Gliederung des Kapitels 1.1

1.1 Algorithmen und Sprachen

~~1.1.1 Darstellung von Algorithmen~~
~~1.1.2 Grundlegende Datenbereiche~~
~~1.1.3 Realisierte Abbildung~~
~~1.1.4 (Künstliche) Sprachen~~
~~1.1.5 Grammatiken~~
1.1.6 BNF, Syntaxdiagramme
1.1.7 Sprachen zur Beschreibung von Sprachen
1.1.8 Übungsaufgaben

Abschnitt 1.1.6 soll Ihnen folgende Inhalte vermitteln:

Programmiersprachen lassen sich weitgehend durch kontextfreie Grammatiken beschreiben. Bereits 1958 wurde eine Variante, die BNF (sog. Backus-Naur-Form), zur Definition von Programmiersprachen eingesetzt; diese wurde später erweitert und durch Syntaxdiagramme grafisch veranschaulicht.

In diesem Abschnitt erlernen Sie die Erweiterung EBNF und wie man mit ihrer Hilfe die Syntax einer Programmiersprache definiert. Als Beispiele werden Teile von Ada 95 verwendet.

1.1.6 BNF, Syntaxdiagramme

Definition 1.1.6.1: Backus-Naur-Form
Eine BNF ist ein Viertupel (V, Σ, P, S) mit

(1) V ist eine nicht-leere endliche Menge (die Menge der Nichtterminalzeichen); alle Elemente sind von der Form < Zeichenkette > (in "Zeichenkette" treten '<' und '>' nicht auf),
(2) Σ ist eine nicht-leere endliche Menge (die Menge der Terminalzeichen) mit $V \cap \Sigma = \emptyset$ und $\mid \notin \Sigma$,
(3) $S \in V$ ist ein Nichtterminalzeichen (das Startsymbol),
(4) $P \subset V \{::=\} (V \cup \Sigma)^* (\{\mid\} (V \cup \Sigma)^*)^*$ ist eine endliche Menge (die Menge der Regeln oder Produktionen).

Eine BNF ist eine kontextfreie Grammatik (vgl. Def. 1.1.5.1), bei der die Nichtterminalzeichen durch Namen der Form "< Zeichenkette >" genauer bezeichnet werden können und bei der alle kontextfreien Regeln, die die gleiche linke Seite besitzen, zu einer Regel zusammengefasst werden, wobei die verschiedenen rechten Seiten durch einen senkrechten Strich getrennt werden.

Statt des Ersetzungspfeils "→" schreibt man "::=" (gesprochen: "Doppel-Doppel-Punkt-Gleich"; die Bedeutung ist: "kann ersetzt werden durch").

Bei Programmiersprachen verwendet man gerne das Nichtterminalzeichen <program> als Startsymbol, aus dem man die zulässigen Programme herleitet.

Beispiel 1.1.6.2: Definition der Bezeichner, vgl. Beispiel 1.1.5.5.

<Buchstabe> ::= 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

<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Zeichen für Bezeichner> ::= _ | <Buchstabe> | <Ziffer>

<Bezeichner> ::= <Buchstabe> |
<Bezeichner> <Zeichen für Bezeichner>

V = {<Bezeichner>, <Zeichen für Bezeichner>, <Buchstabe>, <Ziffer>}
Σ = {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,0,1,2,3,4,5,6,7,8,9,_}
Das Startsymbol ist hier <Bezeichner>. ■

Umwandlung einer kontextfreien Grammatik in die BNF:
Gegeben sei eine kontextfreie Grammatik (V', Σ, P', S').

1. Ersetze jedes Nichtterminalzeichen $A' \in V'$ durch einen aussagekräftigen Namen A der Form <Zeichenkette>, wobei die Zeichen '<' und '>' in dieser "Zeichenkette" nicht verwendet werden dürfen. Dies ergibt die neue Menge der Nichtterminalzeichen V mit dem Startsymbol S (anstelle S').
2. Ersetze in P' jedes Nichtterminalzeichen durch seinen neuen Namen der Form <Zeichenkette>.
3. Wenn $A' \to u_1$, $A' \to u_2$, ..., $A' \to u_k$ alle Regeln in P' mit A' als linker Seite sind und wenn A das zu A' gehörende neue Nichtterminalzeichen aus V ist, so ersetze die k Regeln durch $A ::= u_1 \mid u_2 \mid ... \mid u_k$
So entsteht aus P' die neue Regelmenge P.

Umgekehrt kann man mit dieser Anleitung auch eine BNF in eine "normale" kontextfreie Grammatik zurück verwandeln.

Der Ableitungsbegriff der BNF ist daher der gleiche wie der für kontextfreie Grammatiken. Somit sind die Ableitungsrelationen $\Rightarrow$ und $\Rightarrow^*$ sowie die erzeugte Sprache L(G) auch für die BNF definiert (vgl. 1.1.5.2 und 1.1.5.3).

Die BNF wird nun um einige hilfreiche Abkürzungen zur EBNF erweitert.

Verwenden von Schlüsselwörtern:
Schlüsselwörter der Sprache werden als Zeichenketten dargestellt, die in Apostrophe eingeschlossen werden. (Tritt im Schlüsselwort ein Apostroph auf, so wird dies durch zwei Apostrophe dargestellt.)

Beispiel:
<Boolesche Operatoren> ::= 'and' | 'or'

Hierdurch wir ausgedrückt, dass die Operatoren and und or eigentlich einzelne Terminalzeichen sind, jedoch als Zeichenkette durch Konkatenation anderer Zeichen dargestellt werden. Man könnte ansonsten in BNF auch schreiben:

<And-Operator> ::= and
<Or-Operator> ::= or
<Boolesche Operatoren> ::= <And-Operator> | <Or-Operator>

Einführen von eckigen Klammern ("ein- oder keinmal"):
Symbole oder Folgen von Symbolen, die auch wegfallen dürfen, werden in eckige Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> [<Ziffernfolge>]
ist die Abkürzung für
<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>

<Alternative> ::= 'if' <Boolescher Ausdruck>
'then' <Anweisung>
['else' <Anweisung>] 'fi'

Einführen von geschweiften Klammern ("Iteration"):
Symbole oder Folgen von Symbolen, die beliebig oft (auch keinmal) wiederholt werden dürfen, werden in geschweifte Klammern eingeschlossen.

Beispiele:

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}
ist die Abkürzung für
<Ziffern*> ::= ε | <Ziffer> <Ziffern*>
<Ziffernfolge> ::= <Ziffer> <Ziffern*>

Die Definition von Bezeichnern nach 1.1.6.2 lässt sich mit geschweiften Klammern wie folgt abkürzen:
<Bezeichner> ::= <Buchstabe> { _ | <Buchstabe> | <Ziffer> }

Definition 1.1.6.3: Erweiterte Backus-Naur-Form
Die EBNF ist die Erweiterung der BNF um die Möglichkeiten:

Apostrophe für Schlüsselwörter als Terminalzeichen.

Eckige Klammern mit der Bedeutung: Die eingeschlossenen Teile können hier ein Mal auftreten, müssen aber nicht.

Geschweifte Klammern mit der Bedeutung: Die hierdurch eingeschlossenen Teile können beliebig oft nacheinander auftreten. (Dies entspricht dem Operator * auf Wortmengen.)

Ergänzender Hinweis: Falls eckige oder geschweifte Klammern als Terminalzeichen in den Regeln auftreten können, so bleibt es dem Ersteller der Regeln überlassen, für Eindeutigkeit zu sorgen.

Entscheidendes Kriterium, ob eine Grammatik oder BNF für die Praxis geeignet ist, ist, dass mit ihr möglichst viele der Operationen für Sprachen nachgebildet werden können (vgl. 1.1.4.6). Für die EBNF gilt zum Beispiel:

Die *Vereinigung* wird durch '|' dargestellt,
die *Konkatenation* durch Hintereinanderschreiben
und die *Iteration* durch die geschweiften Klammern.

Die Operatoren "Durchschnitt" und "Komplement" erhalten wir auf diese Weise nicht. Für kontextfreie Sprachen gelingt dies auch nicht, da die Klasse der kontextfreien Sprachen nicht gegen Durchschnitt und nicht gegen Komplement abgeschlossen ist, d.h.: Wenn L_1 und L_2 kontextfreie Sprachen sind, dann sind der Durchschnitt $L_1 \cap L_2$ und das Komplement von L_1 im Allgemeinen keine kontextfreien Sprachen.

Beispiel 1.1.6.4: Darstellung zur Basis b für $2 \le b \le 16$ in Ada. Diese Zahlen werden in der Form
2 # 10010 # oder 8 # 60175 # oder 16 # A2F03 #
dargestellt, wobei zwischen den #-Zeichen mindestens eine Ziffer stehen muss und nur die Basen von 2 bis 16 zugelassen sind. Ansatz, um dies mit einer EBNF zu beschreiben:

< Darstellung mit Basis > ::= <Basis> # <Ziffernfolge> #

Doch dieser Ansatz wird nicht erfolgreich sein, da zu jeder Basis eine andere Menge von Ziffern gehört. Wir gehen daher den aufwändigen Weg, zu jeder Zahl von 2 bis 16 die Regeln neu aufzuschreiben, wobei sich eine offensichtliche Regelmäßigkeit ergibt.
Abkürzend schreiben wir <DarBa2> für die <Darstellung zur Basis 2> und analog für die anderen Basen sowie <DarBa> für das Startsymbol <Darstellung mit Basis>.

<Ziffern2> ::= 0 | 1
<DarBa2> ::= 2 # <Ziffern2> {<Ziffern2>} #
<Ziffern3> ::= <Ziffern2> | 2
<DarBa3> ::= 3 # <Ziffern3> {<Ziffern3>} #
<Ziffern4> ::= <Ziffern3> | 3
<DarBa4> ::= 4 # <Ziffern4> {<Ziffern4>} #
... *usw* ...
<Ziffern11> ::= <Ziffern10> | A
<DarBa11> ::= 11 # <Ziffern11> {<Ziffern11>} #
... *usw* ...
<Ziffern16> ::= <Ziffern15> | F
<DarBa16> ::= 16 # <Ziffern16> {<Ziffern16>} #
<DarBa> ::= <DarBa2> | <DarBa3> | <DarBa4> | <DarBa5> |
<DarBa6> | <DarBa7> | <DarBa8> | <DarBa9> |
<DarBa10> | <DarBa11> | <DarBa12> | <DarBa13> |
<DarBa14> | <DarBa15> | <DarBa16> ■

Hinweis: Es gibt Varianten der EBNF.

Oft schreibt man (...)* statt {...}
und führt auch das Konstrukt (...)+ ein, wenn man die 0-Iteration verbieten möchte.

Statt ::= wird manchmal auch einfach = geschrieben.

Einzelne Terminalzeichen müssen oftmals in Anführungsstriche eingeschlossen werden.
Schlüsselwörter werden dann meist fett gedruckt.
Die Nichtterminalzeichen werden dann nicht mehr in spitze Klammern '<' und '>' eingeschlossen.

Regeln werden mit einem besonderen Zeichen, meist einem Punkt, abgeschlossen.

Einschub: *Überlegung 1.1.6.5:* Es scheint wenig sinnvoll zu sein, fünfzehn Mal im Wesentlichen das Gleiche immer wieder hinschreiben zu müssen. Könnte man nicht ein neues Sprachelement zur EBNF hinzufügen, welches "für k von 2 bis 16" diese Wiederholungen vornimmt?!

In Ada-ähnlicher Notation *könnte* man das Sprachelement

for k := 2 to 16 loop <Folge von Regeln > end loop

einführen. In den Regeln wollen wir diese Variable k benutzen, aber wir können nicht einfach "k" dort verwenden, da nicht klar ist, ob die Variable oder das Zeichen gemeint ist. Wir schließen k daher in ein Sonderzeichen (z.B. $) ein und erlauben, k oder allgemeiner $"Ausdruck, in dem k vorkommt"$ in den Regeln zu verwenden. Dies wird am Beispiel der Darstellungen zu einer Basis klar.

Die Zahldarstellungen zur Basis $2 \le b \le 16$ kann man dann folgendermaßen aufschreiben, wobei wir die einzelnen Ziffern von 0 bis b-1 hier in der Form (0), (1), (2), ..., (b-1) notieren:

<Ziffern2> ::= (0) | (1)
<DarBa2> ::= 2 # <Ziffern2> {<Ziffern2>} #
<DarBa> ::= <DarBa2>

for k := 3 to 16 loop
<Ziffern$k$> ::= <Ziffern$k-1$> | ($k-1$)
<DarBak> ::= k # <Ziffern$k$> {<Ziffern$k$>} #
<DarBa> ::= <DarBa> | <DarBak>
end loop

Die Zeile <DarBa> ::= <DarBa> | <DarBak> soll hier bedeuten: Es wird das alte <DarBa> um <DarBak> erweitert und das Ergebnis heißt wiederum <DarBa>.

Hätte man das Symbol "nichts" oder "leere Menge" ∅ zur Verfügung, so könnte man auch folgende etwas einfachere Definition verwenden:

<Ziffern1> ::= (0)
<DarBa> ::= ∅

for k := 2 to 16 loop
<Ziffern$k$> ::= <Ziffern$k-1$> | ($k-1$)
<DarBak> ::= k # <Ziffern$k$> {<Ziffern$k$>} #
<DarBa> ::= <DarBa> | <DarBak>
end loop

Problem: Was wurde nun eigentlich definiert? Nur <DarBa> oder auch <Ziffern3>, <Ziffern4>, <DarBa3>, <DarBa4> usw.?

Warum gibt es solche Erweiterungen nicht in der Praxis?

Zunächst ist anzunehmen, dass es sehr viele solcher Erweiterungen der EBNF gibt, die von einzelnen Labors oder Firmen auf ihre Bedürfnisse zugeschnitten sind.

Das Problem ist die Bedeutung der Sprachelemente, insbesondere Eindeutigkeit und Verständlichkeit:

- Bleiben die zwischenzeitlich aufgebauten Nichtterminal-zeichen erhalten?
- Wenn solche for-Konstrukte geschachtelt ineinander und dort in Alternativen vorkommen, ist dann das, was entsteht, stets eindeutig bestimmt?
- Und wenn man dies alles exakt definieren kann, ist der Formalismus dann noch leicht erlernbar?

Ende der Überlegung 1.1.6.5 ■

Beispiel 1.1.6.6: Beschreibung der Ada-Anweisungen

<sequence_of_statements> ::= <statement> {<statement>}

<statement> ::= {<label>} <simple_statement> |
{<label>} <compound_statement>

<simple_statement> ::= <null_statement> |
<assignment_statement> | <exit_statement> |
<procedure_call_statement> | <return_statement> |
<entry_call_statement> | <goto_statement> |
<requeue_statement > | <delay_statement> |
<abort_statement> | <raise_statement> | <code_statement>

Stopp! Unterbrechung: Man erkennt nun, warum es Varianten der EBNF gibt. In diesen Regeln treten keine Terminalzeichen auf und die spitzen Klammern sind eher unübersichtlich. Daher gehen wir nun zu der in Ada üblicheren Darstellung über (ohne spitze Klammern, Terminalzeichen werden blau und durch An-führungszeichen, Schlüsselwörter blau und durch Apostrophe hervorgehoben).

Beispiel 1.1.6.6: Nochmals Beschreibung der Ada-Anweisungen

sequence_of_statements ::= statement {statement}

statement ::= {label} simple_statement |
{label} compound_statement

simple_statement ::= null_statement |
assignment_statement | exit_statement |
goto_statement | procedure_call_statement |
return_statement | entry_call_statement |
requeue_statement | delay_statement |
abort_statement | raise_statement | code_statement

Bemerkung: Bisher haben wir hiervon nur das null_statement (= skip), das assignment_statement (= Wertzuweisung) und das exit_statement (= Verlassen einer Schleife) kennen gelernt.

Beispiel 1.1.6.6: Ada-Anweisungen (Fortsetzung)

label ::= "<<" *label*_statement_identifier ">>"

statement_identifier ::= direct_name

null_statement ::= 'null' ";"

assignment_statement ::= *variable*_name ":=" expression ";"

exit_statement ::= 'exit' [*loop*_name] ['when' condition] ";"

goto_statement ::= 'goto' *label*_name ";"

procedure_call_statement ::= *procedure*_name ";" |
*procedure*_prefix actual_parameter_part ";"

return_statement ::= 'return' [expression] ";"

Bemerkung: Die verbleibenden sechs <simple_statement> behandeln wir später bzw. im Programmierkurs.

Beispiel 1.1.6.6: Ada-Anweisungen (Fortsetzung)

Erneuter Hinweis zur EBNF-Schreibweise:
Alle Nichtterminalzeichen, die in Normalschrift geschrieben sind, stehen irgendwo auf der linken Seite einer EBNF-Regel. Manche Nichtterminalzeichen enthalten kursiv geschriebene Teile. Diese sind bedeutungstragende Hinweise und nur das restliche Nichtterminalzeichen (ohne die kursiven Teile) tritt als linke Seite einer EBNF-Regel auf. Zwei Beispiele:
(1) *variable*_name ist ein name
Der Zusatz *variable* besagt, dass an dieser Stelle eine Variable stehen muss und nicht etwa ein Prozedurname oder eine Marke.
(2) *label*_statement_identifier ist ein statement_identifier
Der Zusatz *label* besagt, dass hier der Bezeichner für eine Marke stehen muss, die irgendwo im Programm vorhanden ist.

Beispiel 1.1.6.6: Ada-Anweisungen (Fortsetzung)

compound_statement ::= if_statement | case_statement |
loop_statement | block_statement | accept_statement |
select_statement

if_statement ::= 'if' condition 'then' sequence_of_statements
{'elsif' condition 'then' sequence_of_statements}
['else' sequence_of_statements] 'end if' ";"

condition ::= *boolean*_expression

case_statement ::= 'case' expression 'is'
case_statement_alternative
{case_statement_alternative} 'end case' ";"

case_statement_alternative ::=
'when' discrete_choice_list "=>" sequence_of_statements

Beispiel 1.1.6.6: Ada-Anweisungen (Fortsetzung)

```
loop_statement ::= [loop_statement_identifier ":"]
                   [iteration_scheme] 'loop'
                   sequence_of_statements
                   'end loop' [loop_identifier] ";"
iteration_scheme ::= 'while' condition |
                     'for' loop_parameter_specification
loop_parameter_specification ::=
    defining_identifier 'in' ['reverse'] discrete_subtype_definition
block_statement ::= [block_statement_identifier ":"]
                    ['declare' declarative_part]
                    'begin' handled_sequence_of_statements
                    'end' [block_identifier] ";"
```


Dies waren 20 Regeln von etwa 400 EBNF-Regeln zur Beschreibung der Syntax von Ada 95.

Üben Sie das Lesen und Verstehen der EBNF-Regeln und damit der Syntax von Ada 95 und anderer Programmiersprachen .

Die komplette Syntax von Ada 95 finden Sie in Büchern, aber auch im Internet an mehreren Stellen, z.B. unter

http://www.adahome.com/rm95/rm9x-P.html

oder (einschl. Syntaxdiagrammen) unter

http://cui.unige.ch/db-research/Enseignement/analyseinfo/Ada95/BNFindex.html.

Die Ada-Syntax ist im "Ada Reference Manual" festgelegt; die Auflistungen der Syntax erfolgt nach dem dort benutzten Schema in 13 Abschnitten.

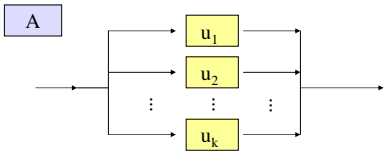
Ende der Anweisungen in Ada 1.1.6.6 ■

Nun fehlt nur noch die grafische Darstellung von EBNF-Regeln. Für jede linke Seite (also für jedes Nichtterminalzeichen) legen wir ein eigenes Diagramm an, das mit dem Nichtterminal bezeichnet wird.

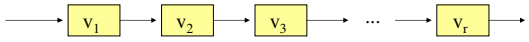
Betrachte eine solche Regel mit dem Nichtterminal A:

$A ::= u_1 \mid u_2 \mid \ldots \mid u_k$

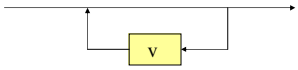
Dann zeichne hierzu das Diagramm



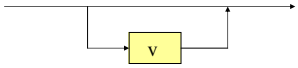
Ist ein u_i von der Form $v_1 v_2 \ldots v_r$ (Konkatenation von Zeichen), so ersetze durch:



Ist ein u_i oder v_j von der Form { v }, so ersetze es durch



Ist ein u_i oder v_j von der Form [v], so ersetze es durch



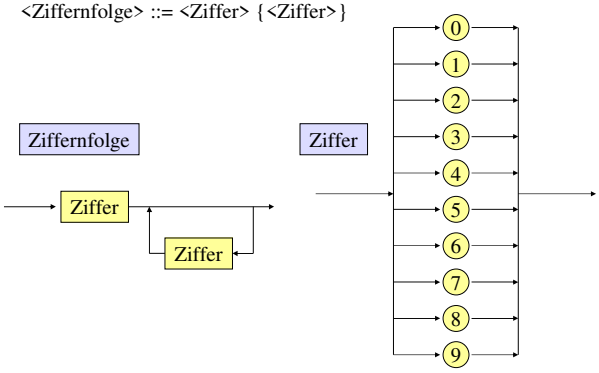
Zum Schluss ersetzt man bei allen Terminalzeichen und Schlüsselwörtern die rechteckigen Umrandungen durch Kreise.

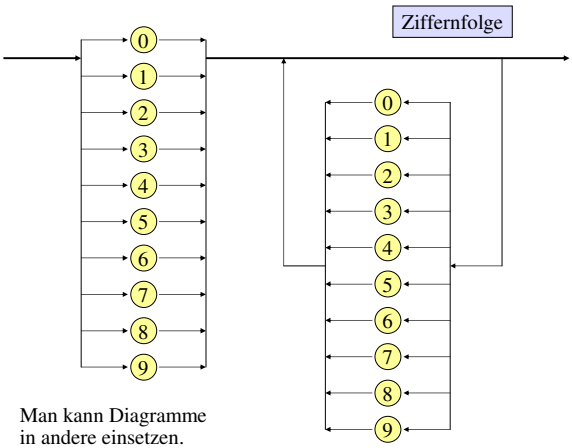
1.1.6.7: Dieses Vorgehen wird hierarchisch fortgesetzt, wodurch sich schließlich für jede Grammatik, bzw. EBNF eine grafische Darstellung, das sog. Syntaxdiagramm, ergibt.

Um Wörter aus einem Nichtterminalzeichen A zu erzeugen, durchläuft man das zu A gehörende Syntaxdiagramm vom Eingangspfeil bis zum Ausgangspfeil. Hier gibt es i.A. viele Wege. Für jeden Weg sammelt man die Folge aller hierbei besuchten Terminalzeichen in der Durchlaufreihenfolge auf. Alle diese Zeichenfolgen bildet dann die Menge der von A erzeugten Wörter. Beispiele und eine genauere Formulierung folgen.

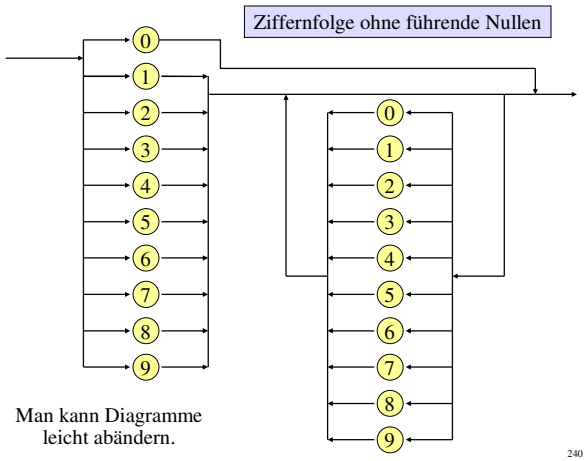
Beispiel: <Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Ziffernfolge> ::= <Ziffer> {<Ziffer>}





Man kann Diagramme in andere einsetzen.

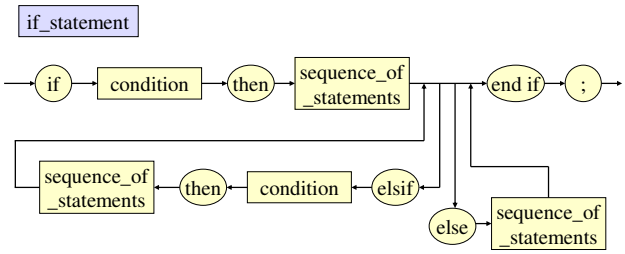


Man kann Diagramme leicht abändern.

Etwas präzisiere Beschreibung der Bedeutung von Syntaxdiagrammen: Gegeben sind Syntaxdiagramme und eine anfangs leere Ausgabe.

Ziel ist es, das Diagramm, das zum Startsymbol gehört, in Richtung der Pfeile vom Eingangspfeil bis zum Ausgangspfeil zu durchlaufen. Trifft man hierbei auf Zeichen in Kreisen, so schreibt man diese in der Durchlaufreihenfolge hinter die bereits vorhandene Ausgabe. Trifft man auf ein Rechteck mit dem Nichtterminalzeichen A, so klebt man das zu A gehörende Diagramm an dieser Stelle ein und durchläuft das neu entstandene Diagramm weiter. (Gibt es kein zu A gehörendes Diagramm, so bricht man ergebnislos ab.)

Alle möglichen Ausgaben, die zu einem vollständigen Durchlauf vom Eingangspfeil bis zum Ausgangspfeil des Startsymbol-Diagramms gehören, bilden die erzeugte Sprache.

Beispiel

if_statement ::= 'if' condition 'then' sequence_of_statements
{'elsif' condition 'then' sequence_of_statements}
['else' sequence_of_statements] 'end if' ";"