Klaus Turowski

Suche schnell gemacht

Schnelle Mustersuche mit determinierten endlichen Automaten

Die Suche von Daten in einer riesigen Datei ist meistens eine äußerst zeitraubende Angelegenheit. Daß Suchen auch schnell gehen kann, haben Knuth, Morris und Pratt sowie Boyer und Moore eindrucksvoll bewiesen, mit sogenannten determinierten endlichen Automaten.

Jeder von uns hat tagtäglich mit Automaten zu tun. Wir sind umgeben von vergleichsweise einfachen Automaten wie Waschmaschinen, Schreibmaschinen, Kaffee- oder Zigarettenautomaten aber auch von ungleich komplizierteren Automaten wie Taschenrechnern und Computern. Den Aufbau und die Funktionsweise einfacher Automaten kann man problemlos verbal beschreiben. Zur Beschreibung komplizierter Automaten eignen sich abstrakte Modelle jedoch besser. Ein dafür geeignetes Modell ist das des determinierten endlichen Automaten, welches uns später bei der Suche weiterhelfen wird.
Alle Automaten reagieren auf Signale aus ihrer Umwelt. Kennzeichnend für den jeweiligen Automaten ist die Menge von Signalen, die er akzeptiert. Diese Menge heißt Eingabealphabet. Betrachtet man beispielsweise einen Eierkocher, so kann man ihm folgendes Eingabealphabet zuordnen: Gerät öffnen (Gö), Eier einlegen (Ee), Gerät schließen (Gs) und Gerät einschalten (Ge). Die Eingabefolgen Gö, Ee, Gs, Ge und Gö, Ee, Ge, Gs führen zu einem korrekten Ergebnis: gekochten Eiern. Probleme dürften dagegen die Eingabefolgen Ee, Gö, Gs, Ge, oder Gö, Gs, Ge, Ee verursachen. Es ist daher sinnvoll, dem Automaten einen inneren Zustand zuzuordnen. Bei einer korrekten, sinnvollen Eingabe soll der Automat in einen Erfolgs- oder Endzustand überführt werden. Dabei können wie im obigen Beispiel Zwischenzustände durchlaufen werden. Der Automat reagiert auf ein Signal durch Übergang in einen anderen Zustand. Dieser Übergang ist abhängig vom aktuellen Zustand und vom anliegenden Signal. Der neue Zustand, in den der Automat übergeht, muß sich nicht vom vorangegangenen unterscheiden.

Getränke aus dem Automaten

Um die oben definierten Begriffe zu illustrieren betrachten wir nun ein ausführliches Beispiel – einen Getränkeautomaten. Der Automat soll ein Getränk in Dosen verkaufen und dafür eine Mark kassieren. Bezahlen kann man entweder durch Einwurf von einem Ein-Mark-Stück oder durch Einwurf von zwei 50-Pfennig-Stücken. Darüber hinaus bietet der Automat noch zwei Manipulationsmöglichkeiten. Er verfügt über eine Getränkeauswurftaste und eine Geldrückgabetaste. Als Eingabealphabet definieren wir:

1 :	Einwurf eines Ein-Mark-Stückes
50 :	Einwurf eines 50-Pfennig-Stückes
R :	Betätigung der Geldrückgabetaste
A :	Betätigung der Getränkeausgabetaste

Der Automat kann die folgenden Zustände annehmen:

(0) Startzustand des Getränkeautomaten, Geldeinwurf wird erwartet
(1) Ein Ein-Mark-Stück wurde eingeworfen, der Automat wartet auf das Betätigen der Getränkeauswurftaste oder der Geldrückgabetaste
(2) Ein 50-Pfennig-Stück wurde eingeworfen, der Automat wartet auf den erneuten Einwurf eines 50-Pfennig-Stückes oder auf das Betätigen der Geldrückgabetaste.

Der Getränkeautomat soll nach Betätigung der Getränkeauswurftaste im Zustand (1) in den Zustand (0) übergehen. Das Betätigen der Getränkeauswurftaste soll in den Zuständen (0) und (2) keine Zustandsänderung bewirken. Auf die Geldrückgabetaste wird nur in Zustand (1) und (2) reagiert. In *Bild 1* erkennt man den Startzustand an einem Pfeil, der diesen als Endknoten hat, dessen Startknoten aber nicht sichtbar ist. Der Endzustand ist von zwei Kreisen eingeschlossen gezeichnet. Der zu dem Automaten gehörende Graph zeigt, daß ein Zustand zugleich Start- und Endzustand sein kann. Ebenso läßt sich nachvollziehen, welche Zustände bei einer bestimmten Folge von Eingabezeichen durchlaufen werden. Bei der Eingabefolge 50, 50, A sind dies die Zustände (2), (1), (0). Da in obigem Beispiel der Startzustand (0) auch der Endzustand ist, führen alle Eingabefolgen zum Erfolg, die nach der letzten Eingabe in den Zustand (0) übergehen. Die Eingabe 50, 50, A ist also ebenso korrekt wie die Eingabe 50, R, 50, R, 1, A. Alle Eingabefolgen, die vom Automaten akzeptiert werden, also in einen Erfolgszustand führen, gehören zur Sprache des Automaten.
Betrachtet man den obigen Beispielautomaten genauer, fällt auf, daß er nicht ganz vollständig ist. So ist zum Beispiel nicht ersichtlich in welchen Zustand der Automat übergeht, wenn im Zustand (2) die Eingabe (1) erfolgt. Um dennoch jedem Zustand und jeder beliebigen Eingabe einen neuen Zustand zuordnen zu können, führt man den Fehlerzustand (F) ein. Der Fehlerzustand kann durch keine Eingabe mehr verlassen werden. Erweitert man den Beispielautomaten entsprechend, kann man ihn wie in *Bild 2* darstellen.
Damit wäre dann auch das letzte Bestandteil eines determinierten endlichen Automaten beschrieben: Die Übergangsfunktion *(Tabelle 1)*. Sie ist das mathematisch formale Äquivalent der obigen Graphen und ordnet jedem Zustand und jeder möglichen Eingabe einen neuen Zustand zu. Aus ihr läßt sich der Graph des Automaten rekonstruieren.

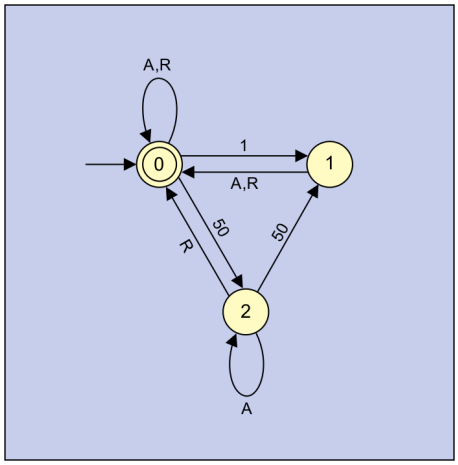

Bild 1. Ein Getränkeautomat abstrakt dargestellt

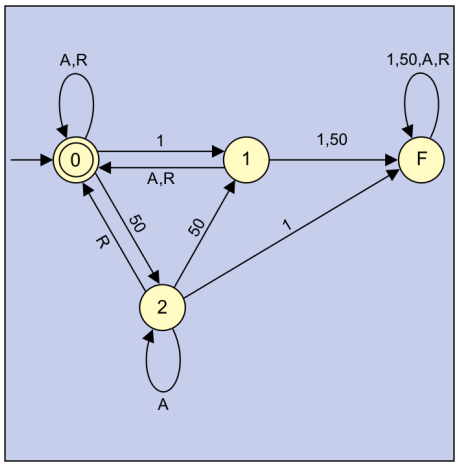

Bild 2. Der Getränkeautomat als vollständiger determinierter endlicher Automat

Tabelle 1. Übergangsfunktion zu Bild 2				
	Eingaben			
Zustand	A	R	1	50
(0)	(0)	(0)	(1)	(2)
(1)	(0)	(0)	(F)	(F)
(2)	(2)	(0)	(F)	(1)
(F)	(F)	(F)	(F)	(F)

Mustersuche in Zeichenketten

Das am häufigsten auftretende Suchproblem ist das, in einem längeren Text jedes Vorkommen eines bestimmten Musters oder Wortes ausfindig zu machen. Dieses Problem findet man sowohl im Bereich der Textverarbeitung als auch im Bereich der verallgemeinerten Mustererkennung, wo es darum geht, in einem beliebigen Eingabestrom bestimmte Muster ausfindig zu machen. Dies kann bei der nachträglichen Bearbeitung digitalisierter Bilder ebenso nötig sein wie bei der Manipulation digitalisierter Tonaufnahmen. Um ein bestimmtes Muster in einer Zeichenkette zu finden wendet man zumeist den folgenden einfachen Algorithmus an. Dabei ist B die Zeichenkette, die das Muster enthält und A die zu durchsuchende Zeichenkette. A[1] und B[1] sind die jeweils ersten Zeichen und A[n] und B[m] die jeweils letzten.

```
FOR k=1 TO n-m+1
{
  FOR l=1 TO m
  {
    IF A[k+l-1] != B[l] THEN BREAK
  }
  IF l == m+1 THEN RETURN gefunden
}
RETURN nicht_gefunden
```

Bei diesem Algorithmus wird also einfach das zu suchende Muster bei Nichtübereinstimmung um ein Zeichen nach links geschoben und der Vergleich neu gestartet. Die benötigte Anzahl von Zeichenvergleichen ist proportional zum Produkt n·m der Längen der Zeichenketten A und B.

Filtern mit dem Automaten

Basierend auf einem determinierten Automaten kann man einen Algorithmus angeben, der das Suchproblem wesentlich flinker löst. Der dafür benötigte Aufwand ist nur noch proportional zur Summe der Längen der Zeichenketten A und B. Dazu konstruiert man zu dem zu suchenden Muster B einen determinierten endlichen Automaten, der A zeichenweise als Eingabe verarbeitet und genau dann in einen Endzustand übergeht, wenn B in A gefunden wurde.
Die Konstruktion des Automaten erfolgt in zwei Abschnitten. Zuerst erzeugt man zu dem Muster B einen unvollständigen Skelettautomaten *(Bild 3)*. B[*i*] sind die Musterzeichen, und E bezeichnet die Menge der zulässigen Eingabezeichen.
Dann vervollständigt man den Skelettautomaten. Der Automat ist genau dann im Zustand (*l*) mit $1 \le l \le m$, wenn die letzten *l* gelesenen Buchstaben der Zeichenkette A mit den ersten *l* Buchstaben des Musters B übereinstimmen. Jetzt unterscheidet man zwei Fälle:

1. Ist das nächste gelesene Zeichen gleich B[*l*+1], dann geht der Automat in den Zustand (*l*+1) über.
2. Ist das nächste gelesene Zeichen ungleich B[*l*+1], dann muß man den neuen Zustand (*i*) mit $i < l$, derart wählen daß B[1] bis B[*i*] das längste Endstück von dem um das neue Eingabezeichen erweiterten B[1] bis B[*k*] ist.

Dazu ein Beispiel. Das zu suchende Muster B ist ababa. Die zu durchsuchende Zeichenkette ist eine beliebige über der Menge a, b. Zuerst konstruiert man den Skelettautomaten *(Bild 4)*. Dann wird der Skelettautomat vervollständigt *(Bild 5)*: Jeder Zustand muß von einem mit a und von einem mit b beschrifteten Pfeil verlassen werden. Zustand (0) ist also vollständig. Bei Zustand (1) muß noch ein Pfeil mit der Beschriftung a hinzugefügt werden. Dazu stellt man folgende Überlegung an: Würde im Zustand (1) die Eingabe a erfolgen, hätte man die Zeichenkette aa gefunden. Der längste Teilstring von aa der mit ababa übereinstimmt ist a. Also führt der Pfeil aus Zustand (1) mit der Beschriftung a wieder in Zustand (1). Eine ähnliche Überlegung kann man bei den

Zuständen (3) und (5) anstellen. Die mit a beschrifteten Pfeile überführen den Automaten hier jeweils in den Zustand (1). Die Zustände (2) und (4) würden bei dem Eingabezeichen b implizieren, daß ein Teilstring mit der Endung abb gefunden wurde. Da es aber keinen Teilstring von B gibt, der mit abb oder b beginnt, muß der Automat in den Zustand (0) übergehen. Wird in Zustand (5) ein b eingegeben, so hat man den Teilstring abab gefunden. Der neue Zustand ist deshalb (4).
Um das Problem der Mustersuche zu lösen, genügt es also, einen geeigneten Automaten für das zu suchende Muster zu konstruieren und ihn auf die zu durchsuchende Zeichenkette anzuwenden.

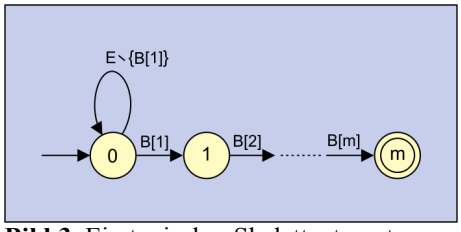

Bild 3. Ein typischer Skelettautomat

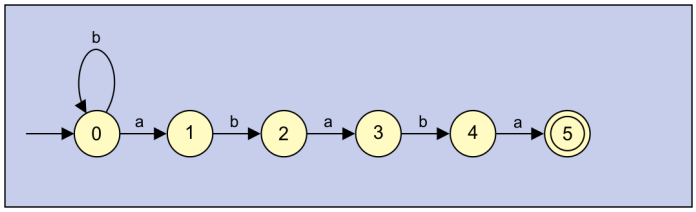

Bild 4. Der Skelettautomat für das Suchmuster ababa

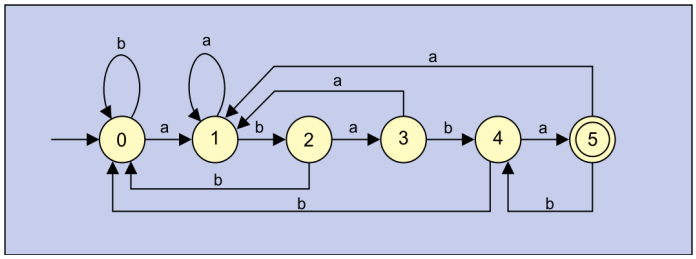

Bild 5. Der vollständige Automat für das Suchmuster ababa

Mustersuche im Simultanbetrieb

Ausgehend von dem Problem, ein Muster in einer Zeichenkette zu suchen, erweitern wir die Aufgabenstellung auf das Problem mehrere Muster gleichzeitig zu suchen und erarbeiten dafür Algorithmen, deren Aufwand, wie oben gefordert, proportional zur Summe aus der Summe der Musterlängen und der Länge der zu durchsuchenden Zeichenkette ist. Das ist der allgemeinste Fall. Die zu suchenden Muster werden mit B(1) bis B(*k*) bezeichnet, die zu durchsuchende Zeichenkette ist wieder A. Die Zeichenketten B(1) bis B(*k*) müssen nicht gleich lang sein. Die Vorgehensweise zur Konstruktion des Suchautomaten ist ähnlich wie bei dem Suchautomaten für ein Schlüsselwort. Auch hier wird der Suchautomat auf die zu durchsuchende Zeichenkette angewandt. Bei Eingabe eines Zeichens ist es aber möglich, daß das mehr als eine Zustandsänderung zur Folge hat. Wird ein Muster gefunden, erzeugt der Suchautomat eine Ausgabe. Damit kann man die Funktionen festlegen aus denen der Suchautomat besteht:

1. Die Goto-Funktion. Diese Funktion entspricht der Zustandsübergangsfunktion des Skelettautomaten im einfachen Fall. Sie ordnet dem Tupel Zustand, Eingabe einen neuen Zustand zu. Ist das nicht möglich wird eine entsprechende Meldung erzeugt.
2. Die Failure-Funktion. Diese Funktion benötigt man immer dann, wenn die Goto-Funktion versagt. Das ist genau dann der Fall, wenn die aktuelle Eingabe zu keinem der Suchmuster paßt. Die Failure-Funktion überführt den Suchautomaten daraufhin in den Zustand, der dem längsten gefundenen Teilmuster entspricht.
3. Die Output-Funktion. Wurden ein oder mehrere Muster gefunden, kann man mit Hilfe der Output-Funktion bestimmen welche.

Wir betrachten dazu ein Beispiel. Als Eingaben sind alle Großbuchstaben ohne Umlaute zugelassen. Die Menge A, B, ... Z entspricht also der Menge E, dem Eingabealphabet. Der Automat soll in einer Zeichenkette alle Vorkommen der Muster AUTAN, AUTOMAT, MAT und TO ausfindig machen. Zuerst wird wieder der Skelettautomat konstruiert *(Bild 6)*. Der dargestellte Skelettautomat ist die grafische Darstellung der Goto-Funktion. Der Zustand (2) mit der Eingabe T wird von der Goto-Funktion mit Goto(2, "T") in den Zustand (3)

überführt. Jede andere Eingabe in Zustand (2) erzeugt eine Failure-Meldung. Der Automat ist wieder so aufgebaut, daß er sich genau dann im Zustand (*k*) befindet, wenn die bis zu diesem Zustand erfolgten Eingaben der Anfang von einem oder mehreren Mustern sind.
Die Failure-Funktion vervollständigt den Automaten. Immer wenn die Goto-Funktion eine Failure-Meldung liefert, kann man mit Hilfe der Failure-Funktion den neuen Zustand ermitteln. Dabei geht man sukzessive vor. Befindet sich der Automat beispielsweise im Zustand (8) und e sei eine beliebige Eingabe. Die Goto-Funktion Goto(8, "e") erzeugt eine Failure-Meldung. Das heißt: e ist ungleich T. Die Failure-Funktion sucht jetzt einen neuen Zustand so, daß der neue Zustand dem längsten Endstück von AUTOMA entspricht. Sie liefert also als ersten Übergang den Zustand (11). Auf diesen neuen Zustand wird daraufhin die aktuelle Eingabe e angewandt. Da aber e ungleich T ist, erzeugt die Goto-Funktion wieder die Failure-Meldung. Deshalb muß ausgehend vom Zustand (11) ein neuer Failure-Übergang gefunden werden. Das ist der Zustand (1). Ist e gleich U, überführt die Goto-Funktion den Automaten in den Zustand (2) und die Failure-Bearbeitung ist beendet. Ist aber e ungleich U, führt die erneute Anwendung der Failure-Funktion zum Zustand(0). Spätestens bei Erreichen des Startzustandes (0) bricht die Failure-Bearbeitung ab. Der Startzustand ist dann der aktuelle Zustand des Automaten *(Bild 7, Tabelle 2)*. Zum Abschluß muß nur noch die Output-Funktion berechnet werden. Sie erzeugt zu jedem Zustand eine Menge der bisher gefundenen Muster. Diese Menge kann auch leer sein. Daran erkennt man, daß man noch kein Muster gefunden hat. In der Tabelle weiter unten sind die zu den Zuständen gehörenden Output-Mengen angegeben. Leere Output-Mengen werden nicht aufgeführt *(Tabelle 3)*.
Sind die Output-, Goto- und Failure-Funktion vorhanden kann man den Suchautomaten folgendermaßen programmieren:

```
Zustand = 0
WHILE ( Eingabezeichen vorhanden )
{
  WHILE ( goto(Zustand, aktuelle Eingabe) == failure )
    Zustand = failure(Zustand)
  Zustand = goto(Zustand, aktuelle Eingabe)
  IF ( output(Zustand) != {})
    Gebe gefundene Muster aus
}
```

