Formelsammlung zur Informatik

Jan Krieger

24. September 2006

'The time has come,' the Walrus said,
'To talk of many things:
Of shoes – and ships – and sealingwax –
Of cabbages – and kings –
And why the sea is boiling hot –
And whether pigs have wings.'

– Tweedldee in Through the Looking-Glass by Lewis Carrol

Inhaltsverzeichnis

1 Einleitung **6**

I Theoretische Informatik und Grundlagen **7**

2 Grundlagen **8**
2.1 Grundlegende Definitionen aus der Mathematik . . . 8
2.2 Landau-Notation . . . 9
2.3 Gödelisierung/Kodierungsfunktionen . . . 9
2.4 Zahlendarstellungen . . . 11
2.4.1 Festkommadarstellungen . . . 11
2.4.2 Gleitkommadarstellungen . . . 12
2.5 Einige Rekursive Beziehungen . . . 13

3 Formale Sprachen **14**
3.1 Alphabete, Wörter, Sprachen . . . 14
3.2 CHOMSKY-Grammatiken und -Sprachen . . . 15
3.3 Schreibweisen und Darstellungen . . . 17
3.3.1 Erweiterte Backus-Naur-Form (EBNF) . . . 17
3.3.2 Syntaxdiagramme . . . 17
3.3.3 Ableitungsbäume . . . 18
3.3.4 Reguläre Ausdrücke . . . 19
3.4 Chomsky-Hirarchie . . . 20
3.5 Eigenschaften von Sprachen und Grammatiken . . . 24
3.6 CHOMSKY-Grammatiken und Berechenbarkeit . . . 25
3.7 Transformation von Grammatiken . . . 27
3.7.1 Reduktion . . . 27
3.7.2 Beseitigung von λ-Ableitungen . . . 29
3.7.3 Beseitigung von Links-Rekursion . . . 29
3.7.4 Linksfaktorisierung . . . 30

4 Berechenbarkeitstheorie und Äquivalenzsatz **32**
4.1 Algorithmenbegriff und Berechenbarkeit . . . 32
4.2 Mathematische Maschinen . . . 34
4.3 TURING-Maschinen . . . 37
4.3.1 Grundversion der TURING-Maschine . . . 37
4.3.2 Berechenbarkeitsbegriffe und TURING-Maschinen . . . 40
4.3.3 TURING-Operatoren . . . 41
4.3.4 Varianten der Bandstruktur von TURING-Maschinen . . . 43
4.3.5 Universelle Turingmaschinen . . . 45
4.3.6 Nichtdeterministische Turingmaschinen . . . 46
4.4 Registermaschinen . . . 46
4.4.1 Registermaschinen . . . 46
4.4.2 Registeroperatoren . . . 47
4.5 rekursive und primitv rekursive Funktionen . . . 47

4.6 Äquivalenzsatz . . . 51
4.7 Unentscheidbare Mengen, Diagonalisierungsmethode, Reduktionsmethode . . . 51
4.8 Rekursiv aufzählbare Mengen . . . 52
4.9 Weitere Ergebnisse der Berechenbarkeitstheorie . . . 53

5 Komplexitätstheorie **55**
5.1 Zeit- und Platzkomplexität von Turingmaschinen . . . 55
5.2 Hierarchiesätze . . . 56
5.3 Komplexität nicht-deterministischer Turingmaschinen . . . 57
5.4 Polynomiell beschränkte Komplexitätsklassen und das P-NP-Problem . . . 57

6 Automatentheorie **60**
6.1 Grundtypen von Automaten . . . 60
6.1.1 einfache Automaten . . . 60
6.1.2 Kellerautomaten/Stackautomaten . . . 61
6.2 Automaten und Sprachen . . . 63

7 SHANNON'SCHE Informationstheorie **66**

II Algorithmen und Datenstrukturen **67**

8 Elementare Datenstrukturen **68**
8.1 Bäume . . . 68
8.2 Prioritätswarteschlangen (PQ) . . . 69
8.2.1 Heaps . . . 69
8.2.2 Binomiale Heaps . . . 71
8.3 Graphen . . . 73
8.3.1 Abspeichern eines Graphen . . . 73
8.3.2 Durchlaufen eines Graphen . . . 74

9 Elementare Algorithmen **75**
9.1 Union-Find-Algorithmen . . . 75
9.2 Algorithmen für Bäume . . . 76
9.2.1 Traversieren von Binärbäumen . . . 76
9.2.2 Zählen der Elemente in einem Binärbaum . . . 76
9.2.3 Tiefe eines Binärbaumes Baum . . . 76
9.2.4 Baum-Isomorphie nach Aho/Hopcroft/Ullman . . . 77
9.3 Kompressionsalgorithmen . . . 77
9.3.1 Huffman Kodierung . . . 77
9.3.2 Lempel-Ziv-Welch . . . 78
9.4 Elementare Sortierverfahren . . . 80
9.4.1 Selection-Sort – Sortieren durch Auswahl . . . 80
9.4.2 Insertion-Sort – Sortieren durch Einfügen . . . 81
9.4.3 Bubble-Sort . . . 81
9.5 Komplexe Sortierverfahren . . . 82
9.5.1 Quicksort . . . 82
9.5.2 Mergesort . . . 83
9.5.3 Heapsort . . . 85
9.6 Suchmethoden . . . 86
9.6.1 Binäre Suche . . . 86
9.6.2 Binärer Suchbaum . . . 86
9.6.3 Randomisierte Binäre Suchbäume . . . 90
9.6.4 2-3-4-Bäume und Rot-Schwarz-Bäume . . . 90

9.6.5 Digitale Suchbäume (digital search trees) 92
9.6.6 Digital Search Tries 92
9.6.7 Hashing 93

III Compilerbau **95**

10 Aufbau eines Compilers **96**

11 Lexikalische Analyse **99**

12 Syntaxanalyse **102**
12.1 Top-Down-Syntaxanalyse 103
12.1.1 Start- und Folgemengen 104
12.1.2 LL(1)-Grammatiken 105
12.2 Bottom-Up-Syntaxanalyse 109
12.2.1 Das Shift-Reduce-Prinzip 109
12.2.2 Handles und Präfixe 110
12.2.3 Operator-Präzedenz-Parser 112
12.2.4 Charakteristische Automaten und LR(0)-Analyse 114

13 Syntaxgesteuerte Übersetzung **117**

14 Semantische Analyse **119**
14.1 Typausdrücke 119
14.2 Symboltabelle 120

15 Zwischencodeerzeugung **122**
15.1 C-Maschine 122

16 3-Adress-Code **123**

A Glossar **124**

1 Einleitung

Die Abschnitte über **theoretische Informatik** (Formale Sprachen, Berechenbarkeit, Komplexitätstheorie...) basiert im wesentlichen auf der Vorlesung bei Prof. Ambos-Spies, die ich im SS04 an der Universität Heidelberg besucht habe. Auf der folgenden Website zur Vorlesung gibt es ein sehr gutes und ausführliches Skript:

http://math.uni-heidelberg.de/logic/SS04/thinf_SS04.html

Die Abschnitte über **Algorithmen und Datensturkturen** basieren auf der Vorlesung von Prof. Hamprecht (Algorithmen und Datenstrukturen, Heidelberg, SS05) und dem Buch:

Sedgewick, Robert (1991): „Algorithmen“, Bonn-München-Reading (Mass.), Addison-Wesley

Da ich im SS2006 die Vorlesung **Compilerbau** bei Prof. Reinelt (Uni Heidelberg) besucht habe, finden sich auch einige Informationen zu diesem Gebiet in dieser Stoffzusammenfassung. Da Compilerbau stark auf formalen Sprachen und Automaten zurückgreift wurden die enstrpechenden Abschnitte erweitert. Die Informationen sind also etwas verstreut im Skript zu finden. Die hier zusammengefassten Informationen basieren auf dem Skript von Herrn Reinelt, das leider nicht frei verfügbar ist und dem Buch

Aho, A.V., Sethi, R., Ullmann, J.D. (1999): „Compilerbau Teil 1, 2“, München-Wien,

Wahrscheinlich ist dieses Skript vor Allem für Höhrer der obigen Vorlesungen zu gebrauchen, da es zwar den Inhalt der zugehörigen Skripte/Bücher zusammenfasst aber natürlich nicht so ausführlich ist. An diversen Stellen werden aber Dinge, die mir unverständlich waren anschaulich erklärt. Das Skript eignet sich also sowohl zur Vorlesungsbegleitung, als auch besonders zur Prüfungsvorbereitung (ist ja auch als solche entstanden ;-)

Teil I

Theoretische Informatik und Grundlagen

2 Grundlagen

2.1 Grundlegende Definitionen aus der Mathematik

Potentzmenge: Die Potenzmenge $\mathcal{P}(S) \equiv 2^S$ zu einer Menge S bezeichnet die Menge aller möglichen Teilmengen von S. Die leere Menge $\{\}$ ist teilmenge jeder Potenzmenge, also $\{\} \in \mathcal{P}(S)$. *Beispiel: Man betrachte die Menge $S = \{A, B, C\}$. Es gilt dann:*

$$\mathcal{P}(S) = \big\{\{\}, \{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\big\}$$

Relationen: Relationen beschreiben Beziehungen zwischen den Elementen einer oder verschiedener Mengen. Eine n-stellige Relation von Mengen $A_1, ..., A_n$ ist eine Teilmenge des kartesischen Produktes $A_1 \times ... \times A_n$ der Mengen. Sind alle Mengen gleich, so heißt eine Menge $R \subset A^n$ Relation in der Menge A.
Für **zweistellige (binäre) Relationen** R über einer Menge A gilt: $R \subset A^2$. Man schreibt oft statt der Postfixnotation $(a, b) \in R$ die sog. Infixnotation aRb. Anschaulich ordnen Relationen jedem Element der Menge A ein anderes Element der Menge A zu.
Binäre Relationen müssen nicht eindeutig sein, d.h. eine Relation R kann einem $c \in A$ durchaus mehrere $c', c'', ... \in A$ zuordnen.
Als Beispiel kann man etwa die Teilbarkeitsbeziehung T notieren:

$$T = \big\{(a, b) \big| a, b \in A \quad \text{und} \quad a \text{ teilt } b\big\}$$

Prädikate: Ein Prädikat $P \subseteq \mathbb{N}$ ist die Menge der Zahlen, auf die eine gewisse Eigenschaft zutrifft. Man schreibt:

$$x \in P \quad \Leftrightarrow \quad P(x)$$

Graph einer Funktion: Sei $f : A \to B$ eine beliebige Funktion (evtl. auch partiell). Dann definiert man ihren raphen G_f als Menge:

$$G_f := \big\{\big(x, f(x)\big) \big| x \in \mathrm{Db}(f)\big\}$$

Eingeschränkte Sutraktion: Für Zahlen $a, b \in \mathbb{N}$ definiert man die eingeschränkte Subtraktion $\dot{-} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$:

$$n \dot{-} k := \begin{cases} n - k & \text{falls } k \leq n \\ 0 & \text{sonst} \end{cases}$$

Stelligkeit einer Funktion: Eine Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt n-stellig und man schreibt: $f^{(n)}(\vec{x}) = f^{(n)}(x_1, ..., x_n)$.

2.2 Landau-Notation

- **obere Grenze:** Eine Funktion $g(N)$ ist aus der Klasse $O(f(N))$, wenn es Konstanten c, N_0 gibt, mit:
$$g(N) \leq c \cdot f(N) \quad \forall N \geq N_0$$
- **untere Grenze:** Eine Funktion $g(N)$ ist aus der Klasse $\Omega(f(N))$, wenn es Konstanten c, N_0 gibt, mit:
$$g(N) \geq c \cdot f(N) \quad \forall N \geq N_0$$
- **Wachstumsrate:** Eine Funktion $g(N)$ ist aus der Klasse $\Theta(f(N))$, wenn es Konstanten c_0, c_1, N_0 gibt, mit:
$$c_0 \cdot f(N) \leq g(N) \leq c_1 \cdot f(N) \quad \forall N > N_0$$

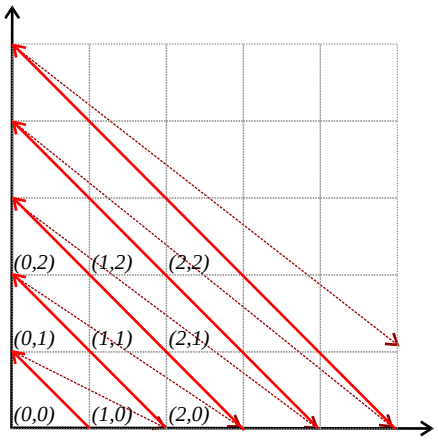
2.3 Gödelisierung/Kodierungsfunktionen

Bei der Gödelisierung geht es darum beliebige endliche Folgen (oder Tupel) natürlicher Zahlen in eine Zahl bijektiv zu kodieren. Ein einfaches Beispiel ist eine Durchnummerierung der Zahlen aus $\mathbb{Q}^+ = \mathbb{N} \times \mathbb{N}$ mit Zahlen aus $\mathbb{N}$.
Insgesamt geht es darum ganze Zahlenfolgen in einzelnen Zahlen zu kodieren, um Probleme auf Zahlenfolgen auf 'einfache' arithmetische Probleme zurückzuführen.

Paarfunktion $\tau : \mathbb{N} \times \mathbb{N} \xrightarrow{\text{bijektiv}} \mathbb{N}$:

$$\tau(x, y) := \left(\sum_{i<x+y} |D_i| \right) + y = \frac{(x+y)(x+y+1)}{2} + y.$$

Dabei ist $|D_i|$ die Anzahl der Paare auf der i-ten Diagonale in folgender Abbildung:



Es gilt:

- τ ist bijektiv und monoton
- τ ist primitiv rekursiv
- Die Umkehrfunktionen (Projektionen) $\pi_1(\tau(x, y)) = x$ und $\pi_2(\tau(x, y)) = y$ sind ebenfalls primitiv rekursiv

Mit der Paarfunktion τ kann man also Zahlentupel $(x, y) \in \mathbb{N} \times \mathbb{N}$ in Zahlen aus $\mathbb{N}$ ein-eindeutig kodieren. Mit den Projektionen π_1, π_2 kann man dann die zwei Zahlen aus der Kodierung wieder extrahieren.

Kodierung von n-Tupeln: Dazu definiert man die Kodierungsfunktion τ_n rekursiv für jedes feste $n \geq 1$:

- $\tau_n : \mathbb{N}^n \to \mathbb{N}$ bijektiv
- $\tau_1(x) := x$
- $\tau_{n+1}(x_1, x_2, ..., x_{n+1}) := \tau(x_1, \tau_n(x_2, ..., x_{n+1}))$

Es gilt:

- τ_n bijektiv und Monoton
- τ_n ist primitiv rekursiv
- Die Projektionen $\pi_i^n(\tau_n(x_1, ..., x_n)) = x_i$ ist ebenfalls primitiv rekursiv.

Kodierung von beliebigen endlichen Folgen: Man definiert die Kodierungsfunktione τ^*, die diese Durchnummerierung erledigt:

- $\tau^* : \mathbb{N}^* \to \mathbb{N}$ ist bijektiv, $\mathbb{N}^* := \bigcup_{n \geq 0} \mathbb{N}^n$
- $\tau^*(\lambda) := 0$, λ:leere Folge
- $\tau^*(x_1, ..., x_n) := \tau(n \dot{-} 1, \tau_n(x_1, ..., x_n)) + 1$

Schreibweisen:

- $\tau^*(x_1, ..., x_n) = \langle x_1, ..., x_n \rangle$
- $\pi(x, i) = (x)_i$
- $\pi(\pi(x, i), j) = (x)_{i,j}$

Hilfsfunktionen zur Gödelisierung: Folgende Funktionen sind primitiv rekursiv:

- **Länge:**
$$l(\tau^*(\vec{x})) := |\vec{x}| = \text{Länge der Folge } \vec{x}$$
- **Kopf:**
$$\text{head}(\tau^*(\vec{x})) := \begin{cases} x_1 & \text{falls } \vec{x} = (x_1, ..., x_n), n \geq 1 \\ 0 & \text{sonst} \end{cases}$$
head gibt das erste Element eines kodierten Vektors zurück
- **Schwanz:**
$$\text{tail}(\tau^*(\vec{x})) := \begin{cases} \tau^*(x_2, ..., x_n) & \text{falls } \vec{x} = (x_1, ..., x_n), n \geq 2 \\ 0 & \text{sonst } (l(\vec{x}) \leq 1) \end{cases}$$
tail gibt alle Elemente ab dem zweiten eines kodierten Vektors zurück
- **Projektion:**
$$\pi^*(\tau^*(\vec{x}), i) := \begin{cases} x_i & \text{falls } 1 \leq i \leq l(\vec{x}) \\ 0 & \text{sonst} \end{cases}$$
π^* gibt ein beliebiges Element eines kodierten Vektors zurück
- **Konkatenation:**
$$\circ(\tau^*(\vec{x}), \tau^*(\vec{x})) = \tau^*(\vec{x}) \circ \tau^*(\vec{y}) := \tau^*(\vec{x}, \vec{y}) = \tau^*(x_1, ..., x_n, y_1, ..., y_m)$$
Die Konkatenation gibt die Gödelnummer des Hinterinanderhängung zweier Vektoren/Folgen zurück

Kodierung von Wörtern über einem Alphabet: Grundsätzlich kann man jedem Zeichen a eines endlichen Alphabets Σ eine Zahl aus $n \in \mathbb{N}$ zuordnen. Also lässt sich ein Wort $a_1...a_k \in \Sigma^*$ auch durch das k-Tupel der zugeordneten Zahlen $(n_1, ..., n_k) \in \mathbb{N}^k$ repräsentieren. Dieses Tupel lässt sich nun aber gödelisieren. Somit kann man Wörter aus Σ^* gödelisieren und man schreibt

$$\ulcorner a_1...a_k \urcorner := \langle n_1, ..., n_k \rangle = \tau^*(n_1, ..., n_k)$$

universelle Funktionen: Sei F eine Klasse von (partiellen) Funktionen über $\mathbb{N}$. Eine $n+1$-stellige Funktion $\varphi^{(n+1)}$ heißt n**-universell für** F, falls:

- $\varphi \in F$
- $\forall \psi^{(n)} \in F \;\; \exists e \in \mathbb{N} : \quad \big(\psi = \varphi_e\big)$, wobei $\varphi_e(\vec{x}) = \varphi(e, \vec{x})$ der e-te Zweig von φ ist. D.h.: Für jede n-stellige Funktion ψF gibt es ein $e \in \mathbb{N}$, sodass $\varphi(e, \vec{x})$ diese berechnet. Mit der Funktion φ lassen sich also alle n-stelligen Funtionen aus F durch einen Parameter e kodiert ausdrücken.

Falls es zusätzlich zu jeder anderen $n+1$-stelligen Funktion $\hat{\varphi}^{(n+1)} \in F$ eine Übersetzungsfunktion $h : \mathbb{N} \to \mathbb{N}$ rekursiv und total gibt, mit der

$$\forall e \in \mathbb{N} : \big(\varphi_e = \hat{\varphi}_{h(e)}\big)$$

gilt, so heißt φ Standardaufzählung oder Gödelnummerierung der n-stelligen Funktionen aus F. Es gibt dann keine andere Funktion $\hat{\varphi} \in F$, die mehr kodiert, als φ.

2.4 Zahlendarstellungen

2.4.1 Festkommadarstellungen

- **Unärdarstellung:** Sei $n \in \mathbb{N}_0$ eine Zahl. Ihre Unärdarstellung lautet:

  $$n = (1\underbrace{1...1}_{n \text{ mal}})_1$$

  Sie besteht also aus genau $n+1$ Einsen. Die Null 0 wird also als eine Eins dargestellt.

  Schreibweisen:

  - Sei $n \in \mathbb{N}$. Dann bezeichnet $\underline{n} := 1^{n+1}$ die Unärdarstellung der Zahl n.
  - Sei $\vec{n} = (n_1, ..., n_k) \in \mathbb{N}^k$, dann bezeichnet $\underline{\vec{n}} = \underline{n_1} b \underline{n_2} b ... b \underline{n_k}$ die Unärdarstellung der Komponenten des Vektors $\vec{n}$, durch b getrennt.

- **pos. Festkommazahl** d **zur Basis** b**:** $(d)_b = \big(d_n d_{n-1} ... d_1 d_0, d_{-1} ... d_{-k}\big)_b$:

  $$d = \sum_{i=-k}^{n} b^i \cdot d_i$$

- **vorzeichenbehaftete Festkommazahlen** d**:**

  - *Betrag und Vorzeichen:* d_n gibt das Vorzeichen an. es gilt:

    $$d = (-1)^{d_n} \cdot \sum_{i=-k}^{n-1} b^i \cdot d_i$$

    größte Zahl: $2^n - 2^{-k}$, kleinste Zahl: $-(2^n - 2^{-k})$, zwei Darstellungen für Null (z.B. $< 000 >= 0 =< 100 >$).

– *1-Komplement:*

$$< d >= \sum_{i=-k}^{n-1} 2^i \cdot d_i - d_n(2^n - 2^{-k})$$

a	000	001	010	011	100	101	110	111
$[a]_1$	0	1	2	3	-3	-2	-1	0

Das Negierte $-d$ einer Zahl d lässt sich einfach berechnen:

$$(\overline{d_n ... d_{-k}})_1 = -(d_n ... d_{-k})_1 = -d$$

– *2-Komplement:*

$$d = \sum_{i=-k}^{n-1} 2^i \cdot d_i - d_n 2^n$$

a	000	001	010	011	100	101	110	111
$[a]_1$	0	1	2	3	-4	-3	-2	-1

kleinste Zahl: -2^n; größte Zahl: $2^n - 2^{-k}$

Das Negative $-d$ einer Zahl d erhält man im 2-Komplement durch:

1. alle Bits invertieren
2. an der niederwertuigsten Stelle 1 addieren.

Beispiel: $k = 0, n = 3 :\ [011]_2 = [\overline{011} + 001]_2 = [100 + 001]_2 = [101]_2 = -3$

2.4.2 Gleitkommadarstellungen

- Gleitkommadarstellung einer Zahl z mit *single precission* nach IEEE 754:

Bits:	31	30..23	22..0
Bedeutung:	Sign S	Charakteristik C	Mantisse M
Zuordnung:	S	$e_7..e_0$	$m_{-1}..m_{-23}$

- Gleitkommadarstellung einer Zahl z mit *double precission* nach IEEE 754:

Bits:	63	62..52	52..0
Bedeutung:	Sign S	Charakteristik C	Mantisse M
Zuordnung:	S	$e_10..e_0$	$m_{-1}..m_{-52}$

- Es gilt bei n Exponentenbits mit obigen Benennungen: $E := C - \text{BIAS}$ mit $\text{BIAS} = 2^{n-1} - 1$,

C	M	Wert
255 (2047)	0	INF (∞)
255 (2047)	$\neq 0$	NaN (*not a number*)
0	0	0
0	$\neq 0$	$(-1)^S \cdot \left(\sum_{i=-1}^{-k} m_i 2^i\right) \cdot 2^{-126}$
$0 < C < 255$ (2047)	beliebig	$(-1)^S \cdot \left(1 + \sum_{i=-1}^{-k} m_i 2^i\right) \cdot 2^{C-\text{BIAS}}$

- **Addition:**
  1. Angleichen des kleineren an den größeren Exponenten
  2. Addition der Mantissen
  3. Normalisierung, Rundung ...
- **Multiplikation:**
  1. Multiplizieren der Vorzeichen
  2. Multiplizieren der Mantissen
  3. Addition der Exponenten:

$$\text{Char}_{erg} = (\text{Char}_1 - \text{BIAS}) + (\text{Char}_2 - \text{BIAS}) + \text{BIAS} = \text{Char}_1 + \text{Char}_2 - \text{BIAS}$$

  4. 4. Normalisierung, Rundung ...

2.5 Einige Rekursive Beziehungen

- Rekursion für $O(N^2)$:

$$\begin{aligned} C_N &= N + C_{N-1} \quad N \geq 2; \quad C_1 = 1 \\ &= N + (N-1) + (N-2) + ... + 2 + 1 \\ &= \frac{N \cdot (N+1)}{2} \in O(N^2) \end{aligned}$$

- Rekursion für $O(\log_2 N)$:

$$\begin{aligned} C_N &= 1 + C_{N/2} \quad N \geq 2; \quad C_1 = 1, \quad N = 2^n \text{ mit } n \in \mathbb{N} \\ &= C_{2^n} = 1 + C_{2^{n-1}} = 1 + 1 + C_{2^{n-2}} = ... \\ &= n \cdot 1 + C_1 = \log_2 N + 1 \in O(\log_2 N) \end{aligned}$$

- Rekursion für $O(2N)$:

$$\begin{aligned} C_N &= N + C_{N/2} \quad N \geq 2; \quad C_1 = 0 \\ &= N + \frac{N}{2} + C_{N/4} = N + \frac{N}{2} + \frac{N}{4} + C_{N/8} \\ &= N \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2N \end{aligned}$$

- Rekursion für $O(N \cdot \log_2 N)$ (Divide and Conquer):

$$\begin{aligned} C_N &= N + 2 \cdot C_{N/2} \quad N \geq 2; \quad C_1 = 0, \quad N = 2^n \text{ mit } n \in \mathbb{N} \\ &= C_{2^n} = 2^n + 2 \cdot C_{2^{n-1}} = 2^n + 2 \cdot \left(2^{n-1} + 2 \cdot C_{2^{n-2}}\right) = ... \\ &= n \cdot 2^n + 2 \cdot C_1 = N \cdot \log_2 N \in O(N \cdot \log_2 N) \end{aligned}$$

3 Formale Sprachen

3.1 Alphabete, Wörter, Sprachen

Alphabete: Ein *Alphabet* Σ ist eine endliche, nichtleere Menge. $a \in \Sigma$ heißt **Buchstabe** des Alphabets. Die Aufzählung eines Alphabets erfolgt für gewöhnlich in der kanonischen Reihenfolge $\Sigma = \{a_0, ..., a_{n-1}\}$.

Worte: Ein *Wort* $w = a_0 a_1 ... a_{n-1}$ über einem Alphabet Σ ist eine endliche Folge von Buchstaben aus Σ. Man vereinbaart folgende Symbole und Schreibweisen:

- λ oder oft auch ϵ bezeichnet das **leere Wort** (beides kommt in diesem Skript vor).
- $|w|$:=Anzahl der Buchstaben in w
- $\sharp_a(w)$:=Anzahl der Vorkommen con 'a' in w $\quad (\sharp_0(1001) = 2 = \sharp_1(1001))$
- Σ^* := Menge aller Wörter über Σ;
  $\Sigma^+ := \Sigma^* - \{\lambda\}$;
  $\Sigma^{=n} \equiv \Sigma^n := \{w \in \Sigma^* \big| |w| = n\}$
- Binärdarstellung über $\Sigma = \{0, 1\}$: $\mathrm{bin}(n) \qquad (\mathrm{bin}(4) = 100)$;

Ordnungen auf Σ^*:

- **lexikographische Ordnung *(partielle Ordnung)*:**

$$v \leq_{\text{LEX}} w :\Leftrightarrow \exists a, b \in \Sigma : \ \exists\, x, y, y' \in \Sigma^* : \ v = xay \ \wedge \ w = xby' \ \wedge \ a < b$$

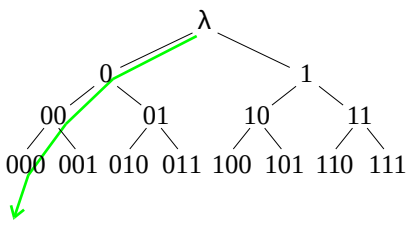
  Die lexikographische Ordnung entspricht einem Tiefendurchlauf eines Baumes. Anschaulich bedeutet die lexikographische Ordnung, dass die Sortierung nach dem ersten Zeichen erfolgt, dass in v und w unterschiedlich ist (von links!). Dies entspricht der Ordnung, die in Lexika verwendet wird.
  Beispiel über Σ_2: $\lambda < 0 < 00 < 000 < 0000 < ... < 1 < ...$, aber z.B. $011001010 =_{LEX} 01100$

- **längen-lexikographische Ordnung *(totale Ordnung)*:**

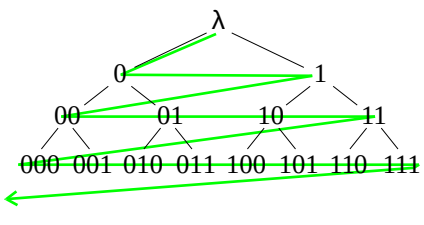
$$v \leq_{ll} w :\Leftrightarrow \quad \underbrace{|v| < |w|}_{\text{Hauptordnungskriterium}} \quad \vee \ \underbrace{(|v| = |w| \ \wedge \ v <_{\text{LEX}} w)}_{\text{Nebenordnungskriterium}}$$

  Die längen-lexikographische Ordnung entspricht einem Breitendurchlauf eines Baumes.
  Beispiel über Σ_2: $\lambda < 0 < 1 < 00 < 01 < 10 < 11 < 000 < 001 < ...$

lexikographische Ordnung



längen-lexikographische Ordnung



Operationen auf Wörtern:

- **Konkatenation:**

$$\circ : \begin{cases} \Sigma^* \times \Sigma^* \rightarrow & \Sigma^* \\ (v, w) & \mapsto v \circ w = vw \end{cases}$$

- **n-fache Iteration:**

$$\begin{aligned} w^0 &:= \lambda \\ w^n &:= w \circ w^{n-1} = \underbrace{w \circ ... \circ w}_{n\text{-mal}} \end{aligned}$$

- **Spiegelwort:**

$$(a_0 ... a_n)^R = a_n ... a_0$$

Eigenschaften von Wörtern: Im folgenden seien $v, w \in \Sigma^*$ zwei beliebige Wörter. Es gilt:

- v heißt **Teilwort** von w, falls gilt: $w = xvy$, mit $x, y \in \Sigma^*$. Das teilwort heißt **echt**, falls $x, v, y \neq \lambda$.
- v heißt **Präfix** von w, falls gilt: $w = vx$, mit $x \in \Sigma^*$. Ein Präfix heißt **echt**, falls $x, w \neq \lambda$
- v heißt **Suffix** von w, falls gilt: $w = xv$, mit $x \in \Sigma^*$. Ein Suffix heißt **echt**, falls $x, w \neq \lambda$

Sprachen über Σ: Eine Sprache L ist eine Teilmenge von Σ^*, also eine Menge von Wörtern über Σ. Auf Sprachen L, L_1, L_2 sind folgende Operationen definiert:

- die üblichen **Mengenoperationen** Vereinigung $L_1 \cup L_2$ und Schnitt $L_1 \cap L_2$ und Komplement $\overline{L}$.
- **Verkettung/Konkatenation:** $L_1 L_2 := \{vw | v \in L_1 \wedge w \in L_2\}$
- **n-fache Iteration:**

$$\begin{aligned} L^n &:= \{v_1 ... v_n | n \geq 0 \wedge v_1, ..., v_n \in L\} \\ L^0 &:= \{\lambda\} \neq \varnothing \end{aligned}$$

- **Kleene-Abschluss:** $L^* = \bigcup_{n=0}^{\infty} L^n$
- **transitiver Abschluss:** $L^+ == L^* - \{\lambda\}$

3.2 CHOMSKY-Grammatiken und -Sprachen

Grammatiken beschreiben (die Struktur von) Sprachen mit Hilfe von Regeln. Bei Chomsky-Grammatiken handelt es sich hier um Ersetzungsregeln.

Produktionen/Ersetzungsregeln: Sei Σ ein Alphabet. Eine Produktion r ist definiert als:

$$r = (u, v) \in \Sigma^* \times \Sigma^*$$

üblicherweise schreibt man dafür:

$$u \rightarrow v$$

wobei u als Prämisse und v als Konklusion bezeichnet wird.
Eine solche Produktion ist einfach eine Ersetzungsregel und kann auf ein Wort $w \in \Sigma^*$ angewendet werden, das die Prämisse u als Teilwort enthält: $w = xuy$. Die Anwendung der Produktion bewirkt, dass das Vorkommen von u in w in die Konklusion v umgewandelt wird:

$$xuy = w \;\rightarrow\; w' = xvy.$$

Termersetzungssystem: Sei Σ ein Alphabet und P eine endliche Menge von Produktionen r. Dann heißt das Paar $E = (\Sigma, P)$ Termersetzungssystem.

Umformungssystem: Sei $E = (\Sigma, P)$ ein Termersetzungssystem. Für das ihm zugeordnete Umformungssystem $U_E = (\Sigma^*, \underset{E}{\Rightarrow})$ gilt:

$$\forall w, w' \in \Sigma^* : \;\; \big(w \underset{E}{\Rightarrow} w' \;\; \Leftrightarrow \;\; \exists (u, v) \in P : \;\; \exists x, y \in \Sigma^* \big(w = xuy \text{ und } w' = xvy\big)\big)$$

Herleitung/Ableitung: Sei $E = (\Sigma, P)$ ein Termersetzungssystem und $U_E = (\Sigma^*, \underset{E}{\Rightarrow})$ das zugehörige Umformungssystem. Eine Herleitung (der Länge n) von w' aus w ist eine U_E-Rechnung (der Länge n). Ist w' eine Stoppkonfiguration von U_E, so heißt w' Normalform. *Eine Ableitung bedeutet also, dass nacheinander Regeln aus der Menge der Produktionen angewendet werden, um vom Axiom S über verschiedene Satzformen als Zwischenergebnisse einen Satz (besteht nur aus Terminalsymbolen) herzuleiten.*

Rechts-/Linksbleitung: Wird während einer Ableitung immer zuerst das rechteste/linkeste Nicht-Terminal zuerst abgeleitet (ersetzt), so spricht man von einer **Rechts-/Linksableitung**.

