Gliederung des Kapitels 1.1

1.1 Algorithmen und Sprachen

1.1.7 Sprachen zur Beschreibung von Sprachen

Eine Sprache, mit der man eine andere Sprache beschreiben kann, nennt man Metasprache. Dabei muss man zwischen mehreren "Ebenen" unterscheiden:

Ebene "Syntax": Hier geht um den korrekten Aufbau der Wörter bzw. Sätze einer Sprache. Der oft zitierte Satz

"Der Tisch ist ein Säugetier."

ist syntaktisch (oder grammatikalisch) korrekt, völlig unabhängig davon, ob er inhaltlich zutrifft.

Kritischer ist dies bei einem Satz wie

"Nachts ist es kälter als draußen."

da es möglicherweise von der deutschen Grammatik her nicht zulässig ist, eine Temperaturangabe mit einer Ortsangabe zu vergleichen. Dennoch klingt der Satz syntaktisch korrekt.

Die formal richtige Anordnung der Sprachelemente lässt sich insbesondere durch Grammatiken oder gleichwertige Kalküle festlegen. Dies wurde in der Informatik in den letzten vierzig Jahren gut untersucht und hier liegen zugleich umfangreiche Erfahrungen aus der Praxis vor.

Will man die Syntax von Programmiersprachen oder anderen Sprachen im mathematisch-technisch-naturwissenschaftlichen Bereich beschreiben, so verwendet man spezielle kontextfreie Grammatiken, die folgende Eigenschaft besitzen: Ist der Aufbau der Wörter bzw. Sätze einer Sprache hierin formuliert, so kann man automatisch einen Algorithmus (einen sog. "Parser") erzeugen lassen, der das Wortproblem in relativ kurzer Zeit löst, d.h., der zu *jedem* Wort über dem Terminalalphabet feststellt, ob es zur Sprache gehört oder nicht, und dies in einer Zeitspanne, die proportional zur Länge des Wortes ist.

Ebene "Semantik": Die Semantik ordnet jedem Wort bzw. Satz einer Sprache seine Bedeutung zu. Die Bedeutung kann recht unterschiedlich sein: Die Bedeutung von Sätzen der natürlichen Sprachen ist meist eine Handlung oder eine Information; bei mathematisch-naturwissenschaftlichen Aussagen geht es meist um deren inhaltliche Korrektheit, "logische Gültigkeit" und Widerspruchsfreiheit; bei Programmiersprachen weist die Semantik jedem Programm seine realisierte Abbildung zu.

Eine genaue Semantik ist unverzichtbar für die Sicherheit von Software. Zum einen muss das Programm korrekt sein, also genau das durchführen, was (in einer sog. "Spezifikation") vorgegeben ist, zum anderen muss es zuverlässig arbeiten, also z.B. gegenüber Veränderungen der Semantik, bedingt durch eine neue Umgebung, stabil sein.

Mit der Semantik ist der Begriff der Programm-"Äquivalenz" verbunden: Zwei Wörter einer Sprache heißen äquivalent, wenn sie die gleiche Bedeutung besitzen. In der Praxis möchte man oft ein Programm durch ein anderes ersetzen, das das Gleiche leistet, aber in irgendeiner Hinsicht besser ist. Hierfür sucht man nach Verfahren, die ein Programm automatisch nach vorgegebenen Kriterien optimieren.

Die beiden Programmstücke (var X, Y, i, A, B, C: natural)

(1) i:=0; while i < X do X:=X+Y; Y:=X+Y; i:=i+1 od

(2) A:=1; B:=1; for i:=2 to 2*X do C:=A+B, A:=B; B:=C od;
if X>0 then C:=(B-A)*X+A*Y; Y:=A*X+B*Y; X:=C fi

sind äquivalent *(wirklich?)*, wenn man nur die Variablen X und Y betrachtet. In der Semantik untersucht man u.A., wie man Äquivalenzen beweisen oder wie kann man aus dem einen das andere Programm automatisch erzeugen lassen kann.

Ebene "Pragmatik": Die Pragmatik untersucht und beschreibt die Beziehungen zwischen der Sprache und deren Benutzern (Menschen, Maschinen) bzw. der jeweiligen Umwelt. Dies können einfache Fragen der Auswirkungen von Wörtern bzw. Sätzen sein, aber auch verwickelte Zusammenhänge zwischen Interessen, die man mit der Benutzung einer Sprache verfolgt.

Die Pragmatik führt aus dem Bereich der über Alphabeten aufgebauten Sprachen hinaus ("welche Wirkung soll ein Satz erzielen?"), wirkt aber auch in sie von außen hinein, indem sie z.B. die Begründung für Sprachelemente oder Darstellungen von Daten ist ("wir führen arrays ein, weil man hiermit die in der Technik verwendeten Vektoren wiedergeben kann").