Angewandt auf die Eingabe AUTOMATEN erhält man die Ausgabe:

```
A
U
T
O -TO
M
A
T -AUTOMAT -MAT
E
N
```

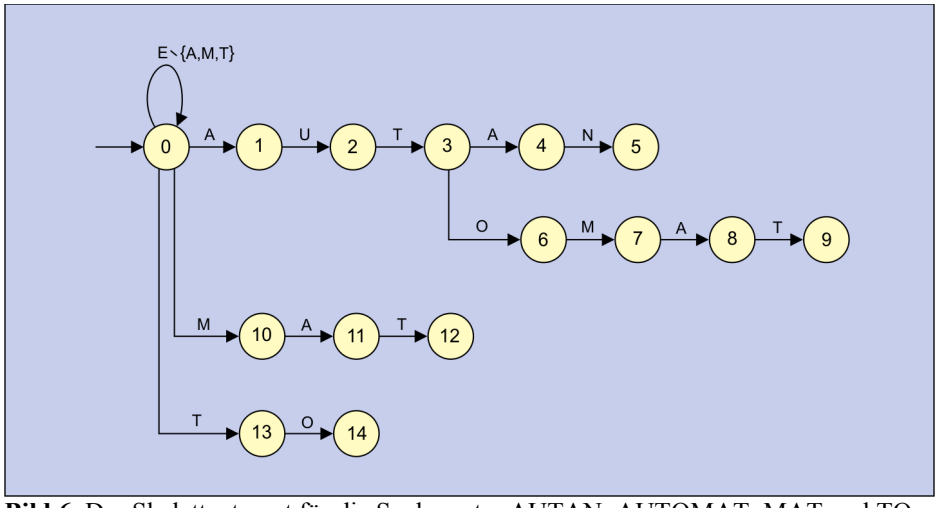

Bild 6. Der Skelettautomat für die Suchmuster AUTAN, AUTOMAT, MAT und TO

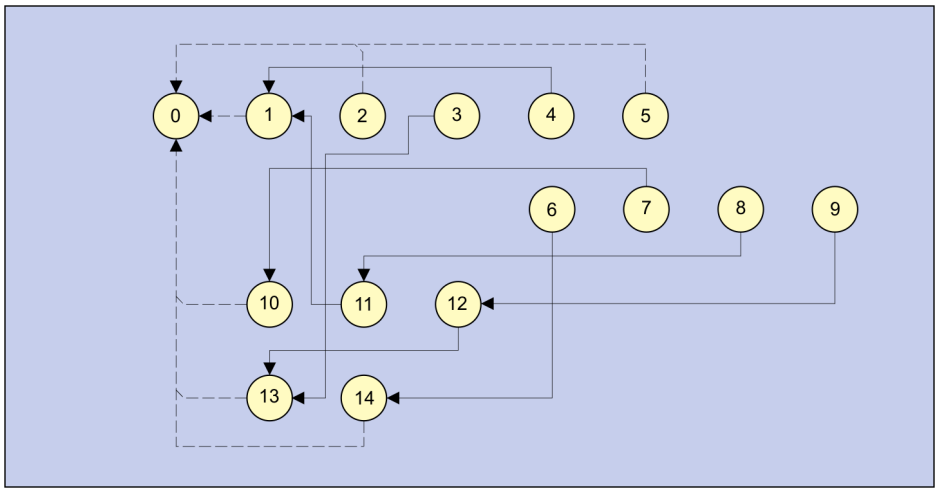

Bild 7. Die zu Bild 6 gehörige Failure-Funktion als Grafik

Tabelle 2. Die zu Bild 6 gehörige Failure-Funktion															
Zustand	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Wert der Failure-Funktion	0	0	0	13	1	0	14	10	11	12	0	1	13	0	0

Tabelle 3. Die zu Bild 6 gehörige Output-Funktion					
Zustand	5	6	9	12	14
Output-Menge	{AUTAN}	{TO}	{AUTOMAT} {MAT}	{MAT}	{TO}

Knuth, Morris, Pratt und Boyer, Moore

Der Algorithmus von Knuth, Morris, Pratt arbeitet nach dem oben geschilderten Verfahren [1]. Das zu suchende Muster wird vor der eigentlichen Suche vorkompiliert und anschließend auf den zu durchsuchenden Eingabestrom angewandt. Bei der Vorkompilierung wird ein entsprechender, etwas optimierter Suchautomat erzeugt. Der Algorithmus bringt also dann einen größeren Vorteil, wenn vor einer Nichtübereinstimmung eine möglichst lange Übereinstimmung zwischen Muster und Eingabestrom gefunden wurde. Damit verbessert der Knuth-Morris-Pratt Algorithmus die Mustersuche, und bietet darüber hinaus den Vorteil, daß der Eingabestrom nicht gepuffert werden muß. Eine andere Methode, die auf endlichen Automaten basiert haben Boyer und Moore erfunden. Sie vergleichen das Muster nicht von vorne mit dem Eingabestrom, sondern von hinten.

Mustervergleich von hinten

Dadurch kann bei einer Nichtübereinstimmung unter Umständen eine ganze Musterlänge übersprungen werden. Das führt nicht nur im schlimmsten Fall zu einem geringeren Aufwand, sondern auch im Durchschnitt. Erkauft wird dieser Vorteil dadurch, daß der Eingabestrom teilweise gepuffert werden muß.
Das C-Programm am Ende des Artikels *Listing 1* erzeugt einen sehr allgemein gehaltenen Suchautomaten für gleichzeitiges Suchen nach mehreren Mustern. Der Automat wird mit Hilfe eines Feldes von Strukturen vom Typ UEBERGANG dargestellt. Diese Struktur ist in der Datei DETAUT.H definiert *(Listing 2)*. In dieser Struktur sind die Output-, Goto- und Failure-Funktion enthalten. Diese Funktionen werden bei dem Aufbau des Automaten berechnet und in der UEBERGANG-Struktur für jeden Knoten festgehalten. Der Automat wird von der Funktion *buildda()* erzeugt. Diese Funktion baut zuerst den Skelettautomaten auf. Anschließend berechnet sie die Failure-Übergänge und die Output-Funktion. Die Funktionswerte werden in der UEBERGANG-Struktur gespeichert, damit sie bei der Mustersuche nicht berechnet werden müssen. Die eigentliche Mustersuche erfolgt mit Hilfe der Funktion *dafind()*. Ihr wird der zu durchsuchende Eingabestrom zeichenweise übergeben. Als Ergebnis erhält man den Index des Zustandes bei dem eine Übereinstimmung gefunden wurde oder einen ungültigen Index, die Konstante NOTFOUND. Um die Indizes der gefundenen Muster zu erhalten, muß man in den entsprechenden UEBERGANG-Strukturen die Komponenten *status* und *output* abarbeiten. Am einfachsten macht man das so, wie in der Funktion *scan()* in der Datei F_CMP.C *(Listing 3)*. Um das Laufzeitverhalten der Funktionen zu verbessern, setzt man am besten bei der Speicherverwaltung an. Dabei sollte man den von *malloc()* angeforderten Speicher durch statischen ersetzen. Weitere malloc()-Aufrufe lassen sich vermeiden, wenn man die in der Datei DETAUT.H definierte Konstante A_VERZW vergrößert.
Die Datei F_CMP.C enthält ein Anwendungsbeispiel. Es handelt sich dabei um ein Programm mit dem man