Beispiel: Binärzahlterme mit Termersetzungssystem

indirekte Definition:	*korrespondierende Ersetzungsregeln:*
(1) Jede Binärzahl ist ein Term	(T1) $T \rightarrow Z$
(2) t_1, t_2 Terme $\Rightarrow (t_1 + t_2)$ ist Term	(T2) $T \rightarrow (T \underline{+} T)$
(3) t_1, t_2 Terme $\Rightarrow (t_1 \cdot t_2)$ ist Term	(T3) $T \rightarrow (T \underline{\cdot} T)$
(4) 0 ist eine Binärzahl	(Z1) $Z \rightarrow \underline{0}$
(5) Ist w ein Binärwort, so ist $1w$ eine Binärzahl	(Z2) $Z \rightarrow \underline{1}w$
(6) λ ist ein Binärwort	(W1) $W \rightarrow \lambda$
(7) ist v ein Binärwort, $\Rightarrow v0, v1$ sind Binärwörter	(W2) $W \rightarrow W\underline{0}$
	(W3) $W \rightarrow W\underline{1}$

Chomsky-Grammatik: Eine Chomsky-Grammatik $G = (N, T, P, S)$ besteht aus den zwei disjunkten Alphabeten N (nichtterminale Zeichen, syntaktische Variablen) und T (terminale Zeichen), einer Menge von Produktionen $P \subseteq ((N \cup T)^* - T^*) \times (N \cup T)^*$ und einem ausgezeichneten Buchstaben $S \in N$ (**Axiom**).

Die Prämisse u (aus $(u, v) \in P$) muss hier also mindestens eine Variable enthalten ($u \in ((N \cup T)^* - T^*)$). Die Konklusion darf beliebig aus Variablen und Terminalen zusammengesetzt werden. Außerdem beginnen alle Herleitungen mit dem Axion S, einer syntaktischen Variable.

Die von einer Chomsky-Grammatik $G = (N, T, P, S)$ erzeugte Sprache wird mit $L(G)$ bezeichnet und es gilt:

$$L(G) = \left\{ w \in T^* \middle| S \overset{*}{\underset{E}{\Rightarrow}} w \right\}$$

Ein aus Terminalzeichen bestehendes Wort $w \in T^*$ bezeichnet man als **Terminalwort** oder **Satz.**
Ein beliebiges Wort $w \in (N \cup T)^*$ heißt **Satzform.** Eine Satzform kann also auch Variablen enthalten. Enthält eine Satzform tatsächlich Variablen, so spricht man auch von einer **echten Satzform.**

Beispiele:

- *Sprache* $L = \left\{ 0^m 1^n \middle| n, m \geq 1 \right\}$*:*zugehörige Grammatik: $G = (N, \{0, 1\}, P, S)$, mit $N = \{S, T\}$. Die Regeln P sind:
  1. $S \rightarrow 0S$
  2. $S \rightarrow 0T$
  3. $T \rightarrow 1T$
  4. $T \rightarrow 1$
- *Sprache* $L = \left\{ 0^n 1^n \middle| n \geq 1 \right\}$*:*zugehörige Grammatik: $G = (\{S\}, \{0, 1\}, P, S)$. Die Regeln P sind:
  1. $S \rightarrow 0S1$
  2. $S \rightarrow 01$

3.3 Schreibweisen und Darstellungen

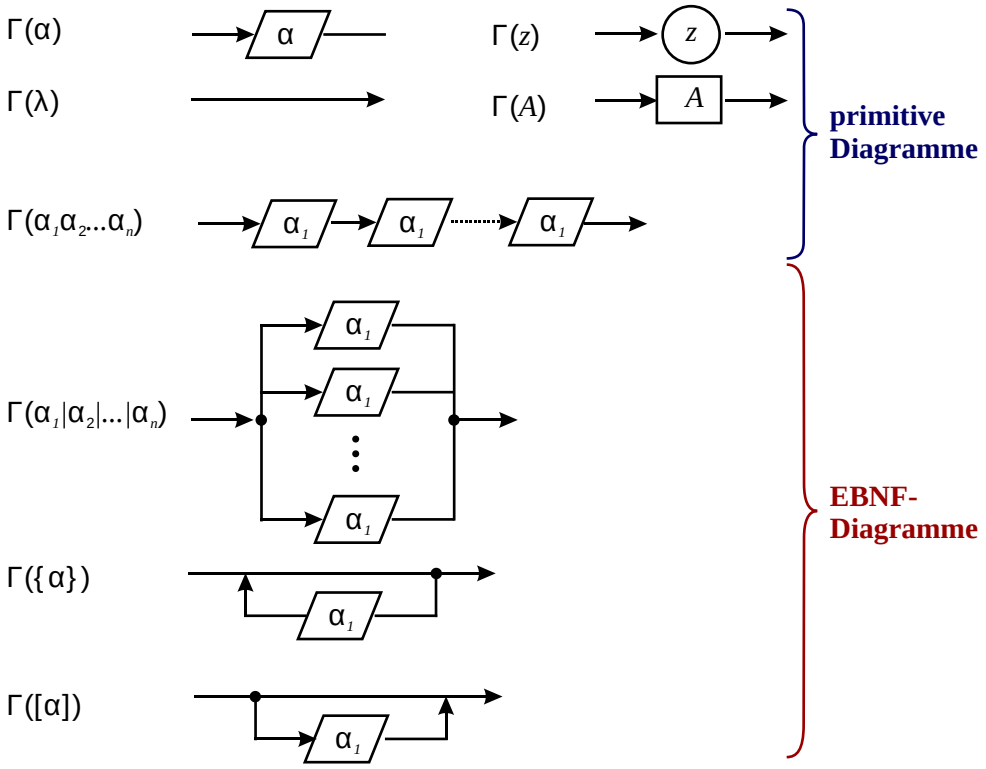
3.3.1 Erweiterte Backus-Naur-Form (EBNF)

Die EBNF erweitert die Darstellung von Produktionen um einige nützliche Schreibweisen:

- **Alternativsymbol:** Sind mehrere Regeln mit gleicher linker Seite vorhanden ($A \rightarrow \alpha$, $A \rightarrow \beta$,...), so kann man diese mit dem Alternativsymmbol | zu einer Regel zusammenfassen: $A \rightarrow \alpha|\beta|...$.
- **beliebige Wiederholungen:** Die Schreibweise $A \rightarrow \{\alpha\}$ steht für eine beliebige Wiederholung von α, also für $A \rightarrow \lambda|\alpha|\alpha\alpha|...$.
- **optionale Generierung:** $A \rightarrow [\alpha]\beta$ steht für eine optionale Generierung von α, also für $A \rightarrow \alpha\beta|\beta$

3.3.2 Syntaxdiagramme

Syntaxdiagramme sind Diagramme, die geeignet sind um Sprachen darzustellen, die von Produktionen der Form $A \rightarrow \alpha, \quad A \in N$ beschrieben werden. Dabei steht auf der linken Seite der Produktion nur ein nicht-terminales Symbol A. Syntaxdiagramme $\Gamma(\cdot)$ sind wie folgt definiert (hier ist $\alpha_{(i)}$ ein beliebiger Ausdruck, A eine Variable z ein Wort und λ das leere Wort):



Diese elementaren Diagramme können zu größeren Verbunden verkettet werden.

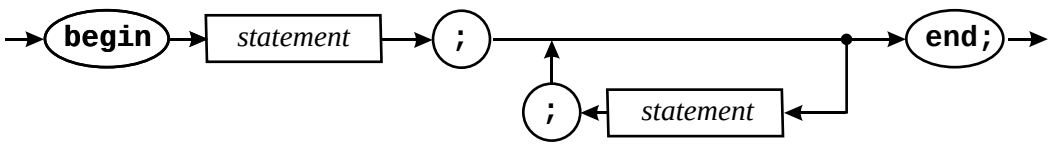
Beispiel: Die folgende Grammatik definiert Programmblöcke, wie sie etwa in Pascal verwendet werden:

$$\begin{array}{lcl} B & \rightarrow & \texttt{begin}\ S\ \texttt{end;} \\ S & \rightarrow & S_e\ \texttt{;}\ \ S \\ S & \rightarrow & S_e\ \texttt{;} \end{array}$$

Diese Grammatik kann man auch in EBNF notieren:

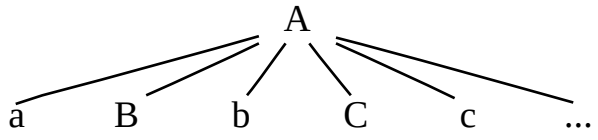
$$\textit{block} \quad \rightarrow \quad \texttt{begin}\ \textit{statement}\ \texttt{;}\ \ \{\ \textit{statement}\ \texttt{;}\ \ \}\ \texttt{end;}$$

Schließlich kann man daraus einen Syntaxgraphen erzeugen:



3.3.3 Ableitungsbäume

Ableitungsbaum/Parsebaum: Ein Parse-Baum zu einer kontextfreien Grammatik ist ein geordneter beschrifteter Baum. Seine Blätter enthalten Terminalsymbole, während sein Knoten die linken Seiten der entsprechenden Produktionen enthalten. Man kann Parse-Bäume rekursiv definieren ($a, b, c \in T^*,\ \ A, B, C \in N$):

Regel	zugehöriger Teilbaum
$A \rightarrow a$	
$A \rightarrow a\ B\ b\ C\ c\ ...$	

3.3.4 Reguläre Ausdrücke

Reguläre Sprachen: Die regulären Sprachen über einem Alphabet Σ können rekursiv definiert werden:

1. $\{\}$ und $\{\lambda\}$ sind regulär.
2. $\{a\}$ ist für jedes $a \in \Sigma$ regulär
3. Sind L_1, L_2 zwei reguläre Sprachen, dann sind auch die Vereinigung $L_1 \cup L_2$, die Konkatenation $L_1 L_2$ und der Abschluss L_1^*, L_2^* regulär.
4. Nur die gemäß der Regeln 1-3 konstruierten Sprachen sind regulär.

Diese Definition definiert gleichzeitig auch reguläre Mengen, wenn man Sprachen L als Mengen auffasst.

Notation: Es gibt eine komfortable Notation für reguläre Ausdrücke. Dabei geht man von einem Alphabet Σ aus und erweitert es um Meta-Zeichen, also $\Sigma \rightarrow \Sigma \cup \{\varnothing, \lambda, (,), |, *\}$. Diese neuen Meta-Zeichen dürfen kein Bestandteil von Σ sein, da ihnen eine Sonderrolle zukommen wird.

1. $\varnothing$ und λ sind reguläre Ausdrücke und beschreiben die Sprachen $\{\}$ und $\{\lambda\}$.
2. Für $a \in \Sigma$ ist a ein regulärer Ausdruck und beschreibt die Sprache $\{a\}$.
3. Sind r_1 und r_2 reguläre Ausdrücke, die die Sprachen L_1 und L_2 beschreiben, so gilt:
   - $(r_1|r_2)$ ist ein regulärer Ausdruck und beschreibt $L_1 \cup L_2$
   - $(r_1 r_2)$ ist ein regulärer Ausdruck und beschreibt $L_1 L_2$
   - $(r_1)*$ ist ein regulärer Ausdruck und beschreibt L_1^*. Dies bedeutet, dass r_1 beliebig oft hintereinander auftreten kann, auch keinmal.
4. keine anderen Elemente aus $\left(\Sigma \cup \{\varnothing, \lambda, (,), |, *\}\right)^+$ sind reguläre Ausdrücke.

Es ist oft sinnvoll die obige Notation noch zu erweitern:

- r^+ bedeutet rr^*, also mindestens einmal r
- $\overline{r}$ ist das Komplement $\overline{L}$ der Sprache L
- $\neg a$ beschreibt $\Sigma - \{a\}$
- ? steht für ein beliebiges Zeichen aus Σ (Wildcard)
- $*$ steht für Σ^* (Wildcard)
- $a - z$ beschreibt in einem geordneten Alphabet Σ alle Zeichen zwischen a und z (Bereich).

Beispiele:

- $0 \mid (1-9)\ (0-9)^*$ beschreibt nicht-negative ganze Zahlen
- $(+ \mid - \mid \lambda)\ (0-9)^+$ beschreiben beliebige ganze Zahlen
- $//(\neg\text{eol}^*)$ eol beschreibt C++-Kommentare

3.4 Chomsky-Hirarchie

1956 hat der Linguist Noam Chomsky folgende Hierarchie der formalen Sprachen eingeführt. Sie wird auch heute noch benutzt. Zunächst werden in einzelnen Abschnitten die verschiedenen Sprachklassen definiert und erläutert. Danach folgt eine kurze Zusammenstellung der Sprachen und ihrer Eigenschaften als Tabelle.

Chomsky-Grammatiken Typ CH-0 (allgemeine Grammatiken): Alle Grammatiken $G = (N, T, P, S)$ sind vom **Typ CH-0**. Die Klasse der Chomsky-0-Grammatiken wird mit $\text{CH}-0 = \text{CH}$ bezeichnet.

CH ist gegen Vereinigung und Durchschnitt, nicht aber gegen Komplementbildung abgeschlossen.

Grammatiken vom Erweiterungstyp: Sei $G = (N, T, P, S)$ eine λ-treue Grammatik. Haben alle Regeln (bis auf die λ-Regeln) die Form

$$v \to w \qquad |v| \leq |w|$$

so ist G vom **Erweiterungstyp**. Die Klasse der Erweiterungstyp-Grammatiken wird mit ERW bezeichnet.

Anschaulich bedeutet das, dass keine Regel eine Verkürzung bewirkt.

Chomsky-Grammatiken Typ CH-1/kontextsensitive Grammatiken: Sei $G = (N, T, P, S)$ eine λ-treue Grammatik. Haben alle Regeln (bis auf die λ-Regeln) die Form

$$xXy \to xwy \quad \text{mit } X \in N,\ \ w \in (N \cup T)^+,\ \ x, y \in (N \cup T)^*$$

so ist G **kontextsensitiv (ks)**. Die Klasse der Chomsky-1-Grammatiken wird mit $\text{CH}-1$ oder KS bezeichnet.

Anschaulich bedeutet das, dass in den Regel nur Variablen ersetzt werden, und das abhängig von den sie umgebenden Zeichen, dem Kontext (eben kontext-sensitiv). Außerdem darf, wegen $w \in (N \cup T)^+$ das Wort, durch das die Variable X ersetzt wird, nicht leer sein. Die einzige λ-regel ist hier natürlich aufgrund der λ-Treue die Regeln $S \to \lambda$.

Zu jeder Kontextsensitiven Grammatig G lässt sich effektiv ein nichtdeterministischer linear beschränkter Automat M finden, der $L(G)$ akzeptiert, und umgekehrt.
KS ist gegen Vereinigung, Durchschnitt und Komplement abgeschlossen.

Chomsky-Grammatiken Typ CH-2/kontextfreie Grammatiken: Sei $G = (N, T, P, S)$ eine Grammatik. Haben alle Regeln die Form

$$X \to w \quad \text{mit } X \in N,\ \ w \in (N \cup T)^+$$

so ist G **kontextfrei (kf)**. Die Klasse der Chomsky-2-Grammatiken wird mit $\text{CH}-2$ oder KF bezeichnet.

Anschaulich bedeutet das, dass in den Regel nur Variablen ersetzt werden, und zwar unabhängig von den

sie umgebenden Zeichen.
Die Menge

$$E = \{X \in N | X \overset{*}{\underset{E}{\Rightarrow}} \lambda\}$$

der eleminierbaren Variablen eine kontextfreien Grammatik G lässt sich effektiv bestimmen.

lineare Grammatiken: Sei $G = (N, T, P, S)$ eine Grammatik. Enthält G nur Regeln der Form

$$X \to xYy \quad \text{mit } X \in N,\ \ Y \in N \cup \{\lambda\},\ \ x, y \in T^*$$

so ist sie eine **lineare Grammatik.** Die Klasse der linearen Grammatiken wird mit LIN bezeichnet.

Anschaulich kann es in linearen Grammatiken keine vermehrung der Variablen geben. Bis zur Anwendung der letzten Regeln (von der Form $X \to x,\ \ x \in T^*$, ergibt sich mit $Y = \lambda$), bleibt immer genau eine Variable Bestandteil der erzeugten Wortes.

Chomsky-Grammatiken Typ CH-3/rechts- & linkslineare Grammatiken: Sei $G = (N, T, P, S)$ eine Grammatik. Enthält G nur Regeln der Form

$$X \to xY \quad \text{mit } X \in N,\ \ Y \in N \cup \{\lambda\},\ \ x \in T^*$$

so ist sie eine **rechts-lineare Grammatik,** bzw. eine Grammatik vom *Typ CH-3*. Die Klasse der rechtslinearen Grammatiken wird mit RLIN oder CH -3 bezeichnet. Analog definiert man linkslineare Grammatiken über die Regeln:

$$X \to Yy \quad \text{mit } X \in N,\ \ Y \in N \cup \{\lambda\},\ \ y \in T^*$$

Die Klasse der linkslinearen Grammatiken wird mit LLIN bezeichnet.

Hierbei handelt es sich um einen Spezialfall der linearen Grammatiken. Die Variable bleibt immer am rechten Ende des Wortes. Das Ergebniswort wird links aufgebaut.

Außerdem gilt:
Alle CH-3-Grammatiken sind eindeutig, d.h. zu jedem Wort der zugehörigen Sprache gibt es nur genau eine Herleitung.

Chomsky-Hierarchiesatz: Zwischen den Klassen der einzelnen Grammatiken gilt folgende hierarchische Beziehung:

$$\underset{=\text{LLIN}=\text{RLIN}}{\text{CH}-3} \subsetneq \text{LIN} \subsetneq \underset{=\text{KF}}{\text{CH}-2} \subsetneq \underset{=\text{KS}=\text{ERW}}{\text{CH}-1} \subsetneq \underset{=\text{CH}}{\text{CH}-0}.$$

Comsky-Normalform: Eine kontextfreie Grammatik $G = (N, T, P, S)$ ist in *Chomsky-Normalform,* wenn G λ-treu ist und neben der eventuellen λ-Regel $S \to \lambda$ nur Regeln der folgenden Form enthalten sind:

$$X \to YZ \quad \text{und} \quad X \to a \quad (X, Y, Z \in N,\ \ a \in T)$$

Jede kontextfreie Grammatik lässt sich effektiv in eine äquivalente kontextfreie Grammatik in Chomsky-Normalform überführen.

Säparierte Grammatiken: Eine Grammatik $G = (N, T, P, S)$ heißt **säpariert**, falls P nur Regeln der folgenden beiden Formen enthält:

- **Umformungsregeln:** $v \to w$ mit $v \in N^+,\ \ w \in N^*$.
- **Substitutionsregeln:** $X \to a$ mit $X \in N,\ \ a \in T$.

Sätz über Chomsky-Sprachen:

- Jede Grammatik des Typs 0..2 und des Erweiterungstyps lässt sich effektiv in eine äquivalente, separierte Grammatik desselben Typs überführen.
- Jede Grammatik vom Erweiterungstyp lässt sich effektiv in eine äquivalente, kontextsensitive Grammatik überführen.

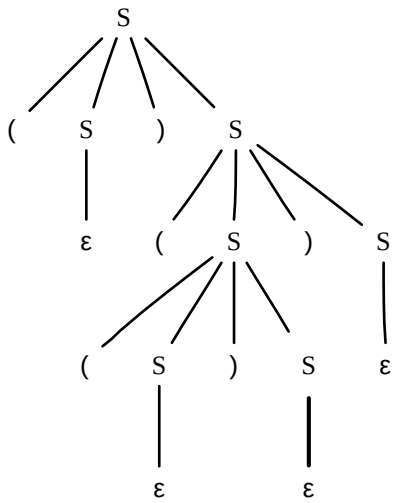
Zusammenfassung der Chomsky-Hierarchie Die folgende Zusammenfassung lehnt sich an den Wikipedia-Artikel `http://de.wikipedia.org/wiki/Chomsky-Hierarchie` an. Griechische Symbole stehen für Wörter, die aus terminalen und nicht-terminalen Symbolen bestehen dürfen $\alpha, \beta\gamma \in (N \cup T)^*$. Große lateinische Buchstaben stehen für Nicht-Terminalsymbole $A, B \in N$. Kleine lateinische Buchstaben stehen für Wörter aus Terminalsymbolen $a \in T^*$.

Grammatik	Regeln	Sprachen	erkennender Automat
CH-0	beliebig	rekursiv aufzählbar	Turing-Maschinen
CH-1	$\alpha A \beta \rightarrow \alpha\gamma\beta$	kontextsensitiv	linear platzbeschränkte, nichtdeterministische Turing-Maschinen
CH-2	$A \rightarrow \gamma$	kontextfrei	nichtdeterministischer Kellerautomat
CH-3	$A \rightarrow aB$ *(rechts-linear)* $A \rightarrow Ba$ *(links-linear)* $A \rightarrow a$	rechts-/linkslineare Sprachen	deterministischer Kellerautomat

Beispiel: Klammerausdrücke: Als Beispiel soll die folgende Sprache betrachtet werden:

$$\begin{array}{lcl} S & \rightarrow & \texttt{(}\ S\ \texttt{)} \\ S & \rightarrow & \epsilon \end{array}$$

Ein mögliches Wort der hierdurch definierten Sprache ist `()(())`. Der Ableitungsbaum zu diesem Wort sieht so aus:



Man kann zeigen, dass diese Sprache alle gültigen Klammerausdrücke beschreibt. Ein gültiger Klammerausdruck ist ein Wort über $T = \{(,)\}$, in dem zu jeder öffnenden Klammer auch eine schließende Klammer vorhanden ist. Diese muss außerdem auf der richtigen Ebene liegen. So sind etwa `))(( ` oder `(()` keine gültigen Klammerausdrücke. Den Beweis der obigen Behauptung führt man per Induktion:

„⇒“ Es werden nur korrekte Klammerausdrücke generiert. Die Induktion erfolgt über die Anzahl der Ableitungsschritte.
Induktionsanfang: ein Ableitungsschritt. Die einzige mögliche Ableitung ist $S \rightarrow \epsilon$. Dies ist aber ein korrekter Klammerausdruck, der einfach keine Klammern enthält.
Induktionsannahme: Die Behauptung gelte für $1 \leq n, m \leq k$ Ableitungschritte, d.h. $S \overset{k}{\Rightarrow} w$ ist ein Wort der Sprache.
Induktionsschluss: Die $(k+1)$-te Regel wird dann als erste Regel angewendet, sodass man erhält:

$$S \overset{1}{\Rightarrow} (\ S\ )\ S \overset{k}{\Rightarrow} (\ u\ )\ v = w$$

Die Ableitung der Worte u, v erfolgt in $n, m \leq k$ Schritten. Da damit u, v korrekte Klammerausdrücke sind, muss auch w ein korrekter Ausdruck sein.

„⇐“ Hier führt man den Beweis mit Induktion über die Länge $|w|$ eines korrekten Klammerausdruckes w.
Induktionsanfang: $|w| = 0 \quad \Rightarrow \quad w = \epsilon$. Das leere Wort lässt sich trivial durch Anwendung der zweiten Regel auf das Axiom erzeugen.
Induktionsannahme: Zu einem korrekten Klammerausdruck w mit $|w| \leq 2k$ sei eine Ableitung gefunden.
Induktionsschluss: Das erste Zeichen im korrekten Klammerausdruck w' mit $|w'| = 2k + 2$ ist sicher (, sodass sich w' folgendermaßen zerlegen lässt:

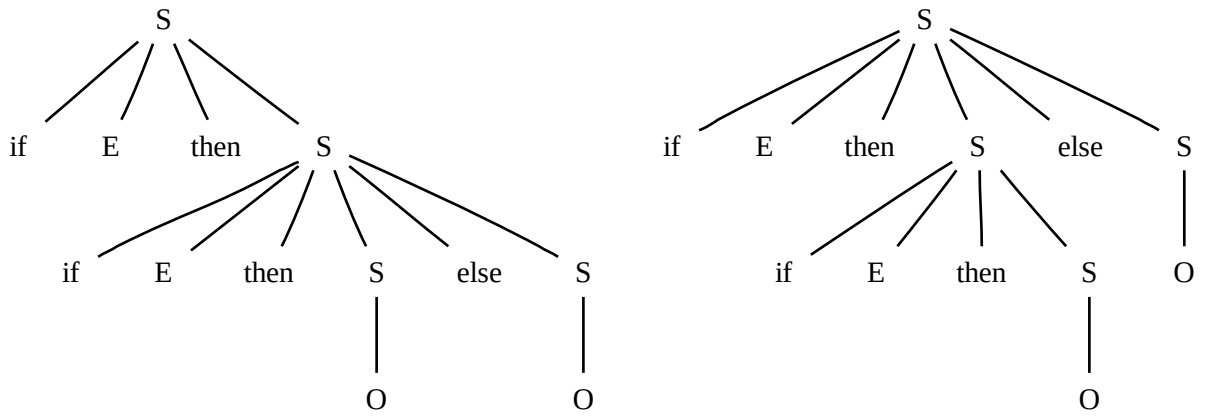
$$w' = (\ u\ )\ v$$

Dabei gilt dann $|u|, |v| \leq 2k$. Damit sind aber nach I-Vorraussetzung die Worte u, v korrekte Klammerausdrücke, für die es eine Ableitung gibt. Die Ableitung von w' erfolgt dann über $S \Rightarrow (\ S\ )\ S \overset{*}{\Rightarrow} (\ u\ )\ v$.

Beispiel: „dangling else“-Problem: Man betrachtet das Problem, in einer Programmiersprache zu entscheiden zu welchem **`if`** ein gegebenes **`else`** gehört. Man betrachtet die folgende Gramatik:

$$\begin{array}{lcl} S & \rightarrow & \texttt{if}\ E\ \texttt{then}\ S \\ & & \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\ & & \mid O \end{array}$$

Man betrachtet dann die Satzform **`if`** E **`then if`** E **`then`** O **`else`** O. Hier stellt sich nun die Frage zu welchen **`if`** die hintere **`else`**-Klausel gehört. Man kann zwei Ableitungsbäume angeben, die das **`else`** dem ersten oder dem zweiten **`if`** zuordnen:



In der Praxis kann man dieses Problem (unter anderem) auf die folgenden zwei Arten lösen:

1. durch erzwingen von Blöcken in einer neue Grammatik:

$$\begin{array}{lcl} S & \rightarrow & \texttt{if}\ E\ \texttt{then}\ S\ \texttt{endif} \\ & & \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{endif} \\ & & \mid O \end{array}$$

2. durch Definition außerhalb der Grammatik. So könnte etwa das linkeste **`else`** immer zum rechtesten **`if`** vor dem **`else`** gehören.

3.5 Eigenschaften von Sprachen und Grammatiken

λ-frei/λ-treu: Sei $G = (N, T, P, S)$ eine Grammatik. Eine Regel $w \rightarrow \lambda$ heißt **λ-Regel**.
Die Grammatik G heißt **λ-frei**, wenn P keine λ-Regel enthält.
Die Grammatik G heißt **λ-treu**, wenn entweder G λ-frei ist, oder $S \rightarrow \lambda$ die einzige λ-Regel in P ist.

Für λ-treues G ist also $\lambda \in L(G)$ genau dann, wenn $S \rightarrow \lambda$ in P ist. In diesem Fall ist $S \rightarrow \lambda$ die einzige Herleitung des leeren Wortes.

mehrdeutige Grammatik: Eine Grammatik G heißt **mehrdeutig**, wenn es zu einem Wort $w \in L(G)$ aus der von ihr erzeugten Grammatik verschiedene Ableitungen gibt, die w erzeugen. Umgekehrt heißt die Grammatik **eindeutig**, wenn es zu jedem $w \in L(G)$ genau eine Herleitung gibt.

inhärent mehrdeutige Sprache: Eine Sprache L heißt inhärent mehrdeutig, wenn alle sie erzeugenden Grammatiken mehrdeutig sind. Im allgemeinen ist es nicht entscheidbar, ob eine vorgegebene Grammatik eindeutig ist.

Äquivalenz von Grammatiken: Zwei Grammatiken G und G' heißen äquivalent, wenn sie die selbe Sprache erzeugen, also $L(G) = L(G')$.

Die Definition einer Grammatik ist sehr frei gehalten. Es können also auch überflüssige Variablen und Produktionen vorkommen.

unerreichbare und unproduktive Symbole, reduzierte Grammatiken: Man geht von einer Grammatik $G(N, T, S, P)$ aus. Es gilt dann:

1. Ein Nicht-Terminal $A \in N$ heißt **unerreichbar**, wenn es keine Ableitung einer Satzform gibt, auf deren rechter Seite es vorkommt. Es gibt also keine $\alpha, \beta \in (N \cup T)^*$, sodass $S \overset{*}{\Rightarrow} \alpha A \beta$ gilt.
2. Ein Nicht-Terminal $A \in N$ heißt **unproduktiv**, wenn es kein Wort $u \in T^*$ aus Terminalsymbolen gibt, das von A abgeleitet werden kann. Es gibt also kein $u \in T^*$ mit $A \overset{*}{\Rightarrow} u$.
3. Eine Grammatik G heißt **reduziert**, falls sie keine unerreichbaren und keine unproduktiven Symbole enthält. Eine reduzierte Grammatik G_{red} zu einer Grammatik G beschreibt die selbe Sprache, wie G (G und G_{red} sind äquivalent), also

$$L(G) = L(G_{\text{red}}).$$

linksrekursive Grammatiken, zykelfreie Grammatiken:

1. Eine Grammatik G heißt **linksrekursiv**, wenn ein Nicht-Terminal $A \in N$ gibt, für das man eine Ableitung $A \overset{+}{\Rightarrow} A\alpha$ mit $\alpha \in (N \cup T)^+$ gibt.
2. Eine Grammatik G heißt **zykelfrei**, wenn es keine Ableitung $A \overset{+}{\Rightarrow} A$ gibt, die A wieder auf sich selber abbildet.

Pumping-Lemma für reguläre Sprachen (CH-0): Sei L eine reguläre Sprache. Dann gibt es eine Zahl n, so dass sich alle Wörter $x \in L$ mit $|x| \geq n$ zerlegen lassen in $x = uvw$ mit:

1. $|v| \geq 1$ und $|uv| \leq n$
2. $uv^iw \in L$, für alle $i \geq 0$

Das Pumping-Lemma besagt also, dass sich jedes Wort aus einer regulären Sprache auf bestimmte Weise zerlegen lässt. Man kann es z.B. benutzen, um zu zeigen, dass eine gegebene Sprache nicht regulär ist. Dazu genügt es ein Wort aus der entsprechenden Sprache zu finden, das die Zerlegung nicht erfüllt.

Beispiel: Man zeige, dass die Sprache $L = \{w \in \{0,1\}^* | \sharp_0(w) > \sharp_1(w)\}$ nicht regulär ist. Man führt einen Widerspruchsbeweis:

- **Annahme:** L sei regulär und $x \in L$.
- Man betrachte nun speziell $x = 1^n0^{n+1} \in L$ Nach Punkt 1 des Pumping-Lemmas besteht die Zerlegung uv nur aus 1, da $|uv| \leq n$ und uv am Anfang von x steht. Es gilt also $v = 1^k$, mit $k \leq n$.
- Damit folgt aber aus dem Pumping-Lemma, dass für $i \geq 0$ auch alle Worte $x'_i = 1^{n-k}(1^k)^i0^{n+1} \in L$ gelten muss. Mit dieser Konstruktion kann man aber über das beliebig zu wählende i ein Wort x'_i konstruieren, für das $\sharp_0(x'_i) > \sharp_1(x'_i)$ nicht mehr gilt. Damit muss aber die Annahme falsch sein und die Sprache L ist nicht regulär.

3.6 CHOMSKY-Grammatiken und Berechenbarkeit

Aufzählbarkeit von Chomsky-Sprachen: Sei G eine Chomsky-Grammatik. Dann ist $L(G)$ rekursiv aufzählbar und man kann zu G effektiv eine Turingmaschine M_G angeben, die $L(G)$ akzeptiert.

Turingmaschinen und Chomsky-Grammatiken: Sei M ein 1-Band-TM und sei $L(M)$ die von M akzeptierte Sprache. Dann lässt sich effektiv eine Chomsky-Grammatik G angeben, die $L(M)$ erzeugt.

*Aus den gerade genannten zwei Lemmata folgt folgender **Satz:***

Zu jeder Turingmaschine M kann man effektiv eine Chomsky-Grammatik G angeben, die die von M akzeptierte Sprache erzeugt, und umgekehrt.