Der Pragmatik wurde in der Informatik bisher wenig Aufmerksamkeit gewidmet; dies ändert sich aber zurzeit.

Fast jede natürliche Sprache (Deutsch, Chinesisch, Latein, Englisch usw.) kann als Metasprache verwendet werden, z.B., indem wir mit dieser Sprache eine andere natürliche Sprache beschreiben und erlernen (Fremdsprachenunterricht).

Natürliche Sprachen besitzen die Eigenschaft, *sich selbst* beschreiben zu können (Muttersprachunterricht einschl. Grammatikkunde). Diese Eigenschaft finden wir bei vielen künstlichen Sprachen auch: Oft lässt sich in einer Sprache die Sprache selbst beschreiben; insbesondere kann man hiermit die Sprache erweitern, indem man die Syntax und die Bedeutung der zusätzlichen Sprachelemente in der Sprache selbst formuliert. Natürliche Sprachen und Programmiersprachen können sich auf diese Weise ständig weiterentwickeln und neue Gebiete der Beschreibung und Bearbeitung erschließen.

Wir demonstrieren diese "Selbstbeschreibungsfähigkeit" an einem Beispiel, indem wir kontextfreie Grammatiken mit Hilfe der EBNF erzeugen (die EBNF ist selbst wieder eine kontextfreie Grammatik, siehe Abschnitt 1.1.6).

Hierzu müssen wir Grammatiken als Wörter einer Sprache aufschreiben. Die Grammatik G_1 aus Abschnitt 1.1.5

$G_1 = (V_1, \Sigma_1, P_1, S_1)$ mit $V_1 = \{S_1\}$,
$\Sigma_1 = \{0, 1\}$, $P_1 = \{(S_1, 1), (S_1, S_1 0), (S_1, S_1 1)\}$

kann man als folgendes Wort der Länge 25

S_1;0,1;(S_1,1)(S_1,$S_1$0)(S_1,$S_1$1);S_1 $\in$ A*

über dem Alphabet A = { '**(**' , '**)**' , '**,**' , '**;**' , '**0**' , '**1**' , '**S_1**' }
auffassen.

Auf diese Weise erzeugen wir nun (kontextfreie) Grammatiken aus einer EBNF.

Beispiel 1.1.7.1: Folgende EBNF (V_G, Σ_G, P_G, <Grammatik>) umfasst die Sprache aller kontextfreien Grammatiken, deren Terminalzeichen alle von der Form Tz und deren Nichtterminalzeichen alle von der Form Nz sind mit z∈ {1}{0,1}*:

V_G = {<Ntz>, <Tz>, <P>, <rSeite>, <Grammatik>},

Σ_G = { '(' , ')' , ',' , ';' , '0' , '1' , 'N' , 'T' },

Regeln:

<Grammatik> ::=
    <Ntz> {',' <Ntz>} ';' <Tz> {',' <Tz>} ';' <P> ';' <Ntz>

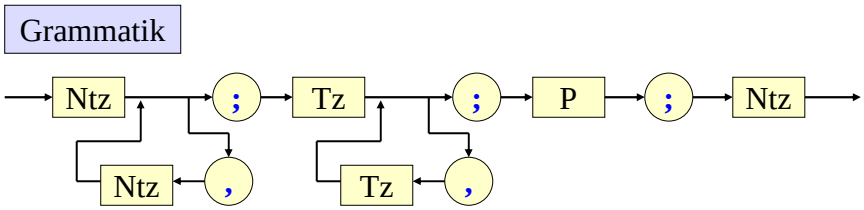
<Ntz> ::= 'N' '1' {'0' | '1'}

<Tz> ::= 'T' '1' {'0' | '1'}

<P> ::= { '(' <Ntz> ',' <rSeite> ')' }

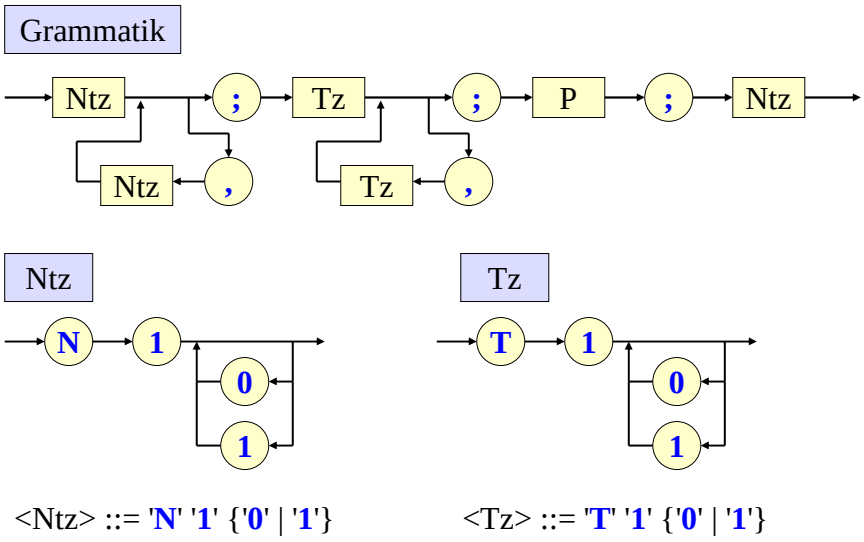
<rSeite> ::= { <Ntz> | <Tz> }

Syntaxdiagramme hierzu zeichnen:



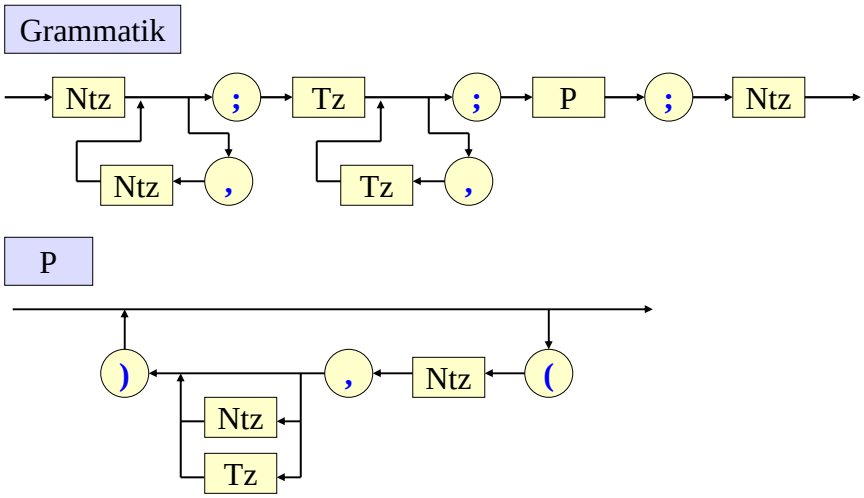
<Grammatik> ::=
    <Ntz> {',' <Ntz>} ';' <Tz> {',' <Tz>} ';' <P> ';' <Ntz>

Syntaxdiagramme hierzu zeichnen:

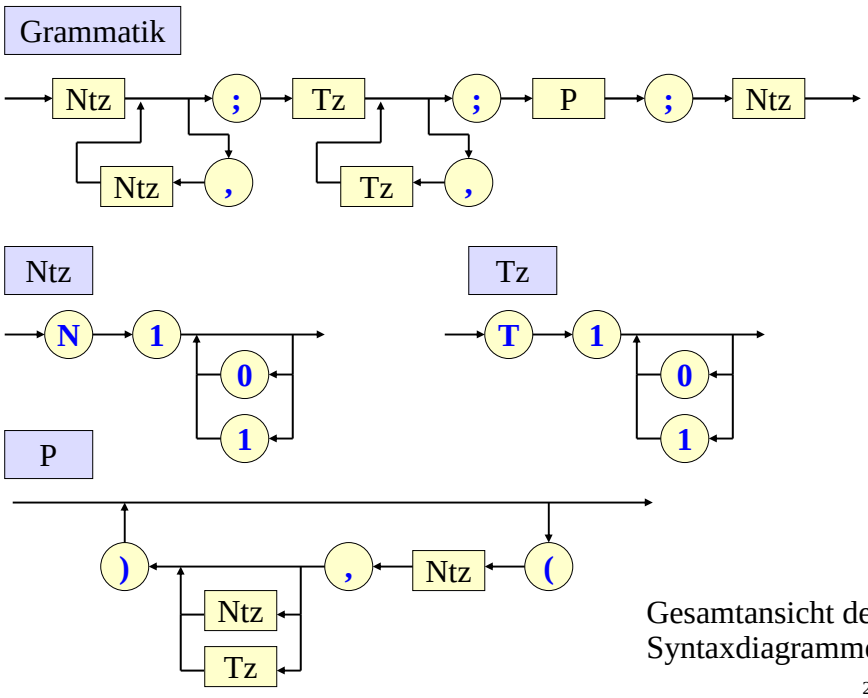


<Ntz> ::= '**N**' '**1**' {'**0**' | '**1**'} <Tz> ::= '**T**' '**1**' {'**0**' | '**1**'}

Syntaxdiagramme hierzu: (<rSeite> kann man leicht einsparen)



