechten Seite RS:
R: LS $\rightarrow$ RS
Im allgemeinen sind LS und RS irgendwelche Satzformen (d.h. Folgen von Zwischen- und/oder End-Symbolen), wobei LS mindestens ein Zwischen-Symbol enthalten muss. Bei speziellen Grammatikformen (z.B. bei kontextfreien Grammatiken oder bei linearen Grammatiken, siehe unten) müssen LS und RS bestimmte Einschränkungen erfüllen.

Ende der Definition von Chomsky-Grammatiken.

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

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

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

Achtung: Eine Satzform ist eine beliebige Folge von Zwischen- und/oder End-Symbolen. D.h.: eine Satzform kann nur aus Zwischen-Symbolen bestehen (z.B. ABC), oder sie kann nur aus End-Symbolen bestehen (z.B. abc) oder sie kann aus End-Symbolen und Zwischen-Symbolen bestehen (z.B. aBcD). Eine Satzform kann auch aus null Symbolen bestehen. Diese leere Satzform (oder: leere Symbolkette) bezeichnet man auch mit dem griechischen Buchstaben ε ("epsilon"). Man sollte die leere Symbolkette ε möglichst nicht mit irgend einem Zeichen, z.B. dem Blank-Zeichen (welches auch als Leerzeichen bezeichnet wird), verwechseln. Den Unterschied kann man sich z.B. so veranschaulichen und merken: wenn man die leere Symbolkette ε auf einem Bildschirm ausgibt, dann rückt der Cursor dadurch null Millimeter nach rechts. Gibt man dagegen irgend ein Zeichen aus (z.B. ein Blank-Zeichen), so rückt der Cursor dadurch um etwa 4 Millimeter nach rechts. Wenn man 10 Zeichen (z.B. Blanks) "hintereinander schreibt", dann hat man eine Symbolkette der Länge 10. Wenn man die leere Symbolkette ε 10 mal hintereinander hängt, dann hat man eine Symbolkette der Länge null (weil 10*0=0 ist).

2.5. Spezielle Grammatikformen

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

2.5.1. Typ 3 Grammatiken (lineare, reguläre Grammatiken)

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

Beispiele:

R1: A → bcd
R2: A → a
R3: A → ε

Gegenbeispiele:

R4: A → Bcd -- RS enthält ein Zwischen-Symbol
R5: AB → cd -- LS besteht aus mehr als einem Symbol
R6: Aa → bc -- LS besteht aus mehr als einem Symbol

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

Beispiele:

R1: A → Bcd
R2: A → B
R3: A → Abc

Gegenbeispiele:

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

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

Beispiele:

R1: A → bcD
R2: A → B
R3: A → abC

Gegenbeispiele:

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

Definition: Eine Grammatik ist vom Typ 3 (man sagt auch: sie ist linear, oder: sie ist regulär) wenn

entweder gilt: Alle Regeln sind abschließend oder links-linear.
oder wenn gilt: Alle Regeln sind abschließend oder rechts-linear.

Beispiel für eine Typ 3 Grammatik, G6:

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

Aufgabe 2.5.1.1.: Begründen Sie kurz, warum folgende Grammatik G7 nicht vom Typ 3 ist:

R1: A → aB
R2: B → Ab
R3: A → ab

Aufgabe 2.5.1.2.: Geben Sie eine Typ 3 Grammatik G8 an für die Menge aller in ihrer Lieblings-Programmiersprache erlaubten Bezeichner (identifier).

Aufgabe 2.5.1.3.: Geben Sie eine Typ 3 Grammatik G9 an für die Menge aller Binärzahlen (Brüche und Ganzzahlen, siehe Aufgabe 2.3.5.).

2.5.2. Typ 2 Grammatiken (kontextfreie Grammatiken)

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

Beispiele:

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

Gegenbeispiele:

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

2.5.3. Typ 1 Grammatiken (kontextsensitive Grammatiken)

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

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

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

2.5.4. Typ 0 Grammatiken

Keine Einschränkung der Regeln.

2.6. Ableitung, Definition

Sei eine Grammatik G=(Z, E, S, P) gegeben. Eine Ableitung aus G ist eine Folge von Zeilen, für die gilt:

1. In der ersten Zeile steht nur das Startsymbol S der Grammatik.
2. In der letzten Zeile steht eine Folge von End-Symbolen ("das abgeleitete Wort").
3. In jeder (anderen) Zeile steht eine Satzform (d.h. eine Folge von End- und/oder Zwischen-Symbolen)
4. Zwischen je zwei Zeilen steht die Nummer einer Produktions-Regel R_i aus der Menge P.
5. In jeder Zeile (ausser der letzten) ist ein Teil der Satzform unterstrichen
und in jeder Zeile (ausser der ersten) ist ein Teil der Satzform überstrichen.
6. Die unterstrichene Symbolkette einer Zeile und die überstrichene Symbolkette der nächsten Zeile müssen genau der linken Seite LS und der rechten Seite RS der Regel R_i entsprechen, deren Nummer zwischen den beiden Zeilen steht.
7. Alle nicht unterstrichenen Zeichen einer Zeile müssen unverändert auch in der nächsten Zeile stehen.

Beispiel:

Unsere Grammatik G1:

R1: B → 0
R2: B → 1
R3: B → 0B
R4: B → 1B

Eine Ableitung für das Wort 1001 aus dieser Grammatik:

```
      B          -- In der ersten Zeile steht nur das Startsymbol.
R4:              -- R4: B  →  1B
                 -- Ein Teil dieser Satzform ist überstrichen, ein Teil ist unterstrichen.
R3:              -- R3: B  →  0B
      1          -- Ein Teil dieser Satzform ist überstrichen, ein Teil ist unterstrichen.
R3               -- R3: B  →  0B
      1 0    -- Ein Teil dieser Satzform ist überstrichen, ein Teil ist unterstrichen.
R2:              -- R2: B  →  1
      1 0 0  -- In der letzten Zeile stehen nur Endsymbole.
```

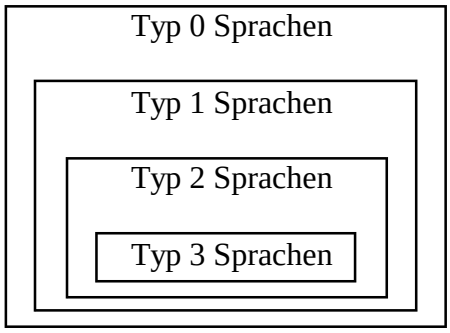
Im folgenden werden wir die Überstreichungen in Ableitungen bisweilen weglassen (wenn sie sich eindeutig aus dem Zusammenhang erschließen lassen). In Klausuren dürfen die Unter- und Überstreichungen nicht weggelassen werden!

2.7. Grammatiken und Sprachen

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

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

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



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

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

2.7.1.: $L1 = \{a^n b^m \mid n \leq 1, m \leq 1\}$

Diese Sprache ist vom Typ 3. Hier eine rechts-lineare Grammatik G10 für L1:

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

2.7.2.: $L2 = \{a^n b^n \mid n \leq 1\}$

Diese Sprache ist vom Typ 2 (aber nicht vom Typ 3!). Hier eine kontextfreie Grammatik G11 für L2:

R1:	S	→	aSb
R2:	S	→	ab

2.7.3.: $L3 = \{w\, w^{-1} \mid$ w ist eine nicht-leere Folge von a´s und b´s$\}$

w^{-1} besteht aus den gleichen Symbolen wie w, aber in umgekehrter Reihenfolge, z.B.

$w = abb$, $w^{-1} = bba$.

Die Sprache L3 ist vom Typ 2 (aber nicht vom Typ 3!). Hier eine kontextfreie Grammatik G12 für L3:

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

2.7.4.: $L4 = \{a^n b^m c^m d^n \mid n \leq 1, m \leq 1\}$

Diese Sprache ist vom Typ 2 (aber nicht vom Typ 3). Hier eine kontextfreie Grammatik G13 für L3:

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

2.7.5.: $L5 = \{a^n b^m c^n d^m \mid n \leq 1, m \leq 1\}$

Diese Sprache ist vom Typ 1 (nicht vom Typ 2!). Hier eine kontextsensitive Grammatik G14 für L5:

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

2.7.6.: $L6 = \{a^n b^n c^n \mid n \leq 1\}$

Diese Sprache ist vom Typ 1 (nicht vom Typ 2!). Hier eine kontextsensitive Grammatik G15 für L6:

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

2.7.7.: L7 = {P | P ist ein haltendes Pascal-Programm}

Diese Sprache ist vom Typ 0 (nicht vom Typ 1!). Es ist möglich, aber ziemlich schwierig, eine Typ 0 Grammatik für L7 anzugeben.

2.7.8.: L8 = {P | P ist ein nicht-haltendes Pascal-Programm}

Diese Sprache kann noch nicht einmal durch eine Typ 0 Grammatik beschrieben werden. Der Beweis dieser Tatsache gehört zu den wichtigsten theoretischen Grundlagen der Informatik.

Aufgabe 2.7.1. Geben Sie für jede der folgenden formalen Sprachen eine Typ 3 Grammatik (reguläre Grammatik) an:

1. Die Sprache aller in Pascal erlaubten Identifier. Ein solcher Identifier muss mindestens 1 Zeichen lang sein, darf nur aus Buchstaben, Ziffern und dem Unterstrich "_" bestehen und muss mit einem Buchstaben beginnen.

2. Die Sprache aller Dezimalbrüche. Zu dieser Sprache sollen z.B. die Worte 0.1, 123.456, 0.1234 gehören. Nicht dazu gehören sollen z.B. die Worte 123. (weil hinter dem Dezimalpunkt keine Ziffern stehen), .123 (weil vor dem Dezimalpunkt keine Ziffern stehen) und 123 (weil hier der Dezimalpunkt fehlt).

Aufgabe 2.7.2. Geben Sie für jede der folgenden formalen Sprachen eine Typ 2 Grammatik (kontextfreie Grammatik) an:

1. Die Sprache aller ungeraden Dezimalzahlen ohne führende Nullen. Zu dieser Sprache sollen z.B. die Worte 3, 17 und 1234567 gehören, nicht aber 1234 (weil diese Zahl gerade ist) oder 011 (weil diese Zahl fürende Nullen besitzt).

2. Die Sprache aller Zeichenketten, die nur aus Nullen ("0") und Einsen ("1") bestehen und zwar aus gleich vielen Nullen wie Einsen. Zu dieser Sprache gehören z.B. die Worte 10, 01, 1001, 1100 und 10110100, nicht dazu gehören z.B. die Worte 011, 101, 1011010. Die leere Zeichenkette soll auch zu dieser Sprache gehören.

Aufgabe 2.7.3. Geben Sie für die folgende formale Sprache eine Typ 1 Grammatik (kontextsensitive Grammatik) an:

$\{a^n b^n c^n \mid n \leq 1\}$

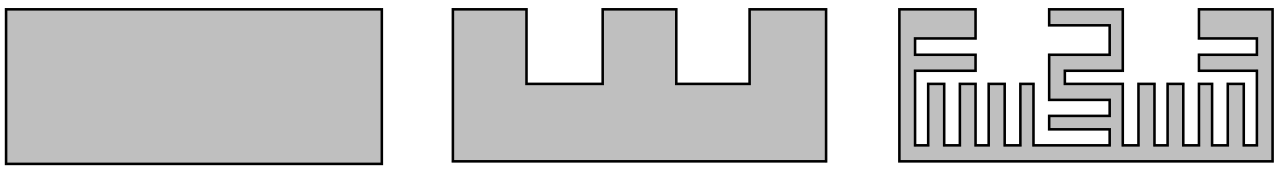
Jedes Wort dieser Sprache beginnt mit einer bestimmten Zahl von ahs. Darauf folgen gleich viele behs und dann gleich viele cehs. Zu dieser Sprache gehören z.B. folgende Worte: abc, aabbcc, aaabbbccc. Nicht dazu gehören z.B. die Worte ab (zu wenig cehs), abcc (zu viele cehs), aabbbccc (zu wenig ahs).

2.8. Zum genaueren Verständnis von Chomsky-Grammatiken

Es liegt nahe, folgendes anzunehmen: je größer eine Sprache ist (d.h. je mehr Worte sie umfaßt), desto schwieriger ist sie zu beschreiben (d.h. desto komplizierter ist ihre Grammatik).

Diese intuitive Annahme ist falsch. Dies soll zuerst anhand eines Vergleichs (mit geometrischen Figuren) und dann an einem Beispiel gezeigt werden.

Geometrische Veranschaulichung:



F1 F2 F3

Von diesen drei geometrischen Figuren hat offensichtlich F1 die größte (schraffierte) Fläche und F3 die kleinste. Andererseits ist F1 wohl am einfachsten zu beschreiben und eine Beschreibung von F3 ist vermutlich komplizierter als eine von F2 oder F1. In diesem geometrischen Beispiel gilt also: die größere Fläche ist leichter zu beschreiben als die kleinere.

Jetzt ein Beispiel mit Grammatiken. Die formale Sprache L9 sei wie folgt definiert:

L9 = {F | F ist eine nicht-leere Folge der Buchstaben a, b, c und d}

Diese Sprache ist besonders leicht zu beschreiben, sie ist vom Typ 3. Hier eine links-lineare Grammatik G14 für L9:

R1:	S	→	Sa	R5:	S	→	a
R2:	S	→	Sb	R6:	S	→	b
R3:	S	→	Sc	R7:	S	→	c
R4:	S	→	Sd	R8:	S	→	d

Jetzt sieht man (hoffentlich) leicht, dass die Sprachen L1 bis L6 Teilmengen der Sprache L9 sind. Trotzdem (oder gerade deswegen) braucht man zur Beschreibung der Sprachen L2 bis L6 "kompliziertere" Grammatik-Formen als zur Beschreibung von L9. Manchmal ist es eben schwierig zu beschreiben, welche Worte nicht zu einer Sprache gehören.

Syntaxanalyse-Problem: darunter versteht man ein Problem ("eine Aufgabe") der folgenden Art:

Gegeben eine Grammatik G und ein Wort w. Gehört das Wort w zu der Sprache, die durch die Grammatik G beschrieben wird (oder: "Kann man w aus G ableiten")?

Da es unendlich viele Grammatiken G und unendlich viele Worte w gibt, gibt es auch unendlich viele solcher Syntaxanalyse-Probleme. Wünschenswert wäre ein Algorithmus, der alle diese Syntaxanalyse-Probleme lösen kann. Dazu gibt es eine schlechte und eine gute Nachricht (von "den Theoretikern"):

Theorem 1: Es ist unmöglich einen Algorithmus zu konstruieren, der für jede Typ 0 Grammatik und jedes Wort w herausfinden kann, ob sich w aus G ableiten lässt.

Theorem 2: Es gibt einen Algorithmus, der (zumindest im Prinzip) für jede Typ 1 Grammatik und jedes Wort w herausfinden kann, ob sich w aus G ableiten lässt.

Dabei heißt "zumindest im Prinzip" soviel wie: "wenn man dem Algorithmus genügend Zeit und Speicherplatz zur Verfügung stellt".

2.9. Formale Sprachen und Grammatiken in der Praxis

Für die Praxis (sehr) wichtig sind Typ-3-Sprachen (reguläre Sprachen) und Typ-2-Grammatiken (kontextfreie Grammatiken). Merkwürdigerweise spielen Typ-3-Grammatiken und Typ-2-Sprachen in der Praxis eine viel geringere Rolle.

Beispiele für Typ-3-Sprachen (reguläre Sprachen) aus der Praxis:

1. Die Menge aller Bezeichner (identifier) einer Programmiersprache
(dazu können gehören: otto, a1, MehrWertSteuer, Tages_Summe etc.)
2. Die Menge aller Ganzzahl-Literale einer Programmiersprache
(dazu können gehören: 0, 123, +45, -67, 1e3, 1e-3, -2e-10 etc.)
3. Die Menge aller Gleitkomma-Literale einer Programmiersprache
(dazu können gehören: 0.0, -123.456, +10.5e4, -10e-4 etc.)
4. Die Menge aller Zeichen-Literale einer Programmiersprache
(dazu können gehören: ´A´, ´a´, ´!´, ´*´, ´´´, ´ ´ etc.)
5. Die Menge aller Symbolketten-Literale einer Programmiersprache
(dazu können gehören: "ABC", "abc", "X!?%&$", "§ § § §", " ", """" etc.)
4. Die Menge aller Schlüsselworte einer Programmiersprache
(dazu können gehören: begin, end, if, fi, do, od, case, esac, function, procedure etc.)

Beispiele für Typ-2-Grammatiken (kontextfreie Grammatiken) aus der Praxis:

So gut wie alle nach 1960 entworfenen Programmiersprachen werden "offiziell" (d.h. in ihrem language reference manual) durch eine Typ-2-Grammatik beschrieben. Zu diesen Sprachen gehören: Algol60, Pascal, Modula, Ada, C, Eiffel und viele andere. Das bedeutet aber nicht, dass diese Sprachen wirklich kontextfrei (vom Typ 2) sind. Diese merkwürdige Tatsache wird im nächsten Abschnitt genauer erläutert.

Typ-3-Sprachen werden häufig nicht durch Chomsky Typ-3-Grammatiken beschrieben, sondern durch gleich starke, aber einfacher zu handhabende Formalismen, z.B. durch so genannte Reguläre Ausdrücke oder durch so genannte Endliche Automaten.

Typ 1 Grammatiken und Typ 0 Grammatiken sind (vereinfacht gesagt) für Menschen und für Maschinen im allgemeinen so schwer zu handhaben, dass sie in der Praxis so gut wie nie angewendet werden. Trotzdem sind sie für "die Theorie" wichtig und können zu einem tieferen Verständnis von Typ 2 Grammatiken beitragen.

2.10. Kontextabhängige Sprachen und kontextfreie Grammatiken

Jedes Pascal-Programm muss eine Reihe von so genannten Kontextbedingungen erfüllen. Und das gilt nicht nur für Pascal, sondern für alle gängigen Programmier-Sprachen wie z.B. Modula, Ada, Eiffel, Cobol, Fortran, C etc.

Beispiele für Kontextbedingungen:

Kontextbedingung 1: Bevor man (in einem Programm) eine Variable anwendet, muss man sie deklarieren.

Kontextbedingung 2: Bevor man (in einem Programm) eine Prozedur anwendet (d.h. aufruft), muss man sie deklarieren.

Entsprechende Kontextbedingungen gibt es auch für benannte Konstanten, Funktionen, Typen, Moduln und andere Größen.

Kontextbedingung 3: Wenn man eine Prozedur mit n formalen Parametern deklariert hat, dann muss man bei jedem Aufruf dieser Prozedur n aktuelle Parameter angeben.

Kontextbedingung 4: Die Datentypen der aktuellen Parameter in einem Prozeduraufruf müssen zu den Datentypen der formalen Parameter in der Prozedur-Deklaration "passen".

Man kann beweisen, dass solche und ähnliche Kontextbedingungen prinzipiell nicht durch eine Typ 2 Grammatik (kontextfreie Grammatik) beschreibbar sind. Da gängige Programmiersprachen (Pascal, Modula, Ada, ...) von ihren Programmen die Einhaltung solcher Kontextbedingungen verlangen sind diese Programmiersprachen keine kontextfreien Sprachen (Typ 2 Sprachen), sondern Typ 1 oder sogar Typ 0 Sprachen.

Typ 1 Grammatiken und Typ 0 Grammatiken sind aber (für Menschen und Maschinen) sehr schwer zu handhaben und somit für die Praxis wenig geeignet. Wie soll man also Pascal, Modula, Ada und alle anderen gängigen Sprachen beschreiben?

Man geht in zwei Schritten vor:

1. Man beschreibt mit einer kontextfreien Grammatik eine "etwas zu große Sprache". Diese enthält alle korrekten Programme aber zusätzlich auch solche, die gewisse Kontextbedingungen nicht erfüllen. Diese "etwas zu große Sprache" ist leichter zu beschreiben als die Sprache, die nur die korrekten Programme umfasst (siehe oben 2.8.).

2. Man beschreibt die Kontextbedingungen z.B. in natürlicher Sprache oder mit einem speziellen Formalismus und "verbietet" alle Programme, die diese Kontextbedingungen nicht erfüllen.

Hier noch die Namen von zwei (miteinander verwandten) Formalismen, mit denen man auch Kontextbedingungen präzise beschreiben kann:

1. Zwei-Stufen-Grammatiken, manchmal auch nach einem ihrer Erfinder van Wijngarden-Grammatiken genannt. Damit wurden z.B. die Sprachen Algol68 und Elan beschrieben.

2. Attributierte Grammatiken. Die Sprache CDL2 (Compiler Definition Language 2) wurde mit solchen Grammatiken beschrieben und erlaubt es (stark vereinfacht gesagt), beliebige andere Sprachen durch eine solche Grammatik zu beschreiben und aus dieser Beschreibung "einen Compiler zu erzeugen".

Anmerkung: Typ 1 Grammatiken heißen kontextsensitiv (oder: kontextabhängig), weil man damit auch Kontextbedingungen beschreiben kann. Hier noch eine zweite, ganz andere Erklärung für die Bezeichnung "kontextsensitiv":

Eine "typische kontextsensitive Regel" kann z.B. so aussehen:

R: aBXcD → aBYzcD

Diese Regel kann man so "lesen": aus X darf man Yz ableiten, aber nur, wenn X im Kontext von aB (links) und cD (rechts) steht. In der folgenden Darstellung der gleichen (kontextsensitiven) Regel ist "ihr kontextfreier Teil" fett hervorgehoben:

R: aBXcD → aBYzcD

2.11. Syntax und Semantik, zwei Auffassungen

Erste Auffassung: Die Begriffe Syntax und Semantik sind sehr komplex und entziehen sich einer kurzen, exakten Definition. Man kann die Gebiete Syntax und Semantik aber durch "typische Fragen" charakterisieren und damit voneinander unterscheiden.

Typische syntaktische Fragen:
1. Gehört die Symbolkette Z zur Sprache L?
2. Ist "abc_def" ein in Pascal-Programmen erlaubter Bezeichner (identifier)?
3. Ist "abc_def" ein in Ada-Programmen erlaubter Bezeichner (identifier)?
4. Ist "procedure a is begin null; end a;" ein Ada-Programm?
5. Welche syntaktische Struktur hat die Symbolkette "if x>1 then y:=2; else y:=3; end if;"?
D.h. welches sind die relevanten Teile dieser Symbolkette?

Typische semantische Fragen:
1. Was bedeutet diese Symbolkette Z?
2. Was macht das Programm "procedure a is begin null; end a;", wenn man es ausführt?
3. Was gibt das Programm Z aus, wenn man ihm die Zahl 17 als Eingabe gibt?

Zweite Auffassung: Die Begriffe Syntax und Semantik sind ganz einfach zu definieren:

Alles, was man mit Typ 2 Grammatiken (kontextfreien Grammatiken) beschreiben kann, gehört zur Syntax einer Sprache bzw. eines Programms. Alles übrige (z.B. Kontextbedingungen) gehört zur Semantik der Sprache bzw. des Programms.

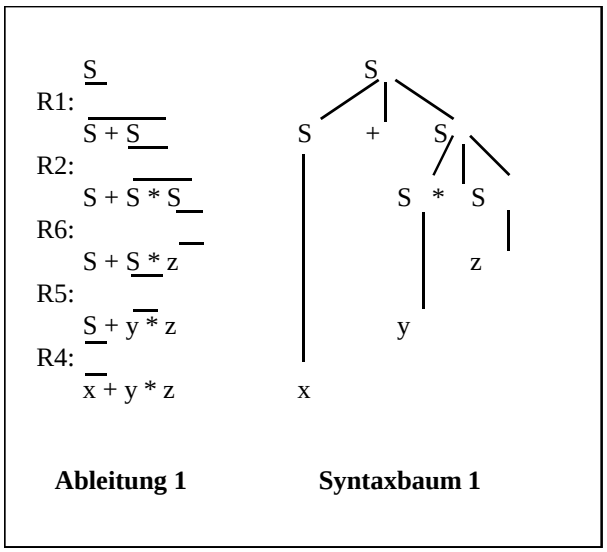
Aufgabe 2.11.1.: Welche dieser beiden Auffassungen ist angemessener? Begründen Sie ihre Antwort.

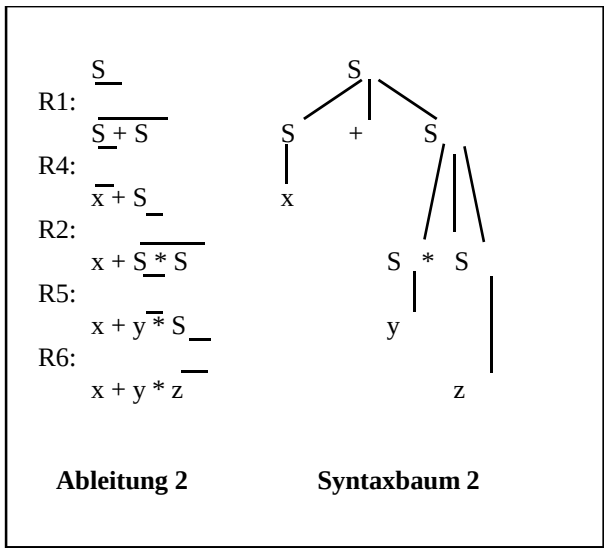
2.12. Eindeutigkeit und Mehrdeutigkeit von Typ 2 Grammatiken

Wir betrachten die Grammatik G14:

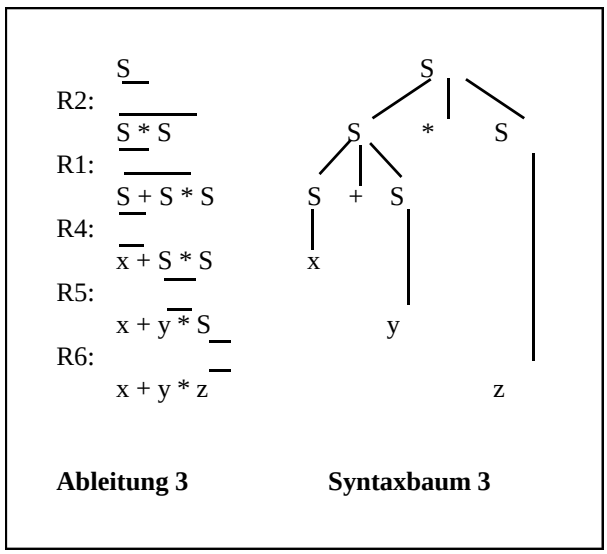
R1:	S	→	S + S	R4:	S	→	x
R2:	S	→	S * S	R5:	S	→	y
R3:	S	→	(S)	R6:	S	→	z

und das (aus G14 ableitbare) Wort x + y * z. Dieses Wort wird jetzt mehrmals abgeleitet, und zu jeder Ableitung wird ein so genannter Syntaxbaum konstruiert:





Die beiden Ableitungen 1 und 2 sind zwei verschiedene Ableitungen. Aber ihre beiden Syntaxbäume sind gleich. Deshalb sagen wir auch: die beiden Ableitungen 1 und 2 sind nur unwesentlich verschieden voneinander (oder: sie sind im wesentlichen gleich).



Der Syntaxbaum 3 unterscheidet sich vom Syntaxbaum 1 bzw. 2. Also sagen wir: die Ableitung 3 unterscheidet sich wesentlich von der Ableitung 1 und von der Ableitung 2.

Definition: Eine kontextfreie Grammatik heißt mehrdeutig, wenn es mindestens ein daraus ableitbares Wort mit zwei verschiedenen Syntaxbäumen (d.h. mit zwei wesentlich verschiedenen Ableitungen) gibt.

Die Grammatik G14 ist mehrdeutig, weil es z.B. für das Wort x + y ∗ z zwei verschiedene Syntaxbäume (d.h. zwei wesentlich verschiedene Ableitungen) gibt.

Syntaktische Struktur:

Der Syntaxbaum 1 (bzw. 2) sagt folgendes über die syntaktische Struktur des Wortes x + y ∗ z :
x + y ∗ z ist eine Summe.
Deren erster Summand ist x und
deren zweiter Summand ist das Produkt y ∗ z.

Der Syntaxbaum 3 sagt folgendes über die syntaktische Struktur des Wortes x + y ∗ z:
x + y ∗ z ist ein Produkt.

Dessen erster Faktor ist die Summe x + y und
deren zweiter Faktor ist z.

Aufgabe 2.12.1: Welcher der beiden Syntaxbäume 1 und 3 ist "besser" oder "richtiger" als der andere? Warum?

Warnung: In aller Regel sind mehrdeutige Grammatiken Quellen von Problemen (z.B. von semantischen Mehrdeutigkeiten). Man sollte sie deshalb möglichst vermeiden.

Allerdings: Als Lösung einer Aufgabe (insbesondere in einer Klausur) ist auch eine mehrdeutige Grammatik zulässig, wenn nicht ausdrücklich eine eindeutige Grammatik gefordert wurde.

Merke: Zwei Ableitungen können gleich, unwesentlich verschieden oder wesentlich verschieden sein. Zwei Syntaxbäume (für das gleiche Wort) können gleich oder verschieden voneinander sein. Es ist (nach unseren Vereinbarungen) nicht sinnvoll, von wesentlich verschiedenen Syntaxbäumen zu sprechen.

Aufgabe 2.12.2: Gegeben zwei verschiedene Ableitungen für das gleiche Wort. Wie stellt man fest, ob die beiden Ableitungen nur unwesentlich verschieden oder wesentlich verschieden sind?

Aufgabe 2.12.3: Zeigen Sie, dass folgende Grammatik G15 mehrdeutig ist:

R1: A → AA
R2: A → a

Aufgabe 2.12.4: Zeigen Sie, dass folgende Grammatik G16 eindeutig ist:

R1: A → aA
R2: A → a

Definition: Eine Ableitung aus einer kontextfreien Grammatik (Typ 2 Grammatik) heißt linkskanonisch, wenn in jeder Zeile (außer der letzten) das jeweils am weitesten links stehende Zwischen-Symbol unterstrichen ist.

Aufgabe 2.12.5.: Definieren Sie, was eine rechtskanonische Ableitung ist.

Aufgabe 2.12.6.: Betrachten Sie obige Ableitungen 1, 2 und 3 des Wortes x + y ∗ z. Welche der Ableitungen sind linkskanonisch, welche rechtskanonisch?

Aufgabe 2.12.7.: Geben Sie eine Ableitung (aus der Grammatik G14) für das Wort x + y ∗ z an, die weder linkskanonisch noch rechtskanonisch ist.

Aufgabe 2.12.8.: Geben Sie eine Grammatik und eine Ableitung daraus an, so dass die Ableitung sowohl linkskanonisch als auch rechtskanonisch ist.

Aufgabe 2.12.9.: Gegeben sei die Grammatik

R1: S → S - S
R2: S → x

Geben Sie alle Syntaxbäume für das Wort "x - x - x" an. Geben Sie für jeden Baum an, welcher "Berechnungsreihenfolge" er entspricht (d.h. welche von den beiden Minus-Operationen in dem Wort "x - x - x" zuerst ausgeführt wird).

Aufgabe 2.12.10.: Geben Sie eine eindeutige Grammatik an, welche die gleiche Sprache beschreibt, wie die (mehrdeutige) Grammatik in der vorigen Aufgabe 2.12.9. Ihre Grammatik soll ausdrücken, dass die Minus-Operationen von links nach rechts ausgeführt werden.

Aufgabe 2.12.11.: Geben Sie eine eindeutige Grammatik an, welche die gleiche Sprache beschreibt, wie die (mehrdeutige) Grammatik in der vorigen Aufgabe 2.12.10. Ihre Grammatik soll ausdrücken, dass die Minus-Operationen von rechts nach links ausgeführt werden.

Theorem: Eine kontextfreie Grammatik ist mehrdeutig, wenn es für mindestens ein Wort zwei verschiedene linkskanonische Ableitungen gibt.

Theorem: Eine kontextfreie Grammatik ist mehrdeutig, wenn es für mindestens ein Wort zwei verschiedene rechtskanonische Ableitungen gibt.

Die Mehrdeutigkeit einer Grammatik kann man also auf drei verschiedene Weisen zeigen: man zeigt, dass man aus der Grammatik ein Wort ableiten kann für das es 1. zwei verschiedene Syntaxbäume oder 2. zwei verschiedene linkskanonische Ableitungen oder 3. zwei verschiedene rechtskanonische Ableitungen gibt.

Die obigen Ableitungen 2 und 3 (für das Wort $x + y * z$) sind verschieden voneinander und beide linkskanonisch. Auch daraus folgt, dass die Grammatik G14 mehrdeutig ist.

Merke: Ableitungen gibt es im Zusammenhang mit allen Typen von Chomsky-Grammatiken (Typ 0 bis Typ 3). Syntaxbäume, linkskanonische und rechtskanonische Ableitungen gibt es nur im Zusammenhang mit Typ-2- und mit Typ-3-Grammatiken (kontextfreien bzw. regulären Grammatiken).

Zusammenfassung: Gegeben eine kontextfreie Grammatik G und ein daraus ableitbares Wort w. Im allgemeinen gilt dann: Es gibt für w mehrere verschiedene Ableitungen. Zu jeder Ableitung gibt es genau einen Syntaxbaum. Verschiedene Ableitungen des gleichen Wortes können verschiedene Syntaxbäume besitzen (dann handelt es sich um wesentlich verschiedene Ableitungen), sie können aber auch gleiche Syntaxbäume besitzen (dann unterscheiden sich die Ableitungen nur unwesentlich).

Aufgabe 2.12.12.: Die Sprache L10 soll alle Ausdrücke umfassen, die man aus den Variablen-Namen x, y und z, den Operations-Zeichen + und * und runden Klammern "(" und ")" zusammensetzen kann. Geben Sie eine kontextfreie Grammatik für L10 an, die

1. eindeutig ist und die
2. die Regel "Punktrechnung geht vor Strichrechnung" widerspiegelt.