Nicht-rekursive Chomsky-Sprachen:

1. Es gibt nicht-rekursive Chomsky-Sprachen.
2. Das **Wortproblem** der Chomsky-Grammatiken ist nicht rekursiv:

$$W_{\text{CH}} := \{(\ulcorner G \urcorner, x) | x \in L(G)\}$$

Hierbei geht man von geeigneten Gödelisierungen von Grammatiken aus. $\ulcorner G \urcorner \in \mathbb{N}$ ist die Gödelisierung der Grammatik G. Damit kann W_{CH} als eine Relation betrachtet werden, die jedem Wort x die Eigenschaft zuordnet Element von $L(G)$ zu sein, oder eben nicht.

3. Das **Leerheitsproblem** ist nicht rekursiv:

$$\textsc{Empty}_{\text{CH}} = \left\{ \ulcorner G \urcorner \,\middle|\, L(G) = \varnothing \right\}$$

4. Das **Unendlichkeitsproblem** ist nicht rekursiv:

$$\textsc{Inf}_{\text{CH}} = \left\{ \ulcorner G \urcorner \,\middle|\, L(G) \text{ unendlich} \right\}$$

5. Das **Äquivalenzproblem** ist nicht rekursiv:

$$\textsc{Empty}_{\text{CH}} = \left\{ (\ulcorner G \urcorner, \ulcorner G' \urcorner) \,\middle|\, L(G) = L(G') \right\}$$

Für Typ-2-Grammatiken ist das Wortproblem W_{CH} in polynomieller Zeit lösbar. Zur Lösung des Problemes kann man folgenden, nach Cocke, Younger und Kasami benannten Algorithmus verwenden:

CYK-Algorithmus: Man betrachtet hier ein Wort $w = a_1 a_2 ... a_n$ der Länge $|w| = n$ und möchte entscheiden, ob es von einer gegebenen Grammatik in Chomsky-Normalform erzeugt wird. Die Grammatik kann also nur Regeln der folgenden Form enthalten:

$$A \to BC, \quad A \to a, \quad S \to \lambda$$

Dabei sind $A, B, B \in N$ nicht-terminale Symbole und $a \in T^*$ ein terminales Wort. $S \in N$ ist das Axiom und λ das leere Wort. Der Algorithmus läuft wie folgt ab:

Algorithm 1 CYK-Algorithmus zur Lösung des Wortproblems einer Typ-2-Grammatik G

procedure CYK($G, w \in T^*$) ▷ G ist eine Grammatik und w ein Wort
  $n \Leftarrow |w|$ ▷ Länge des Wortes
  for $i = 1, 2, ..., n$ und alle Nicht-Terminale $A \in N$ **do**
    $M(A, i, i) \leftarrow \begin{cases} 1 & \text{falls } A \to a_i \in P \text{ eine Produktion ist} \\ 0 & \text{sonst} \end{cases}$
  end for
  for k=1,2,...,n-1 **do**
    for i=1,2,...,n-k **do**
      for j=1,2,...,k-1 **do**
        for all $A \to BC \in P$ **do**
          if $M(A, i, i+k) = 0$ **then**
            $M(A, i, i+k) = M(B, i, i+j) \cdot M(C, i+j+1, i+k)$
          end if
        end for
      end for
    end for
  end for
  if $\Big((n > 0)\text{und}\big(M(S, 1, n) = 1\big)\Big)\text{oder}\Big((n = 0)\text{und}\big(S \to \lambda \in P\big)\Big)$ **then**
    w ist ein Wort der Sprache $L(G)$
  else
    w ist kein Wort der Sprache $L(G)$
  end if
end procedure

Der Algorithmus benutzt ein Feld $M(A, i, j)$. Es hat folgende Eigenschaften:

$$M(A, i, j) = \begin{cases} 1 & \text{falls } A \overset{*}{\Rightarrow} a_i a_{i+1} ... a_j \\ 0 & \text{sonst} \end{cases}$$

Dies bedeutet, dass $M(A, i, j) = 1$ genau dann gilt, wenn vom Nicht-Terminal A das Teilwort aus den Zeichen $a_i a_{i+1} ... a_j$ des Wortes w abgeleitet werden kann. Der Algorithmus baut nun sukzessive dieses Feld auf. Am Ende muss nur noch überprüft werden, ob $M(S, 1, n) = 1$ gilt. Dies bedeutet, dass w vom Axiom S abgeleitet werden kann, also $w \in L(G)$ gilt.

Zunächst werden die offensichtlichen Einträge von M gesetzt, also diejenigen, die zu Regeln der Form $A \rightarrow a_i$ gehören. Danach wird für alle Kombinationen von Teilwörtern der Länge k an beliebigen Positionen und allen Regeln $A \rightarrow BC$ der entsprechende Wert von M berechnet. Die For-Schleife und die If-Abfrage innerhalb der drei Schleifen prüft, ob es eine Regel $A \rightarrow BC$ gibt, die das Teilwort $\underbrace{a_i a_{i+1} ... a_{i+j}}_{B \overset{*}{\Rightarrow} ...} \underbrace{a_{i+j+1} a_{i+j+2} ... a_{i+k}}_{C \overset{*}{\Rightarrow} ...}$ erzeugt, wobei B und C die entsprechenden Unterabschnitte des Wortes erzeugen müssen. Die Schleifenvariable k gibt dabei die Länge des Teilwortes, die Variable i seine Position und die Variable j die Position der Teilung des Teilwortes an. Die drei Schleifen durchlaufen also alle denkbaren Teilwörter und deren Teilungen.

Das Wortproblem ist mit diesem Verfahren in $\mathcal{O}(n^3)$ lösbar. Für lineare Grammatiken (CH-0) ist das Wortproblem in linearer Zeit $\mathcal{O}(n)$ und für kontextsensitive Grammatiken in exponentieller Zeit $\mathcal{O}(2^n)$ lösbar.

3.7 Transformation von Grammatiken

Die folgenden Verfahren beziehen sich im wesentlichen auf Typ-2-Grammatiken, also kontextsensitive Grammatiken.

3.7.1 Reduktion

Entfernung unproduktiver Nicht-Terminale: Hier verwendet man ein konstruktives Verfahren, das eine neue Menge N' aller produktiver Nicht-Terminale aufbaut. Das Verfahren läuft so ab:

Algorithm 2 Entfernung unproduktiver Nicht-Terminale

1. Setze $N' = \{\}$.
2. Wiederhole, bis sich an N' nichts mehr ändert:
   a) Zu N' werden alle Nicht-Terminale $A \in N$ zu N' hinzugefügt, zu denen es eine Regel $A \rightarrow \alpha \in P$ gibt, deren rechte Seite $\alpha \in (N' \cup T)^*$ nur aus Terminalen Zeichen und den Nicht-Terminalen aus N' besteht.
3. Ist das Axiom $S \notin N'$ kein Element von N', so werden von G keine Wörter erzeugt und es gilt $L(G) = \{\}$.
4. Nun wird die neue Menge der Produktionen P' aufgebaut, die nur noch Produktionen der Form $A \rightarrow \alpha$ mit $A \in N'$ und $\alpha \in (N' \cup T)^*$ enthält.

Im ersten Schritt werden alle rechten Seiten der Regeln, die nur auf Terminale, oder λ abbilden ($A \rightarrow a|\lambda, \quad a \in T^*$) hinzugefügt, das in N' noch keine Nicht-terminalen enthalten sind. Im nächsten Schritt kommen dann die Regeln hinzu, die rechts Nicht-Terminale enthalten, die nur auf Terminale abbilden usw.

Entfernung unerreichbarer Nicht-Terminale: Zum Ausschluss der unerreichbaren Symbole bestimmt man die Menge Γ, die aus allen Terminalen und Nicht-Terminalen besteht, die vom Axiom S aus erreicht werden können. Die Menge $N'' = \Gamma \cap N'$ enthält dann alle erreichbaren Nicht-terminale und es kann wieder eine entsprechende Menge P'' von Produktionen generiert werden:

Algorithm 3 Entfernung unerriechbarer Nicht-Terminale

1. $\Gamma = \{S\}$
2. Wiederhole bis sich an $N'' = \Gamma \cap N'$ nichts mehr ändert:
   a) Füge zu Γ Alle Symbole $X \in N' \cup T$ hinzu, die von einer Produktion $Y \rightarrow \alpha X \beta$ erzeugt werden, deren linke Seite Y aus $N'' = \Gamma \cap N'$ stammt.
3. $N'' = \Gamma \cap N', \quad T'' = \Gamma \cap T$
4. Die neue Menge der Produktionen P'' besteht aus allen Produktionen aus P', deren linke Seite aus N'' und deren rechte Seite aus Γ stammt: $P'' = \left\{A \rightarrow \alpha \in P' \middle| A \in N'', \quad \alpha \in \Gamma\right\}$

Im ersten Schritt werden alle Symbole hinzugefügt, die vom Axiom S erzeugt werden können. Danach folgen die Symbole, die von Nicht-Terminalen erzeugt werden, in die das Axiom umgewandelt werden kann.

Beispiel: Als Beispiel betrachtet man die folgende Grammatik G:

$$\begin{array}{lcl} S & \rightarrow & A\ \mathtt{a} \mid B \\ A & \rightarrow & \mathtt{ba} \\ B & \rightarrow & \mathtt{b}\ B \\ C & \rightarrow & \mathtt{a}\ A \end{array}$$

Das Symbol C ist offensichtlich unerreichbar und B und unproduktiv, da es bei jeder möglichen Ableitung wieder ein B, also nie einen Satz, sondern immer nur Satzformen erzeugt. Auf diese Grammatik sollen nun die obigen Beiden Algorithmen angewendet werden, um die zugehörige reduzierte Grammatik G_{red} zu erzeugen:

1. Entfernung der unproduktiven Nicht-Terminale:
   a) $N' = \{\}$
   b) $N' = \{A\}$, da nur A in Wörter rein aus Terminalen abgeleitet werden kann
   c) $N' = \{A, S, C\}$, da nur die Regeln $S \rightarrow A\ \mathtt{a}$ und $C \rightarrow \mathtt{a}\ A$ das Symbol A auf der rechten Seite haben.

   Daraus erhält man dann eine neue Menge von Regeln P':

$$\begin{array}{lcl} S & \rightarrow & A\ \mathtt{a} \\ A & \rightarrow & \mathtt{ba} \\ C & \rightarrow & \mathtt{a}\ A \end{array}$$

2. Entfernung von unerreichbaren Symbolen:
   a) $\Gamma = \{S\}$
   b) $\Gamma = \{S, A, \mathtt{a}\}$, da $S \rightarrow A\ \mathtt{a}$
   c) $\Gamma = \{S, A, \mathtt{a}, \mathtt{b}\}$, da $A \rightarrow \mathtt{ba}$

   Daraus erhält man dann abschließend die reduzierte Grammatik G_{red} mit den Regeln $S \rightarrow A\ \mathtt{a}$ und $A \rightarrow \mathtt{ba}$, die die Sprache $L(G) = \{\mathtt{baa}\}$ erzeugt.

3.7.2 Beseitigung von λ-Ableitungen

Hier wird eine Grammatik G in eine λ-freie Grammatik G' überführt. Die neue Grammatik darf also keine Produktionen der Form $A \to \lambda$ enthalten, außer evtl $S \to \lambda$. Man geht Schrittweise vor:

Algorithm 4 Entfernung von λ-regeln

1. Solange G noch Regeln der Form $A \to \lambda, \quad A \neq S$ enthält:
   a) Erzeuge zu allen Regeln $B \to \alpha$, in denen A auf der rechten Seite vorkommt alle Regeln, die durch Löschen eines oder mehrerer A aus α entstehen.
   b) Entferne die Regel $A \to \alpha$
2. Falls $S \to \lambda$ eine Regel ist:
   - Erzeuge wie oben, zu allen Regeln $B \to \alpha$, in denen S auf der rechten Seite vorkommt alle Regeln, die durch Löschen eines oder mehrerer S aus α entstehen.
   - Ersetze überall S durch ein neue Symbol S', außer in $S \to \lambda$.
   - Erweitere $S \quad \to \lambda \mid S'$.

Ist die Regel $S \to \lambda$ vorhanden, so darf S auf keiner rechten Seite einer Produktion vorkommen. Darum wird ein neues Symbol S' eingeführt, das zusammen mit der Regel $S \quad \to \lambda \mid S'$ das alte S ersetzt. Das Symbol S bleibt aber weiterhin Axiom. Alle anderen λ-regeln können entfernt werden, wenn ihre Auswirkung (ein Symbol auf den rechten Seiten der anderen Regeln verschwindet) in die anderen regeln „hineingezogen“ wird.

Beispiel: Man betrachte die folgende Grammatik, die die λ-Regel $T \quad \to \lambda$ enthält:

$$\begin{array}{lcl} S & \to & S \ \texttt{+} \ T \mid T \\ T & \to & \texttt{(} \ S \ \texttt{)} \mid \texttt{id} \mid \lambda \end{array}$$

Das obige Verfahren erzeugt:

1. $$\begin{array}{lcl} S & \to & S \ \texttt{+} \ T \mid S \ \texttt{+} \mid T \mid \lambda \\ T & \to & \texttt{(} \ S \ \texttt{)} \mid \texttt{id} \end{array}$$

2. $$\begin{array}{lcl} S & \to & \lambda \mid S' \\ S' & \to & S' \ \texttt{+} \ T \mid \texttt{+} \ T \mid S' \ \texttt{+} \mid \texttt{+} \mid T \\ T & \to & \texttt{(} \ S \ \texttt{)} \mid \texttt{id} \end{array}$$

3.7.3 Beseitigung von Links-Rekursion

Will man einen Top-Down-Parser bauen, dessen Grammatik eine linksrekursive Regel enthält, so kann dieser Parser in eine Endlosschleife geraten. Dies sieht man einfach an der folgenden Regel $A \quad \to A \ \texttt{a}$. Ein solcher Parser würde beim Ableiten der Variable A zunächst nach einer Ableitung der Variable A suchen. Dies ist also rekursiv und der Parser würde die Funktion für die Variable A unendlich oft rekursiv aufrufen (Endlosschleife).

Eine direkte Linksrekursion $A \quad \to A \ \texttt{a} \mid \texttt{b}$ kann man leicht in eine Rechtsrekursion übersetzen. Eine solche regel erzeugt Wörter der Form **b** , **ba** , **baa** , ... Man führt sie daher in folgende Regeln über:

$$\begin{array}{lcl} A & \to & \texttt{b} \ A' \\ A' & \to & \texttt{a} \ A' \mid \lambda \end{array}$$

Der folgende Algorithmus führt eine λ- und zykelfreie Grammatik G mit $N = \{A_1, A_2, ..., A_n\}$ in eine äquivalente Grammatik G' ohne Linksrekursionen über:

Algorithm 5 Entfernung von Linksrekursionen

1. Eliminiere alle direkten Linksrekursionen für $A_1 \in N$ nach obigem Schema
2. Für alle Nicht-Terminale $A_i \neq A_1$ $(i = 2, 3, ..., n)$:
   a) Für alle Nicht-Terminale A_j mit $j = 1, 2, ..., i-1$:
      i. Ersetze alle regeln der Form $A_i \rightarrow A_j u$ durch $A_i \rightarrow v_1 u | v_2 u | ... | v_k u$, wobei $A_j \rightarrow v_1 | v_2 | ... | v_k$ alle A_j-Produktionen sind.
      ii. Wenn eine direkte Linksrekursion für A_i vorkommt, eliminiere diese.
3. Eliminiere evtl. entstandene λ-Produktionen

Direkte Linksrekursionen können leicht entfernt werden. Darum versucht der obige Algorithmus direkte Linksrekursionen zu konstruieren, wenn die Linksrekursionen erst nach mehreren Ableitungsschritten auftauchen. Dazu setzt er die Regeln ineinander ein und erzeugt so neue äquivalente Regeln, bis entweder Kombinationen durchprobiert sind. Dabei werden entstehende einfache Linksrekursionen entfernt.

Beispiel: Betrachte die Grammatik

$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \mathtt{a} \\ A_2 & \rightarrow & A_1\ \mathtt{b}\ |\ \mathtt{c} \end{array}$$

Diese Grammatik enthält keine direkten Linksrekursionen, nach zweimaligem Ableiten tritt aber eine Linksrekursion aus: $A_1 \Rightarrow A_2 a \Rightarrow A_1 ba$. Der Algorithmus liefert:

1. Zunächst werden die A_1-Regeln in alle anderen Regeln eingesetzt:
$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \mathtt{a} \\ A_2 & \rightarrow & A_2\ \mathtt{ab}\ |\ \mathtt{c} \end{array}$$
2. Nun wird die entstehende Linksrekursion aufgelöst. Die zweite Regel erzeug die Sprache {**c** , **cab** , **cabab** ...}. Also kann man setzen:
$$\begin{array}{lcl} A_1 & \rightarrow & A_2\ \mathtt{a} \\ A_2 & \rightarrow & \mathtt{c}\ A_2' \\ A_2' & \rightarrow & \mathtt{ab}\ A_2' \end{array}$$

3.7.4 Linksfaktorisierung

Beim Parsen von Ausdrücken kann es hilfreich sein, wenn zwei Regeln zum selben Nicht-Terminal nicht mit der gleichen Zeichenfolge beginnen, also den gleichen Präfix haben. Dann ist es nämlich eindeutig möglich die richtige Regel auszuwählen. Man kann folgende Transformation solange anwenden, wie noch gleiche Präfixe vorhanden sind:

- Man betrachte die Regeln $A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_n$, wobei α der gemeinsame maximale Präfix ist. Die β_i haben keine gemeinsamen Präfixe. Außerdem ist α kein Präfix der γ_i.
- Ersetzte dann obige Regeln durch:
$$\begin{array}{lcl} A & \rightarrow & \alpha A' \mid \gamma_1 \mid ... \mid \gamma_n \\ A' & \rightarrow & \beta_1 \mid ... \mid \beta_n \end{array}$$

Beispiel: betrachte

$$A \quad \rightarrow \quad \mathbf{aa}\ B\ \mathbf{b}\ \mid \mathbf{aa}\ B\ \mathbf{c}\ \mid \mathbf{aa}\ C\ \mid \mathbf{aa}\ D\ \mid \mathbf{b}\ \mid \mathbf{c}$$

Daraus erhält man dann:

$$\begin{array}{lcl} A & \rightarrow & \mathbf{aa}\ B\ A'\ \mid \mathbf{aa}\ C\ \mid \mathbf{aa}\ D\ \mid \mathbf{b}\ \mid \mathbf{c} \\ A' & \rightarrow & \mathbf{b}\ \mid \mathbf{c} \end{array}$$

Ein weiterer Iterationsschritt führt dann zu:

$$\begin{array}{lcl} A & \rightarrow & \mathbf{aa}\ A''\ \mid \mathbf{b}\ \mid \mathbf{c} \\ A' & \rightarrow & \mathbf{b}\ \mid \mathbf{c} \\ A'' & \rightarrow & B\ A'\ \mid C\ \mid D \end{array}$$

4 Berechenbarkeitstheorie und Äquivalenzsatz

4.1 Algorithmenbegriff und Berechenbarkeit

intuitive Definition: Ein Algorithmus A ist eine Rechenvorschrift zur Lösung eines Problems. An einen Algorithmus A werden im Allgemeinen folgende Anforderungen gestellt:

- **Finitheit:** Die Beschreibung von A soll endlich sein.
- **Determiniertheit:** Das Ergebnis von A hängt nur von den Eingaben ab, d.h. mehrmaliges Ausführen von A mit den selben Eingaben führt immer zu den selben Ergebnissen.
- **Effektivität:** A ist tatsächlich ausführbar, d.h. die Beschreibung von A besteht nur aus primitiv ausführbaren Anweisungen. Der ALgorithmus kann also schrittweise ausgeführt werden, indem pro Schritt eine dieser Anweisungen ausgeführt wird.

$$\text{Eingabe } x \xrightarrow{\textbf{Algorithmus } A} \text{Ausgabe } y = \text{res}_A(x)$$

Eigenschaften von Algorithmen:

- **Terminierung:** Ein Algorithmus A terminiert g.d.w. das Ergebnis nach endlich vielen Ausführungsschritten vorliegt.
- **Totalität:** Ein Algorithmus A heißt total, wenn er für jede Eingabe terminiert.
- **Resultatsfunktion:** Bildet ein Algorithmus A eine Eingabe x auf eine Ausgabe y ab, so berechnet er damit sein Resultatsfunktion res_A:

$$\text{res}_A(x) : \begin{cases} I & \to & O \\ x & \mapsto & y = \text{res}_A(x) \end{cases}$$

Berechenbarkeit: Eine partielle Funktion $f : I \to O$ heißt **partiell berechenbar**, falls man einen Algorithmus A (also ein Berechnungsverfahren) angeben kann, dass die Funktion partiell berechnet, also für bestimmte $x \in I$ ein Ergebnis $y \in O$ liefert:

$$f(x) := \begin{cases} y & \text{falls } A \text{ terminiert und } y \text{ liefert} \\ \uparrow & \text{falls } A \text{ nicht terminiert} \end{cases}.$$

Man nennt die Funktion f auch Resultatsfunktion des Algorithmus A und schreibt: $f \equiv \text{res}_A$. Man nennt f total berechenbar, falls A für jede Eingabe x terminiert. Außerdem nennt man:

- **Definitionsbereich:** $\text{Db}(f) := \{x \in I \big| f(x)\}$
- **Wertebereich:** $\text{Wb}(f) := \{f(x) \in O \big| x \in I \ \wedge \ f(x)\downarrow\}$

Entscheidbarkeit:

- Ein **totales Entscheidungsverfahren** ist ein Algorithmus A, der für jede Eingabe x immer *true* $(+/1)$ oder *false* $(-/0)$ liefert.
  Die von A (bzw. dem Entscheidungsverfahren) **akzeptierte Menge** ist

$$L_A := \{x \in I \big| \mathrm{res}_A(x) = 1\}$$

- Ein **partielles Entscheidungsverfahren** ist ein Algorithmus A, der für jede Eingabe $x \in L_A$ immer *true* $(+/1)$ liefert, für $x \notin L_a$ nicht terminiert ($\mathrm{res}_A(x)\uparrow$).
- Eine Menge M heißt **(partiell) entscheidbar**, falls es ein (partielles) Entscheidungsverfahren A gibt, sodass $M = L_A$.
- Jede endliche Menge ist entscheidbar.
- Sprache L entscheidbar $\Leftrightarrow$ $\overline{L}$ entscheidbar

Aufzählbarkeit:

- Ein **Aufzählverfahren** ist ein Algorithmus A, der ohne Eingabe potenziell unendlich lange läuft und sicher jedes Element der aufzuzählenden Menge mindestens einmal ausgibt. Dabei ist die Reihenfolge beliebig
- Die von einem Aufzählungsverfahren A aufgezählte Menge L_A ist:

$$L_A := \{x \in I \big| A \text{ gibt } x \text{ aus}\}$$

- Ein Aufzählverfahren A heißt **monoton**, falls die Elemente in der kanonischen Reihenfolge ausgegeben werden.
- Eine totale Funktion f heißt **Aufzählungsfunktion** einer Menge A, falls f berechenbar und ihr Wertebereich gleich A ist: $\mathrm{Wb}(f) = A$

Zusammenhang zwischen Berechenbarkeit, Entscheidbarkeit und Aufzählbarkeit:

- Für eine Menge M definiert man die charakteristische Funktion c_M und die partielle charakteristische Funktion χ_M:

$$c_M(x) := \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases} \qquad \chi_M(x) := \begin{cases} 1 & x \in M \\ \uparrow & x \notin M \end{cases}$$

- Für Sprachen $L \subseteq \Sigma^*$ gilt:
  - L entscheidbar $\iff$ L monoton aufzählbar und damit aufzählbar
  - L entscheidbar $\iff$ $L, \overline{L}$ aufzählbar
  - L entscheidbar $\iff$ c_L berechenbar
  - L aufzählbar $\iff$ L partiell entscheidbar
  - L aufzählbar $\iff$ χ_L partiell berechenbar
  - L aufzählbar $\iff$ L besitzt Aufzählungsfunktion

Sätze über Berechenbarkeit, Entscheidbarkeit und Aufzählbarkeit bei Funktionen:

- Für eine partiell berechenbare Funktion $\varphi : \Sigma^* \to T^*$ ist der Definitionsbereich $\mathrm{Db}(\varphi)$ aufzählbar.
- Eine SPrache ist genau dann aufzählbar, wenn sie Definitionsbereich einer partiell berechenbaren Funktion ist.
- Eine totale Funktion $f : \Sigma^* \to T^*$ ist genau dann berechenbar, wenn ihr Graph $G_f \subseteq \Sigma^* \times T^*$ entscheidbar ist.
- Eine partielle Funktion $\varphi : \Sigma^* \to T^*$ ist genau dann partiell berechenbar, wenn ihr Graph $G_\varphi \subseteq \Sigma^* \times T^*$ aufzählbar ist.
-

4.2 Mathematische Maschinen

Im folgenden wird der Berechnungsbegriff formalisiert, d.h. man gibt Systeme und Maschinen an, in denen man berechenbare Funktionen ausdrücken kann. In diesem Abschnitt werden Berechnungssysteme und sog. mathematische Maschinen betrachtet. Die nächsten Kapitel 4.3 und 4.4 befassen sich dann mit Turing- und Register-Maschinen, die eine gewisse Veranschaulichung der hier angegebenen Konzepte darstellen.

Will man den Berechnungsbegriff formalisieren, so muss man zunächst den Ablauf einer Rechnung formalisieren. Eine belibige Rechnung läuft für gewöhnlich in kleinen Schritten ab. Man kann ihn also als Abfolge von Zwischenergebnissen (sog. **Konfigurationen**) beschreiben. Eine Solche Konfigurationsfolge wird durch ein **Umformungssystem** beschrieben. Fügt man noch Möglichkeiten zur Ein- und Ausgabe hinzu, so erhält man ein **Rechensystem**.

Weiter definiert man sog. **mathematische Maschinen**, die eine bestimmte Berechnung (bzw. einen bestimmten Algorithmus) ausführen. Eine solche mathematische Maschine besteht aus einer **Basismaschine**, die die möglichen Operationen festlegt und einem Basismaschinen-Programm, dass den Ablauf der Rechnung steuert und nur die Operationen der Basismaschine verwendet.

Die später erläuterten Konzepte von Turing- und Registermaschinen stellen im wesentlichen spezielle Basismaschinen dar.

Umformungssystem

- **Definition:** Sei KON eine Menge von *Konfigurationen* und $\to$ eine zweistellige Relation auf KON. $\to$ ordnet jeder Konfiguration $c \in$ KON mindestens eine Folgekonfiguration $c' \in$ KON zu. Als **Umformungssystem** U bezeichnet man nun das Paar

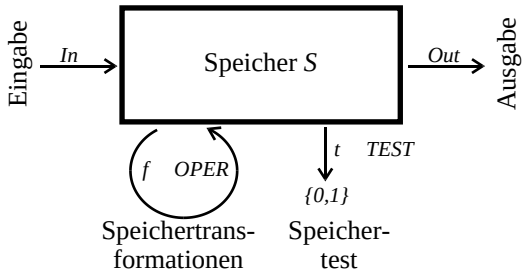
  $$U = (\text{KON}, \to)$$

  Im Allgemeinen sind Umformungssysteme nicht-deterministisch, d.h. es kann zu einer gegeben Konfiguration $c \in$ KON mehrere Folgekonfiguratione $c', c'', \ldots \in$ KON geben. Deshalb ist eine weitere Spezialisierung sinnvoll.
- **deterministische Umformungssysteme:** Ein Umformungssystem U heißt **deterministisch**, falls es zu jeder Konfiguration $c \in$ KON maximal eine (also eine oder keine) Nachfolgekonfiguration $c' \in$ KON mit $c \to c'$ gibt. Existiert zu einem $c_s \in$ KON keine Nachfolgekonfiguration, so heißt c_s Stoppkonfiguration und beendet die Berechnung.
- **Schreibweisen:**
  - $c \xrightarrow{n} c'$: endliche Rechnung, die c in n Schritten in c' überführt.
  - $c \xrightarrow{*} c'$: endliche Rechnung, die c in unbekannt, aber endlich vielen Schritten in c' überführt.
- Eine U-Rechnung heißt **maximal**, falls sie unendlich ist, oder endlich und die letzte Konfiguration eine Stoppkonfiguration ist.

Rechensystem

- **Definition:** Sei $U = (\text{KON}, \rightarrow)$ ein Umformungssystem mit der Konfigurationsmenge KON und einer binären Relation $\rightarrow$ über KON. Weiter seien I und O die sog. Eingabe- und Ausgabemengen, mit den Funktionen $\text{In} : I \mapsto \text{KON}$ und $\text{Out} : \text{KON} \mapsto O$. Man nennt dann das folgende Tupel R eine *Rechenmaschine:*
$$R = \big(I, O, \text{KON}, \text{In}, \text{Out}, \rightarrow\big)$$
- **Funktionsweise:** Zuerst wird ein Element $x \in I$ der Eingabemenge mittels In in die Startkonfiguration $\text{In}(x)$ umgewandelt. Eine R-Rechnung mit der Eingabe x ist dann eine U-Rechnung, die mit $\text{In}(x) \in \text{KON}$ beginnt. Das Ergebnis c' der U-Rechnung ($\text{In}(x) \xrightarrow{*} c'$) wird dann mittels Out auf eine Ausgabe $\text{Out}(c') = y \in O$ abgebildet.
- **Ergebnisrelation:** Man definiert gemäß der obigen anschaulichen Erklärung die sog. Ergebnisrelation $\text{RES}_R \subset I \times O$:
$$(x, y) \in \text{RES}_R \quad :\Leftrightarrow \quad \exists c \in \text{KON} : \big(\text{In}(x) \xrightarrow{*} c \text{ und } c \text{ Stoppkonfiguration und } \text{Out}(c) = y\big)$$
- R heißt *deterministisch*, falls U deterministisch ist.
- Ist R deterministisch, so ist RES_R Graph einer partiellen Funktion, d.h. zu jedem $x \in I$ gibt es ein oder kein $y \in O$, mit $(x, y) \in \text{RES}_R$.

Mathematische Maschine Eine Mathematische Basismaschine besteht ist im wesentlichen ein Speicher S auf dem bestimmte Operationen OPER und Tests TEST erlaubt sind:



Um die mathematische Maschine zu koplettieren benötigt man noch ein Basismaschinen-Programm, dass den Ablauf der Berechnung steuert.

- **Basismaschine:**
  - I, O seien Ein- und Ausgabemengen
  - $\text{in} : I \rightarrow S$ und $\text{out} : S \rightarrow O$ seien Ein- und Ausgabefunktionen
  - S sei die Menge aller möglichen Speicherbelegungen
  - $\text{OPER} = \{f_1, ..., f_n\}$ bezeichnet die Menge der möglichen Speicheroperationen
$$f_i : S \rightarrow S$$
  - $\text{TEST} = \{t_1, ..., t_n\}$ bezeichnet die Menge der möglichen elementaren Speichertests
$$t_i : S \rightarrow \{0, 1\}$$

  Mit obigen Voraussetzungen definiert man eine Basismaschine B als:
$$B = \big(I, O, S, \text{in}, \text{out}, \text{OPER}, \text{TEST}\big)$$
- **Basismaschinen-Programm:** Ein Basismaschinen-Programm oder B-Programm P zu einer Basismaschine B ist eine endliche Menge $P = \{I_0, ..., I_n\}$ von Instruktionen I_k:

- *Operationsinstruktionen:* $I = (i, f, j) \in \mathbb{N} \times \textsc{Oper} \times \mathbb{N}$
  i ist der aktuelle Zustand/die Adresse des Befehls
  j ist der Folgebefehl
  $f \in \textsc{Oper}$ ist die auszuführende Speicheroperation
- *Testinstruktionen:* $I = (i, t, j, k) \in \mathbb{N} \times \textsc{Test} \times \mathbb{N} \times \mathbb{N}$
  i ist der aktuelle Zustand/die Adresse des Befehls
  $t \in \textsc{Test}$ ist der auszuführende Test
  j ist der Folgezustand im Falle $t(S) = 0$
  k ist der Folgezustand im Falle $t(S) = 1$

Ein B-Programm heißt deterministisch, falls: $I \neq I' \;\Rightarrow\; i \neq i'$
An jeder Programmadresse steht also maximal ein Befehl. Das Programm ist also eindeutig.

- **mathematische Maschine** Das Tupel $M = (B, P)$ aus einer Basismaschine B und einem B-Programm P wird als *mathematische Maschine* bezeichnet.
- **Arbeitsweise einer Maschine** $M = (B, P)$**:** Man ordnet M ein Rechensystem $R(M)$ zu:

$$R(M) = \big(I, O, \textsc{Kon}, \mathrm{In}, \mathrm{Out}, \rightarrow\big).$$

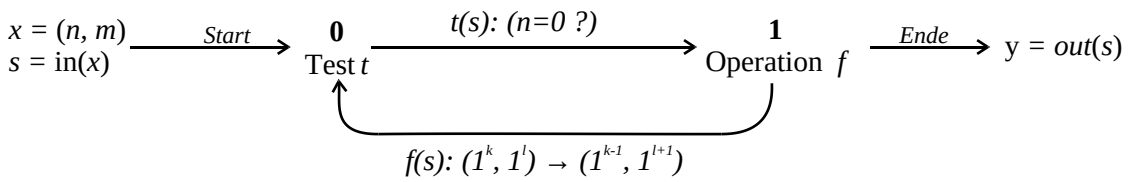
  - $\textsc{Kon} = Z_P \times S$. Dabei bezeichnet $Z_P \subset \mathbb{N}$ die endliche Menge der möglichen Befehlsadressen $i \in \mathbb{N}$ aus dem Programm P.
  - $\mathrm{In}(x) := \big(0, \mathrm{in}(x)\big)$
  - $\mathrm{Out}(i, s) := \mathrm{out}(s), \quad s \in S$
  - $(i, s) \rightarrow \begin{cases} \big(j, f(s)\big) & \text{falls } (i, f, j) \in P \\ \big(j, s\big) & \text{falls } (i, t, j, k) \in P \text{ und } t(s) = 0 \\ \big(k, s\big) & \text{falls } (i, t, j, k) \in P \text{ und } t(s) = 1 \end{cases}$

Beispiel: unäre Addition mit einer mathematischen Maschine: Die unäre Addition entspricht im Wesentlichen der Konkatenation der zwei Zeichenfolgen. Man verwende zur unären Addition folgende Basismaschine:

$$\begin{aligned} I &= \mathbb{N} \times \mathbb{N}; & O &= \mathbb{N} \\ S &= \{1\}^* \times \{1\}^* \\ \mathrm{in}(n, m) &= (1^n, 1^m); & \mathrm{out}(1^n, 1^m) &= m \\ \textsc{Oper} &= \{f\}: & f(1^n, 1^m) &= (1^{n \dot{-} 1}, 1^{m+1}) \\ \textsc{Test} &= \{t\}: & t(1^n, 1^m) &= 1 \Leftrightarrow n = 1 \end{aligned}$$

Der Speicher enthält also zwei unäre Zahlen ($S = \{1\}^* \times \{1\}^*$), die am Anfang mit den beiden Eingaben n und m belegt werden. Das Ergebnis soll hernach in der zweiten unären Zahl stehen. Das folgende Programm führt die Addition aus, indem es solange vom ersten Speicherwort eine 1 wegnimmt und beim zweiten anhängt, bis im ersten Speichertwort die Zahl 0, also genau eine 1 steht. Das Verschieben der 1en erledigt die Operation f und den Test auf 0 der ersten Zahl ist in t implementiert. Das Programm hat nur zwei Befehle und den Endzustand 2:

$$P = \big\{(0, t, 1, 2), (1, f, 0)\big\}$$



Varianten des Instruktionsformates: Man kann die zwei verschiedenen Instruktionen, die oben definiert wurden, auch als eine sog. **bedingte Instruktion** zusammenfassen:

$$I = (i, t, f, j)$$

Dies bedeutet:

Falls akt. Zustand$= i$ und $t(s) = 1$ **dann:** führe f aus $(s \mapsto f(s))$ und gehe in Zustand j über.

Resultatsfunktion einer mathematischen Maschine: Sei $M = (B, P)$ eine deterministische mathematische Maschine und $R(M)$ das ihr zugeordnete Rechensystem. Dann bezeichnet man mit $\varphi_M = \mathrm{res}_M := \mathrm{res}_{R(M)}$ die Resultatsfunktion der mathematischen Maschine, als diejenige (partielle) Funktion, die von M berechnet wird.

B-Berechenbarkeit: Man nennt eine (partielle) Funktion $\varphi : I \to O$ *B-berechenbar*, falls es ein B-Programm P gibt, sodass P zusammen mit B die Funktion φ berechnet:

$$\varphi(x) = \mathrm{res}_{(B,P)}(x).$$

Äquivalenz von mathematische Maschinen: Man nennt zwei Maschinen M und M' äquivalent, wenn sie die selbe Funktion berechnen:

$$\mathrm{res}_M = \mathrm{res}_{M'}\,.$$

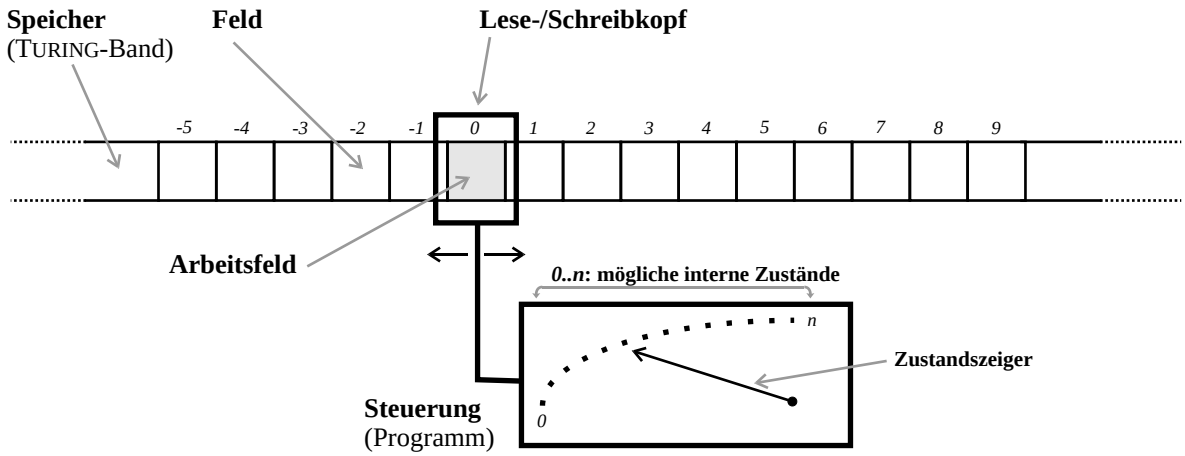
Äquivalenz von Basismaschinen: Man nennt zwei Basismaschinen B und B' äquivalent, es zu jedem B-Programm ein B'-Programm gibt, dass die selbe Funktion berechnet und umgekehrt:

$$\forall M = (B, P)\ \exists M' = (B', P') : M, M' \text{ äquivalent}$$
$$\text{und:}\quad \forall M' = (B', P')\ \exists M = (B, P) : M, M' \text{ äquivalent}$$

4.3 TURING-Maschinen

4.3.1 Grundversion der TURING-Maschine

Die folgende Abbildung zeigt eine schematische Skizze einer Turingmaschine (hier: zweiseitig unendliches Band):

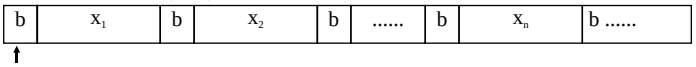


- Jedes Feld des Bandes (Speicher) enthält genau einen Buchstaben des Bandalphabets Γ. Ein Feld kann auch leer sein. Dies realisiert man durch das spezielle Blank-Zeichen $b \in \Gamma$.
- Der Lese-Schreibkopf kann folgende Aktionen ausführen:
  - einen Buchstaben vom Arbeitsfeld (AF) lesen.
  - das Arbeitsfeld mit einem Buchstaben überdrucken
  - den Kopf um ein Feld nach rechts oder links verschieben, bzw. stehen bleiben
- Die Felder wurden zwar in obiger Darstellung durchnummeriert, die Maschine selber hat aber keine inhärente Information darüber, wo sie sich befindet. Eine solche Information muss ebenfalls auf dem Band kodiert werden (z.B. durch Begrenzungszeichen o.ä.).

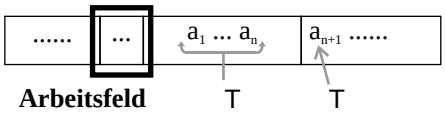
formale Definition: Eine Basis-Turingmaschine $\mathrm{TB}(\Sigma, T, \Gamma, n)$ zur Berechnung von n-stelligen Alphabets-Funktionen $\varphi : (\Sigma^*)^n \to T^*$, über dem Bandalphabet Γ wird als mathematische Basismaschine definiert:

$$TB(\Sigma, T, \Gamma, n) = (I, O, S, \mathrm{in}, \mathrm{out}, \textsc{Oper}, \textsc{Test}).$$

Dabei gelten folgende Setzungen:

- Σ: Eingabealphabet $\Rightarrow$ $I = (\Sigma^*)^n$
- T: Ausgabealphabet $\Rightarrow$ $O = T^*$
- $S = \mathrm{BI} \times \mathbb{Z}$: Speicher = Beschriftung des Bandes + Position des Arbeitsfeldes.
  Dabei ist
  $$\mathrm{BI} = \left\{ f : \mathbb{Z} \to \Gamma \middle| f(z) = b \text{ für fast alle } z \right\}$$
  die Menge der Zuordnungsvorschriften f, die einer Adresse $z \in \mathbb{Z}$ eine Beschriftung aus dem Bandalphabet Γ zuordnet, also im Prinzip die Menge der möglichen Beschriftungen eines Turing-Bandes. Im folgenden wird die Funktion $f(z)$ verwendet. Dies gibt das Zeichen zurück, das sich gerade auf dem Feld $z \in \mathbb{Z}$ befindet. Da die Turingmaschine keine Information über ihren aktuellen Aufenthaltsort hat, kann der Parameter z nur das Arbeitsfeld sein.
- Eingabefunktion in:
  $$(\Sigma^*)^n \to \mathrm{BI} \times \mathbb{Z}$$
  $(x_1, ..., x_n) \to$ 

- Ausgabefunktion $\mathrm{out} : \mathrm{BI} \times \mathbb{Z} \to T^*$:
  Die Ausgabe der Turingmaschine befindet sich rechts vom Arbeitsfeld am Ende der Berechnung:



a_1 ... a_n: Ausgabewort

Die Ausgabefunktion liefert also alle Zeichen vom Band, bis sie auf das erste Zeichen stößt, das nicht aus dem Ausgabealphabet T ist. Also:

$out(...) = a_1 ... a_n$, falls: AF am Anfang auf dem feld links neben $a_1, a_1 ... a_n \in T^*$ und $a_{n+1} \notin T$.

- $\textsc{Oper} = \overline{\Gamma} \cup \{R, L, S\}$. Dabei bedeutet $\overline{\Gamma} := \left\{ \overline{a} \middle| a \in \Gamma \right\}$, die Menge der Druckoperationen ($\overline{1}$ druckt also 1 auf das Band). Die Operationen R, L, S stehen für das Bewegen des Kopfes nach rechts/links und für das Stehenbleiben. Fomrale Definition:
  - $R(f, z) = (f, z + 1)$: Die Zuordnungsfunktion f bleibt gleich, aber das Arbeitsfeld z wird um eins nach rechts verschoben. Analog: $L(f, z) = (f, z - 1)$, $S(f, z) = (f, z)$.

- $\overline{a}(f,z) = \big(f_{(z,a)}, z\big)$, wobei $f_{(z,a)}$ diejenige Zuordnungsfunktion ist, bei der an der Stelle z das Zeichen a steht und die sonst wie f belegt ist:

$$f_{(z,a)}(z') := \begin{cases} f(z') & z' \neq z \\ a & z' = z \end{cases}$$

- TEST $= \tilde{\Gamma} = \big\{\tilde{a} \big| a \in T\big\}$ ist die Menge der Tests, die 1 zurückgeben, wenn das Arbeitsfeld das entsprechende Zeichen enthält und sonst 0:

$$\tilde{a}(f,z) = \begin{cases} 1 & f(z) = a \\ 0 & f(z) \neq a \end{cases}$$

- Das Befehlsformat ist äquivalent zu demjenigen von mathematischen Maschinen. Es gibt also die möglichen Befehle:
  - **Testinstruktion** $I = (i, \tilde{a}, j, k)$: Im Zustand i, teste das Arbeitsfeld auf das Vorhandensein des Zeichens $a \in \Gamma$ und gehe bei positivem Ergebnis zu k, sont zu j über.
  - **Schreibinstruktionen** $I = (i, \overline{a}, j)$: Im Zustand i, schreibe das Zeichen $a \in \Gamma$ auf das Arbeitsfeld und gehe zum Zustand j über.
  - **Bewegungsintruktionen** $I = (i, B, j)$ (mit $B \in \{R, L, S\}$): Im Zustand i bleibe stehen (S), gehe nach rechts oder links (R, L) und gehe inden Zustand j über.

Bedingte Instruktionen als Variante des TURING-Befehlsformates: Statt dem obigen (recht umständlichen) Befehlsformates kann man auch sog. **bedingte Anweisungen** einführen, mit denen sich viele Turing-Programme wesentlich einfacher darstellen lassen. Beide Konzepte sind aber vollständig äquivalent.
Definition: Sei $\mathrm{TB} = (\Sigma, T, \Gamma, n)$ ein Turing-Basismaschine. Weiter seien $i, j \in \mathbb{Z}$ Programmzustände, $a.a' \in \Gamma$ Zeichen des Bandalphabetes und $B \in \{R, L, S\}$ eine Bewegungsoperation. Dann haben bedingte Instruktionen die Form:

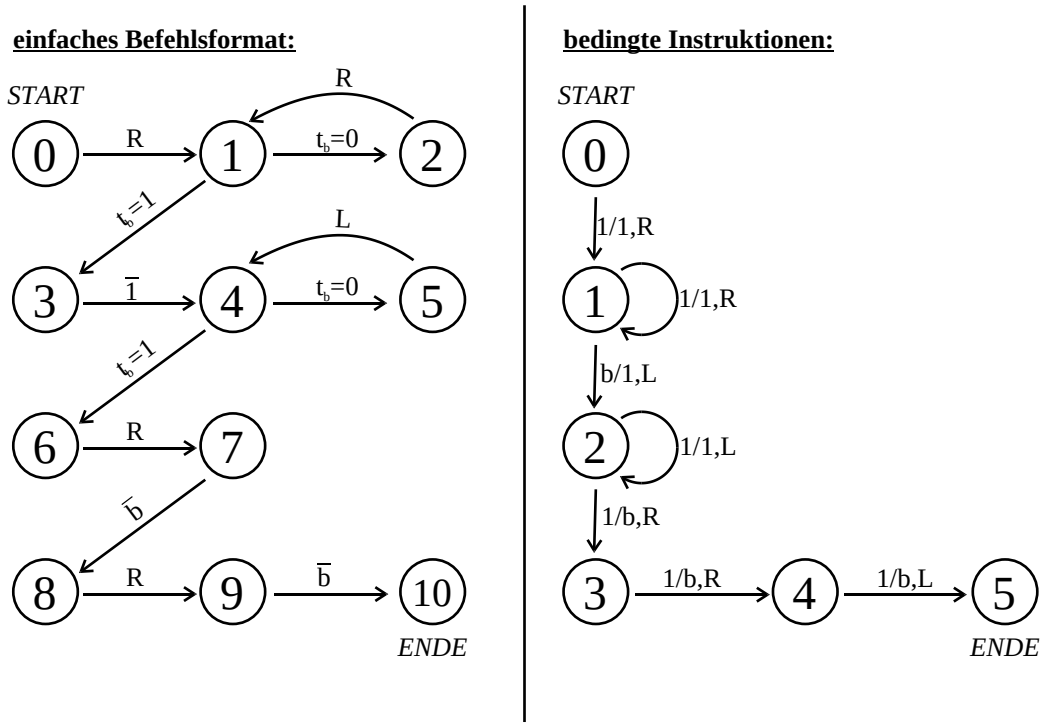
$$I = (i, a, a', B, j).$$

d.h.:

Falls i aktueller Zustand und a Inschrift des Arbeitsfeldes, **dann** schreibe a' auf das Arbeitsfeld, feühre die Bewegung B aus und gehe in den Programmzustand j über.

siehe auch den Absatz Komplexitätsmaß für Turingmaschinen

Beispiel: unäre Addition mit einer Turingmaschine: Folgende Abbildung zeigt Ablaufdiagramme eines Turing-Programmes zur unären Addition in zwei verschiedenen Instruktionsformaten. Beide suchen das erste b (es trennt die zwei Eingabezahlen) und überschreiben es mit einer 1. Danach laufen sie nach links und löschen zwei 1en (Ausgleich für die hinzugef2ugte 1 und die Nulldarstellung!).



4.3.2 Berechenbarkeitsbegriffe und TURING-Maschinen

Beziehung zu mathematischen Maschinen: Da Turing-Maschinen nur eine Verfeinerung von mathematischen Basismaschinen sind (man Spricht von einer Turingmaschine über einer Basismaschine B), ist also das Tupel $M = (\mathrm{TB}, P)$ aus Turing-Basismaschine TB und Turing-Programm P eine mathematische Maschine.

Turing-Berechenbarkeit: Eine partielle Funktion $\varphi : (\Sigma^*)^n \to T^*$ heißt **partiell Turing-berechenbar**, falls es eine Turingmaschine TB und ein Turing-Programm P gibt, das sie berechnet. Ist $\mathrm{res}_{(\mathrm{TB},P)}$ total, so nennt man φ einfach **Turing-berechenbar**.
Dies bedeutet, dass mit der durch TB und P definierten mathematischen Maschine $M := \big(\mathrm{TB}(\Sigma, T, \Gamma, n), P\big)$ gilt:

$$\varphi = \mathrm{res}_M \,.$$

Man bezeichnet die Klasse der Turing-berechenbaren Funktionen mit $F(\mathrm{TM})$.

Turing-Entscheidbarkeit: Eine Menge A heißt **Turing-entscheidbar**, falls ihre charakteristische Funktion c_A Turing-berechenbar ist.

Turing-Aufzählbarkeit: Eine Menge A heißt **Turing-aufzählbar**, falls ihre partielle charakteristische Funktion χ_A Turing-berechenbar ist. (alle partiell entscheidbaren Mengen sind auch aufzählbar)

Church-Turing-These:

Der Begriff Turing-berechenbar und der intuitive Berechenbarkeitsbegriff sind äquivalent.

D.h. Zu jeder berechenbaren Funktion gibt es eine Turingmaschine, die sie berechnet und jede intuitiv berechenbare Funktion ist auch Turing-berechenbar.
ACHTUNG: Hierbei handelt es sich um eine These, also keinen mathematischen Satz. Daher ist die Church-Turing-These nicht beweisbar, allerdings hat sie sich bisher sehr gut bewährt und es wurde noch kein Gegenbeispiel gefunden.

4.3.3 TURING-Operatoren

Dieser Abschnitt führt das Konzept der while-Schleife in Turing-Maschinen ein. Dazu benötigt man die folgenden beiden Definitionen/Konzepte:

- **n-fache Iteration** $\mathrm{Iter}(f,n)(x)$**:** Dieses Konzept entspricht einer for-Schleife:

$$\mathrm{Iter}(f,n)(s) := f(f(...f(s))) = f^n(s)$$

- **Iteration** $\mathrm{Iter}_t(f)(s)$ **nach einem Test** $t : S \to \{0,1\}$**:** Dieses Konzept entspricht der while-Schleife (iteriere f, bis Test erfüllt):

$$\mathrm{Iter}_t(f)(s) := \mathrm{Iter}(f,n_s)(s), \text{ wobei } n_s := \text{kleinestes } n \text{ mit } t\big(f^n(s)\big) = 1$$

Definition: **Turing Operatoren (TO)** werden induktiv über einer Basismaschine definiert. Dabei ist jeder Turing-Operator P eine partielle Funktion $P : S = \mathrm{BI} \times \mathbb{Z} \to S$:

1. **Schreiboperator:** $a \in \Gamma$, a ist Schreib-TO:

$$a(...v\underset{\uparrow}{a'}w...) = (...v\underset{\uparrow}{a}w...)$$

2. **Bewegungsoperator:** $B \in \{L,R,S\}$, B ist Bewegungs-TO:

$$B(...v\underset{\uparrow}{a}a'w...) = (...va\underset{\uparrow}{a'}w...)$$

3. **Hintereinanderausführung:** seien P_1, P_2 TOs, dann ist auch P_1P_2 ein TO:

$$P_1P_2(s) := P_2(P_1(s))$$

4. **Iteration mit Test (while-Schleife):** Sei P ein TO und $a \in \Gamma$, dann ist auch $[P]_a$ ein TO:

$$[P]_a(s) := \mathrm{Iter}_{t_a}(P)(s)$$

   Dabei ist $t_a \in \mathrm{TEST}$ diejenige Testfunktion, die 1 zurückgibt, falls a auf dem Arbeitsfeld steht.

Ergebnisfunktion: Die von einem Turingoperator P bzgl. einer Basismaschine $\mathrm{TB}(\Sigma,T,\Gamma,n)$ berechnete (partielle) Funktion $\mathrm{res}_{:}(\Sigma^*)^n \to T^*$ wird als Ergebnisfunktion des Turing-Operators bezeichnet. Sie ist definiert als:

$$\mathrm{res}_P(\underbrace{w_1,...,w_2}_{=\vec{x}}) := \mathrm{out}\left(P\left(\mathrm{in}\left(w_1,...,w_n\right)\right)\right)$$

Beispiele:

1. $[R]_1$ ist auf einem Band, auf dem keine 1 vorkommt undefiniert.
2. **Unäraddition:** $P_{\text{add}}(...\underset{\uparrow}{b}1^{m+1}b1^{n+1}b...) = (...\underset{\uparrow}{b}1^{m+n+1}b...)$:

$$P_{\text{add}} := R[R]_b 1[L]_b RbRb$$

   Der Ablauf des Additionsoperator Padd ist der selbe, wie im Beispiel der Unäraddition auf Turing-Maschinen.

3. **Rechtsoperatoren/Linksoperatoren:** R_a/L_a verlegt das Arbeitsfeld nach rechts/links bis zum nächsten, mit a beschrifteten Feld:

$$R_a := [R]_a \qquad L_a := [L]_a$$

   R_{aa}/L_{aa} verlegt das Arbeitsfeld nach rechts/links auf die erste Komponente des nächsten Paares 'aa':

$$R_{aa} := R_a R[R_a R]_a L \qquad L_{aa} \text{ entsprechend}$$

4. **Kopieroperator:**

$$K(...v\underset{\uparrow}{b}\underline{m}b\vec{\underline{n}}b...) = (...vb\underline{m}\underset{\uparrow}{b}\underline{vn}b\underline{m}b...)$$

   Kopiere die Unärzahl, die rechts vom AF steht (bis zum ersten 'b') an das rechte Ende der Bandinschrift (markiert durch 'bb'). Der Operator verfährt so, dass er '1' für '1' kopiert und '0' zum merken der Position verwendet:

$$K := R0R_{bb}R1L_0 1R[0R_{bb}1L_0 1R]_b$$

5. **Löschoperator:**

$$E_r(...v\underset{\uparrow}{b}\underline{n}baw...) = (...v\ \underbrace{bb...b}_{n+1 \text{ mal}}\ \underset{\uparrow}{a}w...)$$

$$E_r := R[bR]_b$$

6. **Transportoperator:** Der Transportoperator entfernt Lücken, die aus dem Zeichen 'b' bestehen, indem er die Inschriften rechts der Lücke nach links verschiebt ($l \geq 1$):

$$T(...vb\underline{m}b^l\underset{\uparrow}{b}\underline{n}b...) = (...vb\underline{m}b\underline{n}\underset{\uparrow}{b}...)$$

$$T := ROL_1RR1R_0bR[0L_1R1R_0bR]_bL_1R$$

konservative Turing-Operatoren berechnen res_P so, dass die Eingabe links neben dem Ergebnis erhalten bleibt. Also berechnet P die FUnktion φ konservativ, falls:

$$P(...\underset{\uparrow}{v}bwb...) = \begin{cases} ...\underset{\uparrow}{v}bwb\varphi(w)b... & \text{falls } \varphi(w)\downarrow \\ \uparrow & \text{falls } \varphi(w)\uparrow \end{cases}$$

Turingoperator-berechenbar: Die mit Turngoperatoren berechenbaren Funktionen fasst man zur Klasse $F(\text{TO})$ zusammen. Die konservativ durch TOs berechenbaren Funktionen werden zur Klasse $F_{\text{konservativ}}(\text{TO})$ zusammengefasst.

Beziehung zu Turing-Maschinen: Die Konzepte der Turing-Maschinen und der (konservativen) Turing-Operatoren sind bzgl. der berechenbaren Funktionen Äquivalent:

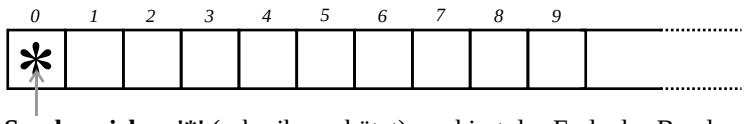
$$F(\mathrm{TO}) = F_{\text{konservativ}}(\mathrm{TO}) = F(\mathrm{TM})$$

Beweisidee für $F(\mathrm{TO}) \subseteq F(\mathrm{TM})$***:*** *Beweis über Induktion nach dem Aufbau der Turing-Operatoren:*
Induktionsanfang: Dir Grundoperatoren Schreiben und Bewegen lassen sich leicht in TM-Programme übersetzen.
Induktionsschritt: Man nimmt die TM-Programm zu den TOs P_1, P_2 *als gegeben an und hängt die Programme einfach hintereinander (ggf. Adressen anpassen!), bzw. konstruiert eine Schleife.*

$$P' = \{(\alpha, t_a, \alpha_1, \omega)\} \cup P_1' \cup \{(\omega_1, S, \alpha)\}$$

4.3.4 Varianten der Bandstruktur von TURING-Maschinen

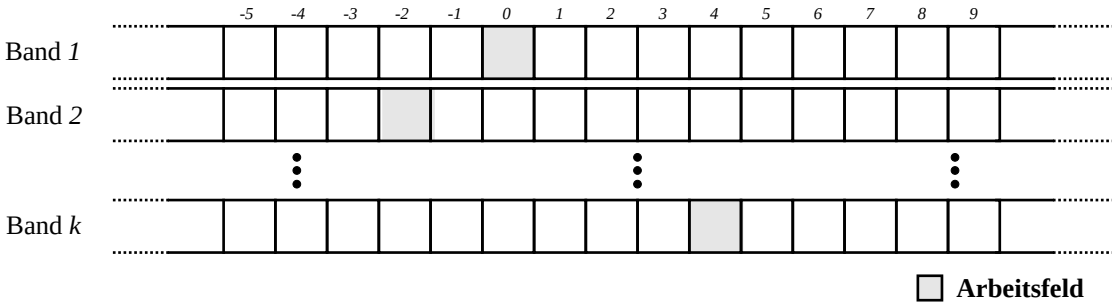
Halbband-Turingmaschine:



Sonderzeichen '*' (schreibgeschützt) markiert das Ende des Bandes

Hier ist das Band einseitig begrenzt und die Felder werden folglich mit Zahlen aus $\mathbb{N}$, satt aus $\mathbb{Z}$ durchnummeriert. Das schreibgeschützte Sonderzeichen $* \in \Gamma$ markiert das (linksseitige) Ende des Bandes. Auf dem Feld mit der Nummer 0, iist also nur die Operation R zugelassen.

k-Band-Turingmaschine:



- Die TM besitzt $k \geq 1$ Bänder, wobei k fest gewählt ist.
- Bedingte Anweisungen hängen jetzt von Programmzustand und von den Inschriften der k Arbeitsfelder ab.
- Es wird neben dem Programmzustand auch die Inschrift und die Position aller Arbeitsfelder aktualisiert.
- *Konvention:* Die Eingabe steht immer auf Band 1 und die Ausgabe immer auf Band k.

Beispiel: unäre Addition mit einer 2-Band-TM: :

$$\left.\begin{array}{l} ...b1^{m+1}b1^{n+1}b... \\ \quad\uparrow \\ ...b..... \\ \quad\uparrow \end{array}\right\} \overset{*}{\Rightarrow} \left\{\begin{array}{l} ...b1^{m+1}b1^{n+1}b... \\ \qquad\qquad\qquad\uparrow \\ ...b1^{m+n+1}b... \\ \quad\uparrow \end{array}\right.$$

Das Vorgehen ist folgendes: Lese das 1. Band von links nach rechts und schreibe gleichzeit die gelsenen !-en auf das zweite Band:

Z	$\times$	Γ	$\times$	Γ	$\longrightarrow$	Z	$\times$	Γ	$\times$	B	$\times$	Γ	$\times$	B
0		b		b	$\rightarrow$	1		b		R		b		S
1		1		b	$\rightarrow$	2		1		R		b		S
2		1		b	$\rightarrow$	2		1		R		1		L
2		b		b	$\rightarrow$	3		b		R		b		S
3		1		b	$\rightarrow$	3		1		R		1		L
3		b		b	$\rightarrow$	4		b		S		b		S

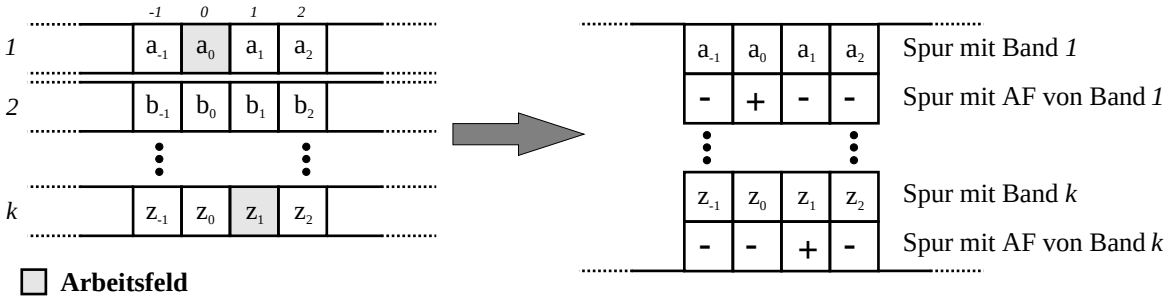
Beziehung zwischen Halbband- und 1-Band-TMs: Zu jeder Halbband-TM M kann effektiv eine äquivalente 1-Band-TM M' gefunden werden.

Beziehung zwischen k-Band- und 1-Band-TMs: Zu jeder k-Band-TM M kann effektiv eine äquivalente 1-Band-TM M' gefunden werden. Für die Rechenzeit von M' gilt mit dem Eingabewort w:

$$\text{time}_{M'}(w) \leq \mathcal{O}\big(\text{time}_M(w)^2\big)$$

Beweisidee: *Benutze die Spurentechnik. Dazu zerlegt man das M'-Band in Spuren, die die M-Bänder beschreiben. Dazu verändert man das Bandalphabet Γ von M hinzu einem Alphabet $\Gamma' = \Gamma^k$, das aus k-Tupeln von zeichen aus Γ besteht: $(a_1, ..., a_b) \in \Gamma^k$.*
Als Verfeinerung speichert man zu jedem Feld in jeder Spur noch ab, ob es gerade Arbeitsfeld ist $(+)$, oder nicht $(-)$. Damit erhält man endgültig:

$$\Gamma' := \big(\{+,-\} \times \Gamma\big)^k$$



Mit diesem Band simuliert man dann die k-Band-Befehle auf einer 1-Band-Maschine. Dazu sind folgende Schritte nötig:

1. *Initialisierung: Bringe die Eingabe in Spurendarstellung*
2. *Simuliere die M-Schritte als M'-Schrittfolgen. dazu:*
   a) *Gehe nach rechts bis zum letzten '+' und merke Dir hierbei die Inschriften der M-Arbeitsfelder im Zustand (=Information, welcher Befehl ausgeführt werden soll).*
   b) *Laufe zurück und führe an den mit '+' markierten Stellen die entsprechenden Befehle aus.*
3. *Aufarbeiten der Ausgabe*

4.3.5 Universelle Turingmaschinen

Existenz universeller Turingmaschinen: Sei Σ ein beliebiges Alphabet und $n \geq 1$. Dann gibt es eine 3-Band-TM $U = U_{\Sigma,n}$ mit Eingabealphabet $\Sigma' = \Sigma \cup \{0,1\}$ und Bandalphabet $\Gamma = \Sigma' \cup \{b\}$ mit:
Zu jeder Turingmaschine M zur Berechnung von n-stelligen Funktionen über Σ gibt es ein Binärwort $w \in \{0,1\}^*$, sodass für alle Eingaben $x \in \Sigma^*$ gilt:

- $\mathrm{res}_U(w,x) = \mathrm{res}_M(x)$
- $\mathrm{time}_U(w,x) = \mathcal{O}(|w| \cdot \mathrm{time}_M(x))$
- $\mathrm{space}_U(w,x) = \mathcal{O}(|w| + \mathrm{space}_M(x))$

Hier kann statt einer 3-Band- auch ein 1-Band-Turingmaschine verwendet werden, wenn man ein Anwachsen der Rechenzeit in Kauf nimmt (Bandreduktionssatz).

Beweisidee:

1. *Normiere TM M (Programm aus bed. Anweisungen, Startzustand 0, ein Stoppzustand,...)*
2. *Kodiere normierte TM $M = (\mathrm{TB}, P)$ durch Binärwörter: $\ulcorner M \urcorner := \ulcorner P \urcorner \in \{0,1\}^*$*
3. *Erstelle U so, dass sie bei Eingabe $(\ulcorner M \urcorner, \vec{x})$ gerade M bei Eingabe $\vec{x}$ Schritt-für-Schritt simuliert.*

Normalformtheorem von KLEENE:

$$\forall n \geq 1 \;\; \exists \text{Funktion } U^{(1)} \text{ und Prädikat } T^{(n+2)} \text{ beide prim. rek.} : \forall \psi^{(n)} \in F(\mathrm{REK}) :$$
$$\exists e \in \mathbb{N} : \;\; \forall \vec{x} \in \mathbb{N}^n : \big(\psi(\vec{x}) = \underbrace{U(\mu y T(e,\vec{x},y))}_{=\varphi(e,\vec{x})=\varphi_e(\vec{x})\in F(\mathrm{REK})}\big)$$

Weiter ist $\varphi_e(\vec{x}) = U(\mu y T(e,\vec{x},y))$ ein Gödelnummerierung der n-stelligen partiellen Funktionen in $F(\mathrm{REK})$.