<P> ::= {'**(**' <Ntz> '**,**' <rSeite> '**)**'} <rSeite> ::= { <Ntz> | <Tz> }



Gesamtansicht der Syntaxdiagramme

255

Nun erzeugen wir ein Wort u mit dieser EBNF, z.B.:

N1,N11;T10,T101;(N1,N1N11)(N1,T10)(N11,T101);N1 $\in \Sigma_G^*$

Zugehörige Grammatik: $G_u = (V_u, \Sigma_u, P_u, S_u)$ mit $V_u = \{$**N1**, **N11**$\}$, $\Sigma_u = \{$**T10**, **T101**$\}$ und $P_u = \{$**N1**$\rightarrow$**N1N11**, **N1**$\rightarrow$**T10**, **N11**$\rightarrow$**T101**$\}$. Die erzeugte Sprache lautet $L(G_u) = \{$**T10**$\}\{$**T101**$\}^*$.

Indem man die Zeichen nach irgendeiner Vorschrift umcodiert (z.B.: **N1** $\leftrightarrow$ S, **N11** $\leftrightarrow$ Y, **T10** $\leftrightarrow$ a, **T101** $\leftrightarrow$ b), gewinnt man aus G_u eine gleichwertige Grammatik G, die bis auf Umbenennung von Zeichen genau dem oben abgeleiteten Wort u entspricht:

$G = (V, \Sigma, P, S)$ mit $V = \{S, Y\}$, $\Sigma = \{a, b\}$ und $P = \{ S \rightarrow SY,\ S \rightarrow a,\ Y \rightarrow b \}$. Die erzeugte Sprache lautet $L(G) = \{a\}\{b\}^* = \{a\,b^k \mid k \geq 0\}$.

■

Hinweis 1: Beliebige Grammatiken lassen sich auf die gleiche Weise beschreiben; man muss nur als "linke Seite" der Regeln eine nicht-leere Folge von Nichtterminalzeichen zulassen.

Hinweis 2: Manches lässt sich nicht mit einer EBNF darstellen, nämlich alle "Kontext bezogenen" Bedingungen. Insbesondere:

- Alle Nichtterminalzeichen müssen paarweise verschieden sein.
- Alle Terminalzeichen müssen paarweise verschieden sein.
- Alle Zeichen, die in den Regeln vorkommen, müssen in den Auflistungen der Terminal- bzw. Nichtterminalzeichen vorkommen.

Hinweis 3: Über die Eindeutigkeit der dargestellten Grammatik kann man hier keine Aussage machen. Sie muss i.A. für jede Grammatik einzeln bewiesen werden.

Hinweis 4: Bei der Definition von Programmiersprachen setzt man bei der Semantik gerne ein schrittweises Vorgehen ein: Zuerst definiert man eine sehr kleine Sprache, wie wir es z.B. mit den Forderungen 1.1.1.5 (A1) bis (A9) (ohne A8b, c, d, in A7 reicht die einseitige Alternative) in Abschnitt 1.1.1 getan haben. Ist die Bedeutung dieser "Kern"-Sprache bekannt, so erweitert man sie und führt die neuen Sprachelemente auf die der Kernsprache zurück. Z.B. kann man auf diese Weise die for- und die repeat-Schleife, die zweiseitige Fallunterscheidung, eine exit-Anweisung usw. schrittweise hinzunehmen und durch äquivalente Programmstücke beschreiben.

Im Übersetzerbau ist dieses Vorgehen als Bootstrapping bekannt, wobei man immer mächtigere Übersetzer einer Sprache in eine andere erhält. Am Ende kann man sogar einen optimierenden Übersetzer in der Sprache selbst schreiben und von einem "schlechten" Übersetzer realisieren lassen.

1.1.7.2: Bemerkungen zur Definition von Programmiersprachen

Die Syntax wird durch eine kontextfreie Grammatik bzw. eine EBNF definiert, der man Zusatzbedingungen umgangssprachlich hinzufügt (vgl. Hinweis 2). Meist lässt sich aus der EBNF ein Parser automatisch erzeugen.

Die Semantik wird meist anhand vieler Beispiele erläutert. Es gibt jedoch auch formale Methoden, um einem Programm die realisierte Abbildung zuzuordnen (Stichwörter: denotationale Semantik, axiomatische Semantik).

Bei der Definition geht man meist schrittweise vor, wobei man die Sprache ständig um neue Sprachelemente anreichert. Dies nutzt man auch im Übersetzerbau als "Bootstrapping".

Die Eindeutigkeit der EBNF einer Programmiersprache muss getrennt nachgewiesen werden (diese Eigenschaft ist "unentscheidbar", siehe später).

Skizze zu Syntax und Semantik: