Algorithmic Recursive Sequence Analysis Algorithmic structuralism: Formalizing Genetic Structuralism: An Attempt to Help Make Genetic Structuralism Falsifiable

Paul Koop M.A.
post@paul-koop.org

June 22, 2023

Abstract

A method for the analysis of discrete finite character strings is presented. Postmodern social philosophy is rejected. A naturalistic sociology with falsifiable models for action systems is approved. The algorithmic recursive sequence analysis (Aachen 1994) is presented with the definition of a formal language for social actions, a grammar inducer (Scheme), a parser (Pascal) and a grammar transducer (Lisp).

Algorithmic Recursive Sequence Analysis (Aachen 1994) is a method for analyzing finite discrete character strings. Ndiaye, Alassane (Rollenübernahme als Benutzermodellierungsmethode : globale Antizipation in einem transmutierbaren Dialogsystem 1998) and Krauße, C. C., Krueger,F.R. (Unbekannte Signale 2002) published equivalent methods. It is ingenious to simply think something simple. Since the beginning of the 21st century, the construction of grammars from given empirical character strings has been discussed in computational linguistics under the heading of grammar induction (Alpaydin, E. 2008: Maschinelles Lernen, Shen, Chunze 2013: Effiziente Grammatikinduktion, Dehmer (2005) Strukturelle Analyse, Krempel 2016: Netze, Karten, Irrgärten). With sequitur, Nevill-Manning and Witten (Nevill-Manning Witten 1999: Identifying Hierarchical Structure in Sequences: A linear-time algorithm 1999) define a grammar induction for the compression of character strings. Graphs, grammars and transformation rules are of course just the beginning. Because a sequence analysis is only complete when, as in the case of algorithmic recursive sequence analysis, at least one Grammar can be specified for which a parser identifies the sequence as well-formed, with which a transducer can generate artificial protocols that are equivalent to the examined empirical sequence, and for which an inducer can generate at least one equivalent grammar. Gold (1967) formulated the problem in response to Chomsky (1965). Algorithmic structuralism is consistent, empirically proven, Galilean, naturalistic, Darwinian and a nuisance for deeply hermeneutic, constructivist, postmodernist and (post)structuralist social philosophers. I welcome heirs who continue the work or seek inspiration. A social action is an event in the possibility space of all social actions. The meaning of a social action is the set of all possible subsequent actions and their probability of occurrence. The meaning does not have to be understood interpretively, but can be reconstructed empirically. The reconstruction can be proven or falsified by probation tests on empirical protocols. From the mid-1970s to the present, irrationalist or anti-rationalist ideas have become increasingly prevalent among academic sociologists in America, France, Britain, and Germany. The ideas are referred to as deconstructionism, deep hermeneutics, sociology of knowledge, social constructivism, constructivism, or science and technology studies. The generic term for these movements is (post)structuralism or postmodernism. All forms of postmodernism are anti-scientific, anti-philosophical, anti-structuralist, anti-naturalist, anti-Galilean, anti-Darwinian, and generally anti-rational. The view of science as a search for truths (or approximate truths) about the world is rejected. The natural world plays little or no role in the construction of scientific knowledge. Science is just another social practice that produces narratives and myths no more valid than the myths of pre-scientific epochs. One can observe the subject of the social sciences as astronomy observes its subject. If the object of the social sciences eludes direct access or laboratory experiments like celestial objects (court hearings, sales talks, board meetings, etc.), the only thing that remains is to observe it purely physically without interpretation and to record the observations purely physically. The protocols could of course also be interpreted without reference to physics, chemistry, biology, evolutionary biology, zoology, primate research and life science. This unchecked interpretation is then called astrology when observing the sky. In the social sciences, this unchecked interpretation is also called sociology. Examples are constructivism (Luhmann), systemic doctrines of salvation, postmodernism, poststructuralism, or theory of communicative action (Habermas). Rule-based agent models have therefore previously worked with heuristic rule systems. These control systems have not been empirically proven. As in astrology, one could of course also create computer models in sociology, which, like astrological models, would have little empirical explanatory content. Some call this socionics. However, the protocols can also be interpreted taking into account physics, chemistry, biology, evolutionary biology, zoology, primate research and life science and checked for empirical validity. The observation of celestial objects is then called astronomy. In the social sciences one could speak of socionomy or sociomatics. That's actually sociology. This would not result in big worldviews, but as in astronomy, models with a limited range that can be empirically tested and can be linked to evolutionary biology, zoology, primate research and life science. These models (differential equations, formal languages, cellular automata, etc.) allow the deduction of empirically testable hypotheses, so they would be falsifiable. Such socionomy or sociomatics does not yet exist. I would prefer formal languages as model languages for empirically proven rule systems. Because rule systems for court hearings or sales talks, e.g. (models with limited range, multi-agent systems, cellular automata) can be modeled with formal languages rather than with differential equations. Algorithmic structuralism is an attempt to help translate genetic structuralism (without omission and without addition) into a falsifiable form and to enable empirically proven systems of rules. The Algorithmically Recursive Sequence Analysis is the first systematic attempt at a naturalistic and computer-based formulation of genetic structuralism as a memetic and evolutionary model. The methodology of Algorithmic Recursive Sequence Analysis is Algorithmic Structuralism. Algorithmic structuralism is a formalization of genetic structuralism. Genetic structuralism (Oevermann) assumes an intention-free, apsychic possibility space of algorithmic rules that structure the pragmatics of well-formed chains of events in text form (Chomsky, McCarthy, Papert, Solomon, Lévi-Strauss, de Saussure, Austin, searle). Algorithmic structuralism is an attempt to make genetic structuralism falsifiable. Algorithmic structuralism is Galilean and just as incompatible with Habermas and Luhmann as Galileo was with Aristotle. Of course, one can try to remain compatible with Luhmann or Habermas and to algorithmize Luhmann or Habermas. All artefacts can be algorithmized, for example astrology or chess. And one can model normative agents of distributed artificial intelligence, cellular automata, neural networks and other models with heuristic protocol languages and rules. This is undoubtedly theoretically valuable. So there will be no sociological theoretical progress. A new sociology is sought that models the replication, variation and selection of social replicators stored in artifacts and neural patterns. This new sociology will be just as incompatible with Habermas or Luhmann as Galileo could be with Aristotle. And their basic theorems will be as simple as Newton's laws. Just as Newton operationally defined the concepts of motion, acceleration, force, body and mass, so this theory becomes the social Define replicators, their material substrates, their replication, variation and selection algorithmically and operationally and secure them using sequence analysis. Social structures are linguistically coded and based on a digital code. We are looking for syntactic structures of a culture-encoding language. But this will not be a philosophical language, but a language that encodes and creates society. This language encodes the replication, variation, and selection of cultural replicators. On this basis, normative agents of distributed artificial intelligence, cellular automata, neural networks and other models will then be able to use protocol languages and rule systems other than heuristics in order to simulate the evolution of cultural replicators. Algorithmic structuralism moves thematically in the border area between computer science and sociology. Algorithmic structuralism assumes that social reality itself (wetware, world 2) is not capable of calculation. In its reproduction and transformation, social reality leaves traces that are purely physical and semantically unspecific (protocols, hardware, world 1). These traces can be understood as texts (discrete finite character strings, software, world 3). It is then shown that an approximation of the transformation rules of social reality (latent structures of meaning, rules in the sense of algorithms) is possible by constructing formal languages (world 3, software). This method is the Algorithmic Recursive Sequence Analysis. This linguistic structure drives the memetic reproduction of cultural replicators. This algorithmically recursive structure is of course not (sic!) compatible with Habermas and Luhmann. Galileo is not compatible with Aristotle either! Through the production of readings and the falsification of readings, the system of rules is generated informally, sequence by sequence. The informal rule system is translated into a K-system. A simulation is then carried out with the K-System. The result of the simulation, a terminal, finite character string, is statistically compared with the empirically verified trace. This does not mean that subjects in any sense of meaning follow rules in the sense of algorithms. Social reality is directly accessible only to itself. The inner states of the subjects are completely inaccessible. Statements about these inner states of subjects are derivatives of the found latent structures of meaning, rules in the sense of algorithms. Before an assumption about the inner state of a subject can be formulated, these latent structures of meaning, rules in the sense of algorithms, must first be validly determined as a space of possibility of meaning and meaning. Meaning does not mean an ethically good, aesthetically beautiful or empathetically comprehended life, but an intelligible connection, rules in the sense of algorithms. The latent structures of meaning, rules in the sense of algorithms, diachronically generate a chain of selection nodes (parameter I), whereby they synchronously generate the selection node t+1 from the selection node t at time t (parameter II). This corresponds to a context-free formal language (K-systems), which is derived from the selection node at time t generates the selection node t+1 by applying production rules. Each selection node is a pointer to recursively nested K-systems. It is possible to zoom into the case structure like with a microscope. The set of K-Systems form a Case Structure Modeling Language "CSML". The approximation can be brought as close as you like to the transformation of social reality. The productions are assigned dimensions that correspond to their empirically secured pragmatics/semantics. Topologically, they form a recursive transition network of discrete, nonmetric sets of events over which an algorithmic rule system works. K systems K are formally represented by an alphabet

$$A = \{a_1, a_2, \ldots, a_n\},$$

all words over the alphabet

$$A^*,$$

production rules

$$p,$$

dem Appearance measure

$$h$$

(pragmatics/semantics) and an axiomatic first String

$$k_0 \in A^*$$

defined:

K-System:

$$
\begin{aligned}
K &= (A, P, k_0) \\
A &= \{a_1, a_2, \ldots, a_n\} \\
P &:= A \rightarrow A \\
p_{a_i} &\in P \\
p_{a_i} &: A \times H \times A \\
H &= \{h \in N \,|\, 0 \leq h \leq 100\} \\
k_0 &\in A^* \\
k_i &\in A \quad (i \geq 1)
\end{aligned}
$$

The appearance dimension

$$h$$

can be expanded using game theory (cf. Diekmann). Starting from the axiom

$$k_0$$

, a K-system produces a string

$$k_0 k_1 k_2 ...$$

by applying the production rule p to the character i of a string:

a$_{i+1} := p_{a_i}(a_i)$
$k_i := a_{i-2}a_{i-1}a_i$
$k_{i+1} := a_{i-2}a_{i-1}a_i p_{a_i}(a_i)$

A strict measure of the reliability of the assignment of the interacts to the categories (preliminary formatives to be approximated in principle ad infinitum) is the number of assignments made by all interpreters in unison (cf. MAYRING 1990, p.94ff, LISCH/KRIZ1978, p.84ff). This number then has to be normalized by relativizing the number of performers. This coefficient is then defined with:

$$R_{\text{ars}} = \frac{N \cdot Z}{\sum_{i=1}^{N} I_i}$$

A value of R = 0.59 was measured for the example used here (see attachment in separate file and Koop,P. github)

$$R_{\text{ars}} = 0.59, \quad p = 0.05$$

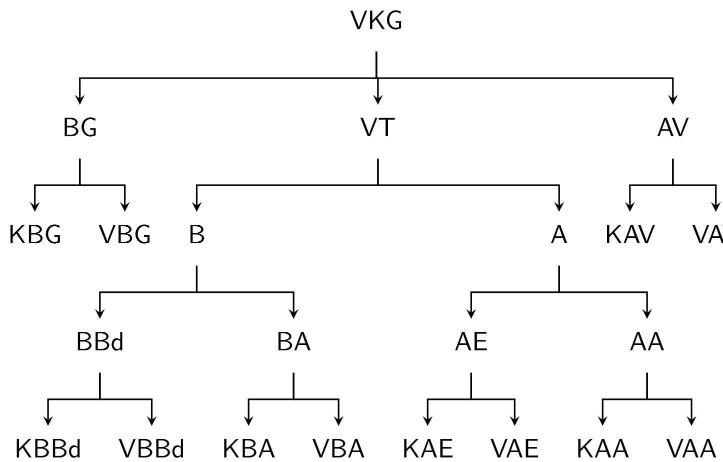
Between 1993 and 1996 I reconstructed and empirically secured a K-system for sales talks at weekly markets (Koop, P. 1993, 1994, 1995, 1996 see appendix).

The rules can be represented as a context-free grammar.

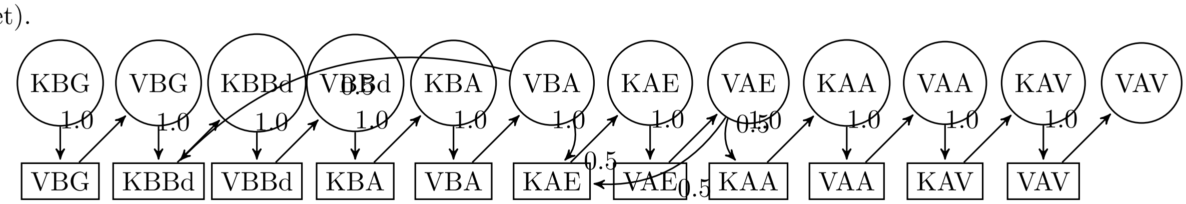
Produktionsregeln:

$$
\begin{aligned}
\text{VKG} &\rightarrow \text{BG VT AV}\\
\text{BG} &\rightarrow \text{KBG VBG}\\
\text{VT} &\rightarrow \text{B A}\\
\text{B} &\rightarrow \text{BBd BA}\\
\text{BBd} &\rightarrow \text{KBBd VBBd}\\
\text{BA} &\rightarrow \text{KBA VBA}\\
\text{A} &\rightarrow \text{AE AA}\\
\text{AE} &\rightarrow \text{KAE VAE}\\
\text{AA} &\rightarrow \text{KAA VAA}\\
\text{AV} &\rightarrow \text{KAV VAV}
\end{aligned}
$$

The grammar can be represented as a structure tree.



The corpus of terminal characters can be represented as a graph (e.g. Petri net).



The characters of the character string have no predefined meaning. Only the syntax of their combination is theoretically relevant. It defines the case structure. The semantic interpretation of the signs is solely an interpretive achievement of a human reader. In principle, a visual interpretation (which can be animated) is also possible, for example for the automatic synthesis of film sequences.:

A human reader can interpret the characters:

Verkaufsgespräche	VKG
Verkaufstätigkeit	VT
Bedarfsteil	B
Abschlußteil	A
Begrüßung	BG
Bedarf	Bd
Bedarfsargumentation	BA
Abschlußeinwände	AE
Verkaufsabschluss	AA
Verabschiedung	AV
vorangestelltes K	Kunde
vorangestelltes V	Verkäufer

Social structures and processes leave purely physically and semantically unspecific traces that can be read as protocols of their reproduction and transformation. Read in this way, the logs are texts, discrete finite character strings. The rules of reproduction and transformation can be reconstructed as probabilistic, context-free grammars or as Bayesian networks. The reconstruction then stands for a causal inference of the transformation rules of social structures and processes. In this example, the log is a tape recording of a sales pitch at a weekly market: (https://github.com/pkoopongithub/ algorithmic-recursive-sequence analysis/). The sequence analysis of the transcribed protocol and the coding with the generated categories are also stored there. The interpretation and the coding with the terminal characters is also stored in a separate file in an appendix to this Rext.

```
;; Korpus
   (define korpus (list 'KBG 'VBG 'KBBd 'VBBd 'KBA '
      VBA 'KBBd 'VBBd 'KBA 'VBA 'KAE 'VAE 'KAE 'VAE '
      KAA 'VAA 'KAV 'VAV));; 0 - 17
;; Korpus durchlaufen
    (define (lesen korpus)
     ;; car ausgeben
    (display (car korpus))
     ;; mit cdr weitermachen
     (if(not(null? (cdr korpus)))
       (lesen (cdr korpus))
       ;;(else)
     )
   )
;; Lexikon
   (define lexikon (vector 'KBG 'VBG 'KBBd 'VBBd 'KBA
      'VBA 'KAE 'VAE 'KAA 'VAA 'KAV 'VAV)) ;; 0 - 12
;; Index fuer Zeichen ausgeben
    (define (izeichen zeichen)
     (define wertizeichen 0)
     (do ((i 0 (+ i 1)))
      ( (equal? (vector-ref lexikon i) zeichen))
      (set! wertizeichen (+ 1 i))
     )
     ;;index zurueckgeben
```

```
      wertizeichen
    )

;; transformationsmatrix
    (define zeile0 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile1 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile2 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile3 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile4 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile5 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile6 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile7 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile8 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile9 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile10 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile11 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile12 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile13 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile14 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile15 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile16 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
    (define zeile17 (vector 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))

    (define matrix (vector zeile0 zeile1 zeile2 zeile3
       zeile4 zeile5 zeile6 zeile7 zeile8 zeile9
       zeile10 zeile11 zeile12 zeile13 zeile14 zeile15
       zeile16 zeile17))

```

```
;; Transformationen zaehlen
;; Korpus durchlaufen
   (define (transformationenZaehlen korpus)
     ;; car zaehlen
      (vector-set! (vector-ref matrix (izeichen (car korpus))) (izeichen (car(cdr korpus))) (+ 1 ( vector-ref  (vector-ref matrix (izeichen (car korpus))) (izeichen (car(cdr korpus))))))
     ;; mit cdr weitermachen
      (if(not(null? (cdr (cdr korpus))))
       (transformationenZaehlen (cdr korpus))
       ;;(else)
      )
   )


;; Transformation aufaddieren

;; Zeilensummen bilden und Prozentwerte bilden


;; Grammatik
   (define grammatik (list '- ))

;; aus matrix regeln bilden und regeln in grammatik einfuegene
   (define (grammatikerstellen matrix)
    (do ((a 0 (+ a 1)))
        ((= a 12) )(newline)
      (do ((b 0 (+ b 1)))
          ((= b 12))
        (if (< 0 (vector-ref  (vector-ref matrix a) b) )
         (display (cons (vector-ref lexikon a) (cons '-> (vector-ref lexikon b))))
         )
      )
    )
   )


  ;; matrix ausgeben
   (define (matrixausgeben matrix)
    (do ((a 0 (+ a 1)))
        ((= a 12) ) (newline)
```

```
      (do ((b 0 (+ b 1)))
          ((= b 12))
        (display (vector-ref  (vector-ref matrix a) b)
            )
      )
    )
  )
```

```
 (transformationenZaehlen korpus)
(grammatikerstellen matrix) (KBG -> .  VBG) (VBG
-> .  KBBd) (KBBd -> .  VBBd) (VBBd -> .  KBA) (KBA
-> .  VBA) (VBA -> .  KBBd)(VBA -> .  KAE) (KAE -> .
VAE) (VAE -> .  KAE)(VAE -> .  KAA) (KAA -> .  VAA)
(VAA -> .  KAV) (KAV -> .  VAV)
```

Figure 1: ASCII-Output des Konsolenprogramms

With this grammar and the empirical probabilities of occurrence, a transducer can then be created that simulates protocols.

```
\begin{verbatim}



(setq w3
'(
 (anfang 100 (s vkg)) ;; hier nur Fallstruktur
     Verkaufsgespraeche
 ((s vkg) 100 ende)
 )
)



(setq bbd
'(
 (kbbd 100 vbbd)
 )
)


(setq ba
'(
 (kba 100 vba)
 )
)
```

```
(setq ae
'(
(kae 100 vae)
 )
)



(setq aa
'(
 (kaa 100 vaa)
 )
)



(setq b
'(
 ((s bbd) 100 (s ba))
 )
)




(setq a
'(
 ((s ae)50(s ae))
 ((s ae)100(s aa))
 )
)


(setq vt
'(
 ((s b)50(s b))
 ((s b)100(s a))
 )
)


(setq bg
'(
```

```
 (kbg 100 vbg)
 )
)


(setq av
'(
 (kav 100 vav)
 )
)


(setq vkg
'(
 ((s bg)100(s vt))
 ((s vt)50(s vt))
 ((s vt)100(s av))
 )
)


;; Generiert die Sequenz
(defun gs (st r);; Uebergabe Sequenzstelle und Regelliste
(cond

   ;; gibt nil zur ck, wenn das Sequenzende ereicht ist
   ((equal st nil) nil)

   ;; gibt terminale Sequenzstelle mit Nachfolgern zurueck
   ((atom st)(cons st(gs(next st r(random 101))r)))

   ;; gibt expand. nichtterm. Sequenzstelle mit Nachfolger zurueck
   (t (cons(eval st)(gs(next st r(random 101))r)))
)
)

;; Generiert nachfolgende Sequenzstelle
(defun next (st r z);; Sequenzstelle, Regeln und
```

```
    Haeufigkeitsmass
(cond

  ;; gibt nil zurueck, wenn das Sequenzende erreicht ist
  ((equal r nil)nil)

  ;; waehlt Nachfolger mit Auftrittsmass h
  (
    (
      and(<= z(car(cdr(car r))))
      (equal st(car(car r)))
    )
   (car(reverse(car r)))
  )

  ;; in jedem anderen Fall wird Regelliste weiter durchsucht
  (t(next st (cdr r)z))
)
)

;; waehlt erste Sequenzstelle aus Regelliste
;;vordefinierte funktion first wird ueberschrieben, alternative umbenennen
(defun first (list)
(car(car list))
)

;; startet Simulation fuer eine Fallstruktur
(defun s (list) ;; die Liste mit dem K-System wird uebergeben
(gs(first list)list)
)


;; alternativ (s vkg) / von der Konsole aus (s w3) oder (s vkg)
(s w3)
```

A more extensive example with the brackets removed:
Now that the grammar is given, the corpus can also be parsed.

```
PROGRAM parser (INPUT,OUTPUT);
USES CRT;

```

```
CL-USER 20 > (s w3) (ANFANG ((KBG VBG) (((KBBD VBBD)
(KBA VBA)) ((KAE VAE) (KAA VAA))) (((KBBD VBBD)
(KBA VBA)) ((KAE VAE) (KAA VAA))) (((KBBD VBBD) (KBA
VBA)) ((KBBD VBBD) (KBA VBA)) ((KAE VAE) (KAA VAA)))
(((KBBD VBBD) (KBA VBA)) ((KBBD VBBD) (KBA VBA))
((KBBD VBBD) (KBA VBA)) ((KAE VAE) (KAA VAA))) (KAV
VAV)) ENDE)
```

Figure 2: ASCII-Output des Konsolenprogramms

```
KBG VBG KBBD VBBD KBA VBA KAE VAE KAA VAA KBBD VBBD
KBA VBA KBBD VBBD KBA VBA KBBD VBBD KBA VBA KAE VAE
KAA VAA KAV VAV KBG VBG KBBD VBBD KBA VBA KAE VAE
KAE VAE KAE VAE KAE VAE KAA VAA KBBD VBBD KBA VBA
KAE VAE KAE VAE KAA VAA KBBD VBBD KBA VBA KAE VAE
KAA VAA KBBD VBBD KBA VBA KBBD VBBD KBA VBA KAE VAE
KAA VAA KAV VAV KBG VBG KBBD KBA VBA KBBD VBBD KBA
VBA KAE VAE KAE VAE KAA VAA KBBD VBBD KBA VBA KBBD
KBA VBA KBBD VBBD KBA VBA KBBD VBBD KBA KAE VAE KAA
VAA KBBD VBBD KBA VBA KAE VAE KAE VAE VAE KAA VAA
KAV VAV
```

Figure 3: ASCII-Output des Konsolenprogramms

```

CONST
   c0               =      0;
   c1               =      1;
   c2               =      2;
   c3               =      3;
   c4               =      4;
   c5               =      5;
  c10               =     10;
  c11               =     11;
  cmax              =     80;
  cwort             =     20;
  CText             :     STRING(.cmax.) = '';
  datei             =     'LEXIKONVKG.ASC';
  blank             =     '␣';

  CopyRight
  =    'Demo-Parser␣Chart-Parser␣Version␣1.0(c)1992␣by␣Paul␣Koop';

TYPE
  TKategorien        = ( Leer, VKG, BG, VT, AV, B, A, BBD, BA, AE, AA,
                        KBG, VBG, KBBD, VBBD, KBA, VBA, KAE, VAE,
                        KAA, VAA, KAV, VAV);


  PTKategorienListe = ^TKategorienListe;
  TKategorienListe  = RECORD
                       Kategorie :TKategorien;
                       weiter    :PTKategorienListe;
                      END;

  PTKante           = ^TKante;
  PTKantenListe     = ^TKantenListe;

  TKantenListe      = RECORD
                       kante:PTKante;
                       next :PTKantenListe;
                      END;

  TKante            = RECORD
                       Kategorie :TKategorien;
                       vor,
                       nach,
```

```
                        zeigt      :PTKante;
                        gefunden   :PTKantenListe;
                        aktiv      :BOOLEAN;
                        nummer     :INTEGER;
                        nachkomme  :BOOLEAN;
                        CASE  Wort:BOOLEAN OF
                         TRUE :
                             (inhalt:STRING(.cwort.);)
                                ;
                         FALSE:
                             (gesucht :
                                PTKategorienListe;);
                        END;


   TWurzel    = RECORD
                  spalte,
                  zeigt     :PTKante;
                 END;

   TEintrag    = RECORD
                 A,I   :PTKante;
                 END;

   PTAgenda    = ^TAgenda;
   TAgenda     = RECORD
                  A,I  :PTKante;
                  next,
                  back : PTAgenda;
                 END;

   PTLexElem   = ^TLexElem;
   TLexElem    = RECORD
                  Kategorie: TKategorien;
                  Terminal : STRING(.cwort.);
                  naechstes: PTLexElem;
                 END;

   TGrammatik  = ARRAY (.c1..c10.)
                 OF
                 ARRAY (.c1..c4.)
                 OF TKategorien;
  CONST
   Grammatik :      TGrammatik =
               (
                (VKG, BG,      VT,    AV),
```

```
                (BG,  KBG,      VBG,   Leer),
                (VT,  B,        A,     Leer),
                (AV,  KAV,      VAV,   Leer),
                (B,   BBd,      BA,    Leer),
                (A,   AE,       AA,    Leer),
                (BBd, KBBd,     VBBd,  Leer),
                (BA,  KBA,      VBA,   Leer),
                (AE,  KAE,      VAE,   Leer),
                (AA,  KAA,      VAA,   Leer)
               );

nummer :INTEGER = c0;



VAR
 Wurzel,
 Pziel       : TWurzel;
 Pneu        : PTKante;

 Agenda,
 PAgenda,
 Paar        : PTAgenda;

 LexWurzel,
 LexAktuell,
 LexEintrag  : PTLexElem;
 Lexikon     : Text;



FUNCTION NimmNummer:INTEGER;
 BEGIN
  Nummer := Nummer + c1;
  NimmNummer := Nummer
 END;




PROCEDURE LiesDasLexikon (VAR f:Text;
                           G:TGrammatik;
                           l:PTLexElem);
  VAR
   zaehler :INTEGER;
   z11     : 1..c11;
```

```
     z4      : 1.. c4;
     ch      :   CHAR;
     st5     : STRING(.c5.);

   BEGIN
    ASSIGN(f,datei);
    LexWurzel := NIL;
    RESET(f);
    WHILE NOT EOF(f)
     DO
      BEGIN
       NEW(LexEintrag);
       IF LexWurzel = NIL
        THEN
         BEGIN
          LexWurzel := LexEintrag;
          LexAktuell:= LexWurzel;
          LexEintrag^.naechstes := NIL;
         END
        ELSE
         BEGIN
          LexAktuell^.naechstes := LexEintrag;
          LexEIntrag^.naechstes := NIL;
          LexAktuell            := LexAktuell^.naechstes;
         END;
       LexEintrag^.Terminal := '';
       st5 := '';
       FOR Zaehler := c1 to c5
        DO
         BEGIN
          READ(f,ch);
          st5 := st5 + UPCASE(ch)
         END;
       REPEAT
        READ(f,ch);
        LexEintrag^.terminal := LexEintrag^.Terminal + UPCASE(ch);
       UNTIL EOLN(f);
       READLN(f);
       IF st5 = 'KBG**' THEN  LexEintrag^.Kategorie := KBG    ELSE
       IF st5 = 'VBG**' THEN  LexEintrag^.Kategorie := VBG    ELSE
       IF st5 = 'KBBD*' THEN  LexEintrag^.Kategorie := KBBD   ELSE
```

```
      IF st5 = 'VBBD*' THEN  LexEintrag^.Kategorie :=
          VBBD   ELSE
      IF st5 = 'KBA**' THEN  LexEintrag^.Kategorie :=
          KBA    ELSE
      IF st5 = 'VBA**' THEN  LexEintrag^.Kategorie :=
          VBA    ELSE
      IF st5 = 'KAE**' THEN  LexEintrag^.Kategorie :=
          KAE    ELSE
      IF st5 = 'VAE**' THEN  LexEintrag^.Kategorie :=
          VAE    ELSE
      IF st5 = 'KAA**' THEN  LexEintrag^.Kategorie :=
          KAA    ELSE
      IF st5 = 'VAA**' THEN  LexEintrag^.Kategorie :=
          VAA    ELSE
      IF st5 = 'KAV**' THEN  LexEintrag^.Kategorie :=
          KAV    ELSE
      IF st5 = 'VAV**' THEN  LexEintrag^.Kategorie :=
          VAV
     END;
  END;


 PROCEDURE LiesDenSatz;
  VAR
   satz:         STRING(.cmax.);
   zaehler:      INTEGER;
  BEGIN
   CLRSCR;
   WRITELN(CopyRight);
   WRITE('----->␣');
   Wurzel.spalte := NIL;
   Wurzel.zeigt  := NIL;
   READLN(satz);
   FOR zaehler := c1 to LENGTH(satz)
    DO satz(.zaehler.) := UPCASE(satz(.zaehler.));
   Satz := Satz + blank;
   Writeln('----->␣',satz);
   WHILE satz <> ''
   DO
   BEGIN
      NEW(Pneu);
      Pneu^.nummer   :=NimmNummer;
      Pneu^.wort     := TRUE;
      NEW(Pneu^.gefunden);
      Pneu^.gefunden^.kante := Pneu;
      pneu^.gefunden^.next  := NIL;
```

```
      Pneu^.gesucht          := NIL;
      Pneu^.nachkomme        :=FALSE;
      IF Wurzel.zeigt = NIL
       THEN
        BEGIN
          Wurzel.zeigt := pneu;
          Wurzel.spalte:= pneu;
          PZiel.spalte := pneu;
          PZiel.zeigt  := Pneu;
          pneu^.vor    := NIL;
          Pneu^.zeigt  := NIL;
          Pneu^.nach   := NIL;
        END
       ELSE
        BEGIN
         Wurzel.zeigt^.zeigt := Pneu;
         Pneu^.vor           := Wurzel.zeigt;
         Pneu^.nach          := NIL;
         Pneu^.zeigt         := NIL;
         Wurzel.zeigt        := Wurzel.zeigt^.zeigt;
        END;
      pneu^.aktiv   := false;
      pneu^.inhalt  := COPY(satz,c1,POS(blank,satz)-c1);
      LexAktuell    := LexWurzel;
      WHILE LexAktuell <> NIL
       DO
        BEGIN
         IF LexAktuell^.Terminal = pneu^.inhalt
          Then
           BEGIN
            pneu^.Kategorie := LexAktuell^.Kategorie;
           END;
         LexAktuell := LexAktuell^.naechstes;
        END;
      DELETE(satz,c1,POS(blank,satz));
     END;
  END;




 PROCEDURE Regel3KanteInAgendaEintragen (Kante:PTKante);
  VAR
```

```
   Wurzel,
   PZiel  :TWurzel;
  PROCEDURE NeuesAgendaPaarAnlegen;
   BEGIN
    NEW(paar);
    IF Agenda = NIL
     THEN
      BEGIN
       Agenda := Paar;
       Pagenda:= Paar;
       Paar^.next := NIL;
       Paar^.back := NIL;
      END
     ELSE
      BEGIN
       PAgenda^.next := Paar;
       Paar^.next    := NIL;
       Paar^.back    := Pagenda;
       Pagenda       := Pagenda^.next;
     END;
   END;

  BEGIN
   IF Kante^.aktiv
    THEN
     BEGIN
      Wurzel.zeigt := Kante^.zeigt;
      WHILE wurzel.zeigt <> NIL
       DO
       BEGIN
        IF NOT(wurzel.zeigt^.aktiv)
         THEN
          BEGIN
           NeuesAgendaPaarAnlegen;
           paar^.A := kante;
           paar^.I := wurzel.zeigt;
          END;
       Wurzel.zeigt  := Wurzel.zeigt^.nach
       END
     END
    ELSE
    BEGIN
      PZiel.zeigt  := Kante;
      WHILE NOT(PZiel.zeigt^.Wort)
       DO PZiel.Zeigt := PZiel.Zeigt^.Vor;
      Wurzel.Zeigt    := PZiel.Zeigt;
```

```
      Wurzel.Spalte    := PZiel.Zeigt;
      PZiel.Spalte     := Pziel.zeigt;
      WHILE wurzel.spalte <> NIL
       DO
       BEGIN
        WHILE wurzel.zeigt <> NIL
        DO
        BEGIN
         IF wurzel.zeigt^.aktiv
          AND (Wurzel.zeigt^.zeigt = PZiel.spalte)
          THEN
           BEGIN
            NeuesAGendaPaarAnlegen;
            paar^.I := kante;
            paar^.A := wurzel.zeigt;
           END;
         Wurzel.zeigt  := Wurzel.zeigt^.nach
        END;
        wurzel.spalte  := wurzel.spalte^.vor;
        wurzel.zeigt   := wurzel.spalte;
       END
      END
     END;


 PROCEDURE NimmAgendaEintrag(VAR PEintrag:PTAgenda);
  BEGIN
     IF PAgenda = Agenda
     THEN
      BEGIN
       PEintrag := Agenda;
       PAgenda  := NIL;
       Agenda   := NIL;
      END
     ELSE
      BEGIN
       PAGENDA       := PAGENDA^.back;
       PEintrag      := PAgenda^.next;
       PAGENDA^.next := NIL;
      END;
  END;
```

```
PROCEDURE Regel2EineNeueKanteAnlegen( Kante     : PTKante;
                                      Kategorie : TKategorien;
                                      Gram      : TGrammatik);
 VAR
   Wurzel             :TWurzel;
   PHilfe,
   PGesuchteKategorie :PTKategorienListe;
   zaehler,
   zaehler2           :INTEGER;

 BEGIN
 Wurzel.zeigt := Kante;
 Wurzel.spalte:= Kante;
 WHILE Wurzel.zeigt^.nach <> NIL
  DO Wurzel.zeigt := Wurzel.zeigt^.nach;
  FOR zaehler := c1 To c11
   DO
    IF  (kategorie = Gram(.zaehler,c1.))
    AND (kategorie <> Leer)
     THEN
     BEGIN
      Gram(.zaehler,c1.) := Leer;
      NEW(pneu);
      Wurzel.zeigt^.nach := pneu;
      pneu^.nummer       := NimmNummer;
      pneu^.vor          := Wurzel.zeigt;
      Pneu^.nach         := NIL;
      Pneu^.zeigt        := wurzel.spalte;
      Wurzel.zeigt       := Wurzel.zeigt^.nach;
      pneu^.aktiv        := true;
      pneu^.kategorie    := kategorie;
      Pneu^.Wort         := false;
      Pneu^.gesucht      := NIL;
      Pneu^.gefunden     := NIL;
      Pneu^.nachkomme    := FALSE;
      FOR zaehler2 := c2 TO c4
       DO
       BEGIN
        IF Gram(.zaehler,zaehler2.) <> Leer
         THEN
          BEGIN
```

```
           NEW(PGesuchteKategorie);
           PGesuchteKategorie^.weiter:= NIL;
           PGesuchteKategorie^.Kategorie := Gram(.
              zaehler,zaehler2.);
           IF Pneu^.gesucht = NIL
            THEN
             BEGIN
              PHilfe         := PGesuchteKategorie;
              Pneu^.gesucht := PHilfe;
             END
            ELSE
             BEGIN
              PHilfe^.weiter := PGesuchteKategorie;
              PHilfe          := PHilfe^.weiter;
             END
          END
       END;
      Regel3KanteInAgendaEintragen (pneu);
      Regel2EineNeueKanteAnlegen(Wurzel.spalte,
                                 pneu^.gesucht^.
                                    kategorie,gram);
    END;
 END;




PROCEDURE Regel1EineKanteErweitern(paar:PTAgenda);
 VAR
  PneuHilf,Pneugefneu,AHilf :PTKantenListe;
 BEGIN

 IF paar^.I^.kategorie = paar^.A^.gesucht^.kategorie
  THEN
   BEGIN
    NEW(pneu);
    pneu^.nummer      := NimmNummer;
    pneu^.kategorie   := Paar^.A^.kategorie;

    Pneu^.gefunden := NIL;
    AHilf := Paar^.A^.gefunden;

    WHILE AHilf <> NIL
     DO
     BEGIN
      NEW(Pneugefneu);
```

```
          IF Pneu^.gefunden = NIL
           THEN
            BEGIN
             Pneu^.gefunden := Pneugefneu;
             PneuHilf       := Pneu^.gefunden;
             PneuHilf^.next := NIL;
            END
           ELSE
            BEGIN
             PneuHilf^.next   := Pneugefneu;
             PneuHilf         := PneuHilf^.next;
             PneuHilf^.next   := NIL;
            END;

          Pneugefneu^.kante      := AHilf^.kante;
          AHilf                  := AHilf^.next;
         END;

         NEW(Pneugefneu);
         IF Pneu^.gefunden = NIL
          THEN
           BEGIN
            Pneu^.gefunden := Pneugefneu;
            Pneugefneu^.next := NIL;
           END
          ELSE
           BEGIN
             PneuHilf^.next   := Pneugefneu;
             PneuHilf         := PneuHilf^.next;
             PneuHilf^.next   := NIL;
           END;
         Pneugefneu^.kante     := Paar^.I;

         Pneu^.wort              := FALSE;
         IF Paar^.A^.gesucht^.weiter = NIL
          THEN Pneu^.gesucht   := NIL
          ELSE Pneu^.gesucht   := Paar^.A^.gesucht^.weiter;
         Pneu^.nachkomme := TRUE;

        IF pneu^.gesucht   = NIL
         THEN Pneu^.aktiv := false
         ELSE Pneu^.aktiv := true;

        WHILE Paar^.A^.nach <> NIL
         DO Paar^.A       := Paar^.A^.nach;
```

```

   Paar^.A^.nach     := pneu;
   pneu^.vor         := Paar^.A;
   pneu^.zeigt       := Paar^.I^.zeigt;
   pneu^.nach        := NIL;

   Regel3KanteInAgendaEintragen (pneu);
   IF Pneu^.aktiv
    THEN Regel2EineNeueKanteAnlegen(Pneu^.zeigt,
                               pneu^.gesucht^.kategorie,
                               Grammatik);
  END;


END;

PROCEDURE SatzAnalyse;
 BEGIN
 WHILE Agenda <> NIL
 DO
  BEGIN
   NimmAgendaEintrag(Paar);
   Regel1EineKanteErweitern(Paar);
  END;

 END;

PROCEDURE GibAlleSatzalternativenAus;
 CONST
  BlankAnz:INTEGER = c2;
 VAR
  PHilf   :PTkantenListe;

 PROCEDURE SatzAusgabe(Kante:PTKante;BlankAnz:INTEGER);
  VAR

  Zaehler:INTEGER;
  PHilf  :PTKantenListe;
  BEGIN
   FOR Zaehler := c1 TO BlankAnz DO WRITE(blank);

   IF Kante^.kategorie = VKG     THEN WRITELN ('VKG ') ELSE
   IF Kante^.kategorie = BG      THEN WRITELN ('BG 
```

```
␣') ELSE
        IF Kante^.kategorie = VT      THEN WRITELN ('VT␣␣') ELSE
        IF Kante^.kategorie = AV      THEN WRITE   ('AV␣␣') ELSE
        IF Kante^.kategorie = B       THEN WRITELN ('B␣␣␣') ELSE
        IF Kante^.kategorie = A       THEN WRITE   ('A␣␣␣') ELSE
        IF Kante^.kategorie = BBD     THEN WRITE   ('BBD␣') ELSE
        IF Kante^.kategorie = BA      THEN WRITELN ('BA␣␣') ELSE
        IF Kante^.kategorie = AE      THEN WRITE   ('AE␣␣') ELSE
        IF Kante^.kategorie = AA      THEN WRITE   ('AA␣␣') ELSE
        IF Kante^.kategorie = KBG     THEN WRITELN ('KBG␣') ELSE
        IF Kante^.kategorie = VBG     THEN WRITELN ('VBG␣') ELSE
        IF Kante^.kategorie = KBBD    THEN WRITELN ('KBBD') ELSE
        IF Kante^.kategorie = VBBD    THEN WRITE   ('VBBD') ELSE
        IF Kante^.kategorie = KBA     THEN WRITELN ('KBA␣') ELSE
        IF Kante^.kategorie = VBA     THEN WRITE   ('VBA␣') ELSE
        IF Kante^.kategorie = KAE     THEN WRITE   ('KAE␣') ELSE
        IF Kante^.kategorie = VAE     THEN WRITELN ('VAE␣') ELSE
        IF Kante^.kategorie = KAA     THEN WRITE   ('KAA␣') ELSE
        IF Kante^.kategorie = VAA     THEN WRITE   ('VAA␣') ELSE
        IF Kante^.kategorie = KAV     THEN WRITE   ('KAV␣') ELSE
        IF Kante^.kategorie = VAV     THEN WRITE   ('VAV␣');

        IF Kante^.wort
         THEN
          WRITELN('---->␣',Kante^.inhalt)
         ELSE
```

```
        BEGIN
        PHilf := Kante^.gefunden;
        WHILE PHilf <> NIL
         DO
          BEGIN
           Satzausgabe(PHilf^.kante,Blankanz+c1);
           PHilf := Philf^.next;
          END
        END
    END;

    BEGIN
      WHILE Wurzel.zeigt^.vor <> NIL
       DO Wurzel.zeigt := Wurzel.zeigt^.vor;

      WHILE Wurzel.zeigt <> NIL
      DO
      BEGIN
       IF (Wurzel.zeigt^.kategorie = VKG)
         AND ((NOT(Wurzel.zeigt^.aktiv))
         AND (wurzel.zeigt^.zeigt = NIL))
         THEN
          BEGIN
           WRITELN('VKG');
           PHilf := Wurzel.zeigt^.gefunden;
           WHILE PHilf <> NIL
            DO
             BEGIN
              Satzausgabe(PHilf^.kante,Blankanz+c1);
              PHilf := Philf^.next;
             END
          END;
      Wurzel.zeigt := Wurzel.zeigt^.nach;
      END;

    END;

  PROCEDURE LoescheDieListe;
   PROCEDURE LoescheWort(kante :PTKante);
    PROCEDURE LoescheSpalte(kante:PTKante);
     VAR
      Pgefunden :PTKantenListe;
      Pgesucht  :PTKategorienListe;
     PROCEDURE LoescheGesucht(p:PTKategorienListe);
      BEGIN
       IF p^.weiter <> NIL
```

```
      THEN LoescheGesucht(p^.weiter);
     IF P <> NIL THEN DISPOSE(P);
    END;
   PROCEDURE LoescheGefunden(Kante:PTKante;p:PTKantenListe);
    BEGIN
     IF p^.next <> NIL
      THEN LoescheGefunden(Kante,p^.next);
     DISPOSE(P);
    END;
   BEGIN (*LoescheSpalte*)
    IF Kante^.nach <> NIL
     THEN LoescheSpalte(kante^.nach);
    IF (NOT Kante^.nachkomme) AND ((Kante^.gesucht <> NIL)
     AND (NOT Kante^.wort))
     THEN LoescheGesucht(Kante^.gesucht);
    IF Kante^.gefunden <> NIL
     THEN LoescheGefunden(Kante,Kante^.gefunden);
    DISPOSE(Kante)
   END; (*LoescheSpalte*)
  BEGIN (*LoescheWort*)
   IF Kante^.zeigt <> NIL
    THEN LoescheWort(Kante^.zeigt);
  LoescheSpalte(Kante);
  END; (*LoescheWort*)
 BEGIN (*LoescheDieListe*)
  WHILE Wurzel.spalte^.vor <> NIL
   DO Wurzel.spalte := Wurzel.spalte^.vor;
  LoescheWort(Wurzel.spalte);
 END; (*LoescheDieListe*)

BEGIN
 Agenda := NIL;
 PAgenda := Agenda;
 LiesDasLexikon(Lexikon,Grammatik,LexWurzel);
 LiesDenSatz;
 WHILE Wurzel.spalte^.vor <> NIL
  DO Wurzel.spalte := Wurzel.spalte^.vor;
 Regel2EineNeueKanteAnlegen(Wurzel.spalte,VKG,Grammatik);
 SatzAnalyse;
 GibAlleSatzalternativenAus;
 LoescheDieListe;
```

```
END.
```

Demo-Parser Chart-Parser Version 1.0(c)1992 by Paul Koop - - - - - > KBG VBG KBBD KBA VBA KAE VAE KAA VAA KAV VAV - - - - - > KBG VBG KBBD KBA VBA KAE VAE KAA VAA KAV VAV VKG BG KBG - - - - > KBG VBG - - - - > VBG VT B BBD KBBD - - - - > KBBD VBBD - - - - > VBBD BA KBA - - - - >. KBA VBA - - - - > VBA A AE KAE - - - - > KAE VAE - - - - > VAE AA KAA - - - - > KAA VAA - - - - > VAA AV KAV - - - - > KAV VAV - - - - > VAV


```
Demo-Parser Chart-Parser Version 1.0(c)1992 by Paul
Koop - - - - - > KBG VBG KBBD KBA VBA KAE VAE KAA
VAA KAV VAV - - - - - > KBG VBG KBBD KBA VBA KAE VAE
KAA VAA KAV VAV VKG BG KBG - - - - > KBG VBG - - -
- > VBG VT B BBD KBBD - - - - > KBBD VBBD - - - - >
VBBD BA KBA - - - - >.  KBA VBA - - - - > VBA A AE
KAE - - - - > KAE VAE - - - - > VAE AA KAA - - - - >
KAA VAA - - - - > VAA AV KAV - - - - > KAV VAV - - -
- > VAV
```


Figure 4: ASCII-Output des Konsolenprogramms

```
import re

# Lesen des Korpus aus einer Datei
#with open("VKGKORPUS.TXT", "r") as f:
#    korpus = f.read()
korpus = "KBG␣VBG␣KBBD␣VBBD␣KBA␣VBA␣KAE␣VAE␣KAA␣VAA␣
    KBBD␣VBBD␣KBA␣VBA␣KBBD␣VBBD␣KBA␣VBA␣KBBD␣VBBD␣KBA␣
    VBA␣KAE␣VAE␣KAA␣VAA␣KAV␣VAV"
# Extrahieren der Terminalsymbole aus dem Korpus
terminals = re.findall(r"[KV][A-Z]+", korpus)

# Entfernen der vorangestellten K- oder V-Zeichen aus
    den Terminalsymbolen
non_terminals = list(set([t[1:] for t in terminals]))

# Erzeugen der Regelproduktionen
productions = []
for nt in non_terminals:
    rhs = [t for t in terminals if t[1:] == nt]
    productions.append((nt, rhs))

# Ausgabe der Grammatikregeln
```

```python
print("Regeln:")
for nt, rhs in productions:
    print(nt + "␣->␣" + "␣|␣".join(rhs))

# Ausgabe der Startsymbol
print("Startsymbol:␣VKG")
```

```
 Regeln:  AV -> KAV | VAV BG -> KBG | VBG AA -> KAA
| VAA | KAA | VAA AE -> KAE | VAE | KAE | VAE BA ->
KBA | VBA | KBA | VBA | KBA | VBA | KBA | VBA BBD
-> KBBD | VBBD | KBBD | VBBD | KBBD | VBBD | KBBD |
VBBD Startsymbol:  VKG
```

Figure 5: ASCII-Output des Konsolenprogramms

A probabilistic context-free grammar with weighted productions can also be induced from the corpus:

```python
from collections import defaultdict
import random

# define the grammar production rules
grammar = defaultdict(list)

# read in the corpus
corpus = "KBG␣VBG␣KBBD␣VBBD␣KBA␣VBA␣KAE␣VAE␣KAA␣VAA␣
    KBBD␣VBBD␣KBA␣VBA␣KBBD␣VBBD␣KBA␣VBA␣KBBD␣VBBD␣KBA␣
    VBA␣KAE␣VAE␣KAA␣VAA␣KAV␣VAV".split()

# get the non-terminal symbols
nonterminals = set([symbol[1:] for symbol in corpus if
     symbol.startswith("K") or symbol.startswith("V")])

# iterate over the corpus and count the production
    rules
for i in range(1, len(corpus)):
    curr_symbol = corpus[i]
    prev_symbol = corpus[i-1]
    if prev_symbol.startswith("K") or prev_symbol.
        startswith("V"):
         grammar[prev_symbol[1:]].append(curr_symbol)

# calculate the probabilities for the production rules
for lhs in grammar.keys():
    productions = grammar[lhs]
    total_count = len(productions)
```

```python
    probabilities = defaultdict(float)
    for rhs in productions:
        probabilities[rhs] += 1.0
    for rhs in probabilities.keys():
        probabilities[rhs] /= total_count
    grammar[lhs] = probabilities

# print the grammar
print("Grammar:")
for lhs in grammar.keys():
    print(lhs + "␣->")
    for rhs in grammar[lhs].keys():
        print("␣␣" + rhs + "␣:␣" + str(grammar[lhs][rhs]))
```

```
 Grammar:  BG -> VBG : 0.5 KBBD : 0.5 BBD -> VBBD
: 0.5 KBA : 0.5 BA -> VBA : 0.5 KAE : 0.25 KBBD :
0.25 AE -> VAE : 0.5 KAA : 0.5 AA -> VAA : 0.5 KBBD
: 0.25 KAV : 0.25 AV -> VAV : 1.0
```

Figure 6: ASCII-Output des Konsolenprogramms

A probabilistic grammar can be interpreted as a Bayesian network. In a Bayesian network, the dependencies between the variables are modeled by directed edges, while the probabilities of the individual variables and edges are represented by probability distributions. In a probabilistic grammar, the production rules are modeled as variables and the terms and nonterminals as states. Every production has a certain probability, which can be represented by a probability distribution. The probability of generating a certain sentence can then be calculated by the production rules and their probabilities. The states in the probabilistic grammar can be interpreted as nodes in the Bayesian network, while the production rules can be represented as directed edges. The probabilities of the production rules can then be modeled as edge conditions. By computing the posterior probability, a probabilistic prediction can then be made as to which proposition is most likely given the observations. The corpus can be understood as a log of the mutual interaction of two software agents in a multi-agent system. The agents of this multi-agent system have access to the last generated terminal character and the probabilistic grammar, which can be interpreted as a Bavarian network. They use this knowledge to generate the next terminal character. An agent K generates the buyer terminal characters. An agent V generates the vendor terminal characters. The corpus can be understood as a log of the mutual interaction of two software agents in a multi-agent system. The agents of this multi-agent system have access to the last generated terminal character and the probabilistic grammar, which can be interpreted as a Bavarian network. They use this knowledge to generate the next terminal character. An agent K generates the buyer terminal characters. An agent V generates the vendor terminal characters. The corpus can be understood as a log of the mutual interaction of two software agents in a multi-agent system. The agents of this multi-agent system have access to the last generated terminal character and the probabilistic grammar, which can be interpreted as a Bavarian network. They use this knowledge to generate the next terminal character. An agent K generates the buyer terminal characters. An agent V generates the vendor terminal characters.

```python
import random

# Die gegebene probabilistische Grammatik
grammar = {
    'BG': {'VBG': 0.5, 'KBBD': 0.5},
    'BBD': {'VBBD': 0.5, 'KBA': 0.5},
    'BA': {'VBA': 0.5, 'KAE': 0.25, 'KBBD': 0.25},
    'AE': {'VAE': 0.5, 'KAA': 0.5},
    'AA': {'VAA': 0.5, 'KAV': 0.25, 'KBBD': 0.25},
    'AV': {'VAV': 1.0},
}

# Zuf llige Belegung von Ware und Zahlungsmittel bei
    den Agenten
agent_k_ware = random.uniform(0, 100)
agent_k_zahlungsmittel = 100 - agent_k_ware
agent_v_ware = random.uniform(0, 100)
agent_v_zahlungsmittel = 100 - agent_v_ware

# Entscheidung  ber  die Rollenverteilung basierend
    auf Ware und Zahlungsmittel
if agent_k_ware > agent_v_ware:
    agent_k_role = 'K ufer'
    agent_v_role = 'Verk ufer'
else:
    agent_k_role = 'Verk ufer'
    agent_v_role = 'K ufer'

# Ausgabe der Rollenverteilung und der Belegung von
    Ware und Zahlungsmittel
print("Agent␣K:␣Rolle␣=", agent_k_role, "|␣Ware␣=",
    agent_k_ware, "|␣Zahlungsmittel␣=",
    agent_k_zahlungsmittel)
print("Agent␣V:␣Rolle␣=", agent_v_role, "|␣Ware␣=",
    agent_v_ware, "|␣Zahlungsmittel␣=",
    agent_v_zahlungsmittel)
print()
```

```python

# Agent K startet den Dialog mit dem Terminalzeichen 'KBG'
last_terminal = 'KBG'

# Maximale Anzahl von Terminalzeichen im Dialog
max_terminals = 10

# Dialog-Schleife
for i in range(max_terminals):
    # Agent K generiert das n chste Terminalzeichen basierend auf der Grammatik und dem letzten Terminalzeichen
    next_terminal = random.choices(list(grammar[last_terminal].keys()), weights=list(grammar[last_terminal].values()))[0]

    # Agent V generiert das n chste Terminalzeichen basierend auf der Grammatik und dem letzten Terminalzeichen
    next_terminal = random.choices(list(grammar[last_terminal].keys()), weights=list(grammar[last_terminal].values()))[0]

    # Aktualisierung des letzten Terminalzeichens
    last_terminal = next_terminal

    # Ausgabe des aktuellen Terminalzeichens
    print("Agent K:", next_terminal)

    # Break, wenn das Terminalzeichen 'VAV' erreicht ist
    if next_terminal == 'VAV':
        break
```

```
Agent K: KBBD Agent V: VBBD Agent K: KBA Agent
V: VAE Agent K: KBBD Agent V: VBBD Agent K: KBA
Agent V: VBBD Agent K: KBA Agent V: VAE Agent K:
KAA Agent V: VAA Agent K: KBBD Agent V: VBBD
Agent K: KBA Agent V: VAE Agent K: KAA Agent
V: VAA Agent K: KAA Agent V: VAA Agent K: KAA
Agent V: VAA Agent K: KAV Agent V: VAV Agent
K: Rolle = Verkäufer | Ware = 60.935380690830155 |
Zahlungsmittel = 39.064619309169845 Agent V: Rolle =
Käufer | Ware = 46.51117771417693 | Zahlungsmittel =
53.48882228582307
Agent K: KBBD Agent V: VBBD Agent K: KBA Agent V:
VAE Agent K: KBBD Agent V: VBBD Agent K: KBA Agent
V: VBBD Agent K: KBA Agent V: VAE Agent K: KAA Agent
V: VAA Agent K: KBBD Agent V: VBBD Agent K: KBA
Agent V: VAE Agent K: KAA Agent V: VAA Agent K: KAA
Agent V: VAA Agent K: KAA Agent V: VAA Agent K: KAV
Agent V: VAV
```

Figure 7: ASCII-Output des Konsolenprogramms

Literatur

- Alpaydin, E.: *Maschinelles Lernen*, 2008
- Chomsky, N.: *Aspects of the Theory of Syntax*, 1965
- Dehmer, Matthias: *Strukturelle Analyse Web-basierter Dokumente*, 2005
- Diekmann, A.: *Spieltheorie: Einführung, Beispiele, Experimente*, 2009
- Gold, E. Mark: *Limiting Recursion, The Journal of Symbolic Logic 30: 28–48*, 1965
- Gold, E. Mark: *Language Identification in the Limit, Information and Control 10: 447–474*, 1967
- Koop, P.: *Über die Entscheidbarkeit der GTG*, 1994
- Koop, P.: *Rekursive Strukturen und Prozesse*, 1995
- Koop, P.: *K-Systeme: Das Projekt ARS*, 1994
- Koop, P.: *Algorithmisch Rekursive Sequenzanalyse*, 1996
- Koop, P.: *Oevermann, Chomsky, Searle*, 1994
- Koop, P.: https://github.com/pkoopongithub/ algorithmisch-rekursive-sequenzanalyse/)

- Krauße, C. C., & Krueger, F. R.: *Unbekannte Signale, Spektrum Dossier 2/2002*
- Krempel, Rasmus: *Netze, Karten, Irrgärten: Graphenbasierte explorative Ansätze zur Datenanalyse und Anwendungsentwicklung in den Geisteswissenschaften*, 2016
- Lisch, R., Kriz, J.: *Grundlagen und Modelle der Inhaltsanalyse*, 1978
- Mayring, P.: *Einführung in die qualitative Sozialforschung*, 1990
- Ndiaye, Alassane: *Rollenübernahme als Benutzermodellierungsmethode: globale Antizipation in einem transmutierbaren Dialogsystem*, 1998
- Nevill-Manning Witten: *Identifying Hierarchical Structure in Sequences: A linear-time algorithm*, 1999
- Oevermann, U.: *Die objektive Hermeneutik als unverzichtbare methodologische Grundlage für die Analyse von Subjektivität. Zugleich eine Kritik an der Tiefenhermeneutik, in: Jung, Th., Müller-Dohm, St. (Hg): »Wirklichkeit« im Deutungsprozess: Verstehen und Methoden in den Kultur- und Sozialwissenschaften*, Frankfurt 1993
- Shen, Chunze: *EDSI - Effiziente Grammatikinduktion*, 2013