2.13. Grammatiken für Ausdrücke

2.13.1. Links- und rechts-assoziative Operatoren

In dem Ausdruck A - B - C steht das B zwischen zwei Minus-Zeichen. Es muss geklärt werden, ob dieses B zum rechten Minus-Zeichen "gehört" oder zum linken Minus-Zeichen. Mit anderen Worten: ist der Ausdruck A - B - C gleichbedeutend mit (A - B) - C (d.h. B gehört zum linken Minus-Zeichen) oder gleichbedeutend mit A - (B - C) (d.h. B gehört zum rechten Minus-Zeichen.

Hier ein konkretes Beispiel für diese Frage: Ist der Ausdruck 16 - 8 - 4 gleichbedeutend mit (16 - 8) - 4 oder gleichbedeutend mit 16 - (8 - 4)? Im ersten Fall hat der Ausdruck den Wert 4, in zweiten Fall den Wert 12.

Üblicherweise legt man fest, dass das B zum linken Minus-Zeichen gehört, d.h. A - B - C ist gleichbedeutend mit (A - B) - C. Dazu sagt man auch: Der Minus-Operator "-" ist links-assoziativ.

Da "-" links-assoziativ ist, kann man einen Ausdruck wie z.B. A - B - C - D - E - F "von links nach rechts" auswerten. Dies entspricht der Klammerung (((((A -B) - C) - D) - E)- F).

Entsprechend einer verbreiteten Konvention sind die Operatoren +, -, * und / alle links-assoziativ. Dagegen ist das Potenzieren ** üblicherweise rechts-assoziativ, d.h. A ** B ** C ist gleichbedeutend mit dem Ausdruck A ** (B ** C), und nicht gleichbedeutend mit (A ** B) ** C.

Konkrete Beispiele:

Es gilt: -400 + 500 + 600 = (-400 + 500) + 600 = 100 + 600 = 700

Man beachte, dass bei dieser Rechnung auch das "Zwischenergebnis" 100 nur dreistellig ist. Wäre der +-Operator rechts-assoziativ dann

würde gelten: -400 + 500 + 600 = -400 + (500 + 600) = -400 + 1100 = 700

Hier ist das Zwischenergebnis 1100 vierstellig. Für einen Rechner, der mit einer festen Stellenzahl rechnet, ist es nicht gleichgültig, ob z.B. der +-Operator links- oder rechts-assoziativ ist.

Es gilt: 600 - 300 - 200 = (600 - 300) - 200 = 300 - 200 = 100
und nicht etwa: 600 - 300 - 200 = 600 - (300 - 200) = 600 - 100 = 500

Es gilt: 0.1 * 100 * 20 = (0.1 * 100) * 20 = 10 * 20 = 200
und nicht: 0.1 * 100 * 20 = 0.1 * (100 * 20) = 0.1 * 2000 = 200

Es gilt: 300 / 10 / 2 = (300 / 10) / 2 = 30 / 2 = 15
und nicht: 300 / 10 / 2 = 300 / (10 / 2) = 300 / 5 = 60

Es gilt: 4 ** 3 ** 2 = 4 ** (3 ** 2) = 4 ** 9 = 262_144

oder 4 ** 3 ** 2 = 4^{3^2} = $4^{(3^2)}$ = 4^9 = 262_144

und nicht: 4 ** 3 ** 2 = (4 ** 3) ** 2 = 64 ** 2 = 4_096

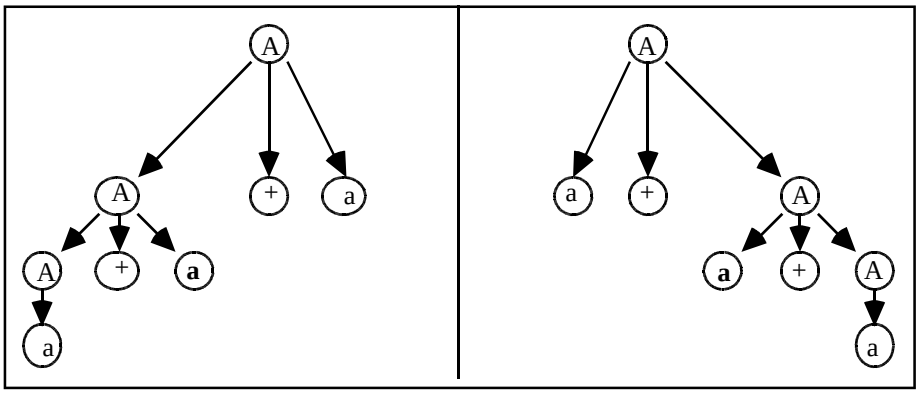
oder: 4 ** 3 ** 2 = 4^{3^2} = $(4^3)^2$ = 64^2 = 4_096

2.13.2. Grammatiken für links- bzw. rechts-assoziative Operatoren

Die formale Sprache {x, x + x, x + x + x, x + x + x + x, ...} aller "Ketten-Additionen" kann u.a. durch die folgenden beiden Grammatiken beschrieben werden:

Grammatik 1:				Grammatik 2:			
R1:	A	→	a	R1:	A	→	a
R2:	A	→	A + a	R2:	A	→	a + A

Beide Grammatiken beschreiben genau die gleiche Sprache, sie tun es aber auf verschiedene Weise. Z.B. ornden sie dem Wort a + a + a die folgenden Syntaxbäume zu:



Der erste Baum entspricht einer Ableitung aus der Grammatik 1. Er ordnet das mittlere a dem linken Plus-Zeichen zu, der zweite Baum entspricht einer Ableitung aus der Grammatik 2. Er ordnet das mittlere a dem rechten Plus-Zeichen zu. Daran kann man erkennen:

Die Grammatik 1 drückt aus, dass das Plus-Zeichen links-assoziativ ist. Die Grammatik 2 drückt aus, dass das Plus-Zeichen rechts-assoziativ ist. Deshalb ist die Grammatik 1 "richtiger" als die Grammatik 2.

Die formale Sprache {x, x ** x, x ** x ** x, x ** x ** x ** x, ...} aller "Ketten-Potenzierungen" wird durch die folgende Grammatik beschrieben:

R1: A → a
R2: A → a ** A

Diese Grammatik drückt aus, dass der Potenz-Operator "**" rechts-assoziativ ist.

2.13.3. Eine einfache Grammatik für Ausdrücke

Im folgenden wollen wir uns mit der formalen Sprache aller Ausdrücke befassen, die man aus Variablen-Namen, einstelligen Operatoren, zweistelligen Operatoren und runden Klammern zusammensetzen kann. Der Einfachheit halber erlauben wir hier nur einen einzigen Variablen-Namen "x". Man könnte dieses Endsymbol x aber auch durch ein Zwischen-Symbol X ersetzen und weitere Regeln angeben, mit denen man aus X alle gewünschten Variablen-Namen ableiten kann.

Variablen-Namen:	x
einstellige Operatoren	+, -
zweistellige Operatoren	+, -, *, /, **
Klammern:	(,)

Hier eine einfache Grammatik für die Sprache aller Ausdrücke, die man aus diesen "Rohmaterialien" zusammensetzen kann. "AUS" ist das einzige Zwischensymbol dieser Grammatik und damit gleichzeitig auch Startsymbol. Alle anderen Zeichen sind Endsymbole.

R1: AUS → + AUS
R2: AUS → - AUS
R3: AUS → AUS + AUS
R4: AUS → AUS - AUS
R5: AUS → AUS * AUS
R6: AUS → AUS / AUS
R7: AUS → AUS ** AUS
R8: AUS → (AUS)
R9: AUS → x

Diese Grammatik ist sehr einfach, sie "leistet aber auch nicht viel". Insbesondere ist die Grammatik in hohem Maße mehrdeutig und legt von keinem der zweistelligen Operatoren +, -, *, / und ** fest, ob er links-assoziativ oder rechtsassoziativ ist.

Aufgabe 2.13.3.1.: Konstruieren Sie mit dieser Grammatik zwei Syntax-Bäume für das Wort "x - x - x".

2.13.4. Eine bessere Grammatik für Ausdrücke

Eine "gute" Grammatik für Ausdrücke sollte die relative Bindungsstärke von Operatoren beschreiben (z.B. Punktrechnung geht vor Strichrechnung"). Außerdem sollte sie für jeden zweistelligen Operator festlegen, ob er links-assoziativ oder rechts-assoziativ ist. Die folgende Beispiel-Grammatik drückt aus:

1. Klammern binden stärker als alle Operatoren.
2. Die einstelligen Operatoren + und - binden stärker als alle anderen Operatoren.
3. Der Potenz-Operator ** bindet stärker als die zweistelligen Operatoren +, -, * und /.
4. Die zweistelligen Operatoren * und / binden stärker als + und - ("Punktrechnung geht vor Strichrechnung).
5. Die zweistelligen Operatoren +, -, * und / sind links-assoziativ.
6. Der (zweistellige) Operator ** ist rechts-assoziativ.

In dieser Grammatik kommen die fünf Zwischen-Symbole AUS1, AUS2, AUS3, AUS4 und AUS5 vor. Alle anderen Symbole sind End-Symbole. AUS1 ist das Startsymbol.

R1: AUS1 → AUS1 + AUS2 -- links-assoziativ, Bindungsstärke 1
R2: AUS1 → AUS1 - AUS2 -- links-assoziativ, Bindungsstärke 1
R3: AUS1 → AUS2

R4: AUS2 → AUS2 * AUS3 -- links-assoziativ, Bindungsstärke 2
R5: AUS2 → AUS2 / AUS3 -- links-assoziativ, Bindungsstärke 2
R6: AUS2 → AUS3

R7: AUS3 → AUS4 ** AUS3 -- rechts-assoziativ, Bindungsstärke 3
R8: AUS3 → AUS4

R9: AUS4 → + AUS5 -- Bindungsstärke 4
R10 AUS4 → - AUS5 -- Bindungsstärke 4
R11 AUS4 → AUS5

R12 AUS5 → (AUS1) -- Bindungsstärke 5
R13 AUS5 → x

Aufgabe 2.13.4.1.: Leiten Sie aus dieser Grammatik die Worte x - x - x, x ** x ** x, x + x * x, x ** x - x und x ** (x - x) ab und zeichnen Sie die entsprechenden Syntaxbäume.

Aufgabe 2.13.4.2.: Kann man aus dieser Grammatik die Worte (- x + - x) und (+ - x + x) ableiten?

Aufgabe 2.13.4.3.: Verbessern Sie die obige Grammatik so, dass in keinem ableitbaren Ausdruck zwei Operatoren direkt aufeinander folgen können. D.h. Worte wie x * - x oder x ++ x etc. sollen "verboten" werden.

2.13.5. Ausdrücke in verschiedenen Programmier-Sprachen

Im folgenden sind Teile der kontextfreien Grammatiken von Algol 60, Pascal, Ada und C wiedergegeben. Es handelt sich jeweils um den Teil der Grammatik, der die zulässigen Ausdrücke beschreibt. Die Regeln sind hier weitgehend in ihrer originalen Form belassen worden. Nur der Zeilenumbruch wurde teilweise verändert und die Regeln wurden numeriert (R1, R2, ...). Eine kurze Anmerkung vor den Regeln erläutert ihre Form, die Bedeutung bestimmter Sonderzeichen und wie Zwischen-Symbole und End-Symbole unterschieden werden. In einigen wenigen Fällen wurden sehr lange Be-

zeichnungen (z.B. simple arithmetic expression) durch Abkürzungen ersetzt (z.B. simple arith. expr.). In den Regeln von Algol 60 werden hier einige Sonderzeichen nicht ganz originalgetreu wiedergegeben (z.B. die Zeichen "≤" und "≥", die im Original mit zwei waagerechten Strichen dargestellt werden, statt nur mit einem. Aber mein Textverarbeitungs-System ...).

Die Regeln stammen aus folgenden Quellen:

"Algol-Manual der ALCOR-Gruppe", von R. Baumann, Oldenbourg Verlag 1965
"Pascal, User Manual and Report, von K. Jensen und N. Wirth, 3rd Edition, Springer-Verlag 1974
"Programmiersprache Ada", DIN-Norm 66 268, Beuth Verlag Berlin 1988
"The C Programming Language", von B. W. Kernighan und D. M. Ritchie, Prentice Hall 1988

2.13.5.1. Ausdrücke in Algol 60

Anmerkung zur Notation der Regeln: Zwischen-Symbole stehen in spitzen Klammern, z.B. <adding operator> oder <arithmetic expression>. Das Zeichen "::=" trennt die linke Seite einer Regel von der rechten Seite. Der senkrechte Strich "|" trennt mehrere rechte Seiten, die zur gleichen linken Seite gehören. End-Symbole, die nur aus Buchstaben bestehen, sind fett gedruckt, z.B. **if**, **then**, **else**. Andere End-Symbole sind nicht-fett gedruckt, z.B. +, -, ×, /, ÷, ∧, ≡, ¬, ⊃ .

Hier die Regeln für arithmetische Ausdrücke und für Boolesche Ausdrücke aus der Grammtik von Algol 60:

Arithmetische Ausdrücke:

R1-2:	<adding operator>	::=	+ | -
R3-5:	<multyplying operator>	::=	× | / | ÷
R6:	<primary>	::=	<unsigned number> |
R7:			<variable> |
R8:			<function designator> |
R9:			(<arithmetic expression>)
R10:	<factor>	::=	<primary> |
R11:			<factor> ↑ <primary>
R12:	<term>	::=	<factor> |
R13:			<term> <multiplying operator> <factor>
R14:	<simple arith. expr.>	::=	<term> |
R15:			<adding operator> <term> |
R16:			<simple arith. expr.> <adding operator> <term>
R17:	<if clause>	::=	**if** <Boolean expression> **then**
R18:	<arithmetic expression>	::=	<simple arith. expr.> |
R19:			<if clause> <simple arith. expr.> **else** <arithmetic expression>

Boolesche Ausdrücke:

R20-25:	<relational operator>	::=	< | ≤ | = | ≥ | > | ≠
R26:	<relation>	::=	<simple arith. expr.> <relational operator> <simple arith. expr.>
R27:	<Boolean primary>	::=	<logical value> |
R28:			<variable> |
R29:			<function designator> |
R30:			<relation> |
R31:			(<Boolean expression>)
R32:	<Boolean secondary>	::=	<Boolean primary> |
R33:			¬ <Boolean primary>
R34:	<Boolean factor>	::=	<Boolean secondary> |
R35:			<Boolean factor> ∧ <Boolean secondary>
R36:	<Boolean term>	::=	<Boolean factor> |
R37:			<Boolean term> ∨ <Boolean factor>
R38:	<implication>	::=	<Boolean term> |
R39:			<implication> ⊃ <Boolean term>
R40:	<simple Boolean>	::=	<implication> |
R41:			<simple Boolean> ≡ <implication>
R42:	<Boolean expression>	::=	<simple Boolean> |
R43:			<if clause> <simple Boolean> **else** <Boolean expression>

Anmerkung: Der Operator "÷" bezeichnet die Ganzzahl-Division (z.B. 7 ÷ 2 = 3), der Operator "/" bezeichnet die "normale" Division (z.B. 7.0 / 2.0 = 3.5). "↑" ist der Potenz-Operator. Der Ausdruck if 4>3 then 2 else 1 liefert den Wert 2. Der Ausdruck if 4>5 then 2 else 1 liefert den Wert 1. Damit kann man Zuweisungs-Anweisungen wie z.B.

X := if Y<Z then Z else Y

notieren.

Aufgabe 2.13.5.1.1.: Sagt diese Grammatik etwas über die relative Bindungsstärke der einzelnen Operatoren +, -, × , / , ÷, ↑ , ∧, ∨, ≡ und ⊃ aus? Wenn ja, was sagt sie?

Aufgabe 2.13.5.1.2.: Sagt diese Grammatik etwas über die Assoziativität der Operatoren +, -, × , / , ÷, ↑ , ∧, ∨, ≡ und ⊃ aus? Wenn ja, was sagt sie?

Aufgabe 2.13.5.1.3.: Ist diese Grammatik Ihrer Ansicht nach "offensichtlich mehrdeutig" oder "vermutlich eindeutig"?

2.13.5.2. Ausdrücke in Pascal

Anmerkung zur Notation der Regeln: Das Gleichheitszeichen "=" trennt die linke Seite einer Regel von der rechten Seite. Der senkrechte Strich "|" trennt mehrere rechte Seiten, die zur gleichen linken Seite gehören. End-Symbole sind in Anführungszeichen eingeschlossen, z.B. "(" oder "+" oder "and" etc.. Jede Regel wird durch einen Punkt "." abgeschlossen. Was in eckigen Klammern steht, "darf man nullmal oder einmal nehmen". D.h. die Regel

A = B [C] D

ist eine Abkürzung für die beiden Regeln

A = B D
A = B C D

Was in geschweiften Klammern steht, "darf man beliebig oft, einschließlich nullmal, nehmen". D.h. die Regel

A = B {C} D

ist eine Abkürzung für die drei Regeln

A = B X D
X = C X
X = ε

Hier die Regeln für Ausdrücke aus der Grammatik von Pascal:

R1:	Expression	=	SimpleExpression [RelationalOperator SimpleExpression].
R2:	SimpleExpression	=	[Sign] Term {AddingOperator Term}.
R3:	Term	=	Factor {MultiplyingOperator Factor}.
R4:	Factor	=	UnsignedConstant |
R5:			BoundIdentifier |
R6:			Variable |
R7:			SetConstructor |
R8:			FunctionDesignator |
R9:			"not" Factor |
R10:			"(" Expression ")".
R11:	UnsignedConstant	=	UnsignedNumber |
R12:			CharacterString |
R13:			ConstantIdentifier |
R14:			"nil"
R15:	SetConstructor	=	"[" [ElementDescription {"," ElementDescripton }] "]".
R16:	ElementDescription	=	OrdinalExpression [".." OrdinalExpression].
R17:	FunctionDesignator	=	FunctionIdentifier [ActualParameterList].
R18-24:	RelationalOperator	=	"=" | "<>" | "<" | "<=" | ">" | ">=" | "in" .
R25-27:	AddingOperator	=	"+" | "-" | "or" .
R28-32:	MultiplyingOperator	=	"*" | "/" | "div" | "mod" | "and" .

Aufgabe 2.13.5.2.1.: Wieviele ElementDescription muss ein SetConstuructor mindestens enthalten? Aus wie viel OrdinalExpression besteht eine ElementDescription mindestens bzw. höchstens?

Aufgabe 2.13.5.2.2.: Sagt diese Grammatik etwas über die relative Bindungsstärke der einzelnen Operatoren "=", "<>", "<", "<=", ">", ">=", "in" "+", "-", "or" "*", "/", "div", "mod" und "and" aus? Wenn ja, was sagt sie?

Aufgabe 2.13.5.2.3.: Sagt diese Grammatik etwas über die Assoziativität der Operatoren "=", "<>", "<", "<=", ">", ">=", "in" "+", "-", "or" "*", "/", "div", "mod" und "and" ? Wenn ja, was sagt sie?

Aufgabe 2.13.5.2.4.: Ist diese Grammatik Ihrer Ansicht nach "offensichtlich mehrdeutig" oder "vermutlich eindeutig"?

2.13.5.3. Ausdrücke in Ada

Anmerkung zur Notation der Regeln: Das Zeichen "::=" trennt die linke Seite einer Regel von der rechten Seite. Der senkrechte Strich "|" trennt mehrere rechte Seiten, die zur gleichen linken Seite gehören. Die eckigen und die geschweiften Klammern haben die gleiche Bedeutung wie in der Grammatik von Pascal (eckige Klammern: nimm nullmal oder einmal, geschweifte Klammern: nimm beliebig oft, auch nullmal ist erlaubt). Alle Sonderzeichen wie z.B. "**" und alle fett gedruckten Zeichenfolgen wie z.B. **and** oder **not** oder **abs** sind End-Symbole.

R1:	Ausdruck	::=	Relation { **and** Relation } |
R2:			Relation { **or** Relation } |
R3:			Relation { **xor** Relation } |
R4:			Relation { **and then** Relation } |
R5:			Relation { **or else** Relation }
R6:	Relation	::=	einfacher_Ausdruck [Relationsoperator einfacher_Ausdruck] |
R7:			einfacher_Ausdruck [**not**] **in** Bereich |
R8:			einfacher_Ausdruck [**not**] **in** Typkennung
R9:	einfacher_Ausdruck	::=	[einstell_Additionsoperator] Term {zweistell_Additionsoperator Term}
R10:	Term	::=	Faktor {Multiplikationsoperator Faktor}
R11:	Faktor	::=	Primärausdruck [** Primärausdruck] |
R12:			**abs** Primärausdruck |
R13:			**not** Primärausdruck
R14:	Primärausdruck	::=	numerisches_Literal |
R15:			**null** |
R16:			Aggregat |
R17:			Zeichenkettenliteral |
R18:			Name |
R19:			Allokator |
R20:			Funktionsaufruf |
R21:			Typkonvertierung |
R22:			qualifizierter_Ausdruck |
R23:			(Ausdruck)
R24-29:	Relationsoperator	::=	= | /= | < | <= | > | >=
R30-32:	zweistell._Additionsoperator	::=	+ | - | &
R33-34:	einstell_Additionsoperator	::=	+ | -
R35-38:	Multiplikationsoperator	::=	* | / | **mod** | **rem**

Anmerkung: Mit dem Operator "&" kann man Zeichenketten und einzelne Zeichen miteinander verbinden (konkatenieren). Z.B. ist "Hal" & "lo!" gleich "Hallo!" und ´H´ & "allo!" ist auch gleich "Hallo!". Die Funktionen mod und rem liefern beide den Rest der Ganzzahl-Division, z.B. ist 10 **mod** 4 gleich 2. Die Funktionen **mod** und **rem** liefern das gleiche Ergebnis, solange ihre beiden Parameter gleiches Vorzeichen haben. Für Parameter mit verschiedenen Vorzeichen weichen ihre Ergebnisse voneinander ab: 14 **mod** -5 ist gleich -1 (weil 14 / -5 gleich -3 Rest -1 ist). Dagegen ist 14 **rem** -5 gleich 4 (weil 14 / -5 gleich -2 Rest 4 ist).

Aufgabe 2.13.5.3.1.: Sagt diese Grammatik etwas über die relative Bindungsstärke der einzelnen Operatoren "=", "/=", "<", "<=", ">", ">=", "in" "+", "-", "or" "*", "/", "div", "mod" und "and" aus? Wenn ja, was sagt sie?

Aufgabe 2.13.5.3.2.: Sagt diese Grammatik etwas über die Assoziativität der Operatoren "=", "/=", "<", "<=", ">", ">=", "in" "+", "-", "or" "*", "/", "div", "mod" und "and" ? Wenn ja, was sagt sie?

Aufgabe 2.13.5.3.3.: Ist diese Grammatik Ihrer Ansicht nach "offensichtlich mehrdeutig" oder "vermutlich eindeutig"?

2.13.5.4. Ausdrücke in C

Anmerkung zur Notation der Regeln: Die linke und rechte Seite einer Regel werden hier durch einen Doppelpunkt ":" voneinander getrennt. Gleichzeitig ist der Doppelpunkt ":" auch ein End-Symbol. Der Leser kann (hoffentlich) am jeweiligen Kontext erkennen, was mit einem Doppelpunkt gemeint ist. Alle anderen Sonderzeichen, z.B. "=", "*=", "^", "[" etc. und alle Zeichen, die in Schreibmaschinen-Schrift (Courier) geschrieben sind, z.B. `sizeof`, sind End-Symbole.

Auch der senkrechte Strich "|" ist hier ein End-Symbol. Buchstabenfolgen in "normaler" Schrift (Times) sind Zwischen-Symbole, z.B. expression, inclusive-OR-expression etc. Verschiedene rechte Seiten mit gleicher linken Seite werden nicht durch senkrechte Striche getrennt sondern beginnen jeweils auf einer neuen Zeile. Wenn man eine von mehreren kurzen Zeichenfolgen wählen kann, dann beginnen diese nicht auf neuen Zeilen sondern werden durch "one of" eingeleitet. Wenn "hinten unten" an einem Zwischen-Symbol "opt" dran steht (z.B. argument-expression-list$_{opt}$.), dann ist das Zwischen-Symbol optional, d.h., man kann es nullmal oder einmal nehmen.

Hier die Regeln für Ausdrücke in C:

R1:	expression	:	assignment-expression
R2:		:	expression , assignment-expression
R3:	assignment-expression	:	conditional-expression
R4:		:	unary-expression assignment-operator assignment-expression
R5-15:	assignment-operator	:	one of = *= /= %= += -= <<= >>= &= ^= |=
R16:	conditional-expression	:	logical-OR-expression
R17:			logical-OR-expression ? expression : conditional-expression
R18:	constant-expression	:	conditional-expression
R19:	logical-OR-expression	:	logical-AND-expression
R20:		:	logical-OR-expression | | logical-AND-expression
R21:	logical-AND-expression	:	inclusive-OR-expression
R22:			logical-AND-expression && inclusive-OR-expression
R23:	inclusive-OR-expression	:	exclusive-OR-expression
R24:			inclusive-OR-expression | exclusive-OR-expression
R25:	exclusive-OR-expression	:	AND-expression
R26:			exclusive-OR-expression ^ AND-expression
R27:	AND-expression	:	equality-expression
R28:			AND-expression & equality-expression
R29:	equality-expression	:	relational-expression
R30:			equality-expression == relational-expression
R31:			equality-expression != relational-expression
R32:	relational-expression	:	shift-expression
R33:			relational-expression < shift-expression
R34:			relational-expression > shift-expression
R35:			relational-expression <= shift-expression
R36:			relational-expression >= shift-expression
R37:	shift-expression	:	additive-expression
R38:			shift-expression << additive-expression
R39:			shift-expression >> additive-expression
R40:	additive-expression	:	multiplicative-expression
R41:			additive-expression + multiplicative-expression
R42:			additive-expression - multiplicative-expression
R43:	multiplicative-expression	:	cast-expression
R44:			multiplicative-expression * cast-expression
R45:			multiplicative-expression / cast-expression
R46:			multiplicative-expression % cast-expression
R47:	cast-expression	:	unary-expression
R48:			(type-name) cast-expression
R49:	unary-expression	:	postfix-expression
R50:			++ unary-expression
R51:			-- unary-expression
R52:			unary-operator cast-expression
R53:			`sizeof` unary-expression
R54:			`sizeof` (type-name)
R55-60:	unary-operator	:	one of & * + - ~ !
R61:	postfix-expression	:	primary-expression
R62:			postfix-expression [expression]
R63:			postfix-expression (argument-expression-list$_{opt}$)
R64:			postfix-expression . identifier
R65:			postfix-expression -> identifier
R66:			postfix-expression ++
R67:			postfix-expression --
R68:	primary-expression	:	identifier
R69:			constant
R70:			string

R71:			(expression)
R72:	argument-expression-list	:	assignment-expression
R73:			argument-expression-list , assignment-expression
R74:	constant	:	integer-constant
R75:			character-constant
R76:			floating-constant
R77:			enumeration-constant

Aufgabe 2.13.5.4.1.: Sagt diese Grammatik etwas über die relative Bindungsstärke der einzelnen Operatoren "=", "*=", "/=" "%=", "+=", "-=", "<<=", ">>=", "&=", "^=", "|=", "| |", "&&", "|", "^", "&", "==", "!=", "<", ">", "<=", ">=", "<<", ">>", "+", "-", "*", "%", "++", "--", "!", "." und "->" aus? Wenn ja, was sagt sie?

Aufgabe 2.13.5.4.2.: Sagt diese Grammatik etwas über die Assoziativität der Operatoren aus ? Wenn ja, was sagt sie?

Aufgabe 2.13.5.4.3.: Ist diese Grammatik Ihrer Ansicht nach "offensichtlich mehrdeutig" oder "vermutlich eindeutig"?

2.14. Vorkommnisse und Instanzen von Zwischen-Symbolen

Zwischen-Symbole kommen in den Regeln einer Grammatik, in Ableitungen und in Syntaxbäumen vor. Häufig steht das gleiche Zwischen-Symbol mehrmals in einer Regel, in einer Ableitung bzw. in einem Syntaxbaum. Als Beispiel betrachten wir die Grammatik G17:

R1: A → BAA
R2: A → a
R3: B → b

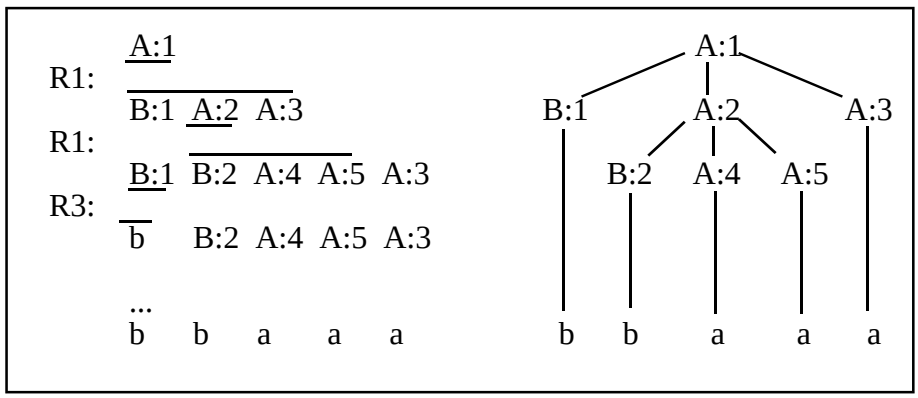
und die (hier nur teilweise wiedergegebene) Ableitung ABL für das Wort bbaaa:

A
R1:
B A A
R1:
B B A A A
R3:
b B A A A
...
b b a a a

In der Regel R1 der Grammatik G17 gibt es drei Vorkommnisse des Zwischen-Symbols A und ein Vorkommnis von B. Wenn wir die einzelnen Vorkommnisse von Zwischen-Symbolen in den Regeln einer Grammatik unterscheiden wollen (bzw. müssen), dann notieren wir die Grammatik wie folgt:

R1: A.0 → B.1 A.1 A.2
R2: A.0 → a
R3: B.0 → b

In der Ableitung ABL gibt es 5 Instanzen des Zwischen-Symbols A und 2 Instanzen von B. Wenn wir die einzelnen Instanzen von Zwischen-Symbolen unterscheiden wollen (bzw. müssen), dann notieren wir die Ableitung bzw. den Syntaxbaum wie folgt:



Merke: Vorkommnisse von Zwischen-Symbolen "leben" in den Regeln einer Grammatik. Sie werden innerhalb von jeder Regel neu "durchnummeriert" und in diesem Skript mit einem einfachen Punkt notiert, z.B. A.0, B.1, A.2 etc. In-

stanzen von Zwischen-Symbolen "leben" in Ableitungen und in Syntaxbäumen. Sie werden innerhalb der gesamten Ableitung (bzw. innerhalb des gesamten Syntaxbaumes) durchnummeriert und in diesem Skript mit einem Doppelpunkt notiert, z.B. A:1, B:2, A:3 etc.

3. Übersetzungsschemen ("theoretische Compiler")

Grammatiken beschreiben formale Sprachen, d.h. sie legen fest, welche Symbolketten zu einer Sprache gehören und welche nicht. Z.B. kann man anhand der Grammatik von Pascal feststellen, ob eine bestimmte Symbolkette ein Pascal-Programm ist oder nicht.

Grammatiken leisten aber noch mehr: hat man ein Wort der beschriebenen Sprache, dann kann man mit Hilfe der Grammatik die syntaktische Struktur (den Syntaxbaum) dieses Wortes ermitteln ("Ist das Wort x+y∗z eine Addition mit einer Multiplikation als Summand, oder ist dieses Wort eine Multiplikation mit einer Addition als Faktor?").

Die syntaktische Struktur eines Wortes gibt an, welches die relevanten Teile dieses Wortes sind. Die drei relevanten Teile des Wortes x+y∗z sind x, + und y∗z (und nicht etwa x+y, ∗ und z oder etwa x+, y und ∗z).

Wenn ein Compiler ein Wort einer Quell-Sprache (z.B. ein Pascal-Programm) in ein entsprechendes Wort einer Ziel-Sprache (z.B. in ein Maschinen-Programm) übersetzen soll, dann ermittelt er erstmal die syntaktische Struktur des zu übersetzenden "Quell-Wortes". Wenn er dann herausgefunden hat, welches die relevanten Teile dieses Wortes sind (und aus welchen relevanten Teile diese Teile bestehen etc.), dann kann er beim Übersetzen folgendermaßen vorgehen:

1. Er übersetzt die relevanten Teile des Wortes und
2. fügt diese "Teile-Übersetzungen" zur Übersetzung des gesamten Wortes zusammen.

Dieses Grundprinzip ist sehr allgemein und genial einfach. Alle Compiler beruhen mehr oder weniger direkt und erkennbar auf diesem Prinzip.

Um die Übersetzung einer Quell-Sprache in eine Ziel-Sprache zu beschreiben, kann man folgendermaßen vorgehen:

1. Man nimmt die kontextfreie Grammatik der Quell-Sprache und ordnet den Zwischen-Symbolen dieser Grammatik so genannte Attribute zu. Z.B. ordnet man dem Zwischen-Symbol A das Attribut A.ü ("ü" wie "Übersetzung" zu). Genauer: man ordnet jedem Vorkommnis von A.0, A.1, A.2 etc. ein Attribut A.0.ü, A.1.ü, A.2.ü etc. zu.

2. Man ergänzt die Produktions-Regeln der Grammatik (die syntaktischen Regeln) durch so genannte semantische Regeln. Die semantischen Regeln beschreiben, wie die Werte der Attribute zu berechnen sind.

<u>Ein ganz einfaches Beispiel:</u>

	syntaktische Regel:		semantische Regel:
R:	A.0 →	B.1 C.1	A.0.ü := B.1.ü ‖ C.1.ü

Diese beiden Regeln sagen zusammen etwa folgendes: Wenn man aus dem Zwischen-Symbol A die Symbolkette BC ableitet, dann erhält man die Übersetzung für A (d.h. A.0.ü) indem man die Übersetzung von B (d.h. B.1.ü) und die Übersetzung von C (d.h. C.1.ü) aneinander hängt (das Zeichen "‖" soll "aneinanderhängen" oder "konkatenieren" bedeuten).

Ein Übersetzungsschema besteht aus einer Reihe von syntaktischen Regeln, denen semantische Reglen zugeordnet wurden. Ein Übersetzungsschema kann die Übersetzung von unendlich vielen Worten einer Quell-Sprache präzise beschreiben, obwohl das Übersetzungsschema selbst nur endlich viele Zeilen lang ist.

3.1. Ein einfaches Übersetzungsschema

Das folgende Übersetzungsschema beschreibt, wie man (ganze) Dezimalzahlen in (ganze) Binärzahlen übersetzen kann. Die Sprache der Dezimalzahlen ist also die Quell-Sprache dieses Übersetzungsschemas und die Sprache der Binärzahlen ist seine Zielsprache. Das Zwischen-Symbol Z soll an "Zahl" und D an "Dezimal-Ziffer" (oder: digit) erinnern.

Syntaktische Regeln	semantische Regeln
R1: Z.0 → Z.1 D.1	Z.0.ü := Z.1.ü * 1010 + D.1.ü
R2: Z.0 → D.1	Z.0.ü := D.1.ü
R3: D.0 → 0	D.0.ü := 0
R4: D.0 → 1	D.0.ü := 1
R5: D.0 → 2	D.0.ü := 10
R6: D.0 → 3	D.0.ü := 11
R7: D.0 → 4	D.0.ü := 100
R8: D.0 → 5	D.0.ü := 101
R9: D.0 → 6	D.0.ü := 110
R10: D.0 → 7	D.0.ü := 111
R11: D.0 → 8	D.0.ü := 1000
R12: D.0 → 9	D.0.ü := 1001

Wichtiger Hinweis: In diesem Übersetzungsschema sind die Operationen * ("mal") und + ("plus") so genannte Compiler-Operationen. D.h. der Übersetzer muss diese Operationen selbst ausführen (und darf sie nicht in das Ziel-Programm einbauen und ihre Ausführung der Ziel-Maschine überlassen). In diesem Beispiel haben die Operationen * und + Binärzahlen als Operanden und liefern eine Binärzahl als Ergebnis.

Mit diesem Übersetzungsschema kann man jede Dezimalzahl in die entsprechende Binärzahl umwandeln. Als Beispiel übersetzen wir die Dezimalzahl 28 (in die Binärzahl 11100):

Zeilen-Nr.	Ableitung	Übersetzung
1	Z:1	Z:1.ü := Z:2.ü * 1010 + D:1.ü
2	R1:	:= 10 * 1010 + 1000
3		:= 11100
4	Z:2 D:1	Z:2.ü := D:2.ü
5	R2:	:= 10
6	D:2 D:1	D:2.ü := 10
7	R5:	
8	2 D:1	D:1.ü := 1000
9	R11:	
10	2 8	

Die Ableitung besagt folgendes über die syntaktische Struktur unseres "Quell-Programms" 28:
Die Zahl Z:1 besteht aus der Zahl Z:2, gefolgt von der Dezimalziffer D:1 (Zeile 1 bis 4). Die Zahl Z:2 besteht nur aus der Dezimalziffer D:2 (Zeile 4 bis 6). D:2 besteht nur aus dem End-Symbol 2 (Zeile 6 bis 8). D:1 besteht nur aus dem End-Symbol 8 (Zeile 8 bis 10).

Wie erhält man die Übersetzungen für die Dezimalziffern D:1 und D:2 und die Übersetzungen für die Zahlen Z:2 und Z:1? Da man auf D:1 die Regel R11 angewendet hat (Zeile 9) erhält man als Übersetzung D:1.ü direkt den Wert 1000. Entsprechendes gilt für D:2 und Z:2. Die Übersetzung für Z:1 erhält man, indem man die Übersetzung von Z:2 mit 1010 ("zehn") multipliziert und die Übersetzung von D:1 dazu addiert (Zeile 1 bis 3).

Aufgabe 3.1.1.: Geben Sie zu obigem Übersetzungsschema die Ableitung und Übersetzung der Zahl 123 an.

Aufgabe 3.1.2.: Geben Sie ein Übersetzungsschema für die Übersetzung von (ganzen) Binärzahlen in (ganze) Dezimalzahlen an.

Aufgabe 3.1.3.: Geben Sie ein Übersetzungsschema für die Übersetzung von Binärbrüchen (z.B. 1.1, 101.101, 0.001 etc.) in Dezimalbrüche (z.B. 1.5, 5.625, 0.125 etc.) an.

Aufgabe 3.1.4.: Welches mathematische Problem tritt auf, wenn man ein Übersetzungsschema für die Übersetzung von Dezimalbrüchen in Binärbrüche entwickeln will? Oder: Warum ist es leichter, Binärbrüche in Dezimalbrüche zu übersetzen als umgekehrt?

3.2. Ein Compiler (Übersetzungsschema) für einen Taschenrechner

Möglicherweise ist es schwierig, an dem sehr einfachen Beispiel im vorigen Abschnitt die "Fähigkeiten" von Übersetzungsschemen zu erkennen. Dezimalzahlen sind ja sooo viel einfacher als "richtige Programme". Deshalb wird in diesem Abschnitt ein Übersetzungsschema entwickelt, welches sehr einfache, aber "richtige Programme" zu übersetzen vermag. Vornehmer ausgedrückt: wir werden die Übersetzung von infix-dezimalen Ausdrücken in postfix-binäre Ausdrücke beschreiben.

3.2.1. Die Ziel-Sprache

Ein Taschenrechner mit einem Stapel (stack) beherrsche die 3 Befehle push, add und sub. Was diese Befehle "bewirken" wird im folgenden erläutert.

Der Befehl push hat eine Binärzahl als Parameter und legt diese auf den Stapel. Beispiel:

Stapel vorher	Befehl	Stapel nachher
110 11 1000	push 1010	1010 110 11 1000

Der Befehl add ersetzt die beiden obersten Zahlen auf dem Stapel durch ihre Summe. Beispiel:

Stapel vorher	Befehl	Stapel nachher
110 11 1000	add	1001 1000

Der Befehl sub ersetzt die beiden obersten Zahlen auf dem Stapel durch ihre Differenz ("zweitoberste Zahl minus oberste Zahl"). Beispiel:

Stapel vorher	Befehl	Stapel nachher
11 111 1000	sub	100 1000

Hier ein Beispiel für ein Programm, welches unser Taschenrechner ausführen kann:

push 101 push 10 sub push 100 add

Das folgende Diagramm soll die Ausführung dieses Programms veranschaulichen:

Befehle	Stapel	Erläuterung
	---	Leerer Stapel
push 101		
	101	
push 10		
	10	
	101	
sub		
	11	
push 100		
	100	
	11	
add		
	111	Ergebnis: sieben!

Vornehm ausgedrückt: unser Taschenrechner kann bestimmte Postfix-Ausdrücke, in denen binäre Zahlen als Argumente der Operationen sub und add vorkommen, auswerten. Dem Programm push 101 push 10 sub push 100 add entspricht der infix-dezimale Ausdruck (5 - 2) + 4.

3.2.2. Die Quell-Sprache

Als Quell-Sprache nehmen wir die uns vertraute Sprache aller infix-Ausdrücke, die man aus ganzen Dezimalzahlen, den Operations-Zeichen + und - und aus runden Klammern "(" und ")" zusammensetzen kann. Zu dieser Sprache gehören z.B. die Worte 123, 1+1, 47-(23+13-1) etc.

Hier eine kontextfreie Grammatik G18 für die Quell-Sprache:

R1:	A	→	A + S	R8:	D	→	0
R2:	A	→	A - S	R9:	D	→	1
R3:	A	→	S	R10:	D	→	2
R4:	S	→	(A)	R11:	D	→	3
R5:	S	→	Z	R12:	D	→	4
R6:	Z	→	Z D	R13:	D	→	5
R7:	Z	→	D	R14:	D	→	6
				R15:	D	→	7
				R16	D	→	8
				R17:	D	→	9

A wie "Ausdruck", S wie "Summand", Z wie "(Dezimal-) Zahl", D wie "Dezimalziffer". A ist das Startsymbol.

Aufgabe 3.2.2.1.: Übersetzen Sie "von Hand" folgende dezimal-infix-Ausdrücke in Programme für unseren Taschenrechner (d.h. in binär-postfix-Ausdrücke):

1. 1 + 2
2. (5 + 3) - (10 + 2)
3. (8 - (4 - (2 - 1)))
4. 8 - 4 - 2 - 1

3.2.3. Das Übersetzungsschema

Hier das Übersetzungsschema, mit dem man jedes Programm der Quell-Sprache (infix-dezimal-Ausdruck) in ein entsprechendes Programm der Zielsprache (binär-postfix-Ausdruck) übersetzen kann:

Syntaktische Regeln	Semantische Regeln
R1: A.0 → A.1 + S.1	A.0.ü := A.1.ü ‖ S.1.ü ‖ add
R2: A.0 → A.1 - S.1	A.0.ü := A.1.ü ‖ S.1.ü ‖ sub
R3: A.0 → S.1	A.0.ü := S.1.ü
R4: S.0 → (A.1)	S.0.ü := A.1.ü
R5: S.0 → Z.1	S.0.ü := push ‖ Z.1.ü
R6: Z.0 → Z.1 D.1	Z.0.ü := Z.1.ü * 1010 + D.1.ü
R7: Z.0 → D.1	Z.0.ü := D.1.ü
R8: D.0 → 0	D.0.ü := 0
R9: D.0 → 1	D.0.ü := 1
R10: D.0 → 2	D.0.ü := 10
R11: D.0 → 3	D.0.ü := 11
R12: D.0 → 4	D.0.ü := 100
R13: D.0 → 5	D.0.ü := 101
R14: D.0 → 6	D.0.ü := 110
R15: D.0 → 7	D.0.ü := 111
R16: D.0 → 8	D.0.ü := 1000
R17: D.0 → 9	D.0.ü := 1001

Wichtiger Hinweis: In diesem Übersetzungsschema sind die Operationen +, * und ‖ ("plus", "mal" und "konkatenieren") Compiler-Operationen, d.h. sie müssen vom Übersetzer ausgeführt werden (und dürfen im Ziel-Programm nicht mehr vorkommen). Dagegen braucht der Übersetzer nicht zu wissen, was die Symbolketten "push", "add" und "sub" bedeuten, er muss sie nur an den richtigen Stellen in das Ziel-Programm "einbauen. Die entsprechenden Operationen push, add und sub werden dann später vom Taschenrechner ausgeführt, wenn man ihm das Ziel-Programm eingibt. Die Operationen + und add "machen das gleiche", d.h. beide Operationen addieren binäre Zahlen. Trotzdem handelt es sich um ganz verschiedene Operationen, weil + vom Übersetzer und add von der Zielmaschine ausgeführt wird.

3.2.4. Eine Ableitung und Übersetzung

Mit dem obigen Übersetzungsschema kann man jedes Programm der Quell-Sprache in ein entsprechendes Programm der Ziel-Sprache übersetzen. Als ein einfaches Beispiel wird im folgenden die Ableitung und Übersetzung des Quell-Programms 12 - 8 wiedergegeben:

Zeilen-Nr.	Ableitung	Übersetzung
1	A:1	A:1.ü := A:2.ü ‖ S:1.ü ‖ sub
2	R2:	:= push 1100 push 1000 sub
3	A:2 - S:1	A:2.ü := S:2.ü
4	R3:	:= push 1100
5	S:2 - S:1	S:2.ü := push ‖ Z:1.ü
6	R5:	:= push 1100
7	Z:1 - S:1	Z:1.ü := Z:2.ü * 1010 + D:1.ü
8	R6:	:= 1 * 1010 + 10
9		:= 1100
10	Z:2 D:1 - S:1	Z:2.ü := D:2.ü
11	R7:	:= 1
12	D:2 D:1 - S:1	D:2.ü := 1
13	R9:	
14	1 D:1 - S:1	D:1.ü := 10
15	R10:	
16	1 2 - S:1	S:1.ü := push ‖ Z:3.ü
17	R5:	:= push 1000
18	1 2 - Z:3	Z:3.ü := D:3.ü
19	R7:	:= 1000
20	1 2 - D:3	D:3.ü := 1000
21	R16:	
22	1 2 - 8	

Aufgabe 3.2.4.1.: Geben Sie (entsprechend dem obigen Übersetzungsschema) die Ableitung und Übersetzung für den Ausdruck (4 + 3) an.

Aufgabe 3.2.4.2.: Geben Sie ein Schema zur Übersetzung von dezimalen-infix-Ausdrücken in "Taschenrechner-Programme" an. Die dezimalen-infix-Ausdrücke sollen die Operationen + und * enthalten können (sonst wie oben). Der Taschenrechner soll die Operation mult kennen (sonst wie oben). Für die dezimalen-infix-Ausdrücke soll gelten: "Punktrechnung geht vor Strichrechnung".

Aufgabe 3.2.4.3.: Betrachten Sie noch einmal das Übersetzungsschema im Abschnitt 3.2.3. Erläutern Sie den Unterschied zwischen Compiler-Operationen (z.B. + , * und ‖) und Operationen der Zielsprache (z.B. push, add und sub).

3.3. Ein Compiler für eine richtige Programmiersprache (Aufgabe)

Sie sollen jetzt ein Übersetzungsschema entwickeln, mit dem man Programme einer sehr einfachen aber "richtigen" Programmiersprache in eine sehr einfache aber "richtige" Maschinensprache übersetzen kann. Dazu werden zuerst die Quell- und die Ziel-Sprache beschrieben. Danach werden die nötigen semantischen Operationen und Variablen beschrieben. Schließlich folgt die genaue Aufgabenstellung.

3.3.1. Ziel-Sprache

Nehmen Sie an, Ihnen steht ein Programm-Ausführer (Computer) zur Verfügung, der einen Speicher (bestehend aus Zellen) und einen Stapel (stack) besitzt und folgende Befehle ausführen kann:

otto: ds 1010 (define storage): Eine Zelle des Speichers wird mit der Marke "otto" versehen und mit der Binärzahl 1010 initialisiert. Als Marken sind alle nichtleeren Folgen von Buchstaben und Ziffern erlaubt, die mit einem Buchstaben anfangen. Als Zahl rechts von "ds" sind alle nichtleeren Folgen der Binärziffern "0" und "1" erlaubt. Vor der Ziffernfolge darf ein Vorzeichen stehen ("+" oder "-").

push otto Der momentane Inhalt der Zelle (mit der Marke) otto wird auf den Stapel kopiert.

pop otto Der oberste Wert auf dem Stapel wird in die Zelle otto kopiert und vom Stapel entfernt.

lesseq (less or equal): Ersetzt die obersten beiden Werte auf dem Stapel durch "true" (wenn der zweitoberste Wert kleiner oder gleich dem obersten Wert ist) bzw. durch "false" (sonst).

equal Ersetzt die obersten beiden Werte auf dem Stapel durch "true" (wenn die obersten beiden Werte gleich sind) bzw. durch "false" (wenn sie ungleich sind).

unequal Ersetzt die obersten beiden Werte auf dem Stapel durch "true" (wenn die obersten beiden Werte ungleich sind) bzw. durch "false" (wenn sie gleich sind).

add Ersetzt die obersten beiden Werte auf dem Stapel durch ihre Summe.

sub Ersetzt die obersten beiden Werte auf dem Stapel durch ihre Differenz (zweitoberster Wert minus oberster Wert).

nop (no operation) Macht nichts.

Vor jeden Befehl kann man eine Marke gefolgt von einem Doppelpunkt ":" schreiben, z.B. so:

losgehts: push otto

x103: pop emil

Zu einer Marke (die nicht vor einem ds-Befehl steht) kann man mit den folgenden branch-Befehlen verzweigen:

bt losgehts (branch if true): Der oberste Wert wird vom Stapel entfernt. Wenn es der Wert "true" war, dann wird zur Marke losgehts verzweigt.

bf losgehts (branch if false): Der oberste Wert wird vom Stapel entfernt. Wenn es der Wert "false" war, dann wird zur Marke losgehts verzweigt.

b losgehts (branch): Es wird zur Marke losgehts verzweigt. Der Stapel wird nicht verändert.

Die Befehle add, sub, lesseq und unequal können nur ausgeführt werden, wenn die obersten beiden Werte auf dem Stapel Binärzahlen sind und nicht etwa Wahrheitswerte (andernfalls gibt der Programm-Ausführer eine Fehlermeldung aus und hält an). Entsprechend können die Befehle bt und bf nur ausgeführt werden, wenn der oberste Wert auf dem Stapel ein Wahrheitswert ist (und nicht etwa eine Binärzahl). Die ds-Befehle eines Programms müssen am Anfang des Programms oder hinter einem branch-Befehl stehen. Jeder Befehl sollte allein auf einer Zeile stehen.

Beispiel für ein Programm der Ziel-Sprache:

```
von:    ds      1
bis:    ds      1010
sum:    ds      0
eins:   ds      1
anf:    push    von
        push    bis
        lesseq
        bf      ende
        push    sum
        push    von
        add
        pop     sum
        push    von
        push    eins
        add
        pop     von
        b       anf
ende:   nop
```

3.3.2. Quell-Sprache

Die Quellsprache wird durch eine kontextfreie Grammatik beschrieben (wie denn sonst?).

Nicht-Terminale Symbole:

P (Programm), DL (Deklarationsliste), BL (Befehls-Liste), D (Deklaration), B (Befehl), N (Name), BA (bool´scher Ausdruck), GA (Ganzzahl-Ausdruck), BU (Buchstabe), ZA (Zahl), ZI (Ziffer).

Syntaktische Regeln

R1: P → prog DL begin BL end
R2: DL → D ; DL
R3 DL → D ;
R4: D → var N := ZA
R5: BL → B ; BL
R6: BL → B ;
R7: B → N := GA

```
R8:   B    →  if BA then BL else BL end
R9:   B    →  while BA do BL end
R10:  BA   →  ( GA = GA )
R11:  BA   →  ( GA ? GA )
R12:  BA   →  ( GA ? GA )
R13:  GA   →  N
R14:  GA   →  ( GA + GA )
R15:  GA   →  ( GA - GA )
R16:  N    →  BU N
R17:  N    →  BU
R18:  BU   →  a
R19:  BU   →  b
...   ...     ...
R43:  BU   →  z
R44:  ZA   →  ZA ZI
R45:  ZA   →  ZI
R46:  ZI   →  0
R47:  ZI   →  1
...   ...     ...
R55:  ZI   →  9
```

Beispiel für ein Programm der Quell-Sprache:

```
prog
  var von := 1;
  var bis := 10;
  var sum := 0;
  var eins  :=   1;
begin
  while (von ? bis) do
     sum  :=  (sum + von);
     von  :=  (von + eins);
  end;
end;
```

3.3.3. Eindeutige Marken in übersetzten if- und while-Befehlen

Der Übersetzer muss in der Lage sein, bei der Übersetzung von jedem if- und jedem while-Befehl neue Labels der Ziel-Sprache zu erzeugen, die noch nirgends sonst im Ziel-Programm vorkommen. Dazu besitzt der Übersetzer eine Variable namens $L. Zu Beginn hat diese Variable den Wert 0. Der Compiler kann die (Compiler-) Operation inc $L ausführen ("erhöhe den Wert von $L um 1"). Steht eine solche inc-Operation am Anfang einer semantischen Regel, so führt der Übersetzer sie aus und ersetzt dann alle (nicht unterstrichenen) Vorkommnisse von "$L" in der Regel durch den momentanen Wert der Variablen $L. Erst nach dieser Ersetzung wendet er die semantische Regel an.

Beispiel: Angenommen die Variable $L hat den Wert 17 und der Übersetzer will die Übersetzung für eine Instanz A:5 erzeugen entsprechend der semantischen Regel

```
A.0.ü  :=          inc    $L
           x$L:    push   abc
                   ...
                   b      x$L
```

Dann erhöht der Übersetzer die Variable $L um 1 (auf 18) und wendet folgende Regel an:

```
A.0.ü  :=  x18:    push   abc
                   ...
                   b      x18
```

Aufgabe 3.3.1.: Was machen die beiden oben angegebenen Beispiel-Programme?

Aufgabe 3.3.2.: Schreiben Sie zwei (möglichst einfache und schlichte) Programme zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen ("Algorithmus von Euklid"), eins in der Quell-Sprache und das andere in der Ziel-Sprache.

Aufgabe 3.3.3.: Geben Sie ein Übersetzungsschema an für die Übersetzung von Programmen der Quell-Sprache in Programme der Ziel-Sprache (dies ist hier die bei weitem wichtigste und umfangreichste Aufgabe!).

Aufgabe 3.3.4.: Übersetzen Sie mit dem Übersetzungsschema der vorigen Aufgabe das folgende Quell-Programm:

prog var a := 1; begin while (a ? a) do a := a; end; end

Dieses Programm macht nichts besonders sinnvolles, es lässt sich aber noch mit Papier und Bleistift (und etwas Geduld) übersetzen.

Aufgabe 3.3.5.: Geben Sie eine Übersetzung für while-do-Schleifen an, die effizienter ist als die, die in der Musterlösung von Aufgabe 3.3.3. (Regel R9) angegeben ist.

Aufgabe 3.3.6.: Erweitern Sie die Quell-Sprache und das Übersetzungsschema um folgende Befehle:

3.3.6.1. do-while-Schleifen, 3.3.6.2. until-do-Schleifen, 3.3.6.3. do-until-Schleifen, 3.3.6.4. if-then-end (ohne "else"), 3.3.6.5. for-Schleifen (ähnlich wie in Ada, z.B. „for i in a .. a+b do sum:=sum+i; end;“)

4. LL-Spracherkennung

4.1. Was ist ein Parser? Was heißt "LL"?

Sei G eine (kontextfreie) Grammatik. Ein Parser für G ist ein Programm P, dem man eine beliebige Symbolkette Z eingeben kann. Der Parser P versucht dann, die Symbolkette Z aus der Grammatik G abzuleiten. Wenn das gelingt, dann liefert der Parser die Ableitung (oder einen Syntaxbaum) als Ergebnis. Wenn es nicht möglich ist, Z aus G abzuleiten, dann gibt der Parser entsprechende Fehlermeldungen aus.

Meistens ist ein Parser kein eigenständiges Programm, sondern Teil eines Compilers.

Ein LL-Parser ist durch folgende Arbeitsweise charakterisiert (und die beiden L´s bedeuten folgendes):

1. Er liest und untersucht seine Eingaben-Symbolkette Z von links nach rechts.
2. Wenn er eine Ableitung als Ergebnis liefert, dann ist das immer eine linkskanonische Ableitung.

LL-Parser sind relativ einfach und relativ leicht verständlich, funktionieren aber auch nur für relativ einfache kontextfreie Grammatiken (für sog. LL-Grammatiken). LL-Parser gehen in einem bestimmten Sinn "von oben nach unten" vor (top down).

4.2. Ein einführendes Beispiel (LL-Parser)

Sei folgende Grammatik G19 gegeben (die "first-Mengen" rechts von den Regeln werden weiter unten erläutert):

R1:	S	→	abcS	first(abcS)	=	{a}
R2:	S	→	bS	first(bS)	=	{b}
R3:	S	→	Ac	first(Ac)	=	{d, e}
R4:	S	→	BD	first(BD)	=	{f, h, k}
R5:	A	→	dB	first(dB)	=	{d}
R6:	A	→	eB	first(eB)	=	{e}
R7:	B	→	C	first(C)	=	{ε, f, h}
R8:	C	→	ε	first(ε)	=	{ε}
R9:	C	→	fg	first(fg)	=	{f}
R10:	C	→	hi	first(hi)	=	{h}
R11:	D	→	klm	first(klm)	=	{k}

Zu parsen (zu analysieren) sei das Wort babcbhiklm. Aus dem aktuellen (unterstrichenen) Zeichen dieses Wortes ermitteln wir jeweils die Regel, die als nächstes angewendet werden muss:

	S	babcbhiklm
R2:		
		babcbhiklm
R1:		
	b	babcbhiklm
R2:		
	babc	babcbhiklm
R4:		
	babcb	babcbhiklm
R7:		

babcbD babcbhiklm

R10:

babcbD babcbhiklm

R11:

babcbhi babcbhiklm

Aufgabe 4.2.1.: Finden Sie nach dieser Methode eine Ableitung für das Wort fgklm

Ein LL-Parser für die Beispiel-Grammatik G19 besteht im wesentlichen aus 5 Prozeduren (für jedes Zwischen-Symbol S, A, B, C und D eine), einer Variablen next und den Hilfsprozeduren match und error. Hier die Prozeduren des LL-Parsers:

```
proc S is
  if     next is in {a}
    then match("abc"); S;
  elseif next is in {b}
    then match("b"); S;
  elseif next is in {d, e}
    then A; match("c");
  elseif next is in {f, h, k}
    then B; D;
  else error;
  end if;
end proc S;
```

```
proc A is
  if     next is in {d}
    then match("d"); B;
  elseif next is in {e}
    then match("e"); B
  else
      error;           -- Hier muss "else error" stehen!
  end if;
end proc A;
```

```
proc B is
  C;                   -- Hier brauchen wir kein if-elseif!
end proc B;
```

```
proc C is
  if     next is in {f}
    then match("fg");
  elseif next is in {h}
    then match("hi");
  else
      do_nothing;      -- Hier darf nicht "else error" stehen!
  end if;
end proc C;
```

```
proc D is
  match("klm");             -- Hier brauchen wir auch kein if-elseif!
end proc D;
```

Der Prozeduraufruf match("abc") bewirkt eine Prüfung, ob in der Eingabe-Symbolkette des Parsers (ab der Stelle, auf die next verweist) die Zeichenkette abc steht. Wenn ja, dann wird der next-Zeiger "hinter diese Symbolkette abc geschoben". Wenn nein, dann wird die Prozedur error aufgerufen.

Die Prozedur error gibt eine Fehlermeldung aus. Wenn error aufgerufen wird, dann bedeutet das soviel wie: die Eingabe-Symbolkette des Parsers kann nicht aus der betreffenden Grammatik abgeleitet werden, d.h. er enthält irgendwelche syntaktischen Fehler.

Damit ist die Beschreibung des LL-Parsers für obige Grammatik G19 abgeschlossen.

Aufgabe 4.2.2.: Analysieren Sie mit obigem Parser die Symbolkette "babcbhiklm". Führen Sie wirklich die einzelnen Prozeduren aus. Ohne diese (oder ähnliche) Aufgaben zu bearbeiten ist es möglicherweise ziemlich schwer, LL-Parser "richtig zu verstehen"!

Aufgabe 4.2.3.: Analysieren Sie mit obigem Parser die Symbolkette "efhc". Diese Symbolkette lässt sich nicht aus der Grammatik G19 ableiten. Wann erkennt der Parser einen Fehler?

Aufgabe 4.2.4.: Warum steht (bei obigem Parser für die Grammatik G19) am Ende der Prozedur A "else error", am Ende der Prozedur C dagegen "else do_nothing"?

Aufgabe 4.2.5.: Warum bestehen (bei obigem Parser für die Grammatik G19) die Prozeduren S, A und C aus einem if-Befehl, wohingegen die Prozeduren B und D keinen if-Befehl enthalten?

4.3. Die Anfangsmenge ("first-Menge") einer Satzform

In allen Beispielen dieses Abschnitts gehen wir von folgender Grammatik G20 aus:

R1:	S	→	aS	R7:	B	→	ef	R13:	E	→	kl
R2:	S	→	AB	R8:	C	→	gh	R14:	E	→	ε
R3:	S	→	CD	R9:	C	→	hi	R15:	F	→	mm
R4:	S	→	EF	R10:	C	→	ε	R16:	F	→	ε
R5:	A	→	cS	R11:	D	→	ij				
R6:	A	→	dS	R12:	D	→	jk				

Zur Wiederholung:

Definition: Eine Satzform (einer Grammatik G) ist eine beliebige Folge von End- und/oder Zwischen-Symbolen (der Grammatik G).

Beispiele:

accAFhD	eine Satzform der Grammatik G20, Länge dieser Satzform: 7.
AaAAkkk	eine Satzform der Grammatik G20, Länge dieser Satzform: 7.
FEC	eine Satzform der Grammatik G20, Länge dieser Satzform: 3.
acd	eine Satzform der Grammatik G20, Länge dieser Satzform: 3.
A	eine Satzform der Grammatik G20, Länge dieser Satzform: 1.
a	eine Satzform der Grammatik G20, Länge dieser Satzform: 1.
ε	eine Satzform der Grammatik G20, Länge dieser Satzform: 0.

Gegenbeispiele:

abc	keine Satzform von G20 (weil b kein End-Symbol von G20 ist).
Ha	keine Satzform von G20 (weil H kein Zwischen-Symbol von G20 ist).

Aus jedem Zwischen-Symbol A einer Grammatik (nicht nur aus dem Startsymbol) sollte man mindestens eine End-Symbol-Kette ableiten können. Denn sonst könnte man alle Regeln der Grammatik, in denen das Zwischen-Symbol A vorkommt, aus der Grammatik entfernen ohne die durch die Grammatik beschriebene Sprache zu verändern.

Vereinbarung: Ab jetzt setzen wir also voraus, dass wir es mit "vernünftigen" Grammatiken zu tun haben und man aus jedem Zwischen-Symbol mindestens eine End-Symbol-Kette ableiten kann.

Dann gilt aber auch: aus jeder Satzform kann man mindestens eine End-Symbol-Kette ableiten.

Beispiele (beziehen sich auf die Grammatik G20):

1. Aus der Satzform aEFc kann man genau 4 End-Symbol-Ketten ableiten, nämlich aklmmc, aklc, ammc und ac. Achtung: dabei ist es egal, ob man die Satzform aEFc selbst (aus dem Startsymbol der Grammatik ableiten kann oder nicht. Wir haben hier nur danach gefragt: "Was kann man aus aEFc ableiten?" (nicht: "Kann man aEFc ableiten?").

2. Aus der Satzform C kann man genau 3 End-Symbol-Ketten ableiten: gh, hi und ε.

3. Aus A kann man unendlich viele End-Symbol-Ketten ableiten: c, d, cc, dd, cjk, ... etc.

4. Aus der Satzform ac kann man genau eine End-Symbol-Kette ableiten, nämlich ac (dazu braucht man null Ableitungsschritte, d.h. die Ableitung besteht nur aus einer Zeile in der ac steht).

5. Aus ε kann man genau eine End-Symbol-Kette ableiten, nämlich ε.

Definition: Die Anfangsmenge einer Satzform enthält

1. die Anfangsstücke der Länge 1 aller aus ihr ableitbaren End-Symbol-Ketten und
2. falls die leere Symbolkette ε aus ihr ableitbar ist, auch die (leere) End-Symbol-Kette ε.

Oder mit anderen Worten:

Die Symbol-Kette a (der Länge 1) gehört genau dann zur Anfangsmenge einer Satzform XYZ wenn man aus XYZ eine End-Symbol-Kette ableiten kann, die mit a beginnt.

Die Symbol-Kette ε (der Länge 0) gehört genau dann zur Anfangsmenge einer Satzform XYZ wenn man aus XYZ die leere Symbolkette ε ableiten kann.

Beispiele (beziehen sich auf die Grammatik G20):

1. first(aEFc) = {a}
2. first(C) = {g, h, ε}
3. first{A} = {c, d}
4: first(ac) = {a}
5. first(ε) = {ε}

Aufgabe 4.3.1.: Bestimmen Sie für jede Regel der Grammatik G20 die first Menge der (Satzform auf) der rechten Seite der Regel, d.h. bestimmen Sie die Mengen first(aS), first(AB), first(CD) etc.

Aufgabe 4.3.2.: Bestimmen Sie für jedes Zwischen-Symbol der Grammatik G20 seine Anfangsmenge, d.h. bestimmen Sie die Mengen first(S), first(A), first(B) etc. (Da ein Zwischen-Symbol allein auch eine Satzform ist, ist diese Aufgabe möglich und sinnvoll!).

Aufgabe 4.3.3.: Seien α und β zwei Satzformen einer Grammatik. Dann ist auch αβ eine Satzform dieser Grammatik. Wie bestimmt man die Menge first(αβ)? Welche Fälle muss man unterscheiden? Berechnen Sie für die Grammatik G20 die Mengen first(DE), first(ED), first(AD) und first(EF).

Merke: Die Anfangsmenge kann man nur von Satzformen bilden. Es gibt keine "Anfangsmenge einer Regel" (Trotzdem gibt es immer wieder Studenten, die dieses Fabelwesen erwähnen. Vermutlich is es mit dem Einhorn, dem Siebenschwein und der Auftakteule verwandt). Es gibt allerdings "die Anfangsmenge der (Satzform auf der) rechten Seite einer Regel".

4.4. Die Folgemenge ("follow-Menge") eines Zwischen-Symbols

Anfangsmengen berechnen wir stets für Satzformen, besonders häufig für die Satzformen auf den rechten Seiten von Regeln einer Grammatik.

Folgemengen berechnen wir (nur) für einzelne Zwischen-Symbole einer Grammatik G. Eine Folgemenge enthält Endsymbol-Ketten der Länge 1 und kann zusätzlich die Symbolkette "$" enthalten. "$" bedeutet soviel wie: "Ende der Zeile" oder "Ende der abgeleiteten Symbolkette".

Definition: Eine End-Symbol-Kette a ist genau dann Element der Folgemenge follow(X), wenn man aus dem Startsymbol S der Grammatik eine Satzform

... Xa ...

ableiten kann, in der a auf X folgt.

Die Symbolkette $ ist Element von follow(X), wenn man aus S eine Satzform

... X

ableiten kann, in der X das letzte Symbol ist (d.h. auf X folgt "Ende der Zeile"). Ende der Def.

Beispiel: Berechnung der Folgemengen aller Zwischen-Symbole der Grammatik G20 (aus 4.3.):

Aus der Regel	kann man schließen
	1. Die Menge follow(S) enthält die Menge {$} (dies gilt immer!)
R1: S → aS	nichts
R2: S → AB	2. Die Menge follow(A) enthält die Menge first(B)\{ε} 3. Die Menge follow(B) enthält die Menge follow(S) Wenn man aus B die leere Symbolkette ableiten könnte, dann würde auch gelten: Die Menge follow(A) enthält die Menge follow(S))
R3: S → CD	4. Die Menge follow(C) enthält die Menge first(D)\{ε} 5. Die Menge follow(D) enthält die Menge follow(S) Wenn man aus D die leere Symbolkette ableiten könnte, dann würde auch gelten: Die Menge follow(C) enthält die Menge follow(S)
R4: S → EF	6. Die Menge follow(E) enthält die Menge first(F)\{ε} 7. Die Menge follow(F) enthält die Menge follow(S) Weil man aus F die leere Symbolkette ableiten kann, gilt auch: 8. Die Menge follow(E) enthält die Menge follow(S)
R5: A → cS	9. Die Menge follow(S) enthält die Menge follow(A)
R6: A → dS	10. Die Menge follow(S) enthält die Menge follow(A)
R7: B → ef ... R16: F → ε	nichts ... nichts

Außerdem gilt:
first(B) = {e}
first(D) = {i, j}
first(F)= {m, ε}

Aus 1. bis 10. können wir jetzt schließen, welche Elemente die Folgemengen der Zwischen-Symbole S, A, B, C, D, E und F enthalten:

Grund:	1.	2.	3.	4.	5.	6.	7.	8.	9.	3.	5.	8.	7.	
follow(S) = {	$,								e					}
follow(A) = {		e,												}
follow(B) = {			$,							e				}
follow(C) = {				i, j,										}
follow(D) = {					$,						e			}
follow(E) = {						m,		$,				e		}
follow(F) = {							$,						e	}

Aufgabe 4.4.1: Betrachten Sie die Grammatik G21:

R1:	S	→	ASB	R3:	A	→	aAa	R5:	B	→	BB
R2:	S	→	ε	R4:	A	→	ε	R6:	B	→	b

Berechnen Sie für diese Grammatik
1. die Anfangsmengen der rechten Seiten aller Regeln
(d.h. first(ASB), first(ε), first(aAa), first(ε), first(BB) und first(b))
2. die Anfangsmengen aller Zwischen-Symbole
(d.h. first(S), first(A) und first(B))
3. die Folgemengen aller Zwischen-Symbole
(d.h. follow(S), follow(A) und follow(B)).

Aufgabe 4.4.2: Betrachten Sie die Grammatik G22:

R1:	A	→	A + T	R4:	T	→	T * F	R7:	F	→	x
R2:	A	→	A - T	R5:	T	→	T / F	R8:	F	→	y
R3:	A	→	T	R6:	T	→	F	R9:	F	→	(A)

Berechnen Sie die first- und Folgemengen wie in der vorigen Aufgabe 4.4.1.

Aufgabe 4.4.3.: Betrachten Sie die Grammatik G23:

R1:	S	→	A B S C	R4:	A	→	a	R6:	B	→	b
R2:	S	→	ε	R5:	A	→	ε	R7:	B	→	ε
R3:	C	→	cS								

Berechnen Sie die first- und Folgemengen wie in der vorigen Aufgabe 4.4.2.

4.5. Konstruktion von LL-Parsern

Sei eine (kontextfreie) Grammatik G gegeben. Ein LL-Parser für G muss für jedes Zwischen-Symbol von G eine Prozedur enthalten (und der Einfachheit halber nennt man "die Prozedur für das Zwischen-Symbol A" meistens auch A). Wie konstruiert man die Prozedur A (für ein beliebiges Zwischen-Symbol A der Grammatik G) ?

1. Man nimmt alle Regeln der Grammatik G, die mit "A →" beginnen:

A	→	α_1
?	→	$\alpha 2$
	...	
?	→	$\alpha\nu$

und berechnet die Anfangsmengen der Satzformen $\alpha_?$, $\alpha_?$, ... α_n.

2. Dann unterscheidet man 2 Fälle:

2.1. Falls keine der Mengen first($\alpha_?$), first($\alpha_?$), ... first(α_n) die leere Symbolkette ε als Element enthält, dann sieht die Prozedur für A so aus:

```
proc A is
  if     next is in first(α?) then ...
  elseif next is in first(α?) then ...
  ...
  elseif next is in first(αn) then ...
  else error;      -- Achten Sie besonders auf diese Zeile
  end if;
end proc A;
```

2.2. Falls (mindestens) eine der Anfangsmengen ε als Element enthält, dann sieht die Prozedur für A so aus:

```
proc A is
  if     next is in first(α?)\{ε} then ...
  elseif next is in first(α?)\{ε} then ...
  ...
  elseif next is in first(αn)\{ε} then ...
  else do_nothing; -- Achten Sie besonders auf diese Zeile
  end if;
end proc A;
```

3. Was man anstelle der Pünktchen ... hinter die then´s schreibt hängt von dem entsprechenden α_i ab:

3.1. Ist α_i eine Folge von End-Symbolen, z.B. "abc", dann schreibt man: match("abc")

3.2. Ist α_i eine Folge von Zwischen-Symbolen, z.B. "DEF", dann schreibt man: D; E; F;

(d.h. man ruft nacheinander die Prozeduren D, E und F auf).

3.3. Ist α_i eine Satzform, die sowohl Zwischen- als auch End-Symbole enthält, z.B. "abcDEFghIJkL", dann schreibt man: match("abc"); D; E; F; match("gh"); I; J; match("k"); L;

Aufgabe 4.5.1.: Betrachten Sie folgende Grammatik G24:

R1:	S	→	ASB	R3:	A	→	aAa	R5:	B	→	bbB
R2:	S	→	ε	R4:	A	→	c	R6:	B	→	d

1. Berechnen Sie für die Grammatik G24 die Anfangsmengen der rechten Seiten aller Regeln (d.h. die Mengen first (ASB), first(e), first(aAa), first(c), first(bbB) und first(d))
2. Konstruieren Sie einen LL-Parser für die Grammatik G24 (d.h. geben Sie die Prozeduren namens S, A und B an. Die Prozeduren match und error dürfen Sie als schon gegeben voraussetzen).
3. Analysieren Sie mit diesem Parser die folgenden Worte:
3.1. cd
3.2. acabbd
3.3. cbbb

Warnung: Nach dem hier beschriebenen "Rezept" kann man zu jeder kontextfreien Grammatik "eine Reihe von Prozeduren" konstruieren. Aber: diese Prozeduren funktionieren nur dann als (LL-) Parser, wenn die Grammatik zwei Bedingungen erfüllt, die wir hier als LL-Bedingung-1 und LL-Bedingung-2 bezeichnen wollen. Diese beiden Bedingungen werden in den folgenden Abschnitten behandelt.

Eine (kontextfreie) Grammatik, die diese beiden Bedingungen (die LL-Bedingung-1 und die LL-Bedingung-2) erfüllt, wird auch als LL-Grammatik bezeichnet, oder man sagt: "die Grammatik ist LL".

4.6. Die LL-Bedingung-1

Aufgabe 4.6.1.: Konstruieren Sie den LL-Parser zu folgender Grammatik G25:

R1:	S	→	ab
R2:	S	→	ac

und analysieren Sie mit diesem Parser das Wort "ac". (Falls das wider Erwarten "gut gehen" sollte, dann analysieren Sie mit diesem Parser das Wort "ab"). Die beiden Worte "ab" und "ac" gehören offenbar zu der Sprache, die von der Grammatik beschriebenen wird, aber der Parser kann sie nicht beide richtig analysieren. Erklären Sie bitte: warum funktioniert der Parser nicht?

Definition: LL-Bedingung-1

Für jedes Zwischen-Symbol A (für das es mehr als eine Regel A → ... gibt) muss gelten:
Die Anfangsmengen der rechten Seiten aller Regeln, die mit "A → " beginnen, müssen paarweise disjunkt sein.
Wenn es also folgende Regeln für A gibt:

A	→	$\alpha_?$
?	?	$?_?$
	???	
?	?	$?_n$

dann muss gelten: die Mengen first($\alpha_?$), first($\alpha_?$), ..., first(α_n) müssen paarweise disjunkt sein (damit nicht mehrere Regeln "in Frage kommen", wenn man das nächste Eingabezeichen kennt).

Achtung: $M1 \cap M2 \cap M3 = \{\}$ bedeutet nicht, dass die Mengen M1, M2 und M3 paarweise disjunkt sind.

Beispiel: es gilt $\{a, b\} \cap \{b, c\} \cap \{c, a\} = \{\}$, aber die Mengen {a, b}, {b, c} und {c, a} sind nicht paarweise disjunkt, da unter anderem $\{a, b\} \cap \{b, c\} = \{b\}$ gilt.

Aufgabe 4.6.2.: Erfüllt die folgende Grammatik G26 die LL-Bedingung-1?

R1:	S	→	ASB	R3:	A	→	aAa	R5:	B	→	bbB
R2:	S	→	ε	R4:	A	→	c	R6:	B	→	d

4.7. Die LL-Bedingung-2

Aufgabe 4.7.1.: Betrachten Sie folgende Grammatik G27:

R1:	S	→	Ax
R2:	A	→	x
R3:	A	→	ε

1. Zeigen Sie, dass man aus G27 die beiden Worte x und xx ableiten kann.
2. Zeigen Sie, dass G27 die LL-Bedingung-1 erfüllt.
3. Konstruieren Sie den LL-Parser für G27.
4. Analysieren Sie mit diesem Parser die Worte xx und x.
5. Erläutern Sie: warum funktioniert der Parser bei der Analyse des Wortes x nicht?

Lösung 4.7.1.: Wenn der Parser das Wort x bzw. das Wort xx analysiert, gerät er in die folgenden beiden Situationen:

Situation 1:

	S xx
R1:	
	Ax xx

Situation 2:

	S x
R1:	
	Ax x

Der Parser kann die beiden Situationen nicht unterscheiden, da er in beiden Situationen nur das nächste Eingabesymbol sieht (in beiden Situationen x) und aus der Satzform Ax etwas ableiten muss, das mit x beginnt. Der Parser muss entscheiden, ob er die Regel R2 oder die Regel R3 anwendet. Da er die beiden Situationen nicht unterscheiden kann, wird er sich in beiden Situationen gleich entscheiden. Entscheidet er sich für R2, dann funktioniert er in Situation 1 richtig und in Situation 2 falsch. Entscheidet er sich für R3, dann funktioniert er in Situation 1 falsch und in Situation 2 richtig.

Kern des Problems: Aus A kann man die leere Symbolkette ε ableiten und von dem x in der Eingabe ist nicht eindeutig klar, ob es zur Menge first(A) gehört oder zur Menge follow(A).

Definition: LL-Bedingung-2

Für jedes Zwischen-Symbol A, aus dem man die leere Symbolkette ε ableiten kann, muss gelten:
die Mengen first(A) und follow(A) müssen disjunkt sein.

Aufgabe 4.7.2.: Betrachten Sie folgende Grammatik G28:

R1:	A	→	BC	R3:	B	→	bB	R5:	C	→	cC
R2:	A	→	a	R4:	B	→	ε	R6:	C	→	ε

1. Aus welchen Zwischen-Symbolen kann man die leere Symbolkette ε ableiten?
2. Erfüllt die Grammatik G28 die LL-Bedingung-2?

Aufgabe 4.7.3.: Betrachten Sie folgende Grammatik G29 (Achtung: jeweils 2 große Buchstaben stellen hier ein Zwischen-Symbol dar, jeder kleine Buchstabe ist ein End-Symbol):

R1:	BF	→	(BE RF	
R2:	RF	→	; BE RF	-- Achtung: zwischen BE und RF liegt kein Blank
R3:	RF	→	)	
R4:	BE	→	EB	
R5:	BE	→	ZB	
R6:	EB	→	ID:=AA	
R7:	EB	→	read(ID)	
R8:	EB	→	write(ID)	
R9:	ZB	→	ifBAthenBEelseBEend	
R10:	ZB	→	untilBAdoBEend	
R11:	ZB	→	BF	
R12:	AA	→	ID RA	-- Achtung: zwischen ID und RA liegt kein Blank
R13:	RA	→	+ID	
R14:	RA	→	- ID	
R15:	BA	→	ID RB	-- Achtung: zwischen ID und RB liegt kein Blank
R16:	RB	→	=ID	
R17:	RB	→	>ID	
R18:	ID	→	x	
R19	ID	→	y	

Erläuterung der Buchstaben-Kombinationen der Zwischen-Symbole:

BF	Befehls-Folge	ID	Identifier
BE	Befehl	AA	arithmetischer Ausdruck
RF	Rest einer Folge	RA	Rest eines AA
EB	elementarer Befehl	BA	boolscher Ausdruck
ZB	zusammengesetzter Befehl	RB	Rest eines BA

1. Zeigen Sie, dass G29 eine LL-Grammatik ist (d.h. dass G29 die LL-Bedingung-1 und die LL-Bedingung-2 erfüllt)
2. Schreiben Sie neben jede Regel der Grammatik G29 die Anfangsmenge ihrer rechten Seite.
3. Führen Sie "von Hand" eine LL-Analyse durch für das Wort "(read(x);ifx>ythenwrite(x)elsey:=x+yend)"
4. Konstruieren Sie einen LL-Parser für die Grammatik G29.

4.8. Grammatiken verbessern ("LL machen"), Teil 1

Viele Grammatiken sind zwar nicht LL, können aber, ohne die beschriebene Sprache zu verändern, in eine LL-Grammatik "umgebaut" oder "verbessert" werden.

Aufgabe 4.8.1.: Warum ist die folgende Grammatik G30 nicht LL:

R1: S → Sb
R2: S → a

Definition: rekursive, links-rekursive, rechts-rekursive Regel

Sei R eine kontextfreie Regel und A das Zwischen-Symbol auf der linken Seite von R.

1. R heißt rekursiv, wenn A auch auf der rechten Seite von R vorkommt, z.B. so:
R: A → BaAbAC

2. R heißt links-rekursiv, wenn die rechte Seite von R mit A beginnt, z.B. so:
R: A → AbC

3. R heißt rechts-rekursiv, wenn die rechte Seite von R mit A endet, z.B. so:
R: A → CbA

Die folgende Regel ist gleichzeitig links-rekursiv und rechts-rekursiv:
R: A → AbCdA

Merke: Eine Grammatik, die (auch nur) eine links-rekursive Regel enthält, kann nicht LL sein (siehe oben, Aufgabe 4.8.1), da sie die LL-Bedingung-1 ist bestimmt nicht erfüllt.

Hier ein Schema, mit dem man einfache links-rekursive Regeln aus einer Grammatik "verbessern" kann, ohne die durch die Grammatik beschriebene Sprache zu verändern:

Einfaches Schema (für die Beseitigung von linksrekursiven Regeln):

R1: A → Aα R2: A → β	⇒	R1: A → βR R2: R → αR R3: R → ε

Dabei sind α und β irgendwelche Satzformen (d.h. Folgen von Zwischen- und/oder End-Symbolen) und R ist ein "neues" Zwischen-Symbol (welches in der ursprünglichen Grammatik noch nicht vorkam).

Beispiel: Eine Grammatik für arithmetische Ausdrücke:

R1: A → A + B R2: A → B R3: B → B * C R4: B → C R5: C → (A) R6: C → id	⇒	R1: A → B R R2: R → + B R R3: R → ε R4: B → C Q R5: Q → * C Q R6: Q → ε R7: C → (A) R8: C → id

Aufgabe 4.8.2: Bauen Sie folgende Grammatik G30 so um, dass sie keine links-rekursive Regel mehr enthält:

R1: S → ScSc
R2: S → ab

Allgemeines Schema (für die Beseitigung von linksrekursiven Regeln):

$A \rightarrow A\,\alpha_?$ $A \rightarrow A\,\alpha_?$??? $A \rightarrow A\,\alpha_m$ $A \rightarrow \beta_?$ $A \rightarrow \beta_?$??? $A \rightarrow \beta_n$	$\Rightarrow$	$A \rightarrow \beta_?{}_?R$ $A \rightarrow \beta_?{}_?R$ ... $A \rightarrow \beta_n\,R$ $R \rightarrow \alpha_?{}_?R$ $R \rightarrow \alpha_?{}_?R$ ... $R \rightarrow \alpha_m\,R$ $R \rightarrow \varepsilon$

Die α_i und β_j sind Satzformen, kein β_j beginnt mit einem A, R ist ein "neues" Zwischen-Symbol.

Aufgabe 4.8.3.: "Verbessern" Sie folgende Grammatik G31 zu einer LL-Grammatik:

R1:	A	$\rightarrow$	A + B	R5:	B	$\rightarrow$	B * C
R2:	A	$\rightarrow$	A - B	R6:	B	$\rightarrow$	B / C
R3:	A	$\rightarrow$	B	R7:	B	$\rightarrow$	C
R4:	C	$\rightarrow$	(A)	R8:	C	$\rightarrow$	id

Außer der bisher behandelten direkten Linksrekursion ("Linksrekursion in einer Regel") kann auch eine indirekte Linksrekursion ("Linksrekursion über mehrere Regeln) eine Grammatik daran hindern, LL zu sein. Beispiel:

R1:	A	$\rightarrow$	BcD
R2:	B	$\rightarrow$	CDab
R3:	C	$\rightarrow$	AbC
	...		

Aus A kann man in 3 Schritten AbCDabBcD ableiten, d.h. diese Grammatik "leidet" unter indirekter Linksrekursion.

Idee zur Beseitigung indirekter Linksrekursionen:

1. Man ordnet die Zwischen-Symbole, d.h. man numeriert sie: A_1, A_2, ..., A_n. Dann führt man für jedes A_i die folgenden beiden Schritte durch:
2.1. In den Regeln für A_i ("mit A_i auf der linken Seite") erlaubt man am Anfang der rechten Seite höchstens ein Aj mit j ? i. A-jots mit j < i ersetzt man durch das, was man aus ihnen ableiten kann.
2.2. Direkte Linksrekursion (i = j) beseitigt man jeweils wie oben beschrieben.

Aufgabe 4.8.4.: Entfernen Sie aus folgender Grammatik G32 alle (direkten und indirekten) Linksrekursionen:

R1:	S	$\rightarrow$	Aa	R3:	A	$\rightarrow$	Ac
R2:	S	$\rightarrow$	b	R4:	A	$\rightarrow$	Sd
				R5:	A	$\rightarrow$	c

Frage: Ist die Lösungs-Grammatik LL?

4.9. Grammatiken verbessern ("LL machen"), Teil 2

Aufgabe 4.9.1.: Warum ist folgende Grammatik G33 nicht LL?

R1:	S	$\rightarrow$	if B then S else S fi
R2:	S	$\rightarrow$	if B then S fi
R3:	S	$\rightarrow$	s
R4:	B	$\rightarrow$	b

Lösung 4.9.1.: Weil die rechten Seiten von zwei Regeln (R1 und R2) für das selbe Zwischen-Symbol (nämlich S) mit der gleichen Symbolkette (nämlich "if B then S") beginnen.

Allgemeines Schema (zur Beseitigung von "gemeinsamen Anfangsstücken" von rechten Seiten von Regeln):

R1: $A \rightarrow \alpha\,\beta_?$ R2: $A \rightarrow \alpha\,\beta_?$	$\Rightarrow$	R1: $A \rightarrow \alpha_?R$ R2: $R \rightarrow \beta_?$ R3: $R \rightarrow \beta_2$

α, $\beta_?$ und $\beta_?$ sind Satzformen, α ist das "gemeinsame Anfangsstück" von R1 und R2, $\beta_?$ und $\beta_?$ unterscheiden sich im 1. Symbol, R ist ein "neues" Zwischen-Symbol.

Diese Technik (gemeinsame Anfangsstücke zu beseitigen) wird auch Ausklammern genannt, weil sie Ähnlichkeit mit dem Ausklammern in arithmetischen Ausdrücken hat:

$\alpha * \beta_? + \alpha * \beta_? = \alpha * (\beta_? + \beta_?)$

Aufgabe 4.9.2.: "Verbessern" Sie die Grammatik G33:

R1:	S	→	if B then S else S fi
R2:	S	→	if B then S fi
R3:	S	→	s
R4:	B	→	b

Frage: Ist die Lösungsgrammatik LL?

5. LR-Parser

Viele "wünschenswerte" Grammatiken sind nicht LL und lassen sich auch nicht in LL-Grammatiken "umbauen". Die so genannte LR-Parsier-Technik ist echt stärker als die LL-Parsier-Technik, d.h. jede LL-Grammatik ist auch eine LR-Grammatik, aber nicht umgekehrt.

Ein LR-Parser ist durch folgende Arbeitsweise charakterisiert (und LR bedeutet folgendes):

1. Er liest und untersucht seine Eingabe-Symbolkette Z von links nach rechts (wie ein LL-Parser).
2. Wenn er eine Ableitung als Ergebnis liefert, dann ist das immer eine rechtskanonische Ableitung.

Grundidee des LR-Parsens:

Man betrachtet die Eingabe-Symbolkette von links nach rechts bis man die rechte Seite einer Regel erkennt. Dann ersetzt man diese rechte Regelseite in der Eingabe durch die linke Seite der Regel (d.h. durch ein Zwischen-Symbol). Das macht man solange, bis man nur noch das Startsymbol hat (dann ist die Analyse gelungen) oder bis "nichts mehr geht" (dann ist die Analyse mißlungen). Beim LR-Parsen werden die Regeln der Grammatik also "verkehrt herum" verwendet: um die rechte Seite durch die linke Seite zu ersetzen. Diese Art, die Regeln anzuwenden, nennt man auch reduzieren (man reduziert die rechte Seite einer Regel zu ihrer linken Seite). LR-Parser gehen in einem bestimmten Sinn "von unten nach oben" vor ("bottom up").

Beispiel: Wir gehen von folgender Grammatik G34 aus:

R1:	S	→	AB	R5:	B	→	gh
R2:	S	→	AC	R6:	B	→	gi
R3:	A	→	de	R7:	C	→	gj
R4:	A	→	df				

Zu analysieren sei das Wort "d f g j":

	d f g j
	R3 oder R4?
	d f g j
R4:	d f wird mit der Regel R4 zu A reduziert
	A g j
	R5, R6 oder R7?
	A g j
R7:	g j wird mit R7 zu C reduziert
	A C
	R1 oder R2?
	AC
R2:	A C wird mit R2 zu S reduziert
	S

Beim LL-Parsen entscheidet man nur anhand des ersten Zeichens einer Symbolkette Z, mit welcher Regel Z abgeleitet wurde. Beim LR-Parsen entscheidet man erst nach dem Lesen von ganz Z, mit welcher Regel Z abgeleitet wurde. Deshalb ist "LR stärker als LL".

5.1. LR-Parsen "mit Gefühl"

Die grundlegende Vorgehensweise beim LR-Parsen soll anhand eines Beispiels erläutert werden.

Beispiel 5.1.1.: Gesucht ist eine Ableitung für das Wort `x/x-x` aus der Grammatik G35:

R1: A → A-B
R2: A → B
R3: B → B/C
R4: B → C
R5: C → x

Für eine LR-Analyse verwendet man zwei Variablen: einen Stapel ST und eine Eingabe EIN. Die Werte (Inhalte) dieser beiden Variablen in einem bestimmten Moment nennt man eine Konfiguration.

In der Anfangs-Konfiguration ist der Stapel leer. Die Eingabe enthält die zu analysierende Symbolfolge. Um das Ende dieser Symbolfolge deutlich zu kennzeichnen, wird sie mit einem speziellen Sonderzeichen abgeschlossen. Dazu verwenden wir hier wie üblich das Dollar-Zeichen "$" (z.B. so: `x/x-x$`).

Den Stapel notieren wir hier so, dass er von links nach rechts gefüllt und von rechts nach links geleert wird. Die "Spitze" des Stapels liegt hier also rechts, nicht oben. Trotzdem werden wir wie üblich "oben auf dem Stapel" und ähnliches sagen.

Das am weitesten links stehende Symbol in der Eingabe wird auch als aktuelles Eingabe-Symbol bezeichnet. Anfangs ist das erste Symbol der Eingabe aktuell.

Eine LR-Analyse besteht im wesentlichen aus zwei Arten von Aktionen: schieben und reduzieren (und lächeln, wenn wir nach der Startregel der Grammatik reduzieren).

Beim Schieben entfernen wir das aktuelle Symbol aus der Eingabe und legen es ("pushen es") oben auf den Stapel.

Wenn wir erkennen, dass oben auf dem Stapel die rechte Seite einer Grammatik-Regel liegt, dann können wir sie reduzieren. Reduzieren nach Regel R_n heißt: die rechte Seite der Regel R_n vom Stapel entfernen ("poppen") und die linke Seite der Regel auf den Stapel legen ("pushen").

Die Analyse des Wortes `x/x-x` kann man etwa so darstellen:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1		x/x-x$	Schieben
2	x	/x-x$	Reduzieren nach R5
3	C	/x-x$	Reduzieren nach R4
4	B	/x-x$	Schieben
5	B/	x-x$	Schieben
6	B/x	-x$	Reduzieren nach R5
7	B/C	-x$	Reduzieren nach R3
8	B	-x$	Reduzieren nach R2
9	A	-x$	Schieben
10	A-	x$	Schieben
11	A-x	$	Reduzieren nach R5
12	A-C	$	Reduzieren nach R4
13	A-B	$	Reduzieren nach R1
14	A	$	Lächeln

Diese Analyse zeigt, dass man das Wort `x/x-x` durch Anwendung der Regeln R5, R4, R5, R3, R2, R5, R4, R1 (in dieser Reihenfolge) zum Startsymbol A der Grammatik reduzieren kann. Die umgekehrte Folge von Regeln (R1, R4, R5, R2, R3, R5, R4, R5) charakterisiert eindeutig eine rechtskanonische Ableitung des Wortes `x/x-x` aus dem Startsymbol A der Grammatik.

Diese Analyse war also erfolgreich. Sie glich aber "einem Ritt über den Bodensee": an verschiedenen Stellen hätten wir "einbrechen" können.

Z.B. hätten wir in Konfiguration 4 das B auf dem Stapel nach Regel R2 zu A reduzieren können (statt zu schieben). Das hätte uns in eine "Sackgasse" gebracht, und wir hätten keine Ableitung gefunden.

Ganz ähnlich wäre es uns ergangen, wenn wir in Konfiguration 13 nach R2 reduziert hätten (statt nach R1) oder wenn wir in Konfiguration 2 geschoben hätten, statt (nach R5) zu reduzieren.

Im allgemeinen können wir bei jedem Schritt zwischen verschiedenen Aktionen wählen. Wenn wir falsch wählen, verlaufen wir uns möglicherweise und finden keine Ableitung, obwohl es eine gibt.

In diesem Abschnitt verlassen wir uns darauf, dass der Leser "mit Gefühl", mit Intuition, mit scharfem Hinsehen oder mit "Ausprobieren" immer die richtige Wahl findet.

Aufgabe 5.1.1.: In der obigen Analyse ist in Konfiguration 1 nur die Aktion "Schieben" möglich. In Konfiguration 2 mussten wir zwischen "Schieben" und "Reduzieren nach R5" wählen (und haben uns für "Reduzieren nach R5" entschieden). Geben Sie für jede Konfiguration an, zwischen welchen Aktionen wir wählen konnten/mussten.

Aufgabe 5.1.2.: Nehmen Sie obige Grammatik G35 und analysieren Sie "mit Gefühl" das Wort `x/x` und das Wort `x-x/x`. In beiden Fällen ist eine erfolgreiche Analyse möglich, aber man muss sich bei jedem Schritt richtig entscheiden. Sie möchten doch lächeln, oder?

Aufgabe 5.1.3.: Gegeben ist die Grammatik G36:

R1:	A	→	A+B
R2:	A	→	B
R3:	B	→	B*C
R4:	B	→	C
R5:	C	→	C^D
R6:	C	→	D
R7:	D	→	x

Analysieren Sie "mit Gefühl" das Wort `x*x+x^x`. Wer möchte, kann sich unter "^" einen Potenz-Operator vorstellen. Aber Vorsicht: dieser Operator ist hier linksassoziativ (und nicht wie üblich rechtsassoziativ).

5.2. LR-Parsen mit einer Tabelle

Im vorigen Abschnitt haben wir beim Analysieren eines Wortes "mit Gefühl" zwischen verschiedenen Aktionen gewählt. In diesem Abschnitt soll das Gefühl durch eine Tabelle ersetzt werden. Die Tabelle sagt uns bei jedem Schritt genau, was wir tun sollen.

Wir benützen als Beispiel wieder die Grammatik G35. Allerdings müssen wir sie (aus Gründen, die möglicherweise später mal klar werden) um ein neues Start-Symbol und eine Regel

R0: neues-Startsymbol ® altes-Startsymbol

ergänzen. Diese "Ergänzung" ist völlig harmlos und lässt sich an jeder Grammatik leicht vornehmen.

Die "ergänzte" Grammatik G35:

R0:	S	→	A
R1:	A	→	A-B
R2:	A	→	B
R3:	B	→	B/C
R4:	B	→	C
R5:	C	→	x

Eine LR-Parser-Tabelle für diese Grammatik kann etwa so aussehen:

Zustand	Aktion				Folgezustand		
	x	-	/	$	A	B	C
Z0	s4				Z1	Z2	Z3
Z1		s5		lächeln			
Z2		r2	s6	r2			
Z3		r4	r4	r4			
Z4		r5	r5	r5			
Z5	s4					Z7	Z3
Z6	s4						Z8
Z7		r1	s6	r1			
Z8		r3	r3	r3			

Eine Analyse des Wortes `x/x-x` sieht dann so aus:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	`Z0`	`x/x-x$`	s4
2	`Z0 x Z4`	`/x-x$`	r5
3	`Z0 C Z3`	`/x-x$`	r4
4	`Z0 B Z2`	`/x-x$`	s6
5	`Z0 B Z2 / Z6`	`x-x$`	s4
6	`Z0 B Z2 / Z6 x Z4`	`-x$`	r5
7	`Z0 B Z2 / Z6 C Z8`	`-x$`	r3
8	`Z0 B Z2`	`-x$`	r2
9	`Z0 A Z1`	`-x$`	s5
10	`Z0 A Z1 - Z5`	`x$`	s4
11	`Z0 A Z1 - Z5 x Z4`	`$`	r5
12	`Z0 A Z1 - Z5 C Z3`	`$`	r4
13	`Z0 A Z1 - Z5 B Z7`	`$`	r1
14	`Z0 A Z1`	`$`	lächeln

Neu sind hier vor allem die Zustände. Auf dem Stapel steht immer abwechselnd ein Zustand, ein Symbol (der Grammatik), ein Zustand, ein Symbol ... etc. Zuoberst liegt immer ein Zustand, der auch als aktueller Zustand bezeichnet wird.

Die nächste Aktion ist jeweils von zwei "Dingen" abhängig:

1. vom aktuellen Zustand (steht ihm Stapel ganz rechts)
2. vom aktuellen Eingabe-Symbol (steht in der Eingabe ganz links)

Die Parser-Tabelle besteht aus zwei Teilen: einem Aktions-Teil und einem Folgezustands-Teil. Im Aktions-Teil kann man jeweils die nächste Aktion nachsehen (in Abhängigkeit vom aktuellen Zustand und vom aktuellen Eingabe-Symbol). Der Folgezustands-Teil der Tabelle wird im Zusammenhang mit Reduzier-Aktionen benützt (siehe unten).

Es gibt vier Arten von Aktionen: Schieben, Reduzieren, Lächeln und einen Fehler melden ("weinen").

1. Schieben: Das aktuelle Eingabe-Symbol und ein Zustand werden oben auf den Stapel gelegt ("gepusht").

Beispiel: Die Aktion s4 bewirkt, dass das aktuelle Eingabe-Symbol und der Zustand Z4 auf den Stapel gelegt werden. Die Aktion s137 bewirkt, dass das aktuelle Eingabe-Symbol und der Zustand Z137 auf den Stapel gelegt werden, etc.

2. Reduzieren: Eine bestimmte Anzahl von Zeichen (Zustände und Symbole) werden vom Stapel entfernt ("gepoppt"). Die linke Seite einer Regel und ein Folgezustand werden auf den Stapel gelegt ("gepusht").

Beispiel: In obiger Analyse entspricht in der Konfiguration 7 der Stapel-Inhalt

`Z0 B Z2 / Z6 C Z8` der Satzform `B/C` (um das zu sehen, sollte man die Zustände auf dem Stapel einfach ignorieren). `B/C` ist die rechte Seite der Regel R3 (der Grammatik G35). Die Aktion r3 besteht aus zwei Schritten:

1. Auf dem Stapel wird `B Z2 / Z6 C Z8` ersetzt durch B (die linke Seite der Regel R3). Stapelinhalt danach: `Z0 B`.

2. Aus der Parser-Tabelle wird der Folgezustand für `Z0` und `B` geholt und dahinter geschrieben. Stapelinhalt danach: `Z0 B Z2`.

3. Lächeln: Wird offiziell auch als "akzeptieren" bezeichnet. Die Analyse wird erfolgreich beendet, eine Ableitung wurde gefunden.

Beispiel: In obiger Analyse ist in Konfiguration 14 der aktuelle Zustand gleich `Z1` und das aktuelle Eingabe-Symbol gleich `$`. Laut Aktions-Teil der Parser-Tabelle ist in diesem Fall Lächeln angesagt.

4. Fehler melden: Wenn als nächste Aktion in der Aktions-Tabelle ein leerer Eintrag steht, dann wird die Analyse erfolglos beendet. Eine Ableitung wurde in einem solchen Fall nicht gefunden.

Aufgabe 5.2.1.: Analysieren Sie ohne Gefühl, streng nach obiger Parser-Tabelle (die für die Grammatik G35 gilt) das Wort `x/x` und das Wort `x-x/x`.

5.3. Eine Parser-Tabelle für eine Grammatik erstellen

Anhand eines Beispiels soll gezeigt werden, wie man zu einer gegebenen Grammatik (G35, welche sonst?) eine LR-Parser-Tabelle konstruieren kann.

Die Konstruktion erfolgt in vier Schritten:

1. Wir erzeugen aus der Grammatik die Zustände für die Tabelle.
2. Aus den Zuständen konstruieren wir einen endlichen Automaten.
3. Wir berechnen für jedes Zwischen-Symbol Z der Grammatik die Menge follow(Z).
4. Aus dem endlichen Automaten und den Folgemengen konstruieren wir die gesuchte Parser-Tabelle.

Schritt 1: Die Zustände erzeugen
Ein Zustand (des LR-Parsers) ist eine Menge von Regeln mit Punkten (kurz: REMPs).

Beispiel: Die REMP `A` → `•A-B` drückt aus, dass wir in der Eingabe eine Symbolfolge erwarten, die der Satzform `A-B` entspricht. Die REMP `A` → `A•-B` drückt aus, dass wir eine dem `A` entsprechende Symbolfolge eingelesen haben und jetzt eine Symbolfolge erwarten, die der Satzform `-B` entspricht. Die Bedeutung der REMP `A` → `A-•B` ergibt sich entsprechend: wir haben `A-` schon erkannt und erwarten jetzt `B`. Die REMP `A` → `A-B•` bedeutet, dass wir `A-B` erkannt haben (und somit bereit und in der Lage sind, nach der Regel `A` → `A-B` zu reduzieren).

Zu einem Zustand gehören alle REMPs, die aufgrund der schon verarbeiteten Eingabe "noch möglich sind".

Als Beispiel nehmen wir auch hier wieder die Grammatik G35 (in der schon um R0 ergänzten Form):

R0: S → A
R1: A → A-B
R2: A → B
R3: B → B/C
R4: B → C
R5: C → x

Zum Anfangs-Zustand Z0 gehört dann die REMP `S` → `•A`, da wir ja hoffen, eine Symbolfolge in der Eingabe zu finden, die dem ("neuen") Startsymbol `S` bzw. dem ("alten" Start-) Symbol `A` entspricht.

Mit der REMP `S` → `•A` gehören aber auch alle Regeln mit `A` auf der linken Seite und einem Punkt am Anfang der rechten Seite zu Z0, also `A` → `•A-B` und `A` → `•B`.

Wegen `A` → `•B` müssen wir jetzt auch alle Regeln mit B auf der linken Seite (und einem Punkt am Anfang der rechten Seite) zu Z0 hinzunehmen, also `B` → `•B/C` und `B` → `•C`.

Wegen `B` → `•C` kommt schließlich noch `C` → `•x` hinzu.

Insgesamt besteht der Anfangs-Zustand Z0 aus folgenden REMPs:

Z0 = {`S` → `•A`, `A` → `•A-B`, `A` → `•B`, `B` → `•B/C`, `B` → `•C`, `C` → `•x` }

Anschaulich (hoffentlich) bedeutet das: am Anfang einer Analyse erwarten wir in der Eingabe eine Symbol-Folge, die der Satzform `A` oder `A-B` oder `B` oder `B/C` oder `C` oder `x` entspricht.

In welcher Reihenfolge wir jetzt weitere Zustände konstruieren, ist eigentlich egal. Eine systematische Reihenfolge erleichtert aber die Übersicht über das Verfahren. Um systematisch vorzugehen, ordnen wir alle Symbole der Grammatik (mit Ausnahme des "neuen, ergänzten " Startsymbols S), z.B. so:

A, B, C, x, -, / und fragen uns jetzt der Reihe nach: erwarten wir in Z0 ein A? Erwarten wir in Z0 ein B? ... Erwarten wir in Z0 ein "/"? Immer wenn die Antwort "ja" lautet, konstruieren wir einen (möglicherweise) neuen Zustand.

Sei Z1 der Zustand, in den der Parser gerät, wenn er im Zustand Z0 war und eine Symbolfolge eingelesen hat, die einem A entspricht. Kurz: Sei Z1 = goto(Z0, A). Dann gehören folgende REMPs zu Z1:

Z1 = {S → A•, A → A•-B}

Diese beiden REMPs entstanden aus den REMPs S → •A und A → •A-B aus der Menge Z0, indem wir den Punkt "•“ "um ein A nach rechts verschoben haben".

Sei entsprechend Z2 = goto(Z0, B), dann gilt:

Z2 = {A → B•, B → B•/C}

Ebenso findet man:

Z3 = goto(Z0, C) = {B → C•}
Z4 = goto(Z0, x) = {C → x•}

Im Zustand Z0 erwarten wir kein "-" und kein "/", d.h. es gilt:

goto(Z0, -) = {}
goto(Z0, /) = {}

Solche leere Mengen zählen nicht als neue Zustände.

Jetzt behandeln wir Z1 genauso, wie vorher Z0, d.h. wir fragen uns: Erwarten wir in Z1 ein A? Oder ein B? Oder ... Oder ein x? Da wir in Z1 nur ein "-" erwarten, gibt es nur einen Folge-Zustand für Z1:

Z5 = goto(Z1, -) = {A → A-•B, ... }

Aber weil die REMP A → A-•B zu Z5 gehört, müssen wir auch hier alle B-Regeln der Grammatik (mit einem Punkt am Anfang der rechten Seite) hinzunehmen:

Z5 = goto(Z1, -) = {A → A-•B, B → •B/C, B → •C, ... }

und wegen der REMP B → •C müssen wir auch alle C-Regeln hinzunehmen (zufällig gibt es nur eine C-Regel):

Z5 = goto(Z1, -) = {A → A-•B, B → •B/C, B → •C, C → •x }

Damit ist der Zustand Z5 fertig konstruiert.

In andere Zustände als Z5 können wir von Z1 aus nicht (direkt) gelangen, d.h. es gilt:

goto(Z1, A) = goto(Z1, B) = goto(Z1, C) = goto(Z1, x) = goto(Z1, /) = {}

Ganz entsprechend finden wir alle anderen Zustände:

Z6 = goto(Z2, /) = {B → B/•C, C → •x}
Z7 = goto(Z5, B) = {A → A-B•, B → B•/C}
Z8 = goto(Z6, C) = {B → B/C•}

Einige Zustände finden wir nicht nur einmal, sondern mehrmals ("auf verschiedenen Wegen"), denn es gilt:

goto(Z5, x) = Z4
goto(Z6, x) = Z4
goto(Z5, C) = Z3
goto(Z7, /) = Z6

Immer, wenn wir einen Zustand konstruiert haben, müssen wir also sorgfältig prüfen, ob er wirklich "neu" ist (und einen neuen Namen verdient), oder ob wir ihn schon früher mal "erfunden haben".

Da eine Grammatik nur endlich viele Regeln enthalten kann, gibt es auch nur endlich viele REMPs und damit endliche viele Mengen von REMPs. Also werden wir immer nur eine endliche Anzahl von Zuständen konstruieren können (und deshalb ist der Automat, den wir konstruieren, ein endlicher Automat).

Aus der Beispiel-Grammatik G35 kann man genau die angegebenen neun Zustände (die wir hier Z0 bis Z8 genannt haben) konstruieren.

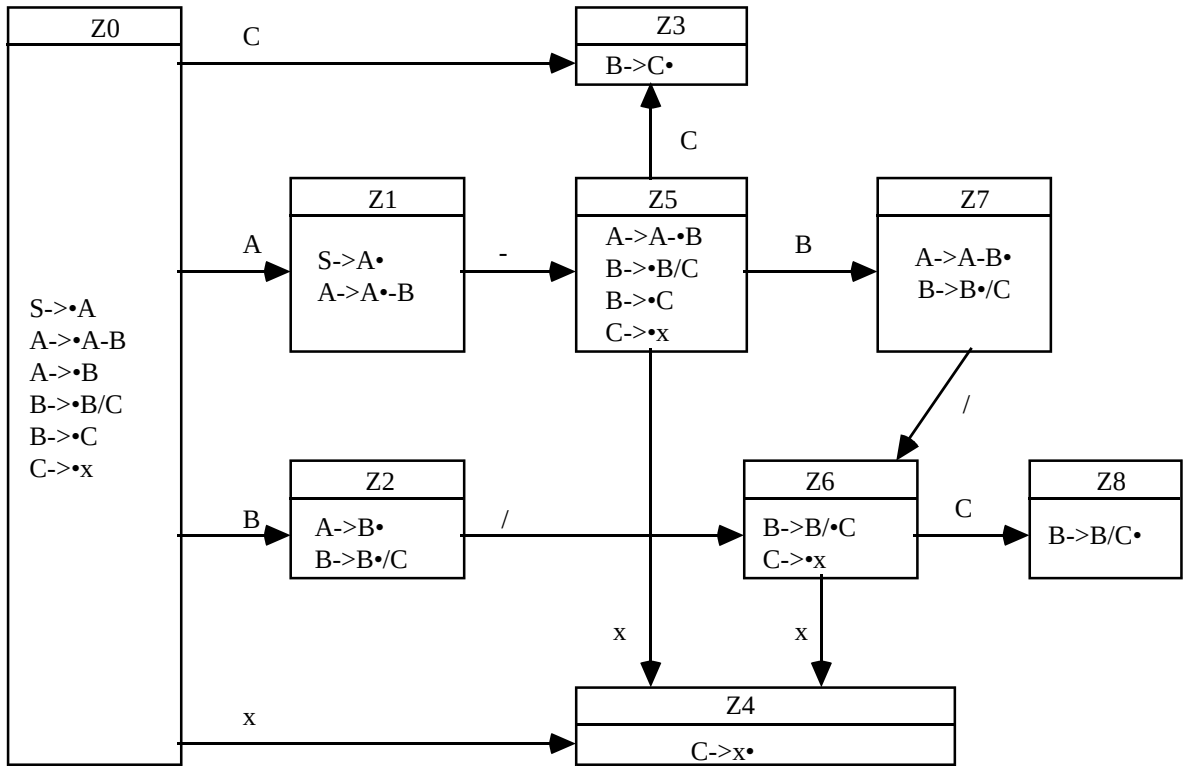
Schritt 2: Konstruktion des endlichen Automaten

Nachdem wir alle Zustände als Mengen von REMPs berechnet haben, können wir einen entsprechenden endlichen Automaten darstellen, z.B. durch ein Diagramm.

Es ist nicht ganz einfach, Automaten-Diagramme so zu zeichnen, dass man sie "überschauen und verstehen" kann. Bei einfachen Automaten genügt es manchmal, wenn man fünf bis zehn Entwürfe zeichnet und dann "den besten" auswählt. Bei komplizierteren Automaten kann es sehr schwer bis unmöglich sein, eine "lesbare" Anordnung der Knoten und Kanten zu finden.

Um eine LR-Parsertabelle zu konstruieren muss man den entsprechenden endlichen Automaten nicht unbedingt graphisch darstellen. Möglicherweise unterstützt aber eine graphische Darstellung des Automaten ein anschauliches Verständnis der Konstruktion und des Parsers.

Hier eine graphische Darstellung des endlichen Automaten mit den oben berechneten neun Zuständen Z0 bis Z8 (für die Grammatik G35):



Schritt 3: Die Folgemengen für alle Zwischen-Symbole der Grammatik:

Wie man die Folgemenge eines Zwischen-Symbols berechnet, wurde im Zusammenhang mit der LL-Syntaxanalyse behandelt (siehe Abschnitt 4.). Für die Zwischen-Symbole A, B und C der Grammatik G35 ergibt sich:

follow(S) = {$} follow(A) = {-, $} follow(B) = {-, /, $} follow(C) = {-, /, $}

Schritt 4: Die Parser-Tabelle "ausfüllen"

Jetzt können wir den endlichen Automaten "in die Parser-Tabelle eintragen". Die noch fast leere Tabelle sieht so aus (die wenigen Eintragungen gehören zu den unten beschriebenen Beispielen):

	Aktion				Folgezustand		
Zustand	x	-	/	$	A	B	C
Z0							
Z1		s5		lächeln			
Z2		r2		r2			
Z3							
Z4							
Z5						Z7	
Z6							
Z7							
Z8		r3	r3	r3			

Für jeden Zustand gibt es eine Zeile, für jedes Symbol der Grammatik gibt es eine Spalte. Die Spalten für die End-Symbole der Grammatik bilden den Aktions-Teil der Tabelle, die Spalten mit den Zwischen-Symbolen bilden den Folgezustands-Teil. Im Aktions-Teil müssen wir Aktionen eintragen (nämlich: schieben, reduzieren, lächeln, Fehlermeldung ausgeben), im Folgezustands-Teil müssen wir (Folge-) Zustände (Z0, Z1, ...) eintragen.

Für den Aktions-Teil gilt folgende Konvention: wo wir keine andere Aktion eintragen, soll die Aktion "Gib eine Fehlermeldung aus und beende die Analyse ohne Erfolg" stehen.

Wir nehmen wir drei Arten von Einträgen vor:

1. Schiebe-Aktionen (im Aktions-Teil der Tabelle).

Wenn im endlichen Automaten ein End-Symbol e an einer Kante von einem Zustand Z_i zu einem Zustand Z_j steht, dann tragen wir die Aktion s_j ein in die Zeile Z_i, Spalte e.

Beispiel: Von Z1 führt eine mit "-" beschriftete Kante nach Z5. Also steht in der Parser-Tabelle in Zeile Z1, Spalte "-" die Aktion s5.

2. Folgezustände (im Folgezustands-Teil der Tabelle):

Wenn im endlichen Automaten ein Zwischen-Symbol Z an einer Kante von einem Zustand Z_i zu einem Zustand Z_j steht, dann tragen wir den Zustand Zj ein in die Zeile Z_i, Spalte Z.

Beispiel: Von Z5 führt eine mit "B" beschriftete Kante nach Z7. Also steht in der Parser-Tabelle in Zeile Z5, Spalte B der Folgezustand Z7.

3. Reduzier-Aktionen (im Aktions-Teil der Tabelle):

Wenn zu einem Zustand Z_i eine REMP gehört, bei der der Punkt ganz rechts steht (wie z.B. bei der REMP $A \rightarrow A\text{-}B\bullet$), dann tragen wir in der Parser-Tabelle in Zeile Z_i eine entsprechende Reduzier-Aktion ein, etwa so:

Beispiel 1: Zum Zustand Z2 gehört unter anderem die REMP $A \rightarrow B\bullet$. Das ist die Regel R2 (mit einem Punkt ganz rechts). Auf der linken Seite dieser Regel steht ein "A". Die Folgemenge von A ist gleich {-, $}. Also tragen wir in Zeile Z2 in den Spalten "$" und "-" die Aktion r2 ("reduziere nach Regel R2") ein.

Beispiel 2: Zum Zustand Z8 gehört die REMP $B \rightarrow B/C\bullet$. Das ist B-Regel R3 (mit einem Punkt ganz rechts). Es gilt follow(B) = {-, /, $}. Also tragen wir in Zeile Z8 in die Spalten "-", "/" und "$" die Aktion r3 ein.

Beispiel 3: Eine Ausnahme bildet die REMP $S \rightarrow A\bullet$, die aus der neuen ("ergänzten") Startregel R0 besteht. Da sie zum Zustand Z1 gehört, tragen wir in Zeile Z1, Spalte "$" unsere Lieblings-Aktion "lächeln" ein.

Aufgabe 5.3.1.: Füllen Sie die obige Parser-Tabelle vollständig aus und vergleichen Sie Ihr Ergebnis mit der Tabelle im vorigen Abschnitt 5.2.

Aufgabe 5.3.2.: Konstruieren Sie eine LR-Parser-Tabelle für die folgende Grammatik. Gehen Sie dabei von folgender Reihenfolge der Symbole aus: A, B, x, +.

R1: A → A+B
R2: A → A -B
R3: A → B

R4: B → x

Vergessen Sie nicht, die Grammatik zuerst einmal zu ergänzen (neues Startsymbol S und neue Regel R0).

Aufgabe 5.3.3.: Analysieren Sie mit Ihrer Tabelle aus der vorigen Aufgabe (und ganz gefühllos) das Wort `x-x+x`.

Aufgabe 5.3.4.: Konstruieren Sie für die folgende Grammatik eine LR-Parser-Tabelle.

Reihenfolge der Symbole: A, B, C, D, x, +, ^.

R1: A → A+B
R2: A → B
R3: B → B*C
R4 B → C
R5: C → C^D
R6: C → D
R7: D → x

Wer Lust hat, kann sich unter "^" einen Potenz-Operator vorstellen. Aber Vorsicht: dieser Operator ist hier links-assoziativ (und nicht wie üblich rechts-assoziativ). Auch diese Grammatik muss zuerst einmal ergänzt werden.

Aufgabe 5.3.5.: Analysieren Sie mit Ihrer Tabelle aus der vorigen Aufgabe das Wort `x*x` und das Wort `x*x+x^x`.

5.4. S/R- und R/R-Konflikte

Die im vorigen Abschnitt beschriebene Methode zur Konstruktion einer Parser-Tabelle führt nicht immer zum Ziel. Bei bestimmten Grammatiken kommt es beim vierten Schritt ("die Parser-Tabelle ausfüllen") zu so genannten Konflikten. Ein solcher Konflikt liegt vor, wenn man an einer Stelle der Tabelle eigentlich zwei (oder mehr) verschiedene Aktionen eintragen müsste. Wenn man an dieselbe Stelle eine Schiebe-Aktion und eine Reduzier-Aktion eintragen müsste, dann spricht man von einem S/R-Konflikt (Schiebe-Reduzier-Konflikt, shift/reduce conflict). Müsste man zwei verschiedene Reduzier-Aktionen an dieselbe Stelle eintragen, dann liegt ein R/R-Konflikt (Reduzier-Reduzier-Konflikt, reduce/reduce conflict) vor.

5.4.1. Ein S/R-Konflikt

Betrachten Sie bitte die folgende Grammatik:

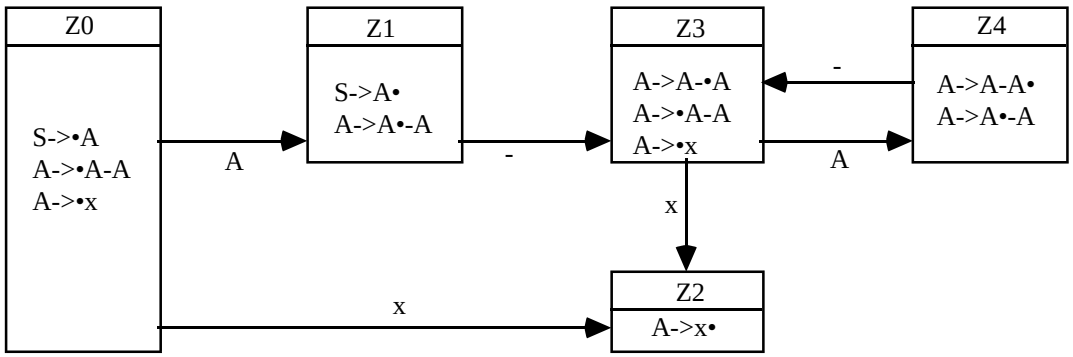
R1: A → A-A
R2: A → x

Offenbar ist diese Grammatik mehrdeutig. Sie legt nicht fest, ob die Minusoperation linksassoziativ oder rechtsassoziativ ist. Wir ergänzen diese Grammatik durch ein neues Startsymbol S und die Regel R0: S → A und konstruieren nach der im vorigen Abschnitt beschriebenen Methode eine Parser-Tabelle:

1. Die Zustände:

Z0 = {S->•A, A->•A-A, A->•x}
goto(Z0, A) = Z1 = {S->A•, A->A•-A}
goto(Z0, x) = Z2 = {A->x•}
goto(Z1, -) = Z3 = {A->A-•A, A->•A-A, A->•x}
goto(Z3, A) = Z4 = {A->A-A•, A->A•-A}
goto(Z3, x) = Z2
goto(Z4, -) = Z3

2. Der endliche Automat:



3. Die Folgemengen: follow(A) = {-, $}

4. Die Parser-Tabelle:

	Aktionen			Folge-Zustand
Zustand	x	-	$	A
Z0	s2			Z1
Z1		s3	lächeln	
Z2		r2	r2	
Z3	s2			Z4
Z4		s3 oder r1?	r1	

In Zeile Z4, Spalte "-" tritt ein S/R-Konflikt auf.

Aufgabe 5.4.1.1.: Entscheiden Sie den obigen S/R-Konflikt zugunsten von s3 und analysieren Sie dann das Wort x-x-x. Welche Ableitung wird gefunden? Ist das "die richtige" Ableitung? "Wie assoziativ" ist das Minuszeichen "-" bei dieser Konfliktlösung, linksassoziativ oder rechtsassoziativ?

Aufgabe 5.4.1.2.: Entscheiden Sie den obigen S/R-Konflikt zugunsten von r1 und analysieren Sie dann das Wort x-x-x. Welche Ableitung wird gefunden? Ist das "die richtige" Ableitung? "Wie assoziativ" ist das Minuszeichen "-" bei dieser Konfliktlösung, linksassoziativ oder rechtsassoziativ?

Aufgabe 5.4.1.3.: Konstruieren Sie zu folgender Grammatik eine Parser-Tabelle. Reihenfolge der Symbole:

ANW, anw, aus, else, if, then.

R1: ANW → anw
R2: ANW → if aus then ANW
R3: ANW → if aus then ANW else ANW

(ANW ist das einzige Zwischensymbol dieser Grammatik, anw, if, aus, then und else sind die Endsymbole). Bei der Konstruktion der Parser-Tabelle wird ein S/R-Konflikt auftreten. Wie sollte man ihn auflösen, d.h. sollte man sich für Schieben oder für Reduzieren entscheiden? Besonders interessant ist die Analyse des Wortes "if aus then if aus then anw else anw". Zu welchem "if" gehört das "else"?

Vereinfachung der Aufgabe: Man kann die Folge von Endsymbolen if aus then (in Regel R2 und R3) zu einem einzigen Endsymbol if_aus_then zusammenfassen.

5.4.2. Ein R/R-Konflikt

Betrachten Sie bitte die folgende Grammatik:

R1: A → B
R2: A → C
R3: B → x
R4: C → x

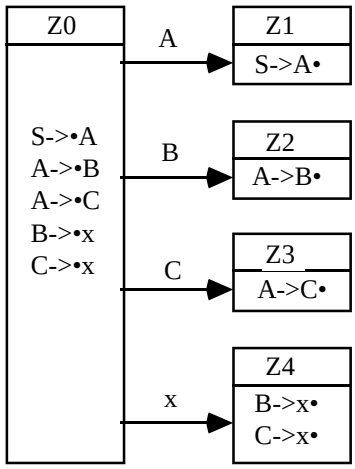
Aus dieser Grammatik kann man nur ein Wort ableiten und dieses Wort besteht nur aus dem einen Endsymbol x. Aber dieses Wort kann man auf zwei Weisen ableiten: "über B" und "über C".

Wir ergänzen diese Grammatik durch ein neues Startsymbol S und die Regel R0: S → A und konstruieren nach der im vorigen Abschnitt beschriebenen Methode eine Parser-Tabelle:

1. Die Zustände:

Z0 = {S->•A, A->•B, A->•C, B->•x, C->•x}
goto(Z0, A) = Z1 = {S->A•}
goto(Z0, B) = Z2 = {A->B•}
goto(Z0, C) = Z3 = {A->C•}
goto(Z0, x) = Z4 = {B-> x•, C-> x•}

2. Der endliche Automat:



3. Die Folgemengen: follow(A) = follow(B) = follow(C) = {$}

4. Die Parser-Tabelle:

	Aktionen		Folgezustände		
Zustand	x	$	A	B	C
Z0	s4		Z1	Z2	Z3
Z1		lächeln			
Z2		r1			
Z3		r2			
Z4		r3 oder r4?			

In Zeile Z4, Spalte "$" tritt ein R/R-Konflikt auf.

Häufig ist es möglich, solche Konflikte aufzulösen, indem man eine der in Konflikt stehenden Aktionen auswählt. In dem trivialen Beispiel könnte man sich z.B. für die Aktion r4 entscheiden. Bei komplizierteren Grammatiken kann es schwierig sein, "die richtige Wahl" zu treffen. Bei einigen Grammatiken ist es nicht möglich, eine "richtige Wahl" zu treffen: egal, wie man sich entscheidet: der Parser macht in einigen Fällen etwas anderes, als man eigentlich möchte.

Abschließende Bemerkungen zu LR-Parsern: Hier wurde nur die grundlegende Methode beschrieben, mit der man LR-Parser-Tabellen konstruieren kann. Der entsprechende Parser hat jeweils nur das nächste Zeichen in der Eingabe benützt, um den "richtigen" Tabellen-Eintrag zu finden. Eine Grammatik hat die LR(1)-Eigenschaft (oder kurz: die Grammatik ist LR(1)), wenn man nach dieser einfachen Methode eine LR-Parser-Tabelle konstruieren kann (ohne dass Konflikte auftreten). Wollte man mit der einfachen LR(1)-Methode eine Parser-Tabelle für eine "mittelgroße Sprache" wie Pascal konstruieren, so müsste man dabei mehrere Tausend Zustände konstruieren.

Für die Praxis wurden verbesserte Methoden zur Konstruktion von LR-Parser-Tabellen entwickelt, z.B. die Methode SLR (simple LR) und LALR (lookahead LR). Ein lookahead LR-Parser betrachtet jeweils nicht nur das nächste Symbol der Eingabe, sondern schaut mehrere Zeichen "voraus". Man spricht dann auch von LR(k)-Parsern, wobei k die Anzahl der Symbole ist, die der Parser jeweils vorausschaut. Für eine Sprache wie Pascal liefern diese Methoden "nur" ein paar Hundert Zustände.

Offenbar ist das Erstellen einer LR-Parser-Tabellen von Hand sehr mühsam und fehleranfällig, auch wenn die Tabelle nur wenige hundert Zustände und damit Zeilen umfasst. Man hat deshalb Programme entwickelt, die aus einer Grammatik automatisch eine LR-Parser-Tabelle bzw. einen vollständigen LR-Parser erzeugen. Besonders berühmt ist das Programm YACC, welches bisher mit jedem UNIX-Betriebssystem mitgeliefert und für die Konstruktion vieler Compiler verwendet wurde. YACC ist eine Abkürzung für "yet another compiler-compiler", das heißt etwa soviel wie "ach Gott, noch so ein Compiler zum Erzeugen von Compilern". Gleichzeitig erinnert der Name YACC an tibetanische Lamghaarbüffel. BISON ist eine Version des YACC, die auf PCs unter MS-DOS läuft. Dem YACC bzw. BISON gibt man (etwas vereinfacht gesagt) eine Typ-2-Grammatik ein und erhält als Ausgabe einen LR-Parser in Form eines C-Programms.

6. Lexikalische Analyse

6.1. Motivation

Genau betrachtet besteht ein Programm aus Zeichen (Buchstaben, Ziffern, Sonderzeichen). Bei der Syntaxanalyse ist es nicht günstig, so genau zu betrachten. Besser ist es, bestimmte Symbolketten als "Einheiten" zu behandlen und davon abzusehen, aus wievielen und welchen Zeichen diese Einheiten bestehen.

Genauer: bei der kontextfreien Syntaxanalyse kann und sollte man z.B. von folgenden Einzelheiten abstrahieren:

1. Wird die Gleichheitsoperation durch "=", ".EQ.", "EQUAL" oder sonst wie dargestellt.

2. Da für alle Relations-Operationen (=, <, ?, ? etc.) die gleichen syntaktischen Regeln gelten ("wo eine Relations-Operation stehen darf, da dürfen auch alle anderen stehen") kann man sie alle durch ein einheitliches so genanntes Token ersetzen, z.B. durch das Token RO.

3. Aus ähnlichen Gründen kann man alle Bezeichner (identifier) z.B. durch das Token ID ersetzen bevor man mit dem Parsieren beginnt.

4. Schlüsselworte verschiedener Länge kann man durch Token einheitlicher Länge ersetzen, z.B. alle "then" durch TH, alle "while" durch WH etc.

Deshalb ist es üblich, in Compilern die Syntaxanalyse in mehrere so genannte Phasen aufzuteilen:

1. Lexikalische Analyse

2. Kontextfreie Analyse

3. Kontextabhängige Analyse

Die lexikalische Analyse liest eine Zeichenfolge ein und gibt eine Tokenfolge aus. Jedes Token "steht für" eine Folge von einem oder mehreren Zeichen. Verschiedene Zeichenfolgen können durch gleiche Token repräsentiert werden. Mit Hilfe einer guten lexikalischen Analyse kann man die kontextfreie Syntaxanalyse erheblich vereinfachen und beschleunigen.

6.2. Muster, Lexem, Token, Definitionen

Muster: Beschreibung einer Menge von Zeichenketten, z.B. der Menge aller Bezeichner ("Bezeichner-Muster"), der Menge aller Relations-Operatoren ("Relop-Muster"). Auch wenn die beschriebene Menge nur eine einzige Zeichenkette enthält, ist es wichtig, zwischen dem Muster und der Menge von Zeichenketten zu unterscheiden. Beispiele: das then-Muster beschreibt die Menge {then}, das Klammer-zu-Muster beschreibt die Menge {)} etc.

Muster können durch natürliche Sprache oder durch spezielle Formalismen (z.B. durch Typ 3 Grammatiken oder durch so genannte reguläre Ausdrücke) beschrieben werden.

Ein Muster beschreibt all die Lexeme, die bei der lexikalischen Analyse in das gleiche Token übersetzt werden sollen.

Lexem: "Kleinste bedeutungstragende Einheit". Eine (typischerweise nicht sehr lange) Zeichenkette, die Teil der Eingabe einer lexikalischen Analyse ist und die durch ein Muster beschrieben wird.

Beispiele: "OTTO" (ein Bezeichner), "<=" (ein Relations-Operator), "THEN" (das Schlüsselwort then). Wenn "OTTO" ein Lexem ist, dann ist der Anfang "OTT" dieser Symbolkette kein Lexem.

Token: Ein Symbol, das bei der lexikalischen Analyse als Repräsentant für ein Lexem ausgegeben wird.

Beispiele: ID (Token für Lexeme wie OTTO, X, ZAEHLER7 etc.), RO (Token für Lexeme wie =, <, <= etc.), TH (Token für then-Lexeme).

Wenn ein Token für verschiedene Lexeme stehen kann (z.B. das Token ID), dann werden die Lexeme bei der lexikalischen Analyse in einer so genannten Symboltabelle abgelegt und das Token um einen Zeiger (pointer) in diese Symboltabelle ergänzt.

Beispiel: Das Token ID17 repräsentiert einen Bezeichner (identifier). Nähere Angaben zu diesem Bezeichner stehen im 17. Eintrag der Symboltabelle.

Merke: Bei der kontextfreien Syntax-Analyse spielen Token (und nicht etwa einzelne Zeichen wie Buchstaben, Ziffern etc.) die Rolle der End-Symbole.

6.3. Endliche Automaten für die lexikalische Analyse

Die lexikalische Analyse-Phase eines Compilers braucht nicht mehr "von Hand" programmiert zu werden. Eine bessere Methode besteht darin, diesen Teil eines Compilers als so genannten endlichen Automaten darzustellen und diesen dann von Hand oder per Programm in ein lauffähiges Programm umzuwandeln.

Definition: Endlicher Automat

Ein endlicher Automat EA = (E, A, Z, f, g) besteht aus
1. einer (nicht-leeren, endlichen) Menge E von Eingaben ("Eingabezeichen")
2. einer (möglicherweise leeren, endlichen) Menge A von Ausgaben ("Ausgabe-Token")
3. einer (nicht-leeren, endlichen) Menge Z von Zuständen
4. einer so genannten (Zustands-) Übergangs-Funktion f: Z, E $\rightarrow$ Z
5. einer (partiellen) Ausgabe-Funktion g: Z, E $\rightarrow$ A

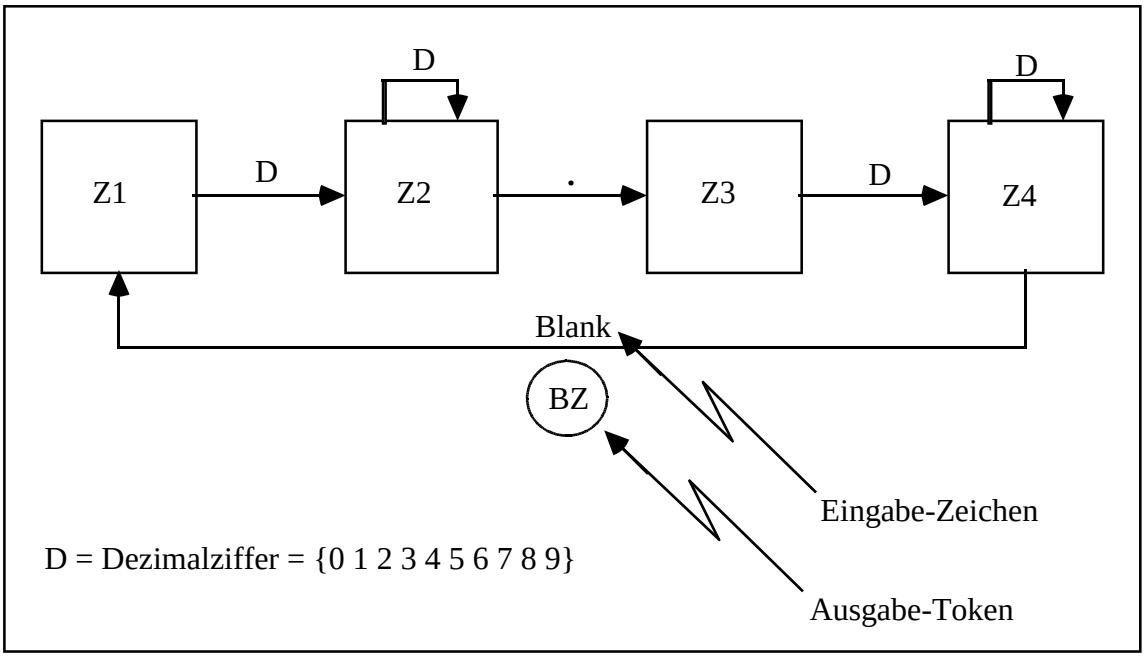
6.4. Einfaches Beispiel für einen lexikalischen Analysator

Ein endlicher Automat EA zum Erkennen von dezimalen Bruchzahlen.

Eine dezimale Bruchzahl soll dabei aus zwei nicht-leeren Folgen von Dezimalziffern bestehen, zwischen denen ein Punkt steht. Abgeschlossen wird eine dezimale Bruchzahl durch ein Blank.

Beispiele für dezimale Bruchzahlen: 123.456 1.23 0.9 9.0 0.0 000.000

Eingabe-Zeichen	E = {0 1 2 3 4 5 6 7 8 9 0 . Blank}
Ausgabe-Token	A = {BZ} -- Nur ein einziges Token
Zustände	Z = {Z1, Z2, Z3, Z4}



Aufgabe 6.4.1.: Entwerfen Sie einen endlichen Automaten zum Erkennen von Bezeichnern (identifier) ihrer Lieblings-Programmiersprache (Achtung: C, Pascal und Ada haben verschiedene Regeln für ihre Bezeichner!).

6.5. Umfangreicheres Beispiel

Hier eine Tabelle mit Mustern, Lexemen und Token:

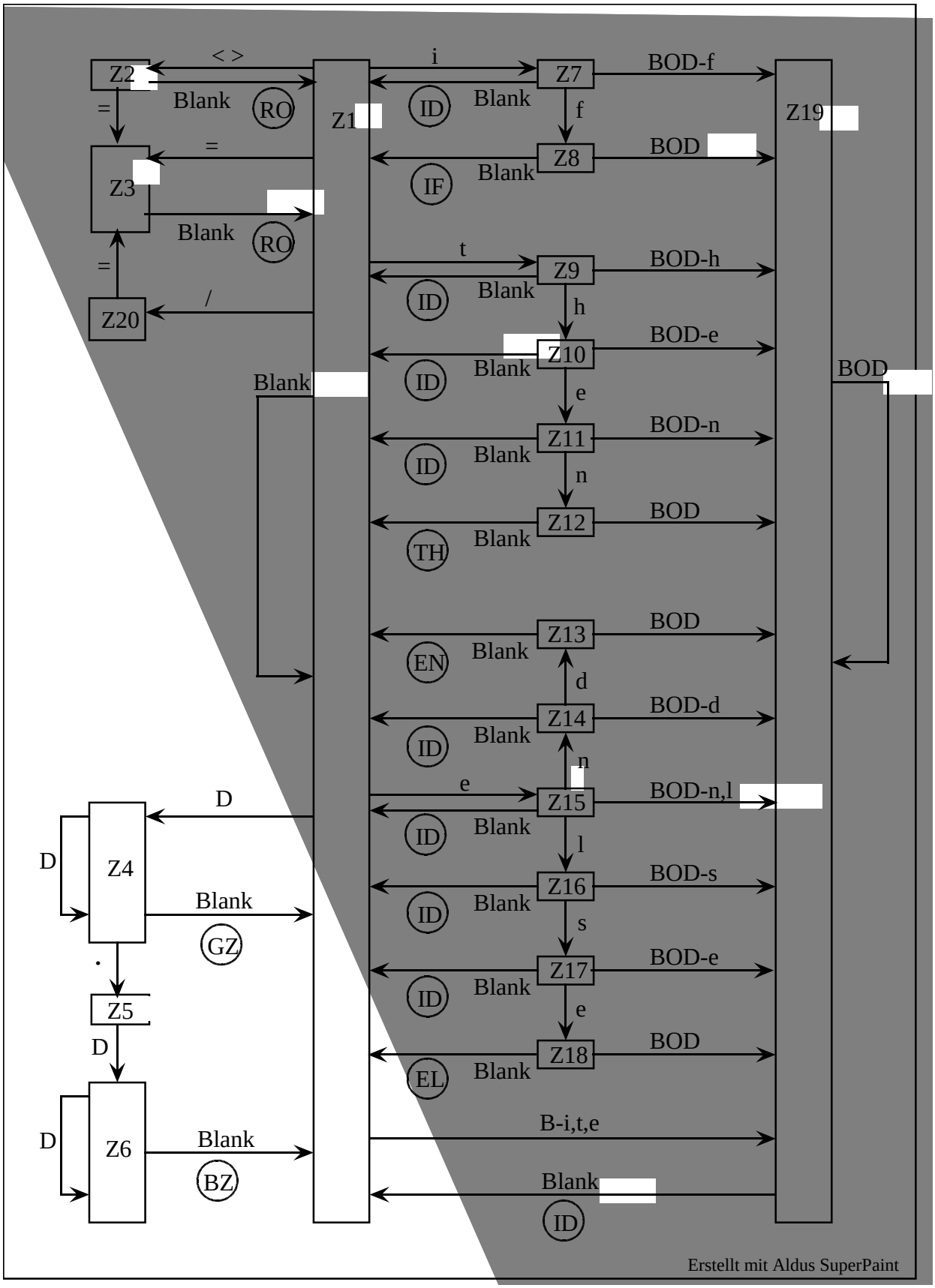
Muster	Beispiel-Lexeme auf die das Muster paßt	Token für diese Lexeme
1. Bezeichner (identifier): nicht-leere Folge von Buchstaben und/oder Ziffern, die mit einem Buchstaben beginnt.	a a007 otto summe5	ID
2. Relations-Operator: = oder <= oder >= oder < oder > oder /=	= <= >= < > /=	RO
3. Ganzzahlliteral: nicht-leere Folge von dezimalen Ziffern	0 9 123 987654	GZ
4. Bruchzahlliteral: zwei Ganzzahlen mit einem Punkt dazwischen	123.456 1.23 0.123 123.0 0.0	BZ
5. Schlüsselwort if: if	if	IF
6. Schlüsselwort then: then	then	TH
7. Schlüsselwort else: else	else	EL
8. Schlüsselwort end: end	end	EN

Gesucht ist ein endlicher Automat, der die hier beschriebenen Lexeme in die entsprechenden Token übersetzen kann. Dieser endliche Automat EA = (E, A, Z, f, g) umfaßt folgende Mengen:

E = {a b c ... z 1 2 3 ... 9 . Blank} -- Eingabezeichen
A = {ID RO GZ BZ IF TH EL EN} -- Ausgabe-Token
Z = {Z1 Z2 Z3 ... Z19} -- Zustände des Automaten, Z1 ist der Anfangszustand

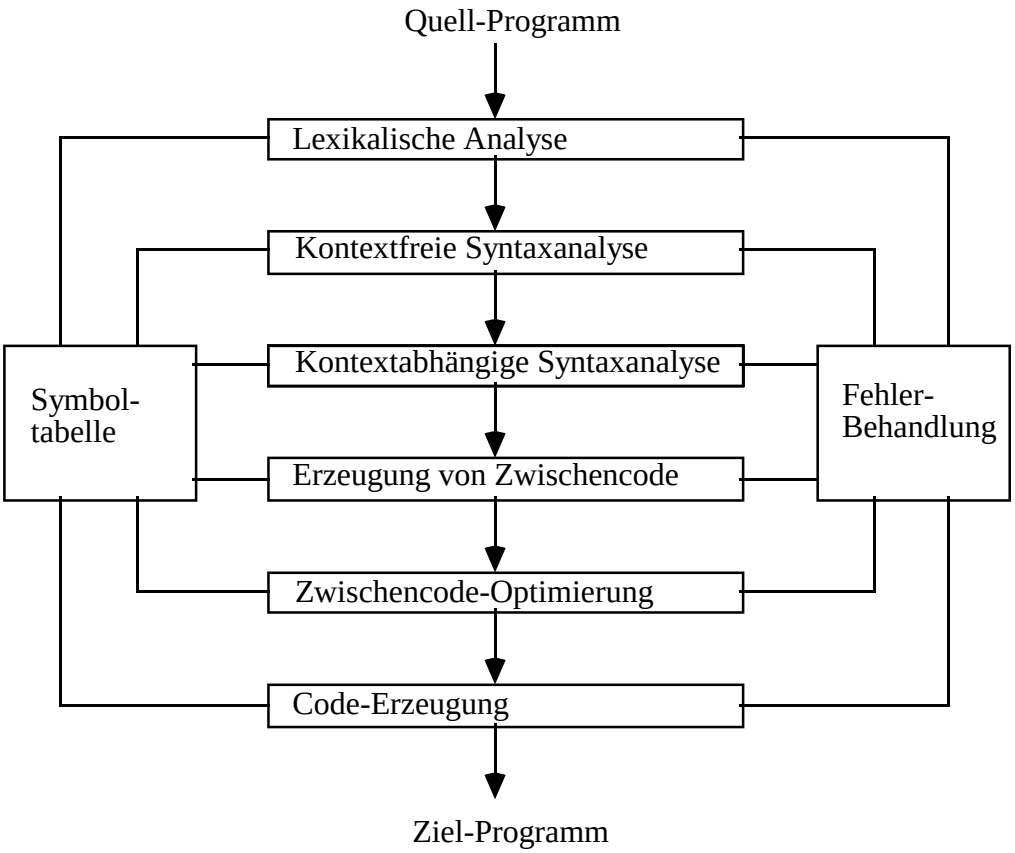
Die Funktionen f und g werden durch das nachfolgende Diagramm beschrieben (siehe nächste Seite). In diesem Diagramm werden folgende Abkürzungen verwendet:

D	für Dezimalziffer	{0 1 2 3 ... 9}
B	für Buchstabe	{a b c ... z}
B-i,t,e	für B \ {i t e}	
BOD	für Buchstabe oder Dezimalziffer	{a b c ... z 0 1 2 ... 9}
BOD-f	für BOD \ {f}	{a b c d e g ... z 0 1 2 ... 9}


Z1
Z2
Z3
Z4
Z5
Z6
Z7
Z8
Z9
Z10
Z11
Z12
Z13
Z14
Z15
Z16
Z17
Z18
Z19
Z20
< >
=
Blank
RO
i
ID
f
BOD-f
BOD
IF
t
h
BOD-h
e
BOD-e
n
BOD-n
TH
EN
d
BOD-d
BOD-n,l
l
BOD-s
s
BOD-e
EL
B-i,t,e
/
D
.
GZ
BZ
Erstellt mit Aldus SuperPaint

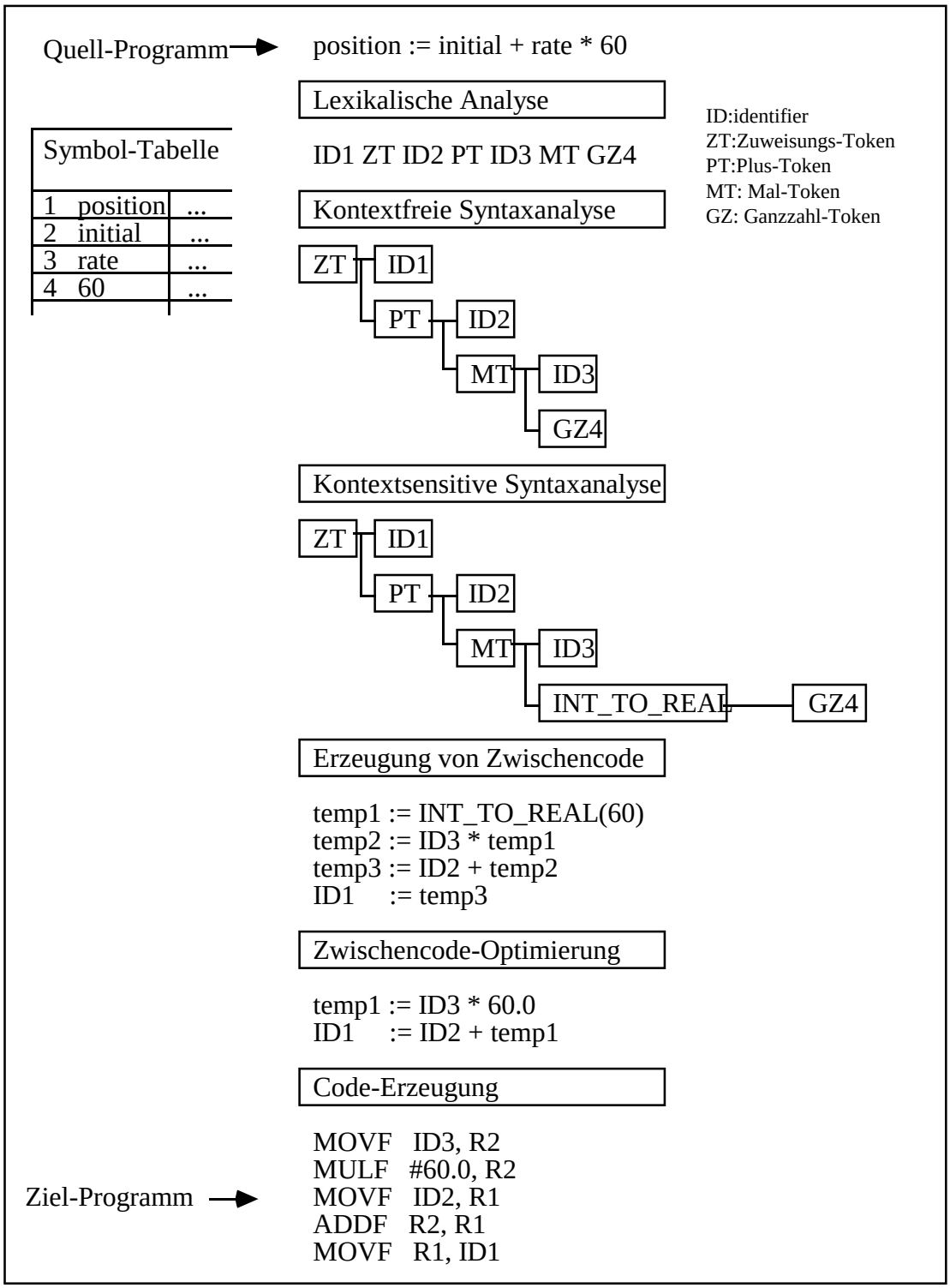
7. Die Phasen ("Teile") eines Compilers

(nach Aho, Sethi, Ullman "Compilers", d.h. nach dem "Drachenbuch"):



Häufig sind mehrere Phasen zu einem so genannten Durchgang (englisch: pass) zusammen gefasst.

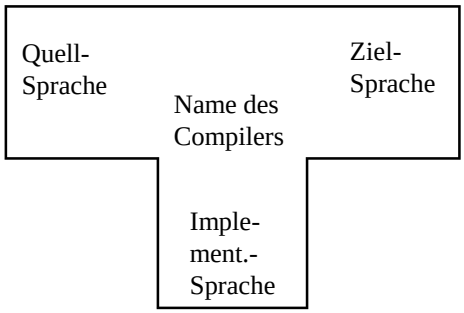
Übersetzung eines "Programms" durch die einzelnen Phasen eines Compilers:



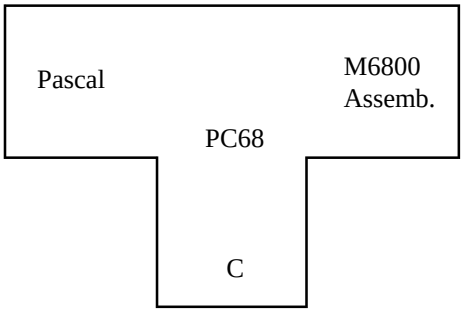
8. Wie man einen Compiler baut (bootstrapping, T-Diagramme)

Am Anfang dieses Skripts wurde anhand eines Diagramms gezeigt, dass ein Compiler durch 2 Programmier-Sprachen gekennzeichnet ist: durch seine Quell-Sprache und seine Ziel-Sprache. Tatsächlich ist noch eine dritte Programmier-Sprache wichtig: die Sprache, in der der Compiler selbst geschrieben ist. Wir nennen diese Sprache hier die Implementierungs-Sprache des Compilers.

Ein T-Diagramm ist ein Diagramm, in dem die 3 für einen Compiler wichtigen Sprachen dargestellt werden. Hier die allgemeine Form eines T-Diagramms:



Hier ein konkretes Beispiel: das T-Diagramm für den Compiler PC68, der Pascal-Programme in Assembler-Programme für den Prozessor M68000 übersetzt und selbst in der Sprache C geschrieben ist:



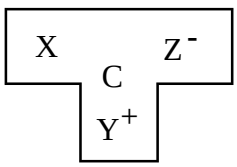
Solche T-Diagramme sind nützlich, wenn man beschreiben will, in welchen Schritten ein Compiler geschrieben und erzeugt wird, insbesondere wenn man dabei ein so genanntes bootstrap-Verfahren ("Schnürsenkel-Verfahren") anwendet.

Bevor dieses bootstrap-Verfahren beschrieben wird, müssen zwei wichtige Eigenschaften eines Compilers erwähnt werden, die man auf keinen Fall durcheinander bringen oder miteinander verwechseln sollte:

Eigenschaft 1: Wie schnell/langsam ist der Compiler selbst und wie viel Speicherplatz benötigt er?

Eigenschaft 2: Wie gut sind die von dem Compiler erzeugten (Ziel-) Programme, d.h. wie schnell/langsam sind sie und wieviel Speicherplatz benötigen sie?

In einem T-Diagramm notieren wir an der entsprechenden Stelle ein Plus-Zeichen bzw. ein Minus-Zeichen, wenn ein Compiler bzüglich dieser Eigenschaften "gut" bzw. "schlecht" ist. Zum Beispiel beschreibt das folgende T-Diagramm

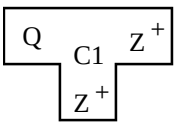


einen Compiler C, der auf einr Y-Maschine läuft und X-Programme in Z-Programme umwandelt. Bezüglich der Eigenschaft 1 ist er gut (Plus-Zeichen am Y), bezüglich der Eigenschaft 2 ist er schlecht (Minus-Zeichen am Z).

Am besten wäre ein Compiler, der bezüglich beider Eigenschaften "sehr gut" ist. Man sollte aber von folgender Grundannahme ausgehen: je bessere Programme ein Compiler erzeugt, desto mehr Zeit und Speicherplatz wird er vermutlich benötigen. Trotzdem kann man Compiler bauen, die bezüglich beider Eigenschaften "gut" sind.

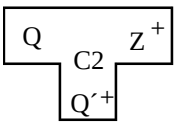
Jetzt können wir uns der Frage zuwenden: wie erstellt man einen Compiler nach dem bootstrap-Verfahren? Die vielleicht erstaunlichste Eigenheit dieses Verfahrens besteht darin, dass man den Compiler "hauptsächlich" in seiner eigenen Quell-Sprache schreibt. Wir beschreiben das Verfahren hier in 6 Schritten:

1. Wir gehen von folgender Situation aus: uns steht eine Maschine zur Verfügung, die in Z geschriebene Programme ausführen kann ("wir haben eine Z-Maschine"). Wir wollen einen Compiler C1 bauen, der selbst in der Sprache Z implementiert ist ("der auf unserer Z-Maschine läuft"). Der Compiler C1 soll Programme der Quell-Sprache Q übersetzen, und zwar in die Sprache Z (damit die Ziel-Programme auch auf unserer Z-Maschine laufen können). Das folgende T-Diagramm beschreibt den Compiler C1, den wir letzten Endes haben möchten:



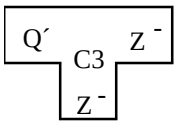
Natürlich wollen wir, dass der Compiler C1 bezüglich der oben erwähnten Eigenschaften 1 und 2 "gut" ist.

2. Wir schreiben "sehr sorgfältig" einen Compiler C2, der bezüglich der oben erwähnten Eigenschaften 1 und 2 "möglichst gut" ist. Dem Compiler C2 entspricht folgendes T-Diagramm:



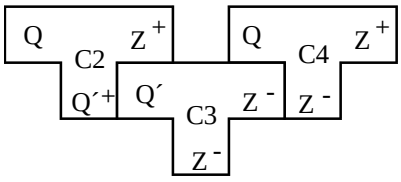
Dabei ist Q´ eine Teilsprache der (Quell-) Sprache Q. Q´ umfasst nur die Befehle und Konstrukte, die man zum Schreiben eines guten Compilers unbedingt braucht. Nicht gebraucht werden z.B.: Gleitpunkt-Arithmetik, Befehle zum Starten und Verwalten von nebenläufigen Prozessen und weitere Konstrukte. Im allgemeinen kann die Sprache Q´ erheblich "kleiner und einfacher" sein als die volle Sprache Q.

3. Wir schreiben "schnell und schmutzig" einen Compiler C3



der bezüglich der Eigenschaften 1 und 2 "sehr schlecht" sein darf (beide Zetts haben ein Minuszeichen!). Allerdings: korrekt muss auch dieses Programm C3 sein.

4. Wir übersetzen unseren "guten Compiler" C2 mit unserem "schlechten Compiler" C3. Das kann man mit Hilfe der T-Diagramme so darstellen:



Man sieht (hoffentlich) leicht, dass in diesem Diagramm die notwendigen "Anschlussbedingungen" erfüllt sind:

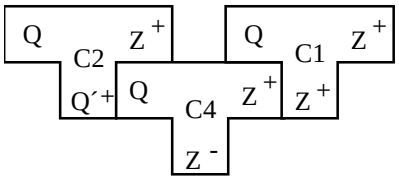
- Die Implementierungs-Sprache von C2 und die Quell-Sprache von C3 stimmen überein (= Q´)
- Die Ziel-Sprache von C4 und die Implementierungs-Sprache von C1 stimmen überein (= Z)

Diese Übersetzung dauert lange und braucht viel Speicherplatz (da C3 bezüglich der Eigenschaft 1 "schlecht" ist). Für das Übersetzungsergebnis, d.h. für den Compiler C4, gilt aber:

4.1. Bezüglich der Eigenschaft 1 ist C4 schlecht (Minuszeichen am "unteren" Z), weil sein Erzeuger, der Compiler C3, bezüglich der Eigenschaft 2 "schlecht" ist.

4.2. Bezüglich der Eigenschaft 2 ist C4 gut (Pluszeichen am Z rechts oben), da wir C2 ja "sorgfältig" geschrieben haben (siehe 2.)

5. Mit dem Compiler C4 übersetzen wir C2 noch einmal. Das Ergebnis dieser Übersetzung nennen wir C1:

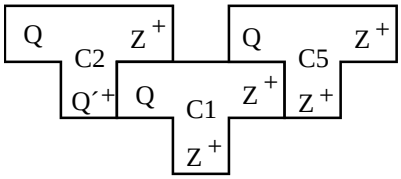


Man sieht leicht, dass in diesem Diagramm die notwendigen "Anschlussbedingungen" erfüllt sind:
- Die Implementierungs-Sprache von C2 (Q´) ist eine Teilmenge der Quell-Sprache von C4 (nämlich Q)
- Die Ziel-Sprache von C4 und die Implementierungs-Sprache von C1 stimmen überein (= Z)

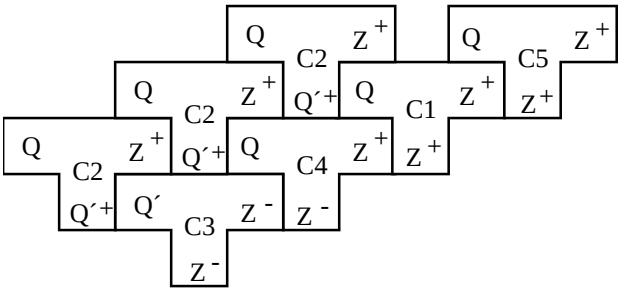
Auch diese Übersetzung dauert lange und braucht viel Speicherplatz, weil der Compiler C4 bezüglich der Eigenschaft 1 schlecht ist (Minuszeichen am "unteren" Z). Aber für das Übersetzungsergebnis, d.h. für den Compiler C1, gilt:
5.1. Bezüglich der Eigenschaft 1 ist C1 gut, weil sein Erzeuger, der Compiler C4, bezüglich der Eigenschaft 2 gut ist.
5.2. Bezüglich der Eigenschaft 2 ist C1 gut, da wir C2 ja "sorgfältig" geschrieben haben (siehe 2.).

Damit haben wir einen einen Compiler C1, der bezüglich beider Eigenschaften gut ist.

7. Der folgende Schritt dient dazu, unseren Compiler C2 (bzw. die daraus per Übersetzung gewonnen Compiler C4 und C1) zu testen. Der Compiler C5 muss exakt („Bit für Bit“) mit C1 übereinstimmen. Können Sie sich klar machen, warum das so sein muss, wenn C2 korrekt programmiert ist?



7. Das folgende Diagramm gibt den ganzen bootstrapping-Prozess noch einmal in kompakter Form wieder:



Aufgabe 8.1.: Was würden Sie tun, um den Compiler C1 von der Z-Maschine auf eine Y-Maschine (d.h. auf eine Maschine, die nur in der Sprache Y geschriebene Programme ausführen kann) zu übertragen? Illustrieren Sie ihren "Portierungsplan" mit Hilfe von T-Diagrammen.

Anhang: Lösungen zu den Aufgaben

Lösung 2.3.1.: Ableitungen aus der Grammatik G1:

	B		B		B
R4:		R3:		R1	
	1B		0B		0
R3:		R3:			
	10B		00B		
R2:		R2:			
	101		001		

Lösung 2.3.2.: Weil es keine Regel gibt, in der das End-Symbol 5 vorkommt.

Lösung 2.3.3.: Die Grammatik G2:

R1:	B	→	1R	R3:	R	→	0R
R2:	R	→	ε	R4:	R	→	1R

(B wie "Binärzahl", R wie "Rest")

Lösung 2.3.4.: Die Grammatik G3:

R1:	B	→	R0	R3:	R	→	R0
R2:	R	→	ε	R4:	R	→	R1

(B wie "Binärzahl", R wie "Rest")

Lösung 2.3.5.: Die Grammatik G4:

R1:	B	→	V . N	R5:	N	→	0
R2:	B	→	V	R6:	N	→	R1
R3:	V	→	0	R7:	R	→	0R
R4:	V	→	1R	R8:	R	→	1R
				R9:	R	→	ε

(B wie "Binär-Bruch", V wie "vor dem Punkt", N wie "nach dem Punkt", R wie "Rest")

Lösung 2.3.6.:

Regel 1, Be geht nach Vau Punkt En
Regel 2, Vau geht nach Null
Regel 3, Vau geht nach Eins Err
...
Regel 8, Err geht nach Epsilon

Lösung 2.3.7.: Die Grammatik G5:

R1:	A	→	A + A	R4:	A	→	x
R2:	A	→	A * A	R5:	A	→	y
R3:	A	→	(A)	R6:	A	→	z

(A wie "Ausdruck)

Lösung 2.5.1.1.: Die Regel R1 ist rechtslinear, R2 ist linkslinear. Eine solche "Mischung" von rechtslinearen und linkslinearen Regeln ist in Typ-3-Grammatiken nicht erlaubt.

Lösung 2.5.1.2.: G8, eine Typ-3-Grammatik für alle in Ada erlaubten Bezeichner. Ein Ada-Bezeichner muss mit einem Buchstaben anfangen. Dananch dürfen beliebig viele (auch null) Buchstaben, Ziffern oder Unterstriche "_" kommen. Unterstriche dürfen aber nicht unmittelbar hintereinander oder am Ende des Bezeichners stehen (d.h. z.B. A__B, AB_ und A__ sind keine gültigen Ada-Bezeichner). In der folgenden Grammatik sind End-Symbole in Anführungsstriche eingefaßt (z.B. "A", "a", "0", "9" oder "_"), um sie von den Zwischen-Symbolen (I, J, K) zu unterscheiden. I ist das Startsymbol der Grammatik.

R1:	I → "a"	R53:	I → "a"J	R105:	J → "a"J	R157:	K → "a"J
R2:	I → "b"	R54:	I → "b"J	R105:	J → "b"J	R157:	K → "b"J
	...		...		...		...
R26:	I → "z"	R78:	I → "z"J	R105:	J → "z"J	R157:	K → "z"J
R27:	I → "A"	R79:	I → "A"J	R105:	J → "A"J	R157:	K → "A"J
R28:	I → "B"	R80:	I → "B"J	R105:	J → "B"J	R157:	K → "B"J
	...		...		...		...
R52:	I → "Z"	R104:	I → "Z"J	R105:	J → "Z"J	R157:	K → "Z"J
				R158:	J → "0"J	R169:	K → "0"J
				R159:	J → "1"J	R170:	K → "1"J
					...		...
				R167:	J → "9"J	R178:	K → "9"J
				R168:	J → "_"K	R179:	J → ε

Lösung 2.5.1.3.: G9, eine Typ-3-Grammatik für die Menge aller Binärzahlen ohne überflüssige Nullen am Anfang oder Ende. Die Grammatik G9 beschreibt die gleiche Sprache wie die Grammatik G4 (siehe Lösung 2.3.5.):

R1:	A → 0	R4:	B → 0	R8:	B → .C
R2:	A → 1	R5:	B → 1	R9:	C → 1
R3:	A → 1B	R6:	B → 0B	R10:	C → 0C
		R7:	B → 1B	R11:	C → 1C

Lösung 2.7.1.: Diese Lösung besteht aus zwei Teillösungen:

Lösung 2.7.1.1: Typ-3-Grammatik für Pascal-Bezeichner: Ein Pascal-Bezeichner muss mit einem Buchstaben beginnen. Dananch dürfen beliebig viele (auch null) Buchstaben, Ziffern und Unterstriche kommen. Unterstriche dürfen auch hintereinander und am Ende des Bezeichners stehen. In der folgenden Grammatik sind Endsymbole in Anführungszeichen eingeschloßen (z.B. so: "a" oder "I"), um sie von dem Zwischensymbol (I) zu unterscheiden:

R1:	I	→	"a"	R53:	I	→	I"a"	R105:	I	→	I"0"
R2:	I	→	"b"	R54:	I	→	I"b"	R106:	I	→	I"1"
...	...	...	...	...	...	...	...	...			
R26:	I	→	"z"	R80:	I	→	I"z"	R114:	I	→	I"9"
R27:	I	→	"A"	R81:	I	→	I"A"	R115:	I	→	I"_"
R28:	I	→	"B"	R82:	I	→	I"B"				
...	...	...	...	...	...	...	...				
R52:	I	→	"Z"	R104:	I	→	I"Z"				

Lösung 2.7.1.2.: Typ-3-Grammatik für Dezimalbrüche

R1:	A	→	0A	R11:	A	→	0.B	R21:	B	→	0B	R31:	B	→	0
R2:	A	→	1A	R12:	A	→	1.B	R22:	B	→	1B	R32:	B	→	1
...	...	...	...	...	...	...	...	...	...	...	...				
R10:	A	→	9A	R20:	A	→	9.B	R30:	B	→	9B	R33:	B	→	9

Lösung 2.7.2.: Diese Lösung besteht aus zwei Teillösungen:

Lösung 2.7.2.1.: Typ-2-Grammatik für ungerade Dezimalzahlen ohne führende Nullen

R1:	A	→	BCD	R6:	D	→	1	R11:	B	→	D	R16:	E	→	B
R2:	A	→	BD	R7:	D	→	3	R12:	B	→	2	R17:	E	→	0
R3:	A	→	D	R8:	D	→	5	R13:	B	→	4				

R4: C → CE R9: D → 7 R14: B → 6
R5: C → E R10: D → 9 R15: B → 8

Lösung 2.7.2.2.: Typ-2-Grammatik für alle 0-1-Ketten mit gleich viel Nullen wie Einsen:

R1: A → 0 A 1 A
R2: A → A 1 A 0
R3: A → ε

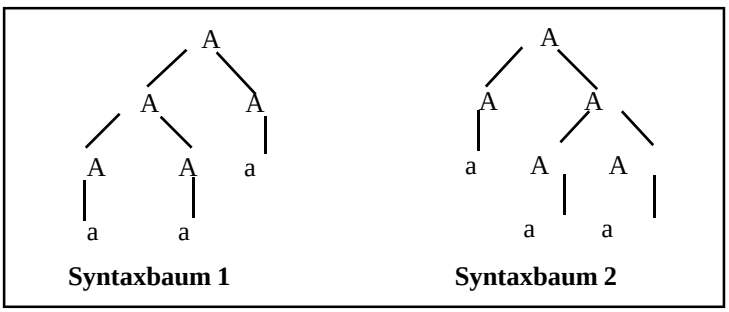
Lösung 2.7.3.: Typ-1-Grammatik für die Sprache $\{a^n b^n c^n \mid n ? 1\}$

R1: S → a S B c
R2: S → abc
R3: bB → bb
R4: cB → Bc

Lösung 2.12. 1.: Der Syntaxbaum 1 ist besser als der Syntaxbaum 3, weil er ausdrückt, dass "Punktrechnung vor Strichrechnung" geht, d.h. dass "*" stärker bindet als "+".

Lösung 2.12.2.: Man bildet zu jeder der beiden Ableitungen den entsprechenden Syntaxbaum und prüft, ob diese beiden Bäume gleich sind (dann sind die Ableitungen nur unwesentlich verschieden) oder ob sie ungleich sind (dann sind die Ableitungen wesentlich verschieden).

Lösung 2.12.3.: Die Grammatik G15 ist mehrdeutig, weil es z.B. für das Wort aaa zwei verschiedene Syntaxbäume gibt:



Lösung 2.12.4.: Jede ableitbare Satzform enthält höchstens ein Zwischen-Symbol, nämlich das Symbol A. Es gibt nur eine Regel für A (d.h. mit A auf der linken Seite). Also gibt es für jede (ableitbare) Satzform nur genau eine Ableitung. Also kann es für keine Satzform zwei (wesentlich verschiedene) Ableitungen geben.

Lösung 2.12.5.: Ein Ableitung aus einer kontextfreien Grammatik (Typ 2 Grammatik) heißt rechtskanonisch, wenn in jeder Zeile (außer der letzten) das jeweils am weitesten rechts stehende Zwischen-Symbol unterstrichen ist.

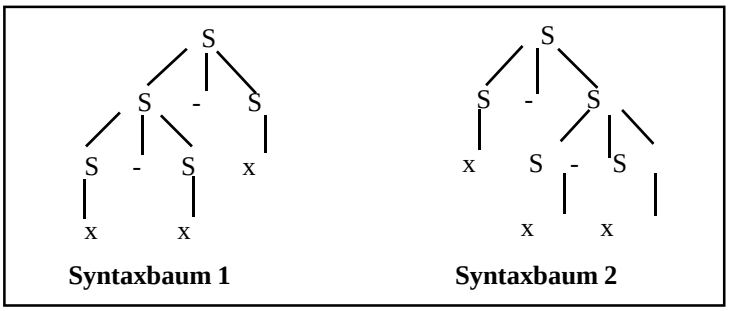
Lösung 2.12.6.: Die Ableitung 1 ist rechtskanonisch, die Ableitungen 2 und 3 sind linkskanonisch.

Lösung 2.12.7.: Eine Ableitung, die weder rechts- noch linkskanonisch ist:

S
R1:
S + S
R2:
S + S * S
R4:
x + S * S
R6:
x + S * z
R5:
x + y * z

Lösung 2.12.8.: Jede Ableitung aus der Grammatik G16 ist gleichzeitig links- und rechtskanonisch.

Lösung 2.12.9: Alle (beide) Syntaxbäume für das Wort "x - x - x" :



Syntaxbaum 1 **Syntaxbaum 2**

Der Syntaxbaum 1 drückt aus, dass das linke (erste) Minuszeichen zuerst "ausgerechnet" wird. Beim Symtaxbaum 2 wird das rechte (zweite) Minuszeichen zuerst ausgewertet.

Lösung 2.12.10: Eine eindeutige Grammatik mit Reihenfolge "von links nach rechts":

R1: $S \rightarrow S - x$
R2: $S \rightarrow x$

Lösung 2.12.11: Eine eindeutige Grammatik mit Reihenfolge "von rechts nach links":

R1: $S \rightarrow x - S$
R2: $S \rightarrow x$

Lösung 2.12.12.: Eine eindeutige Grammatik mit "Punktrechnung geht vor Strichrechnung":

R1: $A \rightarrow A + B$ R3: $B \rightarrow B * C$ R5: $C \rightarrow x$ R7: $C \rightarrow z$
R2: $A \rightarrow B$ R4: $B \rightarrow C$ R6: $C \rightarrow y$ R8: $C \rightarrow (A)$

Lösung 2.13.3.1.: Ersetzen Sie in Lösung 2.12.9. jeweils S durch AUS.

Lösung 2.13.4.1.:
Eine Ableitung für das Wort "x - x - x":
AUS1 R2 AUS1 - AUS2 R2 AUS1 - AUS2 - AUS2 R3 AUS2 - AUS2 - AUS2 R6
AUS3 - AUS2 - AUS2 R8 AUS4 - AUS2 - AUS2 R11 AUS5 - AUS2 - AUS2 R13 x - AUS2 - AUS2 R6
x - AUS3 - AUS2 R8 x - AUS4 - AUS2 R11 x - AUS5 - AUS2 R13 x - x - AUS2 R6
x - x - AUS3 R8 x - x - AUS4 R11 x - x - AUS5 R13 x - x - x

Eine Ableitung für das Wort "x ** x ** x":
AUS1 R3 AUS2 R6 AUS3 R7 AUS4 ** AUS3 R7 AUS4 ** AUS4 ** AUS3 R8 AUS4 ** AUS4 ** AUS4 R11
AUS5 ** AUS4 ** AUS4 R11 AUS5 ** AUS5 ** AUS4 R11 AUS5 ** AUS5 ** AUS5 R13
x ** AUS5 ** AUS5 R13 x ** x ** AUS5 R13 x ** x ** x

Eine Ableitung für das Wort "x + x * x":
AUS1 R1 AUS1 + AUS2 R3 AUS2 + AUS2 R6 AUS3 + AUS2 R8 AUS4 + AUS2 R11
AUS5 + AUS2 R13 x + AUS2 R4 x + AUS2 * AUS3 R6 x + AUS3 * AUS3 R8 x + AUS4 + AUS3 R8
x + AUS4 * AUS4 R11 x + AUS5 * AUS4 R11 x + AUS5 * AUS5 R13 x + x * AUS5 R13 x + x * x

etc. etc.

Lösung 2.13.4.2.: Das Wort (-x + - x) kann man ableiten, das Wort (+ - x + x) kann man nicht ableiten.

Lösung 2.13.4.3.: Tja, wie macht man das wohl am geschicktesten?

Lösung 2.13.5.1.1.: Ja
Lösung 2.13.5.1.2.: Ja, z.B. dass der Potenz-Operator $\uparrow$ linksasassoziativ ist.
Lösung 2.13.5.1.3.: Vermutlich eindeutig

Lösung 2.13.5.2.1.: Ein SetConstructor kann null ElementDescription´s enthalten. Eine ElementDescription besteht aus einem OrdinalExpression oder aus zwei OrdinalExpression (die durch ".." voneinander getrennt sind).
Lösung 2.13.5.2.2.: Ja
Lösung 2.13.5.2.3.: Nein
Lösung 2.13.5.2.4.: Mehrdeutig, z.B. wegen Regel R1.

Lösung 2.13.5.3.1.: Ja
Lösung 2.13.5.3.2.: Nein
Lösung 2.13.5.3.3.: Mehrdeutig, z.B. wegen Regel R1 oder R6.

Lösung 2.13.5.4.1.: Ja
Lösung 2.13.5.4.2.: Ja
Lösung 2.13.5.4.3.: Vermutlich eindeutig.

Lösung 3.1.1.: Das Wort 123 wird abgeleitet und übersetzt:

Zeilen-Nr.	Ableitung	Übersetzung
1	Z:1	Z:1.ü := Z:2.ü * 1010 + D:1.ü
2	R1:	:= 1100 * 1010 + 11
3		:= 1111011
4	Z:2 D:1	Z:2.ü := Z:3.ü * 1010 + D:2.ü
5	R1:	:= 1 * 1010 + 10
6		:= 1100
7	Z:3 D:2 D:1	Z:3.ü := D:3.ü
8	R3	:= 1
9	D:3 D:2 D:1	D:3.ü := 1
10	R4:	
11	1 D:2 D:1	D:2.ü := 10
12	R5:	
13	1 2 D:1	D:1.ü := 11
14	R6:	
15	1 2 3	

Lösung 3.1.2.: Ein Schema für die Übersetzung von (ganzen) Binärzahlen in (ganze) Dezimalzahlen:

Syntaktische Regeln	Semantische Regeln
R1: B.0 → B.1 1	B.0.ü := B.1.ü * 2 + 1
R2: B.0 → B.1 0	B.0.ü := B.1.ü * 2
R3: B.0 → 1	B.0.ü := 1
R4: B.0 → 0	B.0.ü := 0

Lösung 3.1.3.: Ein Schema für die Übersetzung von Binärbrüchen in Dezimalbrüche:

Syntaktische Regeln	Semantische Regeln
R1: B.0 → V.1 . N.1	B.0.ü := V.1.ü + N.1.ü
R2: V.0 → V.1 1	V.0.ü := V.1.ü * 2 + 1
R2: V.0 → V.1 0	V.0.ü := V.1.ü * 2 + 0
R3: V.0 → 1	V.0.ü := 1
R4: V.0 → 0	V.0.ü := 0
R2: N.0 → 1 N.1	N.0.ü := 0.5 + (N.1.ü / 2)
R2: N.0 → 0 N.0	N.0.ü := 0.0 + (N.1.ü / 2)
R3: N.0 → 1	N.0.ü := 0.5
R4: N.0 → 0	N.0.ü := 0.0

Lösung 3.1.4.: Weil es endlich lange Dezimalbrüche gibt (z.B. 0.1), die nur durch unendlich lange (periodische) Binärbrüche dargestellt werden können. Andererseits gilt: jeder endliche Binärbruch kann durch einen endlichen Dezimalbruch dargestellt werden.

Lösung 3.2.2.1.: Übersetzung "von Hand":
1. push 1 push 10 add
2. push 101 push 11 add push 1010 push 10 add sub
3. push 1000 push 100 push 10 push 1 sub sub sub
4. push 1000 push 100 sub push 10 sub push 1 sub

Lösung 3.2.4.1.: Der Ausdruck (4 + 3) wird abgeleitet und übersetzt:

Zeilen-Nr.	Ableitung	Übersetzung
1	A:1	A:1.ü := S:1.ü
2	R3:	:= push 100 push 11 add
3	S:1	S:1.ü := A:2.ü
4	R4:	:= push 100 push 11 add
5	(A:2)	A:2.ü := A:3.ü || S:2.ü || add
6	R1:	:= push 100 push 11 add
7	(A:3 + S:2)	A:3.ü := S:3.ü
8	R3:	:= push 100
9	(S:3 + S:2)	S:3.ü := push || Z:1.ü
10	R5:	:= push 100
11	(Z:1 + S:2)	S:2.ü := push || Z:2.ü
12	R5:	:= push 11
13	(Z:1 + Z:2)	Z:1.ü := D:1.ü
14	R7:	:= 100
15	(D:1 + Z:2)	Z:2.ü := D:2.ü
16	R7:	:= 11
17	(D:1 + D:2)	D:1.ü := 100
18	R12:	
19	(4 + D:2)	D:2.ü := 11
20	R11:	
21	(4 + 3)	
22		

Lösung 3.2.4.2.: Ein Schema für die Übersetzung von dezimalen Infix-Ausdrücken (mit + und * und "Punktrechnung geht vor Strichrechnung") in binäre Postfix-Ausdrücke:

Syntaktische Regeln	Semantische Regeln
R1: A.0 → A.1 + B.1	A.0.ü := A.1.ü || B.1.ü || add
R2: A.0 → B.1	A.0.ü := B.1.ü
R3: B.0 → B.1 * C.1	B.0.ü := B.1.ü || C.1.ü || mult
R4: B.0 → C.1	B.0.ü := C.1.ü
R5: C.0 → (A.1)	C.0.ü := A.1.ü
R6: C.0 → Z.1	C.0.ü := push || Z.1.ü
R7: Z.0 → Z.1 D.1	Z.0.ü := Z.1.ü * 1010 + D.1.ü
R8: Z:0 → D.1	Z.0.ü := D.1.ü
R9: D.0 → 0	D.0.ü := 0
R10: D.0 → 1	D.0.ü := 1
R11: D.0 → 2	D.0.ü := 10
R12: D.0 → 3	D.0.ü := 11
R13: D.0 → 4	D.0.ü := 100
R14: D.0 → 5	D.0.ü := 101
R15: D.0 → 6	D.0.ü := 110
R16: D.0 → 7	D.0.ü := 111
R17: D.0 → 8	D.0.ü := 1000
R18: D.0 → 9	D.0.ü := 1001

Lösung 3.2.4.3.: Compiler-Operationen müssen vom Compiler (d.h. vom Übersetzer) während der Übersetzung eines Programms ausgeführt werden. Die Operationen der Ziel-Sprache werden vom Compiler (nur) in das Zielprogramm

eingefügt, aber nicht ausgeführt. Erst zur Laufzeit des Zielprogramms werden die Operationen der Ziel-Sprache (von der Ziel-Maschine) ausgeführt.

Lösung 3.3.1: Beide Programme machen im wesentlichen das Gleiche. Sie berechnen die Summe der Zahlen von 1 bis 10: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55.

Lösung 3.3.2.: Der Algorithmus von Euklid zur Berechnung des größten gemeinsamen Teilers zweier (nicht-negativer, ganzer) Zahlen kann in der Ziel-Sprache geschrieben z.B. so ausssehen:

```
zahla:   ds      101010
zahlb:   ds      110010
anfang:  push    zahla
         push    zahlb
         unequal
         bf      ende
         push    zahla
         push    zahlb
         lesseq
         bt      aklein
bklein:  push    zahla
         push    zahlb
         sub
         pop     zahla
         b       anfang
aklein:  push    zahlb
         push    zahla
         sub
         pop     zahlb
         b       anfang
ende:    nop
```

Berechnet wird der größte gemeinsame Teiler von zahla (im Beispiel: 45) und zahlb (im Beispiel: 50). Das Ergebnis steht am Ende sowohl in zahla als auch in zahlb.

In der Quell-Sprache geschrieben könnte der gleiche Algorithmus z.B. so aussehen:

```
prog
  var zahla := 45;
  var zahlb := 50;
begin
  while (zahla /= zahlb) do
    if (zahla ? zahlb) then
      zahlb := (zahlb - zahla);
    else
      zahla := (zahla - zahlb);
    end;
  end;
end;
```

Lösung 3.3.3.: Ein Übersetzungsschema zur Übersetzung der Quell-Sprache in die Ziel-Sprache:

Nicht-Terminale Symbole der Grammatik der Quellsprache:

P (Programm), DL (Deklarationsliste), BL (Befehls-Liste), D (Deklaration), B (Befehl), N (Name), BA (bool´scher Ausdruck), GA (Ganzzahl-Ausdruck), BU (Buchstabe), ZA (Zahl), ZI (Ziffer).

Das Übersetzungsschema:

Syntaktische Regeln				Semantische Regeln				
R1:	P.0	→	prog DL.1 begin BL.1 end	P.0.ü	:=		DL.1.ü BL.1.ü	
R2:	DL.0	→	D.1 ; DL.1	DL.0.ü	:=		D.1.ü DL.1.ü	
R3	DL.0	→	D.1 ;	DL.0.ü	:=		D.1.ü	
R4:	D.0	→	var N.1 := ZA.1	D.0.ü	:=	N.1.ü:	ds	ZA.1.ü
R5:	BL.0	→	B.1 ; BL.1	BL.0.ü	:=		B.1.ü BL.1.ü	
R6:	BL.0	→	B.1 ;	BL.0.ü	:=		B.1.ü	
R7:	B.0	→	N.1 := GA.1	B.0.ü	:=		GA.1.ü pop	 N.1.ü
R8:	B.0	→	if BA.1 then BL.1 else BL.2 end	B.0.ü	:=	 then$L: else$L: ifend$L:	inc BA.1.ü bf nop BL.1.ü b nop BL.2.ü nop	$L else$L ifend$L
R9:	B.0	→	while BA.1 do BL.1 end	B.0.ü	:=	 wdanf$L: wdend$L:	inc nop BA.1.ü bf BL.1.ü b nop	$L wdend$L wdanf$L
				-- „wd“ soll an „while-do-Schleife“ erinnern				
R10:	BA.0	→	(GA.1 = GA.2)	BA.0.ü	:=		GA.1.ü GA.2.ü equal	
R11:	BA.0	→	(GA.1 ? GA.2)	BA.0.ü	:=		GA.1.ü GA.2.ü unequal	
R12:	BA.0	→	(GA.1 ? GA.2)	BA.0.ü	:=		GA.1.ü GA.2.ü lesseq	
R13:	GA.0	→	N.1	GA.0.ü	:=		push	N.1.ü
R14:	GA.0	→	(GA .1 + GA .2)	GA.0.ü	:=		GA.1.ü GA.2.ü add	
R15:	GA.0	→	(GA.1 - GA .2)	GA.0.ü	:=		GA.1.ü GA.2.ü sub	

R16:	N.0	→	BU.1 N.1	N.0.ü	:=	BU.1.ü N.1.ü
R17:	N.0	→	BU.1	N.0.ü	:=	BU.1.ü
R18:	BU.0	→	a	BU.0.ü	:=	a
R19:	BU.0	→	b	BU.0.ü	:=	b
...	...		...	...		...
R43:	BU.0	→	z	BU.0.ü	:=	z
R44:	ZA.0	→	ZA.1 ZI.1	ZA.0.ü	:=	ZA.1.ü * 1010 + ZI.1.ü
R45:	ZA.0	→	ZI.1	ZA.0.ü	:=	ZI.1.ü
R46:	ZI.0	→	0	ZI.0.ü	:=	0
R47:	ZI.0	→	1	ZI.0.ü	:=	1
...	...		...	...		...
R55:	ZI.0	→	9	ZI.0.ü	:=	1001

Die unterstrichenen Operationen inc, * und + sind Compiler-Operationen, die der Übersetzer während einer Übersetzung ausführen muss. Dagegen sind ds, push, pop, lesseq, equal, add, sub, bt, b und nop Befehle, die der Übersetzer in das zu erzeugende Ziel-Programm "einbaut", die aber erst von der Zielmaschine ausgeführt werden.

Lösung 3.3.4: Eine Übersetzung mit Hilfe des obigen Übersetzungsschemas:

3.3.4.1.: Eine linkskanonische Ableitung des zu übersetzenden Programms. Um eine Verwechslung mit Vorkommnissen zu erschweren wurden die Instanzen der nichtterminalen Symbole willkürlich mit 10 beginnend nummeriert, und nicht mit 1 beginnend.

```
01          P:10
02      R1:
03          prog DL:10 begin BL:10 end
04      R3:
05          prog D:10 ; begin BL:10 end
06      R4:
07          prog var N:10 := ZA:10 ; begin BL:10 end
08      R17:
09          prog var BU:10 := ZA:10 ; begin BL:10 end
10      R18:
11          prog var  a := ZA:10 ; begin BL:10 end
12      R45:
13          prog var  a := ZI:10 ; begin BL:10 end
14      R47:
15          prog var  a := 1 ; begin BL:10 end
16      R6:
17          prog var a := 1 ; begin B:10 ; end
18      R9:
19          prog var a := 1 ; begin while BA:10 do BL:11 end ; end
20      R12:
21          prog var a := 1 ; begin while (GA:10 ? GA:11) do BL:11 end ; end
22      R13:
23          prog var a := 1 ; begin while (N:11 ? GA:11) do BL:11 end ; end
24      R17:
25          prog var a := 1 ; begin while (BU:11 ? GA:11) do BL:11 end ; end
26      R18:
27          prog var a := 1 ; begin while (a ? GA:11) do BL:11 end ; end
28      R13:
29          prog var a := 1 ; begin while (a ? N:12) do BL:11 end ; end
30      R17:
31          prog var a := 1 ; begin while (a ? BU:12) do BL:11 end ; end
32      R18:
33          prog var a := 1 ; begin while (a ? a) do BL:11 end ; end
34      R6:
35          prog var a := 1 ; begin while (a ? a) do B:11 ; end ; end
36      R7:
37          prog var a := 1 ; begin while (a ? a) do N:13 := GA:12 ; end ; end
38      R17:
39          prog var a := 1 ; begin while (a ? a) do BU:13 := GA:12 ; end ; end
40      R18:
41          prog var a := 1 ; begin while (a ? a) do a := GA:12 ; end ; end
42      R13:
43          prog var a := 1 ; begin while (a ? a) do a := N:14 ; end ; end
44      R16:
45          prog var a := 1 ; begin while (a ? a) do a := BU:14 ; end ; end
46      R18:
47          prog var a := 1 ; begin while (a ? a) do a := a ; end ; end
```

3.3.4.2.: Die Übersetzung erfolgt "von unten nach oben". Die Zeilen-Nummern am linken Rand sollen den Zusammenhang mit der obigen Ableitung deutlich machen:

Zeile	Regel	Attribut		Wert des Attributs			Wert des Attributs	
46	R17:	BU:14.ü	:=	a				
44	R16:	N:14.ü	:=	BU:14.ü	:=		a	
42	R12:	GA:12.ü	:=	push N:14.ü	:=		push	a
40	R17:	BU:13.ü	:=	a				
38	R16:	N:13.ü	:=	BU:13.ü	:=		a	
36	R7:	B:11.ü	:=	GA:12.ü	:=		push	a
				pop a	:=		pop	a
34	R6:	BL:11.ü	:=	B:11.ü	:=		push	a
							pop	a
32	R17:	BU:12.ü	:=	a				
30	R16:	N:12.ü	:=	BU:12.ü	:=		a	
28	R12:	GA:11.ü	:=	push N:12.ü	:=		push	a
26	R17:	BU:11.ü	:=	a				
24	R16:	N:11.ü	:=	BU:11.ü	:=		a	
22	R12:	GA:10.ü	:=	push N:11.ü	:=		push	a
20	R13:	BA:10.ü	:=	GA:10.ü	:=		push	a
				GA:11.ü			push	a
				lesseq			lesseq	
18	R9:	B:10.ü	:=	inc $L	:=	wdanf1:	nop	
				wdanf$L: nop			push	a
				BA:10.ü			push	a
				bf wdend$L			lesseq	
				BL:11.ü			bf	wdend1
				b wdanf$L			push a	
				wdend$L: nop			pop a	
							b	wdanf1
						wdend1:	nop	
16	R6:	BL:10.ü	:=	B:10.ü	:=	wdanf1:	nop	
							push	a
							push	a
							lesseq	
							bf	wdend1
							push	a
							pop	a
							b	wdanf1
						wdend1:	nop	
14	R46:	ZI:10.ü	:=	1				
12	R44:	ZA:10.ü	:=	ZI:10.ü	:=		1	
10	R17:	BU:10.ü	:=	a				
08	R16:	N:10.ü	:=	BU:10.ü	:=		a	
06	R4:	D:10.ü	:=	N:10.ü: ds ZA:10.ü	:=	a:	ds	1
04	R3:	DL:10.ü	:=	D:10.ü	:=	a:	ds	1
02	R1:	P:10.ü	:=	DL:10.ü	:=	a:	ds	1
				BL:10.ü		wdanf1:	nop	
							push	a
							push	a
							lesseq	
							bf	wdend1
							push	a
							pop	a
							b	wdanf1
						wdend1:	nop	

Aus der Instanz P:10 des nicht-terminalen Symbols P wurde das gesamte Quellprogramm abgeleitet. Also ist P:10.ü die Übersetzung des gesamten Quellprogramms.

Lösung 3.3.5.: Eine effizientere Übersetzung für while-Schleifen:

R9: B.0 → while BA.1 do BL.1 od B.0.ü :=

```
                inc     $L
                b       wdpruef$L
wdrumpf$L:      nop
                BL.1.ü
wdpruef$L:      BA.1.ü
                bt      wdrumpf$L
```

Lösung 3.3.6.: Erweiterung der Quellsprache und des Übersetzungsschemas um diverse Schleifen etc.:

R56: B.0 → do BL.1 while BA.1 end B.0.ü :=

```
                inc     $L
dwanf$L:        nop
                BL.1.ü
                BA.1.ü
                bt      dwanf$L$L
```

R57: B.0 → until BA.1 do BL.1 od B.0.ü :=

```
                inc     $L
                b       udpruef$L
udrumpf$L:      nop
                BL.1.ü
udpruef$L:      BA.1.ü
                bf      udrumpf$L
```

R58: B.0 → do BL.1 until BA.1 end B.0.ü :=

```
                inc     $L
duanf$L:        nop
                BL.1.ü
                BA.1.ü
                bf      duanf$L
```

R59: B.0 → if BA.1 then BL.1 end B.0.ü :=

```
                inc     $L
                BA.1.ü
                bf      ifend$L
                BL.1.ü
ifend$L:        nop
```

R60: B.0 → for N.1 in GA.1 .. GA.2 do BL.1 end B.0.ü :=

```
                inc     $L
                b       foranf$L
N.1.ü:          ds      0
bis$L:          ds      0
eins$L:         ds      1
foranf$L:       nop
                GA.1.ü
                pop     N.1.ü
                GA.2.ü
                pop     bis$L
                b       forpruef$L
forrumpf$L:     nop
                BL.1.ü
foradd1$L:      push    N.1.ü
                push    eins$L
                add
                pop     N.1.ü
forpruef$L:     push    N.1.ü
                push    bis$L
                lesseq
                bt      forrumpf$L
```

Lösung 4.2.1.: Eine Ableitung für das Wort fgklm:

S R4 BD R7 CD R9 fgD R10 fgklm

Lösung 4.2.2.: Die Lösung kann hier leider nicht wiedergegeben werden, Sie müssen sie selbst "durchführen".

Lösung 4.2.3.: Bei der Analyse des Wortes efhc rufen sich die Prozeduren des Parsers wie folgt auf:

S ruft A auf, A ruft B auf, B ruft C auf, C ruft match("fg") auf, aber next zeigt auf "fh", deshalb: match ruft error auf.

Lösung 4.2.4.: Weil man aus dem Zwischen-Symbol C die leere Symbolkette e ableiten kann. Dagegen kann man aus A nicht die leere Symbolkette ableiten.

Lösung 4.2.5.: Weil es für die Zwischen-Symbole S, A und C jeweils mehrere Regeln gibt. Dagegen gibt es für B und D jeweils nur eine Regel.

Lösung 4.3.1.: Die Anfangsmengen der rechten Seiten der Regeln der Grammatik G20:

first(aS) = {a}	first(ef) = {e}	first(kl) = {k}
first(AB) = {c, d}	first(gh) = {g}	first(ε) = {ε}
first(CD) = {g, h, i, j}	first(hi) = {h}	first(mm) = {m}
first(EF) = {ε, k, m}	first(ε) = {ε}	first(ε) = {ε}
first(cS) = {c}	first(ij) = {i}	
first(dS) = {d}	first(jk) = {j}	

Lösung 4.3.3.: Fall1: Man kann aus α nicht die leere Symbolfolge ε ableiten.
In diesem Fall gilt: first(αβ) = first(α).
Fall 2: Man kann aus α, aber nicht aus β, die leere Symbolfolge ε ableiten.
In diesem Fall gilt: first(αβ) = first(α) ∪ first(β).
Fall 3: Man kann aus α und aus β die leere Symbolfolge ε ableiten.
In diesem Fall gilt: first(αβ) = first(α) ∪ first(β) ∪ {ε}

Lösung 4.3.2.: Die Anfangsmengen der Zwischen-Symbole der Grammatik G20:

first(S)= {a, c, d, g, h, i, j, k, m, ε}
first(A) = {c, d}
first(B) = {e}
first(C) = {g, h, ε}
first(D) = {i, j}
first(E) = {k, ε}
first(F)= {m, ε}

Lösung 4.4.1.: first- und Folgemengen für die Grammatik G21:

first(ASB) = {a, b}	first(aAa) = {a}	first(BB) = {b}
first(ε) = {ε}	first(ε) = {ε}	first(b) = {b}
first(S) = {a, b, ε}	first(A) = {a, ε}	first(B) = {b}
follow(S) = {$, b}	follow(A) = {a, b}	follow(B) = {$, b}

Lösung 4.4.2: Anfangs- und Folgemengen (first- and follow sets) für die Grammatik G22:

1. Anfangs-Mengen

R1:	A	→	A + T	first(A + T)	= {x, y, (}
R2:	A	→	A - T	first(A - T)	= {x, y, (}
R3:	A	→	T	first(T)	= {x, y, (}
R4:	T	→	T * F	first(T ∗ F)	= {x, y, (}
R5:	T	→	T / F	first(T / F)	= {x, y, (}
R6:	T	→	F	first(F)	= {x, y, (}
R7:	F	→	x	first(x)	= {x}
R8:	F	→	y	first(y)	= {y}
R9:	F	→	(A)	first((A))	= {(}

2. Folge-Mengen:

follow(A) ⊃ {$}
R1: follow(A) ⊃ {+}
follow(T) ⊃ follow(A)
R2: follow(A) ⊃ {-}
follow(T) ⊃ follow(A)
R3: follow(T) ⊃ follow(A)
R4: follow(T) ⊃ {∗}
follow(F) ⊃ follow(T)
R5: follow(T) ⊃ {/}
follow(F) ⊃ follow(T)
R6: follow(F) ⊃ follow(T)
R9: follow(A) ⊃ {)}

Also gilt: follow(A) = {$, +, -,) }
follow(T) = {$, +, -,), ∗, / }
follow(F) = {$, +, -,), ∗, / }

Lösung 4.4.3.:

first(ABSC)	= {a, b, c}	first(ε)	= {ε}	first(cS)	= {c}	first(a)	= {a}
first(ε)	= {ε}	first(b)	= {b}	first(ε)	= {ε}		
first(S)	= {a, b, c, ε}	first(A)	= {a, ε}	first(B)	= {b, ε}	first(C)	= {c}
follow(S)	= {c, $}	follow(A)	= {a, b, c}	follow(B)	= {a, b, c}	follow(C)	= {c, $}

Lösung 4.5.1.: Anfangsmengen und Parser für die Grammatik G24:

first(ASB)	= {a, c}	first(aAa)	= {a}	first(bbB)	= {b}
first(ε)	= {ε}	first(c)	= {c}	first(d)	= {d}

```
proc S is
  if   next is in {a, c} then A; S; B;
  else do_nothing;
  end if;
end proc S;
```

```
proc A is
  if     next is in {a} then match("a"); A; match("a");
  elseif next is in {c} then match("c");
  else   error;
  end if;
end proc A;
```

```
proc B is
  if     next is in {b} then match("bb"); B;
  elseif next is in {d} then match("d");
  else   error;
  end if;
end proc B;
```

Analyse der Worte cd, acabbd und cbbbb

S	cd
ASB	cd
cSB	cd
cB	cd
cd	cd

S	acabbd
ASB	acabbd
aAaSB	acabbd
acaSB	acabbd
acaB	acabbd
acabbB	acabbd
acabbd	acabbd

S	cbbb
ASB	cbbb
cSB	cbbb
cB	cbbb
cbbB	cbbb
cbbbb	cbbb Fehler in Proc B, match("bb")

Lösung 4.6.1.: Zwei Lösungen (d.h. zwei Parser für die Grammatik G25):

```
procedure S is
  if next is in {a} then
    match("ab");
  elseif next is in {a} then
    match("ac");
  else
    error;
  end if;
end S;
```

```
procedure S is
  if next is in {a} then
    match("ac");
  elseif next is in {a} then
    match("ab");
  else
    error;
  end if;
end S;
```

Lösung 4.6.2.: Die Grammatik G26 erfüllt die LL-Bedingung-1, denn es gilt:

1. Die Mengen first(ASB) = {a, c} und first(ε) = {ε} sind disjunkt.
2. Die Mengen first(aAa) = {a} und first(c) = {c} sind disjunkt.
3. Die Mengen first(bbB) = {b} und first(d) = {d} sind disjunkt.

Lösung 4.7.2.:

1. Aus den Zwischen-Symbolen A (!), B und C kann man die leere Symbolkette ε ableiten.
2. Die Grammatik G28 erfüllt die LL-Bedingung-2, denn es gilt:
2.1. Die Mengen first(A) = {a, b, c, ε} und follow(A) = {$} sind disjunkt.
2.2. Die Mengen first(B) = {b, ε} und follow(B) = {c, $} sind disjunkt.
2.3. Die Mengen first(C) = {c, ε} und follow(C) = {$} sind disjunkt.

Lösung 4.7.3.:

1. Die Grammatik G29 ist eine LL-Grammatik, denn es gilt:

1.1. Die Grammatik G29 erfüllt die LL-Bedingung-1, denn es gilt:

1.1.1. Die beiden Mengen first(; BE RF) = {;} und first( )) = {) } sind disjunkt.
1.1.2. Die beiden Mengen first(EB) = {x, y, r, w} und first(ZB) = {i, u,) } sind disjunkt.
1.1.3. Die drei Mengen first(ID := AA) = {x, y}, first(read(ID)) = {r} und first(write(ID)) = {w} sind paarweise disjunkt.
1.1.4. Die drei Mengen first(if BA then BE else BE end) = {i}, first(until BA do BE end) und first(BF) = { (} sind paarweise disjunkt.
1.1.5. Die beiden Mengen first(+ ID) = {+} und first(- ID) = {-} sind disjunkt.
1.1.6. Die beiden Mengen first(= ID) = {=} und first(> ID) = {>} sind disjunkt.
1.1.7. Die beiden Mengen first(x) = {x} und first(y) = {y} sind disjunkt.

1.2. Die Grammatik G29 erfüllt die LL-Bedingung-2, denn sie enthält kein Zwischen-Symbol, aus dem sich die leere Symbolkette ableiten lässt.

2. Die Regeln der Grammatik G29 und die Anfangsmengen der rechten Seiten dieser Regeln:

Regel der Grammatik G29:				Anfangsmenge der rechten Seite
R1:	BF	→	(BE RF	{ (}
R2:	RF	→	; BE RF	{ j }
R3:	RF	→	)	{) }
R4:	BE	→	EB	{ x, y, r, w }
R5:	BE	→	ZB	{ i, u, (}
R6:	EB	→	ID := AA	{ x, y }
R7:	EB	→	read(ID)	{ r }
R8:	EB	→	write(ID)	{ w }
R9:	ZB	→	if BA then BE else BE end	{ i }
R10:	ZB	→	until BA do BE end	{ u }
R11:	ZB	→	BF	{ (}
R12:	AA	→	ID RA	{ x, y }
R13:	RA	→	+ ID	{ + }
R14:	RA	→	- ID	{ - }
R15:	BA	→	ID RB	{ x, y }
R16:	RB	→	= ID	{ = }
R17:	RB	→	> ID	{ > }
R18:	ID	→	x	{ x }
R19	ID	→	y	{ y }

3. Eine "von Hand" ausgeführte LL-Analyse für das Wort (read(x); if x>y then write(x) else y:=x+y end)
(d.h. die Ableitung, die bei einer LL-Analyse herauskommt)

	BF
R1	(BE RF
R4	(EB RF
R7	(read(ID) RF
R18	(read(x) RF
R2	(read(x); BE RF
R5	(read(x); ZB RF
R9	(read(x);i f BA then BE else BE end RF
R15	(read(x); if ID RB then BE else BE end RF
R18	(read(x);i f x RB then BE else BE end RF
R17	(read(x);i f x > ID then BE else BE end RF
R19	(read(x);i f x > y then BE else BE end RF
R4	(read(x);i f x > y then EB else BE end RF
R8	(read(x);i f x > y then write(ID) else BE end RF
R18	(read(x);i f x > y then write(x) else BE end RF
R4	(read(x);i f x > y then write(x) else EB end RF
R6	(read(x);i f x > y then write(x) else ID := AA end RF
R19	(read(x);i f x > y then write(x) else y := AA end RF
R12	(read(x);i f x > y then write(x) else y := ID RA end RF
R18	(read(x);i f x > y then write(x) else y := x RA end RF
R13	(read(x);i f x > y then write(x) else y := x + ID end RF
R19	(read(x);i f x > y then write(x) else y := x + y end RF
R3	(read(x);i f x > y then write(x) else y := x + y end)

Lösung 4.8.1.: Die Grammatik G30 erfüllt die LL-Bedingung-1 nicht, d.h. die Anfangs-Mengen der rechten Seiten der beiden Regeln sind nicht disjunkt. Sie können gar nicht disjunkt sein, da die eine Regel linksrekursiv ist.

Lösung 4.8.2.: Die "umgebaute Grammatik G30" sieht so aus:

R1:	S	→	abR
R2:	R	→	cScR
R3:	R	→	ε

Lösung 4.8.3.: Die "umgebaute Grammatik G31" sieht so aus:

R1:	A	→	B R	R6:	B	→	C S
R2:	R	→	+ B R	R7:	S	→	* C S
R3:	R	→	- B R	R8:	S	→	/ C S
R4:	R	→	ε	R9:	S	→	ε
R5:	C	→	(A)	R10:	C	→	id

Lösung 4.8.4.: Die "umgebaute Grammatik G32" sieht so aus:

R1:	S	→	Aa	R5:	Q	→	a d R Q
R2:	S	→	b	R6:	Q	→	ε
R3:	A	→	b d R Q	R7:	R	→	c R
R4:	A	→	c R Q	R8:	R	→	ε

Dabei sind Q und P neue Zwischen-Symbole. Die Zwischen-Symbole wurden in folgende Reihenfolge gebracht:
1. S, 2. A, 3. R und 4. Q.

Die umgebaute Grammatik ist leider immer noch nicht LL! Pech gehabt.

Lösung 4.9.2.: Die "verbesserte" Grammatik G34:

R1:	S	→	if B then S R	R4:	R	→	else S fi
R2:	S	→	s	R5:	R	→	fi
R3:	B	→	b				

Die verbesserte Grammatik ist eine LL-Grammatik. Diesmal haben wir Glück gehabt!

Lösung 5.1.1.: Mögliche Aktionen, von denen jeweils eine gewählt wurde (oder: das Bodensee-Wasser unter dem Eis):

Nr. der Konfig.	Mögliche Aktionen
1	Schieben
2	Schieben, Reduzieren nach R5
3	Schieben, Reduzieren nach R4
4	Schieben, Reduzieren nach R2
5	Schieben
6	Schieben, Reduzieren nach R5
7	Schieben, Reduzieren nach R3
8	Schieben, Reduzieren nach R2
9	Schieben
10	Schieben
11	Reduzieren nach R5
12	Reduzieren nach R4
13	Reduzieren nach R1, Reduzieren nach R2
14	Hier bleibt einem nichts anderes übrig als zu lächeln

Lösung 5.1.2.: Analyse des Wortes x/x mit Gefühl:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1		x/x$	Schieben
2	x	/x$	Red. nach R5
3	C	/x$	Red. nach R4
4	B	/x$	Schieben
5	B/	x$	Schieben
6	B/x	$	Red. nach R5
7	B/C	$	Red. nach R3
8	B	$	Red. nach R2
9	A	$	lächeln

Analyse des Wortes x-x/x mit Gefühl:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1		x-x/x$	Schieben
2	x	-x/x$	Red. nach R5
3	C	-x/x$	Red. nach R4
4	B	-x/x$	Red. nach R2
5	A	-x/x$	Schieben
6	A-	x/x$	Schieben
7	A-x	/x$	Red. nach R5
8	A-C	/x$	Red. nach R4
9	A-B	/x$	Schieben
10	A-B/	x$	Schieben
11	A-B/x	$	Red. nach R5
12	A-B/C	$	Red. nach R3
13	A-B	$	Red. nach R1
14	A	$	lächeln

Lösung 5.1.3.: Analyse des Wortes x*x+x^x mit Gefühl

Nr. der Konfig.	Stapel	Eingabe	Aktion
1		x*x+x^x$	Schieben
2	x	*x+x^x$	Red. nach R7
3	D	*x+x^x$	Red. nach R6
4	C	*x+x^x$	Red. nach R4
5	B	*x+x^x$	Schieben
6	B*	x+x^x$	Schieben
7	B*x	+x^x$	Red. nach R7
8	B*D	+x^x$	Red. nach R6
9	B*C	+x^x$	Red. nach R3
10	B	+x^x$	Red. nach R2
11	A	+x^x$	Schieben
12	A+	x^x$	Schieben
13	A+x	^x$	Red. nach R7
14	A+D	^x$	Red. nach R6
15	A+C	^x$	Schieben

16	A+C^	x$	Schieben
17	A+C^x	$	Red. nach R7
18	A+C^D	$	Red. nach R5
19	A+C	$	Red. nach R4
20	A+B	$	Red. nach R1
21	A	$	lächeln

Lösung 5.2.1.: Analyse des Wortes x/x mit Tabelle:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x/x$	s4
2	Z0 x Z4	/x$	r5
3	Z0 C Z3	/x$	r4
4	Z0 B Z2	/x$	s6
5	Z0 B Z2 / Z6	x$	s4
6	Z0 B Z2 / Z6 x Z4	$	r5
7	Z0 B Z2 / Z6 C Z8	$	r3
8	Z0 B Z2	$	r2
9	Z0 A Z1	$	lächeln

Analyse des Wortes x-x/x mit Tabelle:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x-x/x$	s4
2	Z0 x Z4	-x/x$	r5
3	Z0 C Z3	-x/x$	r4
4	Z0 B Z2	-x/x$	r2
5	Z0 A Z1	-x/x$	s5
6	Z0 A Z1 - Z5	x/x$	s4
7	Z0 A Z1 - Z5 x Z4	/x$	r5
8	Z0 A Z1 - Z5 C Z3	/x$	r4
9	Z0 A Z1 - Z5 B Z7	/x$	s6
10	Z0 A Z1 - Z5 B Z7 / Z6	x$	s4
11	Z0 A Z1 - Z5 B Z7 / Z6 x Z4	$	r5
12	Z0 A Z1 - Z5 B Z7 / Z6 C Z8	$	r3
13	Z0 A Z1 - Z5 B Z7	$	r1
14	Z0 A	$	lächeln

Lösung 5.3.2.: Eine LR-Parser-Tabelle konstruieren:

1. Die Zustände des endlichen Automaten:

Z0 = { S->•A,
A->•A+B,
A->•A -B,
A->•B,
B->•n}

goto(Z0, A) = Z1 = { S->A•,
A->A•+B,
A->A• -B}

goto(Z0, B) = Z2 = { A->B•}

goto(Z0, x) = Z3 = { B->x•}

goto(Z1, +) = Z4 = { A->A+•B, B->•x}

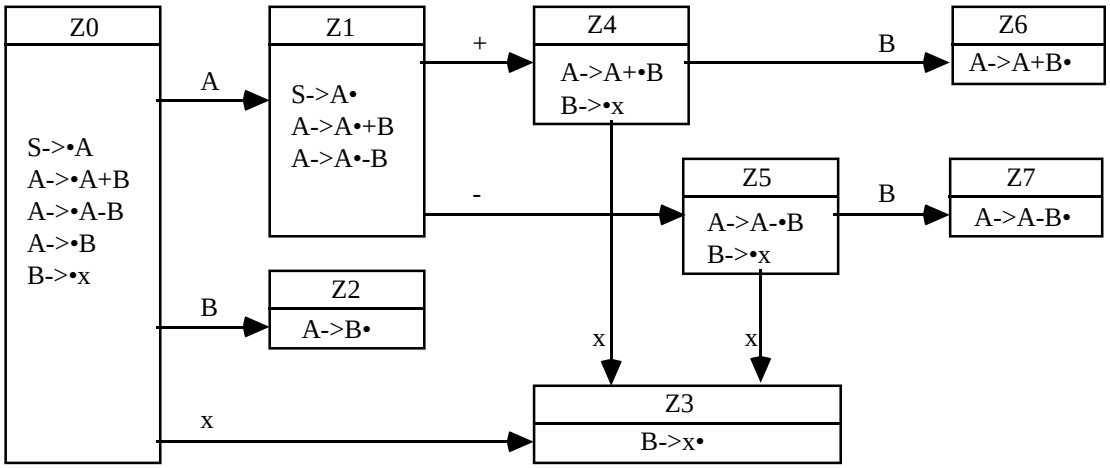
goto(Z1, -) = Z5 = { A->A -•B, B->•x}

goto(Z4, B) = Z6 = { A->A+B•}

goto(Z5, B) = Z7 = { A->A -B•}

Der endliche Automat hat also die 4 Zustände Z0, Z1, Z2 und Z3.

2. Graphische Darstellung des endlichen Automaten:



3. Die Folgemengen der Zwischensymbole: follow(A) = {+, -, $}
follow(B) = {+, -, $}

4. Die Parser-Tabelle:

	Aktionen				Folge-zustände	
Zustand	x	+	-	$	A	B
Z0	s3				Z1	Z2
Z1		s4	s5	lächeln		
Z2		r3	r3	r3		
Z3		r4	r4	r4		
Z4	s3					Z6
Z5	s3					Z7
Z6		r1	r1	r1		
Z7		r2	r2	r2		

Lösung 5.3.3.: Analyse des Wortes x-x+x mit der Tabelle aus Aufgabe 5.3.2., ohne Gefühl:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x-x+x$	s3
2	Z0 x Z3	-x+x$	r4
3	Z0 B Z2	-x+x$	r3
4	Z0 A Z1	-x+x$	s5
5	Z0 A Z1 - Z5	x+x$	s3
6	Z0 A Z1 - Z5 x Z3	+x$	r4
7	Z0 A Z1 - Z5 B Z7	+x$	r2
8	Z0 A Z1	+x$	s4
9	Z0 A Z1 + Z4	x$	s3
10	Z0 A Z1 + Z4 x Z3	$	r4
11	Z0 A Z1 + Z4 B Z6	$	r1
12	Z0 A Z1	$	lächeln

Lösung 5.3.4.: Noch eine LR-Parser-Tabelle konstruieren:

1. Die Zustände: Z0 = { S->•A,
A->•A+B,
A->•B,
B->•B*C,
B->•C,
C->•C^D,
C->•D,
D->•n}

goto(Z0, A) = Z1 = { S->A•,
A->A•+B}

goto(Z0, B) = Z2 = { A->B•,
B->B•*C}

goto(Z0, C) = Z3 = { B->C•,
C->C•^D}

goto(Z0, D) = Z4 = { C->D•}

goto(Z0, x) = Z5 = { D->x•}

goto(Z1, +) = Z6 = { A->A+•B,
B->•B*C,
B->•C,
C->•C^D,
C->•D
D->•x}

goto(Z2, *) = Z7 = { B->B*•C,
C->•C^D,
C->•D,
D->•x}

goto(Z3, ^) = Z8 = { C->C^•D,
D->•x}

goto(Z6, B) = Z9 = { A->A+B•,
B->B•*C}

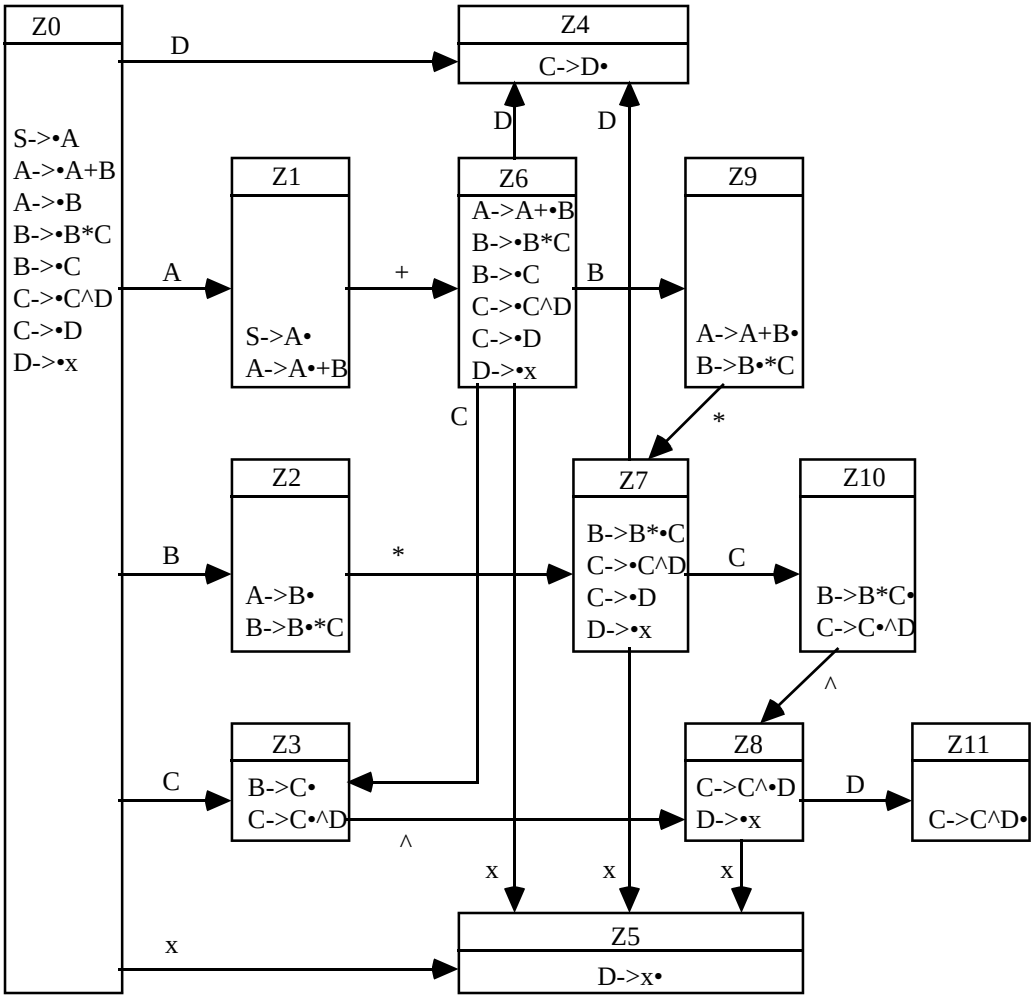
goto(Z7, C) = Z10 = { B->B*C•,
C->C•^D}

goto(Z8, D) = Z11 = { C->C^D•}

Außerdem gilt:

goto(Z6, C) = Z3 goto(Z6, D) = Z4 goto(Z6, x) = Z5 goto(Z7, D) = Z4
goto(Z7, x) = Z5 goto(Z8, x) = Z5 goto(Z9, *) = Z7 goto(Z10, ^) = Z8

2. Graphische Darstellung des endlichen Automaten:



3. Die Folgemengen:

follow(A) = {$, +} follow(C) = {$, +, *, ^}
follow(B) = {$, +, *} follow(D) = {$, +, *, ^}

4. Die Parser-Tabelle:

	Aktionen					Folgezustände			
Zustand	x	+	*	^	$	A	B	C	D
Z0	s5					Z1	Z2	Z3	Z4
Z1		s6			läch				
Z2		r2	s7		r2				
Z3		r4	r4	s8	r4				
Z4		r6	r6	r6	r6				
Z5		r7	r7	r7	r7				
Z6	s5						Z9	Z3	Z4
Z7	s5							Z10	Z4
Z8	s5								Z11
Z9		r1	s7		r1				
Z10		r3	r3	s8	r3				
Z11		r5	r5	r5	r5				

Lösung 5.3.5.: Analyse des Wortes x-x+x mit der Tabelle aus der vorigen Aufgabe 5.3.4.:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x*x$	s5
2	Z0 x Z5	*x$	r7
3	Z0 D Z4	*x$	r6
4	Z0 C Z3	*x$	r4
5	Z0 B Z2	*x$	s7
6	Z0 B Z2 * Z7	x$	s5
7	Z0 B Z2 * Z7 x Z5	$	r7
8	Z0 B Z2 * Z7 D Z4	$	r6
9	Z0 B Z2 * Z7 C Z10	$	r3
10	Z0 B Z2	$	r2
11	Z0 A Z1	$	lächeln

Ananlyse des Wortes x*x+x^x mit der Tabelle aus der vorigen Aufgabe 5.3.4.:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x*x+x^x$	s5
2	Z0 x Z5	*x+x^x$	r7
3	Z0 D Z4	*x+x^x$	r6
4	Z0 C Z3	*x+x^x$	r4
5	Z0 B Z2	*x+x^x$	s7
6	Z0 B Z2 * Z7	x+x^x$	s5
7	Z0 B Z2 * Z7 x Z5	+x^x$	r7
8	Z0 B Z2 * Z7 D Z4	+x^x$	r6
9	Z0 B Z2 * Z7 C Z10	+x^x$	r3
10	Z0 B Z2	+x^x$	r2
11	Z0 A Z1	+x^x$	s6
12	Z0 A Z1 + Z6	x^x$	s5
13	Z0 A Z1 + Z6 x Z5	^x$	r7
14	Z0 A Z1 + Z6 D Z4	^x$	r6
15	Z0 A Z1 + Z6 C Z3	^x$	s8
16	Z0 A Z1 + Z6 C Z3 ^ Z8	x$	s5
17	Z0 A Z1 + Z6 C Z3 ^ Z8 x Z5	$	r7
18	Z0 A Z1 + Z6 C Z3 ^ Z8 D Z11	$	r5
19	Z0 A Z1 + Z6 C Z3	$	r4
20	Z0 A Z1 + Z6 B Z9	$	r1
21	Z0 A Z1	$	lächeln

Lösung 5.4.1.1. 1. Analyse des Wortes x-x-x nach Entscheidung des S/R-Konfliktes zugunsten von s3:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x*x$	s2
2	Z0 x Z2	*x$	r2
3	Z0 A Z1	*x$	s3
4	Z0 A Z1 - Z3	*x$	s2
5	Z0 A Z1 - Z3 x Z2	*x$	r2
6	Z0 A Z1 - Z3 A Z4	x$	s3
7	Z0 A Z1 - Z3 A Z4 - Z3	$	s2
8	Z0 A Z1 - Z3 A Z4 - Z3 x Z2	$	r2
9	Z0 A Z1 - Z3 A Z4 - Z3 A Z4	$	r1
10	Z0 A Z1 - Z3 A Z4	$	r1
11	Z0 A Z1	$	lächeln

Bei dieser Analyse ergibt sich die (rechtskanonische) Ableitung R1, R1, R2, R2, R1. Dabei ist "-" rechtsassoziativ, d.h. das Wort x-x-x entspricht x-(x-x). Das ist unüblich!

1. Analyse des Wortes x-x-x nach Entscheidung des S/R-Konfliktes zugunsten von r1:

Nr. der Konfig.	Stapel	Eingabe	Aktion
1	Z0	x*x$	s2
2	Z0 x Z2	*x$	r2
3	Z0 A Z1	*x$	s3
4	Z0 A Z1 - Z3	*x$	s2
5	Z0 A Z1 - Z3 x Z2	*x$	r2
6	Z0 A Z1 - Z3 A Z4	x$	r1
7	Z0 A Z1	$	s3
8	Z0 A Z1 - Z3	$	s2
9	Z0 A Z1 - Z3 x Z2	$	r2
10	Z0 A Z1 - Z3 A Z4	$	r1
11	Z0 A Z1	$	lächeln

Bei dieser Analyse ergibt sich die (rechtskanonische) Ableitung R1, R2, R1, R2, R2. Dabei ist "-" linksassoziativ, d.h. das Wort x-x-x entspricht (x-x)-x. Das ist üblich!

Lösung 5.4.1.1.: Die gefundene Ableitung entspricht nicht den üblichen Konventionen, da sie das Minus-Zeichen rechtsassoziativ "auffasst". Diese Ableitung ist also "nicht richtig".

Lösung 5.4.1.2.: Die gefundene Ableitung entspricht den üblichen Konventionen, ist also "richtig". Das Minus-Zeichen ist dabei linksassoziativ.

Lösung 5.4.1.3.: Parser-Tabelle für die if-then-else-Sprache:

1. Die Zustände:

Z0 = { S->•ANW,
ANW->•anw,
ANW->•if aus then ANW,
ANW->•if aus then ANW else ANW}

goto(Z0, ANW) = Z1 = { S->ANW•}

goto(Z0, anw) = Z2 = { ANW->anw•}

goto(Z0, if) = Z3 = { ANW->if•aus then ANW,
ANW->if•aus then ANW else ANW}

goto(Z3, aus) = Z4 = { ANW->if aus•then ANW,
ANW->if aus•then ANW else ANW}

goto(Z4, then) = Z5 = { ANW->if aus then•ANW,
ANW->if aus then•ANW else ANW,
ANW->•anw,
ANW->•if aus then ANW,
ANW->•if aus then ANW else ANW}

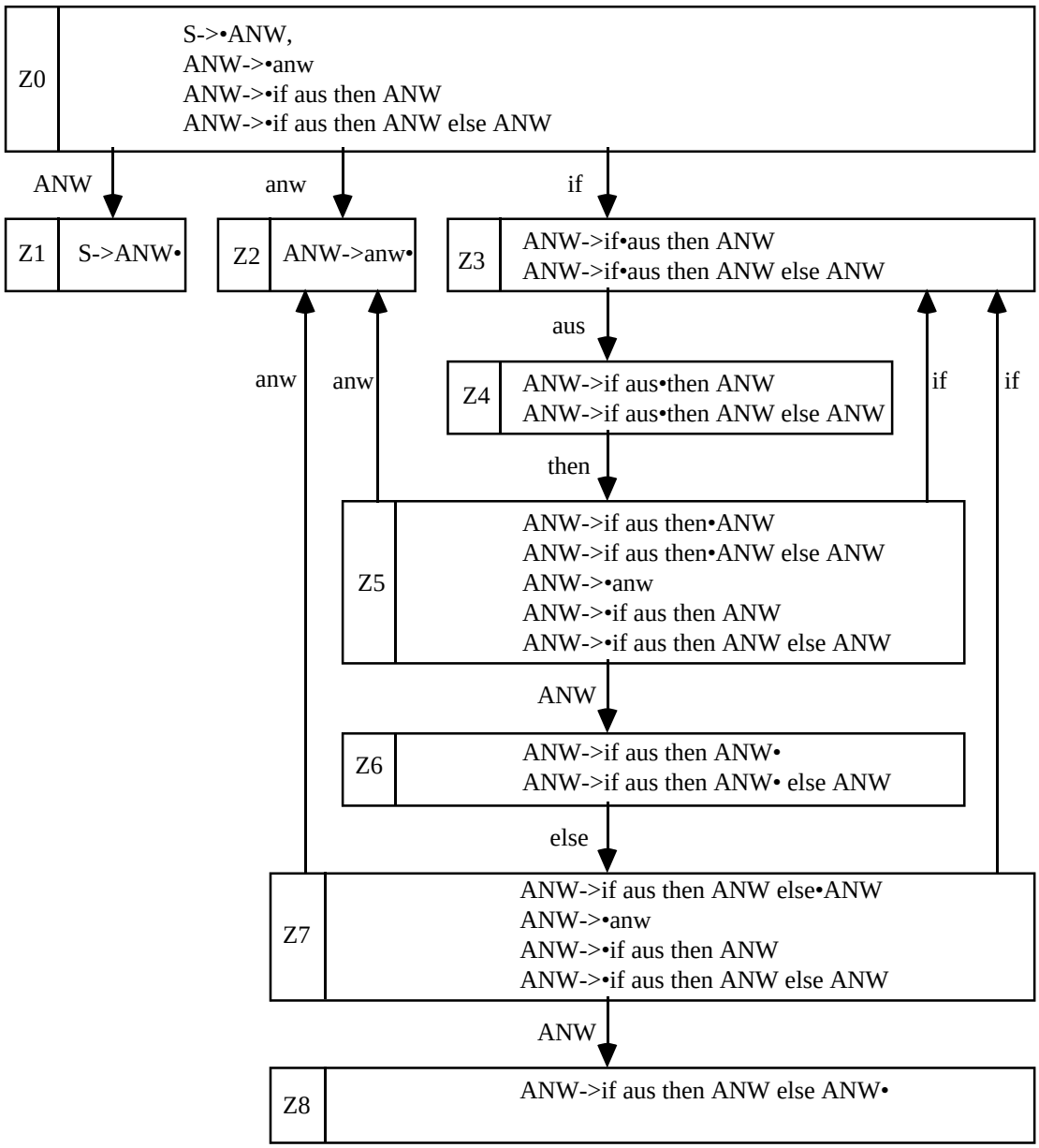
goto(Z5, ANW) = Z6 = { ANW->if aus then ANW•,
ANW->if aus then ANW•else ANW}

goto(Z6, else) = Z7 = { ANW->if aus then ANW else•ANW,
ANW->•anw,
ANW->•if aus then ANW,
ANW->•if aus then ANW else ANW}

goto(Z7, ANW) = Z8 = { if aus then ANW else ANW•}

Außerdem gilt:
goto(Z5, anw) = goto(Z6, anw) = Z2
goto(Z5, if) = goto(Z6, if) = Z3

2. Graphische Datstellung des endlichen Automaten:



3. Die Folgemenge: follow(ANW) = {else, $}

4. Die Parser-Tabelle:

	Aktion						Folge-Zust.
Zustand	anw	if	aus	then	else	$	ANW
Z0	s2	s3					Z1
Z1		s3				lächeln	
Z2					r1	r1	
Z3			s4				
Z4				s5			
Z5	s2	s3					Z6
Z6					s7 oder r2	r2	
Z7	s2	s3					Z8
Z8					r3	r3	

Analyse des Wortes if aus then if aus then anw else anw mit dem S/R-Konflikt entschieden für s7 (gegen r2):

Nr.	Stapel	Eingabe	Aktion
1	0	if aus then if aus then anw else anw$	s3
2	0 if 3	aus then if aus then anw else anw$	s4
3	0 if 3 aus 4	then if aus then anw else anw$	s5
4	0 if 3 aus 4 then 5	if aus then anw else anw$	s2
5	0 if 3 aus 4 then 5 if 3	aus then anw else anw$	s3
6	0 if 3 aus 4 then 5 if 3 aus 4	then anw else anw$	s5
7	0 if 3 aus 4 then 5 if 3 aus 4 then 5	anw else anw$	s2
8	0 if 3 aus 4 then 5 if 3 aus 4 then 5 anw 2	else anw$	r1
9	0 if 3 aus 4 then 5 if 3 aus 4 then 5 ANW 6	else anw$	s7
10	0 if 3 aus 4 then 5 if 3 aus 4 then 5 ANW 6 else 7	anw$	s2
11	0 if 3 aus 4 then 5 if 3 aus 4 then 5 ANW 6 else 7 anw 2	$	r1
12	0 if 3 aus 4 then 5 if 3 aus 4 then 5 ANW 6 else 7 ANW 8	$	r3
13	0 if 3 aus 4 then 5 ANW 6	$	r2
14	0 ANW 1	$	läch.

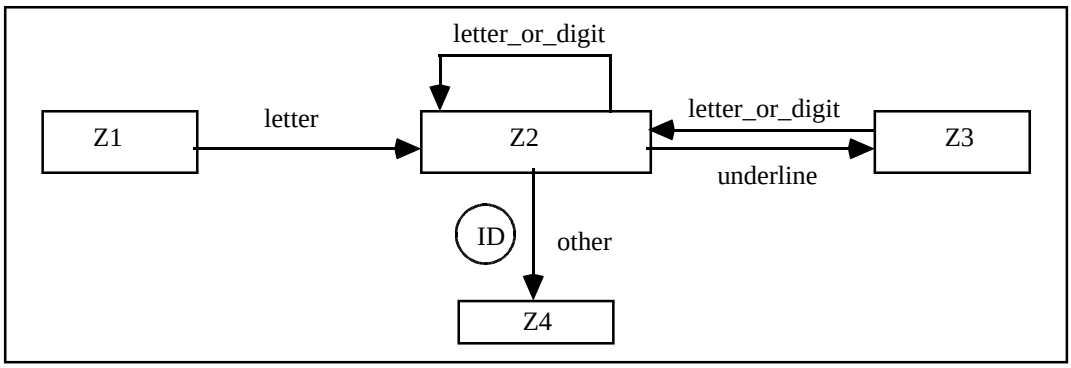
Analyse des Wortes if aus then if aus then anw else anw mit dem S/R-Konflikt entschieden für r2 (gegen s7):

Nr.	Stapel	Eingabe	Aktion
1	0	if aus then if aus then anw else anw$	s3
2	0 if 3	aus then if aus then anw else anw$	s4
3	0 if 3 aus 4	then if aus then anw else anw$	s5
4	0 if 3 aus 4 then 5	if aus then anw else anw$	s2
5	0 if 3 aus 4 then 5 if 3	aus then anw else anw$	s3
6	0 if 3 aus 4 then 5 if 3 aus 4	then anw else anw$	s5
7	0 if 3 aus 4 then 5 if 3 aus 4 then 5	anw else anw$	s2
8	0 if 3 aus 4 then 5 if 3 aus 4 then 5 anw 2	else anw$	r1
9	0 if 3 aus 4 then 5 if 3 aus 4 then 5 ANW 6	else anw$	r2
10	0 if 3 aus 4 then 5 ANW 6	else anw$	r2
11	0 ANW 1	else anw$	???

Hier "verläuft sich die Analyse in einer Sackgasse".

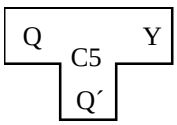
Lösung 6.4.1.: Ein endlicher Automat zur Erkennung von Ada-Bezeichnern (identifier):

Eingabe-Zeichen E = {a b ... z A B ... Z 0 1 ... 9 _ + * - / ? ! ...}
Ausgabe-Token A = { ID} -- Nur ein einziges Token
Zustände Z = { Z1, Z2, Z3, Z4}
Hilfs-Mengen: letter = { a b ... z A B ... Z}
digit = { 0 1 ... 9}
letter_or_digit = letter ∪ digit
underline = { _ }
other = { + * - / ? ! ...}

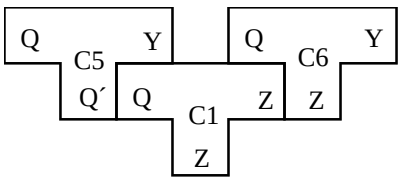


Lösung 8.1. Wir führen die Portierung des Compilers C1 von der Z-Maschine auf die Y-Maschine in drei Schritten durch:

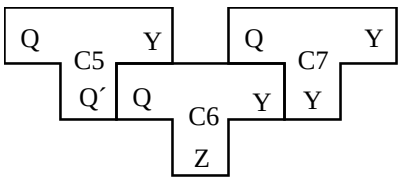
1. Aus dem sorgfältig geschriebenen Compiler C2 erzeugen wir (ebenfalls sorgfältig und) von Hand einen Compiler C5, indem wir die Code-Erzeugungsphase von C2 (welche Z-Code erzeugt) gegen eine neue Code-Erzeugungsphase austauschen, die Y-Code erzeugt. Das folgende T-Diagramm beschreibt den so erzeugten Compiler C5:



2. Wir übersetzen C5 mit C1. Das Ergebnis der Übersetzung heiße C6:

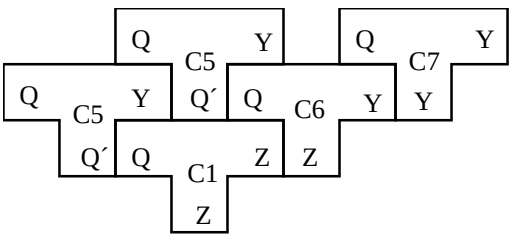


3. Zum Schluß übersetzen wir C5 noch einmal, diesmal mit C6. Das Ergebnis heiße C7:



Der Compiler C7 läuft selbst auf der Y-Maschine und übersetzt Q-Programme in Y-Programme. Er ist bezüglich beider wichtigen Eigenschaften "gut".

4. Hier noch eine Zusammenfassung der Schritte 1. bis 3. in einem T-Diagramm:



Bei dieser Portierung erfordert nur der erste Schritt "menshliche Arbeit" (Austausch eines Code-Erzeugers gegen einen anderen). Die anderen Schritte können von einem Computer durchgeführt werden.