Theoretische Informatik 3. Semester

Prof. Dr. Martin Plümicke

6. Dezember 2023

Inhaltsverzeichnis

Vorwort

Das vorliegende Skript ist im Laufe der Jahre 2000 - 2018 an der Berufsakademie Stuttgart, Außenstelle Horb/DHBW Stuttgart Campus Horb entstanden.

Ein besonderer Dank gilt den Studierenden Sybille Paul (IT_2001), Marcel Ammann (IT_2004), Peter Würth (INF_2017) und Nicolas Haug (INF_2019), die das Skript getippt und Korrektur gelesen haben.

1 Berechenbarkeit und rekursive Funktionen

Wir werden uns nun mit den Möglichkeiten eines Computers beschäftigen. Dabei werden wir sehen, dass alle Computer grundsätzlich die gleichen Fähigkeiten haben. Sie unterscheiden sich nur in der Geschwindigkeit und bei den möglichen Ein- und Ausgabearten. Berechnungen sind als partielle Funktionen $f : \{0,1\}^n \dashrightarrow \{0,1\}^m$, darstellbar. Wenn man die Zahlen vom 2er–System ins 10er–System umrechnet, so erhält man eine Funktion

$$f : \mathbb{N}^{n'} \dashrightarrow \mathbb{N}^{m'}.$$

Beispiel. Man kann eine Funktion

$$add : \{0,1\}^{16} \to \{0,1\}^8$$

betrachten als

$$add : \{0,1\}^8 \times \{0,1\}^8 \to \{0,1\}^8.$$

Wenn man der Berechnung jeweils 8-Bit Zahlen zu Grunde legt, so kann man die Funktion als

$$add : \mathbb{N}^2 \to \mathbb{N}$$

ansehen.

Wir werden nun Schritt für Schritt die Menge aller von Computern berechenbaren Funktionen herleiten.

1.1 Primitive rekursive Funktionen

Definition 1.1 (Grundfunktionen)**.** Die Menge der **Grundfunktionen** enthält folgende totale Funktionen.

1. $C_0^0 :\to \mathbb{N}$ mit $C_0^0 = 0$ **(Nullstellige Nullfunktion)**
2. $C_0^1 : \mathbb{N} \to \mathbb{N}$ mit $C_0^1(n) = 0$ **(einstellige Nullfunktion)**
3. $\Pi_i^k : \mathbb{N}^k \to \mathbb{N}$ mit $\Pi_i^k(n_1, \ldots, n_k) = n_i$ $(i \leq k)$ **(Projektionsfunktionen)**
4. $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = n + 1$ **(Nachfolger)**

Beispiel. Folgende Beispiele veranschaulichen die abstrakten Definitionen

- $\Pi_1^2(7, 22) = 7$
- $\Pi_3^5(1, 2, 4, 3, 5) = 4$
- $s(\,3\,) = 4$

Definition 1.2 (Komposition)**.** Seien $r \in \mathbb{N}_0$ und $s \in \mathbb{N} \setminus \{0\}$ und seien $g : \mathbb{N}^s \dashrightarrow \mathbb{N}$, sowie $h_i : \mathbb{N}^r \dashrightarrow \mathbb{N}$ für $1 \leq i \leq s$ partielle Funktionen, so heißt $f : \mathbb{N}^r \dashrightarrow \mathbb{N}$ mit

$$f(n_1, \ldots, n_r) = \begin{cases} g(h_1(n_1, \ldots, n_r), \ldots, h_s(n_1, \ldots, n_r)) & \text{falls definiert} \\ \perp \text{ (undefiniert)} & \text{sonst} \end{cases}$$

die **Komposition** von g und $h_1, \ldots, h_s$.

Beispiel. Seien folgende Beispiele gegeben:

1. r = 1, s = 2
$$\left.\begin{array}{l} g(n_1, n_2) = \Pi_1^2(n_1, n_2) \\ h_1(n) = s(n) \\ h_2(n) = C_0^1(n) \end{array}\right\} f(n) = g(h_1(n), h_2(n)) = \Pi_1^2(s(n), C_0^1(n)) = s(n)$$
2. r = 1, s = 1
$$\left.\begin{array}{l} g(n) = s(n) \\ h_1(n) = C_0^1(n) \end{array}\right\} f(n) = g(h_1(n)) = s(C_0^1(n)) = s(0) = 1$$

Definition 1.3 (Primitives Rekursionsschema)**.** Seien $k \in \mathbb{N}_0$, $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ eine totale Funktion, dann ist durch

1. $f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$
2. $f(x_1, \ldots, x_k, s(y)) = h(x_1, \ldots, x_k, y, f(x_1, \ldots, x_k, y))$

das zu g und h gehörige primitive Rekursionsschema gegeben. Man sagt f erfüllt das primitive Rekursionsschema, wenn f den Gleichungen 1 und 2 genügt. Das heißt, wir haben ein Gleichungssystem mit einer unbekannten Funktion f.

Herleitung der Funktion, die dem primitiven Rekursionsschema genügt:

Wir leiten nun die Funktion Schritt für Schritt her. Beginnend an der Stelle 0 wird in jedem Schritt eine Funktion bestimmt, die für eine weitere Zahl der gesuchten Funktion entspricht.

An der Stelle 0 ist $f(\, x_1, \ldots, x_k, 0\,)$ durch $g(\, x_1, \ldots, x_k\,)$ bestimmt. Wir nennen die Funktion, die an der Stelle 0 der Funktion f entspricht und allen anderen Stellen undefiniert ist f_0:

$$f_0(x_1, \ldots, x_k, y) = \begin{cases} g(x_1, \ldots, x_k) \text{ für } y = 0 \\ \perp \text{ sonst} \end{cases}$$

Da die Funktion f an der Stelle 1 durch die Funktion h und durch f an der Stelle 0 definiert ist, lässt sich die Funktion f_1, die an den Stellen 0 und 1 der Funktion f entspricht und sonst undefiniert ist, wie folgt darstellen:

$$f_1(x_1, \ldots, x_k, y) = \begin{cases} f_0(x_1, \ldots, x_k, y) \text{ für } y < 1 \\ h(x_1, \ldots, x_k, y-1, f_0(x_1, \ldots, x_k, y-1)) \text{ für } y = 1 \\ \perp \text{ sonst} \end{cases}$$

Analog folgt f_2:

$$f_2(x_1, \ldots, x_k, y) = \begin{cases} f_1(x_1, \ldots, x_k, y) \text{ für } y < 2 \\ h(x_1, \ldots, x_k, y-1, f_1(x_1, \ldots, x_k, y-1)) \text{ für } y = 2 \\ \perp \text{ sonst} \end{cases}$$

Sei weiterhin:

$$f_n(x_1, \ldots, x_k, y) = \begin{cases} f_{n-1}(x_1, \ldots, x_k, y) \text{ für } y < n \\ h(x_1, \ldots, x_k, y-1, f_{n-1}(x_1, \ldots, x_k, y-1)) \text{ für } y = n \\ \perp \text{ sonst} \end{cases}$$

Die Funktion f_n erfüllt das primitive Rekursionsschema für alle $y \leq n$

Die Lösung für das primitive Rekursionsschema ist dann:

$$f = \lim_{n \to \infty} f_n$$

Beispiel. Sei

$$g : \mathbb{N} \to \mathbb{N} \text{ mit } g(x) = x = \Pi_1^1(x)$$

und $h : \mathbb{N}^3 \to \mathbb{N}$ mit

$$h(x_1, x_2, x_3) = x_3 + 1 = s(\, \Pi_3^3(x_1, x_2, x_3)\,).$$

Daraus ergibt sich folgendes primitives Rekursionsschema:

$f(x,0) = g(x) = x$
$f(x,s(y)) = h(x,y,f(x,y)) = f(x,y) + 1$

Herleitung der Lösung:

$$f_0(x,y) = \begin{cases} x, & y = 0 \\ \bot, & \text{sonst} \end{cases}$$

$$f_1(x,y) = \begin{cases} f_0(x,0) = x, & y = 0 \\ f_0(x,0) + 1 = x + 1, & y = 1 \\ \bot, & \text{sonst} \end{cases}$$

$$f_2(x,y) = \begin{cases} f_1(x,0) = x, & y = 0 \\ f_1(x,0) + 1 = x + 1, & y = 1 \\ f_1(x,1) + 1 = x + 2, & y = 2 \\ \bot, & \text{sonst} \end{cases}$$

$$f_3(x,y) = \begin{cases} f_2(x,0) = x, & y = 0 \\ f_2(x,0) + 1 = x + 1, & y = 1 \\ f_2(x,1) + 1 = x + 2, & y = 2 \\ f_2(x,2) + 1 = x + 3, & y = 3 \\ \bot, & \text{sonst} \end{cases}$$

Als Lösung ergibt sich dann:

$$f(x,y) = \lim_{n \to \infty} f_n(x,y) = x + y$$

Das primitive Rekursionsschema kann man direkt in Programmiersprachen übernehmen. Die Additionsfunktion lässt sich beispielsweise in Haskell implementieren:

```
add:: (Int, Int) -> Int

add(x, 0) = x
add(x, n) = add(x, n - 1) + 1
```

Auch in Java lässt sich die Additionsfunktion rekursiv implementieren:

```
class Add {
   int add(int x, int y) {
       if(y == 0) return x;
       else if (y > 0) return (add(x,y-1)+1);
   }
```

}

Satz 1.4. Seien $k \in \mathbb{N}\setminus\{0\}, g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ als totale Funktionen gegeben, so existiert genau eine totale Funktion: $f : \mathbb{N}^{k+1} \to \mathbb{N}$, die das primitive Rekursionsschema erfüllt.

Definition 1.5 (Primitiv rekursive Funktionen)**.** Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **primitiv rekursiv**, wenn sie entweder eine Grundfunktion ist, oder aus diesen in endlich vielen Schritten durch Komposition und durch das primitive Rekursionsschema gebildet wurde.

Satz 1.6. Jede primitive rekursive Funktion ist total.

Satz 1.7. Folgende Funktionen sind primitiv rekursiv:

1. $f(x, y) = x + y$
2. $f_k(x) = k \cdot x$, für alle $k \in \mathbb{N}$
3. $f(x, y) = x \cdot y$
4. $f(x, y) = x^y$
5. $f(x, y) = x^{x^{x^{\cdots}}}$ (y - mal)
6. $f(x, y) = x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & \text{sonst} \end{cases}$
7. $f(x, y) = |x - y|$

1.2 LOOP–Programme

Nun geben wir eine kleine Programmiersprache an, mit der man genau die primitiv rekursiven Funktionen programmieren kann.

Definition 1.8 (LOOP–Programme)**.** Die Programmiersprache LOOP umfasst:

- Variablen:
  $x_0, x_1, \ldots; y_1, y_2, \ldots; z_1, z_2, \ldots$
- Konstanten;
  $0, 1, 2, 3, 4, 5, \ldots$

- Trennsymbole:
  ";" und "="
- Operationszeichen:
  "+" und "-"
- Schlüsselwörter:
  `LOOP`, `DO`, `END`

LOOP–Programme werden wie folgt definiert:

1. $x_i = x_j + c$ ist ein LOOP–Programm.
2. $x_i = x_j - c$ ist ein LOOP–Programm.
3. Seien P_1 und P_2 LOOP–Programme, dann ist auch

   $$P_1; P_2$$

   ein LOOP–Programm.
4. Falls P ein LOOP–Programm ist, so ist auch

   $$\texttt{LOOP}\ x_i\ \texttt{DO}\ P\ \texttt{END}$$

   ein LOOP–Programm.

- Die Bedeutung (Semantik) eines LOOP–Programms P ist dadurch gegeben, dass P eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ berechnen soll. Zunächst gilt für den Aufruf $f(n_1, \ldots, n_k)$, dass die Variablen $x_1 = n_1, \ldots, x_k = n_k$ enthalten. $x_i = 0$ für $i > k$
- Die Wertzuweisungen haben die Bedeutung wie in Java / C.
  Ausnahme: Für $x_i = x_j - c$ und $x_j < c$ gilt $x_i = 0$
- $P_1; P_2$ heißt: Führe zunächst P_1 und dann P_2 aus.
- Die `LOOP`–Schleife hat folgende Bedeutung: `LOOP` x_i `DO` P `END` heißt, P wird sooft ausgeführt wie in x_i zu Beginn der LOOP–Schleife steht. Änderungen von x_i während der Schleife haben keine Auswirkung.
- Das Ergebnis steht am Ende in x_0

Anmerkung. Es ist offensichtlich, dass keine Endlosschleifen mit LOOP programmiert werden können.

Lemma 1.9. Die Grundfunktionen (Def. 1.1) lassen sich durch LOOP–Programme programmieren.

Beweis.

1. $C_0^0 :\to \mathbb{N}$ mit $C_0^0 = 0$
   LOOP–Programm: $x_0 = 0$
2. $C_0^1 : \mathbb{N} \to \mathbb{N}$ mit $C_0^1(x) = 0$
   LOOP–Programm: $x_0 = 0$
3. $\Pi_i^k : \mathbb{N}^k \to \mathbb{N}$ mit $\Pi_i^k(n_1, \ldots, n_k) = n_i$
   LOOP–Programm: $x_0 = x_i$
4. $s : \mathbb{N} \to \mathbb{N}$ mit $s(n) = n + 1$
   LOOP–Programm: $x_0 = x_1 + 1$

□

Der nächste Schritt wäre die Überlegung, ob man die Komposition von gegebenen Funktionen ebenfalls durch LOOP–Programme programmieren kann.

Lemma 1.10. Seien $h_i : \mathbb{N}^k \to \mathbb{N}$ für $1 \leq i \leq s$ primitiv rekursive Funktionen, die durch LOOP–Programme $P_1, \ldots, P_s$ programmiert sind. Sei weiter: $g : \mathbb{N}^s \to \mathbb{N}$ eine primitiv rekursive Funktion, die durch das LOOP–Programm P_g programmiert ist. Dann gibt es ein LOOP–Programm P_f welches $f : \mathbb{N}^k \to \mathbb{N}$ mit $f(x_1, \ldots, x_k) = g(h(x_1, \ldots, x_k), \ldots, h_s(x_1, \ldots, x_k))$ programmiert.

Beweis. Es wird jetzt das LOOP–Programm P_f angegeben:

$z_1 = x_1; z_2 = x_2; \ldots; z_k = x_k$;//Eingaben werden zwischengespeichert
P_1;
$y_1 = x_0$; //Ergebnis von P_1 wird zwischengespeichert
$x_1 = z_1; \ldots; x_k = z_k$;
P_2;
$y_2 = x_0$; //Ergebnis von P_2 wird zwischengespeichert
$x_1 = z_1; \ldots; x_k = z_k$;
P_3;
$y_3 = x_0$; //Ergebnis von P_3 wird zwischengespeichert
...
$x_1 = z_1; \ldots; x_k = z_k$;
P_s;

$y_s = x_0$;
$x_1 = y_1; x_2 = y_2; \ldots; x_s = y_s$;
P_g □

Beispiel. (vgl. Beispiel 1.1) Seien $g : \mathbb{N} \to \mathbb{N}$ mit $g(x) = s(x)$ und $h_1 : \mathbb{N} \to \mathbb{N}$ mit $h_1(x) = C_0^1(x)$ gegeben. Seien weiterhin die zugehörigen LOOP–Programme gegeben durch:

$$P_g \stackrel{\text{def}}{=} x_0 = x_1 + 1 \qquad \text{und} \qquad P_1 \stackrel{\text{def}}{=} x_0 = 0.$$

Dann ergibt sich für P_f:

$z_1 = x_1$;
$x_0 = 0$; //Programm P_1
$y_1 = x_0$;
$x_1 = y_1$;
$x_0 = x_1 + 1$ //Programm P_g

Lemma 1.11. Seien für $k \geq 0$, $g : \mathbb{N}^k \to \mathbb{N}$ und $h : \mathbb{N}^{k+2} \to \mathbb{N}$ primitiv rekursive Funktionen die durch LOOP–Programme P_g und P_h programmiert sind. Dann gibt es für die primitiv rekursive Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$, die das primitive Rekursionsschema

1. $f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$
2. $f(x_1, \ldots, x_k, s(y)) = h(x_1, \ldots . x_k, y, f(x_1, \ldots, x_k, y))$

erfüllt, ein LOOP–Programm P_f, das f programmiert.

Beweis. Es wird jetzt das LOOP–Programm P_f angegeben:

$z_1 = x_1, z_2 = x_2, \ldots, z_k = x_k, z_{k+1} = x_{k+1}$; //Eingaben werden zwischengespeichert
P_g;
$y = 1$;
`LOOP` z_{k+1} `DO`
    $x_1 = z_1; \ldots; x_k = z_k$; //Entfällt bei $k = 0$
    $x_{k+1} = y - 1$;
    $x_{k+2} = x_0$;
    P_h;
    $y = y + 1$
`END` □

Beispiel. (vgl. Beispiel 1.1) Seien für $k = 1$ die Funktionen $g : \mathbb{N} \to \mathbb{N}$ mit $g(x) = x$ und $h : \mathbb{N}^3 \to \mathbb{N}$ mit $h(x_1, x_2, x_3) = x_3 + 1$ gegeben. Dann ergibt sich das primitive Rekursionsschema:

1. $f(x, 0) = g(x) = x$
2. $f(x, s(y)) = h(x, y, f(x, y)) = f(x, y) + 1$.

Weiterhin seien die zugehörigen LOOP–Programme

$$P_g \overset{\text{def}}{=} x_0 = x_1; \qquad \text{und} \qquad P_h \overset{\text{def}}{=} x_0 = x_3 + 1;$$

gegeben.

Daraus ergibt sich dann für P_f:

$z_1 = x_1; z_2 = x_2;$
$x_0 = x_1;$ //Programm P_g
$y = 1;$
`LOOP` z_2 `DO`
    $x_1 = z_1;$
    $x_2 = y - 1;$
    $x_3 = x_0;$
    $x_0 = x_3 + 1;$ //Programm P_h
    $y = y + 1;$
`END`

Satz 1.12. Die Menge der primitiv rekursiven Funktionen entspricht der Menge der LOOP–Programme, d.h.

1. Es gibt zu jeder primitiv rekursiven Funktion ein LOOP–Programm, welches diese programmiert (Lemmata 1.9, 1.10, 1.11).
2. Zu jedem LOOP–Programm gibt es eine primitiv rekursive Funktion, die dessen Bedeutung beschreibt. (ÜA)

Anmerkung. Dieser Satz bedeutet, dass man jedes *primitiv rekursive Problem* sowohl als rekursive Funktion, wie auch als Schleife implementieren kann.

1.3 μ–Rekursive Funktionen

Es stellt sich nun die Frage, ob alle Programme, die auf Computern implementierbar sind, primitiv rekursive Programme sind.

Betrachten wir zunächst die Endlosschleife.

Beispiel. Das einfachste Programm einer beliebigen Programmiersprache mit Schleifen, das die Endlosschleife repräsentiert, ist gegeben durch:

```
while (true){;}
```

Dieses Programm lässt sich nicht als primitiv rekursive Funktion darstellen, da alle primitiv rekursiven Funktionen total sind (Wenn ein Programm für eine bestimmte Eingabe keine Ausgabe liefert, so ist die Funktion, die dieses Programm programmiert an der Stelle nicht definiert, also ist die Funktion nicht total).

Neben den nicht totalen Funktionen gibt es noch eine weitere Klasse von Funktionen, die zwar total, aber trotzdem nicht primitiv rekursiv sind. Betrachten wir dazu einmal den Aufbau der Arithmetik (Addition, Multiplikation, Potenz, …):

1. Addition wird zurückgeführt auf s–Funktion (Nachfolger)
2. Multiplikation wird zurückgeführt auf die Addition.
3. Potenz wird zurückgeführt auf die Multiplikation.

   ...

Wir bilden eine Funktion, deren eines Argument angibt, wieviele Schritte in der eben aufgezeigten Reihe gegangen werden soll. Diese Idee führt zur Ackermannfunktion.

Definition 1.13 (Ackermannfunktion)**.** Die **Ackermannfunktion** $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ ist durch folgende Gleichung definiert:
$A(0, y) = y + 1$
$A(x, 0) = A(x - 1, 1)$
$A(x, y) = A(x - 1, A(x, y - 1))$

Lemma 1.14. Die Ackermannfunktion ist nicht primitiv rekursiv. Die Ackermannfunktion ist auf dem Rechner implementierbar.

Beweis. Als Beweis geben wir die Java–Funktion `Ackermann` an:

```
int A(int x, int y)
{
   if(x == 0) return(y+1);
       else if(y == 0) return A(x-1,1);
   else { return A(x-1, A(x,y-1)); }
}
```

□

Zusammenfassung:

1. Die Endlosschleife ist implementierbar, aber nicht primitiv rekursiv.
2. Die Ackermannfunktion ist implementierbar aber nicht primitiv rekursiv.

Die Klasse der implementierbaren Funktionen ist also "größer" als die Klasse der primitiv rekursiven Funktionen.

Definition 1.15 (μ–Operator)**.** Sei $h : \mathbb{N} \times \mathbb{N}^k \dashrightarrow \mathbb{N}$ eine partielle Funktion. Dann ist der μ–**Operator** definiert durch:

$$(\mu y)[h(y, x_1, \ldots, x_k) = 0] = min\{y \mid h(y, x_1, \ldots, x_k) = 0\} =: z,$$

wobei $h(y, x_1, \ldots, x_k)$ für alle $y < z$ definiert ist (also $\neq \perp$).

Wenn kein solches Minimum existiert, ist der μ–Operator undefiniert.

Definition 1.16 (Erfüllt das μ–Operator - Schema)**.** Man sagt $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ **erfüllt das** μ–**Operator–Schema**, wenn

$$f(x_1, \ldots, x_k) = (\mu y)[h(y, x_1, \ldots, x_k) = 0].$$

Beispiel. Die beiden folgenden Beispiele zeigen, wie mit Hilfe des μ–Operator–Schemas die Endlosschleife und die Ackermannfunktion beschrieben werden kann.

1. Sei $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mit $h(y, x) = 1$. Dann ist durch $f(x) = (\mu y)[h(y, x) = 0]$ die Funktion $f : \mathbb{N} \dashrightarrow \mathbb{N}$ mit $f(x) = \perp$ gegeben.
2. Sei $h : \mathbb{N} \times \mathbb{N}^2 \to \mathbb{N}$ mit $h(y, x_1, x_2) = \begin{cases} 0 \text{ falls } A(x_1, x_2) = y \\ 1 \text{ sonst} \end{cases}$

   Dann gilt $A(x_1, x_2) = (\mu y)[h(y, x_1, x_2) = 0]$

Anmerkung. Grundsätzlich: Die Entscheidung, ob die Anwendung der Funktion h den Wert 0 ergibt, ist primitiv rekursiv.

- Im 1. Beispiel ist das offensichtlich.
- Im 2. Beispiel bedeutet das, dass die Entscheidung, ob die Ackermannfunktion für eine gegebene Eingabe einen bestimmten Wert annimmt, *primitiv rekursiv ist.*

  Dies gilt, obwohl die Ackermannfunktion selbst *nicht primitiv rekursiv ist.*

  Idee: Dies kann man sich plausibel machen, da die Ackermannfunktion streng monoton steigend ist und somit die Frage, ob die Ackermannfunktion einen bestimmten Wert annimmt nach einer vorbestimmten Anzahl von Schritten entschieden ist. Also kann dies durch ein LOOP–Programm berechnet werden. Das wiederum heißt aber, dass die Entscheidungsfunktion primitiv rekursiv ist.

Definition 1.17 (Menge der rekursiven Funktionen)**.** Die Menge der **rekursiven Funktionen** besteht aus den Grundfunktionen und der Anwendung der Komposition, des primitiven Rekursionsschemas und der μ–Operator Bildung.

1.4 WHILE–Programme

Es bleibt jetzt noch die Erweiterung der LOOP–Programme, so dass auch der μ–Operator programmiert werden kann.

Definition 1.18 (WHILE–Programme)**.** Die Programmiersprache der **WHILE–Programme** umfasst Variablen, Konstanten, Trennsymbole, Operationssymbole, sowie alle Schlüsselwörter der LOOP–Programme.

Zusätzlich wird noch das Schlüsselwort „`WHILE`“ hinzugefügt.

Ein WHILE–Programm wird gebildet nach den Regeln 1. - 4. aus Definition 1.8 und der Regel:

5. Sei P ein WHILE–Programm und x_i eine Variable, so ist

   $$\texttt{WHILE}\ (x_i \neq 0)\ \texttt{DO}\ \ P\ \ \texttt{END}$$

   ein WHILE–Programm.

Bedeutung bzw. Semantik: P wird solange ausgeführt, wie $x_i \neq 0$ ist.

Lemma 1.19. Sei $h : \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ eine Funktion aus der Menge der rekursiven Funktionen, die durch das WHILE–Programm P_h programmiert wird. Dann gibt es ein WHILE–Programm P_f, das

$$f(x_1, \ldots, x_k) = (\mu y)[h(y, x_1, \ldots, x_k) = 0]$$

programmiert.

Beweis. Es wird jetzt das WHILE–Programm P_f angegeben:

$x_{k+1} = x_k, x_k = x_{k-1}, \ldots, x_2 = x_1;$
$z_{k+1} = x_{k+1}, z_k = x_k, \ldots, z_2 = x_2;$
$y = 0;$
$x_1 = y;$
$P_h;$
`WHILE` $(x_0 \neq 0)$ `DO`
    $y = y + 1;$
    $x_1 = y;$
    $x_{k+1} = z_{k+1}, x_k = z_k, \ldots, x_2 = z_2;$
    $P_h;$
`END`;
$x_0 = y;$

□

Satz 1.20. Die Menge der rekursiven Funktionen entspricht der Klasse der WHILE–Programme, d.h.:

1. es gibt zu jeder rekursiven Funktion ein WHILE–Programm, das diese programmiert.
2. zu jedem WHILE–Programm gibt es eine rekursive Funktion, die die Berechnung beschreibt.

Beweis. 1. Satz 1.12, Lemma 1.19.

2. Satz 1.12 und Übungs–Aufgabe.

□

Satz 1.21 (Halteproblem)**.** Es gibt kein WHILE–Programm das als Eingabe ein beliebiges WHILE–Programm W (in codierter Form) und eine Eingabe $(n_1, \ldots, n_k)$ für W nimmt und als Ausgabe 1 liefert, wenn die Ausführung von W mit der Eingabe $(n_1, \ldots, n_k)$ keine Endlosschleife ergibt und 0 liefert, wenn sich eine Endlosschleife ergibt.

Spezifikation des Halteproblems:

1. Deklarationen:
   Datentypen: $< \mathcal{W}; execute >$, wobei $\mathcal{W}$ die Menge der codierten WHILE–Programme ist und $execute : \mathcal{W} \times \mathbb{N}^k \dashrightarrow \mathbb{N}$ das Programm mit der gegebenen Eingabe ausführt.

2. Eingabe:
   $p \in \mathcal{W}, (n_1, \ldots, n_k) \in \mathbb{N}^k$

3. Ausgabe:
   $x \in \{0, 1\}$

4. Nachbedingung:
   $(((p, (n_1, \ldots, n_k)) \in Defexecute) \rightarrow x_0 = 1)\ xor$
   $(((p, (n_1, \ldots, n_k)) \notin Defexecute) \rightarrow x_0 = 0)$
   (mit *Def* bezeichnet man den Definitionsbereich)

Das heißt es gibt Probleme, die man zwar spezifizieren aber nicht programmieren kann.

Korollar 1.22. Es gibt durch prädikatenlogische Formeln beschreibbare Funktionen, die nicht durch WHILE–Programme programmierbar sind, also auf Rechnern nicht implementierbar sind.

1.5 Einführung Turingmaschine

Turingmaschinen sind Automaten. Ein Automat ist ein *Objekt*, das aus einer Menge von Zuständen und einer Menge von Zustandsübergängen besteht. Ein Automat erwartet eine Eingabe und verarbeitet diese, indem er Schritt für Schritt unterschiedliche Zustände annimmt. Schließlich liefert der Automat eine Ausgabe.

In Turingmaschinen steuert eine zentrale Einheit mit endlich vielen Zuständen die Verarbeitung. Auf einem Ein-/Ausgabeband können endlich viele Symbole stehen. Beim Start steht ein Eingabewort auf dem Band. Ein Lese-/Schreibkopf (LSK) kann Zeichen vom Band lesen und auf das Band schreiben. Anfangs steht der LSK auf dem linkesten Zeichen (Abbildung 1.1).

Formal kann man eine Turingmaschine wie folgt definieren.

Definition 1.23 (Turingmaschine)**.** Eine Turingmaschine (TM) sei gegeben durch eine Struktur (Q, Σ, Γ, δ, q_0, B, F) mit

- Q ist eine endliche, nicht-leere Menge von Zuständen

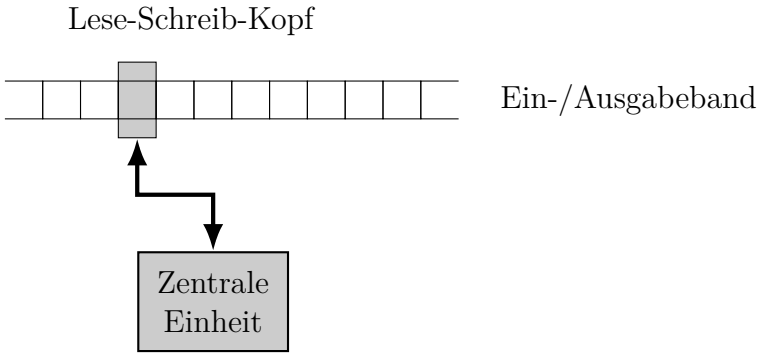


Abbildung 1.1: Turingmaschine

- Σ ist eine endliche Menge (Eingabealphabet)
- Γ ist eine endliche Menge (Symbole auf dem Band), d.h. $\Sigma \subset \Gamma$
- δ: Q $\times$ $\Gamma \rightarrow$ Q $\times$ Γ $\times$ D ist die Übergangsfunktion

  D = {L, R, N} gibt Werte für die Richtung der LSK-Bewegung an
- q_0 ist der Startzustand,
- B ist das Leerzeichen, d.h. das Zeichen, das außerhalb des Eingabewortes auf dem Band steht

  Es muss gelten: B $\in \Gamma$, aber B $\notin \Sigma$
- F $\subseteq$ Q ist die Menge der Endzustände.

Arbeitsweise der Turingmaschine:

- Turingmaschine beginnt im Startzustand, LSK zeigt auf das am weitesten links stehende Zeichen der Eingabe.
- Aktuelle Aktion (aktueller Schritt) hängt vom Zustand und vom gelesenen Zeichen ab.
- Schritt besteht aus drei Teilaktionen:
  - Wechsel des Zustandes
  - Schreiben eines Symbols auf das Band
  - Bewegen des LSK um eine Position nach rechts bzw. nach links oder stehenbleiben
- Die Turingmaschine hält an, wenn die Übergangsfunktion nicht definiert ist.

1.6 Turingmaschine als Automat zur Berechnung von rekursiven Funktionen

Man kann die Turingmaschine auch als Automat zur Berechnung von rekursiven Funktionen betrachten. Folgende Defintion beschreibt, wie die Berechnung erfolgt.

Definition 1.24 (Turing–berechenbar)**.** Eine n–stellige Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt *turing–berechenbar*, wenn es eine Turingmaschine $TM = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ B,\ F)$ gibt, für die folgendes gilt: Schreibt man die Eingabewerte $w_1, w_2, \ldots, w_n$ durch Blanks getrennt auf das Band

$$\ldots\, B\, B\, w_1\, B\, w_2\, B\, \ldots\, B\, w_n\, B\, B\, \ldots,$$

so hält die Turingmaschine TM in einem Finalzustand an und auf dem Band steht

$$\ldots\, B\, B\, f(\, w_1, w_2, \ldots, w_n\,)\, B\, B\, \ldots.$$

Anmerkung. Es muss eine Codierung der natürlichen Zahlen in das Eingabealphabet Σ der Turingmaschine geben.

Diese kann die Binärdarstellung der natürlichen Zahlen sein.

Oftmals wird aber auch die unäre Darstellung der natürlichen Zahlen (n wird durch n-mal 1 dargestellt) verwendet.

Satz 1.25 (Hauptsatz der Algorithmentheorie)**.** Die Menge der turing–berechenbaren Funktionen und die Menge der rekursiven Funktionen sind identisch.

Beweis.

- Es ist zu beweisen, dass es für jede rekursive Funktion eine Turingmaschine gibt, die diese berechnet.
- Es ist zu beweisen, dass es zu jeder Turingmaschine eine rekursive Funktion gibt, die dessen Berechnung beschreibt.

Für den Beweis wollen wir auf die einschlägige Literatur verweisen. □

Wir wollen hier als Beispiele die Turingmaschinen zur Berechnung der Grundfunktionen angeben:

Beispiel.

1. Nullstellige Nullfunktion C_0^0:

$$TM = (\{q_0, q_1\},\ \{0,1\},\ \{0,1,\#\}, \{(q_0,\#) \mapsto (q_1, 0, N)\},\ q_0, \#, \{q_1\})$$

2. Einstellige Nullfunktion C_1^0:

   Idee der Übergangsfunktion:

   - Man liest die Eingabe Zeichen für Zeichen und ersetzt jede Ziffer durch ein Blank
   - Wenn man ein Blank liest, schreibt man eine 0 und geht in den Finalzustand.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_0, q_1\},\ \{0,1\},\ \{0,1,\#\}, \delta,\ q_0, \#, \{q_1\})$$

   mit

$$\begin{aligned}\delta = \{&(q_0, 0) \mapsto (q_0, \#, R),\\ &(q_0, 1) \mapsto (q_0, \#, R),\\ &(q_0, \#) \mapsto (q_1, 0, N)\}\end{aligned}$$

3. Projektionsfunktionen Π_i^n:

   Idee der Übergangsfunktion:

   - Man liest die Eingaben Zeichen für Zeichen und ersetzt jede Ziffer durch ein Blank bis zur Eingabe i-1.
   - Bei jedem gelesenen Blank kommt man in einen neuen Zustand, so kann man feststellen, bei welcher Eingabe man sich im Moment befindet.
   - Die i. Eingabe lässt man stehen.
   - Es werden alle Zeichen ab der Eingabe i+1 durch Blanks ersetzt.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_1, q_2, \ldots, q_i, q_{i+1}, q_{i+2}\},\ \{0,1\},\ \{0,1,\#\}, \delta,\ q_1, \#, \{q_{i+2}\})$$

mit

$$\begin{aligned}\delta = &\{(q_j, e) \mapsto (q_j, \#, R) \mid 1 \leq j \leq i-1,\ e \in \{0,1\}\}\\ &\cup \{(q_j, \#) \mapsto (q_{j+1}, \#, R) \mid 1 \leq j \leq i-1\}\\ &\cup \{(q_i, e) \mapsto (q_i, e, R) \mid e \in \{0,1\}\}\\ &\cup \{(q_i, \#) \mapsto (q_{i+1}, \#, R)\}\\ &\cup \{(q_{i+1}, e) \mapsto (q_{i+1}, \#, R) \mid e \in \{0,1\}\}\\ &\cup \{(q_{i+1}, \#) \mapsto (q_{i+2}, \#, R)\}\\ &\cup \{(q_{i+2}, 0) \mapsto (q_{i+1}, \#, R), (q_{i+2}, 1) \mapsto (q_{i+1}, \#, R)\}\end{aligned}$$

4. Nachfolger-Funktion:

   Idee der Übergangsfunktion:

   - Zunächst einmal muss man in einem Zustand q_0 von links nach rechts über das Eingabewort gehen, damit der LSK auf dem letzten Zeichen steht.
   - Danach wechselt man in den Zustand q_1. Nun kann man die Binärzahl nach links zurückgehen: Befindet sich eine 1 auf dem Band, wird sie durch eine 0 überschrieben und man verbleibt im Zustand q_1 ($1 + 1 = 0$ Übertrag 1).
   - Sobald man auf eine 0 oder das Blank- Zeichen stößt, schreibt man eine 1 aufs Band und wechselt in den akzeptierenden Zustand q_2.

   Formal ergibt sich dann folgende Turingmaschine:

$$TM = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,\#\}, \delta, q_0, \#, \{q_2\})$$

   mit der Übergangsfunktion δ dargestellt als Tabelle:

Zustand	**Eingabesymbol**		
	0	1	#
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \#, L)$
q_1	$(q_2, 1, N)$	$(q_1, 0, L)$	$(q_2, 1, N)$
q_2	-	-	-

Definition 1.26 (k–Band Turingmaschine)**.** Die Maschine $TM = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ B,\ F, k)$ heißt *k–Band Turingmaschine* mit

1. $\Sigma,\ \Gamma,\ q_0,\ B,\ F$ wie in Def. 1.23
2. $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times D^k$ mit $D = \{L, R, N\}$.

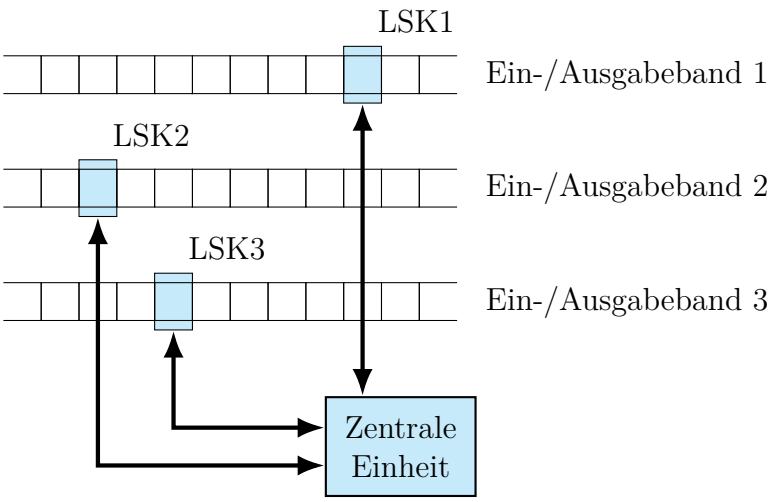


Abbildung 1.2: k-Band Turingmaschine

(vgl. Abbildung 1.2)

Am Anfang der Berechnung befindet sich die Eingabe auf dem ersten Band und die restlichen $k-1$ Bänder sind mit B besetzt. Am Ende der Berechnung befindet sich das Ergebnis auf dem ersten Band und die restlichen $k-1$ Bänder sind wiederum mit B besetzt.

Satz 1.27. Für eine Funktion f sind folgende Aussagen äquivalent:

1. f ist durch eine (1–Band) Turingmaschine berechenbar
2. f ist durch eine k–Band Turingmaschine berechenbar

Anmerkung. Weitere berechnungsäquivalente Turingmaschinenmodelle sind

- Turingmaschinen mit (mehrdimensionalem) Speicher (Abb. 1.3)
- Turingmaschinen mit getrenntem Ein– und Ausgabeband
- Turingmaschinen mit k Bändern und h_i Köpfen auf Band i, $1 \leq i \leq k$ und $h_i \in \mathbb{N}$

Wenn wir uns nochmals den Beweis–Ansatz von Satz 1.25 vergegenwärtigen, wird uns bewusst, dass man für jede rekursive Funktion eine eigene Turingmaschine benötigt. Dies ist wenig sinnvoll, da die Turingmaschine als abstraktes Modell für den Computer dienen soll, würde das bedeuten, dass man für jedes Programm einen eigenen Computer bräuchte.

Dieses Problem wird durch die *universelle Turingmaschine* gelöst.

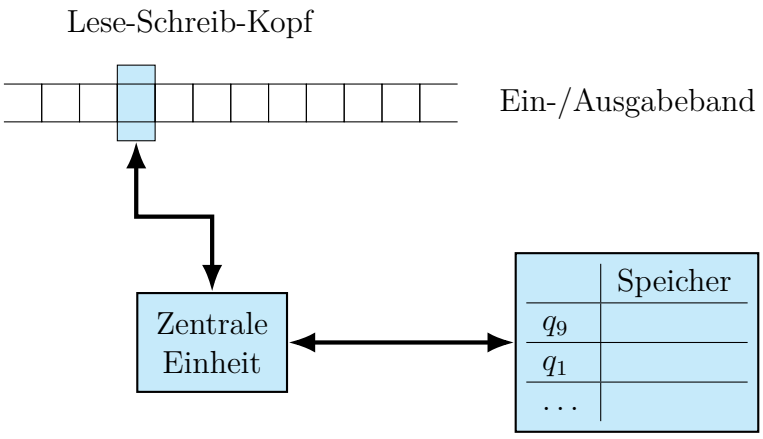


Abbildung 1.3: Modifizierte Turingmaschine: Speicher

Satz 1.28 (Satz über die universelle Turingmaschine)**.** Es gibt eine *universelle Turingmaschine* U, die bei codierter Eingabe des Programms einer beliebigen Turingmaschine T und einer beliebigen Dateneingabe w das gleiche Ergebnis liefert, wie die Turingmaschine T angewendet auf w.

Beweisidee:

- Benutzung einer 3–Band Turingmaschine
- Eingabe w auf Band 1
- Codierung der Funktion δ_T von T auf Band 2
- aktueller Zustand auf Band 3
- Funktion δ_U bearbeitet codiertes δ_T (Band 2) und verändert dabei Band 1 (entspricht dem Band von T) und Band 3 (Zustand von T).

Zusammenfassung

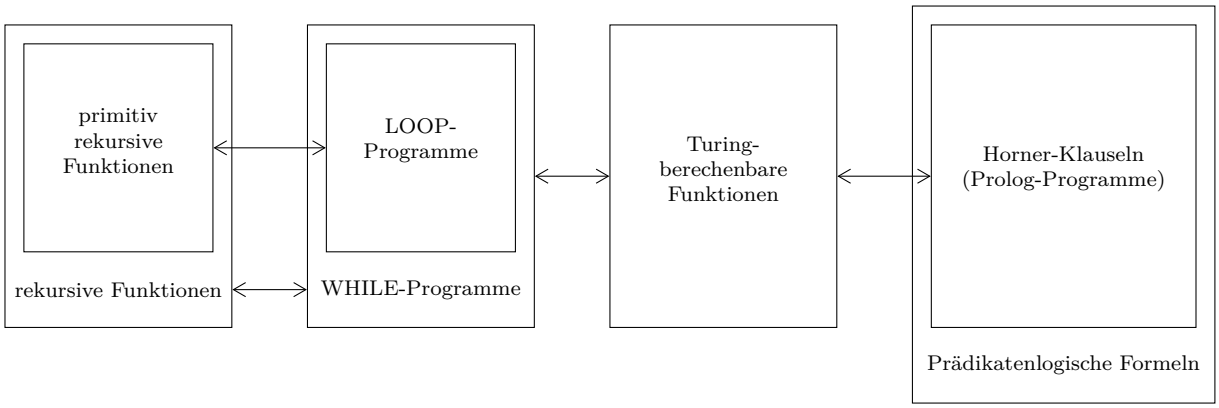


Abbildung 1.4: Hauptsatz der Algorithmentheorie

2 Grundlagen der Theorie der formalen Sprache

Zunächst wollen wir die beiden Begriffe *formale Sprache* und *natürliche Sprache* gegeneinander abgrenzen.

Formale Sprache:

- erlaubt es ihre Regeln (mathematisch) extakt zu definieren.
- Bsp. Java, C++, C - Shell, perl, PIZZA

Natürliche Sprache: Deutsch, Englisch

- über Jahrhunderte gewachsen
- Rechtschreibung
- Grammatik (nicht exakt, viele Ausnahmen)

Im Folgenden sei Σ die Menge von Symbolen (Buchstaben). Man nennt Σ das Alphabet.

Definition 2.1 (Menge aller Wörter)**.** Die kleinste Menge Σ^* die folgende Bedingungen erfüllt, heißt *Menge aller Wörter* über Σ:

1. $\varepsilon \in \Sigma^*$ (ε = leeres Wort)
2. wenn $w \in \Sigma^*$ und $a \in \Sigma$, dann ist $aw \in \Sigma^*$

Die *Länge* eines Wertes w bezeichnen wir mit $|w|$

Beispiel.

1. $\Sigma = \{0, 1\} \Rightarrow \Sigma^* = \{\varepsilon, 0, 1, 10, 11, 110, 111, \dots\}$
2. $\Sigma = \{a, b\}$, $M = \{\varepsilon, x, y, a, ax, bx, \dots\}$
   Frage: Ist M die Menge aller Wörter über Σ
   Nein: alle Elemente mit x und y müssen entfernt werden, da M zwar die zwei Bedingungen erfüllt, aber nicht die kleinste Menge ist, die diese Bedingungen erfüllt.

Definition 2.2 (Sprache)**.** Jede Teilmenge $L \subseteq \Sigma^*$ heißt Sprache.

Beispiel. $\Sigma = \{a, b\}$

- $\Sigma \subseteq \Sigma^*$
- $\Sigma^* \subseteq \Sigma^*$
- $\{aa, bb\} \subseteq \Sigma^*$ (endliche Sprache)
- $\{w \mid w \in \Sigma^*, w \text{ enthält an der 2. Stelle kein } a\}$ (unendliche Sprache)

Definition 2.3 (Semi–Thue–System)**.** Ein Paar (Σ, Π) heißt *Semi–Thue–System*, wenn Σ ein Alphabet ist und $\Pi \subseteq \Sigma^* \times \Sigma^*$ eine *Relation* mit folgenden Eigenschaften ist:
$\Pi = \{(l_i \to r_i) \mid 1 \leqslant i \leqslant n, n \geqslant 0, l_i, r_i \in \Sigma^*\}$

Beispiel. $\Sigma = \{a, b\}$

$\Pi \subseteq \Sigma^* \times \Sigma^* = \{(a \to a), (a \to b), (b \to b), (\varepsilon \to bb)\}$

Definition 2.4 (Ableitung)**.** Sei (Σ, Π) ein Semi–Thue–System, dann nennt man

$$\left\{w \xrightarrow{1} w' \mid w = ulv, w' = urv, l \to r \in \Pi\right\}$$

die Menge der *direkten Ableitungen.*

$$\left\{w \xrightarrow{*} w' \mid x_i \xrightarrow{1} x_{i+1}, w = x_0, w' = x_n, 0 \leq i \leq n\right\}$$

heißt *Menge der Ableitungen.*

Beispiel. $\Sigma = \{a, b\}$ $\Pi = \{(a \to ab), (b \to bb)\}$

- $w = \underbrace{abb}_{u}\underbrace{a}_{l}\underbrace{b}_{v} \xrightarrow{1} \underbrace{abb}_{u}\underbrace{ab}_{r}\underbrace{b}_{v} = w'$
- $w = \underbrace{abbab}_{x_0} \xrightarrow{1} \underbrace{abbabb}_{x_1} \xrightarrow{1} \underbrace{abbabbb}_{x_2} = w' \Rightarrow w \xrightarrow{*} w'$

Definition 2.5 (erzeugte Sprache)**.** Sei (Σ, Π) ein Semi–Thue–System und $l \in \Sigma^*$ so heißt $\mathcal{L}(\Pi, l) = \left\{w \mid l \xrightarrow{*} w\right\}$ die von l *erzeugte Sprache*

Beispiel. $\Sigma = \{A, \ldots Z, a, \ldots z, \{, \}, ;, \sqcup\}$
$\Pi = \{S \to \varepsilon, S \to CS, C \to class\sqcup N\sqcup\{\}; N \to \varepsilon, N \to Na, \ldots, N \to Nz\}$
$\mathcal{L}(\Pi, S) = \{\varepsilon, CS, CCS, \ldots,$
$class\sqcup N\sqcup\{\};,$
$class\sqcup Nz\sqcup\{\};,$
$class\sqcup Ntz\sqcup\{\};,\ldots;,$
$class\sqcup Nfritz\sqcup\{\};,$
$class\sqcup fritz\sqcup\{\};,\ldots\}$

Als Problem ergibt sich, dass in der erzeugten Sprache eigentlich nur Strings der Form

$$class\sqcup fritz\sqcup\{\};class\sqcup armin\sqcup\{\};$$

liegen sollten. Alle anderen Zwischenableitungen sind keine *Mini-Java-Programme*.

Wir führen eine Trennung im Alphabet ein:

1. Zeichen, die in der *Zielsprache* enthalten sind: *Terminale* (dürfen nicht weiter abgeleitet werden)
2. Zeichen, die in der *Zielsprache* nicht vorkommen: *Nichtterminale* (müssen weiter abgeleitet werden)

Diese Trennung macht es nötig, die Begriffe Semi–Thue–System und erzeugte Sprache neu zu definieren.

Definition 2.6 (Grammatik)**.** Eine *Grammatik* $G = (N, T, \Pi, S)$ ist gegeben durch:

- N, das Alphabet der *Nichtterminale*
- T, das Alphabet der *Terminale*
- $(N \uplus T, \Pi)$ ist eine Semi–Thue–System, wobei $N \uplus T$ die disjunkte Vereinigung ist $(N \cap T = \varnothing)$
- $S \in N$ ist das Startsymbol

Definition 2.7 (erzeugte Sprache)**.** Sei $G = (N, T, \Pi, S)$ eine Grammatik. Die von G erzeugte Sprache ist definiert durch $\mathcal{L}(G) = \left\{w \in T^* \mid S \xrightarrow{*} w\right\}$

Beispiel.

- $G = (N, T, \Pi, S)$
  $N = \{S, Z, R\}$
  $T = \{a, b\}$
  $\Pi = \{S \to aZ, S \to bZ, Z \to aR \mid a, R \to \varepsilon \mid aR \mid bR\}$
  $\mathcal{L}(G) = \{aa, aaa, aaaa, aab, aabb, \ldots\}$
  Die Sprache ist dadurch gegeben, dass immer an der 2.Stelle ein 'a' steht. Sonst stehen beliebige 'a' oder 'b'.

- $G = (N, T, \Pi, S)$ mit
  $T = \{c, l, a, s\}$,
  $N = \{S, A, B, C, D\}$,
  $\Pi = \{S \to cA, A \to lB, B \to aC, C \to sD, D \to s\}$
  $\mathcal{L}(G) = \{class\}$

Definition 2.8 (Chomsky–Hierarchie)**.** Sei $G = (N, T, \Pi, S)$ eine Gramatik.

Sprachtyp	Gramatik	Eigenschaft
0	allgemein	keine Einschränkung
1a	kontextsensitiv	Jede Produktion in Π hat die Form $(uAv \to u\beta v)$ mit $A \in N,\ u, v \in (N \uplus T)^*,\ \beta \in (N \uplus T)^* \setminus \{\varepsilon\}$ Ausnahme: $S \to \varepsilon$ ist erlaubt[1]
1b	nichtverkürzende Grammatik	Für jede Produktion in $p \to q \in \Pi$ gilt $\lvert p\rvert \leq \lvert q\rvert$. Ausnahme: $S \to \varepsilon$ ist erlaubt[1]
2	kontextfrei	Jede Produktion in Π hat die Form $(A \to \beta)$ mit $A \in N, \beta \in (N \uplus T)^* \setminus \{\varepsilon\}$ Ausnahme: $S \to \varepsilon$ ist erlaubt[1]
3	regulär	Jede Produktion hat in Π hat die Form $A \to aB$, $A \to a$, mit $a \in T; A, B \in N$. Ausnahme: $S \to \varepsilon$ ist erlaubt[1]

Beispiel. Grammatik für Klassennamen, Variablennamen, Methodennamen, … (*Identifier*)

1. Kontextfreie Grammatik

   $G = (N, T, \Pi, S')$ mit

   $T = \{a, \ldots z, A \ldots Z, 0 \ldots 9, \_\}$

[1]Das Startsymbol S darf in dem Fall auf keiner rechten Seite vorkommen.

$N = \{\mathit{Buchstabe}, \mathit{Ziffer}, \mathit{BezRest}, S'\}$

$$\begin{aligned}\Pi = \{ S' &\rightarrow \mathit{Buchstabe} \mid \mathit{Buchstabe}\ \mathit{BezRest} \\ \mathit{BezRest} &\rightarrow \mathit{Buchstabe}\ \mathit{BezRest} \mid \mathit{Ziffer}\ \mathit{BezRest} \mid \mathit{Buchstabe} \mid \mathit{Ziffer} \\ \mathit{Buchstabe} &\rightarrow a \mid \ldots \mid z \mid A \mid \ldots \mid Z, \\ \mathit{Ziffer} &\rightarrow 0 \mid \ldots \mid 9 \mid \_\}\end{aligned}$$

2. Reguläre Grammatik

$G = (N, T, \Pi, S')$ mit
$T = \{a, \ldots z, A \ldots Z, 0 \ldots 9, \_\}$
$N = \{S', A'\}$

$$\begin{aligned}\Pi = \{ S' &\rightarrow aA' \mid \ldots \mid zA' \mid AA' \mid \ldots \mid ZA' \mid a \mid \ldots \mid z \mid A \mid \ldots \mid Z \\ A' &\rightarrow aA' \mid \ldots \mid zA' \mid AA' \mid \ldots \mid ZA' \mid 0A' \mid \ldots \mid 9A' \mid \_A' \\ &\quad \mid a \mid \ldots \mid z \mid A \mid \ldots \mid Z \mid 0 \mid \ldots \mid 9 \mid \_\}\end{aligned}$$

3 Reguläre Sprachen

3.1 Reguläre Ausdrücke

Definition 3.1 (Reguläre Ausdrücke)**.** Sei Σ ein Alphabet, dann ist die Menge der regulären Ausdrücke über $\Sigma : R(\Sigma)$ definiert als kleinste Menge mit folgen Eigenschaften.

a) $\varepsilon \in R(\Sigma)$

b) $\Sigma \subseteq R(\Sigma)$

c) $\alpha \in R(\Sigma) \wedge \beta \in R(\Sigma) \Rightarrow \alpha\beta \in R(\Sigma)$
$\alpha|\beta \in R(\Sigma)$
$\alpha^* \in R(\Sigma)$

d) $\alpha \in R(\Sigma) \Rightarrow (\alpha) \in R(\Sigma)$

Beispiel. $\Sigma = \{a, b, c\}$

$$R(\Sigma) = \{\varepsilon, a, b, c, ab, ac, aa, \ldots, abac, aaa, \ldots, a|b, a|c, b|c, aa|bc, \ldots, aa|bc|aa, a^*, b^*, aaaa^*, (a), (b), (a|c), (a|c)^*, \ldots\}$$

Definition 3.2 (Reguläre Sprache)**.** Sei $\alpha \in R(\Sigma)$ ein regulärer Ausdruck, so ist die reguläre Sprache $\mathcal{L}(\alpha)$ definiert durch die kleinste Menge mit folgenden Eigenschaften:

1. $\mathcal{L}(\varepsilon) = \{\varepsilon(=\text{""})\}$
2. $\alpha \in \Sigma \Rightarrow \mathcal{L}(\alpha) = \{\alpha\}$
3. a) $\alpha = \beta\gamma \Rightarrow \mathcal{L}(\alpha) = \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\gamma)\}$
   b) $\alpha = \beta|\gamma \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta) \cup \mathcal{L}(\gamma)$
   c) $\alpha = \beta^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon\} \cup \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\beta^*)\}$
4. $\alpha = (\beta) \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta)$

Beispiel.

1. $\Sigma = \{a, b\}$
   a) $\alpha = a \Rightarrow \mathcal{L}(\alpha) = \{a\}$
   b) $\alpha = a|b \Rightarrow \mathcal{L}(\alpha) = \{a, b\}$
   c) $\alpha = a^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \ldots\}$
2. $\Sigma = \{a, b, c\}$
   $\alpha = \underbrace{(a^*|b^*)}_{\beta}\underbrace{c}_{\gamma} \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon c, ac, aac, aaac, \ldots, bc, bbc, bbbc, \ldots\}$

Satz 3.3. Die Menge der regulären Sprachen entspricht den Sprachen vom Typ *Chomsky-3.*

Beweis. Übungsaufgabe □

3.2 Endliche Automaten

Endliche Automaten prüfen, ob ein Eingabestring in einer vorgegebenen regulären Sprache liegt.

Definition 3.4 (Deterministische endliche Automat (DEA))**.** Unter einem DEA versteht man

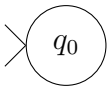
$$A = (Q, \Sigma, \delta, s, F)$$

mit

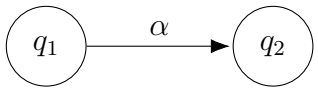
$Q \mathrel{\hat{=}}$ Zustandsmenge (endlich)
$\Sigma \mathrel{\hat{=}}$ Alphabet
$\delta \mathrel{\hat{=}}$ Übergangsfunktion: $Q \times \Sigma \to Q$
$s \in Q \mathrel{\hat{=}}$ Anfangszustand
$F \subseteq Q \mathrel{\hat{=}}$ Menge der Finalzustände

DEAs kann man grafisch darstellen:

- **Zustände:** 

- **Startzustand:** 

- **Finalzustand:** 

- **Übergangsfunktion:** 

Unter einer *Konfiguration* versteht man eine Element der Relation $Q \times \Sigma^*$

Algorithmus 3.5 (DEA-Erkennungsalgorithmus)**.**

Eingabe: $w \in \Sigma^*$, $A = (Q, \Sigma, \delta, s, F)$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: w wird von A erkannt.

1. Man startet mit der Konfiguration $(q, \overline{w}) = (s, w)(\in Q \times \Sigma^*)$
2. Wenn $\overline{w} \neq \varepsilon$, sei $\overline{w} = aw'$ mit $a \in \Sigma$. Falls $\delta(q, a) \neq \perp$: $q = \delta(q, a)$ und $\overline{w} = w'$. Weiter bei Schritt 2.
3. Wenn $\overline{w} = \varepsilon$ und der letzte Zustand q ein Finalzustand ist, dann erg = True, sonst erg = False

Beispiel. Sei A ein DEA (Abbildung 3.1) mit
$A = (Q, \Sigma, \delta, s, F)$, wobei
$Q = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$\delta = \{((q_0, a), q_1), ((q_1, b), q_0)\}$
$s = q_0$
$F = \{q_1\}$

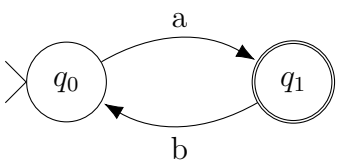


Abbildung 3.1: Graphische Darstellung des DEAs

Anwendung des Algorithmus:

(a) $w = aba$

- $(q_0, aba) \overset{2.}{\Rightarrow} (q_1, ba) \overset{2.}{\Rightarrow} (q_0, a) \overset{2.}{\Rightarrow} (q_1, \varepsilon)$
- $w' = \varepsilon$
- erg = True, da $q_1 \in F$

(b) $w = abb$

- $(q_0, abb) \overset{2.}{\Rightarrow} (q_1, bb) \overset{2.}{\Rightarrow} (q_0, b)$
- $\delta(q_0, b) = \perp$
- erg = False

(c) $w = ab$

- $(q_0, ab) \overset{2.}{\Rightarrow} (q_1, b) \overset{2.}{\Rightarrow} (q_0, \varepsilon)$
- $q_0 \notin F$
- erg = False

Satz 3.6. Wenn $w \in \Sigma^*$ von einem Automaten A akzeptiert wird, so sind die Konfigurationsübergänge eindeutig bestimmt.

Beweisidee: Ursache ist die Eigenschaft dass δ eine Funktion ist.

Anmerkung. Die Eindeutigkeit der Übergänge nennt man *deterministisch*.

Offen ist noch die Frage, wie man zum DEA A im 1. Schritt des Algorithmus (Def.3.4) kommt. Wir werden nun in mehreren Schritten zeigen, wie man aus einem gegebenen regulären Ausdruck einen DEA macht, der die Sprache des regulären Ausdrucks akzeptiert. Dazu definieren wir zunächst den nichtdeterministischen endlichen Automaten.

Definition 3.7 (Nichtdeterministische endliche Automat (NEA))**.** Unter einem *Nichtdeterministischen endlichen Automaten (NEA)* versteht man

$$A = (Q, \Sigma, \Delta, s, F) \text{ mit}$$

$Q \mathrel{\hat{=}}$ der Menge der Zustände (endlich).
$\Sigma \mathrel{\hat{=}}$ Alphabet
$\Delta \subseteq Q \times \Sigma^* \times Q \mathrel{\hat{=}}$ endliche Übergangsrelation

$s \in Q \mathrel{\hat{=}}$ Anfangszustand
$F \subseteq Q \mathrel{\hat{=}}$ Menge der Finalzustände

Die grafische Darstellung von NEAs und der zugehörige Algorithmus sind analog zum DEA (Definition 3.4) definiert.

Beispiel. $A = (Q, \Sigma, \Delta, s, F)$ mit
$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Delta = \{(q_0, a, q_1), (q_1, b, q2), (q_2, a, q_1), (q_1, b, q_0)\}$
$s = q_0$
$F = \{q_1\}$

Der NEA A ist in Abbildung 3.2 graphisch dargestellt.

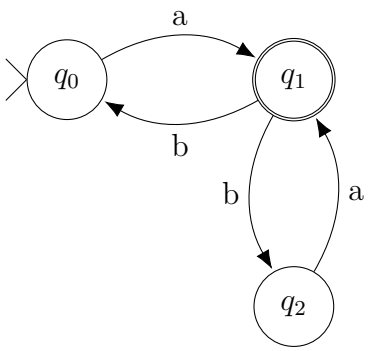


Abbildung 3.2: Nichtdeterministischer endlicher Automat

A ist kein DEA, da man von q_1 mit 'b' sowohl nach q_0 als auch nach q_2 kommen kann.

Anmerkung. Diese Uneindeutigkeit nennt man *nicht–deterministisch*.

Nun stellt sich die Frage, ob man jeden NEA in einen äquivalenten DEA überführen kann.

Satz 3.8. Zu jedem NEA existiert ein DEA, der die gleiche Sprache akzeptiert.

Beweis. Algorithmus der einen NEA in einen DEA umwandelt.

Algorithmus 3.9 (NEA2DEA)**.**

Eingabe: NEA $A = (Q, \Sigma, \Delta, s, F)$

Ausgabe: DEA $A' = (Q', \Sigma, \delta, s', F')$

Nachbedingung: Für eine Eingabe $w \in \Sigma^*$ liefern die Algorithmen zu A und A' das gleiche Ergebnis.

Zunächst definieren wir die folgenden Hilfsfunktionen:

$\varepsilon - closue : \mathcal{P}(\,Q\,) \to \mathcal{P}(\,Q\,)$ mit
$\varepsilon - closure(T) =$ **Abgeschlossene Hülle**$(\,T \cup \{q' \in Q \mid (q, \varepsilon, q') \in \Delta, q \in T\}\,), \quad (T \subseteq Q)$

$move : \mathcal{P}(\,Q\,) \times \Sigma \to \mathcal{P}(\,Q\,)$ mit

$$move(T, a) = \begin{cases} \perp & \nexists q \in Q \text{ mit } (q', a, q) \in \Delta, q' \in T \\ \{q \in Q \mid (q', a, q) \in \Delta, q' \in T\} & \text{sonst} \end{cases}$$

A' wird wie folgt berechnet:

$Q' = \{\varepsilon - closure(\{s\})\}$
while (es gibt einen unmarkierten Zustand T in Q') {
    markiere T;
    for jedes $a \in \Sigma\{$
        $U = \varepsilon - closure(move(T, a))$;
        if($U \notin Q'$) {füge U zu Q' hinzu}
        $\delta(T, a) := U$
    } //for
} //while
$s' = \varepsilon - closure(\{s\})$;
$F' = \{T \mid T \in Q' \wedge T \cap F \neq \varnothing\}$

□

Beispiel. Sei folgender Automat gegeben:

$A = (\{q_0, q_1, q_2, q_3, q_4\}(= Q),$
$\{a, b\}(= \Sigma),$
$\{(q_0, a, q_1), (q_1, b, q_0), (q_1, b, q_2), (q_2, \varepsilon, q_4), (q_4, \varepsilon, q_0), (q_2, a, q_3)\}(= \Delta),$
$q_0(= s), \{q_1, q_3\}(= F))$

Ablauf des Algorithmus:

Start: $Q' = \{\{q_0\}\}$

while–Schleife 1.Schritt: $T = \{q_0\}$
$Q' = \left\{\boxed{\{q_0\}}\right\}$

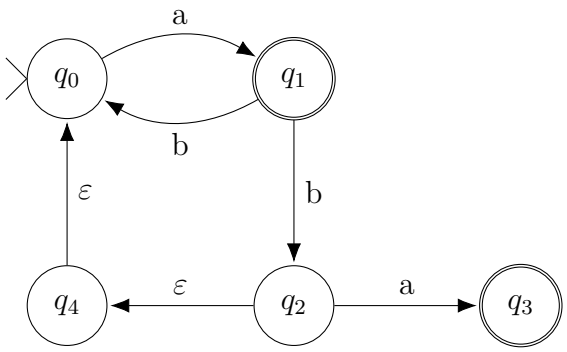


Abbildung 3.3: Graphische Darstellung von A

$a \in \Sigma$**:**

$U = \varepsilon - closure(move(T, a)) = \{q_1\}$

$Q' = \left\{\boxed{\{q_0\}}, \{q_1\}\right\}$

$\delta((\{q_0\}, a)) := \{q_1\}$

$b \in \Sigma$**:**

$move(T, b) = \bot$

while–Schleife 2.Schritt: $T = \{q_1\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}\right\}$

$a \in \Sigma$**:**

$move(T, a) = \bot$

$b \in \Sigma$**:**

$U = \varepsilon - closure(move(T, b)) = \{q_0, q_2, q_4\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}, \{q_0, q_2, q_4\}\right\}$

$\delta(T, b) := \{q_0, q_2, q_4\}$

while–Schleife 3.Schritt: $T = \{q_0, q_2, q_4\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}\right\}$

$a \in \Sigma$**:**

$U = \varepsilon - closure(move(T, a)) = \{q_3, q_1\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \{q_3, q_1\}\right\}$

$\delta(T, a) := \{q_3, q_1\}$

$b \in \Sigma$**:**

$move(T, b) = \bot$

while–Schleife 4.Schritt: $T = \{q_3, q_1\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \boxed{\{q_3, q_1\}}\right\}$

$a \in \Sigma$**:**

$move(T, a) = \perp$

$b \in \Sigma$**:**

$U = \varepsilon - closure(move(T, b)) = \{q_2, q_0, q_4\}$

$Q' = \left\{\boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \boxed{\{q_3, q_1\}}\right\}$

$\delta(T, b) := \{q_2, q_0, q_4\}$

$s' = \{q_0\}$

$F' = \{\{q_1\}, \{q_1, q_3\}\}$

Die graphische Darstellung des berechneten DEAs $A' = (Q', \Sigma, \delta, s', F)$ ist in Figure 3.4 gegeben:

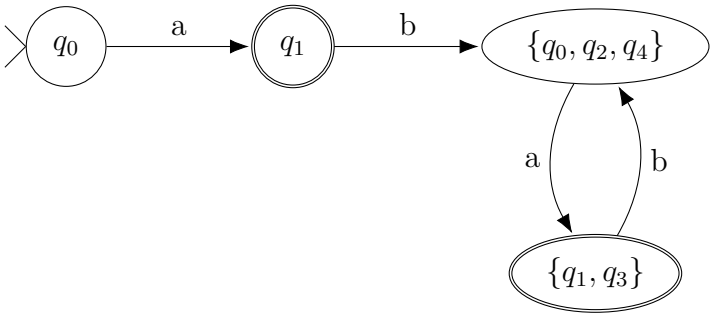


Abbildung 3.4: Graphische Darstellung von A'

Es bleibt noch zu klären wie man aus einem regulären Ausdruck einen *NEA* konstruiert, der die zugehörige Sprache erkennt.

Algorithmus 3.10 ($reg2auto$)**.**

Eingabe: $r \in R(\Sigma)$ (regulärer Ausdruck)

Ausgabe: $A = (Q, \Sigma, \Delta, s, F)$ (NEA)

Nachbedingung: $w \in \mathcal{L}(r) \Leftrightarrow \exists f \in F$, so dass sich aus der Konfiguration (s, w) die Konfiguration (f, ε) ableitbar ist.

Wir geben den Algorithmus *reg2auto* durch rekursive Regeln an:

ε**:**

$\text{reg2auto}(\varepsilon) = (\{q_0, q_1\}, \Sigma, \{(q_0, \varepsilon, q_1)\}, q_0, \{q_1\})$

$a \in \Sigma$**:**

$reg2auto(a) = (\{q_0, q_1\}, \Sigma, \{(q_0, a, q_1)\}, q_0, \{q_1\})$

$\alpha, \beta \in R(\Sigma)$**:**

$reg2auto(\alpha\beta) =$ **let**

$(Q_1, \Sigma, \Delta_1, s_1, \{f_1\}) = reg2auto(\alpha)$

$(Q_2, \Sigma, \Delta_2, s_2, \{f_2\}) = reg2auto(\beta)$

in

$(Q_1 \dot\cup Q_2, \Sigma, (\Delta_1 \dot\cup \Delta_2 \dot\cup \{(f_1, \varepsilon, s_2)\}), s_1, \{f_2\})$

end

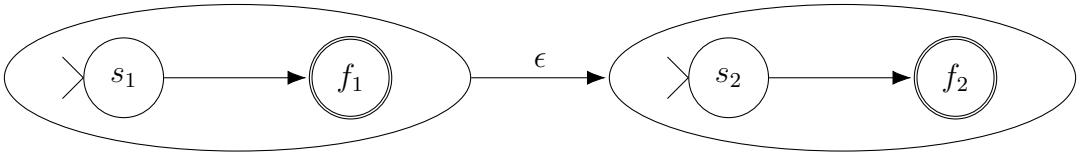


Abbildung 3.5: *reg2auto* Sequenz

$reg2auto(\alpha|\beta) =$ **let**

$(Q_1, \Sigma, \Delta_1, s_1\{f_1\}) = reg2auto(\alpha)$

$(Q_2, \Sigma, \Delta_2, s_2, \{f_2\}) = reg2auto(\beta)$

in

$((Q_1 \dot\cup Q_2 \dot\cup \{s_0, f_3\}, \Sigma, \Delta_1 \dot\cup \Delta_2 \dot\cup \{(s_0, \varepsilon, s_1), (s_0, \varepsilon, s_2), (f_1, \varepsilon, f_3), (f_2, \varepsilon, f_3)\}), s_0, \{f_3\})$

end

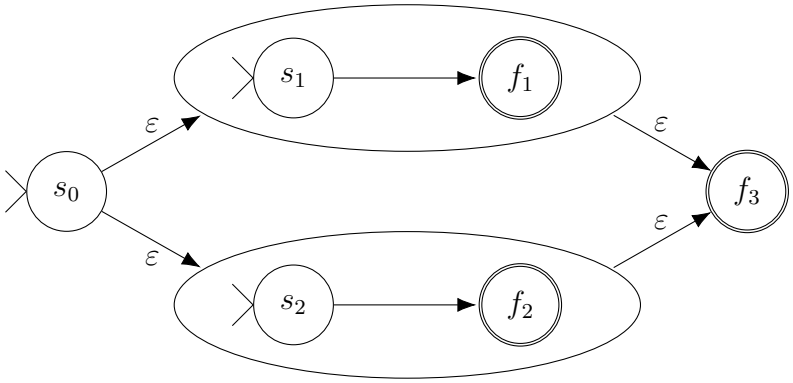


Abbildung 3.6: *reg2auto* Alternative

$reg2auto(\alpha^*) =$ **let**

$(Q_1, \Sigma, \Delta_1, s_1, \{f_1\}) = reg2auto(\alpha)$

in

$(Q_1, \Sigma, \Delta_1 \dot\cup \{(s_1, \varepsilon, f_1), (f_1, \varepsilon, s_1)\}, s_1, \{f_1\})$

end

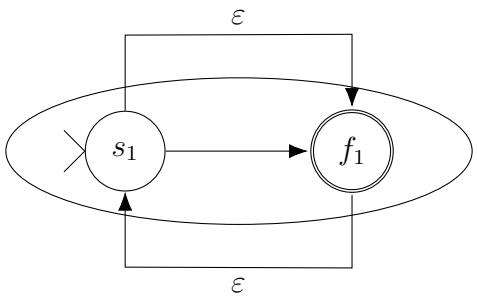


Abbildung 3.7: *reg2auto* Stern

Beispiel. $r = a(ba)^*$

$reg2auto(r)$:

$reg2auto(a) = (\{q_0, q_1\}, \Sigma, \{(q_0, a, q_1)\}, q_0, \{q_1\})$

$reg2auto((ba)^*)$:

$reg2auto(ba)$:

$reg2auto(b) = (\{q_2, q_3\}, \Sigma, \{(q_2, b, q_3)\}, q_2, \{q_3\})$

$reg2auto(a) = (\{q_4, q_5\}, \Sigma, \{(q_4, a, q_5)\}, q_4, \{q_5\})$

$= (\{q_2, q_3, q_4, q_5\}, \Sigma, \{(q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4)\}, q_2, \{q_5\})$

$= (\{q_2, q_3, q_4, q_5\}, \Sigma, \{(q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4), (q_2, \varepsilon, q_5), (q_5, \varepsilon, q_2)\}, q_2, \{q_5\})$

$= (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \Sigma,$

$\{(q_0, a, q_1), (q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4), (q_2, \varepsilon, q_5), (q_5, \varepsilon, q_2), (q_1, \varepsilon, q_2)\}, q_0, \{q_5\})$

Der berechnete Automat ist in Abbildung 3.8 dargestellt.

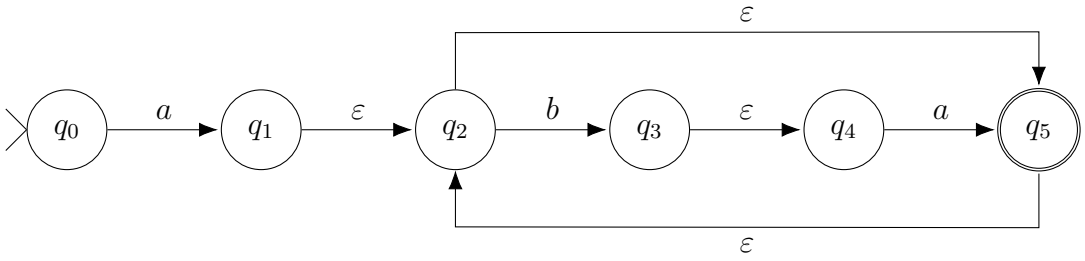


Abbildung 3.8: Ergebnis Beispiel zum Algorithmus *reg2auto*

Definition 3.11 (Automat akzeptiert Sprache)**.** Sei $r \in R(\Sigma)$ ein regulärer Ausdruck und A ein Automat. Der Automat A *akzeptiert* die Sprache $\mathcal{L}(r)$, falls für alle $w \in \mathcal{L}(r)$ aus der Startkonfiguration (s, w) eine Finalkonfiguration (f, ε) ableitbar ist, wobei s der Startzustand und f ein Finalzustand ist.

Satz 3.12 (Korrektheit von *reg2auto*)**.** Sei $r \in R(\,\Sigma\,)$ und $A = reg2auto(r)$. Genau dann akzeptiert A die Sprache $\mathcal{L}(r)$.

Beweis. Der Beweis wird per Induktion geführt. □

Satz 3.13. Folgende Aussagen sind äquivalent:

a) Es gibt eine regulären Ausdruck α.

b) Es gibt eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(\alpha)$.

c) Es gibt einen nichtdeterministischen endlichen Automaten der $\mathcal{L}(\alpha)$ akzeptiert.

d) Es gibt einen deterministischen endlichen Automaten der $\mathcal{L}(\alpha)$ akzeptiert.

Beweis. Der Beweis wurde teilweise erbracht.

a) ⇒ b): Übungsaufgabe 2.6

a) ⇒ c): Algorithmus *reg2auto*, Satz 3.12

c) ⇒ d): Satz 3.8

Für die restlichen Fälle wird auf die Literatur verwiesen. □

3.3 Von regulären Sprachen zu Scannern

Im vorhergehenden Abschnitt haben wir nun kennengelernt, wie man reguläre Sprachen durch reguläre Ausdrücke definiert. Wir werden im Folgenden (Kapitel 4) Sprachen über Alphabete definieren, deren Symbole aus *Tokens* (reguläre Sprachen) bestehen. Elemente der Tokens nennt man *Lexeme*.

Wenn Textfiles, die mehrere Lexeme enthalten, gelesen werden sollen und für jedes Token ein regulärer Ausdruck existiert, für den ein NEA gebildet wurde, so ist nicht klar, welcher NEA zu Erkennung des nächsten Lexems benötigt wird. Zur Lösung dieses Problems benötigt man einen Steuerungsalgorithmus, der im DEA–Simulator (vgl. Abbildung 3.9) enthalten ist.

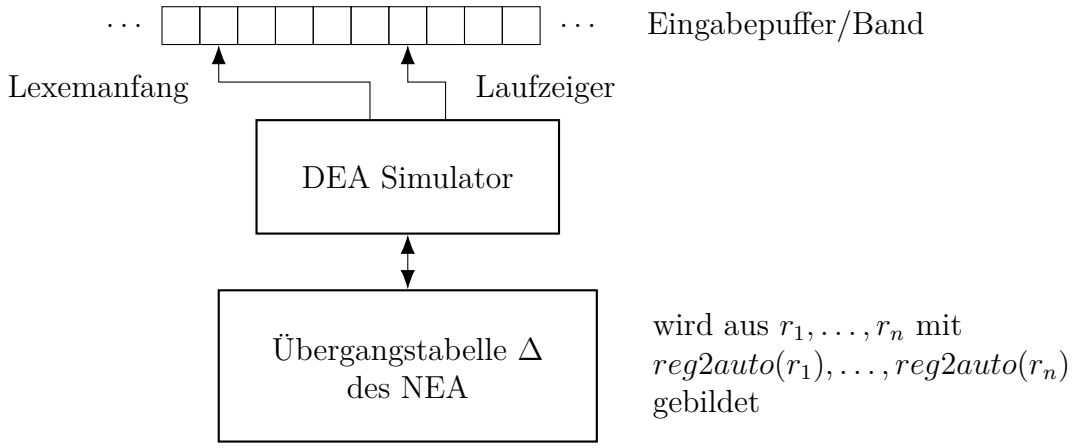


Abbildung 3.9: Scanvorgang eines Textfiles

Man definiert die Tokens einer Programmiersprache in einer *Lex–Spezifikation*:

$$r_1\{action\ 1\}$$
$$\vdots$$
$$r_n\{action\ n\}$$

$r_1 \ldots r_n$ sind reguläre Ausdrücke über dem Alphabet Σ z.B. der ASCII–Zeichen. action 1, ..., action n sind Aktionen (programmiert in einer Programmiersprache), die ausgeführt werden, wenn ein gelesenes Lexem zum jeweiligen regulären Ausdruck passt.

Beispiel.
`public` | `protected` | `private` // (Token *Zugriffsrechte*)
`static` // (Token *STATIC*)
`abstract` // (Token *ABSTRACT*)
`class` // (Token *CLASS*)
`while` // (Token *WHILE*)
`do` // (Token *DO*)
`if` // (Token *IF*)
(`a`|...|`z`|`A`|...|`Z`)(`a`|...|`z`|`A`|...|`Z`|`0`|...|`9`)* // (Token *IDENTIFIER*)
; // (Token *SEMIKOLON*)
"Σ*" // (Token *STRING*)

Problem: Oftmals ist es nicht eindeutig durch welche reguläre Ausdrücke ein erkanntes Lexem akzeptiert wird. (Bsp. `while` (man weiß nicht, ob while durch das Token *WHILE* oder das Token *IDENTIFIER* erkannt wird)

Regel: Principle of Longest match

Es wird immer so weit gelesen, dass nach dem nächsten Zeichen kein regulärer Ausdruck der Lex–Spezifikation mehr passen würde.
Genügt das längste passende Lexem immernoch mehreren regulären Ausdrücken, so wird der reguläre Ausdruck genommen, der den kleinsten Index hat.

Beispiel.
whilei $\Rightarrow$ Token *Identifier*
while $\rightarrow$ Token *WHILE*

Wir entwickeln den DEA–Simulator in 2 Schritten:
Zunächst geben wir einen Simulator an, der nur 1 Lexem akzeptiert. Dieser wird dann zum vollständigen DEA–Simulator erweitert.

Algorithmus 3.14. Sei $r \in R(\Sigma)$ ein regulärer Ausdruck und $A = (Q, \Sigma, \Delta, s_0, F) = reg2auto(\, r\,)$. Seien weiter $\varepsilon - closure$ und $move$ wie im Beweis zu Satz 3.8 gegeben.

Eingabe: $w \in \Sigma^*$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg =$ (w wird von reg2auto(r) akzeptiert)

$s = \varepsilon - closure(s_0)$
$a = nextchar()$ //liest nächstes Zeichen von Eingabestring
while(a $\neq$ "") {
    $s = \varepsilon - closure(move(s, a))$
    $a = nextchar()$
}
if((s $\cap$ F) $\neq \varnothing$)$\{return\ true;\}$
else $\{return\ false\}$

Es ist offensichtlich, das dieser Algorithmus den Algorithmus aus dem Beweis zu Satz 3.8 simuliert. Der Algorithmus liefert true, wenn das Wort akzeptiert wird. Wenn move(s, a) undefiniert ist, liefert der Algorithmus eine Exception (wird als false interpretiert). Wenn der letzte Zustand kein Finalzustand ist, wird false zurückgegeben.

Nächster Schritt: Übergang von einem regulären Ausdruck zu endlich vielen regulären Ausdrücken.

Algorithmus 3.15. Seien in einer Lex–Spezifikation die regulären Ausdrücke $r_1, \ldots, r_n$ gegeben und seien: $A_i = reg2auto(r_i)$ mit $A_i = (Q_i, \Sigma, \Delta_i, s_i, F_i)$. Wir fassen $A_1, \ldots, A_n$ zu einem NEA A zusammen:

$$A = ((\dot{\bigcup_i} Q_i) \dot{\cup} \{s_0\}, \Sigma, (\dot{\bigcup_i} \Delta_i) \dot{\cup} \Delta', s_0, (\dot{\bigcup_i} F_i))$$

wobei $\Delta' = \{(s_0, \varepsilon, s_i) \mid s_i, \text{Startzustand von } A_i\}$

Eine Veranschaulichung ist in Abbildung 3.10 zu sehen.

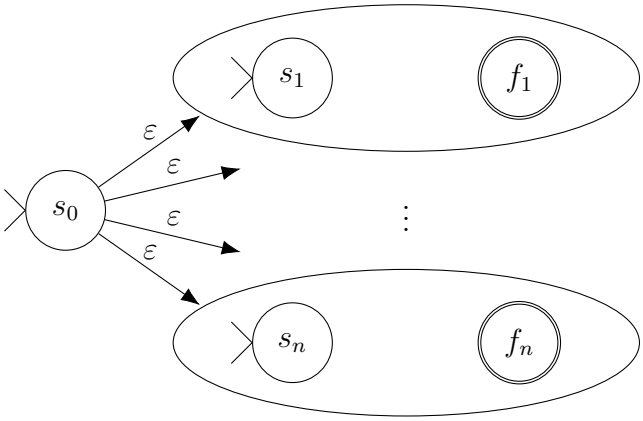


Abbildung 3.10: NEA für den Scanner

Eingabe: w (Eingabestring, z.B. Java - Datei) und A (konstruierter Automat)

Vorbedingung: w enthält nur Lexeme aus den Tokens der Lex-Spezifikation[1]

Ausgabe: $((lex_1, f_1^*), \ldots, (lex_m, f_m^*))$

Nachbedingung: $w = lex_1 + lex_2 + \ldots + lex_m$

$s_1 = w$

$s_{i+1} = lex_{i+1} + \ldots + lex_m$

lex_i ist das Lexem, das bei der Eingabe s_i erkannt wird (Principle of longest match)

f_i^* ist der Finalzustand, der nach dem Lesen von lex_i erreicht wird.

$start = 0$ //Pointer auf das 1. Zeichen von w
$a = nextchar()$
$pointer = 1$ //Pointer auf das 2. Zeichen von w.
$i = 1$ // Laufindex der die Lexeme zählt
while $(a \neq eof)$ { // End - Of - File
$\overline{s} = \varepsilon - closure(s_0)$

[1]Mögliche Fehleingaben können über ein Token der Fehleingaben abgefangen werden

while $(move(\overline{s}, a) \neq \perp)\{$
$\overline{s} = \varepsilon - closure(move, (\overline{s}, a))$
$a = nextchar()$
$pointer++$
$\}$
lex_i = w.substring (start, pointer–1)
f_i^* = Finalzustand in $\overline{s}$ mit kleinsten Index
$start = pointer - 1$
$i++$
$\}$

Beispiel. Lex–Spezifikation:

(" "$|\backslash t|\backslash n$) {;}
while {;}
$(a|\ldots|z)(a|\ldots|z)^*$ {;}

Aus der Lex–Spezifikation ergibt sich der NEA A in Abbildung 3.11.

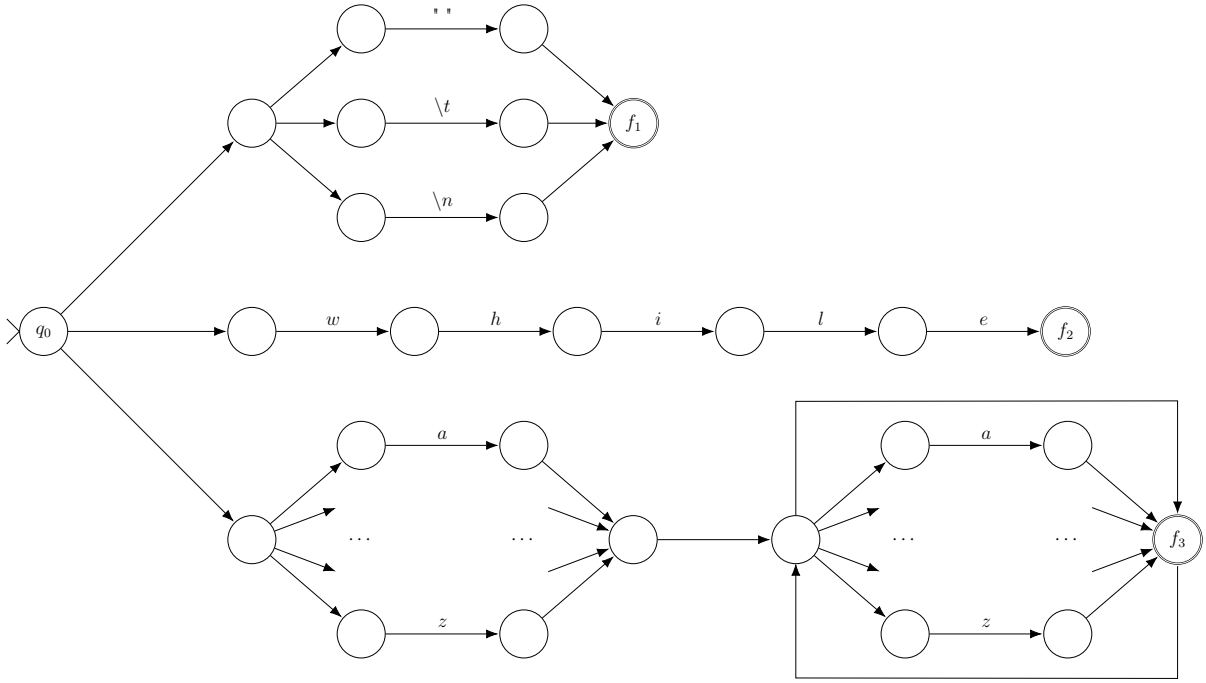


Abbildung 3.11: NEA der Lex–Spezifikation

$w =$ "*while whilez*"

Ablauf des Algorithmus:

1. Lexem:

$start = 0$

$a = "w"$

$pointer = 1$

$i = 1$

$a = "h"$

$pointer = 2$

$a = "i"$

$pointer = 3$

$a = "l"$

$pointer = 4$

$a = "e"$

$pointer = 5$

$a = " "$

$pointer = 6$

$move(\overline{s}, " ") = \perp$

$lex_1 = w.substring(0, 5) = "while"$

$f_1^* = f_2$

$start = 5$

$i = 2$

2. Lexem:

$a = "w"$

$pointer = 7$

$move(\overline{s}, "w") = \perp$

$lex_2 = w.substring(5, 6) = " "$

$f_2^* = f_1$

$start = 6$

$i = 3$

3. Lexem:

$a = "h"$

$pointer = 8$

$a = "i"$

$pointer = 9$

$a = "l"$

$pointer = 10$

$a = "e"$

$pointer = 11$

$a = "z"$

$pointer = 12$
$a = eof$
$pointer = 13$
$move(\overline{s}, eof) = \perp$
$lex_3 = w.substring(6, 12) =$ "$whilez$"
$f_3^* = f_3$

Ergebnis: (("$while$", f_2), (" ", f_1), ("$whilez$", f_3))

3.4 Der Scannergenerator JLex

JLex ist ein Tool, welches aus einer Lex–Spezifikation ein Java–Programm generiert, das den DEA–Simulator implementiert.
Die Actions der Lex–Spezifikation werden in Java ausprogrammiert.

3.4.1 Die JLex - Spezifikation

User Code (Java)
%%
JLex Direktive
%%
Lex–Spezifikation

3.4.2 JLex–Direktiven (Anweisungen)

`%class` *name* **:** definiert den Klassennamen *name* der erzeugten Klasse (default: `Yylex`)

`%function` *name* **:** definiert den Methodennamen *name*, die das nächste Lexem liest (default: `yylex()`)

`%type` *name* **:** definiert den Rückgabetyp von der Methode, die das nächste Lexem liest (default: `Yytoken`)

`%{`
$Java - Code$
`%}` **:** $Java - Code$ wird an den Anfang der erzeugten Klasse kopiert

%init{ $Java - Code$ **%init**}

: $Java - Code$ wird in den Konstruktor der erzeugten Klasse kopiert.

%eof{ $Java - Code$ **%eof**}

: $Java - Code$ wird ausgeführt bei Erreichen von dem Dateiende.

3.4.3 Java–Klasse erzeugen

Das Packet `JLex2.jar` muss in den `CLASSPATH` der Java–Umgebung aufgenommen werden (`tcsh`–Shell: `setenv CLASSPATH .:Jlex2.jar`).

Aufruf des Scannergenerators:

`java -cp JLex2.jar JLex2.Main Datei` (JLex-Spezifikation)

Dadurch wird die Klasse `Datei.java` erzeugt, die den DEA–Simulator für die JLex-Spezifikation `Datei` enthält

3.4.4 Beispiel: `html`–Lexeme

Als Beispiel geben wir eine JLex–Spezifikation für einen kleinen Ausschnitt der Sprache html an.

```
%%

%public
%class browserlexer
%type int
%endofval{
    System.out.println("EOF reached");
    return -1;
%endofval}

a = (a|A)
b = (b|B)
// ... hier noch weitere Abkürzungen

ws = [ \t\r\n\b\015]+

%%
```

```
"<"(h|H)(t|T)(m|M)(l|L)">"  { System.out.println(yytext()); }
"</"(h|H)(t|T)(m|M)(l|L)">" { System.out.println(yytext()); }

"<"{b}">"        { System.out.println(yytext()); }
"</"{b}">"       { System.out.println(yytext()); }

[^\<]+           { System.out.println(yytext()); }
{ws}             { System.out.println(yytext()); }
.                { System.out.println("FEHLER: "+yytext()); }
```

Nach den Direktiven werden durch `a`, `b` und `ws` Abkürzungen für reguläre Ausdrücke definiert. Als reguläre Ausdrücke werden die Tags `<html>`, `</html>` und `<b>`, `</b>` (jeweils alle Buchstaben klein und groß geschrieben) festgelegt. Weiterhin gibt es einen regulären Ausdruck, der alle Strings erkennt, die nicht mit `<` anfangen. Schließlich gibt es einen regulären Ausdruck `ws` für alle `Whitespaces` (Leerzeichen, Tabulatoren, ...) und einen regulären Ausdruck „.“ für alle Strings (außer *Newline*), die sonst nicht erfasst wurden.

```
class Main {
    public static void main(String args[]) throws java.io.
       IOException {
        browserlexer b = new browserlexer (new java.io.
           InputStreamReader(System.in));
        while (b.yylex() != -1);
    }
}
```

In der Methode `main` der Klasse `Main` wird eine Instanz der Klasse `browserlexer`, die aus der JLex–Spezifikation generiert wurde, erzeugt. Dann werden durch eine Schleife alle Lexeme nach dem *Principle of longest match* der Datei gelesen.

Mit folgendem `Makefile` kann man den Scanner generieren und compilieren.

```
Main.class: browserlexer.class Main.java
    javac Main.java

browserlexer.class: browserlexer.java
    javac browserlexer.java

browserlexer.java: browserlexer
    java -cp JLex2.jar JLex2.Main browserlexer

clean:
    rm *.class browserlexer.java
```

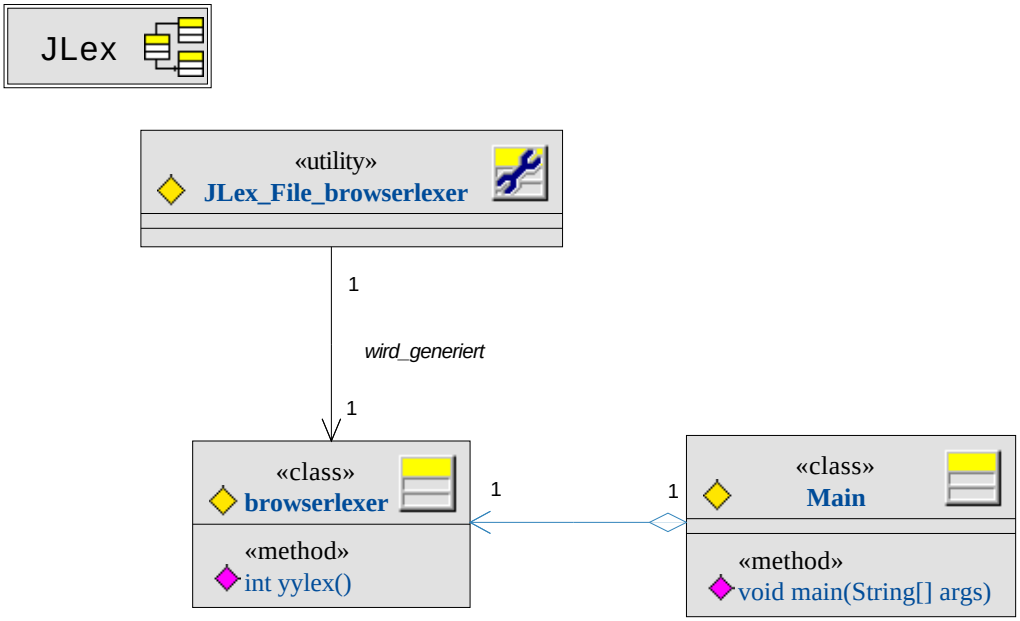


Abbildung 3.12: Klassen–Diagramm der Klassen des Scanners

4 Kontextfreie Sprachen

Nachdem wir nun im letzten Kapitel die Funktionsweise von Scanner betrachtet haben, wenden wir uns nun den Parsern zu. Parser lesen Texte und entscheiden, ob der Text in der Sprache einer vorgegebenen Grammatik liegt. Die Grammatik wird über einem Alphabet von Tokens definiert. Dabei werden die Tokens durch die Lex–Spezifikation eines zugehörigen Scanner festgelegt. Der Parser seinerseits ruft den Scanner auf um das nächste Lexem von der Eingabe zu lesen.

Verkürzt zusammengefasst kann man sagen:

Scanner: Zerlegt den Eingabetext in einzelne Tokens

Parser: Soll erkennen, ob die Anordnung der Tokens in einer „richtigen Reihenfolge“ ist.

WICHTIG: Terminal-Symbole können jetzt auch Strings sein, nicht wie bisher nur einzelne Symbole.

Beispiel (Super-Mini-Java)**.** Sei die Grammatik $G = (N, T, \Pi, S)$ gegeben mit

$T = \{class, \{, \}, ;, while, (,), true\}$
$N = \{S, Classdecl, Classbodydecl\}$
$\Pi = \{$

$$
\begin{aligned}
&S && \rightarrow Classdecl \\
&Classdecl && \rightarrow class\ Classbodydecl \\
&Classbodydecl && \rightarrow \{\} \mid \{while(true)\{;\}\}
\end{aligned}
$$

$\}$

4.1 Cocke-Younger-Kasami-Algorithmus

Definition 4.1 (Chomsky-Normalform)**.** Eine kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ ist in Chomsky-Normalform, wenn alle Produktionen aus Π entweder die Form $A \to BC$ mit $A, B, C \in N$ oder $A \to a$ mit $A \in N$ und $a \in \Sigma$ haben (Ausnahme: $S \to \epsilon \in \Pi$ erlaubt)

Satz 4.2. Für jede kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ existiert eine äquivalente kontextfreie Grammatik $G' = (N', \Sigma, \Pi', S)$ in Chomsky-Normalform mit $\mathcal{L}(G) = \mathcal{L}(G')$, falls $\mathcal{L}(G) \neq \emptyset$.

Beweis.

1. O.B.d.A. können wir davon ausgehen, dass alle Produktionen nach ausschließlich Terminal-Symbolen ableitbar sind. (Alle anderen Ableitungen können entfernt werden.)
2. $G' := G$
3. Alle Produktionen in Π', die bereits in Chomsky-Normalform sind, bleiben bestehen. In allen anderen Produktionen werden alle Terminal-Symbole $a \in \Sigma$ durch neue Nichtterminal-Symbole A besetzt, wobei $A \notin \Sigma$ und A in N' bzw. $(A \to a)$ in Π' ergänzt wird.
4. Für eine Folge von Produktionen $A_1 \to A_2, A_2 \to A_3, \ldots, A_{n-1} \to A_n$ mit $A_i \in N'$ muss eine Regel $A_n \to B_1 \ldots B_m$ oder $A_n \to a$ mit $B_i \in N'$ und $a \in \Sigma$ existieren.

   $\Rightarrow$ Jede Folge $A_1 \to A_2, A_2 \to A_3, \ldots, A_{n-1} \to A_n$ wird durch $A_1 \to B_1 \ldots B_m$ bzw. $A_1 \to a$ ersetzt.
5. Jede Produktion $A \to B_1 \ldots B_m$ mit $m > 2$ wird ersetzt durch,

   $A \to B_1 C_1$

   $C_i \to B_{i+1} C_{i+1}$ für $i \in \{1, \ldots, m-3\}$

   $C_{m-2} \to B_{m-1} B_m$

   wobei $C_1, \ldots, C_{m-2} \in N'$ neue Nichtterminal-Symbole sind.

□

Beispiel: Sie folgende Grammatik $G = (\{ S, C \}, \{ a, b \}, \Pi, S)$ mit

$$\begin{aligned}\Pi = \{ & S \to C \mid \epsilon, \\ & C \to aCb \mid ab \}\end{aligned}$$

gegeben.

Schritt 3:

$C \to aCb \mid ab$ wird ersetzt durch $C \to ACB \mid AB$.

$A \to a$, $B \to b$ wird hinzugefügt.

Schritt 4:

$S \to C$ wird ersetzt durch $S \to ACB \mid AB$.

Schritt 5:

$S \to ACB$ wird ersetzt durch $S \to AD, D \to CB$.

$C \to ACB$ wird ersetzt durch $C \to AE, E \to CB$.

Die äquivalente Grammatik in Chomsky-Normalform lautet:

$$G' = (\{ S, A, B, C, D, E \}, \{ a, b \}, \Pi', S)$$

mit

$$\begin{aligned}\Pi' = \{ & S \to AD \mid AB \mid \epsilon \\ & D \to CB \\ & C \to AE \mid AB \\ & E \to CB \\ & A \to a \\ & B \to b\}\end{aligned}$$

Algorithmus 4.3 (Cocke-Younger-Kasami-Algorithmus)**.** Sei eine kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ in Chomsky-Normalform gegeben.

Eingabe:	$w = a_1 \dots a_n \in \Sigma^*$
Ausgabe:	$erg \in \{True, False\}$
Nachbedingung:	$erg = True$, falls $w \in \mathcal{L}(G)$, sonst $erg = False$

$n = |w|$

for $(i = 1$ to $n)$

$\quad N_{ii} = \{A \mid (A \to a) \in \Pi \text{ mit } a = w[i]\}$

for $(d = 1$ to $n - 1)$

$\quad$ **for** $(i = 1$ to $n - d)$

$\quad\quad j = i + d$

$\quad\quad$ **for** $(k = i$ to $j - 1)$

$\quad\quad\quad N_{ij} = N_{ij} \cup \{A \mid (A \to BC) \in \Pi \text{ mit } B \in N_{ik} \text{ und } C \in N_{(k+1)j}\}$

if $(S \in N_{1n})$ // dann ist w aus S ableitbar

$\quad erg = True$

else

$\quad erg = False$

Beispiel (CYK-Algorithmus)**.** Grammatik $G = (N, T, \Pi, S)$ mit

$T = \{a, b, c\}$

$N = \{S, A, B, C\}$

$\Pi = \{$

$\quad S \to AB$

$\quad A \to AA$

$\quad B \to BC$

$\quad B \to BB$

$\quad C \to CC$

$\quad A \to a$

$\quad B \to b$

$\quad C \to c$

$\}$

$w =$ "$abcc$"

$n = |w| = 4$

1. for-Schleife:

$N_{11} = \{A\} \to a$

$N_{22} = \{B\} \to b$

$N_{33} = \{C\} \to c$

$N_{44} = \{C\} \to c$

2. for-Schleife:

$$
\begin{array}{lll}
d=1,\ i=1,\ j=2: & N_{12}: k=1: & A \in N_{11}, B \in N_{22} \Rightarrow (S \to AB) \in \Pi \quad \Rightarrow N_{12} = \{S\} \\
i=2,\ j=3: & N_{23}: k=2: & B \in N_{22}, C \in N_{33} \Rightarrow (B \to BC) \in \Pi \Rightarrow N_{23} = \{B\} \\
i=3,\ j=4: & N_{34}: k=3: & C \in N_{33}, C \in N_{44} \Rightarrow (C \to CC) \in \Pi \quad \Rightarrow N_{34} = \{C\} \\
d=2,\ i=1,\ j=3: & N_{13}: k=1: & A \in N_{11}, B \in N_{23} \Rightarrow (S \to AB) \in \Pi \quad \Rightarrow N_{13} = \{S\} \\
 & k=2: & S \in N_{12}, C \in N_{33} \Rightarrow \nexists X : (X \to SC) \in \Pi \\
i=2,\ j=4: & N_{24}: k=2: & B \in N_{22}, C \in N_{34} \Rightarrow (B \to BC) \in \Pi \quad \Rightarrow N_{24} = \{B\} \\
 & k=3: & B \in N_{23}, C \in N_{44} \Rightarrow (B \to BC) \in \Pi \Rightarrow \text{---}\,\text{„}\,\text{---} \\
d=3,\ i=1,\ j=4: & N_{14}: k=1: & A \in N_{11}, B \in N_{24} \Rightarrow (S \to AB) \in \Pi \Rightarrow N_{14} = \{S\} \\
 & k=2: & S \in N_{12}, C \in N_{34} \Rightarrow \nexists X : (X \to SC) \in \Pi \\
 & k=3: & S \in N_{13}, C \in N_{44} \Rightarrow \nexists X : (X \to SC) \in \Pi
\end{array}
$$

Ergebnis:

$S \in N_{14} \quad \Rightarrow \quad erg = True$

Überprüfung:

Die Ableitung lautet: $S \to AB \to aBC \to aBCc \to abcc$

Komplexität von CYK

Es handelt sich um zwei ineinander geschachtelte for-Schleifen. Zudem muss noch bei der Mengenbildung von N_{ij} über $1 \leq k < j$ gelaufen werden.

$\Rightarrow\ O(n^3)$

Dies ist keine ideale Laufzeit. Für eine Unterklasse/Teilmenge der kontextfreien Sprachen gibt es jedoch bessere Algorithmen, die eine Laufzeitkomplexität von $O(n)$ besitzen.

4.2 Push-Down-Automaten

Definition 4.4 (Push-Down-Automat (PDA))**.** Der Automat $A = (\Sigma, \Gamma, Q, \delta, \gamma_0, q_0, F)$ mit

1. Σ - nicht leeres, endliches Eingabealphabet
2. Γ - nicht leeres, endliches Kelleralphabet
3. Q - nicht leere, endliche Zustandsmenge
4. δ - Übergangsfunktion: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$[1]
5. $\gamma_0 \in \Gamma$ - Anfangssymbol auf dem Keller/Stack
6. $q_0 \in Q$ - Anfangszustand
7. $F \subseteq Q$ - Menge der Finalzustände

heißt Push-Down-Automat (oder Kellerautomat).

Definition 4.5 (Sprache von einem PDA akzeptiert)**.** Die Menge $L \subseteq \Sigma^*$ heißt von einem PDA $A = (\Sigma, \Gamma, Q, \delta, \gamma_0, q_0, F)$ akzeptiert, wenn gilt:

$$L = \{p \mid p \in \Sigma^* \wedge (\exists q \exists w : q \in F \wedge w \in \Gamma^* \wedge (q_0, p, \gamma_0) \xrightarrow{*} (q, \varepsilon, w))\},$$

wobei $(q, aw, sb) \to (q', w, sc)$ gilt, wenn $(q', c) \in \delta(\, q, a, b\,)$. (Interpretation: Vom Zustand q ausgehend kann ein a nur dann gelesen werden, wenn das oberste Element des Stapels (Stapelspitze) ein b ist. In diesem Fall wird q' zum aktuellen Zustand und die Stapelspitze wird durch ein c ersetzt.)

Gilt für jedes Tupel $(q, x, \gamma) \in (Q \times \Sigma \times \Gamma)$ und jedes $w \in \Gamma : |\delta(q, x, \gamma)| + |\delta(q, \varepsilon, w)| \leq 1$, so arbeitet der PDA deterministisch (DPDA), sonst nicht-deterministisch (NPDA). In erstem Fall ist die Übergangsfunktion gegeben durch:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to (Q \times \Gamma^*).$$

[1] Man kann δ auch als Relation betrachten $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times (Q \times \Gamma^*)$.

Beispiel (DPDA)**.** $A = (\{a, b\}, \{a, \gamma_0\}, \{q_0, q_1, q_2\}, \delta, \gamma_0, q_0, \{q_2\})$ mit

$\delta = \{$

$(q_0, a, \gamma_0) \mapsto \{(q_0, \gamma_0 a)\}$ (1)

$(q_0, a, a) \mapsto \{(q_0, aa)\}$ (2)

$(q_0, b, a) \mapsto \{(q_1, \varepsilon)\}$ (3)

$(q_1, b, a) \mapsto \{(q_1, \varepsilon)\}$ (4)

$(q_1, \varepsilon, \gamma_0) \mapsto \{(q_2, \varepsilon)\}$ (5)

$(q, x, \gamma) \mapsto \emptyset, \text{sonst}$

$\}$

A akzeptiert die Sprache $a^n b^n$:

$$(q_0, a^n b^n, \gamma_0) \xrightarrow{(1)} (q_0, a^{n-1} b^n, \gamma_0 a) \xrightarrow{(2)} (q_0, a^{n-2} b^n, \gamma_0 aa) \xrightarrow{(2)} \ldots \xrightarrow{(2)} (q_0, b^n, \gamma_0 a^n) \xrightarrow{(3)}$$
$$(q_1, b^{n-1}, \gamma_0 a^{n-1}) \xrightarrow{(4)} \ldots \xrightarrow{(4)} (q_1, \varepsilon, \gamma_0) \xrightarrow{(5)} (q_2, \varepsilon, \varepsilon)$$

Die Sprache $a^n b^n$ ist nicht regulär, d.h. DPDAs akzeptieren auch nicht-reguläre Sprachen.

Beispiel (Palindrom-Sprache)**.** $\mathcal{L}_{pal} = \{w\bar{w} \mid w \in \{0,1\}^* \wedge \bar{w} \text{ ist die Umkehrung von } w\}$

Der NPDA $A = (\{0, 1\}, \{\gamma_0, \gamma^0, \gamma^1\}, \{q_0, q_1, q_E\}, \delta, \gamma_0, q_0, \{q_E\})$ mit

$\delta = \{$

$(q_0, 0, \gamma_0) \mapsto \{(q_0, \gamma_0 \gamma^0)\}$ (1)

$(q_0, 1, \gamma_0) \mapsto \{(q_0, \gamma_0 \gamma^1)\}$ (2)

$(q_0, 0, \gamma^0) \mapsto \{(q_0, \gamma^0 \gamma^0), (q_1, \varepsilon)\}$ (3)

$(q_0, 0, \gamma^1) \mapsto \{(q_0, \gamma^1 \gamma^0)\}$ (4)

$(q_0, 1, \gamma^0) \mapsto \{(q_0, \gamma^0 \gamma^1)\}$ (5)

$(q_0, 1, \gamma^1) \mapsto \{(q_0, \gamma^1 \gamma^1), (q_1, \varepsilon)\}$ (6)

$(q_1, 0, \gamma^0) \mapsto \{(q_1, \varepsilon)\}$ (7)

$(q_1, 1, \gamma^1) \mapsto \{(q_1, \varepsilon)\}$ (8)

$(q_0, \varepsilon, \gamma_0) \mapsto \{(q_E, \varepsilon)\}$ (9)

$(q_1, \varepsilon, \gamma_0) \mapsto \{(q_E, \varepsilon)\}$ (10)

$\}$

Idee des Automaten: Gelesene 0er werden mit γ^0 und gelesene 1er mit γ^1 gemerkt. Wenn zwei gleiche Zeichen hintereinander gelesen werden ((3) und (6)), wird nichtdeter-

ministisch geraten, ob w fortgesetzt wird oder ob es sich bereits um das erste Zeichen von $\bar{w}$ handelt. q_0 ist dabei für die linke Hälfte und q_1 für die rechte Hälfte des Wortes zuständig.

Bemerkung:

- $\mathcal{L}_{pal}$ ist eine kontextfreie Sprache
- es gibt keinen DPDA, der $\mathcal{L}_{pal}$ akzeptiert *(ohne Beweis)*

Satz 4.6. Die Menge der Sprachen, die durch einen NPDA akzeptiert werden, ist eine echte Obermenge der Sprachen, die durch einen DPDA akzeptiert werden.

Beweis: siehe z.B. Winter, Theoretische Informatik

Im Folgenden wollen wir den Zusammenhang zwischen kontextfreien Sprachen und PDAs betrachten.

Zunächst soll aus einer kontextfreien Grammatik ein PDA konstruiert werden.

Idee des Automaten:

(1) S wird ohne ein Zeichen zu lesen auf den Stack geschrieben

(2) Ist auf dem Stack ein Nichtterminalsymbol, wird dieses durch seine Ableitung (rückwärts) ersetzt. Dabei wird kein Zeichen gelesen.

(3) Sind die gleichen Terminalsymbole auf Eingabe und Stack, werden beide gelöscht (= akzeptiert)

(4) Wenn alle Zeichen gelesen sind und γ_1 auf dem Stack liegt, ist das Wort akzeptiert

(5) Alle anderen Eingaben werden auf die leere Menge abgebildet, d.h. das Wort wird nicht akzeptiert

Satz 4.7 (Chomsky, Evey)**.** Eine Sprache ist genau dann kontextfrei, wenn sie von einem (N)PDA akzeptiert wird.

Beweis. Sei $G = (N, \Sigma, \Pi, S)$ eine kontextfreie Grammatik und $L = \mathcal{L}(G)$ die zugehörige Sprache.

Zu zeigen: Es gibt einen PDA, der L akzeptiert.

Herangehensweise: Wir konstruieren einen PDA, der L akzeptiert:

$A = (\Sigma, \Sigma \dot\cup N \dot\cup \{\gamma_0, \gamma_1\}, \{q_0, q_E\}, \delta, \gamma_0, q_0, \{q_E\})$ mit

$\delta = \{$

$(q_0, \varepsilon, \gamma_0) \mapsto \{(q_0, \gamma_1 S)\}$ (1)

$(q_0, \varepsilon, A) \mapsto \{(q_0, \bar{p} \mid (A \to p) \in \Pi)\} \quad \forall A \in N$ (2)

$(q_0, x, x) \mapsto \{(q_0, \varepsilon)\} \quad \forall x \in \Sigma$ (3)

$(q_0, \varepsilon, \gamma_1) \mapsto \{(q_E, \varepsilon)\}$ (4)

$(q, x, \gamma) \mapsto \emptyset,$ sonst (5)

$\}$

Für die Rückrichtung des Beweises wird z.B. auf Winter, Theoretische Informatik verwiesen. □

Im Folgenden betrachten wir nur noch die Klasse der kontextfreien Sprachen, die von DPDAs akzeptiert werden.

Wir werden sehen, dass es einen Algorithmus gibt, der diese Sprachen in linearer Zeit akzeptiert.

4.3 Parsertypen

Eingabe: Grammatik $G = (N, \Sigma, \Pi, S)$
$w \in \Sigma^*$

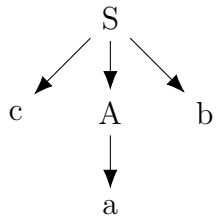
Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg = (w \in \mathcal{L}(G))$

Mit anderen Worten: Es muss eine Ableitung $S \overset{*}{\to} w$ gefunden werden.

Beispiel. $G = (N, T, \Pi, S)$ mit
$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb, A \to ab|a\}$
$w = cab$
$\Rightarrow$ *Ableitung* : $S \to cAb \to cab$
Ergebnis: $erg = True$

Man kann die Ableitung als Baum darstellen.



Die Blätter bilden das abgeleitete Wort.

Es gibt verschiedene Ansätze Algorithmen zu entwickeln, die der Spezifikation eines Parsers entsprechen.

- Man kann den Eingabestring von L inks oder von R echts lesen.
- Man kann die Ableitung unterschiedlich bilden. Man kann das am weitesten L inks stehende Nichtterminal oder das am weitesten R echts stehende Nichtterminal ableiten.

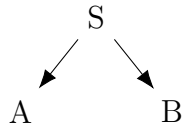
Definition 4.8 (Links- Rechtsableitungen)**.** Sei $w \to w'$ eine direkte Ableitung. Man bezeichnet $w \to_l w'$ als *direkte Linksableitung*, wenn das am weitesten links stehende Nichtterminal abgeleitet wird. Man bezeichnet $w \to_r w'$ als *direkte Rechtsableitung*, wenn das am weitesten rechts stehende Nichtterminal abgeleitet wird . Analog zu Definition 2.4 definert man *Links-* bzw. *Rechtsableitung*. (Bez.: $w \overset{*}{\to}_l w' bzw. w \overset{*}{\to}_r w'$)

Beispiel. Sei $G = (N, T, \Pi, S)$ eine Grammatik mit
$N = \{S, A, B, C\}$
$T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bB, B \to c\}$

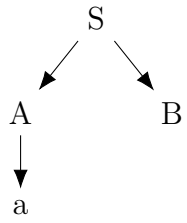
Sei weiterhin $w = abc$ das Eingabewort.

Aufbau des Ableitungsbaums durch Linksableitungen:

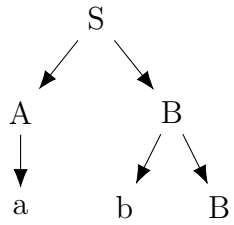
Man liest das nächste Token und wählt dann anhand diesem die Produktion aus.



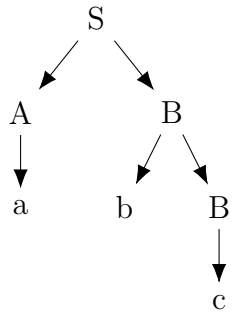
$nexttoken() = a$



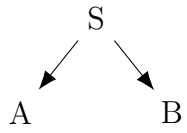
$nexttoken() = b$



$nexttoken() = c$



Aufbau des Ableitungsbaums durch Rechtsableitungen:



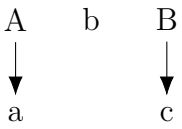
$nexttoken() = a$

Problem: Es ist nicht offensichtlich wie B abgeleitet wird, weil nicht klar ist welche Tokens von A abgeleitet werden können.

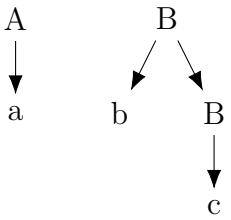
Anderer Ansatz: *Bottomup*: Wir bauen den Baum beginnend bei den *Blättern* zur *Wurzel* auf.

$nexttoken() = a$
wende an: $A \to a$

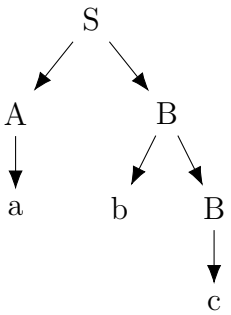
$nexttoken() = b$
$nexttoken() = c$
wende an: $B \to c$



wende an: $B \to bB$



wende an: $S \to AB$



Betrachtet man die Konstruktion rückwärts:

$$\underline{S} \to A\underline{B} \to Ab\underline{B} \to \underline{A}bc \to abc,$$

so erhält man die Rechtsableitung.

Definition 4.9 (Klassifikation der Parser)**.** Parser werden in Klassen eingeteilt, die angeben, ob ein Tokenstring von rechts oder links gelesen werden, ob Rechts- oder Linksableitungen gebildet werden und wieviele Zeichen vom Tokenstring gelesen werden, um die jeweilige Regel auszuwählen.

$LL(n) \mathrel{\hat{=}}$ Tokenstring von links lesen, links ableiten, n Zeichen vorausschauen (lookahead)
$LR(n) \mathrel{\hat{=}}$ Tokenstring von links lesen, rechts ableiten, n Zeichen vorausschauen

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit
$N = \{S, A\}$,
$T = \{a, b\}$
$\Pi = \{S \to A, A \to aA \mid ab\}$.

Sei weiter $w = aab$

Linksableitungen: $S \to A$ nexttoken() = a

Mit 1 Zeichen lookahead, kann man nicht entscheiden ob $A \to aA$ oder $A \to ab$ ausgewählt werden muss.

$\Rightarrow G$ liegt nicht in LL(1). G liegt in LL(2).

Wie sieht die Sprache einer LL(0) - Grammatik aus?

Eigenschaften:

- Es gibt keine Alternativen der Ableitung
- Die Sprache umfasst genau ein Wort

Beispiel (LL(0)-Grammatik)**.** $G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b\}$ und
$\Pi = \{S \to AB, A \to a, B \to b\}$

Es ist nur genau ein Wort ableitbar:

$S \to AB \to aB \to ab$

$L(G) = \{ab\}$

4.4 LR–Syntaxanalyse

Die LR–Syntaxanalyse ist eine Bottom–Up Syntaxanalyse. Man nennt die Syntaxanalyse auch eine ***Shift–Reduce–Syntaxanalyse***. Dabei wird für einen gegebenen Eingabestring (String von Lexemen) ein Parsbaum (Ableitungsbaum) von *Blättern zur Wurzel* (Bottom–Up) aufgebaut.

Beispiel. Sei $G = (N, T, \Pi, S)$ eine Grammatik mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Sei weiter $w = abbcde$ ein Eingabestring.

Bottom–Up Syntaxanalyse:

$$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.b \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow aA.de \longleftarrow aAd.e \longleftarrow aAB.e \longleftarrow$$

Shift-Reduce-Konflikt: $aAA.cde$

Reduce-Reduce-Konflikt: $aAA.e$

$$aABe \longleftarrow S$$

Es entsteht wiederum eine umgekehrte Rechtsableitung

Bei der LR-Syntaxanalyse können sich zwei unterschiedliche Konflikttypen ergeben:

- **Shift–Reduce–Konflikt**: Es ist nicht eindeutig, ob geschiftet oder reduziert werden soll.
- **Reduce–Reduce–Konflikt**: Es ist nicht klar, auf welches Nichtterminal reduziert werden soll.

Bevor wir den Algorithmus der LR-Syntaxtanalyse angeben, benötigen wir einige Hilfsdefinitionen.

Im Folgenden sei $G = (N, T, \Pi, S)$ eine Grammatik.

Definition 4.10 (LR(0)–Element)**.** Unter einem LR(0)–Element einer Grammatik versteht man eine Produktion mit einem Punkt in der rechten Seite.

Beispiel. $S \to A.B$

Definition 4.11 (Hülle)**.** Sei I eine Menge von LR(0)–Elementen einer Grammatik G, so ist die

$$\textit{Hülle}\colon \mathcal{P}(\{lr \mid lr \text{ ein LR(0)–Element}\}) \to \mathcal{P}(\{lr \mid lr \text{ ein LR(0)–Element}\})$$

die kleinste Menge , die durch folgende Regeln bestimmt ist:

- $I \subseteq \textit{Hülle}(I)$

- Wenn $(A \to \alpha.B\beta) \in \mathit{Hülle}(I)$ und $(B \to \gamma) \in \Pi$ ist, so ist $(B \to .\gamma) \in \mathit{Hülle}(I)$, wobei $A, B \in N, \alpha, \beta, \gamma \in (N \cup T)^*$

Beispiel. Seien $\Pi = \{S \to E, E \to E + E \mid E * E \mid id\}$, $N = \{S, E\}$ und $T = \{+, *, id\}$

Dann gilt:

Hülle $(\{S \to .E\}) = \{S \to .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}$
Hülle $(\{E \to E + .E\}) = \{E \to E + .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}$
Hülle $(\{E \to E. + E\}) = \{E \to E. + E\}$

Definition 4.12 (Sprung–Funktion)**.** Die Sprungfunktion

$$\mathit{Sprung} : \mathcal{P}\{lr \mid lr \text{ ist ein LR(0)–Element}\} \times (N \cup T) \to \mathcal{P}\{lr \mid lr \text{ ist ein LR(0)–Element}\}$$

ist gegeben durch

$$\mathit{Sprung}(I, x) = \bigcup_{(A \to \alpha.x\beta) \in I} \mathit{Hülle}\{A \to \alpha x.\beta\}$$

wobei $x \in (N \cup T), \alpha, \beta, \gamma \in (N \cup T)^*$

Beispiel. Seien N, T, Π wie im vorigen Beispiel gegeben.

Dann gilt:

$$\begin{aligned}\mathit{Sprung}(\{S \to E., E \to E. + E\}, +) &= \mathit{Hülle}(\{E \to E + .E\}) \\ &= \{E \to E + .E, E \to .E + E, E \to .E * E, E \to .id\}\end{aligned}$$

Definition 4.13 (Kanonische Sammlung von Mengen LR(0)–Elementen)**.** Sei eine Grammatik $G = (N, T, \Pi, S)$ gegeben. Dann ist $G' = (N \dot\cup \{S'\}, T, \Pi \dot\cup \{S' \to S\}, S')$ *die erweiterte Grammatik*. Die *Kanonische Sammlung von LR(0)-Elementen* C ist die kleinste Menge mit folgenden Bedingungen:

- Hülle $(\{S' \to .S\}) \in C$
- Wenn $I \in C$, dann gilt $\forall x \in (N \cup T)\ \mathit{Sprung}(I, x) \in C$, falls $\mathit{Sprung}(I, x) \neq \varnothing$

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit N, T, Π wie im vorigen Beispiel gegeben. Dann ist die erweiterte Grammatik gegeben durch $G' = (N \dot\cup \{S'\}, T, \Pi \dot\cup \{S' \to S\}, S')$
Im Folgenden geben wir die kanonische Sammlung an:

$H\ddot{u}lle\ (\{S' \to .S\}) = \underline{\{S' \to .S,\ S \to .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_0$

$Sprung\ (I_0, +) = \varnothing,\ Sprung\ (I_0, *) = \varnothing$

$Sprung\ (I_0, id) = H\ddot{u}lle\ (\{E \to id.\}) = \underline{\{E \to id.\}} =: \mathbf{I}_1$

$Sprung(I_0, E) = H\ddot{u}lle(\{S \to E.\})\ \cup\ H\ddot{u}lle(\{E \to E. + E)\ \cup\ H\ddot{u}lle(\{E \to E. * E\})$

$= \underline{\{S \to E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_2$

$Sprung\ (I_0, S) = H\ddot{u}lle\ (S' \to S.\}) = \underline{\{S' \to S.\}} =: \mathbf{I}_3$

$Sprung\ (I_1, x) = \varnothing$ für alle $x \in (N \cup T)$ //Sprungfunktion nicht anwendbar

$Sprung\ (I_2, +) = H\ddot{u}lle\ (\{E \to E{+}.E\}) = \underline{\{E \to E + .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_4$

$Sprung\ (I_2, *) = H\ddot{u}lle\{E \to E * .E\} = \underline{\{E \to E * .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_5$

$Sprung\ (I_2, E) = \varnothing,\ Sprung\ (I_2, id) = \varnothing,\ Sprung\ (I_2, S) = \varnothing$

$Sprung\ (I_3, x) = \varnothing$ für alle $x \in (N \cup T)$ //Sprungfunktion nicht anwendbar

$Sprung\ (I_4, +) = \varnothing\ Sprung\ (I_4, *) = \varnothing$

$Sprung\ (I_4, id) = \{E \to id.\} = \mathbf{I}_1$

$Sprung(I_4, E) = H\ddot{u}lle(\{E \to E + E.\})\ \cup\ H\ddot{u}lle(\{E \to E. + E\})\ \cup\ H\ddot{u}lle(\{E \to E. * E\})$

$= \underline{\{E \to E + E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_6$

$Sprung\ (I_4, S) = \varnothing$

$Sprung\ (I_5, +) = \varnothing,\ Sprung\ (I_5, *) = \varnothing$

$Sprung\ (I_5, id) = I_1$

$Sprung(I_5, E) = H\ddot{u}lle(\ \{E \to E * E.\}\ )\ \cup\ H\ddot{u}lle(\ \{E \to E. + E\}\ )\ \cup\ H\ddot{u}lle(\ \{E \to E. * E\}\ )$

$= \underline{\{E \to E * E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_7$

$Sprung(I_5, S) = \varnothing$

$Sprung(I_6, +) = I_4,\ Sprung(I_6, *) = I_5$

$Sprung(I_6, id) = \varnothing,\ Sprung(I_6, E) = \varnothing,\ Sprung(I_6, S) = \varnothing$

$Sprung(I_7, +) = I_4,\ Sprung(I_7, *) = I_5$

$Sprung(I_7, id) = \varnothing,\ Sprung(I_7, E) = \varnothing,\ Sprung(I_7, S) = \varnothing$

Damit ergibt sich die kanonische Sammlung $C = \{I_0, \ldots, I_7\}$.

Algorithmus 4.14 (LR(0)-Parser)**.**

Datentypen:	ADT Stack
Eingabe:	Eingabestring w Sprung-Funktion Kanonische Sammlung
Ausgabe:	Rechsableitung $rl \vee$ error
Nachbedingung:	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists$ LR(0)-Ableitung $\wedge$ error $\Leftrightarrow \nexists$ LR(0)-Ableitung (vgl. 4.15)
Init:	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer

while $(true)$ {
  $I = Stack.top()$
  $a = w[ip]$
  if $(Sprung(I, a) = I'$ **&&** $\nexists(A \to \alpha.) \in I)$ {
    $Stack.push(I')$
    $ip++$
  }
  else if $(I == \{A \to \beta.\}$ **&&** $A \neq S')$ {
    entferne $|\beta|$ Elemente vom Stack
    $I' = Stack.top()$
    $Stack.push(Sprung(I', A))$
    gib aus: $A \to \beta$
  }
  else if $(I == \{S' \to S.\})$ {
    gib aus: $S' \to S$
    return
  }
  else throw error
}

$\$ = eof$ (end of file)
$|\beta|$ = Länge von β

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit N, T, Π und die Kanonische Sammlung wie im vorigen Beispiel gegeben.

Algorithmusdurchlauf:

Eingabestring: 1 + 1

stack	**Eingabestring**	**nächster Schritt**
I_0	$1 + 1\$$	$Sprung(I_0, 1) = I_1$
I_0, I_1	$+1\$$	$reduce\ E \to id.$ $Sprung(I_0, E) = I_2$
I_0, I_2	$+1\$$	**error**, da $S \to E. \in I_2$

Es kann passieren, dass eine Menge von LR(0)–Elementen entweder

$A \to \alpha.\beta$
$B \to \gamma.$

oder

$A \to \gamma_1.$
$B \to \gamma_2.$

enthält.

In beiden Fällen ergeben sich Mehrdeutigkeiten bei der LR-Syntaxtanalyse. Im ersten Fall ist nicht klar, ob geschiftet oder reduziert werden soll. Im zweiten Fall ist nicht klar, welches LR(0)–Element reduziert werden soll.

Definition 4.15 (LR(0)–Eigenschaft)**.** Eine Grammatik G hat die LR(0)–Eigenschaft, wenn es keine Elemente ihrer Kanonischen Sammlung gibt, die entweder Elemente der Form

$$A \to \alpha.\beta \text{ und } B \to \gamma.$$

oder Elemente der Form

$$A \to \gamma_1. \text{ und } B \to \gamma_2.$$

enthalten, wobei A eine Nichtterminal und α, β, γ_1 und γ_2 Satzformen aus Terminalen und Nichtterminalen sind.

Die LR(0)–Eigenschaft bedeutet, dass eine Bottom–Up LR-Syntaxtanalyse ohne Vorausschau eindeutig aufgebaut werden kann. Man spricht in dem Fall von einer LR(0)–Ableitung.

Definition 4.16 (First)**.** Sei $G = (N, T, \Pi, S)$ eine Grammatik, dann ist

$$First : (N \cup T)^* \to \mathcal{P}(T)$$

gegeben durch folgende Funktion:

$First(a\beta) = \{a\}$, falls $a \in T$, $\beta \in (N \cup T)^*$

$First(A\beta) = \bigcup_{(A\to\alpha)\in\Pi} First(\alpha)$, falls $A \in N$, $(A \to \varepsilon) \notin \Pi$ und $\beta, \alpha \in (N \cup T)^*$

$First(A\beta) = (\bigcup_{\substack{(A\to\alpha)\in\Pi,\\ \alpha\neq\varepsilon}} First(\alpha)) \cup First(\beta)$, falls $A \in N$, $(A \to \varepsilon) \in \Pi$ und $\beta, \alpha \in (N \cup T)^*$

First(γ) ergibt alle Terminalsymbole, die als erstes Zeichen aus γ abgeleitet werden können.

Definition 4.17 (Follow)**.** Sei $G = (N, T, \Pi, S)$ eine Grammatik. Dann ist

$$Follow : N \to \mathcal{P}(T)$$

definiert durch die folgende Funktion:

$$\begin{aligned} Follow(A) &= (\bigcup_{(B\to\alpha A\beta)\in\Pi} (First(\beta))) \\ &\cup (\bigcup_{(B\to\alpha A)\in\Pi} (Follow(B))) \\ &\cup (\bigcup_{(B\to\alpha A\beta)\in\Pi,\beta\overset{*}{\to}\varepsilon} (Follow(B))), \end{aligned}$$

wobei $A, B \in N$ und $\alpha, \beta \in (N \cup T)^*$

Die Funktion Follow bestimmt die Terminale, die in einem beliebigen Ableitungsschritt auf ein Nichtterminal folgen können.

Beispiel. $\Pi = \{S \to ABC, A \to BC \mid a \mid d \mid \varepsilon, B \to b, C \to c \mid \varepsilon\}$
$N = \{A, B, C, S\}$
$T = \{a, b, c, d\}$

$$First(S) = First(ABC) = \underbrace{First(BC)}_{=\{b\}} \cup First(a) \cup First(d) \cup \underbrace{First(BC)}_{=\{b\}} = \{a, d, b\}$$

$$Follow(B) = \underbrace{First(C)}_{=\{c\}} \cup \underbrace{Follow(S)}_{=\emptyset} \cup \underbrace{Follow(A)}_{=First(B)=\{b\}} = \{c, b\}$$

Die nun folgende Action-Funktion gibt an, wie shift-reduce-Konflikte bzw. reduce-reduce-Konflikte aufgelöst werden können.

Definition 4.18 (Action-Funktion)**.** Sei G' eine erweiterte Grammatik und C die zugehörige kanonische Sammlung von Mengen von LR(0)–Elementen. Dann ist die Action-Funktion

$$\text{action: C} \times \mathsf{T} \rightarrow \textbf{auszuführende Funktion}$$

gegeben durch:

action(I, a) = „shift I'“, falls $Sprung(I, a) = I', (a \in T), (A \rightarrow \alpha.a\beta) \in I, \alpha, \beta \in (N \cup T)^*$
action(I, a) = „reduziere $A \rightarrow \alpha$“, falls $(A \rightarrow \alpha.) \in I$, $(A \neq S')$ und a ∈ Follow(A)
action(I, $\underbrace{\$}_{=(eof)}$) = „akzeptiere“, falls $(S' \rightarrow S.) \in I$

Algorithmus 4.19 (SLR-Parser)**.** (**S**imple **L**eft to **R**ight)

Datentypen:	ADT Stack
Eingabe:	Eingabestring w Sprung-Funktion kanonische Sammlung von Mengen von LR(0)–Elementen Action-Funktion
Ausgabe:	Rechsableitung $rl \vee$ error
Nachbedingung:	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists$ SLR-Ableitung $\wedge$ error $\Leftrightarrow \nexists$ SLR-Ableitung
Init:	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer

```
while (true) {
    I = Stack.top()
    a = w[ip]
    if (action(I, a) == „shift I'“) {
        Stack.push(I')
        ip++
    }
    else if (action(I, a) == „reduce A → β“) {
        endferne |β| Elemente vom Stack
        I' = Stack.top()
        Stack.push(Sprung(I', A))
        gib aus: A → β
    }
    else if (action(I, a) == „akzeptiere“) {
        gib aus: S' → S
        return
    }
    else throw error
}
```

$\$ = eof$ (end of file)
$|\beta|$ = Länge von β

Beispiel (SLR-Parser (1 von 2))**.** Sei $G = (N, T, \Pi, S')$ eine Grammatik mit:

$\Pi = \{S' \to S, S \to (S + S), S \to z\}$
$N = \{S', S\}$
$T = \{(,), +, z\}$

Kanonische Sammlung:

$I_0 = \mathit{Hülle}(\{S' \to .S\}) = \{S' \to .S,\ S \to .(S + S),\ S \to .z\}$
$I_1 = \mathit{Sprung}(I_0, S) = \{S' \to S.\}$
$I_2 = \mathit{Sprung}(I_0, () = \{S \to (.S + S),\ S \to .(S + S),\ S \to .z\}$
$I_3 = \mathit{Sprung}(I_0, z) = \{S \to z.\}$
$I_4 = \mathit{Sprung}(I_2, S) = \{S \to (S. + S)\}$
$\mathit{Sprung}(I_2, () = I_2$
$\mathit{Sprung}(I_2, z) = I_3$
$I_5 = \mathit{Sprung}(I_4, +) = \{S \to (S + .S),\ S \to .(S + S),\ S \to .z\}$
$I_6 = \mathit{Sprung}(I_5, S) = \{S \to (S + S.)\}$
$\mathit{Sprung}(I_5, () = I_2$
$\mathit{Sprung}(I_5, z) = I_3$
$I_7 = \mathit{Sprung}(I_6,)) = \{S \to (S + S).\}$

Sprung	S	z	$($	$)$	$+$
I_0	I_1	I_3	I_2	$\perp$	$\perp$
I_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
I_2	I_4	I_3	I_2	$\perp$	$\perp$
I_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
I_4	$\perp$	$\perp$	$\perp$	$\perp$	I_5
I_5	I_6	I_3	I_2	$\perp$	$\perp$
I_6	$\perp$	$\perp$	$\perp$	I_7	$\perp$
I_7	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\perp \mathrel{\hat{=}}$ undefiniert

$\mathit{Follow}(S) = \mathit{First}(+S)) \cup \mathit{First}()) \cup \mathit{Follow}(S') = \{+,), \$\}$

Action	z	$($	$)$	$+$	\$
I_0	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_1	$\perp$	$\perp$	$\perp$	$\perp$	akzeptiere
I_2	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_3	$\perp$	$\perp$	*red* $S \to z$	*red* $S \to z$	*red* $S \to z$
I_4	$\perp$	$\perp$	$\perp$	*shift* I_5	$\perp$
I_5	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_6	$\perp$	$\perp$	*shift* I_7	$\perp$	$\perp$
I_7	$\perp$	$\perp$	*red* $S \to (S+S)$	*red* $S \to (S+S)$	*red* $S \to (S+S)$

1. Algorithmusdurchlauf:

Stack	**Eingabestring**	**action**
I_0	$(z+z)\$$	*shift* I_2
$I_0\ I_2$	$z+z)\$$	*shift* I_3
$I_0\ I_2\ I_3$	$+z)\$$	*reduce* $S \to z$, $Sprung(I_2, S) = I_4$
$I_0\ I_2\ I_4$	$+z)\$$	*shift* I_5
$I_0\ I_2\ I_4\ I_5$	$z)\$$	*shift* I_3
$I_0\ I_2\ I_4\ I_5\ I_3$	$)\$$	*reduce* $S \to z$, $Sprung(I_5, S) = I_6$
$I_0\ I_2\ I_4\ I_5\ I_6$	$)\$$	*shift* I_7
$I_0\ I_2\ I_4\ I_5\ I_6\ I_7$	\$	*reduce* $S \to (S+S)$, $Sprung(I_0, S) = I_1$
$I_0\ I_1$	\$	*akzeptiere*

Ausgabe:

$$\left.\begin{array}{ll} S \to z & (1) \\ S \to z & (2) \\ S \to (S+S) & (3) \end{array}\right\} \Rightarrow \quad S' \to S \xrightarrow{(3)} (S+S) \xrightarrow{(2)} (S+z) \xrightarrow{(1)} (z+z)$$

2. Algorithmusdurchlauf:

Stack	**Eingabestring**	**action**
I_0	$()\$$	*shift* I_2
$I_0\ I_2$	$)\$$	$\perp \Rightarrow error$

Beispiel (SLR-Parser (2 von 2))**.** Sei $G = (N, T, \Pi, S')$ eine Grammatik mit:

$\Pi = \{S' \to S, S \to L = R \mid R, L \to^* R \mid id, R \to L\}$
$N = \{S', S, L, R\}$
$T = \{^*, =, id\}$

Kanonische Sammlung:

$I_0 = H\ddot{u}lle(\{S' \to .S\}) = \{S' \to .S,\ S \to .L = R,\ S \to .R,\ L \to .^*R,\ L \to .id,\ R \to .L\}$
$I_1 = Sprung(I_0, S) = \{S' \to S.\}$
$I_2 = Sprung(I_0, L) = \{S \to L. = R,\ R \to L.\}$
$I_3 = Sprung(I_0, R) = \{S \to R.\}$
$I_4 = Sprung(I_0, ^*) = \{L \to^* .R,\ R \to .L,\ L \to .^*R,\ L \to .id\}$
$I_5 = Sprung(I_0, id) = \{L \to id.\}$
$I_6 = Sprung(I_2, =) = \{S \to L = .R,\ R \to .L,\ L \to .^*R,\ L \to .id\}$
$I_7 = Sprung(I_4, R) = \{L \to^* R.\}$
$\ldots$

$Follow(R) = Follow(S) \cup Follow(L) = Follow(S') \cup First(= R) = \{=, \$\}$

Bereits jetzt ist ersichtlich, dass es zu Konflikten kommen wird:

$$\left.\begin{array}{l} Sprung(I_2, =) = I_6 \\ \quad \Longrightarrow\ action(I_2, =) = \text{„}shift\ I_6\text{“} \\ ((R \to L.) \in I_2) \wedge (= \in Follow(R)) \\ \quad \Longrightarrow\ action(I_2, =) = \text{„}reduce\ R \to L\text{“} \end{array}\right\} \textit{shift-reduce-Konflikt}$$

Definition 4.20 (SLR–Eigenschaft)**.** Eine Grammatik G hat die *SLR–Eigenschaft*, wenn die *action*–Funktion (Def. 4.18) eine wohldefinierte[2] partielle Funktion ist.

Es gibt weitere Algorithmen, die Konflikte in SLR lösen.

Anmerkung. Bei SLR-Parser wird die Follow-Menge für die gesamte Grammatik betrachtet. Man könnte die Follow-Menge auch für einzelne Produktionen betrachten. Dies nennt man LR(1)-Parsen (LR-Parsen mit Vorausschau von einem Symbol).

[2]Wohldefiniertheit einer Funktion bedeutet, dass jedem Wert des Definitionsbereichs maximal ein Wert des Wertebereichs zugeordnet ist.

Satz 4.21 (ohne Beweis)**.** Die Klassen der Sprachen die durch einen deterministischen Push-Down-Automaten akzeptiert werden, stimmt mit der Klasse der LR(1)-Sprachen überein.

Die Klasse der LR(1)-Sprachen stimmt mit der Klasse der LR(k)-Sprachen mit $k \geq 1$ (k Symbole Vorausschau) überein.

Für LR(1) und LALR(1) sei auf die Literatur verwiesen (z.B. Aho, Alfred V. and Lam, Sethi, Ullman, Jeffrey: *Compiler: Prinzipien, Techniken und Werkzeuge*)

5 Turingmaschine als Erkennungsautomat für Chomsky–1 und Chomsky–0 Sprachen

5.1 Turingmaschine als Erkennungsautomat für formale Sprachen

Beispiel (Turingmaschine zum Erkennen einer Sprache)**.** Zunächst geben wir einmal die Spezifikation für einen Automaten, der eine Sprache erkennt, an.

Deklarationen: $< \Sigma; * >$ Alphabet der Sprache mit der Operation *, die die Menge aller Wörter (jede beliebige Kombination aus Zeichen des Alphabets Σ) bildet.

Eingabe: $e \in \Sigma*$ (zu prüfendes Wort)
$L \subseteq \Sigma*$ (Sprache)

Ausgabe: $a \in \{true, false\}$

Nachbedingung: $a = (e \in L)$.

In Worten lässt sich die Spezifikation wie folgt zusammenfassen: Automaten, die eine Sprache erkennen, sollen als Eingabe eine beliebige Kombination von Zeichen akzeptieren und als Ausgabe angeben, ob die Eingabe in der vorgegebenen Sprache liegt.

Nun wollen wir das Alphabet $\Sigma = \{0, 1\}$ und die Sprache L, die alle Worte enthält, die zunächst hintereinander n-mal die 0 und dann n-mal die 1 haben ($L = \{0^n 1^n \mid n \geq 1\}$), betrachten.

Idee der Konstruktion einer Turingmaschine TM:

Am Anfang steht das Eingabewort w $\in$ {0,1 }* auf dem Band.

- TM ersetzt abwechselnd eine 0 durch ein a und eine 1 durch ein b

- Wird im gleichen Durchgang die letzte 0 und die letzte 1 ersetzt, ist das Wort akzeptiert

Folgende Zustände:

- TM beginnt im Startzustand q_0 und zeigt auf die am weitesten links stehende 0.
- Nach Ersetzen von 0 durch a Wechsel in Zustand q_1, beim Bewegen des LSK nach rechts werden alle 0 und b überlesen, bis 1 erreicht wird
- Dann wird diese durch b ersetzt, in den Zustand q_2 gewechselt
- Im Zustand q_2 geht es nach links zurück, bis auf ein a gestoßen wird, dann Rechtsschritt und Wechsel in den Zustand q_0
- Sind alle 0 und 1 durch a und b überschrieben, dann trifft TM bei Rechtsbewegung im Zustand q_0 auf ein b, dann Wechsel nach q_3
- Vom Zustand q_3 geht es unter b weiter nach rechts
- Wird in diesem Zustand auf B getroffen, dann Wechsel in Endzustand q_4

Wenn die Turingmaschine TM im Finalzustand q_4 anhält, ist das Wort in der Sprache L enthalten. Hält die Turingmaschine in einem anderen Zustand an, so ist das Wort nicht in der Sprache L enthalten.

Formal sieht die Turingmaschine dann wie folgt aus:

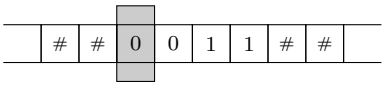
$$TM = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, a, b, \#\}, \delta, q_0, \#, \{q_4\})$$

mit der Übergangsfunktion δ:

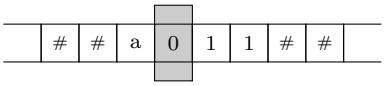
		Symbol				
		0	1	a	b	#
Zustand	q_0	(q_1, a, R)	-	-	(q_3, b, R)	-
	q_1	$(q_1, 0, R)$	(q_2, b, L)	-	(q_1, b, R)	-
	q_2	$(q_2, 0, L)$	-	(q_0, a, R)	(q_2, b, L)	-
	q_3	-	-	-	(q_3, b, R)	$(q_4, \#, L)$
	q_4	-	-	-	-	-

Wir rechnen die Turingmaschine nun mit einem Wort als Eingabe durch, das in der Sprache L enthalten ist:

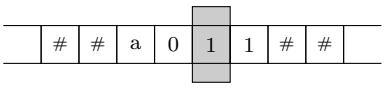
- Eingabe von 0011, Startzustand q_0:



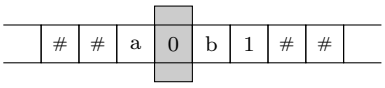
- $(q_0, 0) \mapsto (q_1, a, R)$:



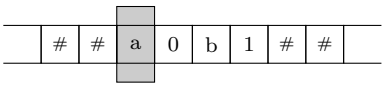
- $(q_1, 0) \mapsto (q_1, 0, R)$:



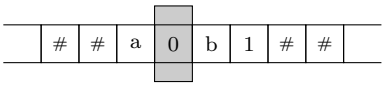
- $(q_1, 1) \mapsto (q_2, b, L)$:



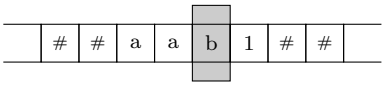
- $(q_2, 0) \mapsto (q_2, 0, L)$:



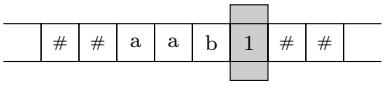
- $(q_2, a) \mapsto (q_0, a, R)$:



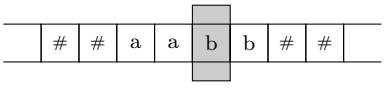
- $(q_0, 0) \mapsto (q_1, a, R)$:



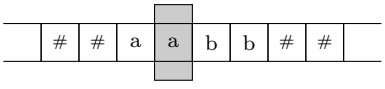
- $(q_1, b) \mapsto (q_1, b, R)$:



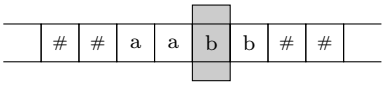
- $(q_1, 1) \mapsto (q_2, b, L)$:



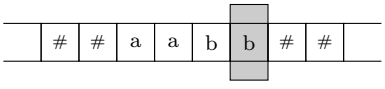
- $(q_2, b) \mapsto (q_2, b, L)$:



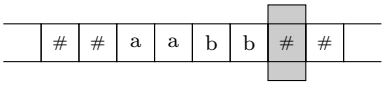
- $(q_2, a) \mapsto (q_0, a, R)$:



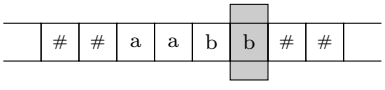
- $(q_0, b) \mapsto (q_3, b, R)$:



- $(q_3, b) \mapsto (q_3, b, R)$:



- $(q_3, \#) \mapsto (q_4, \#, L)$:

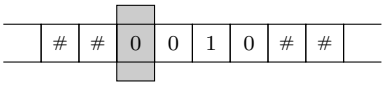


- TM hält an und befindet sich im Finalzustand q_4

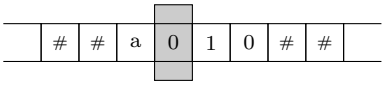
  $\Rightarrow$ Eingabe ist akzeptiert

Nun rechnen wir die Turingmaschine auch noch mit einer Eingabe durch, die nicht in der Sprache L enthalten ist.

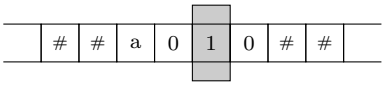
- Eingabe von 0010, Startzustand q_0:



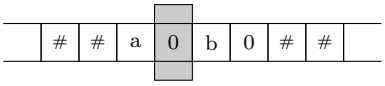
- $(q_0, 0) \mapsto (q_1, a, R)$:



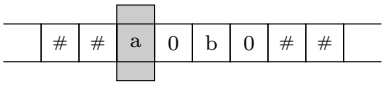
- $(q_1, 0) \mapsto (q_1, 0, R)$:



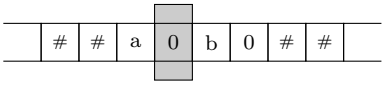
- $(q_1, 1) \mapsto (q_2, b, L)$:



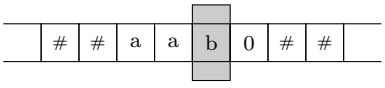
- $(q_2, 0) \mapsto (q_2, 0, L)$:



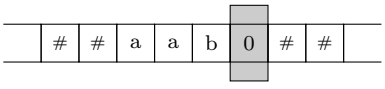
- $(q_2, a) \mapsto (q_0, a, R)$:



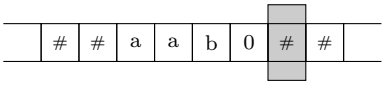
- $(q_0, 0) \mapsto (q_1, a, R)$:



- $(q_1, b) \mapsto (q_1, b, R)$:



- $(q_1, 0) \mapsto (q_1, 0, R)$:



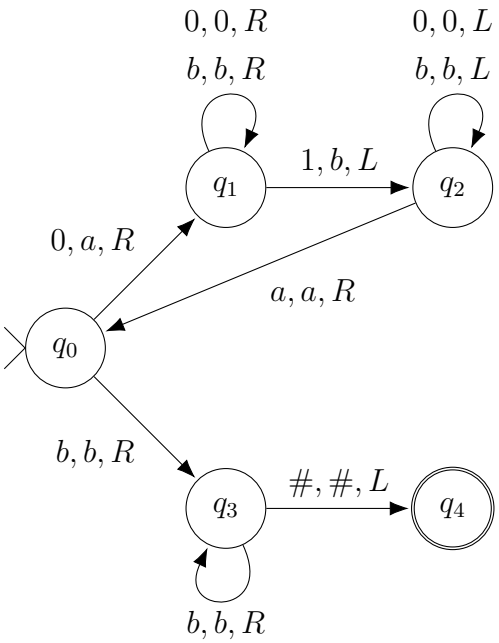
- $(q_1, \#) \mapsto \perp$

  $\Rightarrow$ TM hält an und $q_1 \notin F \Rightarrow$ Eingabe wird nicht akzeptiert

Turingmaschinen kann man auch in Diagrammdarstellungen visualisieren:

- Jeder Zustand wird durch einen Knoten symbolisiert.
- Der Startzustand erhält eine eingehende Kante ohne Quelle.
- Die Endzustände werden doppelt umrandet dargestellt.
- Gibt es zwischen zwei Zuständen q und q‘ einen Übergang mit δ (q, a) = (q‘, b, D), dann wird zwischen q und q‘ eine gerichtete Kante gezeichnet mit der Beschriftung „a, b, D“.

Als Beispiel betrachten wir die Turingmaschine aus Beispiel 5.1 visualisiert:



Im Folgenden betrachten wir die Sprachen, die Turingmaschinen *bearbeiten* können. Dabei werden wir feststellen, dass Turingmaschinen eine Klasse zwischen Chomsky–1 und Chomsky–0 akzeptieren können und die Chomsky–0 Sprachen *aufzählen* können.

5.2 Chomsky-1 und Chomsky-0 Sprachen

In diesem Abschnitt beschäftigen wir uns mit den Erkennungsautomaten zu den jeweiligen Sprachtypen. Wir haben bereits für Chomsky-3 die NEA/DEA und für Chomsky-2 die Kellerautomaten kennengelernt. Im Folgenden wenden wir uns nun Chomsky-1- und Chomsky-0-Sprachen zu.

Definition 5.1 (Abzählbare Menge)**.** Eine Menge M heißt abzählbar, falls es eine surjektive Funktion $f : \mathbb{N} \to M$ gibt oder falls $M = \emptyset$ ist.

Beispiel. Vergleiche Theo. Inf. 1:

- endliche Mengen
- $\mathbb{N}$ $\quad 0 \mapsto 0$
  $1 \mapsto 1$
  $2 \mapsto 2$
  $\dots$

- $\mathbb{Z}$

$0 \mapsto 0$

$1 \mapsto 1$

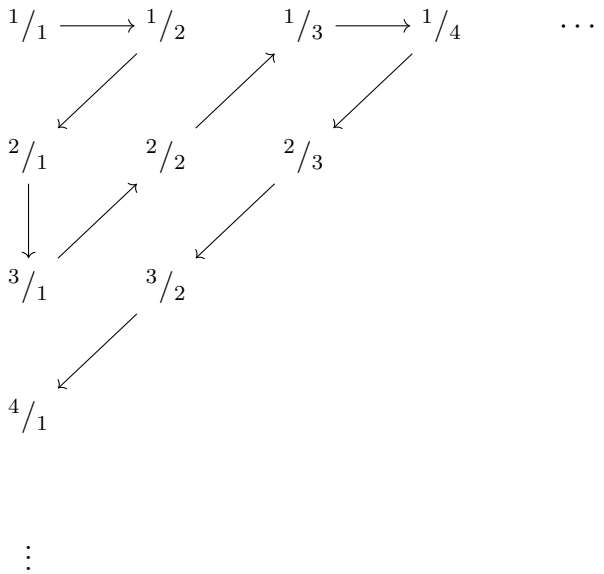
$2 \mapsto -1$

$3 \mapsto 2$

$4 \mapsto -2$

$\dots$

- $\mathbb{Q}$



Satz 5.2. Die Menge Σ^* über einem Alphabet Σ ist abzählbar.

Beweis. Zunächst definiert man eine totale Ordnung auf Σ (z.B. alphabetisch geordnet: $a < b < c < \dots < z$):

1. Man ordnet $w \in \Sigma$ der Länge nach
2. Wörter mit gleicher Länge ordnet man lexikographisch

Für jede Länge gibt es endlich viele Wörter. Man nummeriert sie durch und erhält so eine Abbildung $f : \mathbb{N} \to \Sigma^*$. □

Beispiel. $\Sigma = \{a, b\} \Rightarrow \Sigma^* = \{a, b, aa, ab, ba, bb, \dots\}$

Abbildung $f : \mathbb{N} \to \Sigma^*$:

$0 \mapsto \varepsilon$	$3 \mapsto aa$	$6 \mapsto bb$	$9 \mapsto aba$
$1 \mapsto a$	$4 \mapsto ab$	$7 \mapsto aaa$	$10 \mapsto abb$
$2 \mapsto b$	$5 \mapsto ba$	$8 \mapsto aab$	$\dots$

Satz 5.3. Jede Teilmenge $M' \subseteq M$ einer abzählbaren Menge ist selbst auch abzählbar.

Beweis. Sei $f : \mathbb{N} \to M$ eine surjektive Funktion. Dann sei $f' : \mathbb{N} \to M'$ definiert durch

$$f'(n) = \begin{cases} f(n) & \text{falls } f(n) \in M' \\ m \in M' \text{ beliebig} & \text{falls } f(n) \notin M' \end{cases}$$

f' ist surjektiv und M' damit abzählbar. □

Korollar 5.4. Sei $\mathcal{L} \subseteq \Sigma^*$ eine Sprache über dem Alphabet Σ, dann ist $\mathcal{L}$ abzählbar.

Definition 5.5 ((rekursiv) aufzählbar/semi-entscheidbar)**.** Eine Menge M heißt (rekursiv) aufzählbar/semi-entscheidbar, falls es eine rekursive und surjektive Funktion $f : \mathbb{N} \to M$ gibt oder $M = \emptyset$. (rekursiv $\hat{=}$ berechenbar/programmierbar, vgl. Definition 1.17)

Mit anderen Worten: Es gibt einen (rekursiven) Algorithmus, der $f(n)$ für jedes $n \in \mathbb{N}$ berechnet.

Da jede Sprache eine Menge ist, ist analog zu obigem auch eine *(rekursiv) aufzählbare Sprache* definiert.

Satz 5.6. Sei Σ ein Alphabet, dann ist Σ^* aufzählbar.

Beweis. Wir definieren eine Funktion, die Schrittweise alle Elemente von Σ^* erzeugt.

Sei $\Sigma = \{a_1, \ldots, a_n\}$ mit $a_1 < \cdots < a_n$, dann soll gelten:

$$\begin{array}{ll} f(0) = \varepsilon & \\ f(1) = a_1 & \\ \quad \vdots & \forall i \in \mathbb{N}: \\ & f(n+i+1) \text{ ist das in lexikografischer Ordnung auf } f(n+i) \\ f(n) = a_n & \text{folgende Element} \end{array}$$

□

Beispiel. $\Sigma = \{a, b, c\}$ mit $a < b < c$:

$$\begin{array}{llll} f(0) = \varepsilon & f(4) = aa & f(8) = bb & f(12) = cc \\ f(1) = a & f(5) = ab & f(9) = bc & f(13) = aaa \\ f(2) = b & f(6) = ac & f(10) = ca & f(14) = aab \\ f(3) = c & f(7) = ba & f(11) = cb & \ldots \end{array}$$

Anmerkung. Nicht jede $\mathcal{L} \subset \Sigma^*$ ist aufzählbar.

Definition 5.7 (entscheidbar)**.** Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ über dem Alphabet Σ heißt entscheidbar, wenn es eine rekursive charakteristische Funktion $\chi : \Sigma^* \to \{true, false\}$ mit

$$\chi(w) = \begin{cases} true & \text{falls } w \in \mathcal{L} \\ false & \text{falls } w \notin \mathcal{L} \end{cases}$$

gibt.

Anmerkung. Es ist leicht zu sehen, dass jede endliche Sprache entscheidbar ist.

Satz 5.8. Jede entscheidbare Sprache ist aufzählbar.

Beweis. Sei f die Funktion, die Σ^* aufzählt (vgl. Satz 5.6). So gilt für jede entscheidbare Sprache $\mathcal{L} \subseteq \Sigma^*$:
Es gibt eine Funktion $g : \mathbb{N} \to \mathcal{L}$, die alle Elemente von f der Reihe nach durchgeht und mit χ überprüft, ob das Element in $\mathcal{L}$ enthalten ist. Elemente, die nicht enthalten sind, werden ausgelassen. So können mithilfe von χ alle Elemente der Sprache aufgezählt werden. □

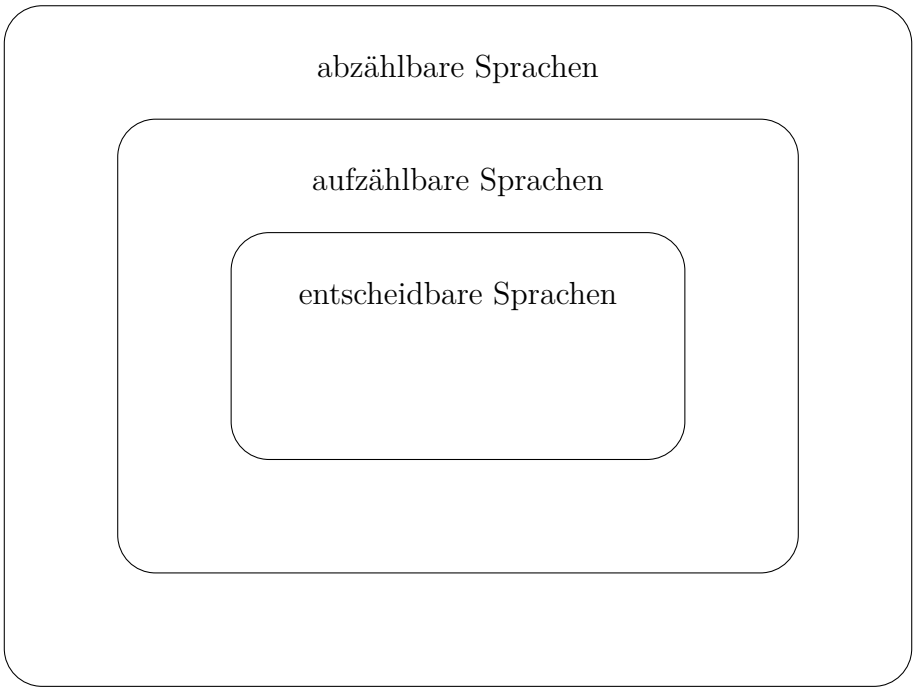


Abbildung 5.1: Zusammenhang zwischen entscheidbar, aufzählbar und abzählbar

Im Folgenden werden wir nun die Chomsky-Hierarchie ansehen und untersuchen, welche Sprachklassen welche Eigenschaften haben.

Satz 5.9. Eine Sprache ist genau dann von einer Grammatik erzeugbar, wenn sie rekursiv aufzählbar ist, d.h. alle von einer Grammatik erzeugbaren Sprachen können von einer Turingmaschine aufgezählt werden.

Beweis. Für den Beweis dieses Satzes benötigen wir die *Effektive Nummerierung* von $\mathbb{N}^2$ (s. Abb. 5.2).

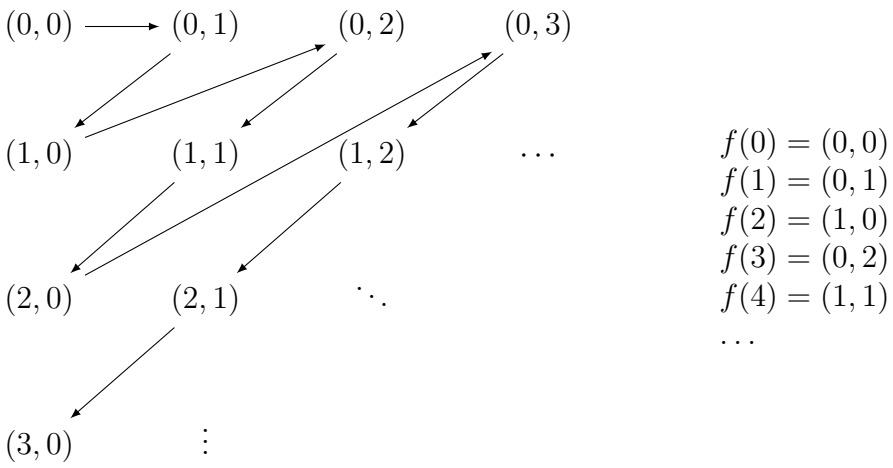


Abbildung 5.2: Effektive Nummerierung von $\mathbb{N}^2$

Man kann die Funktion f aufteilen in $f(n) = (l(n), r(n))$. Alle drei Funktionen f, l, r sind rekursiv. In diesem Beweis wird die effektive Nummerierung von $\mathbb{N}^2$ benötigt für:

$l(n)$: Auflistung aller Wörter die in $l(n)$ Schritten ableitbar sind.

$r(n)$: Das $r(n)$. Wort in der Liste von $l(n)$.

Sei $G = (N, \Sigma, \Pi, S)$ eine beliebige Grammatik. Wir konstruieren nun eine Turingmaschine, die eine total rekursive surjektive Funktion $\Phi : \mathbb{N} \to \mathcal{L}(G)$ berechnet, die $\mathcal{L}(G)$ aufzählt.

Sei $w \in \mathcal{L}(G)$ fest gewählt:

1. Berechne für die Eingabe n die Werte $l(n)$ und $r(n)$
2. Führe $l(n)$ Schritte durch $(0 \leq i \leq l(n))$:
   0. Schritt: Schreibe S auf das Band
   $i+1$. Schritt: Zu allen in Schritt i auf das Band geschriebenen Wörter schreibe sämtliche direkten Ableitungen auf das Band. Da Π endlich ist, sind dies endlich viele Neubildungen. Lösche alle im Schritt i geschriebenen Wörter.

3. Sind mindestens $r(n)+1$ Wörter entstanden, lösche alle Wörter bis auf das $(r(n)+1)$. Wort. Andernfalls gehe zu 5.

4. Besteht das Wort u auf dem Band nur aus Terminalen ($u \in \Sigma^*$ und $u \in \mathcal{L}(G)$, da aus G abgeleitet), so ist u das Ergebnis und die Turingmaschine hält an. Andernfalls gehe zu 5.

5. Schreibe w auf das Band und die Turingmaschine hält an

□

Bevor wir die Rückrichtung betrachten, geben wir ein Beispiel an:

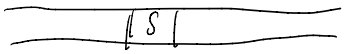
Beispiel. Als Beispiel für eine Turingmaschine, die aus einer Grammatik vom Typ-0 erzeugt wird, betrachten wir folgende Grammatik:

$G = (\{S, X, Y\}, \{0, 1\}, \Pi, S)$
$\Pi = \{$
$\quad S \to XY,$
$\quad X \to 1X \mid Y,$
$\quad Y \to 0Y \mid 0$
$\}$

Anmerkung. $\mathcal{L}(G) = \mathcal{L}(1^*000^*)$ ist eine Sprache vom Typ-3

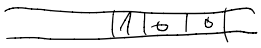
$w = 100$

$n > 0$: 1. $\ell(0) = 0$, $r(0) = 0$

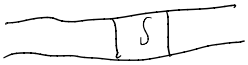
2. 

3. nichts zu löschen

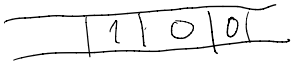
4. $S \notin \Sigma^*$

5. 

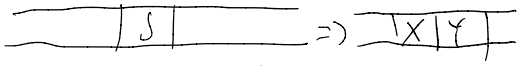
$n = 1$ 1. $\ell(1) = 0$, $r(1) = 1$

2. 

3. Es sind keine $r(1) + 1 = 2$ Worte entstanden

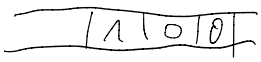
5. 

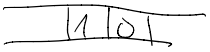
$n = 2$ 1. $\ell(2) = 1$, $r(2) = 0$

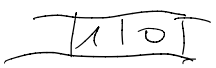
2. 

3. nichts zu löschen

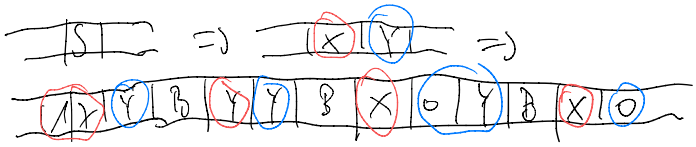
4. $XY \notin \Sigma^*$

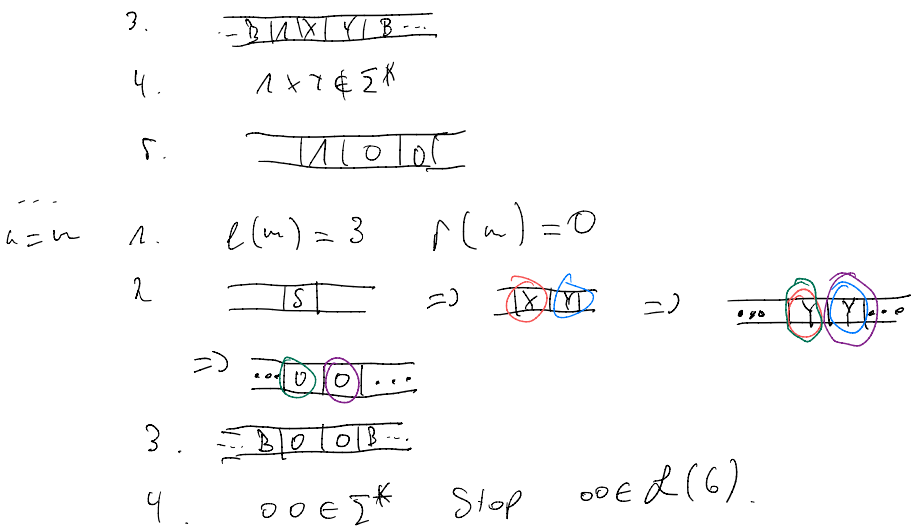
5. 

$n = 3$ 1. $\ell(3) = 0$, $r(3) = 2$ $\Rightarrow$ 

$n = 4$ 1. $\ell(4) = 1$, $r(4) = 1$ $\Rightarrow$ 

$n = 5$ 1. $\ell(5) = 2$, $r(5) = 0$

2. 



Rückrichtung:

Eine Menge $M \subseteq \Sigma^*$ wird durch eine Turingmaschine $TM = (Q, \Sigma_{TM}, \Gamma, \delta, q_0, B, \{q_E\})$ aufgezählt, d.h. TM berechnet eine surjektive Funktion $\Phi : I \to M$, wobei $I \subseteq \Sigma_{TM}^*$ mit $\Sigma_{TM} \subseteq \{0, \ldots, 9\}$. Weiterhin gilt: $\Gamma = \Sigma \cup \{B\} \cup \Sigma_{TM}$, $\Gamma \cap Q = \emptyset$ und nach der Berechnung befindet sich der LSK der TM immer auf dem ersten Symbol des Ergebnisses.

Im Folgenden wird die Grammatik G mit $\mathcal{L}(G) = M$ erzeugt:

$$G = (Q \dot\cup (\Gamma \setminus \Sigma) \dot\cup \{S, L, P, A\}, \Sigma, \Pi, S)$$

mit folgenden Produktionen:

(1) $S \to LAP$,
$A \to q_0$,
$A \to Ax, \qquad \forall x \in \Sigma_{TM}$

(Damit wird die Anfangskonfiguration Lq_0pP mit $p \in \Sigma_{TM}^*$ der TM erzeugt)

(2) Sei $Q' = Q \setminus \{q_E\}$, dann $\forall q \in Q'$:
$qx \to q'y$ für $\delta(q, x) = (q', y, N)$
$qx \to yq'$ für $\delta(q, x) = (q', y, R)$
$x'qx \to q'x'y$ $\forall x' \in \Gamma$ und $\delta(q, x) = (q', y, L)$

(Der LSK wird durch den Zustand q bzw. q' links von dem Zeichen repräsentiert.)

(3) $qP \to qBP \qquad \forall q \in Q'$

$Lq \to LBq \qquad \forall q \in Q'$

(Einfügen von Blanks am rechten und linken Rand; wie beim Band der TM)

(4) $Bq_E \to q_E$

$Lq_E \to q_E$

$q_E x \to x q_E \qquad \forall x \in \Sigma$

$q_E B \to q_E$

$q_E P \to \varepsilon$

(Zeichenketten der Form $LB \dots Bq_E wB \dots BP$ mit $w \in M$ werden auf w reduziert $\to B, L, P, q_E$ löschen)

Beispiel. Sei folgende Turingmaschine gegeben:

$TM = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{1\}, \{a, 1, B\}, \delta, q_0, B, \{q_5\})$

$\delta = \{$

$(q_0, 1) \mapsto (q_1, a, L)$

$(q_1, B) \mapsto (q_2, a, R)$

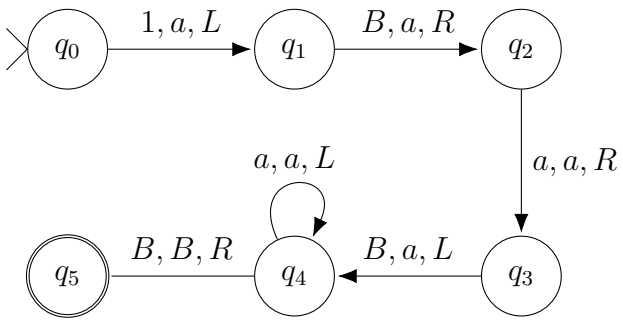
$(q_2, a) \mapsto (q_3, a, R)$

$(q_3, B) \mapsto (q_4, a, L)$

$(q_4, a) \mapsto (q_4, a, L)$

$(q_4, B) \mapsto (q_5, B, R)$

$\}$



Die Turingmaschine zählt nur das Wort aaa auf.
Umwandlung in eine Grammatik:

$G = (\{q_0, q_1, q_2, q_3, q_4, q_5\} \cup \{1, B\} \cup \{S, L, P, A\}, \{\, a \,\}, \Pi, S)$

$\Pi = \{$

$S \to LAP$ (1)
$A \to q_0$ (2)
$A \to A1$ (3)
} (1)

$q_1 B \to a q_2$ (4)
$q_2 a \to a q_3$ (5)
$q_4 B \to B q_5$ (6)
} R

$a q_0 1 \to q_1 a a$ (7)
$1 q_0 1 \to q_1 1 a$ (8)
$B q_0 1 \to q_1 B a$ (9)
$a q_3 B \to q_4 a a$ (10)
$1 q_3 B \to q_4 1 a$ (11)
$B q_3 B \to q_4 B a$ (12)
$a q_4 a \to q_4 a a$ (13)
$1 q_4 a \to q_4 1 a$ (14)
$B q_4 a \to q_4 B a$ (15)
} L } (2)

$q_i P \to q_i B P \quad i \in \{0 \dots 4\}$ (16)
$L q_i \to L B q_i \quad i \in \{0 \dots 4\}$ (17)
} (3)

$B q_5 \to q_5$ (18)
$L q_5 \to q_5$ (19)
$q_5 a \to a q_5$ (20)
$q_5 B \to q_5$ (21)
$q_5 P \to \varepsilon$ (22)
} (4)

$\}$

Ableitung von S:

$$S \xrightarrow{(1)} LAP \xrightarrow{(3)} LA1P \xrightarrow{(2)} Lq_0 1P \xrightarrow{(17)} LBq_0 1P \xrightarrow{(9)} Lq_1 BaP \xrightarrow{(4)} Laq_2 aP \xrightarrow{(5)}$$
$$Laaq_3 P \xrightarrow{(16)} Laaq_3 BP \xrightarrow{(10)} Laq_4 aaP \xrightarrow{(13)} Lq_4 aaaP \xrightarrow{(17)} LBq_4 aaaP \xrightarrow{(15)}$$
$$Lq_4 BaaaP \xrightarrow{(6)} LBq_5 aaaP \xrightarrow{(18)} Lq_5 aaaP \xrightarrow{(19)} q_5 aaaP \xrightarrow{(20)^3} aaaq_5 P \xrightarrow{(22)} aaa$$

Satz 5.10 (Zusammenhang zu Chomsky–0 Grammatiken)**.** Für jede rekursive Funktion f existiert eine Chomsky–0 Grammatik, deren Ableitungen der Berechnung der Funktion entspricht.

Beweis. Nach dem Hauptsatz dem Algorithmentheorie (Satz 1.25) gibt es eine Turingmaschine

$$TM = (Q, \{\,0, 1\,\}, \{\,0, 1, \#\,\}, \delta, q_0, \#, F)$$

die f berechnet. Sei weiter

$$TM' = (Q', \{\,0, 1\,\}, \{\,0', 1', 0, 1, \#\,\}, \delta', q_0', \#, F')$$

eine erweiterte Turingmaschine, die TM entspricht und am Ende alle Zeichen 0 auf dem Band durch $0'$ und alle Zeichen 1 durch $1'$ ersetzt, und den LSK auf das Zeichen ganz links verschiebt. Es ist offensichtlich, dass TM und TM' äquivalente Funktionen berechnen.

Im Beweis von Satz 5.9 wird aus einer gegebenen Turingmaschine, die eine Sprache aufzählt, eine Chomsky–0 Grammatik

$$G = (Q' \dot\cup \{\,0, 1, \#\,\} \dot\cup \{S, L, P, A\}, \{\,0', 1'\,\}, \Pi, S)$$

der Sprache angegeben. Die Ableitungen dieser Grammatik entsprechen also der Berechnungen der Funktion f. □

Satz 5.11 (Satz von Chomsky)**.** Jede Sprache einer nicht-verkürzenden Grammatik (Typ 1b) ist rekursiv-entscheidbar.

Beweis. Sei $\mathcal{L} \subseteq \Sigma^*$ eine Sprache einer nicht-verkürzenden Grammatik $G = (N, \Sigma, \Pi, S)$. Wir geben eine Turingmaschine an, die $\mathcal{L}$ entscheidet:

1. w steht als Eingabe auf dem Band
2. Länge von w wird bestimmt
3. Alle Ableitungen aus S bis zu einer Länge von $|w| + 1$ werden erzeugt (terminiert, weil G nicht-verkürzend ist)
4. a) Wenn w im 3. Schritt erzeugt wurde, gibt die Turingmaschine 1 ($true$) aus

   b) Wenn w im 3. Schritt nicht erzeugt wurde, dann wird w auch durch weitere Ableitungen nicht mehr erzeugt (weil G nicht-verkürzend ist) und die Turingmaschine gibt 0 ($false$) aus

□

Satz 5.12. Nicht jede rekursiv-entscheidbare Sprache ist vom Typ 1b.

Beweis. Sei $G_0, G_1, \ldots$ eine effektive Nummerierung aller nicht-verkürzender Grammatiken (surjektive rekursive Abbildung: $\mathbb{N} \to \{G \mid G \text{ ist Grammatik vom Typ 1b}\}$) über einem Alphabet Σ. Dann sei $\mathcal{L}_0, \mathcal{L}_1, \ldots$ eine effektive Nummerierung aller entsprechenden Sprachen vom Typ 1b. Sei nun:

$$\mathcal{L} = \{w \mid w \in \Sigma^* \wedge w \notin \mathcal{L}_{|w|}\}$$

Wir zeigen nun, dass $\mathcal{L}$ zwar rekursiv-entscheidbar, aber nicht vom Typ 1b ist.

1. $\mathcal{L}$ ist rekursiv-entscheidbar:
   Aus $|w|$ folgt, dass $\mathcal{L}_{|w|}$ bestimmbar ist, da $\mathcal{L}_i$ effektiv nummeriert ist. $\mathcal{L}_{|w|}$ ist eine Sprache vom Typ 1b, somit ist nach Satz 5.11 $\mathcal{L}_{|w|}$ rekursiv entscheidbar. Wir können also entscheiden, ob $w \in \mathcal{L}_{|w|}$ liegt, somit ist auch $\mathcal{L}$ rekursiv-entscheidbar, weil auch $w \notin \mathcal{L}_{|w|}$ entscheidbar ist.

2. $\mathcal{L}$ ist nicht vom Typ 1b:
   Wir zeigen, dass es kein i gibt mit $\mathcal{L} = \mathcal{L}_i$:
   Angenommen es gäbe ein i mit $\mathcal{L} = \mathcal{L}_i$, dann existiert ein w mit $|w| = i$. Nun gibt es 2 Möglichkeiten:

   a) $w \in \mathcal{L}$: Dann gilt nach Definition von $\mathcal{L}$: $w \notin \mathcal{L}_i$

   b) $w \notin \mathcal{L}$: Dann gilt nach Definition von $\mathcal{L}$: $w \in \mathcal{L}_i$

   Beide Fälle sind ein Widerspruch zur Annahme $\mathcal{L} = \mathcal{L}_i$

□

Satz 5.13. Die Menge der Sprachen vom Typ 1a und Typ 1b sind identisch.

Beweis. siehe z.B. [Winter, Satz 4.7] □

Daraus ergibt sich, dass die Sätze 5.11 und 5.12 analog auch für Sprachen kontextsensitiver Grammatiken (Typ 1a) gültig sind.

Satz 5.14. Jede Sprache einer kontextsensitiven Grammatik ist rekursiv-entscheidbar.

Korollar 5.15. Nicht jede rekursiv-entscheidbare Sprache ist Sprache einer kontextsensitiven Grammatik (Typ 1a). (analog zu Satz 5.12)

Definition 5.16 (linear-beschränkte Turingmaschine (LBTM))**.** Eine Turingmaschine heißt linear beschränkt, wenn sie nur den Speicherplatz benötigt, den die Eingabe belegt hat.

Beispiel. $\mathcal{L} = \{0^n 1^n 2^n\}$

Wir geben nun eine Turingmaschine an, die $\mathcal{L}$ erkennt. Idee:

1. Lösche die erste 0
2. Überschreibe die letzte 1 mit 2
3. Lösche die letzten beiden 2er

Anmerkung. Bisher haben wir nur nicht zwischen deterministischen und nichtdeterministischen Turingmaschinen unterschieden, weil die Menge der berechenbaren Funktionen sich für allgemeine Turingmaschinen nicht unterscheiden. Bei LBTMs ist dies bisher nicht bekannt. Deshalb kann folgender Satz nur für nicht-deterministische LBTMs bewiesen werden.

Satz 5.17 (Kuroda, Weber 1964)**.** Eine Sprache ist genau dann kontextsensitiv, wenn sie von einer nicht-deterministischen linear-beschränkten Turingmaschine akzeptiert wird.

Beweis. Sei $G = (N, \Sigma, \Pi, S)$ eine nicht-verkürzende Grammatik. Wir konstruieren nun eine nicht-deterministische LBTM:

$$T = (\{q_0, \ldots, q_E\}, \Sigma, \Gamma = \Sigma \cup N \cup \{B, *\}, \delta, q_0, B, \{q_E\})$$

Die Konstruktion der Relation δ wird durch die folgenden Schritte beschrieben:

1. Steht $w \in (\Sigma \cup N)^*$ auf dem Band, dann gehe zu 2., sonst zu 6.
2. Steht für $(p \to q) \in \Pi$ wqw' auf dem Band, gehe zu 3., sonst zu 5. (nicht-deterministisch bei der Bestimmung der Produktion)
3. q wird durch $p*^{|q|-|p|}$ ersetzt (z.B. $q \to p * **$, wenn $|q| - |p| = 3$)
4. $*$ werden gelöscht und die anderen Bandsymbole zusammengeschoben. Neues Wort auf dem Band wird mit w bezeichnet. Gehe zu 2.
5. Ist $w = S$, ersetze S durch 1 ($true$), gehe zu q_E und stoppe.
6. Das Band von T wird gelöscht und mit 0 ($false$) ersetzt. Gehe zu q_E und stoppe.

T ist beschränkt, da G nicht-verkürzend ist, d.h für $(p \to q) \in \Pi$ gilt $|q| - |p|$. Nach Konstruktion akzeptiert T die Sprache $\mathcal{L}(G)$.

Rückrichtung:

Prinzipiell wird ähnlich vorgegangen wie im Beweis von Satz 5.9, allerdings ist bei der Konstruktion der Grammatik zu beachten, dass sie nicht-verkürzend wird.

Für den konkreten Beweis wird auf [Winter, Satz 4.13] verwiesen. □

Beispiel. Sei $G = (\{S, B\}, \{0, 1, 2\}, \Pi, S)$ mit

$$\Pi = \{$$
$$\quad S \to 0SB2 \mid 012,$$
$$\quad 2B \to B2,$$
$$\quad 1B \to 11$$
$$\}$$

gegeben.

Ablauf der LBTM mit der Eingabe 001122:

001122	
$001B22$	
$0012B2$	$0012B2$
$00122B$	$0S^{**}B2$
$0S^{**}2B$	$0SB2$
$0S2B$ ↯	S^{***}
	S ✓

$\mathcal{L} = \{0^n 1^n 2^n\}$

Zusammenhang der Sprachtypen und der Chomsky-Hierarchie mit den Automaten

Chomsky-Sprachtyp	Beschreibung		Automat	Algorithmus	Tool
Typ-0	allgemein	$\overset{5.9}{\Leftrightarrow}$	TM (nur aufzählbar, nicht entscheidbar)		
	$\cup$ (5.12, 5.15)				
	entscheidbare Sprachen	$\overset{\text{per Definition}}{\Leftrightarrow}$	TM		
Typ-1	nicht-verkürzend		NDLBTM		
	$\Updownarrow$				
	kontextsensitiv	$\overset{5.13/5.17}{\Leftrightarrow}$	NDLBTM		
Typ-2	kontextfrei	$\overset{4.7}{\Leftrightarrow}$	NPDA		
			$\cup$		
			DPDA	LR(k) mit k $\geq$1	
			$\Leftrightarrow$		
				$\cup$	
				LALR	JAY/JAOOY
				$\cup$	
				SLR	
Typ-3	regulär	$\overset{3.13}{\Leftrightarrow}$	DEA $\overset{3.13}{\Leftrightarrow}$ NEA		JLEX