Dieser Satz sichert also die Existenz einer Gödelnummerierung $\varphi_e(\vec{x}) = U(\mu y T(e,\vec{x},y))$ für die partiell rekursiven Funktionen $F(\mathrm{REK})$ und damit auch für die primitiv rekursiven Funktionen.
Man schreibt statt $\varphi_e^{(n)}(...)$ oft einfach $\{e\}^{(n)}(...)$ für die e-te n-stellige partiell rekursive Funktion. Die explizite Angabe der Stelligkeit kann entfallen, wenn sie klar ersichtlich ist. Also:

$$\{e\}(x) \equiv \varphi_e(\vec{x}).$$

4.3.6 Nichtdeterministische Turingmaschinen

Einführung: Bei deterministischen Turingmaschinen ist der Rechenweg eindeutig festgelegt. D.h. die Maschine hat an keiner Stelle ihrer Rechnung die Auswahl zwischen verschiedenen erlaubten Instruktionen. Für bedingte Instruktionen ergibt das die Bedingung:

$$(i, a, b, B, j) \neq (i', a', b', B', j') \quad \Rightarrow \quad (i, a) \neq (i', a')$$

Bei nicht-deterministischen Turingmaschinen muss diese Bedingung nicht mehr gelten. Sie definieren statt einer eindeutigen Rechnung also einen Rechenbaum, in desse Wurzel die Startkonfiguration steht. Die Blätter sind dann alle erreichbaren Stoppkonfigurationen.
Jeder Pfad in so einem Rechenbaum ist eine eigenständige Rechnung, mit einem eigenen Ergebnis. Darum ist folgende Definition sinnvoll:

akzeptierte Sprachen: Man sagt eine nichtdet. TM M akzeptiert eine Sprache L, wenn ihre N-te mögliche Rechnung (Pfad im Baum) L akzeptiert. D.h. akzeptiert auch nur ein Rechenweg, so wird die Sprache von der TM M akzeptiert.

Totale, nicht-det. Turingmaschinen: Eine nicht-det. TM M heißt **total,** falls jeder mögliche Rechenweg terminiert.

siehe auch das Kapitel 5.3: Komplexität nicht-deterministischer Turingmaschinen.

4.4 Registermaschinen

4.4.1 Registermaschinen

Definition: Ein k-Registermaschine zur Berechnung einer n-stelligen Funktion $\varphi : \mathbb{N}^n \to \mathbb{N}$ hat folgenden Aufbau:

- **Speicher:** Es existieren k Register n_i, die je eine (beliebig große) natürliche Zahl speichern können. Die Eingabe steht in den Registern ab 1. die Ausgabe steht im Register k.
- **Operationen:**
  - **Inkrementieren:** $a_i : n_i \mapsto n_i + 1$ (Erhöhen des i-ten Registers um 1).
  - **Dekrementieren:** $s_i : n_i \mapsto n_i \dot{-} 1$ (Erniedrigen des i-ten Registers um 1).
- **Test:** Es gibt nur die Nulltests $t_i = 1 \quad \Leftrightarrow \quad n_i = 0$.
- **Befehlsformat** wie bei mathematischen Maschinen.

Beispiel: 3-Registermaschine zur Addition: Hierzu werden die zwei Operanden in die Register 1 und 2 geschrieben. Danach dekrementiert man Register 1, bis auf 0 und inkrementiert gleichzeit Register 3. Genauso verfährt man mit register 2:

$(0, t_1, 1, 3)$, $(1, s_1, 2), (2, a_3, 0)$	dekrementiere n_1 und inkementiere gleichzeit n_3
$(3, t_2, 4, 6)$, $(4, s_2, 5), (5, a_3, 3)$	dekrementiere n_2 und inkementiere gleichzeit n_3

0: Startzustand, 6: Endzustand.

4.4.2 Registeroperatoren

Definition: Ein n-Registeroperator (RO) ist eine partielle Funktion $P : \mathbb{N}^n \to \mathbb{N}^n$ und induktiv über eine n-Registermaschine definiert:

1. $a_i(x_1, ..., x_n) = (x_1, ..., x_i + 1, ..., x_n)$ ist Inkrement-RO
2. $s_i(x_1, ..., x_n) = (x_1, ..., x_i - 1, ..., x_n)$ ist Dekrement-RO
3. Seien P_1, P_2 ROs, dann ist auch die hintereinanderausführung P_1P_2 ein RO:

$$P_1P_2(\vec{x}) = P_2(P_1(\vec{x}))$$

4. Sei P ein RO und $1 \leq i \leq n$. Dann ist auch $[P]_i$ ein RO, der einer while-Schleife entspricht (führe P solange aus, bis $n_i = 0$):

$$[P]_i(\vec{x}) := \mathrm{Iter}_{t_i}(P)(\vec{x})$$

primitve Registeroperatoren (PRO): Wenn man obige Regel 4 noch etwas verschärft, so erhält man primitive Registeroperatoren. Diese sind immer total, terminieren also immer:
Regel 4': Sei P ein PRO und $1 \leq i \leq n$. Außerdem darf P nicht auf das Register i zugreifen (kein a_i, s_i) Dann ist auch $[s_iP]_i$ ein PRO, der einer for-Schleife entspricht (führe P n_i-mal aus).

Konservative Registeroperatoren sind wie oben so definiert, dass die Eingabe durch die Rechnung nicht zerstört wird.

Beispiele:

- Addition: $sum := [s_1a_3]_1[s_2a_3]_2$ ist ein primitiver RO
- Registertransfer $T_{i\to j,k}$ ist primitiver RO: kopiere den Inhalt von Register i ins Register j unter Benutzung des Registers k als Hilfsregister:

$$T_{i\to j,k}(...n_i...n_j...n_k...) = (...n_i...n_i...n_k...)$$

$$T_{i\to j,k} := \underbrace{[s_j]_j[s_k]_k}_{n_j,n_k \text{ auf 0 setzen}} \underbrace{[s_ia_ja_k]_i}_{\text{kopiere } a_i\to a_j, a_i=0} \underbrace{[s_ka_i]_k}_{a_i \text{ wiederherstellen}}$$

- Registertransfer mit Löschen $\mathrm{TL}_{i\to j}$ ist primitiver RO:

$$\mathrm{TL}_{i\to j}(...n_i...n_j...) = (...0...n_i...)$$

$$\mathrm{TL}_{i\to j} := [s_j]_j[s_ia_j]_i$$

- konservative Summenberechnung:

$$sum_{kons} := T_{1\to 5,4}T_{2\to 6,4}sumT_{5\to 1,4}T_{6\to 2,4}$$

4.5 rekursive und primitv rekursive Funktionen

Idee: Ausgehend von einfachen, berechenbaren Grundfunktionen werden komplexere Funktionen zusammengesetzt. Das System wird dann gegen einfache Operationen abgeschlossen, die berechenbare in berechenbare Funktionen überführen. So erhält man ein System, mit dem man für bestimmte Funktionen zeigen kann, dass sie berechenbar sind.

Grundlegende Definitionen:

- **Grundfunktionen:**
  - Nachfolger: $S(x) := x + 1$
  - n-stellige Projektion auf i-te Komponente: $U_i^n(\vec{x}) = U_i^n(x_1, ..., x_n) := x_i$
  - n-stellige konstante Funktion: $C_i^n(\vec{x}) := i$
- **simultane Substitution:** Es seien die m-stellige (partielle) Funktion $g : \mathbb{N}^m \to \mathbb{N}$ und die m n-stelligen (partiellen) Funktionen $h_1, ..., h_m : \mathbb{N}^n \to \mathbb{N}$ gegeben. Durch simultane Substitution erhält man dann mit $\vec{x} \in \mathbb{N}^n$ die (partielle) Funktion $f^{(n)}(\vec{x}) = g(h_1(\vec{x}), ..., h_m(\vec{x}))$.
  Die durch simultane Substitution entstandene Funktion $f^{(n)}(\vec{x})$ ist genau dann definiert, wenn alle Teilfunktionen $g, h_1, ..., h_m$ definiert sind:

$$f(\vec{x})\downarrow \quad \Leftrightarrow \quad \exists y_1, ..., y_m : \Big(y_1 = h_1(\vec{x})\downarrow \ \wedge ... \wedge \ y_m = h_m(\vec{x})\downarrow \quad \wedge \quad g(y_1, ..., y_m)\downarrow\Big)$$

- **n-fache Iteration:** Die n-fache Iteration $\mathrm{Iter}(f, n) = f^n$ von f ist rekursiv definiert durch:

$$\begin{aligned} \mathrm{Iter}\big(f, 0\big)(x) &= f^0(x) = x \\ \mathrm{Iter}\big(f, n+1\big)(x) &= f^{n+1}(x) = f(f^n(x)) = f\Big(\mathrm{Iter}\big(f, n\big)(x)\Big) \end{aligned}$$

  Die n-fache Iteration $\mathrm{Iter}_t(f)$ von f nach einem dualen Test $t(x) : x \mapsto \{0, 1\}$ ist definiert als:

$$\mathrm{Iter}_t\big(f\big)(x) = \begin{cases} f^{n_s}(x) & \exists n_s : (n_s \text{ ist kleinstes } n \text{ mit } t(f^n(x)) = 1) \\ \uparrow & \text{sonst} \end{cases}$$

  D.h.: Iteriere f, bis der Test t das erste mal erfüllt ist.
- **primitive Rekursion:** Es seien die (partiellen) Funktionen $g : \mathbb{N}^n \to \mathbb{N}$ und $h : \mathbb{N}^{n+2} \to \mathbb{N}$ gegeben. Dann definiert man die durch primitive Rekursion aus g und h hervorgegangene $n+1$-stellige Funktion $f^{(n+1)} = \mathrm{PR}(g, h)$ als:

$$\begin{aligned} \forall \vec{x} \in \mathbb{N}^n : \qquad & f(\vec{x}, 0) &= g(\vec{x}) \\ \forall \vec{x} \in \mathbb{N}^n \ \forall y \in \mathbb{N} : \qquad & f(\vec{x}, y+1) &= h(\vec{x}, y, f(\vec{x}, y)) \end{aligned}$$

  Beispiel: Die Addition lässt sich rekursiv definieren als:

$$\begin{aligned} x + 0 &= x \\ x + (y+1) &= (x+y) + 1 = S(x+y) \end{aligned}$$

  Mit den Setzungen $g^{(1)}(x) = x = U_1^1(x)$, $h^{(3)}(x, y, z) = S(z)$ bzw. $h^{(3)}(\vec{x}) = S(U_3^3)$ zeigt man, dass die Addition primitiv rekursiv ist durch $f(x, y) = PR(g^1, h^3)$:

$$\begin{aligned} f(x, 0) &= g^{(1)}(x) = U_1^1(x) = x \\ f(x, y+1) &= h^{(3)}(x, y, f(x, y)) = S(f(x, y)) = S(x+y) = (x+y) + 1 \end{aligned}$$

- **simultane primitive Rekursion:** Die $n+1$-stelligen Funktionen $f_1^{(n+1)}, ..., f_m^{(n+1)}$ entstehen aus den n bzw. $n+1+k$-stelligen Funktionen $g_1^{(n)}, ..., g_m^{(n)}$ und $h_1^{(n+1+k)}, ..., h_m^{(n+1+k)}$ durch simultane primitive Rekursion: $(f_1, ..., f_k) = \mathrm{SPR}(g_1, ..., g_k, h_1, ..., h_k)$. Komponentenweise (für alle i mit $1 \leq i \leq k$) gilt:

$$\begin{aligned} f_i(\vec{x}, 0) &= g(\vec{x}) \\ f_i(\vec{x}, n+1) &= h_i(\vec{x}, n, f_1(\vec{x}, n), ..., f_k(\vec{x}, n)) \end{aligned}$$

- **μ-Operator:** Die Funktion

$$f^{(n)} = \mu\big(g^{(n+1)}\big)(\vec{x}) := \mu y\big(g(\vec{x},y) = 0 \;\wedge\; \forall z < y:\; g(\vec{x},z)\downarrow\big)$$

gibt die kleineste Zahl y zurück, die die Eigenschaft $g(\vec{x},y) = 0$ erfüllt.
Der beschränkte μ-Operator ist definiert als (mit gegebenem und konstantem y_0):

$$f^{(n)} = \mu\big(g^{(n+1)}\big)(\vec{x}) := \mu y < y_0\big(g(\vec{x},y) = 0 \;\wedge\; \forall z < y:\; g(\vec{x},z)\downarrow\big)$$

Man kann den μ-Operator auch algorithmisch darstellen:

```
1: z := 0
2: while g(x⃗, z) ≠ 0 do
3:     z := z + 1
4: end while
5: return z
```

F(PRIM) und F(REK) und ihre Eigenschaften:

- **Definition:** Die Klasse der **primitiv rekursiven** Funktionen $F(\text{PRIM})$ ist induktiv definiert durch:
  1. $S, U_i^n, C_j^m \in F(\text{PRIM})$
  2. $g^{(m)}, h_1^{(n)}, ..., h_m^{(n)} \in F(\text{PRIM}) \;\Rightarrow\; g(h_1, ..., h_m) \in F(\text{PRIM})$
  3. $g^{(n)}, h^{(n+2)} \in F(\text{PRIM}) \;\Rightarrow\; PR(g,h) \in F(\text{PRIM})$
- **Definition:** Die Klasse der **partiell rekursiven** Funktionen $F(\text{REK})$ ist induktiv definiert durch:
  1. $S, U_i^n, C_j^m \in F(\text{REK})$
  2. $g^{(m)}, h_1^{(n)}, ..., h_m^{(n)} \in F(\text{PRIM}) \;\Rightarrow\; g(h_1, ..., h_m) \in F(\text{PRIM})$
  3. $g^{(n)}, h^{(n+2)} \in F(\text{PRIM}) \;\Rightarrow\; PR(g,h) \in F(\text{PRIM})$
  4. $g^{(n+1)} \in F(\text{REK}) \;\Rightarrow\; \mu(g) \in F(\text{REK})$
- Es gilt $\boxed{F(\text{PRIM}) \subset F(\text{REK})}$.
- **Definition:** Eine Menge M heißt **(primitiv) rekursiv**, falls ihre charakteristische Funktion c_M (primitiv) rekursiv ist. Man kann Mengen M mit ihren zugehörigen **Prädikaten** P_M identifizieren und erhält damit (primitiv) rekursive Prädikate:

$$P_M(\vec{x}) \;\; (\text{P trifft auf } \vec{x} \text{ zu}) \quad \Leftrightarrow \quad \vec{x} \in M$$

Indem Relationen $r(\vec{x})$ mit mehrstelligen Mengen $M := \{(y,\vec{x}):\; y = r(\vec{x})\}$ identifiziert lassen sich (primitiv) rekursive Funktionen und Relationen auf (primitiv) rekursive Mengen zurückführen (und über c_M umgekehrt).
Man schreibt dann $M \in F(...)$ bzw. $P \in F(...)$.

Beispiele für Funktionen aus F(PRIM) und F(REK):

- folgende Funktionen f sind primitiv rekursiv (also $f \in F(\text{PRIM})$):

$$f(x,y) = x + y \qquad \text{(Summe)}$$
$$f(x,y) = x \cdot y \qquad \text{(Produkt)}$$
$$f(x,y) = x \dot{-} y \qquad \text{(Differenz auf } \mathbb{N}\text{)}$$
$$f(x,y) = |x - y|$$
$$f(x,y) = \max(x,y)$$
$$f(x,y) = \min(x,y)$$
$$f(x,y) = \text{sgn}(x,y) \qquad \text{(Signum / Vorzeichen)}$$
$$f(x,y) = \overline{\text{sgn}}(x,y) \qquad \text{(negiertes Signum / Vorzeichen)}$$

- Die Relationen $=, \neq, <, \leq, >, \geq$ sind primitiv rekursiv.

Abschlusseigenschaften:

- $F(\text{PRIM})$ und $F(\text{REK})$ sind gegen **explizite Definition** abgeschlossen.
- Sei $g^{(n+1)}$ eine (partielle) $n+1$-stellige Funktion. Dann sind $F(\text{PRIM})$ und $F(\text{REK})$ gegen gegen die **beschränkten Summen** $\sum\limits_{i=u}^{o} g^{(n+1)}(\vec{x}, i)$ bzw. $\sum\limits_{i \leq u} g^{(n+1)}(\vec{x}, i)$ und **Produkte** $\prod\limits_{i=u}^{o} g^{(n+1)}(\vec{x}, i)$ bzw. $\prod\limits_{i \leq u} g^{(n+1)}(\vec{x}, i)$ abgeschlossen.
- Sei $g^{(n+1)}$ eine (partielle) $n+1$-stellige Funktion. Dann sind $F(\text{PRIM})$ und $F(\text{REK})$ gegen die **beschränkte Maximums-/Minimumsbildung** abgeschlossen: $\max\big(g\big)(\vec{x}, y) := \max\{g(\vec{x}, i) : \ i < y\}$ bzw. $\min\big(g\big)(\vec{x}, y) := \min\{g(\vec{x}, i) : \ i < y\}$.
- $F(\text{PRIM})$ und $F(\text{REK})$ sind gegen **Iteration** abgeschlossen. D.h. für $f \in F$ gilt $\text{Iter}\big(f, m\big)(\vec{x}) = f^m(\vec{x}) \in F$.
- Die Klasse der primitiv rekursiven Prädikate ist gegen **Einsetzung totaler (primitiver) rekursiver Funktionen** abgeschlossen. D.h. für ein primitiv rekursives Prädikat P und die totalen (primitiv) rekursiven Funktionen $f_1^{(n)}, ..., f_m^{(n)}$ ist $Q(\vec{x}) \quad \Leftrightarrow \quad P(f_1(\vec{x}), ..., f_m(\vec{x}))$ wieder primitiv rekursiv.
- $F(\text{PRIM})$ und $F(\text{REK})$ (Mengen) sind gegen die Aussagenlogischen Junktoren $\neg, \wedge, \vee$ abgeschlossen.
- $F(\text{PRIM})$ und $F(\text{REK})$ (Mengen) sind gegen die beschränkten Existenz- und Allquantoren abgeschlossen. D.h. mit einem (primitiv) rekursiven $n+1$-stelligen Prädikat P sind die Prädikate $P_\forall$ und $P_\exists$ ebenfalls (primitiv) rekursiv, mit:

$$\begin{aligned} P_\exists(\vec{x}, y) \quad &:\Leftrightarrow \quad \exists z < y : \Big(P(\vec{x}, z)\Big) \\ P_\forall(\vec{x}, y) \quad &:\Leftrightarrow \quad \forall z < y : \Big(P(\vec{x}, z)\Big) \end{aligned}$$

- $F(\text{PRIM})$ ist gegen den **beschränkten μ-Operator** abgeschlossen.
- $F(\text{PRIM})$ und $F(\text{REK})$ sind gegen **endliche Fallunterscheidungen** abgeschlossen.
- $F(\text{PRIM})$ und $F(\text{REK})$ sind gegen **simultane Rekursion** abgeschlossen.
- **Endliche Mengen** sind primitiv rekursiv.
- $F(\text{PRIM})$ und $F(\text{REK})$ sind gegen **endliche Varianten** (Menge, die sich nur in endlich vielen Elementen unterscheiden) abgeschlossen.

ACKERMANN-Funktion: Die Ackermann-Funktion $\alpha(x, y)$ ist eine rekursive Funktion, die nicht Primitiv rekursiv ist, also: $\alpha \in F(\text{REK})$, aber $\alpha \notin F(\text{PRIM})$. Die Funktion α steigt schneller, als alle primitiv rekursiven Funktionen. Ihre Zweige α_x sind aber primitiv rekursiv. Sie ist definiert durch:

$$\begin{aligned} \alpha(0, y) &= y + 1 \\ \alpha(x, 0) &= \alpha(x-1, 1) \\ \alpha(x, y) &= \alpha(x-1, \alpha(x, y-1)) \end{aligned}$$

4.6 Äquivalenzsatz

Satz: Alle bisher vorgestellten Berechnungskonzepte sind äquivalent:

$$F(\mathrm{TM}) = F(k - \mathrm{TM}) = F(\mathrm{TO}) = F(\mathrm{RM}) = F(\mathrm{RO}) = F(\mathrm{REK}).$$

Beiweisidee zu $F(\mathrm{TM}) \subseteq F(\mathrm{REK})$:
Man gödelisiere die Syntax und Semantik von Turingmaschinen. Mit Hilfe dieser Kodierung kann man dann die Arbeitsweise von Turingmaschinen mit primitiv rekursiven Funktionen und Prädikaten beschreiben.

4.7 Unentscheidbare Mengen, Diagonalisierungsmethode, Reduktionsmethode

$F_{tot}(\mathrm{REK}), F_{tot}(\mathrm{PRIM})$ enthalten keine universellen Funktionen: Sei F eine Klasse totale (!) Funktionen über $\mathbb{N}$. Außerdem seien $S, U_i^n \in F$ und F sei gegen die simultane Substitution abgeschlossen. Dann besitzt F keine n-universelle Funktion $n \geq 1$.

Beweis:

- *Wiederspruchsannahme: Es gebe eine $n+1$-stellige Funktion $f^{(n+1)} \in F$, die für F n-universell ist.*

$\Rightarrow$ *Es gilt, dass die Funktion*

$$g(x_1, ..., x_n) := S(f(x_1, x_1, ..., x_n)) = f(x_1, x_1, ..., x_n) + 1$$

ebenfalls zu F gehört (siehe Vorraussetzungen des Satzes!).

$\Rightarrow$ *Wegen der n-Universalität von f gibt es also einen Index $e \in \mathbb{N}$, bezüglich dem f die Funktion g darstellt:*

$$f(e, x_1, ..., x_n) = g(x_1, ..., x_n).$$

$\Rightarrow$ *Wählt man e als Eingabe für den ersten Parameter von g, so erhält man zusätzlich:*

$$g(e, x_2, ..., x_n) = f(e, e, x_2, ..., x_n) + 1$$

$\Rightarrow$ *Aus den beiden gerade aufgeschriebenen Gleichungen erhält man sofort einen Wiederspruch:*

$$g(e, x_2, ..., x_n) \underset{1}{=} f(e, e, x_2, ..., x_n) + 1 \underset{2}{=} f(e, e, x_2, ..., x_n)$$

QED.

n-universelle Funktionen und $F(\mathrm{REK})$: Sei $\Psi^{(n+1)}$ eine n-universelle Funktion für $F(\mathrm{REK})$ und $n \geq 1$. Dann ist der Definitionsbereich $\mathrm{Db}(\Psi)$ nicht rekursiv.

Halteprobleme: Das allgemeine Halteproblem ist gegeben durch die Menge der Eingaben und TM-Programme, für die ein TM-Programm anhält. Die TM-Programme lassen sich nach dem Äquivalenzsatz auch durch die Gödelnummerierung $\{e\} = \varphi_e$ der rekursiven Funktionen $F(\mathrm{REK})$ darstellen. Damit erhält man folgende Halteprobleme:

1. **Allgemeines Halteproblem:** $K := \big\{\langle e, x\rangle \,\big|\, \{e\}(x)\downarrow\big\}$
   $\Leftrightarrow$ Die e-te normierte Turingmaschine hält bei Eingabe x.
2. **Diagonales Halteproblem:** $K_d := \big\{e \,\big|\, \{e\}(e)\downarrow\big\}$
   $\Leftrightarrow$ Die e-te normierte Turingmaschine hält, wenn sie auf sich selbst angewandt wird.
3. **Initiales Halteproblem:** $K_0 := \big\{e \,\big|\, \{e\}(0)\downarrow\big\}$
   $\Leftrightarrow$ Die e-te normierte Turingmaschine hält bei Eingabe 0.

Satz: K ist nicht rekursiv.

Many-One-Reduzierung: Seien $A, B \subseteq \mathbb{N}$. A heißt **many-one-reduzierbar** auf B, wenn es eine 1-stellige total rekursive Funktion f gibt, mit:

$$\forall n \in \mathbb{N} : \big(n \in A \Leftrightarrow f(n) \in B\big).$$

Man sagt dann, dass A auf B via f m-reduzierbar ist und man schreibt:

$$A \leq_m B \ \ (\text{via } f).$$

Es gilt:

- Aus einer Beziehung $A \leq_m B$ folgt direkt: $\exists f : A = f^{-}(B)$.
- Es seien $A, B \subseteq \mathbb{N}$ Mengen mit $A \leq_m B$. Dann gilt:
  - A nicht rekursiv $\Rightarrow$ B nicht rekursiv.
  - B rekursiv $\Rightarrow$ A rekursiv.

  Hiermit erhält man sofort aus der Tatsache, dass K nicht-rekursiv ist auch, dass K_0, K_d nicht-rekursiv sind, denn es gilt:
  $$K \leq_m K_0, \qquad K \leq_m K_d.$$

Zwei Mengen A, B heißen **many-one-Äquivalent** ($A =_m B$)**,** wenn gilt:

$$A \leq_m B \quad \text{und} \quad B \leq_m A.$$

Es gilt sogar: $K =_m K_0 =_m K_d$

m-Härte: Sei C ein Klasse von Teilmengen von $\mathbb{N}$. Eine Menge A heißt **m-hart** für C, wenn

$$\forall B \in C : \ \big(B \leq_m A\big)$$

Gilt zusätzlich, dass $A \in C$, so nennt man A **m-vollständig.**
Es gilt:

- Ist A m-hart für C und gilt $A \leq_m B$, so ist auch B m-hart.
- Enthält C eine nicht-rekursive Menge, so ist jede m-harte Menge für C nicht-rekursiv.
- Die Halteprobleme K, K_0, K_d sind m-hart.

4.8 Rekursiv aufzählbare Mengen

Rekursiv aufzählbare Mengen: Eine Menge A heißt **rekursiv aufzählbar,** wenn sie Definitionsbereich einer partiell rekursiven Funktion $\varphi : \mathbb{N}^n \to \mathbb{N}$ ist.

Projektionssatz: Für eine Menge $A \subseteq \mathbb{N}^n$ ist äquivalent:

- A ist rekursiv aufzählbar
- A ist Projektion einer (primitv) rekursiven Menge $B \subseteq \mathbb{N}^{n+1}$, d.h.

$$\forall \vec{x} \in \mathbb{N}^n : \quad \big(\vec{x} \in A \quad \Leftrightarrow \quad \exists y \in \mathbb{N} : ((\vec{x}, y) \in B)\big)$$

Beweis:

'$\Uparrow$' *Sei A die Projektion der rekursiven Menge B. Dann ist $A = \mathrm{Db}(\Psi)$, mit:*

$$\Psi(\vec{x}) = \mu y(B(\vec{x}, y)) = \mu y(c_{\overline{B}}(\vec{x}, y))$$

'$\Downarrow$' *Sei A Definitionsbereich einer partiell rekursiven Funktion $\Psi^{(n)}$. Für einen Index e von Ψ gilt dann nach dem Normalformsatz:*

$$\vec{x} \in A \quad \Leftrightarrow \quad \Psi(\vec{x})\downarrow \quad \Leftrightarrow \quad \exists y : \big(T(e, \vec{x}, y)\big)$$

d.h. A ist die Projektion der primitiv rekursiven Menge $B := \big\{(\vec{x}, y) \big| T(e, \vec{x}, y)\big\}$.

Es gilt:

- Für eine Menge $A \subseteq \mathbb{N}$ sind äquivalent:
  - A ist rekursiv aufzählbar.
  - A ist leer, oder A ist Wertebereich einer einstelligen total rekursiven Funktion über $\mathbb{N}$.
  - A ist der Wertebereich einer einstelligen total rekursiven Funktion über $\mathbb{N}$.
- Eine Menge A ist genau dann rekursiv, wenn A und $\overline{A}$ rekursiv aufzählbar sind.
- Nach dem Projektionssatz ist das Halteproblem K rekursiv aufzählbar, da nach seiner Definition gilt:

  $$\{e\}(x)\downarrow \quad \Leftrightarrow \quad \exists y : \; T(e, x, y).$$

  Also ist K Projektion der rekursiven Menge $B := \big\{(x, y) \big| T(e, x, y)\big\}$
- Seien A, B Mengen, mit $A \leq_m B$. Dann gilt:

  $$B \text{ rekursiv aufzählbar} \quad \Rightarrow \quad A \text{ rekursiv aufzählbar}$$

4.9 Weitere Ergebnisse der Berechenbarkeitstheorie

Fixpunktsatz: Zu jeder 1-stelligen, total rekursiven Funktion f und zu jedem $n \geq 1$ gibt es eine Zahl $k \in \mathbb{N}$ mit:

$$\forall \vec{x} \in \mathbb{N}^n : \quad \big(\varphi(f(k), \vec{x}) = \{f(k)\}(\vec{x}) = \varphi(k, \vec{x}) = \{k\}(\vec{x})\big).$$

Indexmengen: Die Indexmenge $\textsc{Ind}(\Psi)$ einer 1-stelligen partiell rekursiven Funktion Ψ ist die Menge aller Indizes von Ψ:

$$\textsc{Ind}(\Psi) = \big\{e \in \mathbb{N} \big| \{e\} = \Psi\big\}$$

Eine Indexmenge $\textsc{Ind}(F)$ eine Menge F 1-stelliger partiell rekursiver Funktionen ist die Menge:

$$\textsc{Ind}(F) = \bigcup_{\Psi \in F} \textsc{Ind}(\Psi) = \big\{e \in \mathbb{N} \big| \{e\} \in F\big\}.$$

Eine Menge $I \subseteq \mathbb{N}$ heißt Indexmenge, wenn sie Indexmenge einer Menge $F \subseteq F^{(1)}(\mathrm{REK})$ ist. I heißt nichttriviale Indexmenge, wenn $I = \textsc{Ind}(F)$ für $\varnothing \subset F \subset F^{(1)}(\mathrm{REK})$ ist.

Satz von RICE: Sei I eine nichttriviale Indexmenge. Dann ist I nicht rekursiv.

Beweis: *Sei I nicht-triviale, rekursive Indexmenge. Wegen der Nichttrivialität kann man $y, z \in \mathbb{N}$ wählen, mit $y \in I$ und $z \notin I$. Wegen der Rekursivität von I ist die folgende Funktion $f(x)$ total rekursiv:*

$$f(x) = \begin{cases} z & \text{falls } x \in I \\ y & \text{sonst} \end{cases}$$

f bildet also ein $x \in I$ auf ein $z \notin I$ und ein $x \notin I$ auf ein $y \in I$ ab.
$\Rightarrow \{f(x)\}(\cdot) \neq \{x\}(\cdot)\ \forall x$, da I eine Indexmenge ist, müsste es aber nach dem Fixpunktsatz mindestens ein $x \in I$ für jede total-rek. Funktion f geben, für das $\{f(x)\}(\cdot) = \{x\}(\cdot)\ \forall x$ gilt. Also Wiederspruch. QED.

5 Komplexitätstheorie

5.1 Zeit- und Platzkomplexität von Turingmaschinen

Komplexitätsmaß für Turingmaschinen: Sei M eine Turingmaschine.

- $\mathrm{time}_M(x)$ ist die Anzahl der Schritte, bis die Turingmaschine M für die Eingabe x terminiert, also die Länge der M-Rechnung mit der Eingabe x.
- $\mathrm{space}_M(x)$ ist die Anzahl der besuchten Felder einer 1-Band-TM, während einer M-Rechnung mit Eingabe x. Im Falle einer k-Band-TM wird das Maximum über alle einzelnen Bänder genommen.

Definition: Sei $f : \mathbb{N} \to \mathbb{N}$ eine total rekursive Funktion. Eine totale Mehrband-TM M zur Berechnung einer einstelligen Funktion über dem Alphabet Σ ist $f(n)$**-zeitbeschränkt**, falls

$$\overset{\infty}{\forall} x \in \Sigma^* : \quad \big(\mathrm{time}_M(x) \leq f(|x|)\big)$$

und M ist $f(n)$**-platzbeschränkt**, falls

$$\overset{\infty}{\forall} x \in \Sigma^* : \quad \big(\mathrm{space}_M(x) \leq f(|x|)\big)$$

Man definiert folgende Zeitklassen:

- $\mathrm{FDTIME}_k^m(t(n))$ = Menge der totalen Funktionen $f : \Sigma_m^* \to \Sigma_m^*$, die von einer $t(n)$-zeitbeschränkten k-Band-TM berechnet werden.
- $\mathrm{DTIME}_k^m(t(n))$ = Menge der Mengen $A \subset \Sigma_m^*$, dere charakteristische Funktion c_A, von einer $t(n)$-zeitbeschränkten k-Band-TM berechnet werden.

