

Eine Einführung in die theoretische Informatik

Theoretische Informatik bedient sich der Methoden und Modelle aus der Mathematik zur Formulierung von Algorithmen und zur Rechnerkonstruktion. Sie befasst sich mit der Theorie der Formalen Sprachen, mit der Automatentheorie, der Theorie der Datentypen, der Berechenbarkeit, der Komplexität und der Semantik.

Endliche Automaten mit Ausgabe

Im täglichen Leben begegnen uns überall die verschiedensten Automaten. Durch Abstraktion lässt sich die mehr oder minder komplexe Funktionsweise der Automaten im Modell darstellen. Mit Hilfe derartiger abstrakter Modelle sind aber auch algorithmische Lösungen zur Syntaxanalyse bei Programmiersprachen beschreibbar.

Ein Endlicher Automat mit Ausgabe (Transduktor) soll hier an einem **Beispiel** verdeutlicht werden:

Automatentafel: Getränkeautomat

Als Beispiel soll ein Getränkeautomat dienen. Alle Getränke kosten 1,50 DM. Es können 1 DM - und 50 Pf - Münzen eingeworfen werden. Wird beim Einwurf einer Münze der Preis von 1,50 DM überschritten, dann fällt die Münze in das Geldausgabefach. Wenn der Betrag von 1,50 DM eingeworfen wird, dann kann durch Drücken der Wahl taste das gewünschte Getränk ausgewählt werden. Das Betätigen der Korrekturtaste führt zur Ausgabe des bisher eingeworfenen Geldes.

Die Eingabemöglichkeiten sind: 50Pf, 1DM, Wahl taste, Korrekturtaste.

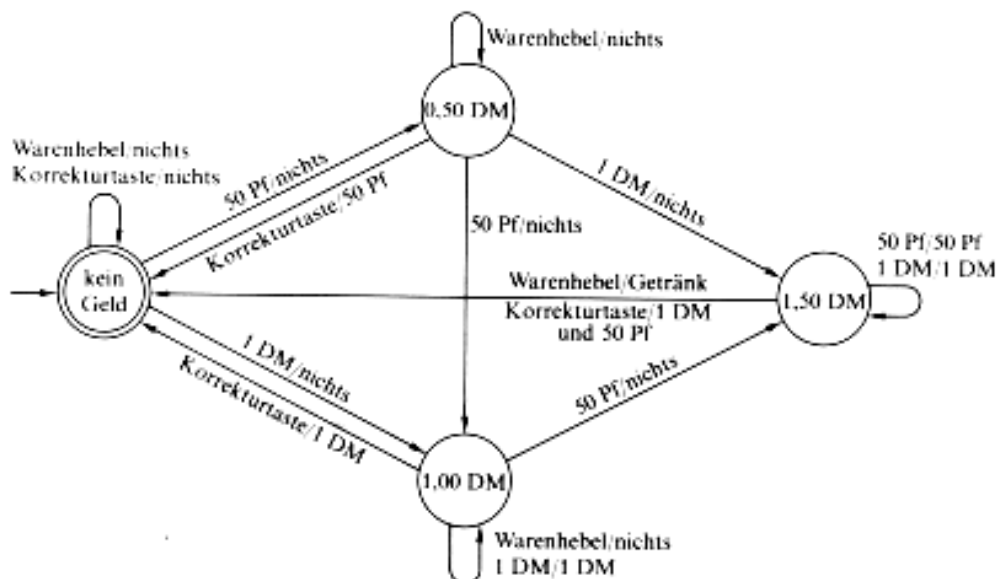
Die Ausgabemöglichkeiten sind: 50Pf, 1DM, 1DM und 50Pf, Getränk, nichts.

Die Zustände des Getränkeautomaten sind: **kein Geld, 0,50 DM, 1,00 DM, 1,50 DM**; der Zustand **kein Geld** ist sowohl Anfangs-, als auch Endzustand .

Eine Automatentafel fasst dies alles zusammen:

Zustand \ Eingabe	kein Geld	0,50DM	1,00DM	1,50DM
50Pf	0,50DM / nichts	1,00DM / nichts	1,50DM / nichts	1,50DM / 0,50DM
1DM	1,00DM / nichts	1,50DM / nichts	1,00DM / 1,00DM	1,50DM / 1,00DM
Warenhebel	kein Geld / nichts	0,50DM / nichts	1,00DM / nichts	kein Geld / Getränk
Korrekturtaste	kein Geld / nichts	kein Geld / 0,50DM	kein Geld / 1,00DM	kein Geld / 1,50DM

Die Arbeitsweise des Automaten kann auch durch einen **gerichteten Graphen** verdeutlicht werden:



Ein Automat besitzt eine endliche Anzahl von Zuständen. Der Übergang von einem Zustand in einen anderen geschieht durch eine entsprechende Eingabe. Für jede Eingabe hat man einen bestimmten zugeordneten Folgezustand. Zu Beginn befindet sich der Automat in seinem Anfangszustand und bei korrekter Behandlung am Schluss in seinem Endzustand. Mit jeder Eingabe erfolgt auch eine Ausgabe.

Beispiel: Glücksspielautomat

Ein vereinfachter Glücksspielautomat in Form eines einarmigen Banditen besitzt zwei Walzen mit den Symbolen **Apfel**, **Birne** und **Krone**. Erscheinen nach Betätigung des Hebels zwei **Kronen** in den Sichtfenstern, erhält der Spieler den doppelten Einsatz, bei zwei gleichen sonstigen Symbolen wird der Einsatz von 1,- DM zurückgezahlt. Neben den Symbolkombinationen im Fenster sind noch Münzeinwurf von 1,- DM und **Hebel betätigen** mögliche Eingaben des Automaten.

Ein Pascal-Programm simuliert diesen Automaten:

```
PROGRAM Steffen;
uses crt;
TYPE ASTRA=STRING[2];
VAR
  ins, zu1, zu2: INTEGER;
  zu3: ASTRA;
  ch: CHAR;

PROCEDURE Konto (a: INTEGER);
BEGIN
  ins := ins+a;
END;

PROCEDURE Ausgabe (ein: ASTRA);
BEGIN
  IF ein = 'KK' THEN
  BEGIN
    Writeln ('*****');
    Writeln ('* 1DM Gewinn *');
    Writeln ('*****');
    Konto(2);
  END;
  IF ein = 'AA' THEN
  BEGIN
    Writeln ('*****');
    Writeln ('* Einsatz zur ck *');
    Writeln ('*****');
    Konto(1);
  END;
  IF ein = 'BB' THEN
  BEGIN
    Writeln ('*****');
    Writeln ('* Einsatz zur ck *');
    Writeln ('*****');
    Konto(1);
  END;
  IF ein = 'AB' THEN
    Writeln ('!!! Leider Verloren !!!');
  IF ein = 'BA' THEN
    Writeln ('!!! Leider Verloren !!!');
  IF ein = 'KA' THEN
    Writeln ('!!! Leider Verloren !!!');
  IF ein = 'AK' THEN
    Writeln ('!!! Leider Verloren !!!');
  IF ein = 'BK' THEN
    Writeln ('!!! Leider Verloren !!!');

End;

PROCEDURE Mat;
```

```

VAR auto:ARRAY [0..8] OF ASTRA;

BEGIN
  auto[0]:='KK'; auto[1]:='KA'; auto[2]:='KB';
  auto[3]:='AK'; auto[4]:='AA'; auto[5]:='AB';
  auto[6]:='BK'; auto[7]:='BA'; auto[8]:='BB';

  zu3:=auto[random(9)];
END;

PROCEDURE Hebel;
VAR ch:CHAR;
BEGIN
  REPEAT
    Writeln ('                Soll der Hebel betaetigt werden ?');
    Write ('                (j/n): ');
    ch :=readkey;
    CASE ch OF
      'j': zu2:=1;
      'n': zu2:=0;
      ELSE Writeln ('Falsche Taste!!!!');
    END;
  UNTIL (ch='j') OR (ch='n');
  IF ch='j' THEN
    Mat;
END;

PROCEDURE Geld;
VAR ch:CHAR;
BEGIN
  REPEAT
    Writeln ('                Ihr Kontostand: ',ins:0,' DM');
    writeln;
    Writeln ('                Ein Spiel kostet 1 DM, wollen Sie spielen ?');
    writeln;
    Write ('                (j/n): ');
    ch := readkey;
    Writeln;Writeln;
    CASE ch OF
      'j': BEGIN
              zu1:=1;
              Konto(-1);
            END;
      'n': zu1:=0;
      ELSE Writeln ('Falsche Taste!!!!');
    END;
  UNTIL (ch='j') OR (ch='n');
  Writeln ('                Neuer Kontostand: ',ins:0,' DM');
  IF ch='j' THEN
    Hebel;
END;

PROCEDURE Feld;
BEGIN
  Writeln;Writeln;
  Writeln ('                SPIELAUTOMAT');
  Writeln ('                -----');
  Writeln;Writeln;
  Writeln ('                -----');
  Writeln ('                / Es gibt drei verschiedene Symbole /');
  Writeln ('                / (A)pfel, (B)irne und (K)rone /');
  Writeln ('                / und zwei Walzen gleich best ckt. /');
  Writeln ('                / Wird AA oder BB gezogen, wird der /');

```

```

Writeln ('          / Einsatz, der 1,-DM betr„gt          /');
Writeln ('          / zur ckgezahlt.          /');
Writeln ('          / Zieht man jedoch KK, wird der          /');
Writeln ('          / Einsatz plus 1,-DM Gewinn gezahlt. /');
Writeln ('          / Alle anderen Kombinationen f hren /');
Writeln ('          / zu einem Verlust des Einsatzes. /');
Writeln ('          /          Na dann viel Gl ck !!! /');
Writeln ('          -----');
Writeln ('          Weiter mit <ENTER>');
Readln;
clrscr;
END;

BEGIN
textbackground(black);
clrscr;
write(',Layout Gangolf Kern');

window(10,3,70,23);
textbackground(blue);textcolor(white); clrscr;

    clrscr;
    ins:=5;
    Feld;
REPEAT
    clrscr;
    zu1:=0; zu2:=0;

    Writeln;Writeln;
    Geld;
    Writeln;
    Writeln ('          -----');
    Writeln ('          / Ihre Ergebnis: ',zu3,' /');
    Writeln ('          -----');
    Writeln;
    Ausgabe (zu3);
    Writeln;
    Writeln ('          Neuer Kontostand ',ins:0,' DM');
    Writeln;
    Write ('          Ein weiteres Spiel? (j/n): ');
    ch := readkey;
    Writeln;
UNTIL ch = 'n';
END.

```

Der erkennende Automat als Parser

Zeichenfolgen, die korrekt gebildet werden, können durch den Einsatz von erkennenden Automaten von nicht korrekt gebildeten Zeichenketten unterschieden werden. Man nutzt sie deshalb zur Überprüfung von Zeichenketten. Ein Beispiel dafür ist der Parser. Der Parser (to parse = zerlegen) ist ein Modul des Compilers und überprüft die korrekte Einhaltung der syntaktischen Regeln bei der Bildung von Zeichenketten. Anschließend wird zum Beispiel für ein PASCAL-Programm der Maschinencode erzeugt.

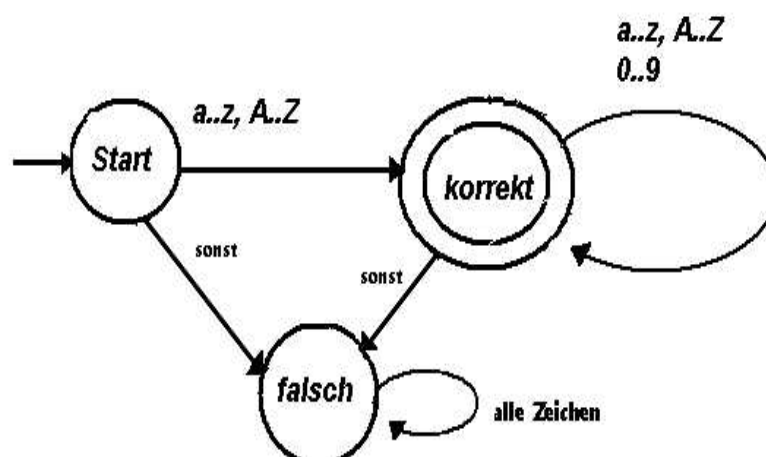
Der enge Zusammenhang zwischen erkennenden Automaten und Zeichenketten wird durch die folgenden üblichen Bezeichnungen unterstrichen:

Das Alphabet ist die Eingabemenge des Akzeptors (=endlicher Automaten ohne Ausgabe). In PASCAL besteht das Alphabet aus allen Zeichen des Typs CHAR. Ein Wort über dem Alphabet ist eine endliche Zeichenfolge aus Zeichen des Alphabets. Im PASCAL besteht diese Zeichenfolge aus max. 255 Zeichen. Akzeptierte Worte sind Worte, die den Akzeptor vom Zustand **start** in den Zustand **korrekt** überführen. Die Sprache des Akzeptors ist die Menge aller korrekten Bezeichner.

Aus dem entsprechenden **Zustandsgraphen** lässt sich relativ einfach ein Turbo-Pascal-Programm gewinnen!

Zustandsgraph für einen erkennenden Automaten

Dieser Beispielautomat überprüft, ob Wörter korrekt eingegeben wurden. Bei der Eingabe eines Buchstaben als erstes Zeichen geht der Automat vom Anfangszustand **start** in den Zustand **korrekt** über. Bei allen anderen Eingaben als erstes Zeichen wird jedoch der Zustand **falsch** erreicht. Das zweite und jedes folgende Zeichen kann dann ein Buchstabe oder eine Ziffer sein, um im Zustand **korrekt** zu verweilen. Jedes andere Zeichen bewirkt wiederum den Übergang in den Zustand **falsch**. Der Zustand **korrekt** ist der Endzustand.



```

PROGRAM akzeptor;
uses crt;
TYPE zustand = (s,f,k);
VAR l,i:INTEGER;
    zu:zustand;
    zei:char;
    bez:string;

FUNCTION uebergang(zei:char;zu:zustand):zustand;
BEGIN
CASE zu OF
s:IF zei in ['a'..'z','A'..'Z']
    THEN uebergang:=k
    ELSE uebergang:=f;
k:IF zei in ['a'..'z','A'..'Z','0'..'9','_']
    THEN uebergang:=k
    ELSE uebergang:=f;
f:uebergang:=f;
END;
END;

BEGIN
textbackground(black);
clrscr;
write(',Layout Gangolf Kern');

window(10,3,70,23);
textbackground(blue);textcolor(white); clrscr;

repeat
clrscr;
writeln('Eingabe der Zeichenkette:');
readln(bez);
l:=length(bez);
zu:=s;
FOR i := 1 TO l DO
    zu:=uebergang(bez[i],zu);
    if zu=k then writeln('Bezeichner korrekt')
    else writeln('Bezeichner nicht korrekt');
    writeln;

writeln('Weiter mit <enter>, Abbruch mit <e> !'); readln(zei)
until zei='e'
END.

```

Formale Sprachen

Einleitung

Um Informationen austauschen zu können, bedient man sich der Sprache. Dabei müssen Sätze nach bestimmten Regeln aufgestellt werden, damit ihre Bedeutung verstanden wird. Diese Regeln werden Grammatik genannt. Natürliche Sprachen haben einen hohen Wortschatzumfang und komplexe Grammatiken. Somit kann man komplizierte Sachverhalte sprachlich ausdrücken.

In technischen oder theoretischen Systemen reicht jedoch meistens eine geringe Wortzahl, um Informationen auszutauschen. Daher ist es nicht sinnvoll, sich an natürlichen Sprachen zu orientieren. Vielmehr versucht man, die Sprachen den Bedürfnissen der Systeme anzupassen. Dadurch entstehen formale Sprachen wie die mathematische Formelsprache und die Computersprachen.

Formale Sprachen sind nicht nur für den Informationsaustausch wichtig, sondern auch für die Grundlagenforschung. So werden etwa Methoden gewonnen zur Sprach- und Zeichenerkennung bei natürlichen Sprachen.

Im folgenden wird eine Definition Formaler Sprachen gegeben und es werden Beschreibungsmöglichkeiten erarbeitet. Es werden Methoden zur Analyse Formaler Sprachen vorgestellt, die auch im Zusammenhang mit der Untersuchung von PASCAL – Programmen auf syntaktische Korrektheit wichtig sind.

Die grammatikalischen Regeln einer Sprache bestimmen ihre Syntax. Bei Formalen Sprachen reichen häufig wenige Regeln und ein stark eingeschränkter Wortschatz aus. Wichtig ist, dass die grammatischen Regeln eindeutig sind.

Beispiel 1: Umgangssprachliche Sätze

Es werden folgende Sätze als kleiner Ausschnitt der deutschen Sprache untersucht:

"Die Katze jagt",

"Das Buch frisst Heu",

"Susanne liest das Buch".

Grammatikalisch gesehen haben diese Sätze die Nominalphrase und die Verbalphrase gemeinsam. Die Nominalphrase ist daran erkennbar, daß sie ein Nomen oder ein Pronomen enthält. Die Verbalphrase ergänzt die Nominalphrase zu einem vollständigen Satz. In Form einer Grammatikregel wird der Sachverhalt folgendermaßen ausgedrückt:

(1) <Satz> --> <Nominalphrase><Verbalphrase>

Der Pfeil wird als '**besteht aus**' gelesen. Die Nominalphrasen dieser Sätze bestehen aus Eigennamen oder einem Substantiv mit zugehörigem Artikel; dies formalisiert folgende Grammatikregel:

(2.1) <Nominalphrase> --> <Eigename>

(2.2) <Nominalphrase> --> <Artikel><Substantiv>

Die Verbalphrase besteht entweder aus einem Verb oder aus einem Verb mit Akkusativobjekt; letzteres ist wieder eine Nominalphrase:

(3.1) <Verbalphrase> --> <Verb>

(3.2) <Verbalphrase> --> <Verb><Nominalphrase>

Außerdem ergibt sich ein kleines Lexikon:

<Eigename> --> *Susanne*

<Substantiv> --> *Katze | Pferd | Heu | Buch*

<Artikel> --> *der | die | das*

<Verb> --> *jagt | frißt | liest*

Die Gesamtheit der Ersetzungsregeln (1) – (3) heißt **Grammatik**.

Anwendung: Es soll der Satz '**Susanne liest**' hergeleitet werden!

Die Herleitung eines Satzes geschieht dadurch, dass jeweils der linke Teil einer Regel durch deren rechten Teil ersetzt wird, wobei stets mit <Satz> zu beginnen ist:

<Satz>	=> <Nominalphrase><Verbalphrase>	nach Regel (1)
	=> <Eigename><Verbalphrase>	nach Regel (2.1)
	=> <i>Susanne</i> <Verbalphrase>	Lexikon
	=> <i>Susanne liest</i>	Lexikon

Den Pfeil liest man als '**wird ersetzt durch**'. Der zweite Beispielsatz oben macht deutlich, daß sich auch semantisch sinnlose Sätze herleiten lassen. Möchte man sicherstellen, dass eine Grammatik nur sinnvolle Sätze erzeugt, muss man zusätzlich semantische Regeln festlegen.

Beispiel 2: Wohlgeformte Klammerterme

Wenn man in einem Rechenausdruck sämtliche Buchstaben und Zahlen weglässt, so erhält man Klammerterme der Form (); (()) oder (((()))). Ein solcher Term muß gleich viele öffnende wie schließende Klammern besitzen. Die Grammatikregeln zur Erzeugung wohlgeformter Klammerterme lauten:

$$(1) S \rightarrow () \quad (2) S \rightarrow (S) \quad (3) S \rightarrow SS$$

Mit (3) kann man Klammerterme aneinandereihehen, mit (2) eine rekursive Verschachtelung vornehmen und mit (1) schließlich den Erzeugungsvorgang abbrechen.

Die Klammerfolge (())(())() kann so abgeleitet werden:

$$S \Rightarrow SS \Rightarrow (S) S \Rightarrow (()) (S) \Rightarrow (()) (S S) \Rightarrow (()) (() S) \Rightarrow (()) (() ())$$

In den Grammatikregeln der zwei Beispiele kommen zwei Arten von Zeichen vor:

1. **Terminalzeichen**: endgültige Zeichen; das sind Zeichen, die zum Alphabet gehören; in Beispiel 2 sind es die Klammern.
2. **Nicht-Terminalzeichen**: vorläufige Zeichen; diese treten innerhalb einer Herleitung, aber nicht in einer endgültig abgeleiteten Zeichenfolge auf; im Beispiel 2 ist es das Zeichen S).

Formale Sprache:

Wenn X ein Alphabet, also eine endliche Menge von Zeichen ist, dann ist X^* die Menge aller endlichen Zeichenfolgen über X. Jede Teilmenge L von X^* heißt Formale Sprache.

Wenn G eine Grammatik mit X als Menge der Terminalzeichen ist, so ist $L(G)$ die Menge aller Zeichenfolgen über X, welche mit Hilfe der Ersetzungsregeln aus G erstellt werden können. $L(G)$ ist also die von G erzeugte formale Sprache.

Beispiel 3: Augewogene Bitmuster

Über den Terminalzeichen der Menge {0;1} wird eine Sprache betrachtet, bei der jedes Wort gleich viele Einsen wie Nullen enthalten muss. Die Worte 01, 10, 001011 oder 110010 gehören dazu. Als Nicht-Terminalzeichen dienen S, A und B, wobei S das Startzeichen ist. Dabei sollen aus A alle Wörter herleitbar sein, die genau eine Eins mehr als Nullen enthalten und aus B soll man alle Wörter erhalten, die genau eine 0 mehr als Einsen besitzen. Das führt zu folgenden Grammatikregeln:

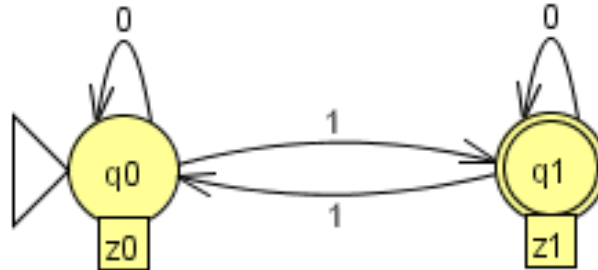
$$(1) S \rightarrow 0A \mid 1B \quad (2) A \rightarrow 1 \mid 1S \mid 0AA \quad (3) B \rightarrow 0 \mid 0S \mid 1BB$$

Das Wort 110010 kann dann wie folgt abgeleitet werden:

$$S \Rightarrow 1B \Rightarrow 11BB \Rightarrow 11B0 \Rightarrow 110S0 \Rightarrow 1100A0 \Rightarrow 110010$$

Reguläre Sprachen

Es soll zunächst eine Sprache entwickelt werden, die von einem Paritätsprüfer akzeptiert wird mit dem folgenden Zustandsgraph:



Dieser Automat erkennt Wörter, die eine ungerade Anzahl an Einsen haben, zwischen die beliebig viele Nullen eingestreut sind.

Die Menge der Terminalzeichen der zu suchenden Sprache ist $T = \{0,1\}$. Der Zustandsgraph des Paritätsprüfers hat zwei Zustände, den Startzustand z_0 und den Endzustand z_1 . Für den Startzustand z_0 wird nun der Buchstabe S und für den Endzustand z_1 wird A benutzt. Es sei damit $N = \{S,A\}$ die Menge der Nicht-Terminalzeichen.

Die zugehörige Grammatik lautet :

$$\begin{aligned} S &\rightarrow 1A \mid 0S \\ A &\rightarrow 0A \mid 1S \mid \emptyset \quad (\emptyset \text{ entspricht dem leeren Wort}) \end{aligned}$$

Eine Grammatik heisst **regulär**, wenn alle Produktionen von dieser Form sind:

$$A \rightarrow aB \quad \text{oder} \quad A \rightarrow a \quad \text{oder} \quad A \rightarrow \emptyset$$

Eine **Formale Sprache** heißt **regulär**, wenn sie eine reguläre Grammatik besitzt.

Die Sprache zum Paritätsprüfer ist demnach regulär und beinhaltet offenbar genau die vom Paritätsprüfer erkannten Worte.

Umgekehrt lässt sich auch zu jeder regulären Grammatik ein endlicher Automat angeben, der die von der Grammatik erzeugten Worte erkennt.

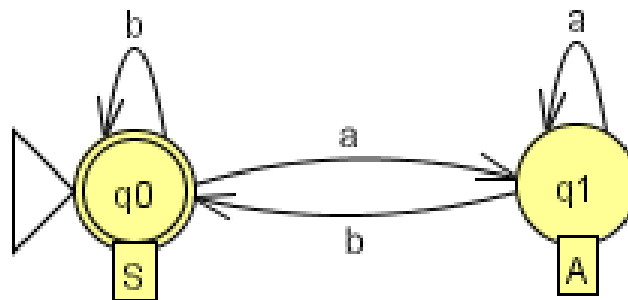
Jedem Nicht-Terminalzeichen der Grammatik wird genau ein Zustand des Automaten zugeordnet. Jeder Produktion entspricht eine Kante im Zustandsgraph. Diese Kante ist dann mit dem Terminalzeichen der zugehörigen Produktion beschriftet.

Somit gilt der allgemeine Satz :

Jede von einem endlichen Automaten akzeptierte Sprache ist regulär. Zu jeder regulären Sprache gibt es einen endlichen Automaten, der sie akzeptiert.

Weitere Beispiele:

Akzeptorb:

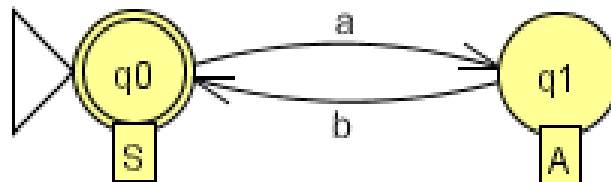


Der obige Automat akzeptiert Wörter, die nach folgender Grammatik erzeugt werden:

$$S \rightarrow aA \mid bS \mid \emptyset$$

$$A \rightarrow bS \mid aA$$

Akzeptor für $(ab)^n$:



Dieser Automat akzeptiert Worte, die mit folgender Grammatik erzeugt werden können:

$$S \rightarrow aA \mid \emptyset$$

$$A \rightarrow bS$$

Die gleiche Sprache könnte man auch durch die folgende nicht reguläre Grammatik erhalten:

$$S \rightarrow ab \mid SS .$$

Frage: Gibt es formale Sprachen, die von keinem endlichen Automaten erkannt werden können?

Es existieren in der Tat solche Grenzen endlicher Automaten.

Die Grenzen Endlicher Automaten

Da die Wörter regulärer Sprachen sehr einfach strukturiert sind, drängt sich die Frage auf, ob es noch andere als reguläre Sprachen gibt und ob es Formale Sprachen gibt, die nicht von einem Endlichen Automaten erkannt werden können.

Beispiel: Klammerpaare

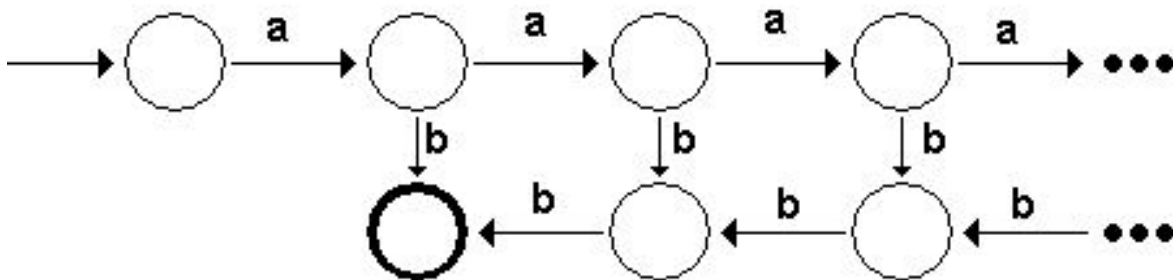
In korrekt gebildeten arithmetischen Termen treten öffnende und schließende Klammerausdrücke immer paarweise auf, d.h. in der Praxis ist die Anzahl der öffnenden Klammern gleich der der schließenden Klammern. Ähnlich verhält es sich mit Verbundanweisungen in Turbo-Pascal. Auch hier treten BEGIN...END oder REPEAT...UNTIL immer paarweise auf. Vergleiche dazu auch Beispiel 2 über wohlgeformte Klammerterme in Formale Sprachen.

Eine Maschine, die einen solchen Term auf Korrektheit überprüfen soll, muss sich also merken können, wieviele öffnende Klammern schon aufgetreten sind. Wenn man den öffnenden Klammerausdruck mit a und den schließenden Klammerausdruck mit b bezeichnet, so hat die Sprache folgende Gestalt:

$$L = \{ a^n b^n \mid n = 1, 2, 3, \dots \}.$$

In dieser Sprache beginnen alle Wörter mit einer beliebigen Anzahl von a 's, denen gleich viele b 's folgen.

Der Klammerautomat



Wenn man die Anzahl der a 's beschränkt, so kann der folgende Automat ein Wort der Sprache $L = \{ a^n b^n \mid 1 \leq n \leq N, N \text{ eine feste natürliche Zahl} \}$ erkennen:

Zu jedem N gibt es einen Automaten, der die Worte dieser Sprache erkennt. Die Anzahl der a 's ist aber in dieser Sprache beschränkt. Wenn die Anzahl der a 's aber nicht beschränkt ist, dann gibt es keinen Automaten, der alle Worte erkennt, da die Anzahl der Zustände des Automaten beschränkt ist. Dies leuchtet ein, da es sich um einen Endlichen Automaten handelt.

Um das Klammerproblem zu lösen, benötigt der Automat ein Gedächtnis. Er muß sich merken können, wieviele Klammern er schon geöffnet hat. Dieses Problem kann der Kellerautomat lösen.

Der Kellerautomat als Parser kontextfreier Sprachen

Die Leistungsfähigkeit Endlicher Automaten hat Grenzen:

Es fehlt ein **Gedächtnis**, um sich beliebig tief geschachtelte Klammern zu merken. Sprachen mit derartigen Klammerstrukturen sind **kontextfrei** und können demnach mit **Akzeptoren** nicht erkannt werden.

Beispiel:

Die kontextfreie Grammatik über dem Alphabet $\{ (,), P \}$ mit den Produktionsregeln $S \rightarrow S S \mid (S) \mid (P)$ erzeugt zum Beispiel folgende Worte:

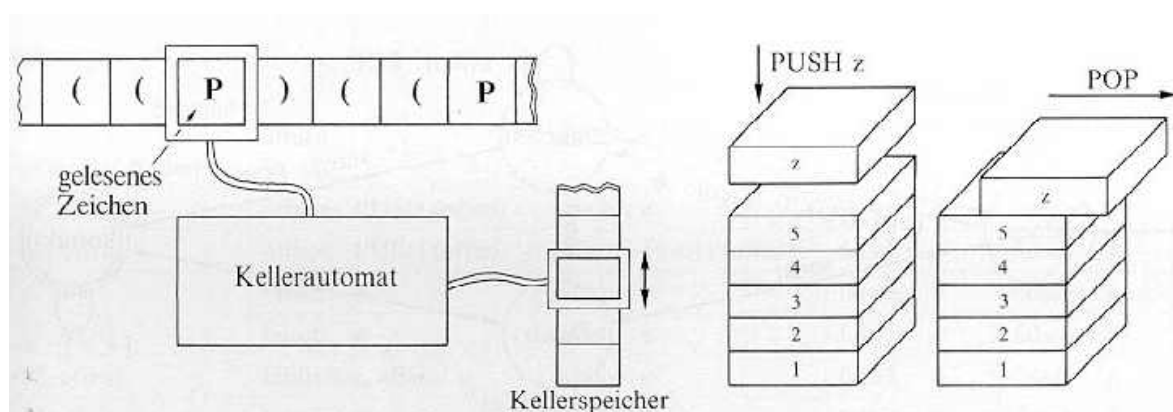
(P), ((P)(P)), (((P)(P)(P))(P)).

Aufgabe des Kellerautomaten ist es nun, die Anzahl der öffnenden und schließenden Klammern zu vergleichen. Dazu braucht er ein **Gedächtnis**. Als Gedächtnis dient dem Automat ein **Kellerspeicher** oder **Stapel (stack)**.

Die Arbeitsweise des Kellerautomaten

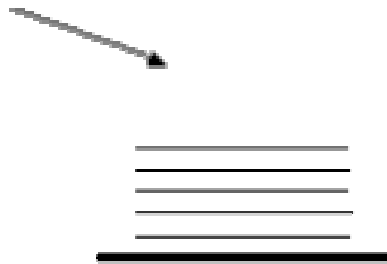
Der Kellerautomat besitzt wie die Akzeptoren endlich viele Zustände. Ausgehend vom Startzustand endet der Automat nach endlich vielen Schritten entweder im Endzustand oder im Falschzustand.

In Abhängigkeit vom Eingabezeichen, vom aktuellen Zustand und vom obersten Kellersymbol wird ein Zustandswechsel ausgelöst und die Kellerspitze verändert. Im Keller können beliebig viele Symbole abgelegt werden.

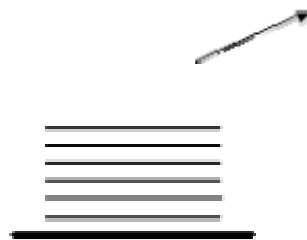


PUSH z	legt das Zeichen z zuoberst in den Keller
POP	entfernt das oberste Kellerzeichen
#	ändert den Keller nicht

Es gilt somit das Prinzip: Last-In , First-Out .

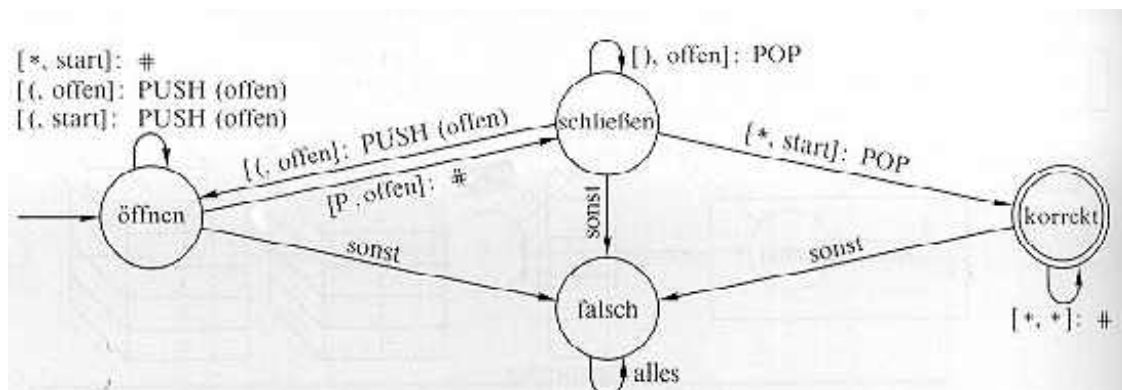


Das neue Zeichen wird zuoberst in den Speicher gelegt. Es können in einem Schritt auch mehrere Zeichen hinterlegt werden.



Das zuletzt hinterlegte Zeichen wird wieder zuerst entfernt.

Ein Kellerautomat zum [Eingangsbeispiel](#):



Man erhält für die Zeichenkette ((P)((P))) folgende Parsingtabelle :

Reststring	Zustand	Inhalt des Kellerspeichers
((P)((P)))	öffnen	start
(P)((P)))*	öffnen	start, offen
P)((P)))*	öffnen	start, offen, offen
)((P)))*	schließen	start, offen, offen
((P)))*	schließen	start, offen
(P)))*	öffnen	start, offen, offen
P)))*	öffnen	start, offen, offen, offen
)))*	schließen	start, offen, offen, offen
))*	schließen	start, offen, offen
)*	schließen	start, offen
*	schließen	start
	korrekt	*

Das Zeichen * wird in der Tabelle in zwei Bedeutungen benutzt:

- 1) der Keller ist leer
- 2) es erfolgt keine Eingabe.

Es sollen jetzt Parsingtabelle zu

a : (((P)(P)((P))))

b : ((P)(P)P((P)(P)))

erstellt werden!

(s = start, o = offen)

zu a :

$((((P)(P)((P))))^*$		öffnen	s
$((P)(P)((P)))^*$		öffnen	so
$(P)(P)((P))^*$		öffnen	soo
$P)(P)((P))^*$		öffnen	sooo
$)P)((P))^*$		schließen	sooo
$(P)((P))^*$		schließen	soo
$P)((P))^*$		öffnen	sooo
$)((P))^*$		schließen	sooo
$((P))^*$		schließen	soo
$(P))^*$		öffnen	sooo
$P))^*$		öffnen	soooo
$))^*$		schließen	soooo
$)^*$		schließen	sooo
$)^*$		schließen	soo
$)^*$		schließen	so
*		schließen	s
		korrekt	*

Der Term ist korrekt.

zu b :

$((P)(P)P((P)(P)))^*$		öffnen	s
$(P)(P)P((P)(P))^*$		öffnen	so
$P)(P)P((P)(P))^*$		öffnen	soo
$)P)P((P)(P))^*$		schließen	soo
$(P)P((P)(P))^*$		schließen	so
$P)P((P)(P))^*$		öffnen	soo
$)P((P)(P))^*$		schließen	soo
$P((P)(P))^*$		schließen	so
$((P)(P))^*$		falsch	so
$(P)(P))^*$		falsch	so

Der Fehlerzustand kann nicht verlassen werden. Der Term ist falsch.

Der Zustandsgraph ist in Pascal implementiert:

```
program kellerautomat;
uses crt;
const kg = 50;
type ka = (o,s,l);
      zm = (oe, sl, f, k);
var keller: ARRAY [0..kg] of ka;
    t: 0..kg;
    z: zm;
    ez: 1..80;
    wort,zeichen: string;

procedure push(zei:ka);
begin
  if t < kg then begin t:= t+1; keller[t] := ze; end;
end;

procedure pop;
begin
  if t > 0 then begin keller[t] := l; t := t-1; end;
end;

function uebergang (e: char; kz: ka; z:zm): zm;
begin
  uebergang:=f;
  Case z Of
    oe: begin
      if e = '(' Then begin uebergang := z; push(o);end;
      if (e='p') AND (kz=o) THEN uebergang := sl;
      end;

    sl: begin
      if (e=')') and (kz=o)
      then begin uebergang := z; pop; end;
      if (e='(') and (kz=o)
      then begin uebergang := oe; push(o); end;
      if (e='*') and (kz=s)
      then begin uebergang := k; pop; end;
      end;

    k: if (e='*') and (kz=l)
      then uebergang := z else uebergang := f;
    f: uebergang := z;
  end;
end;

Begin
  textbackground(black);
  clrscr;
  write(',Layout Gangolf Kern');

  window(10,3,70,23);
  textbackground(blue);textcolor(white); clrscr;
  repeat
  clrscr;
  writeln ('Bitte geben Sie das Wort ein');
  readln (wort);
  writeln;
  for t:= kg downto 1 do keller[t] := l;
  keller[t] := s;
  z:= oe;
  for ez:= 1 to length (wort) do
  z:= uebergang (wort[ez], keller[t], z);
  z:= uebergang ('*', keller[t], z);
  if z = k then writeln('Klammerterm korrekt')
    else writeln('Klammerterm nicht korrekt');
    writeln;
  writeln('Weiter mit <enter>, beenden mit <e>!');readln(zeichen)
  until zeichen='e'
End.
```

Es gibt Sprachen, die mit dem Kellerautomaten nicht erkannt werden können.

Beispiel: $L = \{ a^n b^n c^n \mid n \text{ eine natürliche Zahl} \}$
(sprich: **a hoch n**)

Das Wort $w = \mathbf{aaabbbccc}$ ist dann aus der Sprache L . Der Kellerautomat könnte dieses Wort nicht verarbeiten, denn wenn er die Zeichenkette \mathbf{aaa} in den Speicher gelegt hat, würde er mit \mathbf{bbb} diese wieder abarbeiten. Die Zeichenkette \mathbf{ccc} würden dann aber im Keller unbearbeitet liegen bleiben, denn es gäbe keine Eingabezeichen mehr, mit denen man den Keller leeren und den Automat wieder in den Zustand **korrekt** überführen könnte.

Diese Sprache wird also nicht erkannt.

Es lässt sich zeigen, dass die durch den Kellerautomaten erkannten Sprachen gerade die kontextfreien Sprachen sind.

Kontextfreie Sprachen

Kontextfreie Sprachen sind ausdrucksfähiger als **reguläre Sprachen**. Bei ihren Grammatiken besteht die linke Seite einer Ersetzungsregel ebenfalls nur aus einem einzigen Nicht-Terminalzeichen. Es besteht jedoch keine Einschränkung hinsichtlich der rechten Seite. Die im Abschnitt über Formale Sprachen angegebenen Beispiele von Grammatiken sind durchweg kontextfrei. Diese Sprachen haben damit grundsätzliche Bedeutung zur Beschreibung natürlicher und künstlicher Sprachen wie Programmiersprachen.

Bei der Übersetzung von Programmiersprachen arbeiten Parser entweder aufsteigend (bottom-up) von den Terminalzeichen bis zum Startzeichen oder absteigend (top-down) vom Startzeichen bis zu den Terminalzeichen.

Die grammatikalische Struktur von Ausdrücken lässt sich durch Syntaxbäume veranschaulichen.

Beispiel: Arithmetische Terme

Terminalzeichen: $\{ +, -, *, /, (,), a, b, \dots, z \}$

Nicht-Terminalzeichen: $\{ T, S, F, V \}$

Startzeichen: T .

Die Produktionsregeln: (G 1)

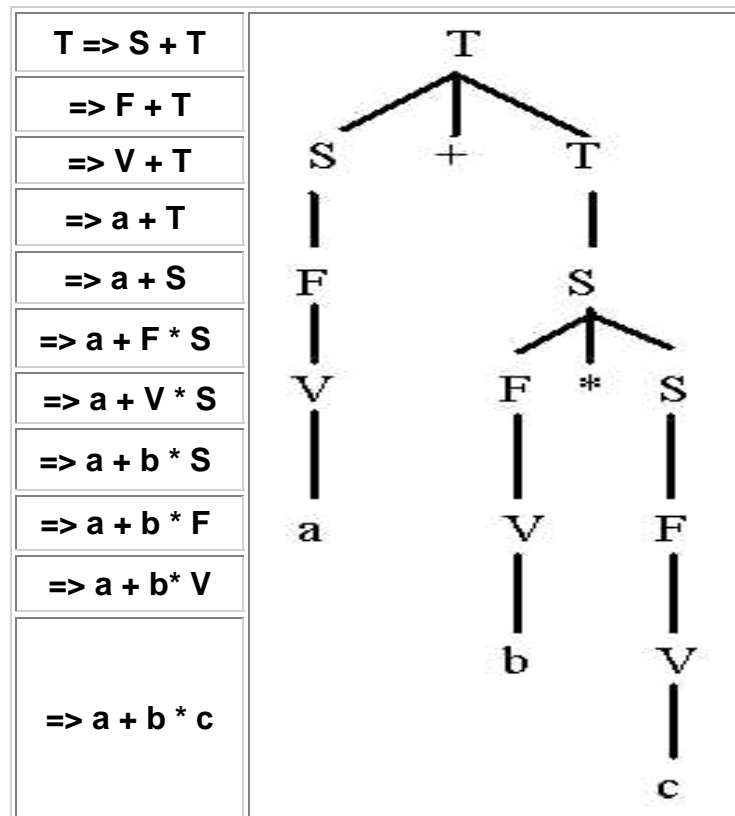
(1) $T \rightarrow S \mid S + T \mid S - T$

(2) $S \rightarrow F \mid F * S \mid F / S$

(3) $F \rightarrow V \mid (T)$

(4) $V \rightarrow a \mid b \mid \dots \mid z$

Der arithmetische Term $a + b + c$ lässt sich nun top-down als Linksableitung herleiten. Das am weitesten links stehende Nicht - Terminalzeichen wird in einer Linksableitung zuerst ersetzt :



Bei einer Rechtsableitung wird das am weitesten rechts stehende Nicht - Terminalzeichen zuerst ersetzt:

$$T \Rightarrow S + T \Rightarrow S + S \Rightarrow S + F * S \Rightarrow S + F * F \Rightarrow S + F * V \Rightarrow S + F * c \\ \Rightarrow S + V * c \Rightarrow S + b * c \Rightarrow F + b * c \Rightarrow V + b * c \Rightarrow a + b * c$$

Liest man diese Ableitungsfolge in umgekehrter Reihenfolge, also von den Terminalzeichen ausgehend, erhält man eine Bottom-up-Zerlegung des Ausdrucks.

Links- wie Rechtsableitung führen zum gleichen Syntaxbaum!

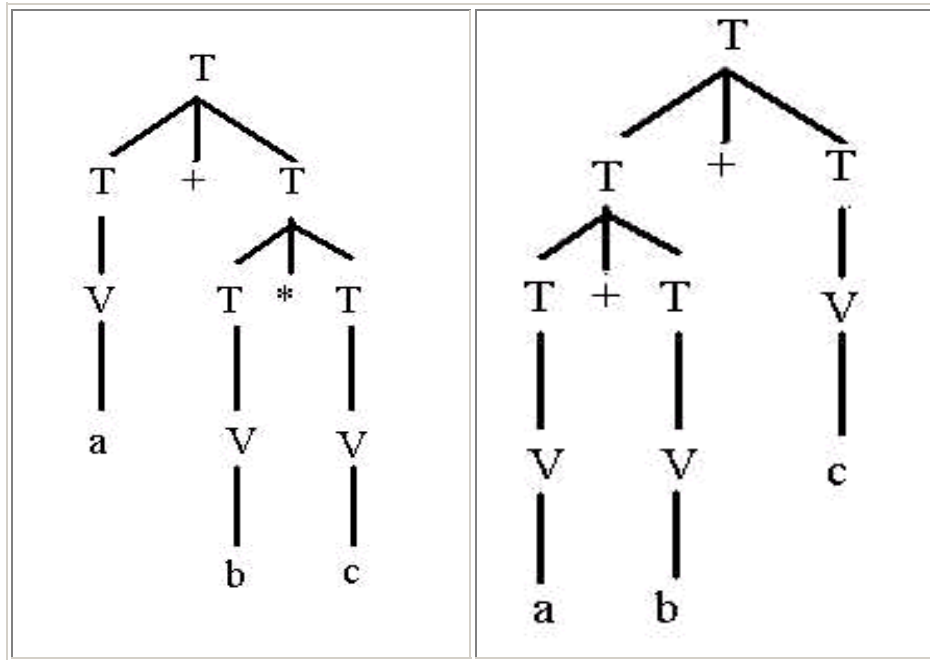
Mehrdeutige Grammatiken

Die Produktionsregeln der Grammatik (G 1) aus dem Beispiel werden durch die folgenden Regeln ersetzt:

$$(G 2) \quad (1) T \rightarrow V \mid (T) \mid T + T \mid T * T \quad (2) V \rightarrow a \mid b \mid c \mid \dots \mid z$$

Man kann zeigen, dass (G 1) äquivalent zu (G 2) ist, d.h. beide Grammatiken erzeugen die gleiche Sprache. (G 2) ist offenbar einfacher. Der Ausdruck $a + b * c$ lässt sich aber auf zwei verschiedene Weisen als Linksableitung herstellen:

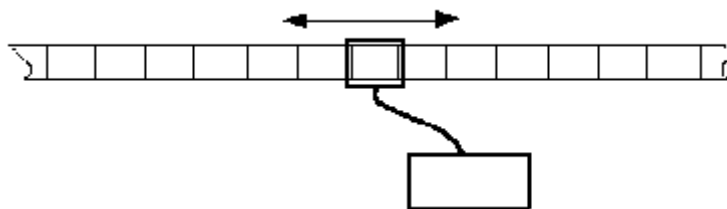
Beginnt man mit $T \Rightarrow T + T$, erhält man den linken Baum. Fängt man mit $T \Rightarrow T * T$ an, so bekommt man den rechten Baum .



Die Bäume stimmen in ihrer Struktur nicht überein! Eine solche Grammatik wie (G 2) bezeichnet man als mehrdeutig. Einsetzen von Zahlenwerten in die Variablen führen entsprechend zu unterschiedlichen Resultaten.

Turingmaschinen

Die Steuereinheit der Turingmaschine besteht wie bei den Kellerautomaten aus endlich vielen Zuständen. Der Kellerspeicher wird ersetzt durch ein beiseitig unbegrenzt Arbeitsband, welches als Eingabe-, Ausgabe- und Speichermedium benutzt wird. Das Arbeitsband ist unterteilt in Felder, die jeweils ein Zeichen eines Bandalphabets aufnehmen können. Der Lese-Schreib-Kopf (LS-Kopf) kann sich beliebig über das Band bewegen und ein Feld lesen oder neu beschreiben.



Arbeitsweise der Turingmaschine:

- ein Symbol auf dem Arbeitsfeld wird gelesen,
- abhängig vom *gelesenen Symbol und dem aktuellen Zustand*
 - 1) schreibt der LS-Kopf ein Symbol auf das Arbeitsfeld,
 - 2) läuft der LS-Kopf entweder ein Feld nach links oder rechts oder verharrt auf dem Feld,
 - 3) geht die Maschine in einen neuen Zustand über.

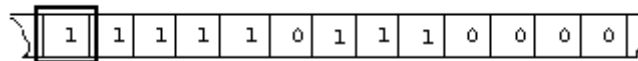
Turingmaschinen wurden 1936 eingeführt von dem Namensgeber Alan Turing zum Zweck der **Präzisierung des Begriffs der Berechnung**. Es wurde damals mit Nachdruck die Frage gestellt, ob denn wirklich alle denkbaren irgendwie formalisierbaren Probleme durch *Rechnen* lösbar wären, wie noch der Philosoph Leibniz im 18. Jahrhundert behauptete. Schnell einsehbare einfache Beispiele von Turingberechnungen sind etwa die *Addition zweier natürlicher Zahlen* oder das *Verdoppeln einer natürlichen Zahl*.

Beispiele von Turingberechnungen

Wir wollen eine sehr einfache Bandbeschriftung wählen: Fast das gesamte Band ist mit *Leerzeichen* (bei uns die 0) beschrieben. Natürliche Zahlen werden durch entsprechende Anzahl von 1'en dargestellt.

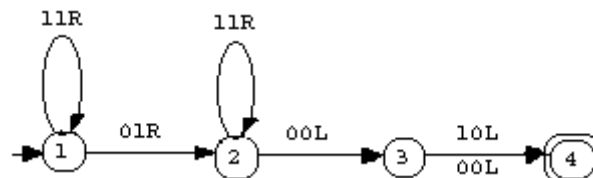
Addition zweier natürlicher Zahlen: 5 + 3

Bandbeschriftung zu Beginn:



Verfahrensidee:

Der LS-Kopf läuft solange nach rechts bis die erste 0 (sie trennt die beiden Summanden !) gefunden ist, setzt hier eine 1 und löscht die am weitesten rechts stehende 1. Das Ergebnis sind wie gewünscht acht hintereinander stehende 1'en.



Zustandsgraph der Turingmaschine

Zur Notation der Kantenbeschriftung: (11R z.B. am Zustand 2) Liest der LS-Kopf das Symbol 1, wird eine 1 geschrieben (der Inhalt des aktuellen Feldes ändert sich also nicht) und der LS-Kopf geht einen Schritt nach rechts, die Turingmaschine bleibt im Zustand 2. (L entspricht einem Linksschritt)

Der Zustandsgraph kann unmittelbar in eine Turingtabelle, oft als *Turingprogramm* bezeichnet, überführt werden:

Zustand	gelesenes Symbol	neues Symbol	Richtung LS-Kopf	neuer Zustand
1	1	1	R	1
1	0	1	R	2
2	1	1	R	2
2	0	0	L	3
3	1	0	L	4
3	0	0	L	4

Einfache Turingprogramme sind für folgende Aufgaben entwickelbar:

- 1) Verdoppeln einer natürlichen Zahl
- 2) Kopie einer Folge von 1'en.
- 3) Beschreiben des Turingbandes mit unendlich vielen 1'en.
- 4) Alle Worte der Sprache $L = \{ a^n b^n c^n / n > 0 \}$

Präzisierung des Begriffs der Berechnung und unlösbare Probleme

Im Bereich der formalen Sprachen stellt sich häufig das folgende Wortproblem:

Gibt es ein Verfahren, welches für jedes Wort w über dem Alphabet von L entscheidet, ob w aus L ist oder nicht!

In diesem Sinn sind etliche Wortprobleme entscheidbar. Jede kontextfreie Sprache ist entscheidbar und damit natürlich auch jede reguläre Sprache. Insbesondere sind somit unsere Programmiersprachen entscheidbar und die korrekte Arbeit von Parser-Programmen bei der Syntaxanalyse glücklicherweise gesichert! Eine jede solche Entscheidbarkeit setzt ein entsprechendes Verfahren voraus und die Sicherheit, was denn als Verfahrensweise (**Algorithmus**) anerkannt werden soll.

Der intuitive Algorithmusbegriff schreibt eine endliche Folge von Handlungsanweisungen vor, die von jedem nachvollziehbar ist. Man kann sich dabei auch ein mechanisch ausführbares Verfahren vorstellen, zum Beispiel eine bestimmte Art auf dem Abacus die Perlen zu bewegen um zielgerichtet eine Addition oder Multiplikation durchzuführen. Um prinzipielle Grenzen auszumachen, was alles entscheidbar oder ganz allgemein berechenbar ist, reicht der zu vage gehaltene intuitive Algorithmusbegriff nicht aus. Eine Möglichkeit der Präzisierung stellt gerade die Turingmaschine dar.

Man ist seit langem der Überzeugung:

Alles, was intuitiv berechenbar ist, ist auch auf einer Turingmaschine berechenbar (Churchs These)!

Man kann beweisen:

Alles, was in so mächtigen Programmiersprachen wie Pascal oder C++ programmiert werden kann, kann auch auf Turingmaschinen programmiert werden und umgekehrt!

Mit Hilfe von Turingmaschinen erweisen sich nun einige (nicht gerade bescheiden gestellte) Probleme als unentscheidbar und damit generell auch auf besten zukünftigen Computern algorithmisch unlösbar:

- a) **Das Wortproblem für beliebige Sprachen L**
- b) **Ist eine Aussage in einer mathematischen Theorie wahr oder falsch?**
Das Beweisen oder Widerlegen von mathematischen Sätzen ist im allgemeinen nicht mechanisierbar. Für eingeschränkte mathematische Gebiete gibt es allerdings durchaus sehr leistungsfähige Theorem-Beweis-Programme.
- c) **Das Halteproblem: Gibt es ein Verfahren, welches zu einem gegebenen Programm und einer Eingabe entscheidet, ob das Programm stoppt?**

Das Halteproblem

Hier soll eine Version des Halteproblems für Pascalprogramme dargestellt werden: Sei p ein Pascalprogramm und x die Zeichenfolge der Eingabe.

Frage: Gibt es eine *boolesche Funktion*, die entscheidet, ob p angewandt auf x hält?

Bei positiver Antwort hätte man damit einen sehr mächtigen Interpreter von Pascalprogrammen, der das Terminieren von Programmen entscheidet! Nimmt man p in Form einer Textdatei als weitere Eingabe, so erhielte die gesuchte Funktion im Prinzip die Gestalt :

```
FUNCTION
haelt(p,x:TEXT):boolean;
BEGIN
  IF <p terminiert bei x >
  THEN haelt:= TRUE
  ELSE haelt:= FALSE
END;
```

Die ganze Problematik einer solchen Funktion steckt natürlich in der nicht weiter ausgeführten Abbruchbedingung.

Das folgende Programm verhält sich nun ausgesprochen seltsam, falls für x das Programm p selbst eingegeben wird:

```
PROGRAM seltsam;  
FUNCTION haelt...;  
BEGIN  
  lies( $p$ );  
  WHILE haelt( $p,p$ ) do;  
    WRITELN('Fertig')  
END.
```

Fall 1: *seltsam* stoppt, somit wird *haelt* = TRUE , also kommt *seltsam* in eine Endlosschleife, womit *seltsam* nicht stoppt!

Fall 2: *seltsam* stoppt nicht, somit wird *haelt* = FALSE, also schreibt *seltsam* 'Fertig', womit *seltsam* stoppt!

In beiden Fällen entsteht ein Widerspruch: Das Programm *seltsam* stoppt genau dann, wenn es nicht stoppt!

Dies ist der Beweis zum Satz von Turing (ursprünglich für Universelle Turingmaschinen formuliert): Es gibt kein Pascalprogramm, das für ein beliebiges Pascalprogramm p und Eingabe x entscheidet, ob p angewandt auf x nach endlichen vielen Schritten hält.