九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Suchen in natürlichen Texten (I): Verfahren und Hilfen zur Datenverarbeitung in Sprach- und Literaturwissenschaft

Michel, Wolfgang
College of General Education, Kyushu University

https://hdl.handle.net/2324/9504

出版情報：KAIROS. (20), pp.161-190, 1982-11-20. Kairos Gesellschaft fuer Germanistik, Fukuoka
バージョン：
権利関係：

昭和 57 年 11 月 20 日 発行
「かいろす」第 20 号 別 冊
Kairos 20 (1982)

Suchen in natürlichen Texten (I)
—Verfahren und Hilfen zur Datenverarbeitung in Sprach- und Literaturwissenschaft—

Wolfgang Michel

Kairos-Gesellschaft für Germanistik
Fukuoka Japan 1982

Suchen in natürlichen Texten (I)
—Verfahren und Hilfen zur Datenverarbeitung in Sprach- und Literaturwissenschaft—

Wolfgang Michel

Das Durchsuchen großer Textmengen zu Referenzzwecken gehört in sprachorientierten Wissenschaften wohl zu den häufigsten Tätigkeiten. Großrechenanlagen bieten zwar oft einige Hilfen in dieser Hinsicht in Form von 'Editoren' mit Suchkommandos u. ä., die Erfahrung jedoch zeigt, daß viele Fragestellungen der Sprach- und Literaturwissenschaft andere Instrumente verlangen, von den kleineren Anlagen oder dem Microcomputer ganz zu schweigen. Andererseits finden sich in der Computer-Fachliteratur nur verstreut konkrete Anweisungen zur Entwicklung geeigneter Software. Und oft sind die Konzeptskizzen so abstrakt, daß ihre Umsetzung außerordentlich aufwendig wird. In dieser Arbeit versuche ich daher zweierlei: zum einen soll das Suchen in natürlichen Texten ein wenig systematisiert werden, zum anderen möchte ich eigene Ansätze und solche aus der spezielleren Fachliteratur soweit ordnen und weiterentwickeln, daß sich auf dieser Grundlage relativ leicht eigene Programme für die jeweilige spezifische Fragestellung aufstellen lassen.

1.0 Vorüberlegungen

Vom 'planlosen Suchen' abgesehen geht man auch im alltäglichen Leben zumeist von bestimmten Annahmen über die Art des Gesuchten, die Wahrscheinlichkeit seines Vorkommens aus. Manchmal gelangt man rascher ans Ziel, wenn man sich über das klar ist, was man nicht sucht. Oder man findet zwar nicht den erhofften Gegen-

stand, aber doch einen sehr ähnlichen, mit dem man auch zufrieden sein kann. Falls ich mehrere Dinge suche, sollte ich das gleichzeitig tun, um den Aufwand zu verringern. Wenn ein bestimmter Suchbereich des öfteren durchforstet wird, könnte ich mir eine Methode dafür ausdenken, ich müßte dies sogar tun, wenn die Suche umfassend sein sollte. Zuweilen hat man jedoch keine festumrissenen Vorstellungen von dem, was man finden wird, und man begnügt sich mit einigen Minimalkriterien.

Dies alles trifft auch auf die Verarbeitung von Textdaten zu, die ich nachfolgend nur in der Form graphisch fixierter Ketten behandele:

(1) Ein 'Alphabet' ist eine endliche (finite) Menge von Symbolen.

(2) Eine Kette (string) X der Länge n über einem Alphabet I ist eine endliche Folge von Symbolen $x_1, x_2, x_3 \ldots, x_n$ aus I ($x_i \in I$ für $1 \leq i \leq n$).

(3) Die leere Kette der Länge 0 enthält keine Symbole und wird mit ϵ bezeichnet.

(4) Durch Verkettung (concatenation) zweier Ketten X und Y entsteht die Kette XY.

(5) Sei XYZ eine Kette, dann ist X ein Präfix, Y eine Unterkette (substring) und Z ein Suffix von XYZ.

Ein Text läßt sich so entweder als elementare Kette interpretieren oder durch Verkettung einer begrenzten Menge von Teilketten konstituieren. Für den Vergleich zweier (Teil-)Ketten gilt:

(6) Zwei Ketten $X = x_1, x_2, \ldots x_n$ und $Y = y_1, y_2, \ldots y_m$ sind dann gleich, wenn $n = m$ und $x_i = y_i$ für $1 \leq i \leq n$.

Elektronisch gespeicherte Texte sind Ketten über einem 'Alphabet', dessen Symbole durch die üblichen Codierungskonventionen der jeweiligen Rechenanlagen vorgegeben sind, normalerweise entweder im Extended Binary Coded Decimal Interchange Code (EBCDIC) oder dem American Standard Code for Information Interchange (ASCII). Da die Maschine nur binäre Zustände erfaßt, wird jedem Symbol, vereinfachend gesagt, ein bestimmter Zahlenwert zugewiesen, auch den Ziffern, den Satzzeichen und dem die Wortgrenze

erste und zweite Dezimalstelle \ dritte Dezimalstelle	1	2	3	4	5	6	7	8	
6				SP	｡	「	」	＼	･
7	ｦ	ｱ	ｲ	ｳ	¢ / £	．	く	(	+	｜
8	&	ｴ	ｵ	ｬ	ｭ	ｮ	ツ		—	
9	!	¥	*	)	;	¬	—	／		
10							,	%	under line _
11	>	?							
12			:	#	@	'	=	°		ｱ / a
13	ｲ / b	ｳ / c	ｴ / d	ｵ / e	ｶ / f	ｷ / g	ｸ / h	ｹ / i	ｺ	
14	ｻ	ｼ	ｽ	ｾ	ｿ	ﾀ / j	ﾁ / k	ﾂ / l	ﾃ / m	ﾄ / n
15	ﾅ / o	ﾆ / p	ﾇ / q	ﾈ / r	ﾉ			ﾊ	ﾋ	ﾌ
16			ﾍ / s	ﾎ / t	ﾏ / u	ﾐ / v	ﾑ / w	ﾒ / x	ﾓ / y	ﾔ / z
17	ﾕ		ﾖ	ﾗ	ﾘ	ﾙ				
18							ﾚ	ﾛ	ﾜ	ﾝ
19	ﾞ	｡	｜	A	B	C	D	E	F	G
20	H	I							｜	J
21	K	L	M	N	O	P	Q	R		
22					$		S	T	U	V
23	W	X		Z						
24	ϕ	1	2	3	4	5	6	7		9

Abb. 1 Die Symbole des EBCDIC einschließlich der in Japan verbreiteten Katakana-Kodifizierung; der numerische Wert der einzelnen Zeichen - normalerweise hexadezimal notiert, hier jedoch ins Dezimalsystem umgeschrieben - ergibt sich aus der Kombination der Dezimalstellen: A=193, a=129 etc.
Die Steuerzeichen etc. des Bereichs bis 64 wurden weggelassen.

markierenden Leerzeichen ('blank' oder 'space', in Abb. 1 als SP abgekürzt).

Nun wurde in Japan der ursprünglich für das Englische konzipierte EBCDI-Code um die Katakana-Silbenschrift-Symbole erweitert, dies aber unglücklicherweise so, daß sich bestimmte Kana-Zeichen und lateinische Kleinbuchstaben überlappen und bei so manchem Display-System oder Zeilendrucker keine 'gemischten' Texte möglich sind, in denen auch Minuskel auftreten (s. Abb. 1). Doch das störte erst, als sich ein Weg zur Verarbeitung der zahlreichen chinesischen Charaktere ('Kanji') und damit zur Entwicklung von 'word processors' für die neben den chinesischen Zeichen auch

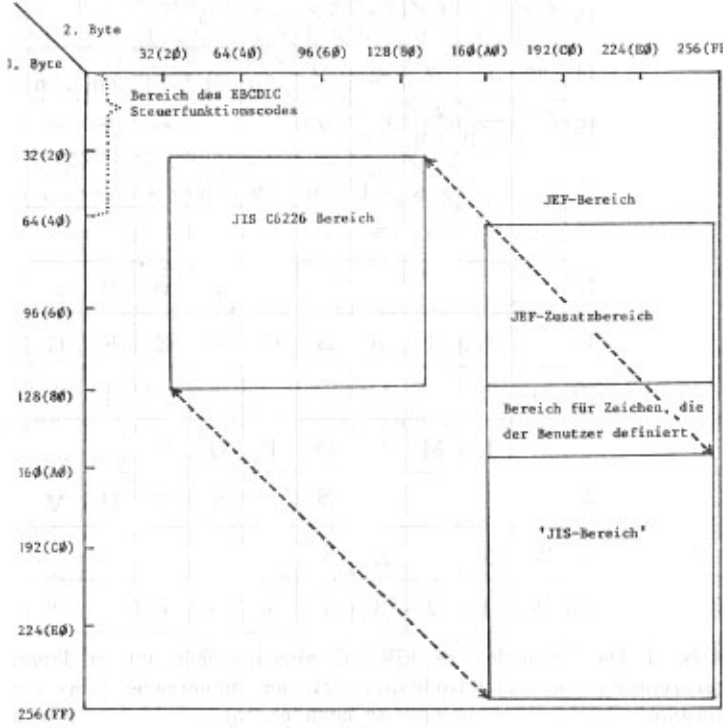


Abb. 2 JIS C6226-Code und JEF-Code; die Zahlenwerte sind dezimal (hexa-dezimal) notiert.

zwei Silbenalphabete und häufig lateinische Buchstaben umfassenden japanischen Gebrauchstexte abzeichnete. Man entschloß sich dann zur Kodifizierung in 16 Bits gegenüber den bis dahin üblichen 8 Bits und nahm 1978 in den JIS C6226 genannten Standard gleich diverse westliche Alphabete (Griechisch, Lateinisch, Russisch) auf. Trägt man nun den ersten und zweiten Byte (je 8 Bits) gegeneinander auf, so veranschaulicht sich der numerische Bereich dieses Kodes als Fläche; der traditionell in einem Byte untergebrachte EBCDIC fällt als Linie auf die linke Senkrechte. Besonders wegen der Überlappung von EBCDIC-Steuerzeichen und dem ersten Byte bestimmter JIS-Zeichen sind dann wieder firmeneigene Systeme entstanden, wie das JEF (Japanese processing Extended Feature), die nur z. T. in den JIS transponiert werden können (s. Abb. 2). Die nachfolgenden Überlegungen und Vorschläge erstrecken sich zwar implizit auch auf Kanji-Daten und gemischte Systeme, die Programme sind jedoch aus technischen Gründen in dieser Form nur auf 8-Bit-Codes anwendbar.

2.0 Sequentielle Suche

Beim Durchsuchen natürlicher Texte möchte ich grob zwei Verfahrensweisen unterscheiden: (I) das lineare Vorgehen in der Reihenfolge des Textablaufes und (II) andere Verfahren wie Hash-Coding u. ä., die als 'nicht sequentiell' zusammengefaßt werden sollen. In diesem Teil der Überlegungen konzentriere ich mich auf die sequentielle Methode und sehe auch von Wortformenlisten mit Adressenangaben und ähnlichem ab.

2.1 Ermittlung des 'Alphabets' von Textketten

In der Praxis wird wie beschrieben die Identität bzw. Verschiedenheit zweier Symbole durch den Vergleich der sie repräsentierenden Zahlenwerte ermittelt. Die Majuskel A ist dabei um 64 größer als die Minuskel, die Ziffer 5 hat im EBCDIC mit 245 einen viel höheren Wert als die Zahl 5.

Diese Tatsache können wir elegant nutzen, falls wir beispiels-

weise wissen wollen, welche Symbole überhaupt in einem Text und in welcher Verteilung sie vorhanden sind, d. h. wie das 'Alphabet' der Kette aussieht. In den üblichen Programmieranleitungen pflegt man hierzu eine Liste (table) des vermuteten Inventars aufzustellen, dann für jedes Textsymbol nachzuprüfen, ob es in der Liste vorhanden ist und ggf. dessen Zähler (counter) um den Wert 1 zu erhöhen. Für einen Text von sagen wir 1000 Seiten, also rund 500 000 Symbolen kann man sich leicht ausmalen, welch ein Aufwand pro Symbol getrieben werden muß, egal wie auch immer die Suchliste konstruiert wäre. Bei einem Text im EBCDIC z. B. genügte es jedoch völlig, einen linearen Array ALPHABET mit 249 Zellen vorzusehen, die zunächst alle auf den Wert Null gestellt werden (s. Abb. 3).

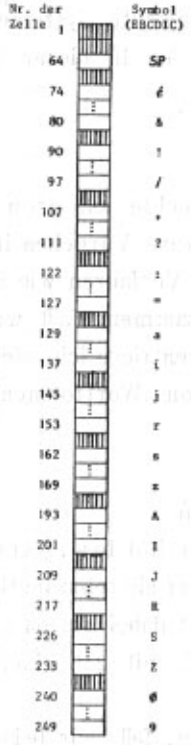


Abb. 3 Array zur Registrierung der Struktur und Verteilung des 'Alphabets' von Textketten; die schraffierten Intervalle werden im EBCDIC normalerweise nicht genutzt, für ASCII benötigt man nur den Bereich zwischen Zelle 32 und 126

Nun verarbeitet man Textzeichen für Textzeichen, indem man einfach den Wert derjenigen Zelle um eins vermehrt, deren Positionsnummer dem numerischen Wert des Textzeichens entspricht, für 'W' also : ALPHABET(230) = ALPHABET(230) +1, bzw. allgemeiner für ein beliebiges Textsymbol SYMBOL formuliert: ALPHABET(SYMBOL) = ALPHABET(SYMBOL) +1. Zum Schluß überspringt man alle Zellen, deren Wert immer noch auf Null steht, und verarbeitet die anderen Daten je nach Zielsetzung weiter.*

2.2 Suchen nach vorgegebenen einstelligen Ketten

Angenommen, ich suche in einer Textkette eine ganz bestimmte Teilkette (key, Suchbegriff) der Länge 1. Dem herkömmlichsten Verfahren gemäß liest man den Text zeilenweise (abschnittsweise, blockweise) in den Kernteil der Rechenanlage (central processing unit) ein, um dann Zeichen für Zeichen mit dem ebenfalls eingelesenen, gesuchten Symbol zu vergleichen. In der Form eines Prozessfließbildes sähe dies wie in Abb. 4a aus.

Ein dementsprechendes Rechenprogramm benötigte für die 80 Symbole einer Zeile 160 Vergleiche, da man jedes Mal auch kontrollieren muß, ob man bereits das Zeilenende erreicht hat oder nicht. Liest man jedoch als 81. Zeichen den gesuchten Key ein, so braucht man nur noch in den Fällen, in denen der Vergleich positiv ausfiel, nachzuprüfen, ob es sich nicht möglicherweise um dieses letzte Symbol handelt (s. Abb. 4b).

Enthält aber die gesuchte Kette mehr als ein Symbol, so verliert dieser kleine Kunstgriff an Wert, tauchen doch im Vergleichsverfahren selbst neue Aspekte auf. Ein 'Zeiger' (pointer ; in der Abbildung 5 als ↑ dargestellt) wird nötig, um jeweils die erste Stelle

* Dies Verfahren funktioniert am einfachsten in FORTRAN 66, wo auch nichtnumerische Symbole direkt als 'Integer' eingelesen werden; in anderen, 'moderneren' Programmiersprachen wie PL/1 muß man die 'Character'-Variablen erst entsprechend in ihre numerische Äquivalente konvertieren. Dies belegt einmal mehr, daß scheinbar leicht zu schreibende Sprachen manchmal sehr unbequeme Seiten haben können.

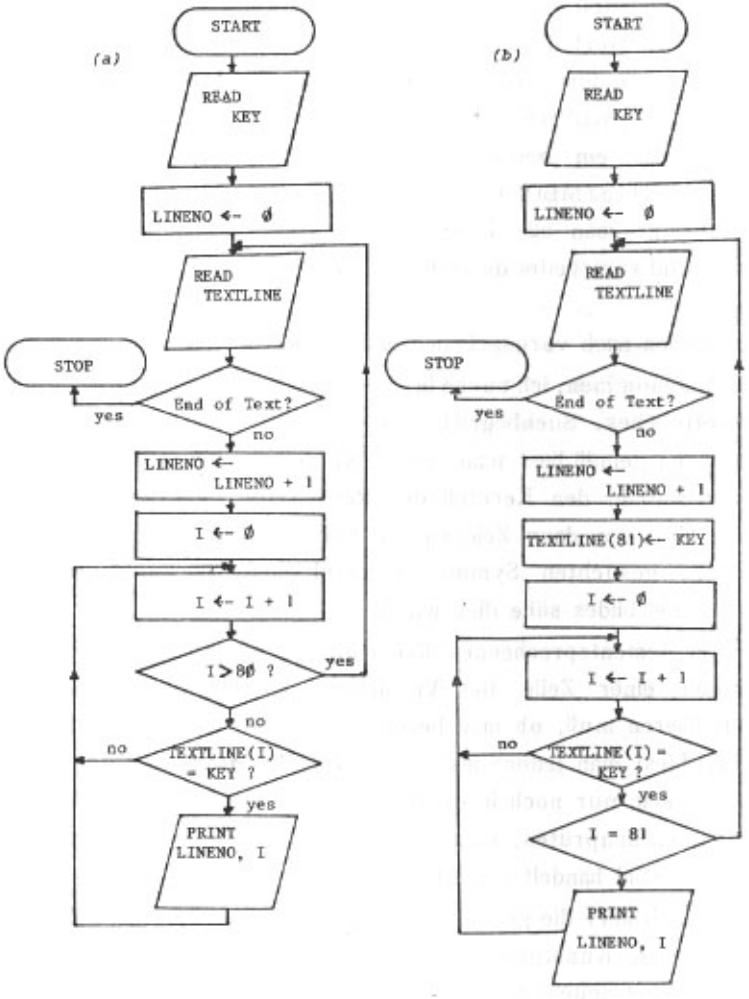


Abb. 4 Fließbild zur Suche einstelliger Ketten (KEY)

in der Eingabekette zu kennzeichnen, die mit dem ersten Symbol der Suchbegriff-Kette kongruent war (Abb. 5).

Ist die Suche im Sinne der Definition (6) erfolgreich, so wird die Position des Pointers registriert und je nach Zweck des Programms die entsprechende Textzeile ausgedruckt o.ä. Falls X_i und Y_i

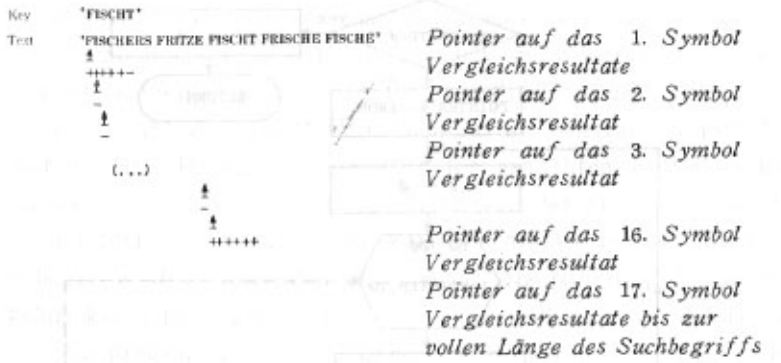


Abb. 5 Suche nach mehrstelligen Ketten mit Hilfe eines Pointers

($1 \leq i \leq n$) an einer Stelle verschieden sind (failure), rückt der Pointer um eine Stelle nach rechts.*

Da ich in der Praxis u. U. nur einen ganz bestimmten Teil einer Textzeile, nennen wir ihn 'Zeilensuchbereich' - durchmustern, gegebenenfalls aber die gesamte Textzeile weiterverarbeiten möchte, führe ich die diesen Bereich eingrenzenden Variablen FROM und TO ein. Eine Funktion namens INDEXI, die mir zu den Argumenten TEXT, LINE, FROM, TO, KEY und KEYLENGTH die erste Position in der Kette TEXT liefert, von der an der Vergleich mit dem KEY der Länge KEYLENGTH ($0 < KEYLENGTH \leq 80$) erfolgreich war (match), bzw. bei erfolgloser Suche den Wert Null ausgibt, wäre etwa wie in Abb. 6 zu konstruieren.

Ähnliche Funktionen sind in einigen Programmiersprachen(FORTRAN 77 ; PL/1) bereits vorgesehen, von Fall zu Fall erweist sich die oben skizzierte Funktion wegen der freien Wahl des Zeilensuchbereichs als flexibler.

*Denn: würde man bis zu dem Punkt vorrücken, wo der Vergleich negativ ausfiel, liefe man Gefahr, sich 'überlappende' Stellen zu überspringen. Bei einem hypothetischen Suchbegriff 'BDBA' und einer Textkette 'ABDBDBADBA' existiert bis zur Position 4 eine identische Teilkette, bei Position 5 scheitert die Suche zwar, aber von Position 4 bis 8 ist eine identische Teilkette vorhanden.

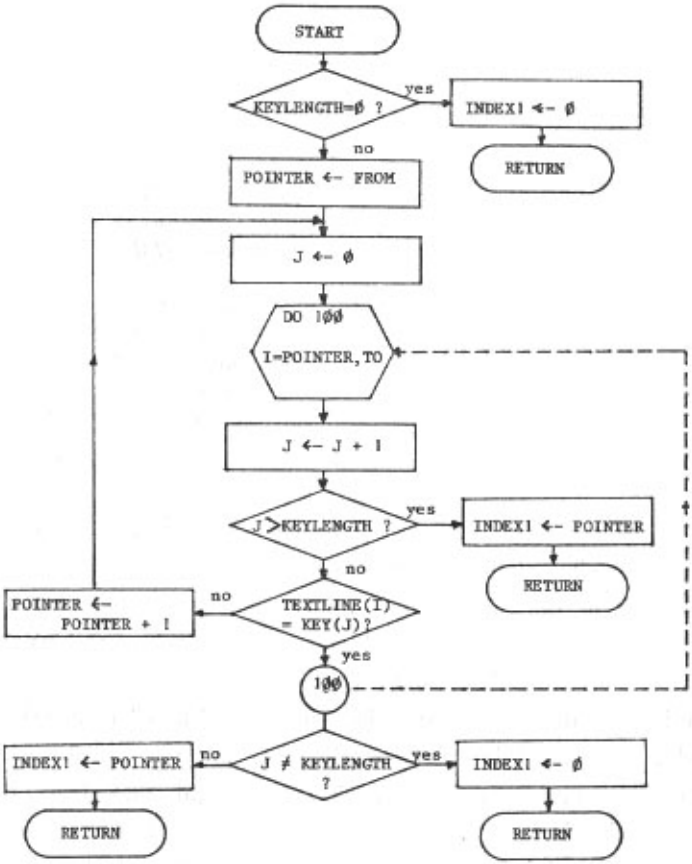


Abb. 6 Struktur der INTEGER FUNCTION INDEX1 (TEXTLINE, FROM, TO, KEY, KEYLENGTH) zur Ermittlung ein- oder mehrstelliger vorgegebener Teilketten. Die Namen der Variablen mit über 6 Buchstaben müssen in vielen Programmsprachen dann infolge von Schreibkonventionen auf diese Zahl gekürzt werden.

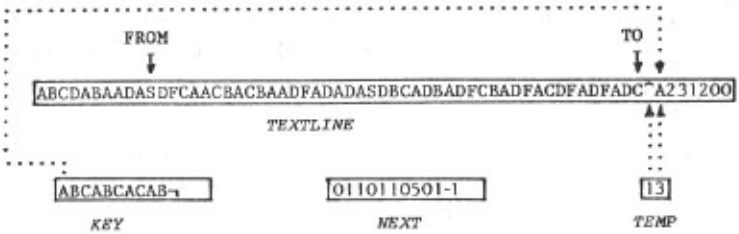


Abb. 7 Die Einfügung von ungebräuchlichen Symbolen bzw. Werten zur Reduktion häufiger routinemäßiger Schritte im Rechenprogramm

Ärgerlich stimmt jedoch das langsame Vorrücken des Pointers, so daß viele Symbole mehrmals verarbeitet werden, wenn z. B. eine Symbolfolge zunächst mit der des gesuchten Keys übereinstimmt, dann aber an einer bestimmten Stelle abweicht, der Pointer indes aus den genannten Gründen nur um eine Stelle verschoben werden darf. Da wir die Struktur der gesuchten Kette kennen, wissen wir auch, wie die Textkette von der Pointerposition bis zu der Stelle aussieht, wo der Vergleich negativ ausfiel. Wenn es gelingt, in dem folgenden allgemeinen Algorithmus, den Schritt "shift key appropiately" durch einen Formalismus zu definieren, ihn also programmierbar zu machen, wird die Suche in vielen Fällen wesentlich beschleunigt :

```
begin
place key at left ;
while key not fully matched
  and text not exhausted do
  begin
    while key symbol differs from
    current text symbol
    do shift key appropiately ;
    advance to next symbol of text
  end
end ;
```

Knuth et al. (1977) schlugen dann vor, den Key mit sich selbst zu vergleichen. Verschiebt man diese Kette X der Länge m gegen sich selbst, so kann man ein Table NEXT(j) aufstellen, das für jeden Wert von $X = x_1, x_2, \ldots, x_m$ den Betrag angibt, um den man bei der Textsuche den Key gegen die Textkette verschieben darf, ohne daß eine der erwähnten 'Überlappungen' übersehen würde. Die Variablen 't' und 'j' erfüllen lediglich Hilfsfunktionen und werden später nicht mehr benötigt :

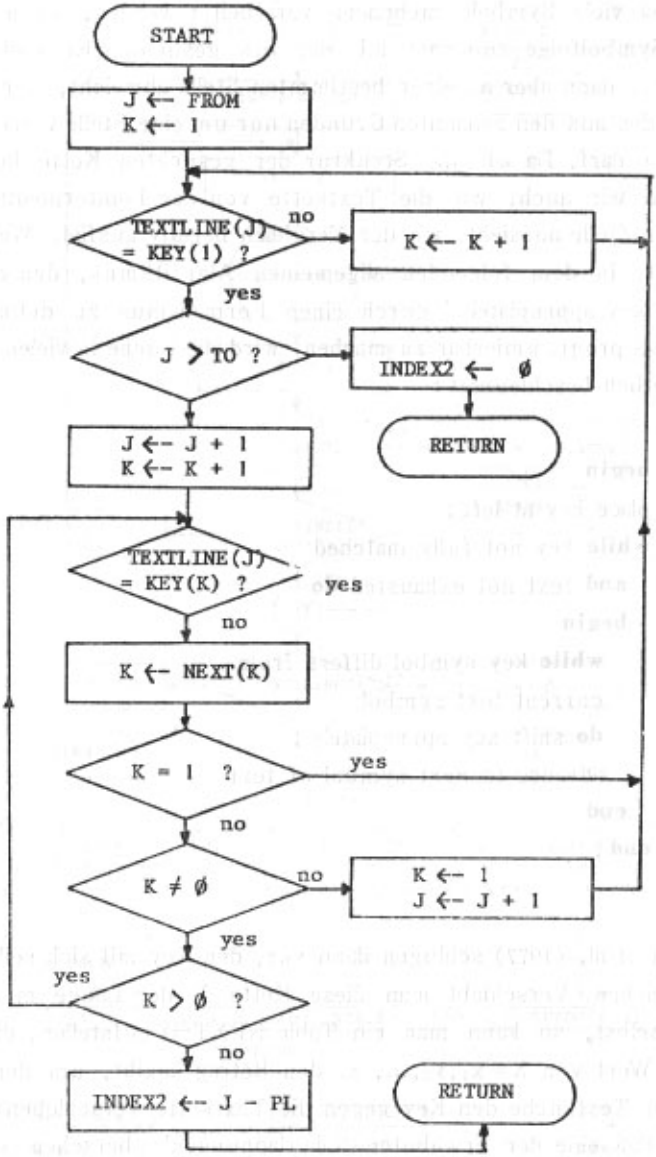


Abb. 8 Allgemeine Struktur der Integer Function INDEX2

```
begin
j←1, t←φ, NEXT(1)←φ ;
while j<KEYLENGTH do
  begin
    while t>φ  and KEY(j)≠KEY(t)
      do t←NEXT(t) ;
    t←t+1, j←j+1 ;
    if KEY(j)=KEY(t)
    then NEXT(j)←NEXT(t)
    else NEXT(j)←t ;
  end
end ;
```

Bei kurzen Texten oder ungünstig strukturierten Suchbegriffen zahlt sich der Aufbau eines solchen Tables nicht allzu sehr aus, in langen Texten kann die auf dieser Basis erstellte Funktion INDEX2 (s. Abb. 8) allerdings sehr effektiv sein.

Ich habe dabei wieder den durch die Variablen FROM und TO gekennzeichneten 'Zeilensuchbereich' eingeführt. Um die Zahl der Tests, ob J>KEYLENGTH oder I>TO zu verringern, setze ich KEY(KEYLENGTH+1)='¬', NEXT(KEYLENGTH+1) = −1 und TEXTLINE(TO+1) = '∼', d. h. auf Werte, die während der Verarbeitung natürlicher Texte nie anzutreffen sein dürften. Setze ich dann noch TEXTLINE(TO+2) = KEY(1) , so kann ich die erwähnten Routinekontrollen, ob denn nun das Ende des Suchbereichs oder des Suchbegriffes erreicht wurde, in weniger oft benutzten Programmteilen unterbringen.
Für den Fall, daß ich nach der Identifikation einer gesuchten Teilkette die gesamte TEXTLINE, die ja nun in den Positionen TO+1 und TO+2 verändert wurde, weiterverarbeiten will, kann ich die ursprünglichen Symbole vor der Manipulation in zwei Zellen TEMP(1) und TEMP(2) vorübergehend notieren und dann später wieder in die TEXTLINE zurückkopieren (s. Abb. 7).

2.3 gleichzeitiges Suchen nach ein- und/oder mehrstelligen Ketten

Bei einer Liste von sagen wir 10 Suchbegriffen jedoch und einem Text von 1000 Seiten, d.h. schätzungsweise 500 000 Zeichen, müßte man nun diesen Text portionsweise in die zentrale Recheneinheit einlesen und zehn Mal anhand der beschriebenen Indexfunktionen durchgehen, also im denkbar ungünstigsten Falle 5 Millionen Zeichen verarbeiten. Was man bei kurzen Texten noch hinnehmen mag, ist bei großen Datenmengen kaum vertretbar.

Zweifellos wäre es günstig, die einzelnen Zeichen der Suchbegriffliste so zu arrangieren, daß wir für jedes gerade behandelte Textzeichen schnell feststellen können, ob dieses vorhanden ist und ob die davorstehenden Zeichen den davorstehenden Suchbegriffzeichen entsprechen. Der Aufbau einer solchen Struktur in Form eines Baumes (tree) bietet keine Schwierigkeiten, indes, was mache ich, wenn für einige Zeichen eine solche Äquivalenz festgestellt wird, der Vergleich dann aber scheitert, jedoch sehr ähnliche andere Suchbegriffe vorhanden sind? Wie springt man dann zur 'richtigen' Stelle des Such-Baumes?

Bei solchen Fragestellungen hat sich das abstrakte Modell des 'Automaten' als sehr nützlich erwiesen *, der für unseren Fall modifiziert bestünde aus

(a) einer Menge S von Kontrollzuständen ('states') : $S = \{s_1, \ldots s_n\}$,

(b) einer Menge X von Eingabesymbolen ('input symbols') über einem Alphabet I : $X = \{x_1, \ldots x_m\}$ und

(c) einer Übergangsfunktion ('state transition function'), die angibt, in welchen Zustand der Automat übergeht, wenn er in einem bestimmten Zustand ist und ein bestimmtes Eingabesymbol verarbeitet.

Der Kontrollzustand s_1 ist der Ausgangszustand ('initial state'), die Zielzustände ('accepting states') F bilden eine Teilmenge von

* Zur Automaten-Theorie siehe A.V. Aho, J.E. Hopcroft, J.D. Ullman: The Design and Analysis of Computer Algorithms. Reading (Mass.) 1974, Kapitel 9 und 10.

S:(F⊆S).

So könnte man beispielsweise die Suchprozesse in Abschnitt 2.2 und 2.3 in der Form eines Automaten beschreiben, der mit 3 Zuständen arbeitet. Die Eingabesymbole wären die der Textkette, die Übergangsfunktion entscheidet aufgrund des Vergleiches mit dem Suchbegriff, in welchen Zustand der Automat übergehen soll. Der Ausgangszustand wäre ϕ, scheitert in diesem Zustand ein Vergleich zwischen Text- und Suchbegriffssymbol, so bleibt der Automat in ϕ. Andernfalls gelangen wir in den Kontrollzustand 1. Bei einem mehrstelligen Key würde der Automat nun solange in diesem Zustand bleiben, bis das Ende der Kette erreicht ist, und schließlich in den Zustand 2 überleiten, der den Erfolg der Suche anzeigt.

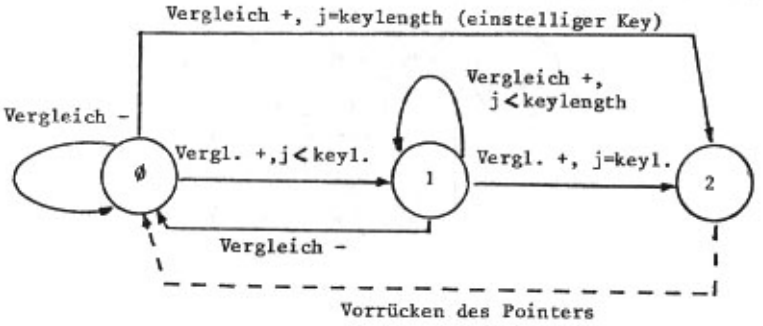
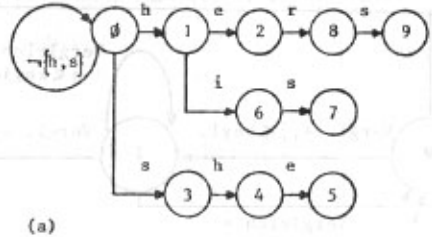


Abb. 9 Kontrollzustände eines Automaten für die Suche nach einem ein- oder mehrstelligen Key

In diesem Zustand verschiebt man nun den Pointer, der den Beginn des Vergleichs in der Textkette indiziert hatte, um den jeweils zulässigen Wert und geht zurück in den Ausgangszustand.

Ein Rechenprogamm, das man unter diesem Denkansatz schriebe, sähe für 2.2 und 2.3 genauso aus wie 'konventionelle' Programme, nur daß man eine zusätzliche Variable STATE eingeführt hätte, deren Werte man ständig überprüft und zur Entscheidungsgrundlage für den weiteren Ablauf des Programms macht.

Bei solch einfachen Sachverhalten wie oben jedoch, darf man getrost auf dieses heuristische Mittel verzichten. Anders sieht es für die komplexe Problematik des gleichzeitigen Suchens nach mehreren Ketten verschiedener Längen aus, die man mit den herkömmlichen Programmiertechniken kaum bewältigen konnte. Hier erwies sich die Konzeption des Automaten als sehr hilfreich. Aho und Corasick (1975) konnten aus diesem Denkansatz heraus eine "pattern matching machine", sprich 'einen Automaten', für endliche Mengen von Suchbegriffen und beliebige Textketten entwerfen. Ihre 'Maschine, mit dem Ausgangszustand ϕ besteht aus einer "Goto-function" g(state,x_i), einer "Failure-function" f(state) und einer "Output-function" output(state). Die "Goto-function" läßt sich – wie weiter oben schon angedeutet – als Baumgraph aus den Symbolen der Suchbegriffliste konstruieren, der dann für jeweils das gerade



(a)

i	1	2	3	4	5	6	7	8	9
f(i)	ø	ø	ø	1	2	ø	3	ø	3

(b)

i	output(i)
2	{he}
5	{she,he}
7	{his}
9	{hers}

(c)

Abb. 10 Die drei Funktionen der "pattern matching machine" für die Suchbegriffe 'he', 'she', 'his', 'hers': goto-function (a), failure-function (b), output-function (c)
(nach Aho, Corasick)

eingegebene Symbol x_i der Textkette X und den derzeitigen Zustand (state) der 'Maschine' angibt, in welchen Zustand diese überzugehen hat (in Abb. 10 durch Pfeile veranschaulicht).
Falls dies nicht unmittelbar möglich ist, wird die "Failure-function" abgerufen, die einen alternativen Übergang anbietet, weil eine ähnliche Symbolfolge auch in einem anderen Key existiert, oder aber den Automaten wieder in Zustand ϕ bringt. Ein Scheitern im Zustand ϕ führt zur Beibehaltung dieses Zustandes, daher der rückläufige Pfeil. Die "Goto-function" der Abb. 10 würde beispielsweise für den Zustand ϕ und alle Eingabesymbole außer 'h' und 's' ($\neg$ {h,s}) sofort zum nächsten Eingabesymbol überleiten. Andererseits geht der Automat nur dann in den Zustand 9 über, wenn er zuvor die Zustände 8, 2 und 1 in dieser Reihenfolge durchlaufen hatte, was wiederum die Eingabe der Symbole 'r', 'e' und 'h' voraussetzt. D.h., bestimmte Zustände indizieren zugleich auch, daß ein bestimmter Suchbegriff gefunden wurde. Die entsprechende Information darüber vermittelt mir die sogenannte "Output-function".

Als allgemeiner Algorithmus zusammengefaßt, hat dieser Automat die folgende Form :

```
begin
  state←φ ;
  for i←1 until n be
    begin
      while g(state,x_i)=fail do state←f(state) ;
      state←g(state,x_i) ;
      if output(state)≠empty then
      print i, print output(state) ;
    end
end
```

(nach Aho,Corasick ; geringfügig geändert)

Vier Probleme sind hierbei zu lösen : die Regelung der Eingabe mehrerer Suchbegriffe sowie die Konstruktion der Goto-, Failure- und Output-function.

2.31. Ein 'sicheres' Verfahren zur Eingabe von Suchbegriffen

Was mache ich, wenn ein Key mit einem Blank endet? Leerzeichen übersieht man nur allzu leicht, so daß es ratsam scheint, Anfang und Ende des Suchbegriffes bei der Eingabe mit einem, in natürlichen Texten normalerweise ungebräuchlichen Zeichen zu markieren. Da man sich auch am Zeilenanfang leicht vertut, sollte man jeden Suchbegriff zunächst in einen Zwischenspeicher BUF einlesen, wobei es egal ist, wo der Key beginnt (Abb.11).

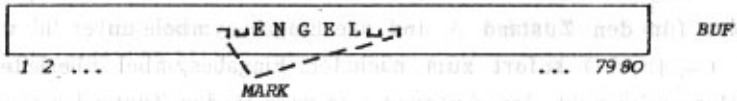


Abb. 11 Zwischenspeicher BUF mit frei eingegebenem Suchbegriff

Anschließend suche ich das erste, linke Markierungszeichen (MARK) und kopiere die folgenden Zeichen in den eigentlichen Array KEY um, bis die zweite, rechte Markierung angezeigt wird. Wird die maximal zulässige Länge MAXLEN überschritten, so bricht das Programm nach Ausdruck einer entsprechenden Mitteilung ab. Andernfalls trägt man die festgestellte Länge des Suchbegriffs in das Register KEYLEN unter der Position KEYNO ein und beginnt, sofern kein Signal kommt, daß alle Keys eingegeben sind, die Prozedur wieder von vorne. Am Schluß entspricht der Wert des Registers KEYNO genau der Zahl der gesuchten Ketten. Das in Abb. 12 veranschaulichte Verfahren funktioniert natürlich auch für nur einen einzigen Suchbegriff.

2.32 Die Gestaltung der drei Funktionen

Nach vielen Versuchen habe ich die Zahl der Eingabesuchbegriffe auf 20 und deren maximale Länge auf 30 festgelegt. Längere Ketten

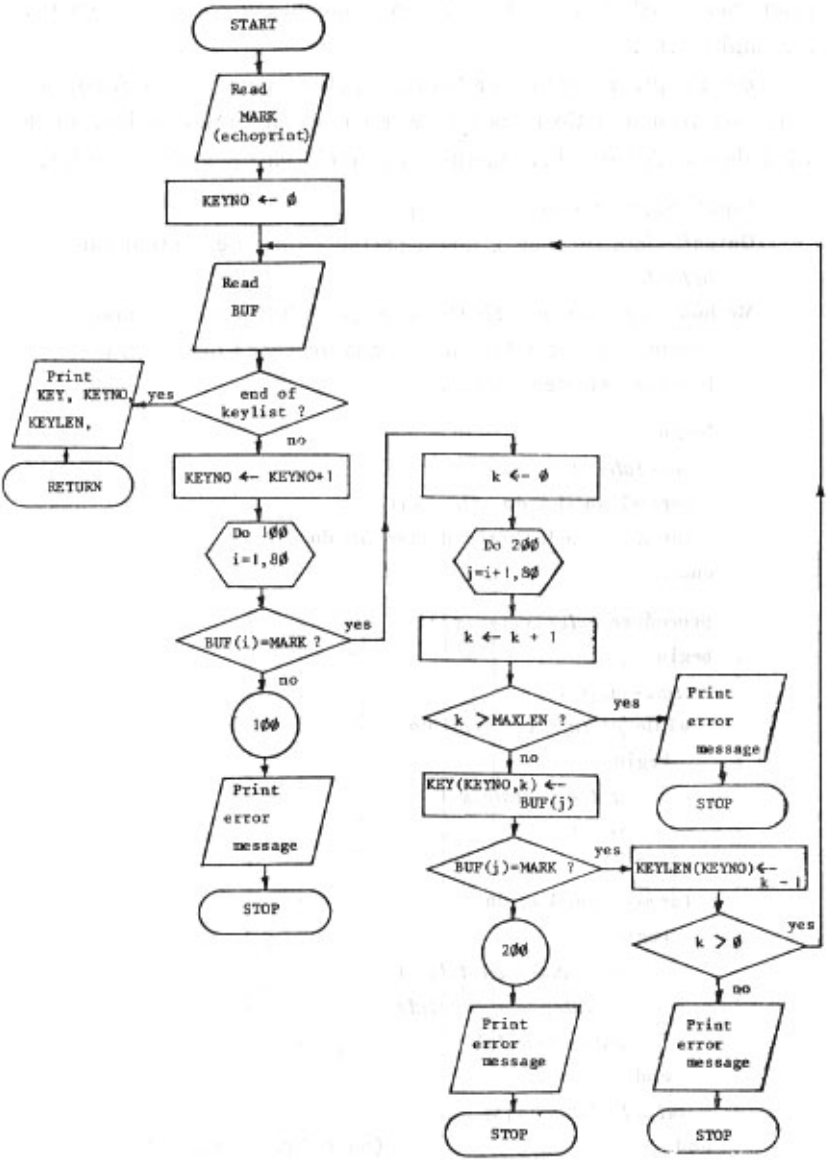


Abb. 12 Fließbild eines Unterprogramms zur flexiblen Eingabe von Suchbegriffen und der Bestimmung ihrer Längen

fallen sehr selten an und verlangen ohnehin andere Verfahren (s. Abschnitt 2.5).

Das Programmieren der 'Goto-function' gemäß dem Algorithmus von Aho und Corasick ist zwar ein wenig umständlich, doch wird dieser Teil des Programms ja nur ein einziges Mal benötigt.

Input. Set of keywords $K=\{X_1, X_2, \ldots\ldots, X_k\}$.

Output. Goto function g and a partially computed output function *output*.

Method. We assume *output*(s) is empty when state s is first created, and $g(s,x)=fail$ if x is undefined or if $g(s,x)$ has not yet been defined.

```
begin
  newstate←0
  for i→1 until k do enter(X1)
  for all x such that g(0,x)=fail do g(0,x)←0
end

procedure enter(x1x2...xm):
begin
  state←0, j←1
  while g(state, xj)≠fail do
     begin
        state←g(state, xj)
        j←j+1
     end
  for p←j until m do
     begin
        newstate←newstate+1
        g(state, xp)←newstate
        state←newstate
     end
  output(state←{x1x2...xm}
end
```

(nach Aho, Corasick)

Die fertige Programm-Funktion G(STATE,SYMBOL) ist kurz und einfach. Sie besteht im wesentlichen aus einer Tabelle mit Zuständen (ST), den diesen zugeordneten Suchbegriffssymbolen (SY) und einem Link-Register SL, das anzeigt, wo ein anderes Symbol zusammen mit dem derzeitigen Zustand eingetragen ist bzw. daß alle Vergleichsversuche scheiterten. Um sofort an die richtige Stelle der Tabelle zu gelangen, sieht man für den Kontrollzustand STATE im Pointer-Register PTR (STATE) nach. Falls dort allerdings eine Null steht, weiß ich, daß überhaupt kein Suchbegriffssymbol zu diesem Zustand existiert. Den Ausgangszustand, bei Aho, Corasick als ϕ gesetzt, nehme ich lieber als 1, da man mit diesem Wert direkt indizieren kann. Die Werte der Kontrollzustände in Abb. 9 werden dadurch um den Betrag 1 erhöht.

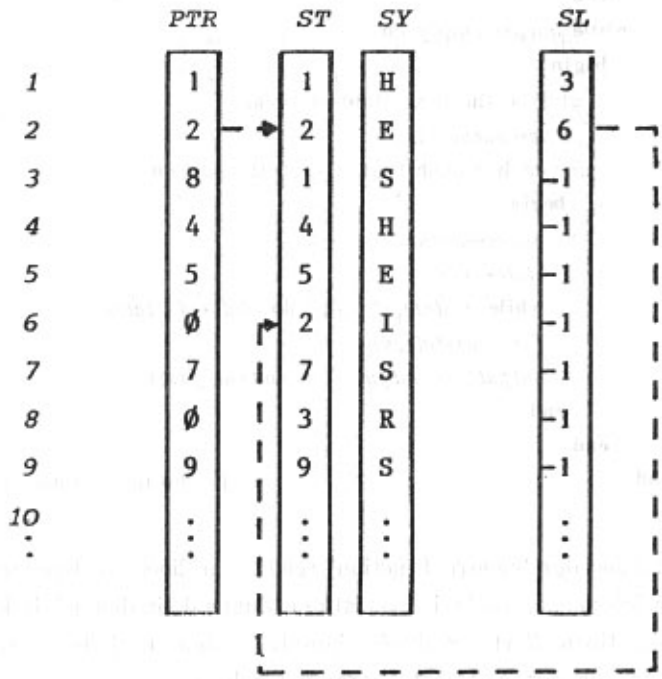


Abb. 13 Anordnung der Register in der Funktion G(STATE, SYMBOL)

Auch die 'Failure-' und 'Output-function' sind etwas sperrig während der Konstruktion, danach aber sehr einfach in der Handhabung :

```
Input. Goto function g and output function output from the above
    Algorithm.
Output. Failure function f and output function output.
Method.
begin
    queue←empty
    for each x such that g(0, x)=s≠0 do
      begin
        queue←queue∪{s}
        (s)←0
      end
    while queue≠empty do
      begin
        let r be the next state in queue
        queue←queue−{r}
        for each x such that g(r, x)=s≠fail do
          begin
            queue←queue∪{s}
            state←f(r)
            while g(state, x)=fail do state←f(state)
            f(s)←g(state, x)
            output(s)←output(s) ∪ output(f(s))
          end
      end
end
```

(nach Aho und Corasick)

Im Falle der 'Failure-function' reicht ein lineares Register F, in dem ich an der Stelle i – das ist der Zustand, in dem ich in keinen direkten Goto-Wert ermitteln konnte, – den nächsten Zustand finde, in den der Automat überzugehen hat.

In der 'Output-function' dient in gleicher Weise ein linearer

Array OUT dazu, für einen eingegebenen Zustand STATE an der Stelle OUT(STATE) anzuzeigen, ob ein Suchbegriff gefunden wurde oder nicht. Negative Meldungen werden direkt durch den Wert Null vermittelt, andernfalls ist der Eintrag als Pointer zu einer Hilfstabelle zu verstehen, in der die Nummer des identifizierten Suchbegriffes steht. Da zuweilen mehrere Suchbegriffe mit einem bestimmten Zustand assoziiert sind, setze ich noch ein Link-Register OL ein:

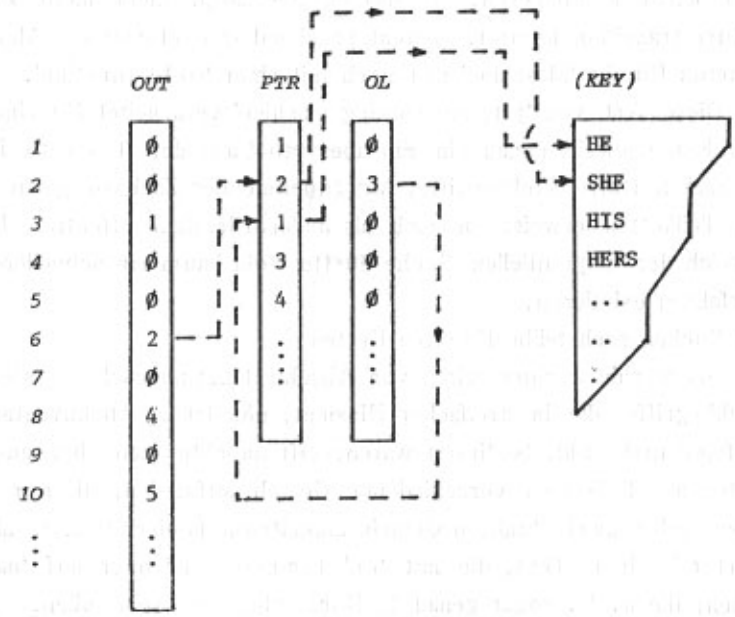


Abb. 14 Die Hauptregister der Funktion OUTPUT(STATE) und ihre Beziehung zum Register KEY

Zweifellos gibt es bei dieser Art Design noch Spielraum, um Speichervolumen einzusparen bzw. die Geschwindigkeit zu steigern. Zum einen könnte man das doppelte Speichern der Suchbegriffe im Register KEY und SY vermeiden, indem man jeden Key direkt aus dem Zwischenspeicher BUF in den Goto-Baum einfügt. Beim Ausdruck während der Suche läßt sich der identifizierte Suchbegriff ohnehin auf vielerlei Weise signalisieren. Zum anderen ist die 'Failure-function' vom Standpunkt der Automatentheorie gesehen, ja Teil der Übergangsfunktion. Nachdem die 'Goto-function' und die 'Failure-funcion' fertiggestellt sind, kann man deren Register sehr leicht kombinieren, so daß es überhaupt nicht mehr zur 'failure transition' kommt. Dementsprechend arbeitet dann der Algorithmus für die 'Maschine' nur noch mit einer 'Goto-function'.

Diese Art von 'pattern mathing machine' verarbeitet für einen Durchgang jeweils genau ein Eingabesymbol aus der Textkette im Hinblick auf *alle* Suchbegriffe. Mit zunehmender Zahl von gesuchten Teilketten erweist sie sich als außerordentlich effektiv. Im Bereich der sequentiellen Suche dürfte wohl kaum ein schnelleres Verfahren existieren.

2.4 Suchen nach teilbestimmten Ketten

Bis hierhin suchten wir - von Abschnitt 2.1 abgesehen - stets Suchbegriffe, die in dreifacher Hinsicht, nämlich Symbolinventar, -abfolge und -zahl, bestimmt waren. Oft möchte man aber auch Listen mit Teilketten verschiedener Gestalt aufstellen, die nur in einem oder zwei Punkten Gemeinsamkeiten aufweisen: z.B. alle Wörter in einem Text, die mit 'un' beginnen und/oder auf 'bar' enden, die u. U. sogar genau 11 Buchstaben lang sein sollen.

2.41

Zunächst Fall (a), bei dem nur das Präfix (P) und das Suffix (S) der zu ermittelnden Ketten vorgegeben ist:

(a) [P ? S]

P und S verstehen wir im Sinne der Definition (5), was auch aus den folgenden Anwendungsbeispielen hervorgehen sollte:

P='␣UEBER', S='␣': (Wörter, die mit 'über' beginnen)

P='␣', S='TION␣': (Wörter, die auf 'tion' enden)
P='␣', S='␣': (alle Wörter des Textes)
Bei P='␣' und S='␣' schließen wir durch Konvention leere Ketten als Ausgabe für '?' aus, da dies unerwünschte Effekte haben könnte.

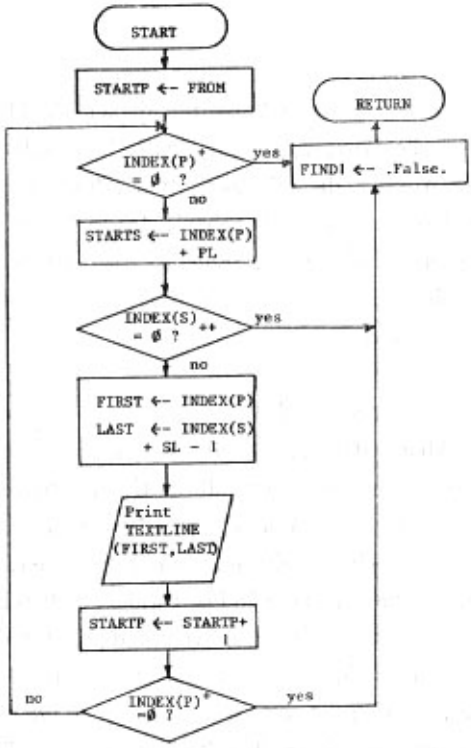


Abb. 15 Allgemeine Struktur der logischen Funktion FIND1. STARTP: Ausgangspunkt der Suche nach P; PL: Länge von P; STARTS: Ausgangspunkt der Suche nach S; PS: Länge von S; FIRST: Beginn der identifizierten Teilkette; LAST: Ende der identifizierten Teilkette. Die Argumente der Funktion INDEX sind aus Platzgründen verkürzt, sie müßten normalerweise neben TEXTLINE, P, PL, STARTP bzw. S, SL, STARTS auch TO umfassen. Falls man sich für INDEX2 entschieden hätte, wäre darüber hinaus auch der Zugang zum Table NEXT zu sichern.

Wenn wir eine der beiden bereits entwickelten Index-Funktionen nutzen, ist die Strategie denkbar einfach. Wir stellen zunächst fest, ob überhaupt eine Teilkette P existiert, schauen dann nach S und drucken gegebenenfalls die gesamte, gefundene Teilkette von Anfang P bis Ende S samt den gewünschten Nebendaten aus. Das Fließbild in Abb.15 zeigt die Struktur einer solchen (logischen) Funktion FIND1.

2.42

Es sind noch komplexere Varianten dieses Problems denkbar: gesucht werden alle Wörter, die auf 'bar', 'lich' oder 'abel' enden, oder z. B. alle Wörter, die mit 'zer', 'ver', 'ent' beginnen bzw. im Extremfall alle Wörter, die mit 'über', 'unter' beginnen und auf 'ung', 'heit' oder 'keit' enden. Solche Alternativen sollen nachfolgend durch einen Schrägstrich symbolisiert werden:

(b) $[P ? S_1/.../S_n]$

(c) $[P_1/.../P_m ? S]$

(d) $[P_1/.../P_m ? S_1/.../S_n]$

Da wir in Abschnitt 2.3 bereits einen 'endlichen Automaten' für das parallele Suchen nach alternativen Suchbegriffen konstruiert haben, bietet der Aufbau der Programme für die Fälle (b), (c) und (d) keine weiteren Schwierigkeiten. Wir ersetzen einfach in Abb. 15 die jeweilige Index-Funktion durch den ebenfalls als Funktion etwas modifizierten Automaten, der dann nicht nur die Stelle mitteilen muß, wo eins der gesuchten P_1. bzw. S_1. erkannt wurde, sondern auch, um welches Affix es sich handelt. Ja, man kann sogar alle vier Fälle mit ein und demselben Programm FIND2 angehen, das mit zwei Automaten arbeitet. Denn diese Automaten funktionieren auch für nur einen einzigen Suchbegriff.

2.43

Nun eine neue Fallgruppe: neben dem Prä- und dem Suffix ist ein Teil des 'Infixes' I bekannt, z.B., wenn man Wörter sucht, in denen die Buchstabenfolge 'lauf' enthalten ist. Das wird wie in 'Schlaufe' zweifellos nicht immer der Stamm 'lauf' sein, den wir

eigentlich im Blick hatten, doch das nächtragliche Aussortieren dieser Fehlschüsse reicht für unsere Zwecke völlig. Die allgemeine Suchformel lautet in der oben benutzten Schreibweise

(e) (P ? I ? S).

Für das Beispiel wären P=␣,S=␣und I='lauf'. Lassen wir auch Alternativen zu, so ergeben sich nach den Regeln der Permutation noch weitere fünf Möglichkeiten:

(f) (P_1/.../P_m ? I ? S),

(g) (P ? I_1/.../I_n ? S),

(h) (P ? I ? S_1/.../S_o),

(i) (P_1/.../P_m ? I ? S_1/.../S_o,)

(j) (P_1/.../P_m ? I_1/.../I_n ? S_1/.../S_o).

Natürlich könnte man für jeden einzelnen Fall ein Programm (FIND 5 bis FIND10) entwerfen, ich halte jedoch wie beim vorherigen Problemkreis zwei Programme, nämlich FIND5 für (e) und das auch auf (f) bis (i) anwendbar Programm FIND10, das mit drei Automaten zu konstruieren ist, für optimal.

2.44

Gehen wir noch einen Schritt weiter und führen wir Längenangaben (*l) für den inhaltlich unbestimmten Teil ein, um beispielsweise alle Wörter mit vier Buchstaben (P='␣', S='␣',*l=4) aufzulisten oder solche mit 8 Buchstaben und 'ig' am Ende (P='␣',S='ig',*l=6) – allgemeiner ausgedrückt:

(e) (P ? *lS)

Der Lösungsweg dürfte aus dem bisher Gesagten klar sein. Das Programm ist in der Tendenz schneller als das für den Fall (a), da wir, nachdem ein in der Position i endendes P festgestellt wurde, direkt zur Position i+1+ l springen und dort den Vergleich mit S beginnen. Zwar gibt es noch 5 weitere Variationen dieses Typs, (f) bis (j), von denen ich jedoch nur eine exemplarisch herausgreife und kurz erläutere, da die praktische Bedeutung dieser Fälle nicht allzu groß ist:

(j) (P_1/.../P_m ? *l_1/.../$l_m$$S_1$/.../$S_o$)

P und S werden wie bisher mit Hilfe zweier 'pattern matching machines' verarbeitet. Die Zahlenwerte von l_1 bis l_m speichert man in einem Array, nennen wir es hier LTABLE:

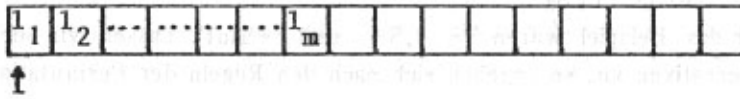


Abb. 16

Ist dann irgendein P_i gefunden worden, geht man anhand eines Pointers diesen Array von 1 bis m durch und prüft an den sich rechnerisch ergebenden Stellen, nämlich letzte Stelle von $P+l_i+1$, auf die S-Alternativen.

2.45

Eine letzte Form scheint mir noch für die Praxis relevant. Ich hätte z. B. gerne eine Liste aller Textteilketten, in denen die Vokale 'a', 'e' und 'i' genau in dieser Reihenfolge enthalten sind, allgemeiner:

(k) $(P ? I_1 ? \ldots I_k \ldots ? I_n ? S)$.

Hier benötigt man zwei Arrays, das eine für die Symbolfolgen I_k und ein weiteres für die jeweils entsprechende Längen:

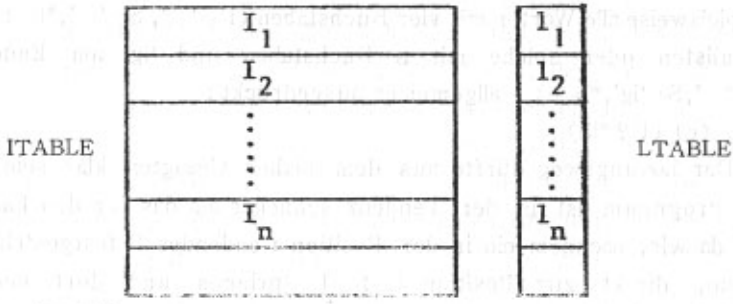


Abb. 17

Das Verfahren selbst ist einfach. Ich durchmustere den Text auf P, falls dieses angetroffen wird, anschließend auf I_1 etc., bis ich schließlich mit dem Nachweis von S die Suche erfolgreich abschließen

kann. Im Gegensatz zum Beispiel (f) scheitert ein Suchzyklus, sobald eines der Glieder nicht nachweisbar ist.

2.5 Phrasen, Sätze und Stellen

Prinzipiell taugen die entwickelten Instrumente auch für längere Ketten – über die grammatische Ebene des Wortes hinaus – wie ein paar Beispiele veranschaulichen mögen :

(PIS)	'␣das dunkle Meer␣'
(P ? S)	'␣ein ? Tisch␣'
(P ? I ? S)	'␣der ? Mann ? nicht nach Hause␣'
(P ? I_1/I_2 ? S)	'␣sie ? keinen/gerne ? Wein'␣.

Ein Blick auf den Textausschnitt in Abb.11 zeigt jedoch, daß bei Suchketten mit zunehmender Länge die Gefahr von Fehlschlägen wächst. Zum einen haben wir unsere Programme auf der Grundlage von Zeileneinheiten konzipiert. Natürlich könnte man auch an formatfreie Speicherungen oder andere Eingabeverfahren denken, was dann wiederum die Behandlung der Nummerierungen wie auch den Einsatz von längenmäßig nicht eingegrenzten Funktionen für Fälle wie (P ? S), (P ? I ? S) u.ä. erschweren würde.

```
EIN SCHRIFTVERS BESAGT : SEIN GEWAND IST WEI$ WIE SCHNEE UND SEIN   CH14A001
HAUPTHAAR REIN WIE WOLLE . FERNER STEHT GESCHRIEBEN : SEINE LOCKEN  CH14A002
SIND WIE DATTELRISPEN UND RABENSCHWARZ . DAS IST KEIN WIDERSPRUCH . CH14A003
HIER IST GEMEINT : IN DER SITZUNG , UND DA IST GEMEINT : IM KAMPF . CH14A004
DER MEISTER SAGTE NAEMLICH : ES GIBT NICHTS SCHOENERES IM KAMPF ALS CH14A005
EINEN JUENGLING , UND ES GIBT NICHTS SCHOENERES IN DER SITZUNG ALS  CH14A006
EINEN GREIS .                                                       CH14A007
```

Abb. 18 Beispiel eines unbearbeiteten Speichertextes mit Positionsnummerierung

Doch selbst davon einmal ganz abgesehen–ein Blank zuviel irgendwo oder ein kleiner unentdeckter Tippfehler bei der Speicherung der Textdaten lassen uns bei rigorosem Vorgehen leicht scheitern. So lohnt sich die Mühe einer 'maschinenfreundlicheren', sprich

'menschenunfreundlicheren', Textform kaum. Überdies ist die Wahrscheinlichkeit, einen bestimmten längeren Satz in genau der angegebenen Form zu finden, ziemlich gering. Tatsächliche Suchprobleme sehen meist anders aus. Ich erinnere mich beispielsweise nur undeutlich an eine Textstelle, wo von 'Demonstrationen' oder 'Demonstranten' wegen eines 'Atomkraftwerks' oder ‚Kernkraftwerks' und der 'Polizei', 'Polizisten' bzw. 'Bullen' die Rede war. Auf Flexionsendungen kann ich mich nicht festlegen lassen, ob das alles in einem Satz oder einer Zeile zu finden war, weiß ich nicht. In einem solchen Fall ist der Einsatz der 'pattern matching machine' aus Abschnitt 2.3 effektiv. Ich reduziere auf 'DEMO','KRAFTWERK', 'POLIZ' und 'BULLE' und sehe dann im Output einmal nach, wo all diese Dinge relativ dicht beeinanderliegen. Dieses Vorgehen anhand möglichst spezifischer Unterketten aus der eigentlich gesuchten Kette erweist sich in der Praxis allen anderen Manipulationen überlegen – mit Ausnahme der Fälle selbstverständlich, in denen ich den Output direkt maschinell weiterbearbeiten möchte.

Literatur :

A. V. Aho, M. J Corasick : Efficient String Matching : An Aid to Bibliographic Search. Communications of the ACM, Vol. 18, No.6, June 1975, 333—340.

B. W. Kerningham, P. J. Plauger : Software Tools. Reading (Mass.), 1976.

D.E. Knuth, J.H. Morris, V. R. Pratt : Fast Pattern Matching in Strings. SIAM J.COMPUT., Vol. 6, No. 2, June 1977, 323—351

FACOM OS IV, JEF 解説書, 富士通, 東京 1980 (JEF-Handbuch, Fujitsu, Tokyo 1980)