Entsprechend definiert man die Platzklassen:

$$\mathrm{FDSPACE}_k^m(s(n)) \quad \text{und} \quad \mathrm{DSPACE}_k^m(s(n)).$$

Wichtige Zeitkomplexitätsklassen:

- Deterministische Linearzeit: $\mathrm{LIN} := \mathrm{DTIME}(\mathcal{O}(n))$
- Det. Polynomialzeit: $\mathrm{P} = \mathrm{PTIME} := \mathrm{DTIME}(\mathrm{poly}(n))$
- Det. lineare Exponentialzeit: $\mathrm{E} := \mathrm{DTIME}(2^{\mathcal{O}(n)})$
- Det. polynomiale Exponentialzeit: $\mathrm{EXP} := \mathrm{DTIME}(2^{\mathrm{poly}(n)})$

Wichtige Platzkomplexitätsklassen:

- $\mathrm{LOGSPACE} := \mathrm{DSPACE}(\log(\mathcal{O}(n)))$
- $\mathrm{PSPACE} := \mathrm{DSPACE}(\mathrm{poly}(n))$
- $\mathrm{EXPSPACE} := \mathrm{DSPACE}(2^{\mathrm{poly}(n)})$

1. Bandreduktionssatz: Zu jeder $t(n)$-zeit- und $s(n)$-platzbeschränkten Mehrband-TM gibt es eine äquivalente $\mathcal{O}(t(n)^2)$-zeit- und $\mathcal{O}(s(n))$-platzbeschränkte 1-Band-TM. Daraus erhält man sofort:

- $\mathrm{DSPACE}(s(n)) \subseteq \mathrm{DSPACE}_1(\mathcal{O}(s(n)))$
- $\mathrm{DTIME}(t(n)) \subseteq \mathrm{DTIME}_1(\mathcal{O}(t(n)^2))$

2. Bandreduktionssatz: Zu jeder $t(n)$-zeitbeschränkten Mehrband-TM M gibt es eine äquivalente $\mathcal{O}(t(n) \cdot \log(t(n)))$-zeitbeschränkte 2-Band-TM. Daraus erhält man sofort:

$$\mathrm{DTIME}(t(n)) \subseteq \mathrm{DTIME}_2(\mathcal{O}(t(n) \cdot \log(t(n))))$$

Beschränkungen für Turingmaschinen:

- Jede $t(n)$-zeitbeschränkte TM ist auch $t(n)$-platzbeschränkt.
- Jede $s(n)$-platzbeschränkte TM ist auch $\mathcal{O}\left(2^{\mathcal{O}(s(n))}\right)$-zeitbeschränkt.

5.2 Hierarchiesätze

Platzkonstruierbarkeit: Eine rekursive Funktion $s(n) \geq \log(n)$ ist **platzkonstruierbar**, wenn es eine Mehrband-TM M gibt, für die $\mathrm{space}_M(x) = s(|x|)$ für alle Wörter x gibt.

Zeitkonstruierbarkeit: Eine rekursive Funktion $t(n)$ ist **zeitkonstruierbar**, wenn es eine Mehrband-TM M gibt, für die $\mathrm{time}_M(x) = t(|x|)$ für alle Wörter x gibt.

Platzhierarchiesatz: Seien $s(n), S(n) \geq \log(n)$ rekursive Funktionen, für die gilt:

(S1) $S(n) \notin \mathcal{O}(s(n))$, also sind $s(n), S(n)$ von unterschiedlicher Ordnung, unterscheiden sich also um mehr als einen multiplikativen Faktor.

(S2) $S(n)$ platzkonstruierbar

Dann gilt $\mathrm{DSPACE}(S(n)) \nsubseteq \mathrm{DSPACE}(s(n))$. Werden $s(n)$ und $S(n)$ zusätzlich so gewählt, dass $s(n) \leq S(n)$ für fast alle n, so gilt also:

$$\mathrm{DSPACE}(s(n)) \subsetneq \mathrm{DSPACE}(S(n)).$$

Aus dem Platzhierarchiesatz erhält man direkt folgende exten Inklusionen:

- $\mathrm{DSPACE}(\log(n)) \subset \mathrm{DSPACE}(n)$
- $\mathrm{DSPACE}(n^k) \subset \mathrm{DSPACE}(n^k \log(n)) \subset \mathrm{DSPACE}(n^{k+1})$
- $\mathrm{DSPACE}(n^k) \subset \mathrm{DSPACE}(n^{\log(n)})$

Zeithierarchiesatz: Seien $t(n), T(n) > n$ rekursive FUnktionen, für die gilt:

(S1) $T(n) \notin \mathcal{O}(t(n) \log(t(n)))$

(S2) $S(n)$ zeitkonstruierbar

Dann gilt $\mathrm{DTIME}(T(n)) \nsubseteq \mathrm{DTIME}(t(n))$. Sind t und T so gewählt, dass zusätzlich $t(n) \leq T(n)$ fast überall, so gilt:

$$\mathrm{DTIME}(t(n)) \subsetneq \mathrm{DTIME}(T(n)).$$

5.3 Komplexität nicht-deterministischer Turingmaschinen

Zeit- und platzbeschränkte nicht-det. Turingmaschinen: Eine totale nichtdet. TM M heißt $f(n)$-**zeit- bzw. platzbeschränkt,** wenn für jede mögliche Rechnung von M bei der Eingabe x, mit $n = |x|$, jeder mögliche Rechenweg $f(n)$-zeit- bzw. platzbeschränkt ist.

Man definiert damit die nicht-deterministischen Platz- und Zeitkomplexitätsklassen:

- $\mathrm{NTIME}(f(n)) = \left\{L \subseteq \Sigma_2^* \middle| \text{Es gibt eine nicht-det. } f(n)\text{-zeitbeschränkte Mehrband-TM, die } L \text{ erkennt.}\right\}$
- $\mathrm{NSPACE}(f(n)) = \left\{L \subseteq \Sigma_2^* \middle| \text{Es gibt eine nicht-det. } f(n)\text{-platzbeschränkte Mehrband-TM, die } L \text{ erkennt.}\right\}$

Daraus erhält man entsprechend:

- $\mathrm{NLIN} = \mathrm{NTIME}(\mathcal{O}(n))$
- $\mathrm{NP} = \mathrm{NTIME}(\mathrm{poly}(n))$
- $\mathrm{NLOGSPACE} = \mathrm{NSPACE}(\mathcal{O}(\log(n)))$
- $\mathrm{NPSPACE} = \mathrm{NSPACE}(\mathrm{poly}(n))$

P-NP-Problem: Bis heute ist offen, welche Inklusionsbeziehung zwischen den Klassen P und NP gilt ($\mathrm{P} \subsetneq \mathrm{NP}$), oder ob sie sogar gleich sind $\mathrm{P} = \mathrm{NP}$.

5.4 Polynomiell beschränkte Komplexitätsklassen und das P-NP-Problem

Nach der Church-Turing-These ist eine Problem dann entscheidbar, wenn es von einer TM berechnet wird. Allerdings können die Kosten dafür so groß sein, dass es praktisch unlösbar, also praktisch unentscheidbar ist. Dies motiviert die folgende These:

COOK'sche These: Die Klasse der *praktisch/tatsächlich* lösbaren Probleme fällt mit der Klasse der mit polynomiellem Aufwand auf TMs lösbaren Probleme P zusammen. Da die wechselseitige SImulation in Polynomialzeit möglich ist, ist P von der Art der deterministischen (!) Maschine uinabhängig.

Zugehörigkeit zu NP: Eine Menge $A \subseteq \Sigma^*$ liegt genau dann in NP, wenn sie die p-beschränkte Projektion einer 2-stelligen Menge $B \subseteq \Sigma^* \times \Sigma^*$ aus P ist. D.h. es gibt ein Polynom q, mit:

$$\forall x \in \Sigma^* : \quad \Big(x \in A \quad \Leftrightarrow \quad \exists y \in \Sigma^* : \big(|y| \leq q(|x|) \text{ und } (x, y) \in B\big)\Big)$$

Erfüllbarkeitsproblem der Aussagenlogik SAT: Die Menge CNF der aussagenlogischen Formeln in konjunktiver Normalform, ist induktiv definiert:

- Eine abzählbare Menge an **Aussagen-Variablen** $x_1, x_2, \ldots$ und die *Konstanten* 0 (falsch) und 1 (wahr) bilden den Ausgangspunkt und sind Elemente von CNF.
- Ein **Literal** $l \in$ CNF ist eine (evtl. negierte) Variable ($l = x, \neg x$) oder Konstante ($l = i, \neq i, \quad i \in \{0, 1\}$).
- Eine $\vee$**-Klasuel** $c \in$ CNF ist die endliche Disjunktion von Literalen: $c = l_1 \vee l_2 \vee \ldots \vee l_n$.
- Eine **Formel** $\alpha \in$ CNF ist eine endliche Konjunktion von $\vee$-Klauseln: $\alpha = c_1 \wedge c_2 \wedge \ldots \wedge c_n$.

Eine Formel, die keine Variablen enthält heißt **geschlossen**.
Einer geschlossenen Formel α kann man einen Wahrheitswert $w(\alpha)$ zuordnen. Um beliebigen Formeln Wahrheitswerte zuzuordnen, muss man die Variablen mit entsprechenden Werten belegen.
Der Wahrheitswert einer Formel $\alpha \in \text{CNF}$ lässt sich durch einmaliges lesen der Formel ermitteln, d.h.:

$$W = \big\{\alpha \in \text{CNF} \big| \alpha \text{ geschlossen und } w(\alpha) = 1\big\} \in \text{DTIME}(\mathcal{O}(n)).$$

Man nennt eine Formel $\alpha \in \text{CNF}$ erfüllbar, wenn es eine Belegung ihrer Variablen gibt, die sie wahr macht ($w(\alpha) = 1$). Das Erfüllbarkeitsproblem SAT ist damit definiert als:

$$\text{SAT} = \big\{\alpha \in \text{CNF} \big| \alpha \text{ erfüllbar}\big\}.$$

Es gilt:

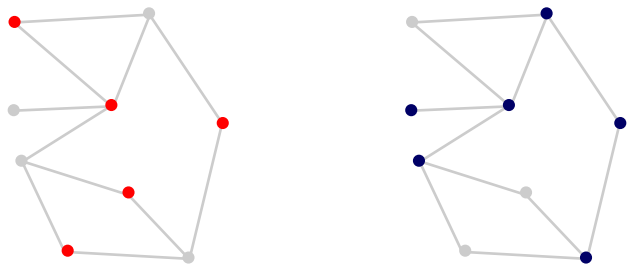
$$\text{SAT} \in \text{NP}$$

Beweisidee: *Eine Belegung B der Variablenlänge kann man als Binärwort $w \in \{0,1\}^*$ auffassen. Für n Variablen gibt es also 2^n mögliche Belegungen, die alle überprüft werden müssen (im Schnitt zumindest die Hälft, im schlimmsten Fall alle, was nichts an der exponentielle Komplexitätsklasse ändert). Die Überprüfung einer Belegung erfolgt nach obigen Aussagen in Polynomialzeit, während das Erzeugen und Durchtesten aller möglicher Belegungen auf einer nichtdeterministischen TM Exponentialzeit beansprucht ($\mathcal{O}(\text{poly}(n) \cdot 2^n) \subset \mathcal{O}(2^n)$). Auf einer nicht-deterministischen TM können alle möglichen Belegungen quasi-parallel geprüft werden. Es ist also nur Polynomialzeit erforderlich und also gilt* $\text{SAT} \in \text{NP}$. *QED.*

Knotenüberdeckungsproblem: Ein endlicher (ungerichteter) Graph $G = (V, E)$ besteht aus einer endlichen Knotenmenge V und einer Kantenmenge E, die einen Teil der Knoten verbindet. D.h. $E = V \times V$ ist eine zweistellige Relation zwischen Knoten.
Eine Knotenüberdeckung U des Graphen G ist eine Teilmenge $U \subset V$, sodass von jeder Kante von G mindestens ein Endpunkt zu U gehört, d.h.:

$$\forall i, j \in V: \quad \big((i,j) \in E \quad \Rightarrow \quad i \in U \text{ oder } j \in U\big).$$

Die folgende Abbildung zeigt zwei verschiedene Überdeckungen (rot, blau) eines Beispiel-Graphen (grau):



k = 5 *k = 6*

Das Knotenüberdeckungsproblem VC fragt nach der Existenz von Überdeckungen der Länge k eines Graphen G:

$$\text{VC} := \big\{(G,k) \big| \text{Graph } G \text{ besitzt Knotenüberdeckung } U \text{ mit } \|U\| = k\big\}.$$

Auch hier gilt:

$$\text{VC} \in \text{NP}$$

p-m-Reduzierbarkeit: Eine Menge $A \subseteq \Sigma^*$ heißt *p-m-reduzierbar* ($A \leq_m^p B$), wenn es eine Funktion $f : \Sigma^* \to \Sigma^*$ in der Klasse FP, der in Polynomialzeit von einer deterministischen TM berechenbaren Funktionen gibt, sodass gilt:

$$\forall x \in \Sigma^* : \quad \big(x \in A \quad \Leftrightarrow \quad f(x) \in B\big).$$

Dies bedeutet, dass $c_A(x) = c_B(f(x))$. Die Definition von $\leq_m^p$ unterscheidet sich von $\leq_m$ nur dadurch, dass hier auf die Berechenbarkeit in Polynomialzeit, also die tatsächliche Berechenbarkeit, bezug genommen wird.
Es gilt:

- Gilt $A \leq_m^p B$ und $B \in$ ¶, so gilt auch $A \in \mathrm{P}$.
- Gilt $A \leq_m^p B$ und $A \notin$ ¶, so gilt auch $B \notin \mathrm{P}$.

p-m-Härte: Eine Menge A ist *p-m-hart* für eine Komplexitätsklasse $\mathcal{C}$, falls $B \leq_m^p A$ für alle $B \in \mathcal{C}$ gilt. Gilt zusätzlich noch, dass A selbst in $\mathcal{C}$ liegt, so ist A *p-m-vollständig* für $\mathcal{C}$.
Für jede Komplexitätsklasse $\mathcal{C}$, die nicht in P enthalten ist, folgt sofort, dass die $\mathcal{C}$-harten und $\mathcal{C}$-vollständigen Mengen nicht in P liegen.
Es gilt:

$$A \,\mathrm{NP}\text{-hart und } A \leq_m^p B \quad \Rightarrow \quad B \,\mathrm{NP}\text{-hart.}$$

Anschaulich bedeutet das, dass ein $\mathcal{C}$-harte/vollständige Menge A ein schwierigeres Problem darstellt, als alle Probleme $B \in \mathcal{C}$. Damit lässt sich jedes Problem in $\mathcal{C}$ auf das Problem A zurückführen, bzw. eine Lösung des Problemes A impliziert sofort auch Lösungen für alle Probleme in $\mathcal{C}$. Kann man also etwa ein NP-vollständiges Problem Z in det. Polynomialzeit lösen, so sind alle andere NP-Probleme ebenfalls in Polynomialzeit lösbar, da sie in Polynomialzeit auf Z zurückführbar sind.

NP-Vollständigkeit: Falls $\mathrm{P} \neq \mathrm{NP}$ (P-NP-Problem) und A NP-vollständig, so gilt: $A \notin$ ¶.

Satz von Cook: SAT ist NP-vollständig

6 Automatentheorie

6.1 Grundtypen von Automaten

6.1.1 einfache Automaten

(Nicht)deterministischer, linear beschränkter Automat (NLBA): Ein NLBA lässt sich als Variante von Turing-Maschinen beschreiben: Im Prinzip handelt es sich um eine 1-Band-TM, die nur Felder besucht, auf denen auch eine Eingabe stand. Man realisiert Dies durch Randmarken, die vor und nach der Eingabe auf das Band geschrieben werden und weder überschrieben, noch überschritten werden dürfen.
Einen deterministischen, linear beschränkten Automaten bezeichnet man entsprechend mit DLBA.
Offensichtlich gehört ein NLBA zur linearen Platzkomplexitätsklasse $\mathrm{NSPACE}(\mathcal{O}(n))$ und ein DLBA zur Klasse $\mathrm{DSPACE}(\mathcal{O}(n))$.

Endlicher Automat (EA): Ein EA ist ein 5-Tupel $\mathrm{EA} = (\Sigma, S, \delta, s_0, S_T)$ mit:

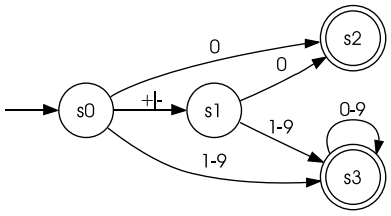
- Eingabealphabet Σ
- Zustandsmenge S
- Startzustand $s_0 \in S$
- Menge der Endzustände $S_T \subseteq S$
- und einer Übergangsfunktion
$$\delta : S \times \Sigma \to 2^S.$$
Dabei bezeichnet 2^S die Potenzmenge über S.

Ein solcher EA startet im Zustand s_0. In einem beliebigen Zustands $s \in S$ liest liest er ein Zeichen $z \in \Sigma$ von der Eingabe. Danach geht er in den den Zustand $s' = \delta(s, z)$ über. Hier kann es durchaus mehrere Möglichkeiten geben.

Es gilt:

1. **deterministischer Automat** EA**:** Ein EA heißt **deterministisch**, wenn die Übergangsfunktion die spezielle Form
$$\delta : S \times \Sigma S$$
hat. Sie bildet also immer auf exakt einen Folgezustand ab.
2. **λ-Automat (λ EA):** Ein Automat EA heißt λ-Automat (λ EA), wenn die Übergangsfunktion die Form
$$\delta : S \times (\Sigma \cup \{\lambda\}) \to 2^S$$
hat, also auch **λ-Übergänge** erlaubt sind. Der Automat wechselt dann den Zustand ohne ein Zeichen von der Eingabe zu lesen.
3. Liefert δ keinen Folgezustand, so vereinbart man, dass sich der Automat in einem **irregulären Endzustand** befindet.
4. **Akzeptierte Sprache:** Eine Sprache L heißt akzeptierte Sprache des Automaten EA, wenn dieser in einem regulären Endzustand aus S_T anhält, wenn er ein beliebiges Wort $W \in L$ vollständig eingelesen hat.

Darstellung: Automaten werden oft als Graphen dargestellt. An den Übergängen stehen dann die eingelesenen Zeichen (evtl. als regulärer Ausdruck). Die Kreise bezeichnen die Zustände. Ein Doppelkreis bezeichnet einen Endzustand. Der folgende Automat akzeptiert z.B. vorzeichenbehaftete Integer-Konstanten:

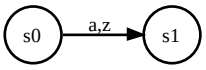


Man kann die Funktion δ auch als Tabelle darstellen, die dann **Zustandstabelle** genannt wird.

Automat mit Ausgabe: Ein EA mit Ausgabe ist ein 6-Tupel $\mathrm{EA} = (\Sigma_e, \Sigma_a, S, \delta, s_0, S_T)$ mit:

- Eingabealphabet Σ_e
- Ausgabealphabet Σ_a
- Zustandsmenge S
- Startzustand $s_0 \in S$
- Menge der Endzustände $S_T \subseteq S$
- Übergangsfunktion
$$\delta : S \times \Sigma_e \to 2^S.$$
Dabei bezeichnet 2^S die Potenzmenge über S.
- und einer Ausgabefunktion
$$\gamma : S \times \Sigma_e \to \Sigma_a.$$
Dabei bezeichnet 2^S die Potenzmenge über S.

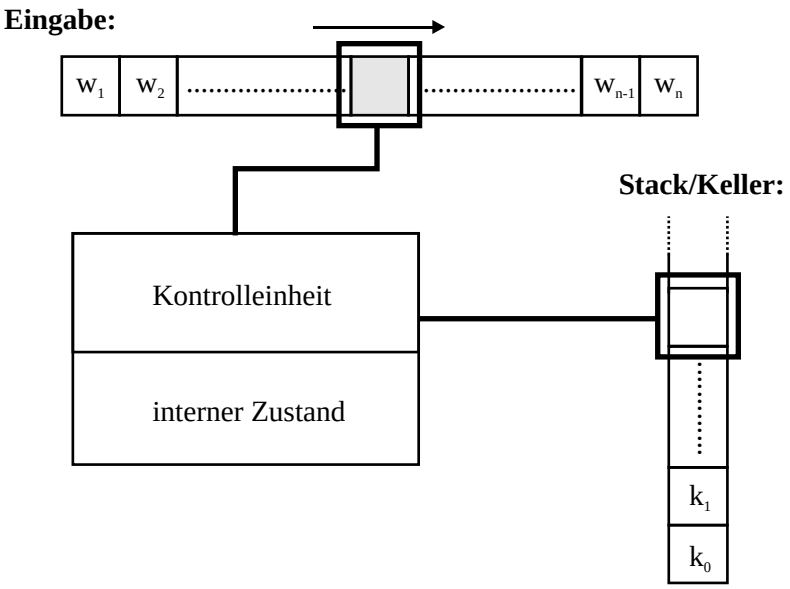
Der Automat funktioniert wie ein gewöhnlicher EA. Zusätzlich gibt er in jedem Schritt ein Zeichen aus, das von γ bestimmt wird. Man notiert das so:



Dabei ist a das eingelesen Zeichen und z die Ausgabe.

6.1.2 Kellerautomaten/Stackautomaten

Man kann das Automatenkonzept um einen internen Speicher erweitern (zusätzlich zu Zustand des Automaten). Man führt dabei einen speziellen Speicher, den Stack oder Kellerspeicher ein. Man kann ein Symbol oben auf den Stack legen und das oberste Symbol lesen. Bildlich sieht ein solcher Kellerautomat so aus:



Man definiert dann noch formal:

nichtdeterministischer Kellerautomat: Ein nichtdeterministischer Kellerautomat KA ist ein 7-Tupel $KA = (\Sigma_e, S, \Sigma_k, \delta, s_0, k_0, F)$ mit:

endlichem Eingabealphabet Σ_e und endlichem Kelleralphabet Σ_k

nichtleerer Zustandsmenge S, einem Startzustand $s_0 \in S$ und einer Menge von Endzuständen $F \subset S$ und

einem Kellerstartsymbol $k_0 \in \Sigma_k$

einer Übergangsfunktion $\delta : S \times \left(\Sigma_e \cup \{\lambda\}\right) \times \Sigma_k \rightarrow 2^{S \times \Sigma_k^*}$

Der Automat wechselt seinen Zustand in Abhängigkeit von seinem aktuellen Zustand, dem Symbol oben auf dem Kellerspeicher und dem aktuellen Eingabesymbol. Bei einem Übergang geht der Automat in einen neuen Zustand über und schreibt evtl. einige Zeichen auf den Stack. Im Englischen heißt ein solcher Automat auch ***push down automaton (PDA)***

nichtdeterministischer Zustandskellerautomat: Ein nichtdeterministischer Zustandskellerautomat ZKA ist ein 5-Tupel $\text{ZKA} = (\Sigma, S, \Sigma_k, \delta, s_0, S_T)$ mit:

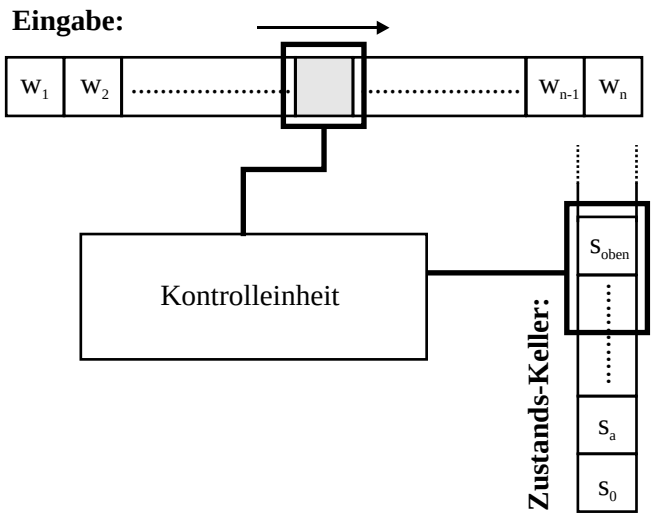
endlichem Eingabealphabet Σ

endlicher Zustandsmenge S, einem Startzustand $s_0 \in S$ und einer Menge von Endzuständen $S_T \subseteq S$ und

einem Kellerstartsymbol $k_0 \in \Sigma_k$

einer Übergangsfunktion $\delta : S^+ \times \left(\Sigma \cup \{\lambda\}\right) \rightarrow 2^{S^*}$

Der Automat liest eine Folge von Zuständen vom Stack (Keller) und schreib wieder eine Folge von Zuständen auf den Stack, während er ein Eingabezeichen (oder λ) liest. Der Automat startet mit s_0 als oberstem Kellersmybol. Der Automat hängt also nicht von einem expliziten internen Zustand, sondern von den Zuständen, die sich gerade oben auf dem Stack befinden. Die folgende Abbildung veranschaulicht den Automaten:



6.2 Automaten und Sprachen

von endlichen Automaten akzeptierte Sprachen: Die von einem endlichen Automaten akzeptierten Sprachen kann man mit rechtslinearen Grammatiken erzeugen. Deterministische Automaten und nicht-deterministische λ-Autonaten akzeptieren die gleichen Sprachen.

Beweisidee:

1. *Ordne jedem Nicht-Terminal einen Zustand des Automaten zu. Jede λ-Regel führt zu einem Endzustand. Zu jeder anderen Regel erzeugt man einen entsprechenden Übergang.*
2. *Durch folgende Konstruktion kann man aus einem nicht-deterministischen $\lambda\,\mathrm{EA}$ einen deterministischen Automaten EA konstruieren:*
   a) ***Elimination der λ-Übergänge:***
      - *Die Zustände und der Startzustand bleiben gleich $S' = S$, $s'_0 = s_0$.*
      - *Die Terminalzustände $S'_T = S_T \cup S_{TN}$ werden um diejenigen Zustände S_{TN} erweitert, die mit einem λ-Übergang von einem $s \in S_T$ erreichbar sind.*
      - *Die Übergangsfunktion wird so abgewandelt: Zu jeder möglichen Übergangsfolge $s_i \xrightarrow{z} s_j \xrightarrow{\lambda} s_k$ und $s_i \xrightarrow{\lambda} s_j \xrightarrow{z} s_k$ wird der Übergang $s_i \xrightarrow{z} s_k$ hinzugefügt. Die λ-Übergänge können dann entfernt werden.*

   b) ***Konstruktion eines EA:***
      - *Die Zustände des EA sind alle möglichen Teilmengen von S', als $S'' = 2^{S'}$.*
      - *Die neue Übergangsfunktion ist:*

$$\delta''(S, z) = \bigcup_{s \in S} \delta'(s, z)$$

        Die Menge $\delta''(S, z)$ enthält also alle Zustände, die von einem Zustand aus S erreichbar waren.
      - *Die Menge S''_T der Endzustände besteht aus den Zuständen, die wenigstens einen Endzustand aus S'_T enthalten.*

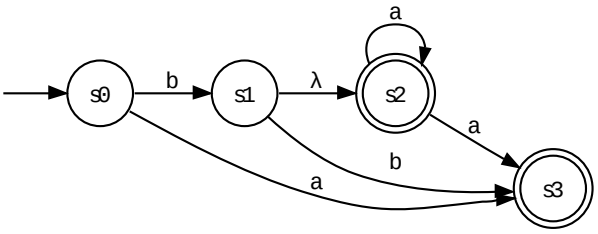
   Diese schematische Konstruktion erzeugt $2^{|S'|}$ Zustände (Anzahl der Elemente in der Potenzmenge $2^{S'}$). Diese werden evtl. aber gar nicht alle benötigt. Deswegen kann man den Automaten auch sukzessive vom Startzustand s_0 aus konstruieren. Man verfährt dann so:

   i. *Die Zustände des EA sind alle möglichen Teilmengen von S', als $S'' = 2^{S'}$.*
   ii. *berechne $\delta''(s_0, z) = S_{s_0,z}$ für alle $z \in \Sigma$ und füge die dabei auftretenden Zustände $S_{s_0,z}$ zu S'' hinzu.*

iii. *berechne nun $\delta''(s_i, z) = S_{s_i,z}$ für alle $z \in \Sigma$ und alle Zustände aus S''. Entstehen dabei neue Zustände werden sie ebenfalls zu S'' hinzugefügt. Führe dies fort, bis sich an S'' nichts mehr ändert.*

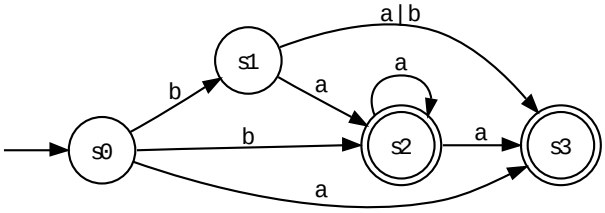
iv. *Die Menge S''_T der Endzustände besteht aus den Zuständen, die wenigstens einen Endzustand aus S'_T enthalten.*

Beispiel: Der folgende nichtdeterministische λ-Automat soll umgewandelt werden:



1. Elimination der λ-Übergänge:
   - $s1$ wird zusätzlicher Endzustand, weil der Endzustand $s2$ von $s1$ aus über einen λ-Übergang erreichbar ist.
   - Der λ-Übergang wird entfernt. Dafür kommen folgende Übergänge hinzu: $s0 \xrightarrow{b} s2$ ($s0 \xrightarrow{b} s1 \xrightarrow{\lambda} s2$), $s1 \xrightarrow{a} s2$ ($s1 \xrightarrow{\lambda} s2 \xrightarrow{a} s2$), $s1 \xrightarrow{a} s3$ ($s1 \xrightarrow{\lambda} s2 \xrightarrow{a} s3$)

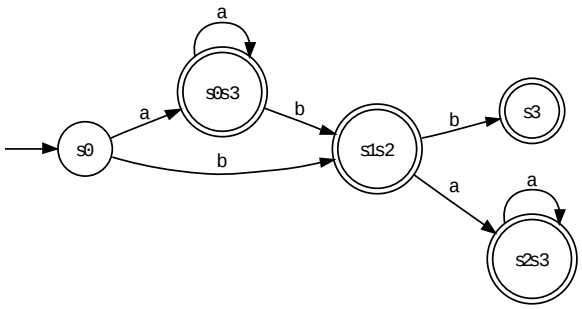
Man erhält dann also:



Nun konstruiert man sukzessive vom Startzustand aus den neuen Automaten:

1. $\delta''(\{s0\}, a) = \{s0, s3\}$ $\quad \delta''(\{s0\}, b) = \{s1, s2\}$
2. $\delta''(\{s0, s3\}, a) = \{s0, s3\}$ $\quad \delta''(\{s0, s3\}, b) = \{s1, s2\}$
3. $\delta''(\{s1, s2\}, a) = \{s2, s3\}$ $\quad \delta''(\{s1, s2\}, b) = \{s3\}$
4. $\delta''(\{s2, s3\}, a) = \{s2, s3\}$ $\quad \delta''(\{s2, s3\}, b) = \{\}$

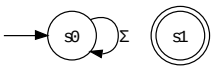
Man benötigt also nicht alle Zustände aus der Potenzmenge $2^{S'}$, sondern nur eine Auswahl. Man erhält so den folgenden Automaten:



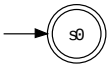
Reguläre Ausdrücke und Automaten: Reguläre Ausdrücke und endliche Automaten beschreiben die gleiche Sprachen.

Konstruktion eines λEA zu einem gegebenen regulären Ausdruck a: Die Konstruktion erfolgt rekursiv über die möglichen Elemente des regulären Ausdruckes a:

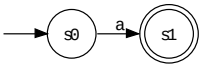
leere Sprache



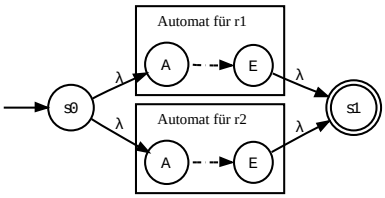
leeres Wort λ



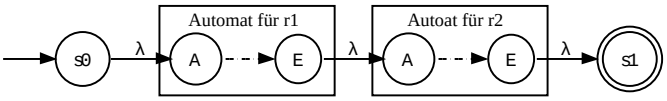
Zeichen $a \in \Sigma$



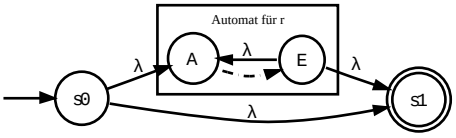
Alternative $r1 \mid r2$



Konkatenation $r1\ r2$



Iteration r^*



7 SHANNON'SCHE Informationstheorie

- Eine **Nachricht** ist eine Folge von Zeichen $a_0 a_1 ... a_{n-1}$ aus dem zugrundeliegenden Alphabet Σ. Die Wahrscheinlichkeitsverteilung über den Zeichen $a_i \in \Sigma$ des Alphabets wird mit $p(a_i)$ bezeichnet.
- Für den Informationsgehalt $I(a)$ eines Zeichens werden folgende EIgenschaften gefordert:
  - $I(a)$ ist monoton
  - $I(a)$ ist umso größer, je unwahrscheinlicher das Zeichen a ist
  - $I(a)$ eines Zeichens, das sicher auftritt ($p(a) = 1$) ist $I(a) = 1$

  Die Logarithmusfunktion erfüllt diese Anforderungen und man definiert:

  $$I(a) := K \cdot \log_2 \left(\frac{1}{p(a)} \right) = -K \cdot \log_2(p(a))$$

  Mit $K = 1$ und dem binären Logarithmus ergibt sich die Einheit der Größe zu: $[I(a)] = 1 \text{ Bit}$
- Der **Informationsgehalt einer Nachricht** $a = (a_0..a_n)$ ergibt sich damit zu:

  $$I(a) = \sum_{i=0}^{n} I(a_i) = I(a) = -K \cdot \sum_{i=0}^{n} \log_2(p(a_i))$$

- Die **Entropie** $H(a)$ einer Nachricht $a = (a_0..a_n)$ bezeichnet den mittleren Infromationsgehalt pro Zeichen:

  $$H(a) = \sum_{i=0}^{n} p(a_i) \cdot I(a_i) = -K \cdot \sum_{i=0}^{n} p(a_i) \cdot \log_2(p(a_i))$$

  Ihre Einheit ergibt sich mit $K = 1$ und dem binären Logarithmus zu: $[H(a)] = 1 \, \frac{\text{Bit}}{\text{Zeichen}}$.
  Anschaulich bezeichnet die Entropie die mittlere Anzahl von Bits, die benötigt wird um eine Zeichen einer vorgegebenen Nachricht zu kodieren.

Teil II

Algorithmen und Datenstrukturen

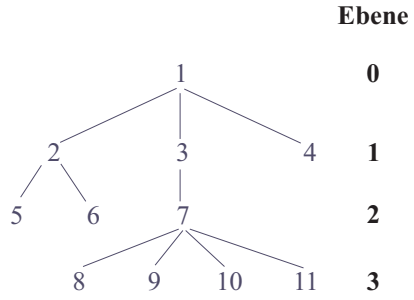
8 Elementare Datenstrukturen

8.1 Bäume

- **induktive Definition:** Ein Baum besteht aus:
  1. einem Knoten und
  2. $n \geq 0$ weiteren Bäumen, auf die der Knoten verweißt. Diese Bäume heißen Unterbäume oder Teilbäume.

  Knoten ohne Nachfolge heißen **Blätter**. Jeder andere Knoten heißt *interner Knoten*. Der oberste Knoten in einem Baum heißt *Wurzel*.
- **Terminologie** (nach *R. Sedgewick: „Algorithmen“*):
  - Ein **Pfad** ist eine Liste von unterschiedlichen Knoten, in der aufeinanderfolgende Knoten durch Kanten verbunden sind.
  - Die **Pfadlänge** eines Baumes ist die Summe der Ebenen aller seiner Knoten. Betrachtet man nur innere Knoten, so nennt man die dadurch berechnete Pfadlänge **innere Pfadlänge**. Addiert man nur die Pfadlängen äußerer Knoten, so erhält man die **äußere Pfadlänge.**
  - Die **Ebene** eines Knotens ist die Anzahl der Knoten auf dem Pfad von ihm zur Wurzel (ohne sich selbst). Die Wurzel liegt also auf Ebene 0, ihre direkten Nachfahren auf Ebene 1 usw.
  - Die **Höhe** eines Baumes ist definiert, als das Maximum der Ebenen aller seiner Knoten, also als die maximale Ebene, die unter seinen Knoten auftritt.
  - Besitzt jeder interne Knoten eines Baumes genau n Nachfolger, so nennt man den Baum n-är.
  - Ein Binärbaum heißt **voll**, wenn jeder Knoten entweder Blatt ist, oder zwei Kinder hat, d.h. es gibt keine Knoten die nur ein Kind haben.
  - Ein Binärbaum heißt **vollständig**, wenn alle Blätter die gleiche Tiefe haben.

Beispiel für einen beliebigen Baum:



Anzahl der Knoten = 11
Tiefe = 3 innere Pfadlänge = 5
Pfadlänge = 21 äußere Pfadlänge = 17

- **Eigenschaften von Bäumen**:
  - Für je zwei beliebige Knoten in einem Baum existiert genau ein Pfad, der sie verbindet.
  - Ein Baum mit N Knoten hat genau $N - 1$ Kanten.

  - Ein binärer Baum mit N inneren Knoten hat $N + 1$ äußere Knoten (Blätter).
    Beweis durch Induktion: Ein binärer Baum ohne innere Knoten hat einen äußeren Knoten. Für $N > 0$ hat jeder Binärbaum k innere Knoten in seinem linken Ast und nach Induktionsvorraussetzung $N - 1 - k$ innere Knoten in seinem rechten Teilbaum. Zusammen mit der Wurzel (innerer Knoten) ergeben sich also $k + (N - 1 - k) + 1 = N$ innere Knoten und insgesamt $N + 1$ außere Knoten `QED`.
  - Die äußere Pfadlänge eines beliebigen binären Baumes mit N inneren Knoten ist um $2N$ größer, als die innere Pfadlänge.
  - Die Höhe eines vollen Binärbaumes mit N inneren Knoten beträgt etwa $\log_2 N$.
    zum Beweis: Auf der 0. Ebene befindet sich ein Knoten. Auf der 1. Ebene sind 2 Knoten. Auf jeder darauffolgenden Ebene verdoppelt sich die Zahl der Knoten, sodass auf der untersten Ebene h (= Höhe des Baumes) 2^h Blätter liegen. Da ein voller Binärbaum mit N inneren Knoten $N + 1$ Blätter hat, gilt also: $2^{h-1} < N + 1 \leq 2^h$.

- **Durchläufe:** Ein Baum kann auf folgende Arten durchlaufen werden:
  - **Inorder:** linker Teilbaum – Wurzel – rechter Teilbaum
  - **Preorder:** Wurzel – linker Teilbaum – rechter Teilbaum
  - **Postorder:** linker Teilbaum – rechter Teilbaum – Wurzel

  So lautet etwa der mathematische Ausdruck $1 * 5 + 6$, der sich als Binärbaum schreiben lässt:
  Inorder: $(1 * 5) + 6$; *Preorder*: $+ (* (1, 5), 6)$; *Postorder*: $((1, 5) *, 6) +$

8.2 Prioritätswarteschlangen (PQ)

Eine Prioritätswarteschlange (PQ von *priority queue*) ist ein Abstrakter Datentyp (ADT) und hat folgende Definition:

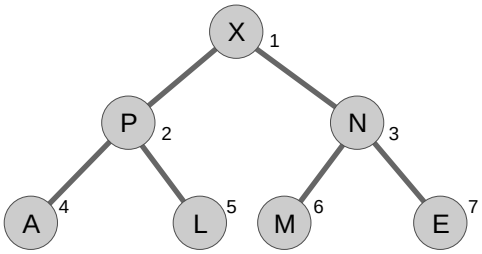
- Operiert auf einer Menge K von Schlüsseln, auf denen eine Ordnungsrelation definiert ist
- grundlegende Operationen:
  - Einfügen
  - Größtes Element entfernen
  - Erzeugen
  - Test, ob leer
  - Zertören
  - Kopieren

8.2.1 Heaps

Ein Heap ist eine spezielle PQ. Es handelt sich um die Repräsentation eines vollständigen Binärbaumes mit Heap-Eigenschaft in einem Feld. In einem Heap (als Binärbaum) gilt die folgende **Heap-Eigenschaft**:

Die Schlüssel werden in Knoten und Blättern gespeichert. Der Schlüssel in jedem Knoten muss größer sein, als die Schlüssel in seinen zwei Kindern.

Damit ist sichergestellt, dass die Wurzel das größte Element im Baum enthält. Man kann nun eine Repräsentation eines solchen Baumes in einem Array vornehmen, indem man die Blätter in Level-order durchnummeriert:



Man kann die Nummer des Knotens auch als Array-Index interpretieren. Aus obigem Baum würde sich dann folgendes Array ergeben:

k	1	2	3	4	5	6	7
a[k]	X	P	N	A	L	M	E

Damit erhält man folgende Regeln, um im Array zu navigieren (Der aktuelle Knoten ist i, das Ergebnis ist j:

Operation	Ergebnis $j = ...$
Eltern-Knoten	`j = i div 2` (Integer-Division/Abrunden)
Kind-Knoten (links)	`j=2*i`
Kind-Knoten (rechts)	`j=2*i+1`

Als Besonderheit erhält kann man daraus noch folgende Eigenschaften ableiten:

- In der zweiten Hälfte des Arrays befinden sich nur Blätter der untersten Ebene des Baumes

Möchte man nun ein neues Element in den Heap einfügen, so muss man sicherstellen, dass die Heap-Eigenschaft erfüllt bleibt. Man vergößert zuerst das Feld und fügt dann ganz hinten das neue Element ein. Ist dieses neue Element nun größer als sein Vorgänger, so werden die beiden vertauscht. Dies geht rekursiv solange weiter, bis das neue Element an einem Platz ist, an dem die Heap-Eigenschaft erfüllt ist. Da die Tiefe eines vollständigen Binärbaumes $\log N$ ist, werden auch maximal $\mathcal{O}(\log N)$ Vertauschungen benötigt, um die Heap-Eigenschaft wieder herzustellen. Der folgende Pascal-Code implementiert das Einfügen:

```
1  { Heap der Länge N in Array a[1..N] }
2  procedure insert(v: datatype);
3  begin
4    N:=N+1;
5    a[N]:=v;
6    upheap(N);
7  end;
8
9  procedure upheap(k:integer);
10 var v:datatype;
11 begin
12   v:=a[k];
13   a[0]:=dummyvalue;
14   while a[k div 2]<=v do begin
15     a[k]:=a[k div 2];
16     k:=k div 2;
17   end;
18   a[k]:=v;
19 end;
```

Damit ist es möglich effizient einen Heap aufzubauen. Das zweite zu Lösende Problem ist das Entfernen der Wurzel aus dem Heap. Ist der Heap in einem Array implementiert, so kann einfach das erste Element entfernt werden. Nun stellt sich die frage, welches Element als neue Wurzel eingefügt wird. Da das Array um eins kürzer wird, scheint es sinnvoll einfach das letzte Element, also das Element auf der untersten Ebene des Baumes, ganz rechts in die Wurzel zu verschieben. Dadurch wird aber natürlich die Heap-Eigenschaft verletzt, sodass man eine Funktion benötigt, die die Heap-EOigenschaft ab einem bestimten Knoten wieder herstellt. Die im folgenden beschriebene Funktion `downhead()` leistet dieses.

```
1  { Heap der Länge N in Array a[1..N] }
2
3  procedure downheap(k:integer);
4  var v:datatype;
5      i,j:integer;
6  begin
7    v:=a[k];
8    while k<=N div 2 do begin { solange nicht auf unterster Ebene }
9      { j zeigt auf linkes Kind
10       falls rechtes Kind größer, dann zeigt j auf rechtes Kind }
11     j:=2*k;
12     if j<N then
13       if a[j]<a[j+1] then j:=j+1;
14
15     { beenden, falls Heap-Eigenschaft erfüllt (betrachteter Knoten
16       größer, oder gleich seinem größten Kind }
17     if v>=a[j] then break;
18
19     { Knoten nach unten schieben }
20     a[k]:=a[j];
21     k:=j;
22   end;
23   a[k]:=v; { Knoten an der richtigen Stelle einfügen }
24 end;
```

Die Funktion bewegt sich vom Knoten k aus nach unten im Baum und vertauscht den aktuellen Knoten solange mit dem größeren seiner zwei Kinder, bis die Heap-Eigenschaft wieder erfüllt ist. Damit nimmt dann `removemax()` eine sehr einfache Form an:

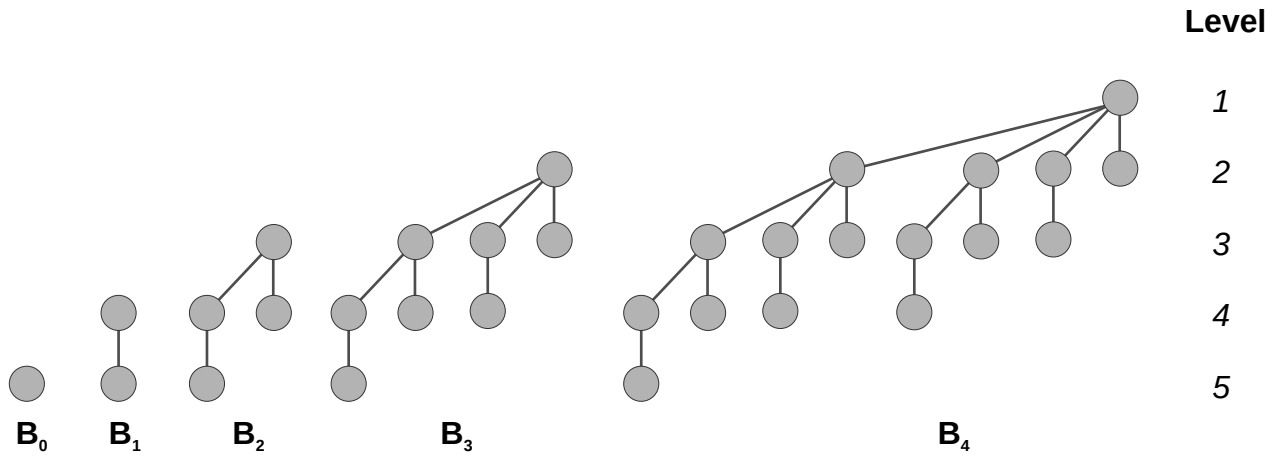
```
1  { Heap der Länge N in Array a[1..N] }
2
3  function removemax:datatype;
4  begin
5    result:=a[1]; { oberstes Element zurückgeben }
6
7    { letztes Element an die Wurzel verschieben }
8    a[1]:=a[N];
9    N:=N-1;
10
11   { Heap reparieren }
12   downheap(1);
13 end;
```

8.2.2 Binomiale Heaps

Ein binomialer Heap ist ein spezieller Heap, der auch das Vereinigen zweier Heaps in logarithmischer Zeit erlaubt, was it Standard-heaps nicht möglich ist. Er ist folgendermaßen rekursiv definiert:

1. Ein Binomialer Baum B_0 besteht aus einem Knoten.
2. Ein Binomialer Baum B_k besteht aus zwei binomialen Bäumen B_{k-1}, die so zusammenhängen, dass die Wurzel des einen Baumes das linkeste Kind des anderen Baumes ist.

Die folgende Abbildung verdeutlicht das:



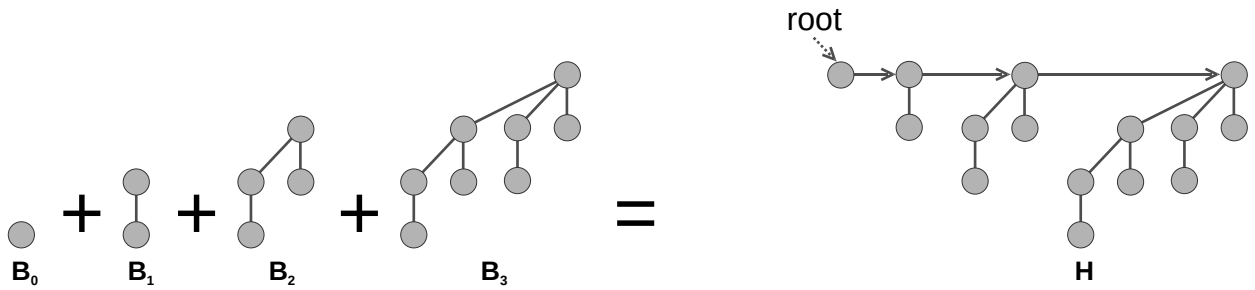
Für einen binomialen Baum B_k gelten folgende Eigenschaften:

1. Es gibt genau 2^k Knoten.
2. Die Höhe des Baum es ist k
3. Es gibt genau $\binom{k}{i}$ Knoten der Tiefe i.
4. Die Wurzel hat genau k Kinder.
5. Die maximale Anzahl von Kindern eines Knotens in B_k ist $\log k$.

Ein binomialer Heap $H_n = \{B_0, B_1, ..., B_n\}$ ist eine Liste von Binomialen Bäumen B_i, mit folgenden Eigenschaften:

1. Die Schlüssel werden in Knoten und Blättern der Bäume in H_n gespeichert.
2. Der Schlüssel in jedem Knoten muss kleiner oder gleich dem Schlüssel seines Eltern-Knotens sein.
3. Zu jeder ganzen Zahl $0 \leq k \leq n$ gibt es höchstens einen binomialen Baum B_k, mit obiger Eigenschaft in H_n.

Die folgende Abbildung zeigt das Konzept:



8.3 Graphen

Definition 8.1 (Graph) *Ein Graph ist eine Menge von Knoten n_i, die durch Kanten e_i verbunden sind. Dabei können gerichtete und ungerichtete Kanten auftreten. Eine gerichtete Kante verbindet einen Knoten n_1 mit einem Knoten n_2, nicht aber n_2 mit n_1. Man notiert dies so: $n_1 \rightarrow n_2$. Eine ungerichtete Kante verbindet die Knoten in beide Richtungen. Dies kann man so darstellen: $n_1 - n_2$ oder $n_1 \leftrightarrow n_2$.*

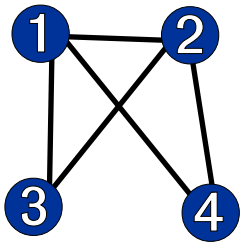
8.3.1 Abspeichern eines Graphen

Um einen Graph zu speichern gibt es mehrere Möglichkeiten:

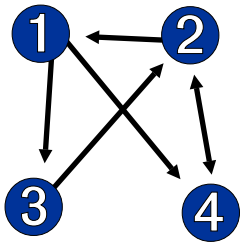
Adjazenzmatrix: Zu einem gerichteten Graphen mit V Knoten ist die Adjazenzmatrix eine $V \times V$-Matrix, die so definiert ist:

$$A_{ij} = \begin{cases} 1 & \text{wenn eine Kante } n_i \rightarrow n_j \text{ existiert} \\ 0 & \text{wenn } n_i, n_j \text{ nicht verbunden sind} \end{cases}$$

Bei ungerichteten Graphen ist $A_{ij} = A_{ji}$. Die Matrix ist also symmetrisch. Die Hauptdiagonale einer solchen Matrix ist an jeder Position mit 0 besetzt. Hier ein Beispiel:



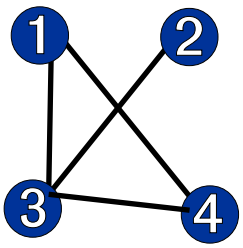
	n_1	n_2	n_3	n_4
n_1	0	1	1	1
n_2	1	0	1	1
n_3	1	1	0	0
n_4	1	1	0	0



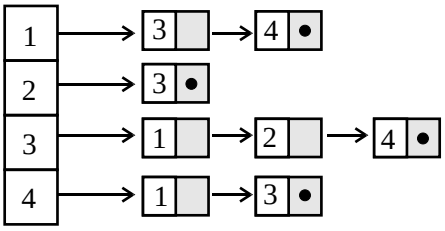
	n_1	n_2	n_3	n_4
n_1	0	0	1	1
n_2	1	0	0	1
n_3	0	1	0	0
n_4	0	1	0	0

Speichern in Listen: Eine weitere Möglichkeit ist es, den Graphen als verkettete Liste zu Speichern. Zu jedem Knoten n_i wird dann eine Liste mit den Nachfolgeknoten gespeichert. Dies lässt sich evtl. auch in einem Array speichern:

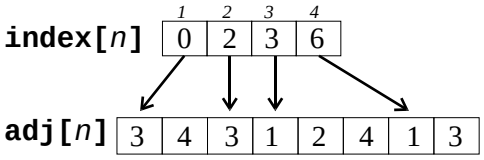
Graph:



verkettete Listen:



Arrays:



In der Array-Variante gibt es ein Array `index[n]`, das für jeden Knoten $n = 1..V$ angibt, ab welcher Position im Array `adj[...]` sich seine Nachfolger befinden.

8.3.2 Durchlaufen eines Graphen

Man kann den Graphen auf zwei Arten durchlaufen:

Tiefendurchlauf: Dieser folgt den Verbindungen zum ersten Nachbarknoten und verfolgt diese bis es nicht mehr weitergeht. An jedem neuen Knoten folgt dieser Algorithmus rekursiv wieder dem ersten Nachbarn, dann dem zweiten usw.:

1. markiere den aktuellen Knoten als in bearbeitung
2. falls noch unbesuchte Nachbarn vorhanden: gehe zum ersten solchen Nachbarn markiere ihn als besucht und fahre dort rekursiv mit Punkt 1 fort.
3. sind keine unbesuchten Nachbarn mehr vorhanden, kehre zum vorherigen Knoten zurück

Breitendurchlauf: Dieser durchläuft zunächst alle Nachbarn des Startknotens, danach alle deren Nachbarn usw. Der Algorithmus verwendet einen FIFO-Speicher (Schlange) Q:

1. alle Knoten sind unbesucht (weiß markiert), füge den ersten Knoten in die Liste Q ein.
2. beenden, falls Q leer.
3. hole einen Knoten u aus der Liste (dabei aus Q löschen)
4. markiere den aktuellen Knoten u als „besucht“ (schwarz) und füge alle seine Kinder hinten in Q ein, die noch weiß sind. Markiere diese als „in Liste“ (grau).
5. gehe zu 2.

9 Elementare Algorithmen

9.1 Union-Find-Algorithmen

- **Aufgabenstellung:** Man habe N Knoten eines Graphen gegeben, die anfangs noch nicht verbunden sind. Einem Programm werden dann Wertepaare $p-q$ übergeben. Das Programm prüft, ob die Knoten p und q bereits verknüpft sind (FIND) und verknüpft sie falls nicht (UNION). Im ersteren Fall soll $p-q$ ausgegeben werden, im zweiteren Fall nichts.
- **QUICK-FIND-Algorithmus:**
  - Unterhalte ein Array `id[]` mit einem Namen für jede Komponente. Sind zwei Komponenten p, q verbunden, so haben sie den selben Namen.
  - `FIND:` überprüfe `id[p]=id[q]` $\Rightarrow$ $\mathcal{O}(1)$
  - `UNION:` Um p und q zu verbinden, müssen alle Elemente, die den selben Namen, wie p haben, den Namen von q erhalten: $\Rightarrow$ $\mathcal{O}(N)$

```
pid:=id[p];
for i:=0 to N-1 do begin
  if id[i]=pid then id[i]:=id[q]
end;
```

- **QUICK-UNION-Algorithmus:**
  - Unterhalte ein Array `id[]` mit einem Namen für jede Komponente. Sind zwei Komponenten p, q verbunden, so haben sie die selbe Wurzel.

```
function ROOT(p:integer):integer
var i:integer;
begin
  i:=p;
  while (i<>id[i]) do
    i:=id[i];
end;
```

  - `FIND:` überprüfe `ROOT(p)=ROOT(q)` $\Rightarrow$ $\mathcal{O}(N)$
  - `UNION:` Wenn die Wurzel von q bereits aus dem FIND-Schritt bekannt ist, so muss man nur noch p mit dieser Wurzel verbinden: `id[p]=rootQ` $\Rightarrow$ $\mathcal{O}(1)$
- **WEIGHTED-QUICK-UNION:** In den bisherigen Algorithmen kann jeweils ein Schritt sehr aufwendig sein (bei großen $N \gg 1$), also modifiziert man QUICK-UNION so, dass große Bäume vermieden werden. Dazu hängt man beim UNION-Schritt jeweils den kleineren Baum an die Wurzel des Größeren an.
  - Dazu muss ein zusätzliches Array `size[]` unterhalten werden, das die Größe des Baumes mit der entsprechenden Wurzel enthält.
  - `FIND:` wie bei QUICK-Union
  - `UNION:`

```
if size[rootP] < size[rootQ] then begin   { P ist im kleineren Baum}
  id[rootP]=rootQ;       { ROOT(P) an ROOT(Q) anhängen}
  root=rootQ;
  size[rootQ]=size[rootQ]+size[rootP];
end else begin           { Q ist im kleineren Baum}
  id[rootQ]=rootP;       { ROOT(Q) an ROOT(P) anhängen}
  root=rootP
  size[rootP]=size[rootP]+size[rootQ];
end;
```

- **WEIGHTED-QUICK-UNION mit Pfadkompression:** Man modifiziert WEIGHTED-QUICK-UNION so, dass der Pafd von p, bzw. q zu ihren jeweiligen Wurzeln komprimiert wird. Man macht also einen zweiten Durchlauf der Pfade (unten stehendes Programm muss also verdoppelt werden) und verbindet alle passierten Knoten mit der neuen Wurzel. Dadurch wird die Laufzeit nicht wesentlich verlängert, dafür bleibt der Baum aber praktisch flach.

```
1      i:=p;
2      while (i<>id[i]) do begin
3        i:=id[i];
4        id[i]=root;
5      end;
```

9.2 Algorithmen für Bäume

9.2.1 Traversieren von Binärbäumen

- Es gibt eine einfache rekursive Implementierung:

```
1    procedure traverse(t:link)
2    begin
3      traverse(t^.left);
4      visit(t);
5      traverse(t^.right);
6    end;
```

- Man kann das Problem auch nicht-rekursiv lösen, indem ein Stack (LIFO) verwendet wird:

```
1    procedure traverse(t:link)
2    begin
3      push(t);
4      repeat
5        t:=pop;
6        visit(t);
7        if t<>nil then begin
8                push(t^.left);
9                push(t^.right);
10             end;
11           until stackempty
12   end;
```

Dieses Programm traversiert den Baum in Preorder. Verwendet man statt einem Stack eine Queue (FIFO), so ergibt sich eine Level-Order-Traversierung, die mit einem rekursiven Programm auf einfache Weise nicht möglich ist.

9.2.2 Zählen der Elemente in einem Binärbaum

```
1    procedure count(t:link)
2    begin
3      if t=nil then result:=0
4      else result:=count(h^.left)+1+count(h^.right);
5    end;
```

9.2.3 Tiefe eines Binärbaumes Baum

```
1    procedure height(t:link)
2    int u,v;
3    begin
4      if h=nil then result:=-1    { Ende des Baumes (nil-Pointer) erreicht}
5      else begin
6        u:=height(h^.left);       { Höhe links}
7        v:=height(h^.right);      { Höhe rechts}
```

```
      result:=max(u,v)+1;        { maximale Höhe+1 zurückgeben}
    end;
  end;
```

9.2.4 Baum-Isomorphie nach Aho/Hopcroft/Ullman

Der folgende Algorithmus bestimmt in linerarer Zeit, ob zwei Bäume T_1, T_2 mit der Höhe h zueinander isomorph sind. Er weißt jedem Knoten eine Signatur zu und zwar derart, dass die Bäume genau dann isomorph sind, wenn den Wurzeln die gleiche Sinatur zugeordnet wurde. Den Knoten werden genauer 2-Tupel von Werten zugeordnet, die einerseits die Daten λ_ν enthalten, die ohnehin im Knoten gespeichert sind und andererseits eine Signatur (string).

Man beginnt auf dem untersten Level und ordnet jedem Blatt ν das Tupel $(\lambda_\nu, 0)$ zu. Danach springt man ein Level höher und ordnet dort jedem Blatt $(\lambda_\nu, 0)$ zu. Jeder innere Knoten erhält ein Tupel aus λ_ν und den sortiert hiterinander gehängten Signaturen aller seiner Kinder (Hat also ein Knoten zwei Kinder mit $(\lambda, 0)$, so ist sein Signatur $(\lambda, 00)$). Man ordnet dann jedem Knoten auf dem entsprechenden Level eine Zahl zu, die sich ergibt, wenn man die unterschiedlichen Signaturen auf einem Level sortiert und dann durchnumeriert (gleiche Signaturen erhalten gleiche Muster). So wäre etwa auf dem zweiten Level dem Tupel $(\lambda, 0)$ die Zahl 0 und dem Tupel $(\lambda, 00)$ die Zahl 1 zugeordnet (dabei wird davon ausgegangen, dass alle Knoten die selbe Information λ enthalten). Diese neue Zahl wird dann die neue Signatur des jeweiligen Knotens. Man fähr dann so fort, bis man bei der Wurzel angekommen ist. Sind die Zahlen, die der Wurzel zugeordnet wurden identisch, so sind die Bäume isomorph.

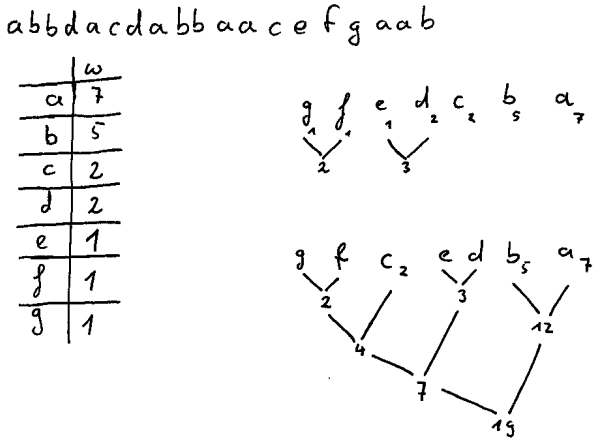
9.3 Kompressionsalgorithmen

9.3.1 Huffman Kodierung

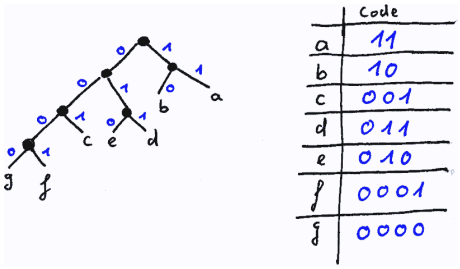
Das Ziel der Huffman Kodierung ist es eine Kodierung für ein Alphabet Σ zu finden, die möglichst wenig Platz benötigt. Dabei werden häufige Buchstaben mit wenigen und weniger häufige Buchstaben mit mehr Bits codiert. Der Algorithmus läuft so ab:

1. bestimme die Häufigkeit $w(a_i)$ für alle Zeichen $a_i \in \Sigma$ des Alphabets in der Nachricht $n \in \Sigma^*$.
2. Ordne die Zeichen nach ihrer Häufigkeit.
3. verbinde die zwei Zeichen mit niedrigster Häufigkeit $w(a_i), w(a_j)$ und füge den so entstandenen neuen Knoten mit der Häufigkeit $w(a_i) + w(a_j)$ wieder in die Liste ein.
4. Fahre kehre zu Punkt 3 zurück und fahre fort, bis nur noch ein Knoten in der Liste ist.

Dabei entsteht ein Baum, wie ihn die folgende Abbildung zeigt.



Man kann nun jeder rechten Kante das Bit '1' und jeder linken Kante das Bit '0' zuordnen. Folgt man nun dem Baum bis zu den einzelnen Blättern, so bildet man den Code aus den passierten Kanten des Baumes.



Nun kann man die Nachricht codieren. Ein Trennzeichen zwischen den Zeichen ist nicht nötig. Zum dekodieren verwendet man die kodierte Nachricht als Anweisung, wie durch den Baum zu laufen ist. Für eine '0' nimmt man den linken und für '1' den rechten Teilbaum. Wenn man an einem Blatt ankommt wird das zugehörige Zeichen ausgegeben. Man beginnt danach wieder an der Wurzel des Baumes.

Satz 9.1 (perfekter Präfixcode) *Der Huffman Code ist ein Präfixcode. Dies bedeutet, dass kein Codewort Präfix eines anderen Codewortes ist. Der Code ist bezüglich der Länge der Codewörter ein idealer/perfekter Code, wenn die Häufigkeiten $w(a_i)$ in der Nachricht bekannt sind.*

9.3.2 Lempel-Ziv-Welch

Der LZW-Algorithmus dient zum komprimieren von Strings. Er erzeugt beim komprimieren sein eigenes Codebuch, das nicht mitübertragen werden muss. Das Codebuch ist eine Map (ADT), die Strings ein Codewort (Integer/Bittfolge) zuordnet. Die Codewörter sind im Allgemeinen kürzer als die zu codierenden Strings, was eine Kompression bewirkt. LZW eignet sich vor Allem für Zeichenketten, in denen oft das selbe Zeichen aufeinander folgt, da sich im Codebuch so sehr schnell lange Ketten aus diesem Zeichen aufbauen, die dann sehr effizient codiert werden. Damit eignet sich LZW gut für Zeichnungen mit wenigen Farben und vielen großen gleichfarbigen Flächen (→ GIF-Format).

Das codieren erfolgt nach folgendem Algorithmus:

```
1  procedure LZWCode(data:string);
2  var K:char;
3      i:integer
4      w:string;
5  begin
6          InitializeStringTable();
7          WriteCode(ClearCode);
8          w:="";                              { the empty string }
9          for i:=1 to length(data) do begin
10           K := data[i];                     { GetNextCharacter() }
11           if isInCodeTable(w+K) then begin
12             w := w + K;
13           end else begin
14             WriteCode(CodeFromString(w));
15             AddTableEntry( w+K );
16             w := K;
17           end;
18         end;
19         WriteCode(CodeFromString(w));
20 end;
```

Der Algorithmus macht folgendes:

1. Suche den längsten Teilstring `w` ab der aktuellen Position, der im Codebuch bereits enthalten.
2. Gib den Code von `w` aus.

3. Füge zu `w` noch ein weiteres Zeichen `K` hinzu und füge den so erhaltenen String `w+K` in das Codebuch ein.
4. Setze für den nächsten Durchlauf `w=K`, das das Zeichen `K` noch nicht codiert wurde. Fahre bei 1. fort.

