Musterlösung

Übung 2

Aufgabe 2.1: Syntax, Semantik

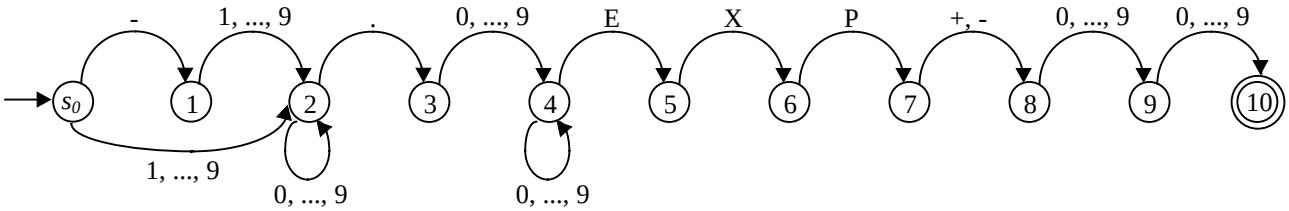
a) *Syntax*: Die Syntax einer Programmiersprache S ist die Definition aller in S zulässigen Aussagen, die in einer Sprache formuliert werden können.
Semantik: Die Semantik einer Sprache S ist die Definition der den zulässigen Aussagen zugeordneten Bedeutungen. Syntaktisch falsche Aussagen haben keine Semantik. Auch syntaktisch korrekte Aussagen haben nicht immer eine Semantik.
Somit ist Syntax ein formaler Begriff, Semantik ein inhaltlicher.
Die Syntax einer Programmiersprache läßt sich ohne Rücksicht auf eine Semantik definieren, d.h. die Syntax läßt sich ohne eine dahinterstehende Semantik definieren. Jedoch macht die Syntax einer Programmiersprache alleine (d.h. ohne Semantik) wenig Sinn. Ein Programm sagt nichts aus ohne eine Semantik. Es ist lediglich ein formales Konstrukt.
Um hingegen die Semantik einer Programmiersprache zu definieren, ist die vorherige Definition einer Syntax notwendig, da durch die Semantik den syntaktischen Konstrukten (bzw. einem Teil dieser) eine Bedeutung zugewiesen wird.

b) Gleiche Syntax impliziert *nicht* gleiche Semantik.
Man betrachte z.B. eine if-Anweisung der Form `if` Bedingung `then` Aktion `endif` . Zunächst kann man dieser die “intuitive” Semantik zuordnen. D. h. wenn die Bedingung (wobei diese auch einer Syntax genügen muß, und die Erfülltheit von der Semantik abhängt) erfüllt ist, dann soll die Aktion (welche ebenfalls die Regeln der Syntax erfüllen muß und die Ausführung von der Semantik abhängt) ausgeführt werden. Man könnte der if-Anweisung jedoch auch folgende Semantik zuordnen: Wenn die Bedingung nicht erfüllt ist, dann soll die Aktion ausgeführt werden.
Die Liste der Möglichkeiten für die Semantik läßt sich beliebig fortsetzen. Die Syntax ist also die gleiche, aber die Semantik nicht.

c) Syntaktisch korrekt bedeutet lediglich, daß ein Programm den Regeln der Syntax genügt. Korrektheit ist jedoch eine inhaltliche Bedingung, die die syntaktische Korrektheit voraussetzt.
Genauer bedeutet *Korrektheit*, daß ein Programm bestimmte Eigenschaften besitzt (festgelegt in den Anforderungen bzw. bei der Vorgabe, welches Problem zu lösen ist). Korrektheit bedeutet also, daß ein Algorithmus auch tatsächlich (für eine beliebige Eingabe) das vorgegebene Problem löst. Durch *syntaktische Korrektheit* ist dies nicht garantiert. Z.B. betrachte man zwei verschiedene Probleme (verschieden bedeutet hier, daß mindestens eine Eingabe existiert, auf die nach A bzw. B verschiedene Antwortengegeben werden müssen) A und B sowie Programme P_A und P_B, welche dieses Problem jeweils korrekt lösen (P_A für A und P_B für B - es wird vorausgesetzt, daß solche existieren). Dann sind P_A und P_B, zwar syntaktisch korrekt aber P_A ist nicht korrekt für B, und P_B ist nicht korrekt für A.

Aufgabe 2.2: Endliche Automaten

Der endliche Automat mit folgendem Zustandsdiagramm leistet das Gewünschte:



Durch den von den Zuständen s_0, 1 und 2 induzierten Teil wird die Zahl vor dem Dezimalpunkt beschrieben. Der Übergang in den Zustand 3 ist nur durch das Lesen des Dezimalpunktes möglich. Eine Transition aus Zustand 3 ist nur durch das Lesen einer Ziffer möglich, denn nach dem Dezimalpunkt muß eine Ziffer folgen.
Um den Endzustand (hier gibt es genau einen Endzustand, dieser ist 10) zu erreichen, müssen die Zustände 5, 6 und 7 in dieser Reihenfolge durchlaufen werden, was nur durch die Folge „EXP“ möglich ist. Ein Übergang aus Zustand 7 ist nur möglich durch Lesen von „+“ oder „-“. Schließlich müssen genau zwei Ziffern folgen, damit das Wort den Bedingungen genügt. Daher kann vom Zustand 8 der Endzustand genau dann erreicht werden, wenn genau zwei Ziffern gelesen werden.

Aufgabe 2.3: EBNF, Syntaxdiagramme

a) Folgende EBNF leistet das Gewünschte:

Pfadname	=	letter “:“ “\“ {dirspec} filename.
rootsymbol	=	“\“.
dirspec	=	basename “\“.
filename	=	basename “.“ extension.
basename	=	namechar {namechar}.
extension	=	namechar {namechar}
namechar	=	(letter | digit | “_“ | “^“ | “$“ | “!“ | “#“ | “%“ | “&“ | “-“ | “{“ | “}“ |“(“ | “)“ | “@“ | “´“ | “`“)
letter	=	(a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |w| x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z)
digit	=	(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

(x_1 | x_2 | ... | x_n) ist die Abkürzung für (x_1 | (x_2 | (... |x_n) ...))

Die Definition von *Pfadname* beschreibt die grobe Struktur eines Pfadnamens wie vorgegeben. Er besteht aus dem Buchstaben für das Dateisystem, gefolgt von einem Doppelpunkt, der Spezifikation des Verzeichnisses und dem Dateinamen. Die Spezifikation des Verzeichnisses ist eine endliche Folge von Verzeichnisnamen, welche jeweils durch „\“ getrennt sind. Am Ende steht ebenfalls „\“. Die übrigen Definitionen definieren die benutzten Namen, welche im wesentlichen endliche Buchstaben - (und Sonderzeichen-) Folgen sind.

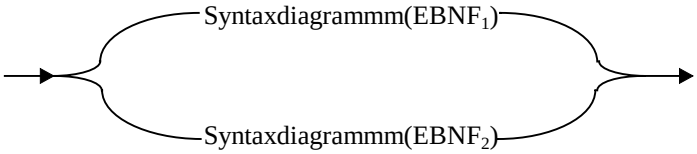
b) Der Algorithmus arbeitet rekursiv über den Aufbau der EBNF. Eine Fallunterscheidung nach dem Konstrukt auf der obersten Ebene wird durchgeführt. Es wird jeweils ein Syntaxdiagramm angegeben, in das die Syntaxdiagramme der EBNFs, die auf den tieferen Ebenen auftreten, wie jeweils angegeben einzusetzen sind. Hierbei bezeichnet Syntaxdiagramm(EBNF) das Syntaxdiagramm zu EBNF, wie es durch den Algorithmus bestimmt wurde.
Der folgende Algorithmus leistet das Gewünschte:

Wenn die EBNF die Form Name = EBNF' besitzt, dann ist Syntaxdiagramm(EBNF):

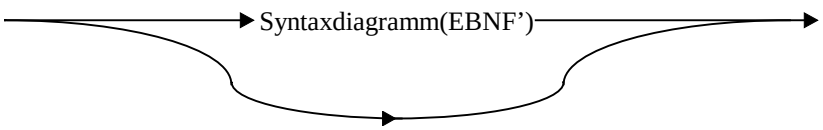
Name

Syntaxdiagramm(EBNF')

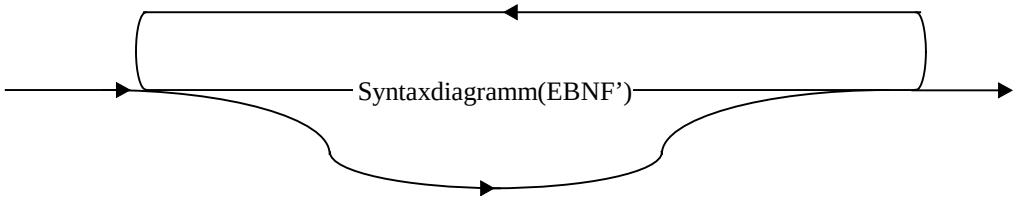
($EBNF_1$| $EBNF_2$) besitzt, dann ist Syntaxdiagramm(EBNF):



[EBNF'] besitzt, dann ist Syntaxdiagramm(EBNF):



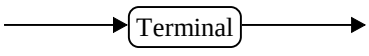
{EBNF'} besitzt, dann ist Syntaxdiagramm(EBNF):



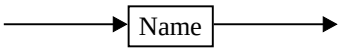
EBNF'.besitzt, dann ist Syntaxdiagramm(EBNF):

Syntaxdiagramm(EBNF')

„Terminal" besitzt, wobei Terminal für ein beliebiges Terminal steht, dann ist Syntaxdiagramm(EBNF):



Name besitzt, wobei Name die Bezeichnung für eine andere EBNF ist, dann ist
Syntaxdiagramm(EBNF):



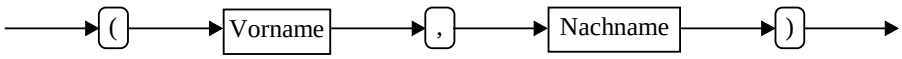
Erläuterung zu dem Algorithmus beispielhaft für den Fall ($EBNF_1$| $EBNF_2$):
Genau eine der Alternativen $EBNF_1$, $EBNF_2$ muß stehen. Genau dieses ist auch im zugehörigen Syntaxdiagramm der Fall. Abgesehen von den möglichen Pfaden innerhalb von Syntaxdiagramm($EBNF_1$) und Syntaxdiagramm($EBNF_2$), gibt es genau zwei mögliche Pfade durch das Syntaxdiagramm: Durch Syntaxdiagramm($EBNF_1$) und durch Syntaxdiagramm($EBNF_2$). Dies entspricht der EBNF.

Rekursion ist hierbei *nicht notwendig*. Der obige Algorithmus kann einfach wiei folgt in einen nicht-rekursiven Algorithmus überführt werden:
Bestimme (zur eingegebenen EBNF) die Menge aller Teil-EBNFs.
Wende die oben angegebenen Regeln zunächst auf die „kleinsten“ EBNFs (dies sind Namen und Terminale) an. Speichere die Ergebnisse ab und gehe hier zur nächst höheren Ebene (die Menge aller Teil-EBNFs, deren Teil-EBNFs nur die EBNFs der vorherigen Ebene und sie selbst sind und welche noch nicht in der Ebene darunter liegen) über. Wende die Regeln auf diese an (da die Teil-EBNFs bereits in Syntaxdiagramme überführt wurden, findet hierbei kein rekursiver Aufruf statt).
Verfahre wie oben beschrieben mit allen Ebenen, bis eine Ebene erreicht ist, zu welcher es keine höhere Ebene mehr gibt (eine solche existiert, da die Menge der Teil-EBNFs endlich ist).

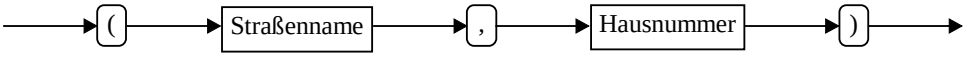
c)

Adresse

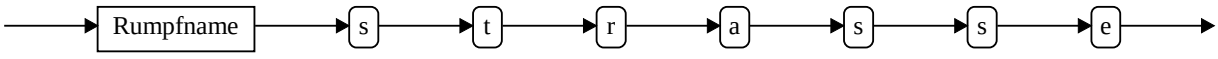
Name



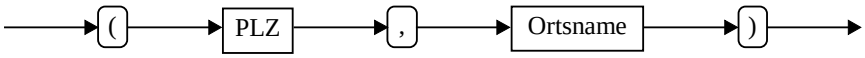
Straße



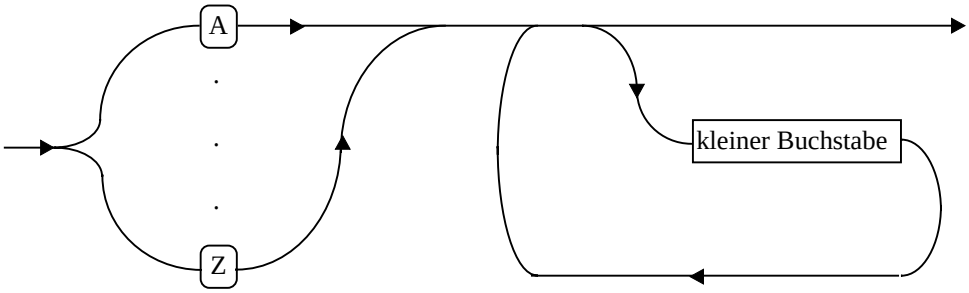
Straßenname



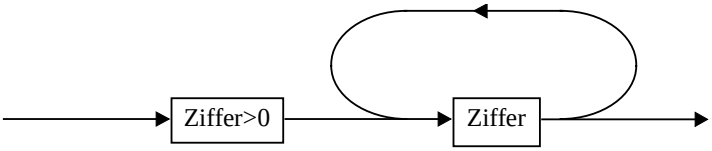
Ort



Vorname, Nachname, Rumpfname, Ortsname

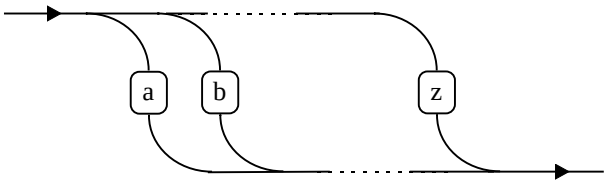


Hausnummer

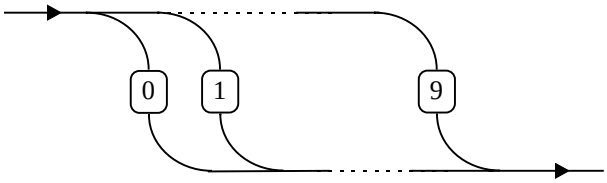


PLZ

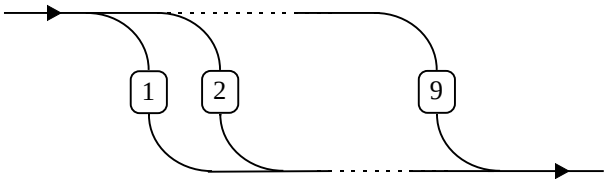
kleiner Buchstabe



Ziffer



Ziffer>0



Zunächst wird durch das mit Adresse bezeichnete Syntaxdiagramm die grobe Struktur einer Adresse beschrieben. Die übrigen Syntaxdiagramme definieren die benutzten Komponenten.