zu Kapitel 3

Stand: 30.1.2008

Einführung in die Informatik III

(für Studierende des 3. Fachsemesters)

Pflichtvorlesung für die Diplomstudiengänge "Informatik" und "Automatisierungstechnik in der Produktion" sowie wählbare Vorlesung für weitere Studiengänge, z. B. für Mathematik, Softwaretechnik, Lehramt Informatik usw.

Universität Stuttgart, Wintersemester 2007/2008

Dozent: Volker Claus

3. Maschinennahe (abstrakte) Programme

3.1 Registermaschinen (= random access machine = RAM)

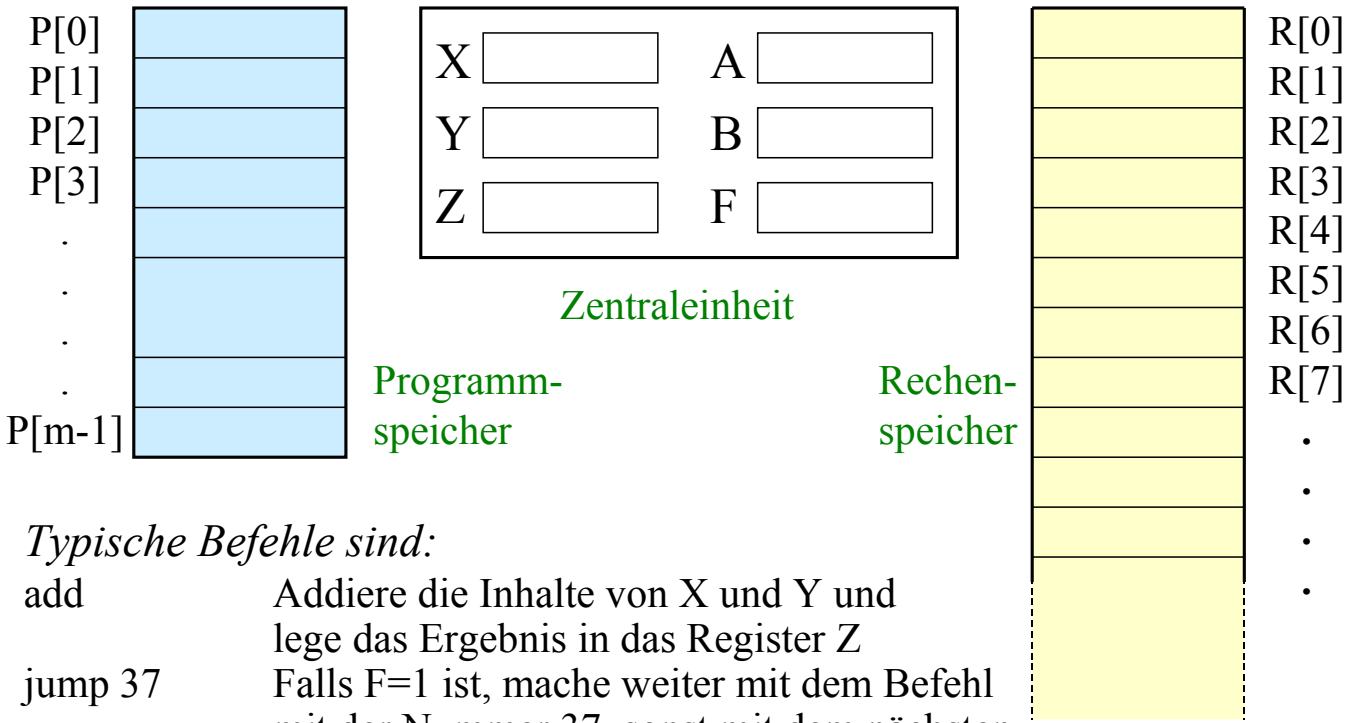
Die folgenden 10 Folien sind eine Wiederholung aus der Grundvorlesung, siehe in "Informatik I" den Abschnitt 6.6.

3.1.1 Veranschaulichung von Registermaschinen

Eine Registermaschine ist wie ein kleiner Mikroprozessor aufgebaut. Sie besteht aus

1. einer *Zentraleinheit* mit 6 Registern (Speicherzellen): 3 Register X, Y und Z für arithmetische und logische Operationen, ein Adressregister A für den Zugriff auf Rechenspeicherzellen, ein Befehlszählregister B, in dem die Adresse des auszuführenden Befehls steht, und ein "Flag"-Register F, in dem das Ergebnis von Operationen abgelegt wird;
2. einem *Programmspeicher* P, in dem nacheinander die Befehle des abzuarbeitenden endlichen Programms stehen;
3. einem (unendlich langen) *Rechenspeicher* R mit durchnummerierten Speicherzellen, die jeweils eine (beliebig große) ganze Zahl aufnehmen können.

Struktur der RAM



Typische Befehle sind:

add	Addiere die Inhalte von X und Y und lege das Ergebnis in das Register Z
jump 37	Falls F=1 ist, mache weiter mit dem Befehl mit der Nummer 37, sonst mit dem nächsten
LeftShift X	Schiebe den Inhalt von X eine Stelle nach links und füge eine 0 an
comp(Y>Z)	if der Inhalt von Y ist größer als der von Z then F:=1 else F:=0 fi
read X	X := $R_{<A>}$, d.h., sei a der Inhalt von A, so weise zu X:=R_a
load A,4	A := 4 (Wertzuweisung einer Konstanten an ein Register)

3.1.2 *Der übliche Befehlssatz einer Registermaschine* lautet (hierbei bezeichnen V und V' beliebige Register, c ist eine ganze Zahl, $b \in \mathbb{N}_0$, R_k ist die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ ist eine Vergleichsoperation; außer beim Jump-Befehl wird nach jedem Befehl B um 1 erhöht):

Befehl	Bedeutung	Befehl	Bedeutung
load V,c	V := c	copy V,V'	V := V'
read V	V := $R_{<A>}$	write V	$R_{<A>}$:= V
succ	X := X+1	add	X := Y+Z
sub	X := Y-Z	shift	X := X div 2

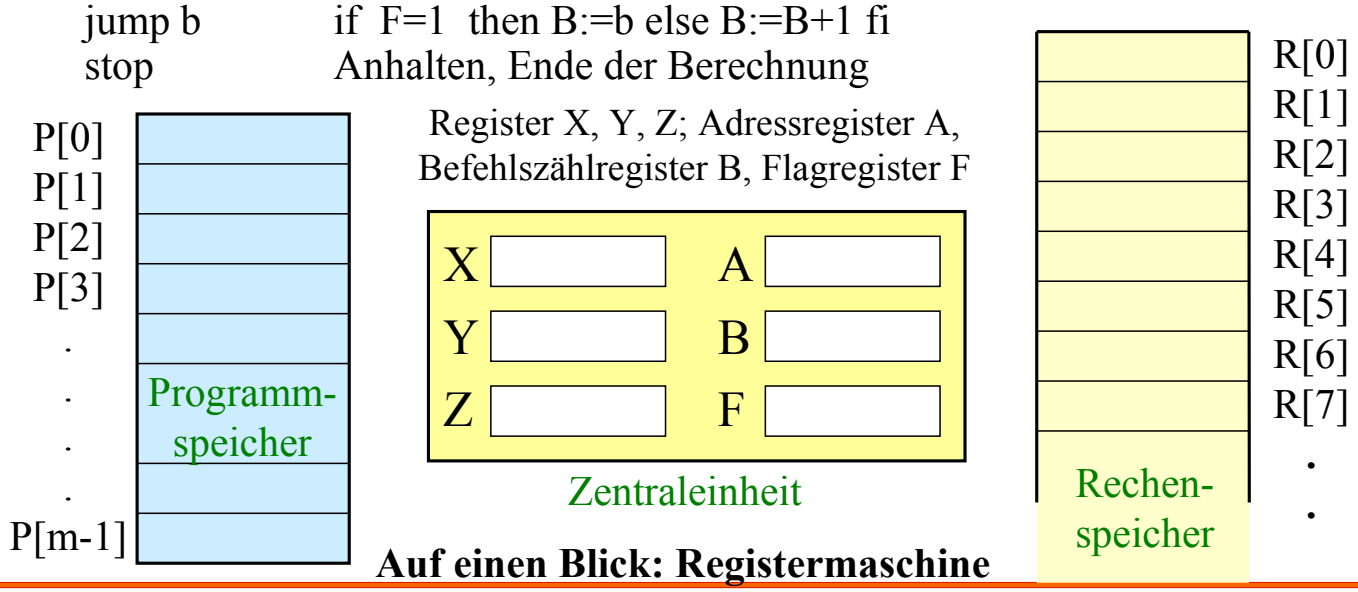
comp (σ)	if XσY then F:=1 else F:=0 fi
jump b	if F=1 then B:=b else B:=B+1 fi
stop	Anhalten, Ende der Berechnung

Diese Befehle finden sich in allen Mikroprozessoren; diese haben in der Regel aber noch viel mehr Befehle, insbesondere bzgl. der Adressierung, der Verschiebungen von Daten, der eingebauten Kellermechanismen und der Unterbrechungsbefehle. Die Ausformulierung von Algorithmen als Registermaschine ähnelt daher der Maschinenprogrammierung.

Der übliche Befehlssatz einer Registermaschine (V und V' seien Register, c eine ganze Zahl, $b \in \mathbb{N}_0$, R_k die k-te Speicherzelle, $\sigma \in \{>, \geq, <, \leq, =, \neq\}$ eine Vergleichsoperation; außer bei jump wird nach jedem Befehl B um 1 erhöht):

Befehl	Bedeutung	Befehl	Bedeutung
load V,c	V := c	copy V,V'	V := V'
read V	V := $R_{<A>}$	write V	$R_{<A>}$:= V
succ	X := X+1	add	X := Y+Z
sub	X := Y-Z	shift	X := X div 2

comp (σ)	if XσY then F:=1 else F:=0 fi
jump b	if F=1 then B:=b else B:=B+1 fi
stop	Anhalten, Ende der Berechnung



Auf einen Blick: Registermaschine

Beispiel 3.1.3: Test auf Teilbarkeit durch 2 (die Zahl n ≥ 0 stehe anfangs in R_0).
Naheliegende Lösung: Subtrahiere von n ständig die Zahl 2. Die Zahl n sei hier der Variablen I zugewiesen. Dann lautet das Programmstück: **while I>1 do I:=I-2 od;**
Anschließend steht in I der Rest der Division von n durch 2.
Übertrage dieses Programmstück in den Befehlssatz der Registermaschine (I ⇔ Register Y, Zahl 2 ⇔ Z; anfangs steht n in der Rechenspeicherzelle R_0, am Ende stehe das Ergebnis in R_0; sei a der Inhalt von A, so bezeichnet $R_{<A>}$ den Inhalt der Rechenspeicherzelle R_a). Man erhält folgendes Registermaschinenprogramm:

Nummer	Befehl	Erläuterung
0:	load A,0	A := 0
1:	read Y	Y := n (= R_0 = $R_{<A>}$)
2:	load Z,2	Z := 2
3:	load X,1	X := 1
4:	comp (≥)	teste, ob X ≥ Y ist (beachte: in X steht eine 1)
5:	jump 10	if 1 ≥ Y then weiter bei Befehl mit Nummer 10 fi
6:	sub	X := Y-Z (also X:=Y-2)
7:	copy Y,X	Y := X
8:	load F,1	F:=1 (um einen Sprung nach 3 zu erzwingen)
9:	jump 3	weitermachen beim Befehl mit der Nummer 3
10:	write Y	in A steht noch 0, also: R_0 := Y
11:	stop	

Betrachtet man nun Registermaschinen, die nur *auf ganzen Zahlen* arbeiten, so muss man zuvor prüfen, ob n < 0 ist oder nicht. Wir fügen daher vor das oben angegebene Programmstück noch die Anweisung if I < 0 then I := 0 - I fi , übertragen dieses Ada-Programm wiederum in ein Registermaschinenprogramm und erhalten das rechts stehende Programm:

Man kann sich überzeugen, dass jedes Ada-Programm in ein solches Registermaschinenprogramm übersetzt werden kann und dass sich dieser Prozess automatisieren lässt (siehe 3.2). Solche einfachen Sprachen heißen in der Praxis "Maschinensprachen" oder "Maschinencode"; sie werden meist in Form von Assemblern ein wenig lesbarer und komfortabler gemacht.

```
 0: load A,0
 1: read Y
 2: load X,0
 3: comp (≤)
 4: jump 9
 5: copy Z,Y
 6: load Y,0
 7: sub
 8: copy Y,X
 9: load Z,2
10: load X,1
11: comp (≥)
12: jump 17
13: sub
14: copy Y,X
15: load F,1
16: jump 10
17: write Y
18: stop
```

Die verschiedenen Typen von Registermaschinen unterscheiden sich in ihren (elementaren) Datentypen und in ihren Befehlssätzen, siehe Übungen. Meist fügt man noch weitere Speicher (z. B. ein Eingabeband, das nur gelesen werden darf, und ein Ausgabeband, das nur beschrieben werden darf, sowie einen Stack) hinzu.

Die englische Bezeichnung "random access machine" stammt daher, dass die Maschine über das Register A einen "wahlfreien Zugriff" auf die Rechenspeicherzellen erlaubt.

Registermaschinen geben in der Regel die Komplexität von Programmen recht gut wieder. Sie werden daher vielen theoretischen Untersuchungen zugrunde gelegt.

3.1.4 *Hinweis zur Übersetzung von Ada-Programmen*

In den Programmen von Registermaschinen gibt es nur zwei Kontrollstrukturen:

- Übergang zum nächsten Befehl (dies entspricht dem ";" in Ada). Dies wird dadurch realisiert, dass nach jeder Ausführung eines Befehls das Register B um 1 erhöht wird.
- Bedingter Sprung (falls F=1) zum Befehl mit der Nummer b (jump b), sofern die Nummer b im Programm vorkommt (anderenfalls Fehlerabbruch).

Man kann alle strukturierten Anweisungen in höheren Programmiersprachen durch diese beiden Kontrollstrukturen simulieren. In Ada sind Sprünge mit dem Schlüsselwort goto zugelassen. Anstelle der Nummern verwendet Ada *Marken*, dies sind Bezeichner, die in << ... >> eingeschlossen und vor eine Anweisung gesetzt werden. Zwei Beispiele folgen.

```
if X ≥ Y then Z := 1; else X := Y; end if; X := X+1;
```

wird im Registerprogramm zu (willkürlich wurde 20 als Nummer des ersten Befehls gewählt):

```
20: comp (<)        vergleiche X mit Y bzgl. "kleiner"
21: jump 25         springe zum else-Zweig
22: load Z,1        Z := 1 (then-Zweig ausführen)
23: load F,1        bereite einen "unbedingten" Sprung vor
24: jump 26         überspringe den else-Zweig
25: copy X,Y        X := Y (else-Zweig ausführen)
26: succ            X := X+1
```

In Ada kann man dieses Programmstück direkt nachbilden:

```
F := X < Y;  if F then goto ELSE_ZWEIG; end if;

Z := 1; goto DANACH;
<<ELSE_ZWEIG>>  X := Y;
<<DANACH>>  X := X+1;
```



```
while X < Y loop X := X+1; end loop; Z:=X;
```

wird im Registerprogramm zu:

```
30: comp (≥)        vergleiche X mit Y bzgl. nicht-"kleiner"
31: jump 35         überspringe die Schleife
32: succ            X := X+1  (Schleifenrumpf)
33: load F,1        bereite einen "unbedingten" Sprung vor
34: jump 30         zurück zur Schleifen-Bedingung
35: copy Z,X        Z := X
```

In Ada lautet dieses Programmstück:

```
<<Schleife>> F := X ≥ Y;
if F then goto danach; end if;
X := X+1; goto Schleife;
<<danach>>  Z := X;
```

Ähnliche Übersetzungen von Ada-Konstrukten in einfache Programme, die nur aus Wertzuweisungen, Sprüngen und Hintereinanderausführungen (einschließlich Marken) bestehen, lassen sich leicht angeben.

Zumindest für die Kontrollstrukturen (=Anweisungsteil) sollte daher klar sein, dass man sie in eine Registermaschine übertragen kann. Bei der Übertragung der Datenstrukturen muss man sich den erforderlichen Speicherbedarf aller Teilstrukturen merken und beim Zugriff auf Variablen oder deren Komponenten entsprechende Werte in das Adressregister laden.

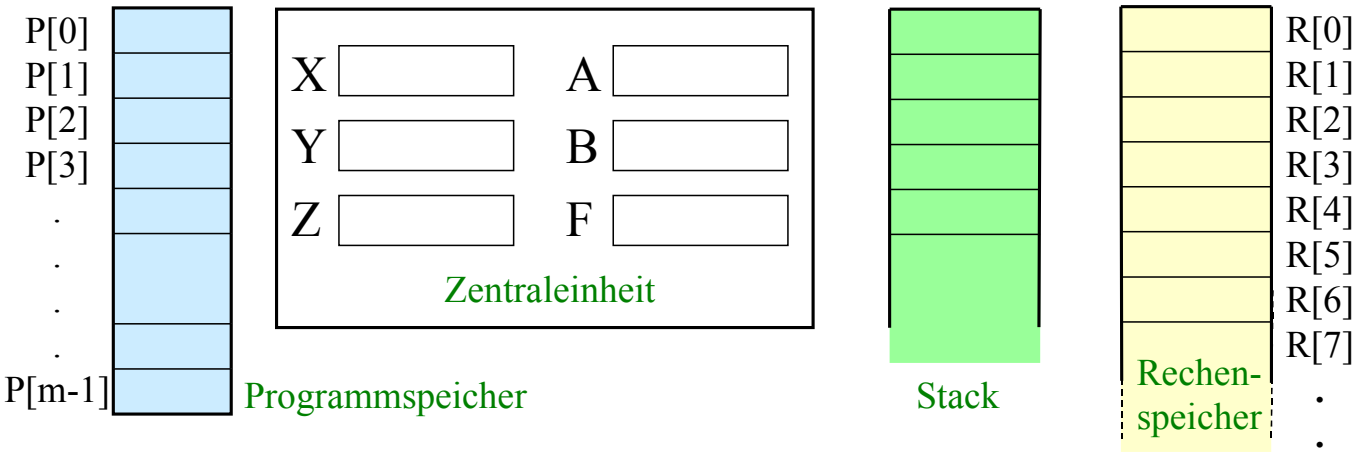
Somit ist im Prinzip klar, dass man jedes Ada-Programm in eine Registermaschine umwandeln kann, die die gleiche Resultatsfunktion berechnet. Diesen Prozess kann man teilweise automatisieren.

3.2 Stackmaschine

Wir fügen der Registermaschinen nun einen weiteren Speicher zur einfacheren Abarbeitung von Klammerstrukturen (Ausdrücke, Rekursion) hinzu. Solche Strukturen werden mit Hilfe eines Stacks (Keller, Pushdown, siehe Grundvorlesung 3.5.4, 4.3.3.a und 6.6.3) bearbeitet. Bei der Übersetzung von Programmiersprachen wird als Zwischensprache oft die Sprache einer Stackmaschine verwendet.

Im Folgenden beschreiben wir die Stackmaschine mit Hilfe der Registermaschine. Manche Befehle der RAM werden dann überflüssig, wenn man alle Berechnungen nur noch über den Stack laufen lässt und die zugrunde liegende Maschine darüber hinaus nur die Abarbeitungsreihenfolge überwacht.

Struktur der Stackmaschine (Luxusausführung)



Befehle der Stackmaschine

empty	Stack löschen
isempty	F:=1, falls Stack leer ist, sonst F:=0
pop	Lösche oberstes Stackelement
push(V)	Lege Inhalt von V auf den Stack (fehlt V, so ist das Register X gemeint)
top(V)	Kopiere oberstes Stackelement nach V (fehlt V, so ist das Register X gemeint)

addStack ≡ top(Z); pop; top(Y); pop; X:=Y+Z; push(X);

subStack, multStack, divStack, modStack *analog* ("+" durch die jeweilige Operation ersetzen)

shiftStack ≡ top; pop; shift; push; (*analog* succStack ≡ top; pop; succ; push;)

LoadStack c ≡ Load X,c; push;

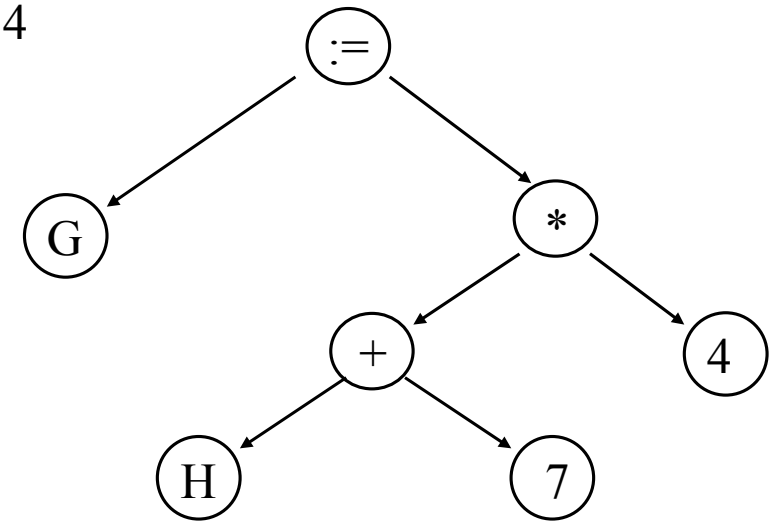
ReadStack read X; push; (*analog* WriteStack ≡ top; write X; pop; *beachte:* hier wird gelöscht)

compStack(σ) ≡ top(Y); pop; top(X); push(Y); comp(σ);

Alle weiteren Befehle wie bei der Registermaschine.

Beispiel:

G := (H + 7) * 4



A := "Adresse von H im Rechenspeicher"; ReadStack;

LoadStack 7; addStack; LoadStack 4; multStack;

A := "Adresse von G im Rechenspeicher"; WriteStack;

Der Ableitungsbaum wird *postorder* ausgeführt und legt hierdurch die Abarbeitungsreihenfolge fest.
Wertzuweisungen können also durch Stack-Befehle vollständig ersetzt werden. Somit können die auf Registern basierenden Befehle add, sub, succ, shift usw. entfallen.
Streicht man die Register bezogenen Befehle und fügt weitere Speicherstrukturen hinzu (für das Programm, den Kellerspeicher und die Halde, damit sie durch die Maschine verwaltet werden können), so erhält man eine Stackmaschine, wie sie seit Anfang der 1970er Jahre für Pascal (sog. P-Code) und später für andere Sprachen (z.B.: JVM = Java Virtual Machine) standardmäßig verwendet wird.

Die Java Virtual Machine (JVM), in die Java-Programme vor (bzw. während) der Ausführung übersetzt werden, besteht aus Stackbefehlen, z. B. (die Variable H möge in der Rechenspeicherzelle 18 stehen; Java findet die Adresse der Variablen in der Symboltabelle, siehe 3.4):

Beschreibung	unsere Darstellung	JVM-Befehl
Lade die Variable H auf den Stack	Load A,18; ReadStack	iload H
Addiere die obersten Stackelemente	addStack	iadd
Bedingte Verzweigung	comp (σ); ...	if σ ...
Sprung zum Befehl, der 5 Stellen im Programm weiter steht	< wir benutzen hier nur absolute Adressen; gemeint ist B := B+5 >	goto 5

Viele Programmiersprachen nutzen bei der Übersetzung mathematische (oder abstrakte) Maschinen aus. Hierdurch wird die Übersetzung in der Regel durchsichtiger und somit prinzipiell auch leichter verifizierbar. Beispiele:

Pascal benutzt den P-Code,

Java die JVM (Java Virtual Machine),

Smalltalk den Bytecode,

Haskell die Spineless Tagless G-Machine (STGM),

Prolog die Warren Abstract Machine (WAM).

Genaueres siehe Vorlesung über Compilerbau.

3.3 Attributierung von Grammatiken

3.3.1 Erinnerung EBNF (Grundvorlesung 1.10). Man kann diese stets in eine kontextfreie Darstellung übertragen.

EBNF Darstellung	kontextfreie Darstellung
$X ::= u_1 \mid u_2 \mid \ldots \mid u_r$	$X \to u_1$; $X \to u_2$ ...; $X \to u_r$
$X ::= uv$	$X \to uv$
$X ::= u_1 (v_1 \mid v_2) u_2$	$X \to u_1 v_1 u_2$; $X \to u_1 v_2 u_2$
$X ::= u[v]w$	$X \to uw$; $X \to uvw$
$X ::= \{u\}$	$X \to H$; $H \to \varepsilon$; $H \to uH$

3.3.2 Wiederholung (vergleiche Definition 2.7.1 der Grundvorlesung):

Eine kontextfreie Grammatik ist ein Viertupel $G = (N, T, P, S)$ mit

(1) N ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen oder Variablen),
(2) T ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $N \cap T = \emptyset$,
(3) $S \in N$ ist ein Nichtterminalzeichen (das Startsymbol),
(4) $P \subset N \times (N \cup T)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen; statt (X,u) schreibt man $X \to u$).

Auf $(N \cup T)^*$ führt man die "Ableitungsrelationen" $\Rightarrow$ und $\Rightarrow^*$ ein:

(a) Es gilt $u \Rightarrow v$ genau dann, wenn man die Wörter u und v in der Form $u = xAy$, $v = xwy$ mit $x, y \in (N \cup T)^*$ und $A \to w \in P$ schreiben kann. Man sagt: v ist aus u in einem Schritt ableitbar.

(b) Es gilt $u \Rightarrow^* v$ genau dann, wenn entweder $u = v$ ist oder wenn es Wörter $z_0, z_1, ..., z_k \in (N \cup T)^*$ für ein $k \geq 1$ gibt mit $u = z_0$, $v = z_k$ und $z_i \Rightarrow z_{i+1}$ für alle $i = 0, 1, ..., k-1$. Man sagt, v ist aus u ableitbar.

Die von G erzeugte Sprache ist die Menge der aus S ableitbaren Terminalwörter $L(G) = \{ w \in T^* \mid S \Rightarrow^* w \} \subseteq T^*$.

3.3.3 Beispielgrammatik G_1 mit $N=\{S,R,Z\}$, $T=\{ . , 0 , 1 \}$:

ausführlich wie in BNF	knappe Darstellung
<Binärzahl> → <Ziffernfolge>	S → R
<Binärzahl> → <Ziffernfolge>.<Ziffernfolge>	S → R.R
<Ziffernfolge> → <Ziffer> <Ziffernfolge>	R → ZR
<Ziffernfolge> → <Ziffer>	R → Z
<Ziffer> → 0	Z → 0
<Ziffer> → 1	Z → 1

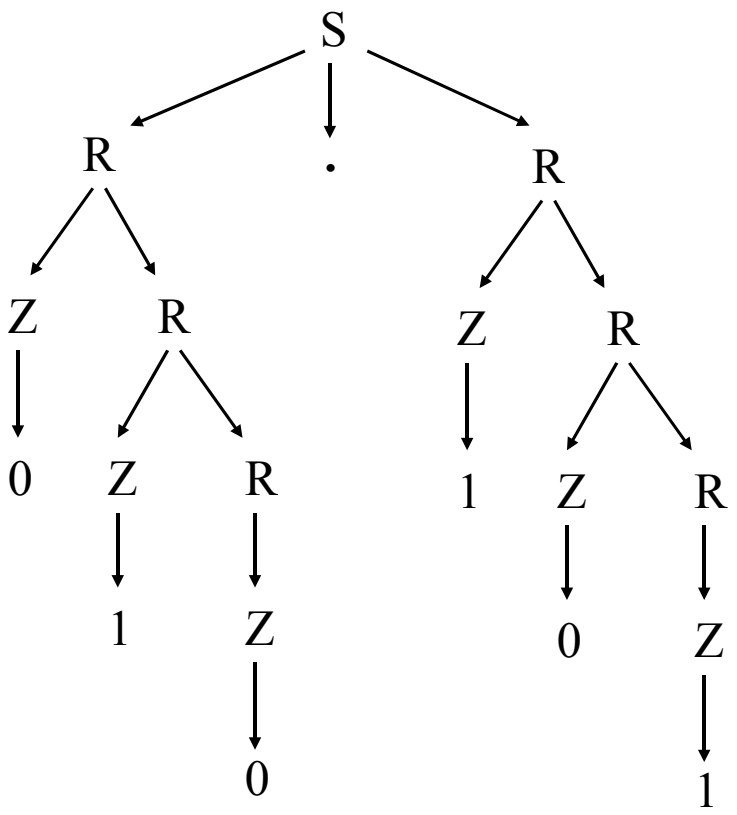
Erzeugte Sprache:

$L(G_1) = TT^* \cup TT^* \{.\} TT^*$

$= \{u \in T^* \mid u \neq \varepsilon\} \cup \{u.v \mid u,v \in T^*, u \neq \varepsilon \text{ und } v \neq \varepsilon\}$.

Man kann dies als die Menge aller binär dargestellten Zahlen auffassen (mit führenden Nullen). Zum Wort w = 010.101 gehört also die Zahl $0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 2.625$ (dezimal)

Ableitungsbaum für w = 010.101



Ziel: Wir wollen nun die Bedeutung dieses abgeleiteten Wortes w, also die Zahl 2.625, aus dem Baum berechnen.

3.3.4 Hinzunahme von Attributen. Wir ordnen jedem Zeichen aus N ∪T Attribute zu, die mit Gleichungen, welche den Produktionen der Grammatik angefügt werden, berechnet werden.

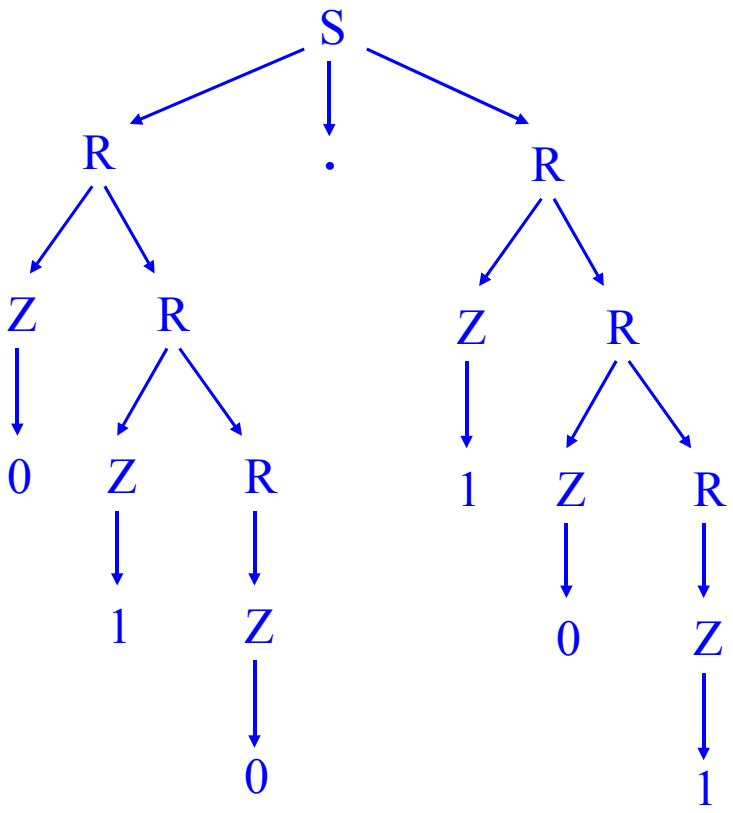
Wenn a ein Attribut für X ist, so schreiben wir X.a. Gleiche Zeichen müssen durch einen Index 1, 2, ... unterschieden werden.

Attribute in unserem Beispiel sollen sein:
S.Wert, R.Wert, Z.Wert, 0.Wert, 1.Wert, R.Länge.

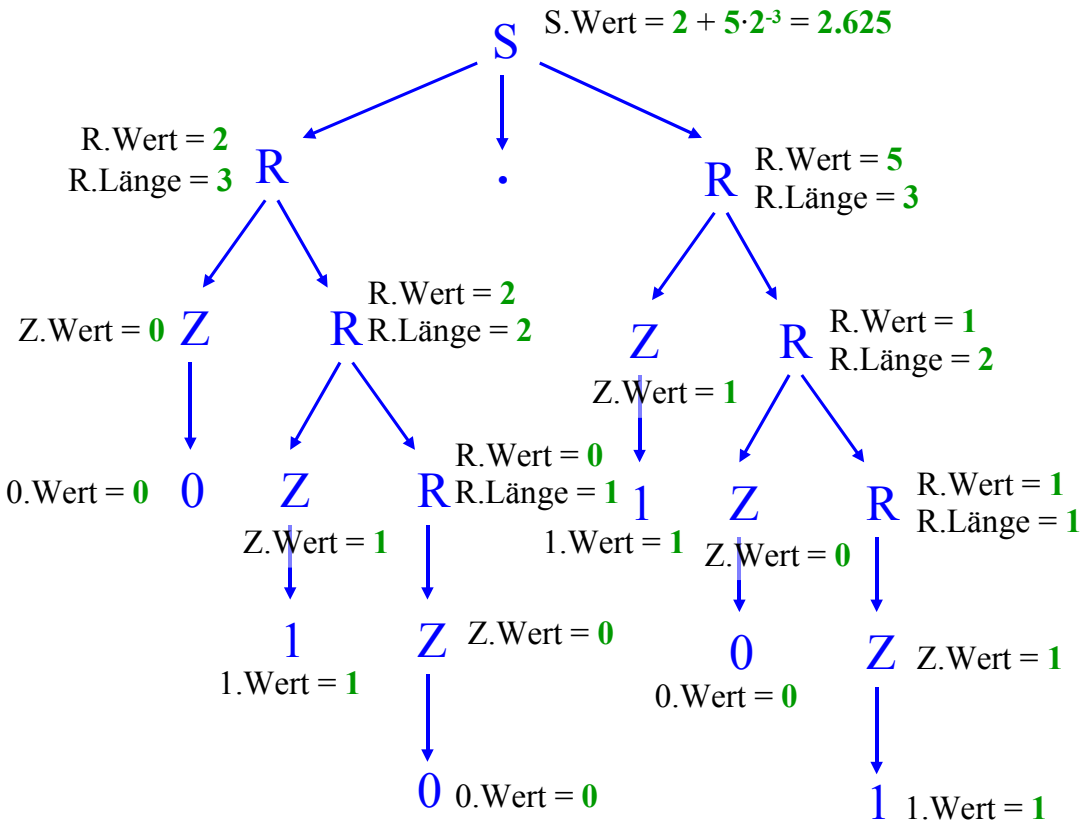
Die Gleichungen zu den Produktionen lauten dann (die Zahlen werden hier fett und grün dargestellt):

Produktion	Gleichungen
$S \to R$	$S.Wert = R.Wert$
$S \to R_1.R_2$	$S.Wert = R_1.Wert + R_2.Wert \cdot \mathbf{2}^{-R_2.Länge}$
$R_1 \to ZR_2$	$R_1.Wert = Z.Wert \cdot \mathbf{2}^{R_2.Länge} + R_2.Wert$ $R_1.Länge = R_2.Länge + \mathbf{1}$
$R \to Z$	$R.Wert = Z.Wert,\ R.Länge = \mathbf{1}$
$Z \to 0$	$Z.Wert = 0.Wert,\ 0.Wert = \mathbf{0}$
$Z \to 1$	$Z.Wert = 1.Wert,\ 1.Wert = \mathbf{1}$

Diese Gleichungen fügen wir nun an den Ableitungsbaum an



Diese Gleichungen fügen wir nun an den Ableitungsbaum an



Oft ist ein spezielles Attribut ausgezeichnet, welches die Bedeutung des abgeleiteten Wortes ist. Im Beispiel oben ist dies S.Wert .

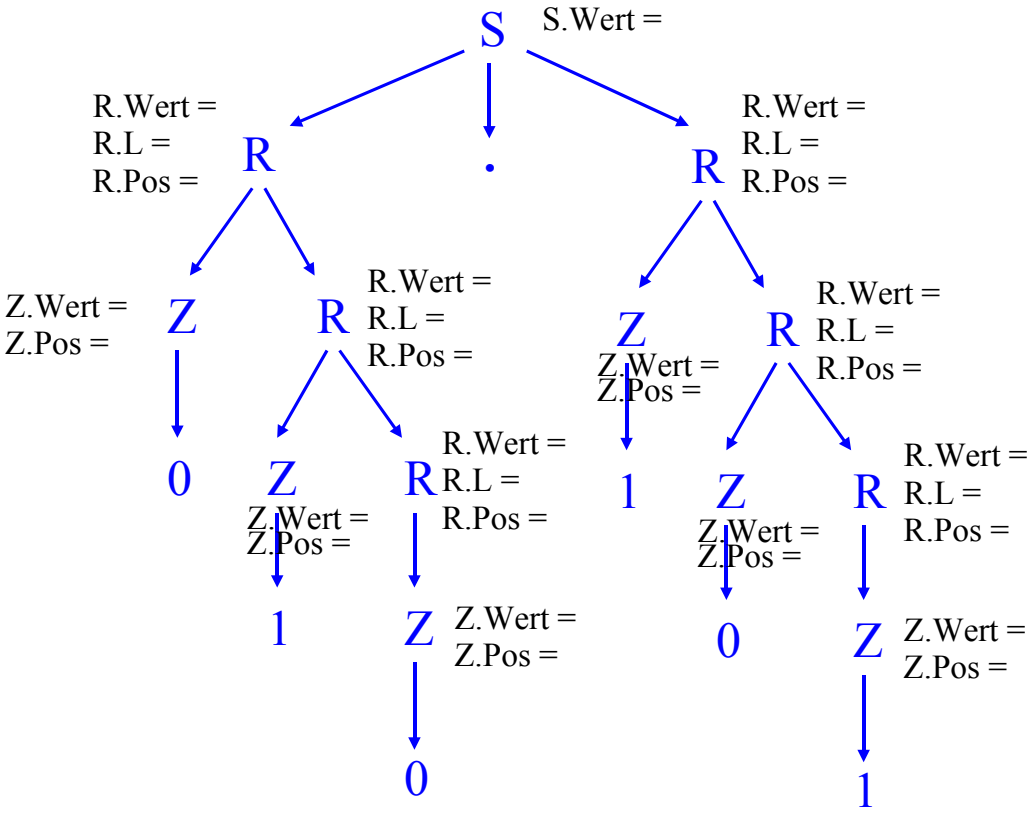
Im Beispiel wurden die Attribute stets von den Blättern aufwärts zur Wurzel hin berechnet (bottom up). Dies muss aber nicht sein. Die Attribute können auch von oben nach unten (top down) oder sowohl aufwärts wie auch abwärts ermittelt werden.
Es folgt nun ein Beispiel für eine nicht unmittelbar durchschaubare Auswertungsreihenfolge mit der gleichen Beispielgrammatik G_1, aber mit anderen Attributen.

3.3.5 Andere Attribute mit anderer Auswertungsreihenfolge

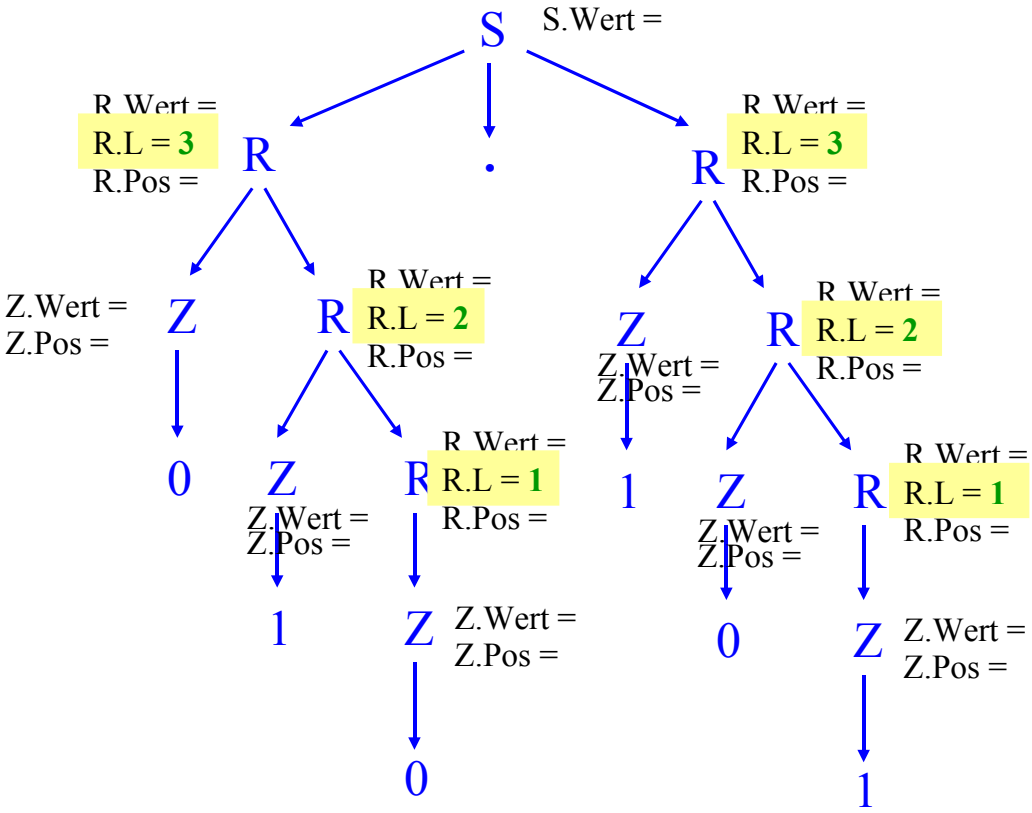
Wir verwenden die Attribute Wert, Pos (für "Position") und L (für "Länge"), und zwar S.Wert, R.Wert, R.L, R.Pos, Z.Wert und Z.Pos.

Produktion	Gleichungen
$S \to R$	$S.Wert = R.Wert$ $R.Pos = R.L - 1$
$S \to R_1.R_2$	$S.Wert = R_1.Wert + R_2.Wert$ $R_1.Pos = R_1.L - 1$ $R_2.Pos = -1$
$R_1 \to ZR_2$	$R_1.Wert = Z.Wert + R_2.Wert$ $Z.Pos = R_1.Pos$ $R_2.Pos = R_1.Pos - 1$ $R_1.L = R_2.L + 1$
$R \to Z$	$R.Wert = Z.Wert$ $Z.Pos = R.Pos$ $R.L = 1$
$Z \to 0$	$Z.Wert = 0$
$Z \to 1$	$Z.Wert = 2^{Z.Pos}$

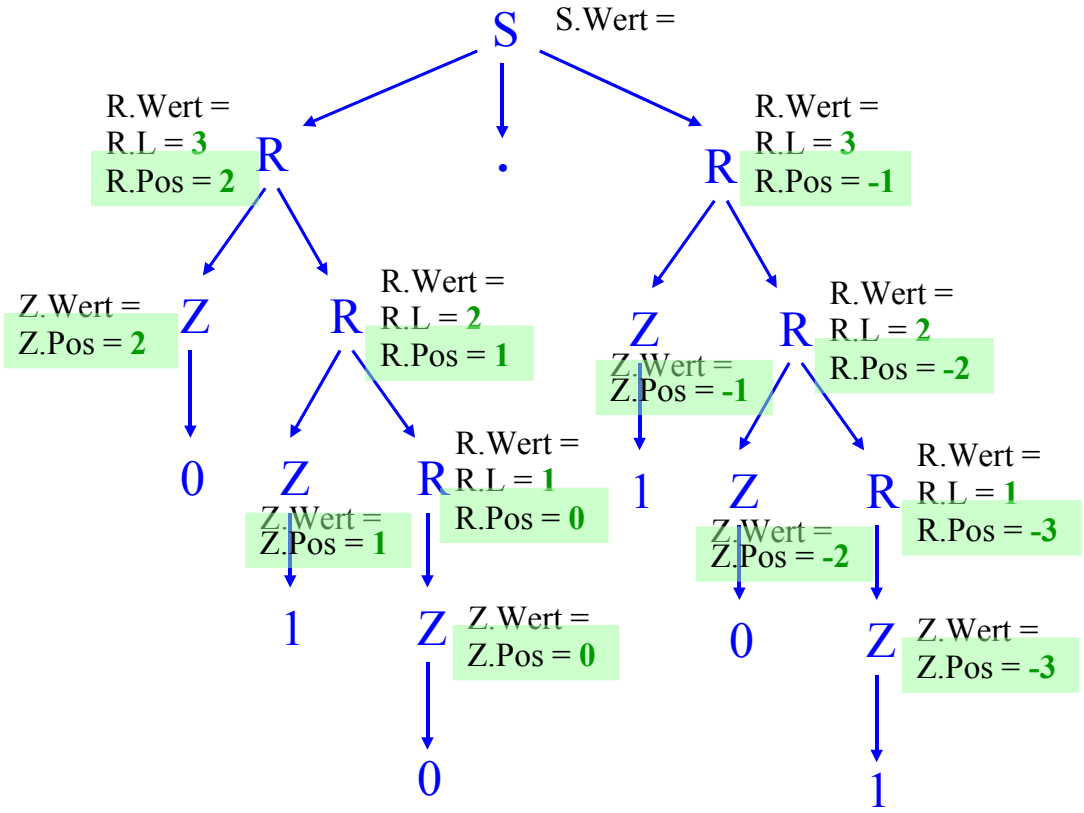
Diese Gleichungen fügen wir erneut an den Ableitungsbaum an (hier nur die linken Seiten)



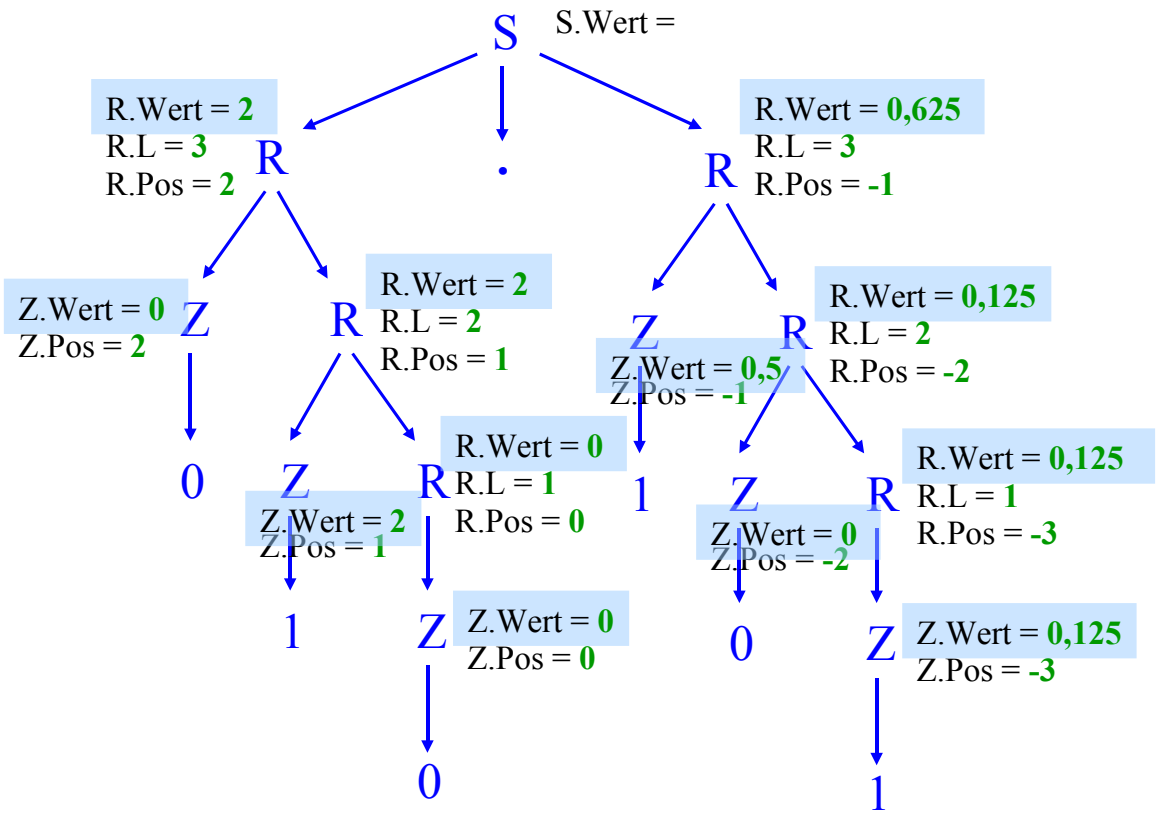
Nun berechnen wir die Attribute. Man sieht: Zunächst lässt sich die Länge L von unten nach oben berechnen:



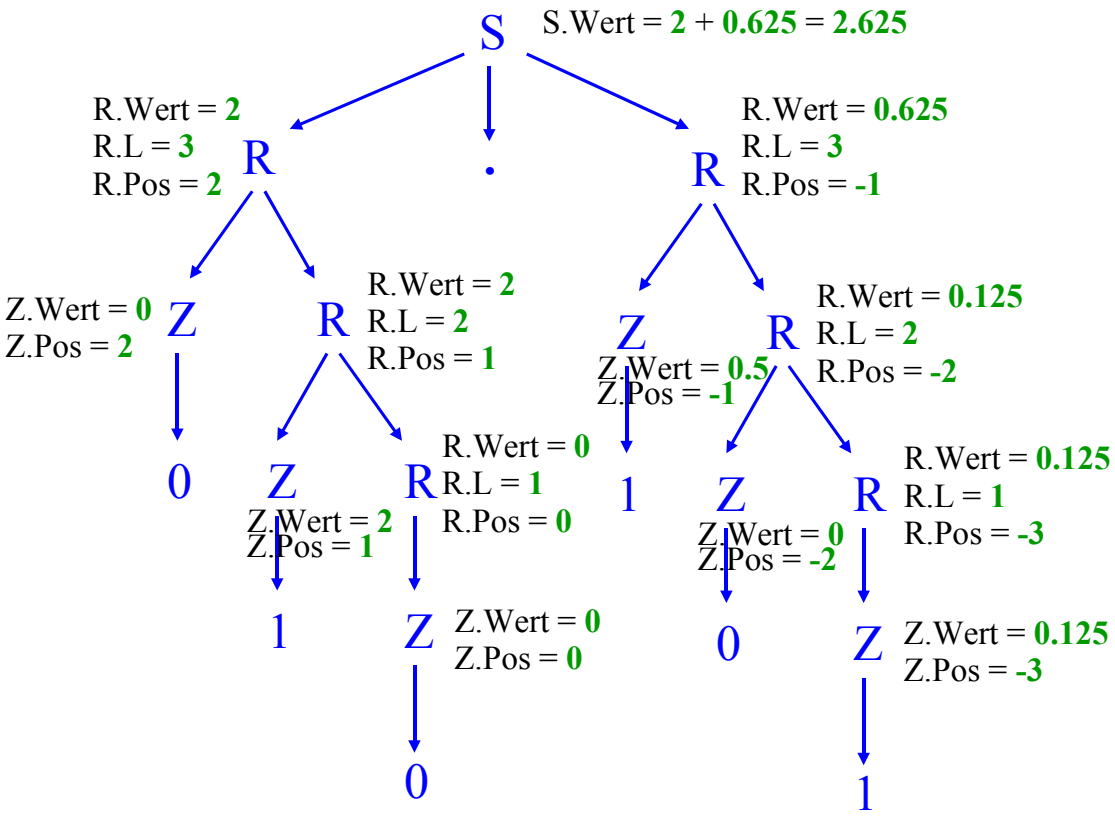
Nun kann man die Position Pos von oben nach unten berechnen:



Schließlich kann man den "Wert" von unten nach oben berechnen:



Ergebnis: S.Wert = 2.625



Definition 3.3.6: Attributierte Grammatik (Knuth, 1968)

Eine kontextfreie Grammatik G = (N, T, P, S) heißt attributierte Grammatik, wenn jedem Zeichen $Z \in N \cup T$ eine Menge $\hat{\mathbf{A}}(Z)$ (= Menge der Attribute von X), jedem Attribut $a \in \hat{\mathbf{A}}(Z)$ eine (Werte-) Menge W_a und jeder Produktion $X \to Y_1 Y_2 ... Y_m \in P$ eine Menge von Abbildungen mit folgenden Eigenschaften zugeordnet ist.

(1) Jedes $\hat{\mathbf{A}}(Z)$ lässt sich in zwei disjunkte Teilmengen zerlegen:
$\hat{\mathbf{A}}(Z) = \hat{\mathbf{A}}_0(Z) \cup \hat{\mathbf{A}}_1(Z)$ mit $\hat{\mathbf{A}}_0(Z) \cap \hat{\mathbf{A}}_1(Z) = \varnothing$.

(2) Für jedes $a \in \hat{\mathbf{A}}_0(X)$ existiert genau eine Abbildung

$$f_a^X: W_{a_1} \times W_{a_2} \times ... \times W_{a_r} \to W_a \quad (r \geq 0)$$

und für jedes i = 1, 2, ..., m und jedes $a \in \hat{\mathbf{A}}_1(Y_i)$ existiert genau eine Abbildung $f_a^{Y_i}: W_{a_1} \times W_{a_2} \times ... \times W_{a_r} \to W_a \quad (r \geq 0)$

mit $a_1, ..., a_r \in \hat{\mathbf{A}}(X) \cup \hat{\mathbf{A}}(Y_1) \cup \hat{\mathbf{A}}(Y_2) \cup ... \cup \hat{\mathbf{A}}(Y_m)$, d. h. dies sind nur Attribute von Zeichen, die in der Produktion vorkommen.
(f_a^X und $f_a^{Y_i}$ werden meist als Gleichungen geschrieben; diese werden von rechts nach links ausgewertet.)

Die Attribute aus $\hat{\mathbf{A}}_0(Z)$ bezeichnet man als zusammengesetztes oder synthetisches oder synthetisiertes Attribut (englisch: synthesized).

Die Attribute aus $\hat{\mathbf{A}}_1(Z)$ bezeichnet man als ererbtes oder vererbtes oder inherites Attribut (englisch: inherited).

Zu einem Wort w berechnet man den Syntaxbaum $S \Rightarrow^* w$, ordnet den Zeichen im Baum ihre Attribute zu und rechnet diese mit Hilfe der Regeln aus. Um genau eine Bedeutung zu erhalten, sollte die Grammatik "eindeutig" sein (siehe Grundvorlesung Definition 2.7.11), d.h., zu jedem ableitbaren Wort darf es nur genau einen Syntaxbaum geben.

Im Allgemeinen zeichnet man ein Attribut des Startsymbols S aus, das die Bedeutung des abgeleiteten Wortes angibt.

3.3.7 Problem: Gibt es stets eine eindeutige Auswertung der Attribute? (Genaueres: Vorlesung über Compilerbau)

Im Allgemeinen nein. Es können zyklische Definitionen auftreten. Zum Beispiel schon bei einer Produktion:

$S \rightarrow XY$ $\qquad$ $S.L = Y.Pos + X.L$

$\qquad\qquad\qquad$ $Y.Pos = S.L$

Eine attributierte Grammatik heißt "zulässig" (oder "wohldefiniert"), wenn sich alle Attribute für jeden Syntaxbaum $S \Rightarrow^* w$ stets auswerten lassen.

Zwar ist das Problem, ob eine kontextfreie Grammatik eindeutig ist, algorithmisch nicht entscheidbar, jedoch kann man die Eigenschaft der Zulässigkeit entscheiden, wie man sich folgendermaßen klar machen kann.

Vorgehensmodell hierfür: Gegeben ein Wort w der Sprache. Konstruiere hierzu einen Syntaxbaum $S \Rightarrow^* w$. An jedes Zeichen Z im Baum füge man die Attribute $a \in \hat{A}(Z)$ an. Sodann geben die jeweiligen Produktionen an, wie die Attribute zu berechnen sind. Nun konstruiere man einen Graphen, der die Paare

(Knoten des Syntaxbaums, zugehöriges Attribut)

als Knoten besitzt. Man ziehe eine Kante von (X,a) nach (Y,b) genau dann, wenn (auf Grund der hier benutzten Produktion) für die Berechnung des Wertes des Attributs $a \in \hat{A}(X)$ das Attribut $b \in \hat{A}(Y)$ benötigt wird.

Die Attributberechnung ist in diesem Syntaxbaum genau dann möglich, wenn dieser Graph azyklisch ist und jeder Knoten, der keine einlaufende Kante besitzt, auf Grund der Regeln eine Konstante als Wert zugewiesen bekommt.

Nun muss man sich noch überlegen, dass man mit endlich vielen Wörtern w für jede Grammatik auskommt (nämlich Wörter, deren Syntaxbäume keine Teil-Wiederholungen enthalten, vgl. Theorie-Vorlesungen). Folglich kann man das Problem prinzipiell entscheiden.

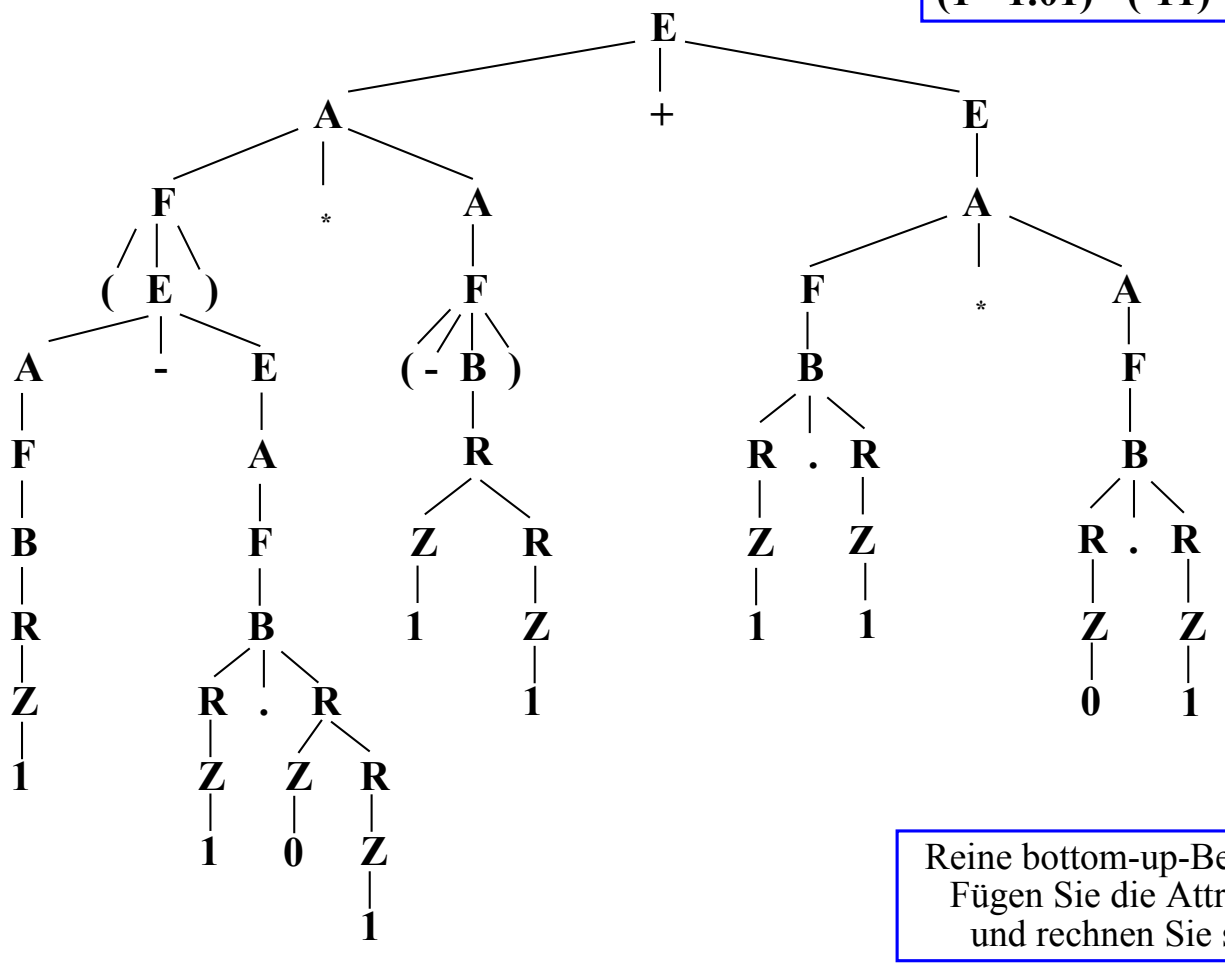
3.3.8 Arithmetische Ausdrücke

"expression" E, "additiver Term" A, "Faktor" F, "Binärzahl" B, "rationale Zahl" R, "Ziffer" Z; $N = \{A, B, E, F, R, Z\}$, $T = \{ 0, 1, +, -, *, (,) \}$, Startsymbol ist E.

Produktionen	Gleichungen
$E \to A$	$E.Wert = A.Wert$
$E_1 \to A + E_2$	$E_1.Wert = A.Wert + E_2.Wert$
$E_1 \to A - E_2$	$E_1.Wert = A.Wert - E_2.Wert$
$A \to F$	$A.Wert = F.Wert$
$A_1 \to F * A_2$	$A_1.Wert = F.Wert \cdot A_2.Wert$
$F \to B$	$F.Wert = B.Wert$
$F \to (- B)$	$F.Wert = - B.Wert$
$F \to (E)$	$F.Wert = E.Wert$
$B \to R$	$B.Wert = R.Wert$
$B \to R_1.R_2$	$B.Wert = R_1.Wert + R_2.Wert \cdot 2^{-R_2.Länge}$
$R_1 \to ZR_2$	$R_1.Wert = Z.Wert \cdot 2^{R_2.Länge} + R_2.Wert$
	$R_1.Länge = R_2.Länge + 1$
$R \to Z$	$R.Wert = Z.Wert$, $R.Länge = 1$
$Z \to 0$	$Z.Wert = 0$
$Z \to 1$	$Z.Wert = 1$

Die Grammatik ist eindeutig.

Berechne den Wert von
(1 - 1.01) * (-11) + 1.1 * 0.1



Reine bottom-up-Berechnung.
Fügen Sie die Attribute an
und rechnen Sie sie aus!

Auswertung postorder mit
einer Stackmaschine

Berechne den Wert von
(1 - 1.01) * (-11) + 1.1 * 0.1

Die Folge von Stackoperationen:

push("+"); push("*"); push("-");
push(1); push("+"); push(1); push("/"); push("+"); push("*");
push(0); push(2 hoch 1); auswerten; push(1); auswerten;
push(2 hoch 2); auswerten; auswerten; auswerten; push("-");
push("+"); push(1); push("*"); push(1); push("2 hoch 1");
auswerten; auswerten; unär auswerten; auswerten; push("*");
push("+"); push(1); push("/"); push(1); push(2 hoch 1);
auswerten; auswerten; push("+"); push(0); push("/"); push(1);
push(2 hoch 1); auswerten; auswerten; auswerten; auswerten;

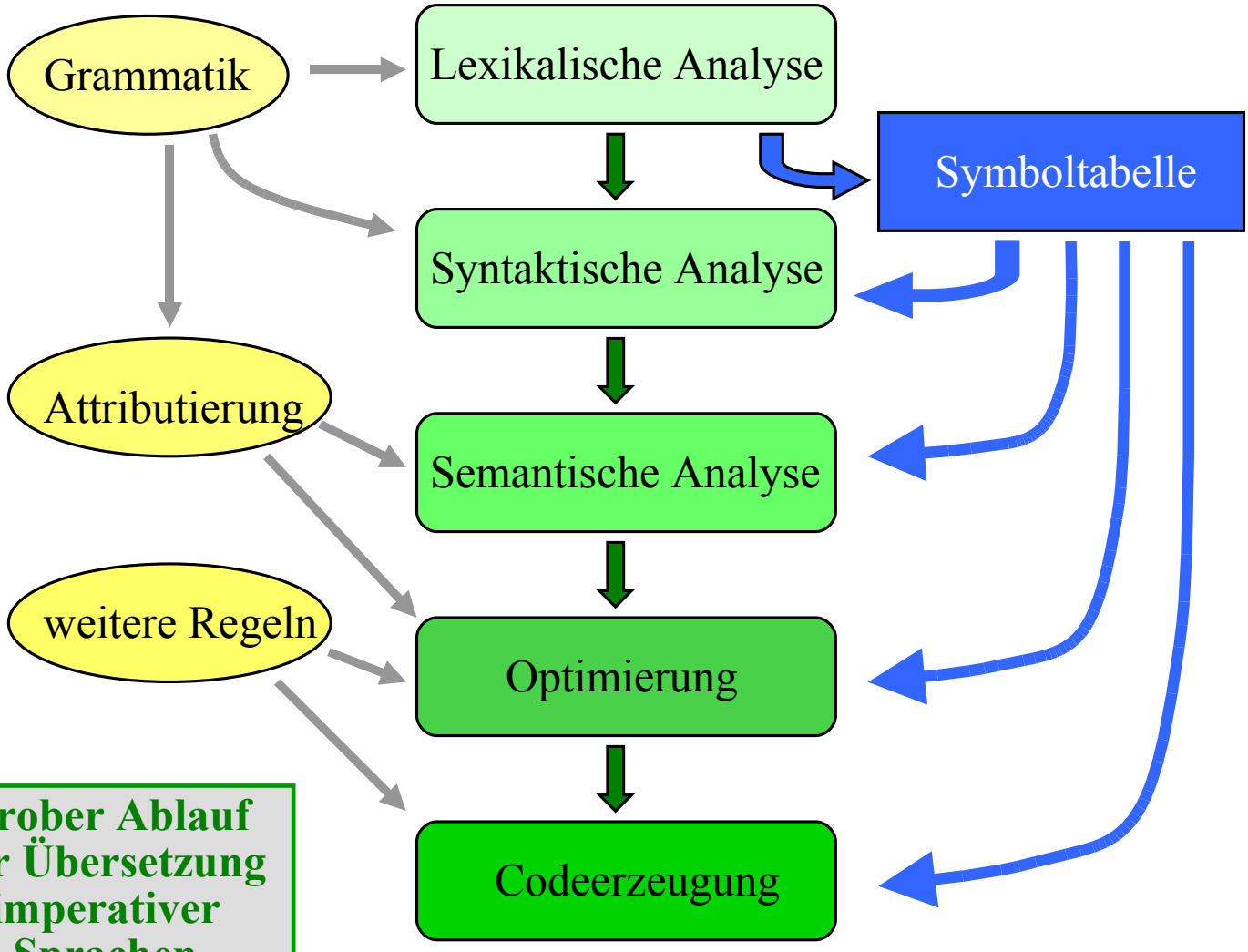
blau = die fünf Operatoren, die im Ausdruck stehen und ihre zugehörige Auswertung; alle übrigen Operationen berechnen die Zahlen.

3.4 Prinzip der Übersetzung

3.4.1 Allgemeines zur Übersetzung von Sprachen

Wir haben in Abschnitt 3.1 gesehen, dass man gewisse Teile von Ada in den Machinencode einer Register- oder einer Stackmaschine übertragen kann. Man überlegt sich, dass dies prinzipiell für die gesamte Sprache Ada, aber auch für jede andere imperative Programmiersprache gilt.

Die Syntax einer Programmiersprache enthält gewisse Teile, die man vorab leicht identifizieren kann (sog. reguläre Anteile, die man mit einem "endlichen Automaten" erkennen kann). Dies sind vor allem die "lexikalischen Einheiten", siehe Abschnitt 3.2 der Grundvorlesung. Dabei kann man zugleich die verwendeten Identifikatoren (Namen, Konstanten, strings, Wortsymbole) in eine besondere Tabelle, die sog. Symboltabelle, schreiben. Der grobe Ablauf bei der Übersetzung lautet:



Grober Ablauf der Übersetzung imperativer Sprachen

3.4.2 Was schreibt man in die Symboltabelle?

Grundsätzlich werden hier alle Symbole, die der Programmierer selbst eingeführt hat, notiert, insbesondere die Typen und die Variablen; zugleich legt man fest, wie diese Symbole auf die "Zielmaschine", also in deren Maschinencode, abgebildet werden. Bei einer Variablen gibt man z. B. an, in oder ab welcher Rechenspeicherzelle sie stehen und wie viele Byte (oder Speicherwörter) sie belegen wird. Bei einem Datentyp gibt man an, wie viele Speicherplätze man insgesamt benötigt, an welchen Stellen hierbei die einzelnen Selektoren stehen und welche elementaren Datentypen jeweils vorliegen (dies braucht man zum Beispiel, um bei der Übersetzung zu prüfen, ob Operatoren in Ausdrücken richtig verwendet werden). An einem Beispiel skizzieren wir kurz das Konzept, das in jedem Compiler in irgendeiner Variante benutzt wird.

3.4.3 Beispielprogramm (Demonstration, ohne konkreten Sinn)

```
procedure BSP is
type RAT is record Z: Integer; N: Natural; end record;
type XX is record N: array (1..20) of Character;
     L: Float; G: Boolean; R: RAT; end record;
A,B,C: RAT;  Y: XX;
begin
     GET(A.Z); GET(A.N); B:=A;
     C.Z:=A.Z*(A.N+B.Z)-4; C.N:=B.N;
     if A.N > 3 then Y.N(2) := 'E'; else Y.G:= true; end if;
     PUT (A.N);
end;
```

3.4.4 Die lexikalische Analyse ermittelt die lexikalischen Einheiten und legt die Bedeutung von Bezeichnern und deren Zuordnung zum Speicher in der Symboltabelle ab.

Laut 3.2.2 sind dies in unserem Beispiel:
Bezeichner: BSP, RAT, RAT.Z, RAT.N, XX, XX.N, XX.L, XX.G, XX.R, A, B, C, Y, GET, PUT
Literale: 4, 3, 'E', true
Begrenzer: : ; .. ( ) := * + - >
Reservierte Wörter: procedure is type record end array of begin if then else end

Hinzu kommen Trennzeichen und Kommentare, die beim ersten Lesen des Programmtextes unmittelbar durch interne Darstellungen ersetzt oder überlesen werden. Das Ergebnis der lexikalischen Analyse könnte sein:

Nr.	Identifikator	Art	Typ	Beginn R[..]	Länge in Byte	Bezug (Nr)	... ...
1	BSP	PROC	-	0	?	-	
2	RAT	Typ	-	-	8	-	
3	RAT.Z	Selektor	Integer	0	4	2	
4	RAT.N	Selektor	Natural	4	4	2	
5	XX	Typ	-	-	34	-	
6	XX.N	Selektor	array	0	20	5	
7	XX.N( )	Typ	Char	-	1	6	
8	XX.L	Typ	Float	-	4	5	
9	XX.G	Typ	Bool	-	1	5	
10	XX.R	Typ	RAT	-	8	5	
11	A	Var	RAT	40	8	2	
12	B	Var	RAT	48	8	2	
13	C	Var	RAT	56	8	2	
14	Y	Var	XX	64	34	5	
15	GET	FKT	-	-	-	-	
16	4	Konst	Natural	-	4	-	
17	3	Konst	Natural	-	4	-	
18	E	Konst	Char	-	1	-	
19	true	Konst	Bool	-	1	-	
20	PUT	FKT	-	-	-	-	

"-" ist als "nicht zutreffend" zu lesen.
"?" bedeutet, dass die Eintragung erst später erfolgt.
Wir nehmen hier an, dass die Variablen ab Rechenspeicherzelle 40 stehen und in der Reihenfolge, wie sie im Programm aufgeführt sind, abgespeichert werden. Die Konstanten c fassen wir wie Variablen auf, die aber keinen eigenen Rechenspeicherplatz erhalten, sondern direkt aus der Symboltabelle in die jeweiligen Anweisungen LOAD V,c später eingesetzt werden.

Machen Sie sich klar, dass man diese Tabelle mit nur einem Durchlauf durch das Programm aufstellen kann. Natürlich muss in einer "richtigen Symboltabelle" noch mehr stehen. Z. B. ist anzugeben, in welchem Block man sich befindet, wie der Indextyp bei Feldern lautet, an welchen Stellen in der Bibliothek man die global definierten Funktionen (GET, PUT, ...) findet usw. Dies hängt von der zu übersetzenden Programmiersprache ab.

3.4.5 Die weiteren Phasen

Die syntaktische Analyse ermittelt aus dem Programm den Syntaxbaum. Hierzu gibt es Standardalgorithmen für beliebige kontextfreie Grammatiken, die allerdings eine Laufzeit von $O(n^3)$ besitzen, wobei n die Länge des Programmtextes ist. Dies dauert für die Praxis zu lange.

Man schränkt daher die kontextfreien Grammatiken auf solche eindeutige Grammatiken ein, deren Syntaxanalyse nur lineare Zeit erfordert (Stichwörter: LL, LR(1), LALR(k)).

Der Syntaxbaum wird anschließend mit Methoden, wie sie in Abschnitt 3.3 besprochen wurden, in ein "Zielprogramm" übersetzt. Während dieser Phase und/oder anschließend wird der übersetzte Code optimiert (z.B. Entfernen überflüssiger Befehle, bessere Nutzung der Register, Ersetzen durch eine kürzere gleichwertige Befehlsfolge).

Das übersetzte Programm in den Code der Stackmaschine aus Abschnitt 3.2 würde (ohne Optimierung) also lauten:

```
 0: IN (A.Z);                 1: IN (A.N);
 2: ReadStack A.Z;            3: WriteStack B.Z;
 4: ReadStack A.N;            5: WriteStack B.N;
 6: ReadStack A.Z;            7: ReadStack A.N;
 8: ReadStack B.Z;            9: addStack;
10: multStack;               11: LoadStack 4;
12: subStack;                13: WriteStack C.Z;
14: ReadStack B.N;           15: WriteStack C.N;
16: ReadStack A(N);          17: LoadStack 3;
18: compStack (<=);          19: jump 24;
20: LoadStack 'E';           21: WriteStack Y.N(2);
22: Load F,1;                23: jump 26;
24: LoadStack true;          25: WriteStack Y.G;
26: OUT (A.N);               27: stop
```

IN und OUT ist die maschinenabhängige Ein-/Ausgabe. Die Adressen der Variablen entnimmt man der Symboltabelle und setzt sie hier ein.

3.4.6 Compiler-Compiler

Die Ziele dieser theoretischen Überlegungen sind zum einen Techniken, um einen Compiler für eine konkrete Programmiersprache zu schreiben, und zum anderen Methoden, um einen solchen Compiler automatisch aus einer attributierten Grammatik erzeugen zu lassen. Ein Programm, das dieses leistet, nennt man Compiler-Compiler.

Hierzu muss man eine eindeutige kontextfreie Grammatik für die zu übersetzende Programmiersprache zugrunde legen, die eine rasche Syntaxanalyse erlaubt. Diese muss man mit Attributen versehen, deren Auswertung den übersetzten Code eines Programms liefert. An folgendem Beispiel für Wertzuweisungen wird dies klar. Es gibt hier nur ein Attribut "Code".

Produktion	Gleichungen
$W \rightarrow V := E$	W.Code = **WriteStack** V.Code **;**
$E_1 \rightarrow A + E_2$	E_1.Code = A.Code E_2.Code **addStack ;**
$E \rightarrow A$	E.Code = A.Code
$A_1 \rightarrow F * A_2$	A_1.Code = F.Code A_2.Code **multStack ;**
$A \rightarrow F$	A.Code = F.Code
$F \rightarrow (E)$	F.Code = E.Code
$F \rightarrow B$	F.Code = **LoadStack** B.Code **;**
$F \rightarrow V$	F.Code = **ReadStack** V.Code **;**

Die Terminalzeichen sind fett grün dargestellt. In den Produktionen steht V für eine Variable und B für eine binär dargestellte natürliche Zahl. Negative Zahlen kann man durch Hinzunahme eines Komplementbefehls "Negiere oberstes Stackelement" oder durch "0 minus Stackelement" behandeln. Das Attribut W.Code liefert dann das übersetzte Programm auf einer Stackmaschine.

Man muss die Befehle noch durchnummerieren, über die Symboltabelle die korrekte Verwednung der Operatoren prüfen usw. Aber im Prinzip müsste klar sein, dass man solche Compiler-Compiler bauen kann.

3.4.7 Schlussbemerkung: Sie erkennen an diesem Beispiel auch, wozu man Theorie braucht.

Die Theorie liefert den Formalismus für Grammatiken und ihre Attribute, sie beschreibt unzweideutig die Auswertungen und die Bedeutungen, und sie beweist die Korrektheit und Terminierung.

Dabei entsteht eine Fülle von Erkenntnissen und Spezialisierungen. Zum Beispiel die genannte eindeutige Grammatik, vor allem LALR(1). Aber auch die Attributierungen müssen aus Effizienzgründen beschränkt werden. Eine Attributierung, die nur synthetisierte Attribute besitzt, heißt *S-Attributierung*. In der Praxis genügt die *L-Attributierung*, bei der sich jedes Attribut eines Nichtterminals X durch die Attribute der links davon stehenden Nichtterminale berechnen lässt, usw. usw.