Dateien nach verschiedenen Mustern gleichzeitig durchsuchen kann. Das Programm wird mit dem Dateinamen der zu durchsuchenden Datei, gefolgt von den zu suchenden Mustern, aufgerufen. Als Include-Datei wird F_CMP.H benötigt *(Listing 4)*.
Als weitere Anwendungsgebiete für gleichzeitige Mustersuche bietet sich die lexikalische Analyse von Quelltexten oder deren Tokenisierung an. Dazu erzeugt man beispielsweise einen Automaten der alle Schlüsselwörter einer Programmiersprache sucht. Da dieser Automat nicht dauernd geändert werden muß, wäre es sinnvoll ihn als statische Variable abzulegen.
Um ein Schlüsselwort eindeutig identifizieren zu können, kann es notwendig werden, zusätzliche Zeichen vor oder nach den eigentlichen Schlüsselwörtern zu berücksichtigen. Denn immer dann, wenn Schlüsselwörter als Variablen- oder Prozedurnamen oder als Teile davon zugelassen sind, kann nur aus dem Kontext ermittelt werden, ob es sich um ein Schlüsselwort handelt. Den vollständigen Quelltext der Programme DETAUT.C und F_CMP.C sowie der Include-Dateien DETAUT.H und F_CMP.H finden Sie auf Seite 125 ff.

Literatur

[1] *Albert, J., Ottmann, T.:* Automaten, Sprachen und Maschinen für Anwender. B.I. Wissenschaftsverlag, 1987.
[2] *Wirth, N.:* Algorithmen und Datenstrukturen mit Modula-2. B.G. Teubner, 1986.
[3] *Kernighan, Ritchie:* Programmieren in C. Hanser, 1986

Listing 1. DETAUT.C erzeugt einen determinierten endlichen Automaten

```
/* DETAUT.C                                                          */
/*                                                                   */
/* Erzeugt einen determinierten Automaten aus beliebig vielen Such- */
/* begriffen und stellt dazugehörige Funktionen zur Verfügung.       */
/*                                                                   */
/* Funktionen: buildda()        Automat erzeugen.                    */
/*             dafree()         Automat löschen.                     */
/*             dafind()         Automat auf Eingabestrom anwenden.   */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "detaut.h"

UEBERGANG **buildda(muster, anz)
/* Erzeugt einen determinierten Automaten aus den übergebenen       */
/* Suchbegriffen. Die einzelnen Suchmuster müssen als '\0' ter-     */
/* minierte Zeichenketten übergeben werden.                         */
/* Fehler: NULL allgemeine Fehlermeldung, Speicherplatzmangel.      */

char *muster[];           /* Zeiger auf die zu suchenden Muster.    */
int anz;                  /* Anzahl der zu suchenden Muster.        */
{
  int len=0,              /* Summe der Zeichen in muster[].         */
      i;                  /* Zählvariable.                          */
  UEBERGANG **temp;       /* Ergebnis, Zeiger auf den Automaten.    */

  for (i=0; i < anz; i++)        /* Maximal nötige Zahl von Über-   */
    len+=(int)strlen(muster[i]); /* gängen und maximal in bdafail() */
                                 /* nötige Pufferlänge.             */
  if ( (temp=bdagoto(muster, anz, len)) == NULL )
    return NULL;               /* Die Grundstrukturen des Automaten */
                               /* und die Gotofunktion erzeugen.    */

  if ( bdafail(temp, len) )    /* Die Failureeinträge für die ein-  */
  {                            /* zelnen Zustände des Automaten be- */
    dafree(temp, -1);          /* rechnen.                          */
    return NULL;
  };

  bdaoutput(temp);

  return temp;
}

static UEBERGANG **bdagoto(muster, anz, len)
  /* Erzeugt die Gotofunktion für den determinierten Automaten und  */
  /* allokiert den benötigten Speicher, (interne Funktion)          */

  /* Fehler: NULL goto-Funktion kann nicht erzeugt werden wegen     */
  /*         Speicherplatzmangel.                                   */
char *muster[];           /* Zu suchende Muster.                    */
int anz;                  /* Anzahl der zu suchenden Muster.        */
int len;                  /* Summe der Zeichen in muster[].         */
{
  UEBERGANG **org;        /* Zeiger auf den Automaten.              */
  int orglen=0,           /* Aktueller Knoten.                      */
      orgmax=1,           /* Letzter Knoten.                        */
      i, j, max;          /* Hilfsvariablen.                        */

  if ( (org=(UEBERGANG **)malloc((len+2)*sizeof(UEBERGANG *)))==NULL )
    return NULL;

  org[0]=NULL;                /* Initialisierung => Automat leer.  */
  for (i=0; i < anz; i++)
  {                       /* Alle Suchmuster in den Automaten einfü- */
                          /* gen. Benötigt werden len=anz*max(i) Auf- */
```

```
                        /* rufe von insert().                          */
    orglen=0;                   /* Initialisierung => eventuelle       */

    max=(int)strlen(muster[i]); /* Gleichheit mit vorherigem Such-     */
                                /* muster beim Einfügen berück-        */
                                /* sichtigen.                          */
    for (j=0; j < max; j++)
    {                    /* Nächstes Suchmuster zeichenweise in den    */
                         /* bestehenden Automaten einfügen.            */
      if ( (orglen=insert(org, orglen, orgmax, muster[i][j]))==ALCERR )
      {
        dafree(org, orgmax);   /* Automat löschen.                     */
        return NULL;
      };
      if ( orglen == orgmax )   /* Orgmax ist der Index des letzten    */
        orgmax++;               /* Eintrages in dem Übergangsfeld.     */
    };
    org[orglen]->status=i;      /* => Muster i gefunden.               */
  };

  org[orgmax]=NULL;          /* Endemarkierung damit die Liste auch    */
                             /* ohne Längenangabe gelöscht werden      */
                             /* kann.                                  */
  return org;
}

static int insert(org, pos, end, c)
  /* Fügt ein Zeichen (Zustand) in den determinierten Automaten ab     */
  /* der Stelle pos ein. Gibt den Index zurück unter dem das Zei-      */
  /* chen eingefügt wurde.   (interne Funktion)                        */

  /* Fehler: ALCERR Der dafür benötigte Speicherplatz kann nicht       */
  /*         allokiert werden.                                         */

UEBERGANG **org;   /* Zeiger auf den determinierten Automaten.        */
int pos,           /* Suchanfang. (Index des letzten zu dem aktuel-   */
                   /* len Muster gehörenden Eintrages)                */
    end;           /* Listenende. (Index für Neueinträge)             */
char c;            /* Einzufügendes Zeichen.                          */
{
  int i,           /* Zählvariable.                                   */
      cmp, ver,    /* Temporäre Speicher für Vergleichswerte.         */
      test;        /* Flag.                                           */
  int  *pver;      /* Hilfszeiger für Kopieren und Allokieren.        */
  char *pein;

  test=( org[pos] != NULL ) ? ( org[pos]->z == 0 ) : 0;
                             /* Flag für Eintrag erzeugen.           */
  if ( org[pos] == NULL || test )
  {                          /* Fall: Eintrag existiert nicht oder   */
                             /* leerer Eintrag -> letzter Eintrag.   */
    if ( !test )             /* Eintrag erzeugen.                    */
      if ( (org[pos]=(UEBERGANG *)malloc(sizeof(UEBERGANG))) == NULL )
        return ALCERR;

    org[pos]->z=1;              /* Eintrag gefunden.                  */
    org[pos]->eing.s[0]=c;      /* Initialisieren.                    */
    org[pos]->verw.v[0]=end;

    if ( !test )                /* => Eintrag existiert, d.h. das     */
      org[pos]->status=NOTFOUND; /* Statusfeld ist schon gesetzt.     */

    if ( (org[end]=(UEBERGANG *)malloc(sizeof(UEBERGANG))) == NULL )
      return ALCERR;            /* Neuen Endknoten erzeugen.          */

    org[end]->z=0;              /* Endknoten initialisieren.          */
    org[end]->status=NOTFOUND;

    return end;                 /* Neuer größter Index.               */
```

```
};

for (i=0; i < org[pos]->z; i++)
{                               /* Überprüfen ob das einzufügende  */
                                /* Zeichen bereits vorhanden ist.  */

  if ( org[pos]->z > A_VERZW ) /* Vergleichszeichen und Verweis   */
  {                             /* je nach Grad der Belegung       */
    cmp=org[pos]->eing.sp[i];  /* lesen.                          */
    ver=org[pos]->verw.vp[i]; }
  else
  {
    cmp=org[pos]->eing.s[i];
    ver=org[pos]->verw.v[i];
  };

  if ( c == cmp )               /* Eingabe schon vorhanden =>     */
    return ver;                 /* kein neuer Eintrag.            */
};

if ( org[pos]->z >= A_VERZW )   /* Umkopieren, neu allokieren.    */
{
  if ( org[pos]->z == A_VERZW ) /* Sonderfall: 1. Mal dynamische  */
  {                             /* Verwaltung.                    */
    if ( (pein=(char *)malloc((A_VERZW+1)*sizeof(char))) == NULL )
      return ALCERR;
    if ( (pver=(int  *)malloc((A_VERZW+1)*sizeof(int ))) == NULL )
    {
      free(pein);
      return ALCERR;
    };
                                /* Daten umkopieren.              */
    memcpy(pein, org[pos]->eing.s, A_VERZW*sizeof(char));
    memcpy(pver, org[pos]->verw.v, A_VERZW*sizeof(int ));

    org[pos]->eing.sp=pein;
    org[pos]->verw.vp=pver;
  }
  else                          /* Vorhandene Zeigerfelder ver-   */
  {                             /* größern.                       */
    if ( (pein=(char *)malloc((i+1)*sizeof(char))) == NULL )
      return ALCERR;
    if ( (pver=(int  *)malloc((i+1)*sizeof(int ))) == NULL )
    {
      free(pein);
      return ALCERR;
    }
                                /* Daten umkopieren.              */
    memcpy(pein, org[pos]->eing.sp, i*sizeof(char));
    memcpy(pver, org[pos]->verw.vp, i*sizeof(int ));

    free(org[pos]->eing.sp);
    free(org[pos]->verw.vp);

    org[pos]->eing.sp=pein;
    org[pos]->verw.vp=pver;
  };
  org[pos]->eing.sp[i]=c;       /* Zeichen und Index eintragen.   */
  org[pos]->verw.vp[i]=end;
}
else
{
  org[pos]->eing.s[i]=c;        /* Zeichen und Index eintragen.   */
  org[pos]->verw.v[i]=end;
};
org[pos]->z++;                  /* Verzweigungszähler erhöhen.    */

if ( (org[end]=(UEBERGANG *)malloc(sizeof(UEBERGANG))) == NULL )
  return ALCERR;
```

```
  org[end]->z=0;                  /* Neuer Endknoten.              */
  org[end]->status=NOTFOUND;

  return end;
}

void dafree(org, orglen)
  /* Determinierten Automat löschen. Ist orglen == -1 wird solange  */
  /* gelöscht bis org[i] == NULL ist.                               */

  /* Fehler: Keine.                                                 */
UEBERGANG **org;     /* Zeiger auf den Automaten.                  */
int orglen;          /* Anzahl der Zustände des Automaten.         */
{
  int i=0;                       /* Zählvariable.                  */
  if ( orglen == -1 )            /* Löschen bei unbekannter Zustand- */
    while ( org[i] != NULL )     /* anzahl. Aufruf erfolgt meist bei */
    {                            /* einem Fehler während der Auto-   */
      if ( org[i]->z > A_VERZW )/* mat erzeugt wird.               */
      {
        if ( org[i]->eing.sp != NULL )
        {
          free(org[i]->eing.sp);
          if ( org[i]->verw.vp != NULL )
            free(org[i]->verw.vp);
        };
      };
      free(org[i]);
      i++;
    }
  else                           /* Ordnungsgemäßes Freigeben des    */
  {                              /* Automaten bei bekannter Zustands-*/
                                 /* zahl.                            */
    for (i=0; i < orglen; i++)
    {
      if ( org[i]->z > A_VERZW )
      {
        free(org[i]->eing.sp);
        free(org[i]->verw.vp);
      };
      free(org[i]);
    };
  };
  free(org);
}

static int dagoto(a, zu, e)
  /* Ermittelt den durch die Eingabe e im Zustand zu neu erreichten */
  /* Zustand. Ist die Eingabe e an dieser Stelle nicht zulässig,    */
  /* wird FAIL zurückgegeben, sonst der neue Zustand.               */
  /*                                  (interne Funktion)            */

  /* Fehler: Keine.                                                 */

UEBERGANG **a;       /* Zeiger auf den Automaten.                  */
int zu;              /* Zustand in dem sich der Automat befindet.  */
char e;              /* Eingabezeichen.                            */
{
  char test;                    /* Vergleichszeichen.              */
  int i;                        /* Zählvariable.                   */
   i=a[zu]->z;                  /* Anzahl der Vezweigungen vom aktu- */
                                /* ellen Zustand aus.              */
  while ( i-- > 0 )             /* Eingabezeichen mit allen erlaub-  */
  {                             /* ten Eingabezeichen vergleichen.   */
    test=(a[zu]->z > A_VERZW) ? a[zu]->eing.sp[i] : a[zu]->eing.s[i];

    if ( test == e )
      return (a[zu]->z > A_VERZW) ? a[zu]->verw.vp[i] : a[zu]->verw.v[i];
```

```
  };
  if ( zu == 0 )               /* Von Zustand 0 wird bei allen      */
    return 0;                  /* Falscheingaben Zustand 0 erreicht */

  return FAIL;                 /* Eingabe ist nicht erlaubt.        */
}

static int bdafail(org, len)
  /* Berechnet die Failure-Übergänge und trägt diese bei den        */
  /* jeweiligen Zuständen ein.         (interne Funktion)           */

  /* Fehler: ALCERR                                                 */
UEBERGANG **org;            /* Zeiger auf den Automaten.            */
int len;                    /* Größtmögliche Länge von schlange.    */
{
  char c;                   /* Eingabezeichen für Automaten.        */
  int *schlange,            /* FiFo-Puffer zur failure Berechnung.  */
      r, s, zu,             /* Zwischenspeicher für Zustände.       */
      i;                    /* Zählvariable.                        */
  unsigned int start=0,     /* Erster Eintrag in schlange.          */
               end=0;       /* Letzter Eintrag in schlange.         */
  if ( (schlange=(int *)malloc((len+1)*sizeof(int))) == NULL )
    return ALCERR;

  for (i=0; i < org[0]->z; i++)
  {
    s=(org[0]->z > A_VERZW) ? org[0]->verw.vp[i] : org[0]->verw.v[i];
    schlange[end++]=s;      /* schlange = schlange U s              */
    org[s]->f=0;            /* failure(s) == Zustand 0              */
  };
  while ( start%len != end%len )
  {
    r=schlange[(start++)%len];

    for (i=0; i<org[r]->z; i++)
    {
      s=(org[r]->z > A_VERZW) ? org[r]->verw.vp[i] : org[r]->verw.v[i];
      c=(org[r]->z > A_VERZW) ? org[r]->eing.sp[i] : org[r]->eing.s[i];

      schlange[(end++)%len]=s;   /* schlange = schlange U s         */
      zu=org[r]->f;

      while ( dagoto(org, zu, c) == FAIL )
        zu=org[zu]->f;

      org[s]->f=dagoto(org, zu, c);
    };
  };
  org[0]->f=0;

  free(schlange);               /* Ringpuffer wieder freigeben.    */

  return OK;
}

static void bdaoutput(org)
  /* Erzeugt die Output-Funktion. Die bei einem bestimmten Zustand  */
  /* gefundenen Muster werden über die output-Komponente der UEBER- */
  /* GANG-Struktur in einer Liste verkettet.                        */

  /* Fehler: Keine.                                                 */
UEBERGANG **org;      /* Zeiger auf den determinierten Automaten.   */
{
  int zustand=-1,                  /* Aktueller Zustand.           */
      temp;                        /* Zwischenspeicher.            */

  while ( org[++zustand] != NULL )  /* Alle Zustände abarbeiten.    */
  {                                 /* Dabei überprüfen ob bei einem*/
    org[zustand]->output=NOTFOUND;  /* Zustand ein oder mehrere     */
```

```
                                    /* Muster gefunden wurden.      */
    if ( org[zustand]->f == 0 )
      continue;
    temp=zustand;
    while ( (temp=org[temp]->f) != 0 )
    {
      if ( org[temp]->status != NOTFOUND )
      {
        org[zustand]->output=temp;
        break;
      };
    };
  };
}

int dafind(org, c, flag)
  /* Sucht mit dem durch org beschriebenen Automaten nach Mustern.  */
  /* Dabei wird der Eingabestrom durch c zeichenweise an die Funk-  */
  /* tion übergeben. Als Ergebnis erhält man den Index des Zu-      */
  /* standes bei dem Übereinstimmungen gefunden wurden. Die Über-   */
  /* einstimmungen können in der status-Komponente, in der output-  */
  /* Liste oder in beiden verzeichnet sein.                         */

  /* Fehler: Keine.                                                 */
UEBERGANG **org;            /* Zeiger auf den Automaten.          */
char c;                     /* Aktuelles Zeichen des Eingabestroms. */
int flag;                   /* Flag: Automat zurücksetzen.        */
{
  static int zustand=0;     /* Aktueller Zustand.                 */
  int temp;                 /* Zwischenspeicher.                  */

  if ( flag == FIRST )      /* Initialisierung. Der Automat wird in */
    zustand=0;              /* den Startzustand zurückversetzt und  */
                            /* ermöglicht so die Überprüfung      */
                            /* eines neuen Eingabestroms.         */

  if ( (temp=dagoto(org, zustand, c)) == FAIL )
  {                         /* zustand = goto(failure(zustand), c). */
                            /* Das neu eingegebene Zeichen führt  */
                            /* zu keinem erlaubten Zustand.  =>   */
                            /* failure-Pfad abarbeiten.           */
    zustand=org[zustand]->f;
    return dafind(org, c, flag);
  };

  zustand=temp;             /* zustand = goto(zustand, c).        */
                            /* Das neu eingegebene Zeichen stimmt */
                            /* mit dem aktuell untersuchten Muster  */
                            /* überein.                           */
  if (org[zustand]->status != NOTFOUND || org[zustand]->output != NOTFOUND)
    return zustand;         /* Überprüfen ob in diesem Zustand    */
                            /* bereits ein Muster gefunden wurde. */

  return NOTFOUND;          /* Kein Muster gefunden.              */
}
```

Listing 2. Von DETAUT.C benötigte Include-Datei

```
#ifndef DETAUTH
#define DETAUTH

/* DETAUT.H                                                    */
/*                                                             */
/* Include Datei für DETAUT.C.                                 */

  /* Definitionen.                                             */
#define NOTFOUND  -1            /* Vorbesetzung für Übergänge.     */
                                /* Meldung von dafind().           */

#define FAIL      -1            /* Goto: Illegale Eingabe.         */

#define FIRST      0            /* Modi von dafind().              */
#define NEXT       1

  /* Fehlermeldungen.                                          */
#define OK        0
#define ALCERR  -1              /* Allokations-Fehler.             */

  /* Typendeklarationen.                                       */
#define A_VERZW  2          /* Anzahl von Verzweigungen ab der ein  */
                            /* Knoten dynamisch verwaltet wird.     */
                            /* >= sizeof(char *)/sizeof(int)        */

typedef struct uebergang    /* Struktur die einen Knoten (Zustand)  */
{                           /* des endlichen Automaten beschreibt.  */

  int status;                   /* Gefundenes Muster oder NOTFOUND. */
  int z;                        /* Anzahl der Verzweigungen.        */
  int output;                   /* Zeiger auf eine Liste mit den    */
                                /* bisher gefundenen Mustern.       */
  union
  {                             /* Eingabezeichen bei denen die     */
    char *sp;                   /* failure-Funktion nicht aufgerufen*/
    char s[A_VERZW];            /* werden muß.                      */
  } eing;

  union
  {                             /* Zu obigen Eingabezeichen korres- */
    int *vp;                    /* pondierende Nachfolgeknoten.     */
    int v[A_VERZW];
  } verw;

  int f;             /* Wert der failure-Funktion. Index des neuen  */
                     /* Zustandes, wenn das Eingabezeichen nicht in */
                     /* eing.s[] (eing.sp[]) verzeichnet war.       */
} UEBERGANG;

/* Funktions-Prototypen.                                       */

  /* EXTERNE Funktionen.                                       */
UEBERGANG **buildda(char *muster[], int anz);
int         insert(UEBERGANG **org, int pos, int end, char c);
int         dafind(UEBERGANG **org, char c, int flag);

  /* INTERNE Funktionen.                                       */
UEBERGANG  **bdagoto(char *muster[], int anz, int len);
void         dafree(UEBERGANG **org, int orglen);
int          dagoto(UEBERGANG **a, int zu, char e);
int          bdafail(UEBERGANG **org, int len);
void         bdaoutput(UEBERGANG **org);

#endif
```

Listing 3. F_CMP.C durchsucht Dateien nach verschiedenen Mustern

```
#include <stdio.h>
#include <stdlib.h>

#include "detaut.h"
#include "f_cmp.h"

int main(argc, argv)
  /* Datei nach mehreren Suchmustern durchsuchen. Als Ausgabe er-   */
  /* hält man die Zeilennummer in der das Muster gefunden wurde     */
  /* und das Muster.                                                */
  /*                                                                */
  /* Aufruf: name Datei Muster1 Muster2 Muster3 ...                 */
int  argc;
char *argv[];
{
  int         erg;            /* Anzahl der gefundenen Muster.      */
  FILE       *fp;             /* Zeiger auf die Eingabedatei.       */
  UEBERGANG **automat;        /* Zeiger auf den Automaten.          */

  if ( argc < 3 )
    puterr("F_CMP", ARGERR);

  if ( (fp=fopen(argv[1], "r")) == NULL )
    puterr("F_CMP", OPNERR);

  if ( (automat=buildda(&(argv[2]), argc-2)) == NULL )
    puterr("F_CMP", ALCERR);

  erg=scan(fp, automat, &(argv[2]));    /* Datei durchsuchen.       */

  printf("\n%s: %d Übereinstimmungen gefunden.\n", "F_CMP", erg);

  fclose(fp);
  dafree(automat, -1);
  return OK;
}

int scan(datei, org, muster)
  /* Durchsucht eine Datei nach verschiedenen Mustern und gibt diese*/
  /* mit der entsprechenden Zeilennummer aus. Als Ergebnis wird die */
  /* Anzahl der gefundenen Muster zurückgegeben.                    */

  /* Fehler: Keine.                                                 */
FILE *datei;            /* Die zu durchsuchende Datei.            */
UEBERGANG **org;        /* Zeiger auf Automaten.                  */
char **muster;          /* Die Suchmuster.                        */
{
  int c,                /* Aktuelle Zeichen aus Eingabestrom.     */
      pat,              /* Ergebnis von dafind().                 */
      mus,              /* Gefundenes Muster.                     */
      znum=1,           /* Aktuelle Zeilennummer.                 */
      anz=0;            /* Anzahl Fundstellen.                    */

  while ( (c=getc(datei)) != EOF )
  {
    if ( (char)c == '\n' )          /* Zeilen zählen.             */
      znum++;

    if ( (pat=dafind(org, (char)c, NEXT)) != NOTFOUND )
    {                               /* Gefundene Muster ausgeben. */
      if ( (mus=org[pat]->status) != NOTFOUND )
      {
        anz++;
        printf("%6d: %s \n", znum, muster[mus]);
      };
      while ( (pat=org[pat]->output) != NOTFOUND )
      {
        anz++;
```

```
        printf("%6d: %s \n", muster[org[pat]->status]);
      };
    };
  };
  return anz;
}

static char *errors[] =
{
  "Fehler bei der Erzeugung des Automaten wegen Speichermangel.",
  "Eingabedatei konnte nicht geöffnet werden.",
  "Zuwenig Eingabeargumente."
};

void puterr(name, fnum)

  /* Fehlermeldung über stderr ausgeben und mit exit() terminieren. */
char *name;        /* Name des Programmes.                          */
int  fnum;         /* Index der Fehlermeldung.                      */
{
  fprintf(stderr,"\n%s: %s\n", name, errors[abs(fnum+1)]);

  exit(fnum);
}
```

Listing 4. Von F_CMP.C benötigte Include-Datei

```
#ifndef F_CMPH
#define F_CMPH

/* F_CMP.H                                                          */
/*                                                                  */
/* Include Datei für F_CMP.C.                                       */

  /* Fehlermeldungen.                                               */
#define OK       0
#define OPNERR  -2      /* Datei kann nicht geöffnet werden.        */
#define ARGERR  -3      /* Zuwenig Eingabeargumente.                */

  /* Funktionsprototypen.                                           */
int   main(int argc, char *argv[]);
int   scan(FILE *fp, UEBERGANG **org, char *argv[]);
void  puterr(char *name, int fnum);

#endif
```

aus mc 07/90 – Seite 120-128