Das Dekodieren eines LZW-gepackten Strings erfolgt so:

```
1  procedure LZWDecode(data:string);
2  var Code, OldCode, OutString : string;
3  begin
4    InitializeTable();
5    Code=GetNextCode;
6    { solange noch nicht letztes Codewort gelesen wurde}
7    while (Code<>EoiCode) do begin
8      if (Code = ClearCode) then begin { Spezialcode zum Löschen des Codebuches}
9        InitializeTable();
10       Code = GetNextCode();
11       if (Code = EoiCode)
12         break;
13       WriteString(StringFromCode(Code));
14       OldCode = Code;
15     end else begin
16       if (IsInTable(Code)) then begin
17         WriteString(StringFromCode(Code));
18         AddStringToTable(StringFromCode(OldCode)+FirstChar(StringFromCode(Code)));
19         OldCode = Code;
20       end else begin {* K omega K case *}
21         OutString = StringFromCode(OldCode)+FirstChar(StringFromCode(OldCode));
22         WriteString(OutString);
23         AddStringToTable(OutString);
24         OldCode = Code;
25       end;
26     end;
27     Code=GetNextCode;
28   end; {of while}
29 end;
```

Der Algorithmus macht folgendes:

1. Zuerst Codebuch initialisieren und das erste Zeichen einlesen.
2. solange das aktuelle Zeichen nicht das Ende des Code-streams bezeichnet, wird geprüft ob es sich um
   a) ... den speziellen `ClearCode` handelt, der anzeigt, ob das Codebuch reinitialisiert werden muss
   b) ... ob das Zeichen im Codebuch enthalten ist:
      - ... falls ja wird der entsprechende decodierte Text ausgegeben und die <u>letzte</u> Ausgabe + dem ersten Zeichen der aktuellen Ausgabe zum Codebuch hinzugefügt.
      - ... falls nein, handelt es sich um einen $K\omega K$-Fall, der auftreten kann, weil das Codebuch beim Dekodieren um einen Schritt gegenüber dem Codieren nachhängt. Dieser Fall kann aber nur auftreten, wenn die zu codierende Zeichenfolge $K\omega K$ war, wobei $K\omega$ die letzte Ausgabe war. Dieser Fall wird nun umgangen, indem man das Wissen über das Aussehen des zu codierenden Textes nutzt. Das aktuelle Wort muss $K\omega K$ sein. Dieses Wort kann ausgegeben werden und wird in die Codetabelle eingefügt.
3. lese ein neues Codewort ein und fahre bei 2. fort.

Als Optimierung wäre z.B. denkbar, dass ein Sonderzeichen eingeführt wird, das dazu führt, dass das Codebuch gelöscht und neu aufgebaut wird. Dies kann sinnvoll sein, wenn die Größe des Buches beschränkt ist und die Kompressionsrate abfällt. Dies ist im Decoder hier bereits implementiert.

$K\omega K$-Fall: Zunächst ein Beispiel. Es soll die Zeichenkette `APAPAPAPAPAP` codiert und danach wieder decodiert werden:

Eingabe	erkanntes Zeichen	neuer Eintrag
APAPAPAPAPAP	A	AP (=256)
PAPAPAPAPAP	P	PA (=257)
APAPAPAPAP	AP	APA (=258)
APAPAPAP	APA	APAP (=259)
PAPAP	PA	PAP (=260)
PAP	PAP	

Daraus ergibt sich folgende codiertes Wort:

```
A P <256> <258> <257> <260>
```

Nun soll versucht werden das Wort zu dekodieren:

Eingabe	erkannt	*neuer Eintrag*
A	A	
P	P	AP (=256)
<256>	AP	PA (=257)
<258>	???	

An der markierten Stelle ist die Dekodierung nicht weiter möglich, weil der zugehörige Wörterbucheintrag noch nicht existiert. Die einzige Möglichkeit für das Auftreten dieses Falles ist, dass das Wort die Form $K\omega K$ hatte, wobei $K \in \Sigma$ ein Zeichen und $\omega \in \Sigma^*$ eine beliebige Zeichenkette ist. Dies bedeutet, dass die zuletzt eingefügte Zeichenfolge $K\omega$ entspricht. Dies bedeutet, dass die gesuchte Zeichenfolge $K\omega K$, also APA ist. Dieses neue Wort muss auch eingefügt werden. Man kann nun also weitercodieren:

<258>	$K\omega K$: APA	APA (=258)
<257>	PA	APAP (=259)
<260>	$K\omega K$: PAP	PAP (=260)

9.4 Elementare Sortierverfahren

9.4.1 Selection-Sort – Sortieren durch Auswahl

Hier wird zuerst das kleinste Element im Feld gesucht und an die erste Stelle gesetzt. Danach wird das zweitkleinste gesucht und an die zweite Stelle gesetzt usw. Eine Implementierung lautet:

```
1  { N: länge des Feldes a[1..N], das sortiert werden soll }
2  procedure SelectionSort;
3  var i,j:integer;
4            min:integer;
5  begin
6    for i:=1 to N-1 do begin
7      min:=i;
8      for j:=i+1 to N do begin
9        if a[j]<a[min] then
10         min:=j;
11     end;
12     swap(a[min],a[i]);
13   end;
14 end;
```

Es werden maximal N Vertauschungs-Operationen durchgeführt, aber N^2 Vergleiche. Die Gesamtkomplexität ist damit $\mathcal{O}(N^2)$. Die Komplexität im Best Case (bereits sortierte Liste) ist $\mathcal{O}(N)$. SelectionSort eignet sich also recht gut, wenn der Vertauschungsschritt (Kopieren eines Datensatzes) sehr aufwendig ist, z.B. bei großen Datensätzen.

9.4.2 Insertion-Sort – Sortieren durch Einfügen

Betrachte die Elemente eines nach dem Anderen und füge jedes Element an seinen Platz unter den bereits betrachteten ein. Eine Implementierng:

```
1  { N: länge des Feldes a[1..N], das sortiert werden soll }
2  procedure InsertionSort;
3  var i,j:integer;
4            v:Item;
5  begin
6    for i:=2 to N do begin
7      v:=a[i];
8      j=i;
9      while a[j-1]>v do begin
10       a[j]:=a[j-1];
11       j:=j-1;
12     end;
13     a[j]:=v;
14   end;
15 end;
```

Zuerst werden die ersten beiden Elemente betrachtet. Sind sie in der falschen Reihenfolge, so werden sie ausgetauscht.Alle Elemente $< i$ können immer als sortiert angenommen werden. Damit sucht die while-Schleife unter diesen Elementen die Position an der das i-te Element eingefügt werden muss. Sie verschiebt dabei alle Elemente zwischen i und der richtigen Position um eins nach rechts, um danach das neue Element einzufügen. Es werden $\mathcal{O}(N^2)$ Vergleiche und Austauschoperationen benötigt. Für sortierte Felder (Best Case) ist InsertionSort linear $\mathcal{O}(N)$, da die richtige Position jedes Elementes sofort erkannt wird.

ShellSort: InsertionSort ist ineffektiv, weil immer nur Verschiebungen um einen Platz möglich sind. Man kann deswegen optimieren, indem man Verschiebungen über größere Distanzen ermöglicht. Man nennt einen großen Datensatz h-sortiert, wenn man eine sortierte Liste erhält, wenn man jedes h-te Element auswählt. Eine h_1-sortierte Datei kann leichter h_2-sortiert werden ($h_1 > h_2$), als eine vollkommen unsortierte Datei. Man kann eine Datei also sortieren, indem man sie nacheinander h-sortiert mit einer beliebigen Folge von h-Werten, die absteigend bei 1 endet.

9.4.3 Bubble-Sort

Hierbei werden solange aufeinanderfolgende Zeichen vertauscht, bis keine Vertauschungen mehr möglich sind. In diesem Fall ist das Array sortiert. Eine Implementierung lautet:

```
1  { N: länge des Feldes a[1..N], das sortiert werden soll }
2  procedure BubbleSort;
3  var i,j:integer;
4  begin
5    for i:=N downto 1 do begin
6      for j:=2 to i do begin
7        if a[j]<a[j-1] then
8          swap(a[j+1], a[j]);
9      end;
10   end;
11 end;
```

Der erste Durchlauf schiebt das größte Element nach ganz rechts. Damit muss das rechteste Element nicht mehr beachtet werden und es kann das nächst größere nach rechts durchgeschoben werden. Man kann den Algorithmus etwas optimieren, indem man über eine Zustandavariable überprüft, ob ein Austausch stattgefunden hat. Dies führt dann zu linearer Laufzeit auf bereits sortierten Listen, verschlechtert aber die quadratische Laufzeit in allen anderen Fällen nicht. Der Algorithmus hat quadratische Laufzeit, da gilt:

$$C_N = \underbrace{N}_{\text{innere Schleife}} + \underbrace{C_{N-1}}_{\text{äußere Schleife}} = N + (N-1) + (N-2) + ... + 1$$

9.5 Komplexe Sortierverfahren

9.5.1 Quicksort

Quicksort ist ein Sortierverfahren mit einer Zeitkomplexität von $\mathcal{O}(N \cdot \log N)$, das auf dem Teile und hersche Prinzip basiert. Dies ist allerdings nur die Laufzeit im Durchschnittsfall. Im schlimmsten Fall kann Quicksort $\mathcal{O}(N^2)$ Schritte benötigen. Außerdem ist das Verfahren nicht stabil, d.h. liegen zwei Elemente mit gleichem Schlüssel in der Eingabe hintereinander, so müssen sie in der Ausgabe nicht in der gleichen Reihenfolge vorliegen. Quicksort ist zwar schnell und einfach zu implementieren, hat aber den Nachteil, dass er nur effizient abläuft, wenn man auf die Elemenete der zu sortierenden Daten in konstanter Zeit zugreifen kann. Für verkettete Listen ist er damit nicht geeignet. Dafür benötigt er keinen zusätzlichen Speicher und läuft in-place (also auf dem Feld selber) ab.

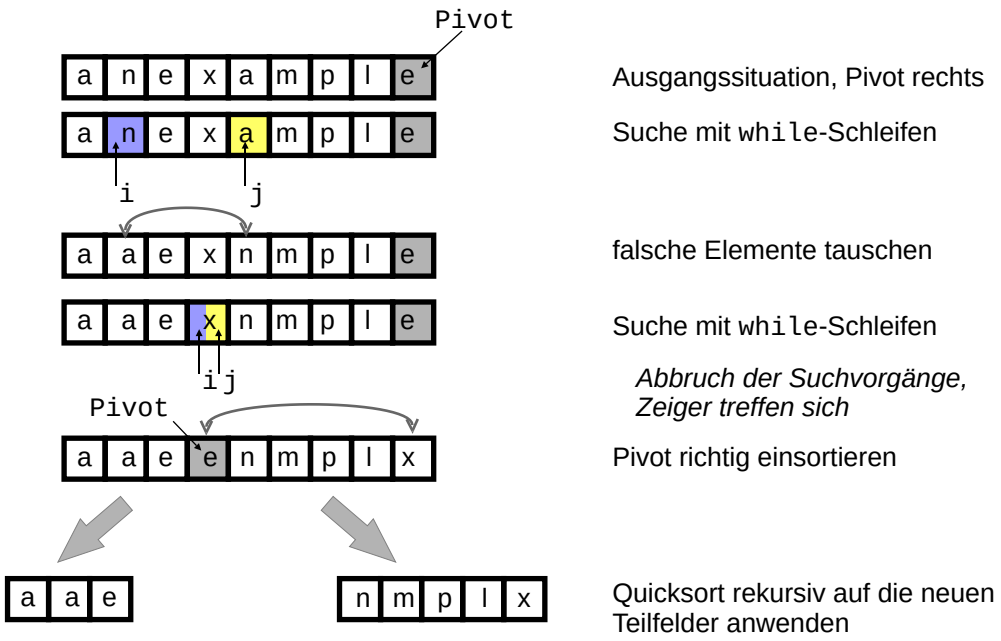
Quicksort teilt rekursiv das Datenfeld auf und zwar so, dass links eines bestimten Trennelementes (Pivot) nur kleinere und rechts nur größere Elemente stehen. Das Pivot ist dann bereits an seiner endgültigen Stelle einsortiert. Danach wird diese Prozedur rekursiv für beide so entstandenen Hälften wiederholt. Der Algorithmus läuft folgendermaßen ab:

```
1  { N: länge des Feldes a[1..N], das sortiert werden soll }
2  procedure quicksort(links:integer, rechts:integer);
3  var pivot:integer;
4  begin
5    if (links < rechts) then begin
6      pivot := partition(links, rechts);
7      quicksort(links, pivot-1);
8      quicksort(pivot+1, rechts);
9    end;
10 end;
```

Für das Pivot-Element wählt man üblicherweise das rechte Element in der Teilliste. Ist die Liste anfangs unsortiert, so ist diese Wahl recht gut. Für sortierte Listen wählt man aber so immer das größte Element aus und der Algorithmus läuft in $\mathcal{O}(N^2)$ ab. Man kann dies umgehen, indem man den Median, oder ein zufälliges Element als Pivot auswählt. Damit erreicht man auch bei vorsortierten Listen eine Laufzeit, die praktisch $\mathcal{O}(N \cdot \log N)$ ist. Der Aufwändigste Teil des Algorithmus ist die Prozedur `partition(l,r)`, die die Aufteilung des Arrays in zwei Hälften vornimmt:

```
1  procedure partition(links:integer, rechts:integer);
2  var i,j,pivot: integer;
3  begin
4     i := links;
5     j := rechts-1;
6     pivot := rechts; { hier kann man evtl. auch ein anderes Pivot auswählen }
7     while (true) begin
8       { suche von links ein Element, das >= daten[pivot] ist }
9       while (a[i] < a[pivot]) do
10        i := i + 1;
11
12      { suche von rechts ein Element, das < daten[pivot] ist }
13      while ((a[pivot]<a[j]) and (j>links)) do
14        j := j - 1;
15
16      { Abbruch, wenn sich i und j "kreuzen" }
17      if (i >= j) then break;
18
19      { vertausche zwei unpassende Elemente }
20      exchange(a[i], a[j])
21    end;
22
23    { setze Pivot an die richtige Stelle und gib seine Position zurück }
24    exchange(a[i], a[pivot])
25    result:=i;
26 end;
```

Die folgende Abbildung verdeutlich die Funktionsweise



Man kann Quicksort verbessern, wenn man den Quicksort-Algorithmus selber nur die grobe Arbeit erledigen lässt und für den Rest ein anderes Verfahren anwendet (**Hybrid-Quicksort**). Geht man von einem sehr großen Datensatz aus, so werden die meißten Rekursions-Schritte nötig, wenn man kleine Unter-Datensätze betrachtet. Um diese vielen Schritte zu vermeiden, kann man den Rekursionsprozess abbrechen, wenn die Teilfelder kleiner als eine Konstante $M \approx 5..10$ sind. Danach sortiert man nochmals mithilfe eines elementaren Sortierv-Verfahrens, das auf nahezu-sortierten Daten sehr effektiv ist (z.B. Insertion-Sort). Um dies zu implementieren muss man in der Quicksort-Prozedur die folgende Zeile ändern:

```
if (links < rechts) then ...
```

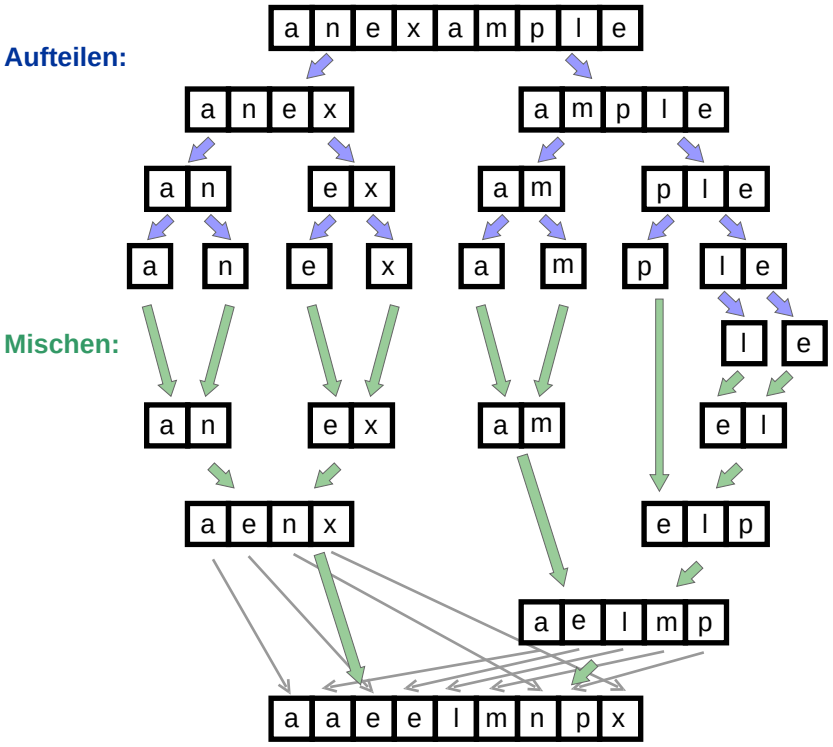
wird zu

```
if (rechts-links > M) then ...
```

9.5.2 Mergesort

Mergesort ist ein Sortierverfahren, dass (wie Quicksort) in der Zeitkomplexitätsklasse $\mathcal{O}(N \cdot \log N)$ liegt. Er zeigt bessere Performance als Quicksort im Worst-Case-Szenario und ist ein stabiles Sortierverfahren. Da auf die Elemente im zu sortierenden Feld sequentiell zugegriffen wird, ist er auch gut zur Implementation mit verketteten Listen geeignet. Allerdings benötigt Mergesort $\mathcal{O}(N)$ zusätzlichen Speicher, währen Quicksort in-place abläuft. Die Idee von Mergesort ist, dass sich zwei sortierte Listen in linearer Zeit zu einer sortierten Liste vereinigen lassen. Außerdem wird ausgenutzt, dass eine Liste mit nur einem Element immer sortiert ist. Der Algorithmus läuft ebenfalls rekursiv und nach dem teile-und-hersche-Prinzip ab.

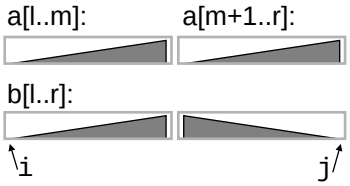
Die folgende Abbildung verdeutlich das Prinzip des Algorithmus:



In Pascal sieht Mergesort so aus:

```
1  { N: länge des Feldes a[1..N], das sortiert werden soll }
2
3  procedure merge(links:integer, mid:integer, rechts:integer);
4  var i,j,k:integer;
5      b: array[links..rechts] of datatype;
6  begin
7    { Mischen der sortierten Unterfelder
8      dabei wird ein Hilfsfeld b verwendet
9        1. Daten in Hilfsfeld kopieren }
10   for i:=mid downto links do
11     b[i] := a[i];
12   for j:=mid+1 to rechts do
13     b[rechts+mid+1-j] := a[j];
14
15   {       Die Felder sind jetzt in b[] so angeordnet, dass sich die
16           jeweils größten Elemente in der Mitte befinden
17           i und j zeigen auf auf den linken, bzw. rechten Rand
18
19         2. Mischen: }
20   for k:=links to rechts do
21     if b[i]<b[j] then begin
22       a[k]:=b[j];
23       i:=i+1;
24     end else begin
25       a[k]:=b[j];
26       j:=j-1;
27     end;
28 end;
29
30 procedure mergesort(links:integer, rechts:integer);
31 var mid:integer;
32 begin
33   if (rechts-links>0) then begin { Abbruchbedingung für Rekursion }
34     { Mittleres Element bestimmen und rekursiver Abstieg }
35     mid := (rechts+links) div 2;
36     mergesort(links, mid);
37     mergesort(mid+1, rechts);
38
39     merge(links, mid, rechts);
40   end;
41 end;
```

Der trickreichste Anteil der Algorithmus ist wohl der Mergschritt: Wie man in der Abbildung rechts sieht werden die Elemente geschickt im Hilfsfeld `b[]` angeordnet. Die „Zeiger“ `i` und `j` wandern von links, bzw. rechts zum Zentrum. Es wird jeweils das kleinere Element eingefügt und der entsprechende Zeiger verschoben, bis alle Elemente eingefügt sind. Die Stabilität von Mergesort hängt nur von der Stabilität des Merge-Schrittes ab!



Obiger Algorithmus nutzt einen Top-Down-Ansatz. Man kann Heapsort aber auch mit einem Bottom-Up-Ansatz programmieren. Dazu Teilt man `a[]` zuerst in lauter Subarrays der Länge $m_1 = 1$ und führt merge für jeweils zwei dieser Elemente aus. Danach teilt man `a[]` in Subarrays der Länge $m_2 = 2 \cdot m_1 = 2$ und mischt wieder jeweils zwei. Dies geht unter Verdopplung von $m \equiv m_i = 2 \cdot m_{i-1}$ immer weiter, bis nur noch zwei Teilfelder vorhanden sind. In Pascal sieht das so aus (Es wird die selbe `merge`-Prozedur, wie oben verwendet):

```
1  procedure mergesortBU(links:integer, rechts:integer);
2  var i,m:integer;
3  begin
4    m:=1;
5    while (m< rechts-links) do begin
6      i:=1;
7      while (i <= rechts-m) do begin
8        merge(i, i+m-1, min(i+2*m-1, r)); { min() stellt sicher, dass das Array
9                                            maximal bis r geht! }
10       i:=i+2*m;
11     end;
12     m:=2*m;
13   end;
14 end;
```

Man kann Mergesort unter Anderem mit folgenden Verfahren verbessern:

- Hybrid-Verfahren, wie bei Quicksort
- Kopieroperationen vermeiden

9.5.3 Heapsort

Der Algorithmus Heapsort basiert auf Prioritätswarteschlangen (siehe 8.2), wie z.B. Heaps. Eine solche Datenstruktur enthält eine Menge von Elementen a_i und gibt jeweils das größte Element zurück. Es ist möglich solche Heaps aus N Elementen in $\mathcal{O}(N \cdot \log N)$ aufzubauen. Das hinzuzufügen oder entfernen eines Elementes ist in $\mathcal{O}(\log N)$ möglich. Es ist dann nur noch nötig die Elemente nacheinander aus dem Heap zu entfernen, was in linearer Zeit möglich ist. Damit kann man mit Heapsort in $\mathcal{O}(N \cdot \log N)$ sortieren. Das Verfahren ist nicht stabil, kann aber in-place implementiert werden.

Die Funktion `downheap()` und damit ebenfalls `removemax()` läuft in logarithmischer Zeit ($\mathcal{O}(\log N)$) ab. Damit kann man nun Heapsort ebenfalls sehr einfach implementieren:

```
1    { Array a[1..M] sortieren, N: Länge des Heaps }
2    N:=0; { Heap ist am Anfang leer }
3    for k:=1 to M do insert(a[k]); { Elemente einfügen }
4    for k:=M downto 1 do a[k]:=removemax; { Elemente nacheinander entfernen }
```

Da M Elemente eingefügt werden sollen und alle Heap-Operationen in $\mathcal{O}(\log M)$ ablaufen, läuft somit HEapsort in $\mathcal{O}(M \cdot \log M)$ ab. Diese Implementation benötigt aber zwei Arrays; eines für die Daten (Ein-/Ausgabe) und ein weiteres für den Heap. Man kann aber Heapsort auch in-place implementieren:

```
1  { M: länge des Feldes a[1..M], das sortiert werden soll; N: Länge des Heaps }
2
3  procedure Heapsort;
4  var k:integer;
5                  t:datatype;
```

```
6  begin
7    N:=M; { Heap belegt ganzes (unsortiertes Feld) }
8    { nun muss die Heap-Eigenschaft hergestellt werden:
9        * für die unterste Ebene ist sie immer erfüllt (keine Nachfolger)
10       * für alle anderen Ebenen muss sie von unten nach oben
11         hergestellt werden. }
12   for k:=M div 2 downto 1 do
13     downheap(k);
14   repeat
15     t:=a[1]; a[1]:=a[N]; a[N]:=t; { erstes und letztes Element vertauschen }
16     { Heap verkürzen (letztes Element ist größtes)
17       und damit letztes Element schützen }
18     N:=N-1;
19     downheap(1); { Heap-Eigenschaft wieder herstellen }
20   until N<=1; { bis Heap leer }
21 end;
```

9.6 Suchmethoden

9.6.1 Binäre Suche

Wenn in einem sortierten Feld `a[1..N]` gesucht werden soll, so ist dies mit $\mathcal{O}(\log N)$Vergleichen möglich. Das folgende Pascal-Programm erledigt die Suche:

```
1  function BinarySearch(find:datatype):integer
2  var x,l,r:integer;
3  begin
4    l:=1;
5    r:=N; {linken und rechten Rand festelegen}
6    repeat
7      x:=(l+r) div 2; { Mitte festlegen }
8      if find<a[x].key then r:=x-1 else l:=x+1; { Bereich einschränken }
9    until (find=a[x].kea) or(l>r) { bis gefunden, oder renze kollidieren }
10   if find=a[x].key then result:=x else result:=-1;
11 end;
```

9.6.2 Binärer Suchbaum

Ein binärer Suchbaum ist ein binärer Baum mit folgenden Eigenschaften:

1. Die Schlüssel werden in den Knoten und Blättern gespeichert.
2. Die Schlüssel der linken Kinder den Knoten k sind alle echt kleiner als der Schlüssel des Knotens k und die Schlüssel seiner rechten Kinder sind immer größer oder gleich seinem eigenen Schlüssel.

Im folgenden werden Knoten in Suchbäumen durch den Datentyp `node` repräsentiert:

```
1  type pnode = ^node;
2       node  = record
3                 key:keytype;
4                 data:datatype;
5                 left,right:pnode;
6               end;
7  var head: pnode; { zeigt auf den Kopf des Suchbaumes}
```

Für die Schlüssel muss immer gelten, dass auf ihnen eine totale Ordnung definiert ist.

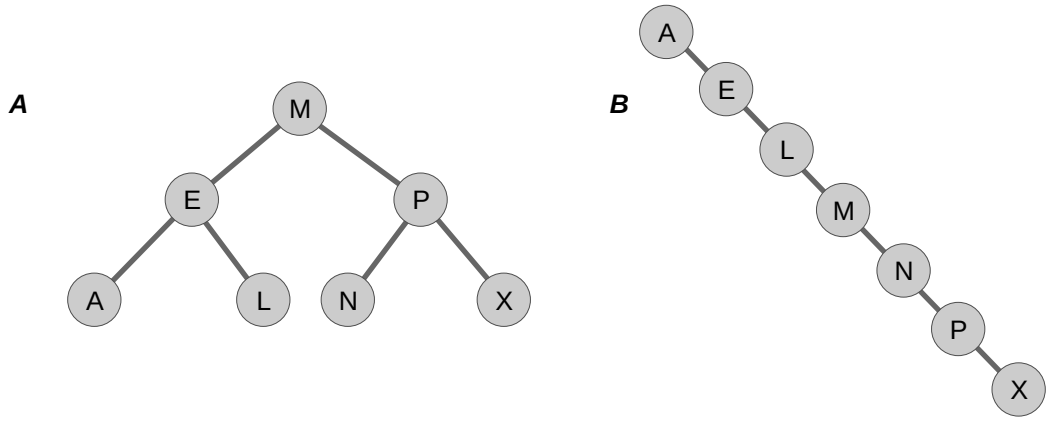
Suchen

Man erhält dann eine einfache binäre Suche in diesem Baum (Die Variable `startfrom` kann für totale Suchen auf `head` gesetzt werden). Es gibt eine nicht-rekursive und eine rekursive Implementation:

```
function binarySearch(find:keytype; startfrom: pnode);
var x: pnode;
begin
  x:=startfrom;
  if x^.key<>find then repeat
    if find<x^.key then x:=x^.left else x:=x.right;
  until (find = x^.key)or((x^.left=nil)and(x^.right=nil));
  result:=x;
  if (x^.key<>find) then result:=nil; { nicht gefunden }
end;

function binarySearchRec(find:keytype; x: pnode);
begin
  if (x=nil) then result:=nil { Ende des Baumes }
  else begin
    if x^.key=find then result:=x { gefunden! }
    else begin
      if find<x^.key then result:=binarySearchRec(find, x^.left) { links suchen }
      else result:=binarySearchRec(find, x^.right); { rechts suchen }
    end;
  end;
end;
```

Man kann sich leicht überlegen, dass die Performance dieses Algorithmus von der Form des Baumes abhängt. Es gibt zu jeder Menge an Knoten/Schlüsseln eine zugehörige Menge von unterschiedlichen Bäumen. Die folgende Abbildung zeigt zwei Extremfälle. Im Baum A kann in logarithmischer Zeit gesucht werden. Im Baum B benötigt die Suche lineare Zeit:

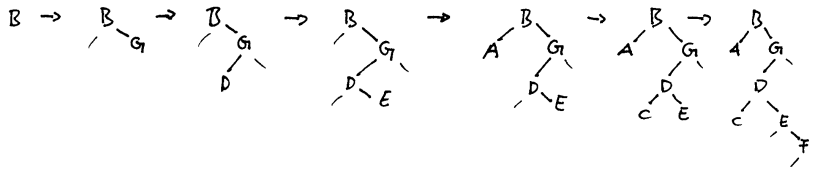


Einfügen

Das nächste Problem ist also das Einfügen in einen Suchbaum. Hierzu gibt es ein einfaches Verfahren, dass aber u.U. Bäume des Types B in obiger Abbildung erzeugt. Es sucht rekursiv nach dem einzufügenden Element und fügt es an der Stelle ein, an der die Suche im Misserfolgsfalle abbrechen würde:

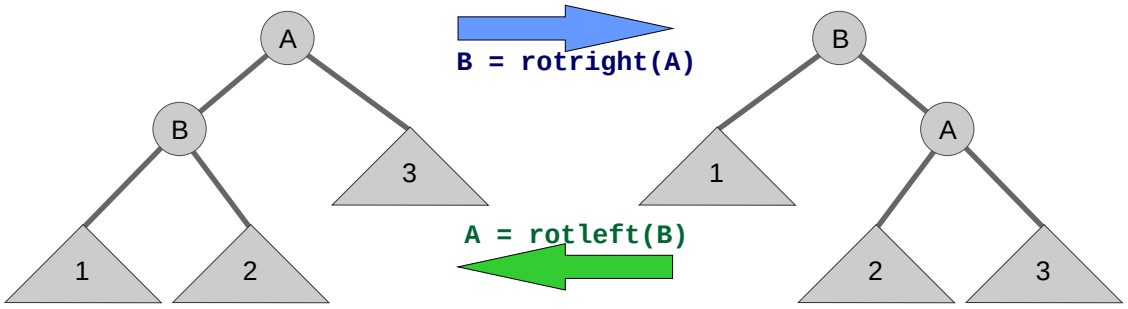
```
function insert(data:node; x: pnode):pnode;
begin
  if (x=nil) then begin
    result:=NEWnode(data); { falls nicht gefunden: neuen Knoten erzeugen und
                             anstatt nil-Blatt anhängen }
  end else if x^.key=data.key then result:=x  { bereits vorhanden: nichts machen }
  else begin
    if data.key<x^.key then x^.left:=insert(data, x^.left) { links weiter }
    else x^.right:=insert(data, x^.right); { rechts weiter }
    result:=x;
  end;
end;
```

Die folgende Abbildung zeigt, was passiert, wenn man in einen anfangs leeren Suchbaum die Zeichen `B G D E A C F` in dieser Reihenfolge einfügt:



Wie man sieht hat der so aufgabeute Baum relative große Tiefe. Er könnte zumindest wesentlich breiter sein. Erfolgt das Einfügen so, dass sich ein ausgeglichener Baum ergibt, so ist das Einfügen in $\mathcal{O}(\log N)$ Schritten möglich. Im schlimmsten Falle wird das hier geschilderte Einfügen (wie das Suchen) linear.

Man kann den Algorithmus dahingehend verbessern, dass man neue Knoten immer als Wurzel einfügt. Dies erlaubt kürzere Suchzeiten, wenn nach kürzlich eingefügten Knoten gesucht wird. Dazu benötigt man sog. „Rotationen“ im Baum:



Eine solche Rotation tauscht also zwei Knoten aus, wobei die Suchbaum-Eigenschaft erhalten bleibt. Zu beachten ist, dass die graphische Reihenfolge der Kinder 1, 2, 3 gleich bleibt. Der folgende Pascal-Code gibt eine Implementation der Rotationen:

```
function rotleft(b:pnode):pnode;
var a:pnode;
begin
  a:=b^.right;
  b^.right:=a.left;
  a^.left:=b;
  result:=a;
end;
```

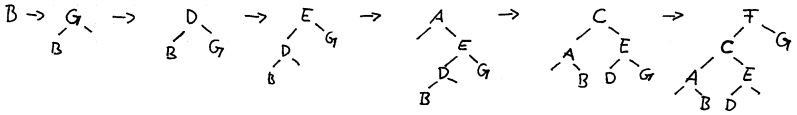
```
function rotright(a:pnode):pnode;
var b:pnode;
begin
  b:=a^.left;
  a^.left:=b^.right;
  b^.right:=a;
  result:=b;
end;
```

Damit erhält man folgendes Code für das Einfügen eines Knotens als Wurzel (`rootinsert`). Der Knoten wird zuerst ganz normal eingefügt. Danach wird er bis zur Wurzel hochrotiert.

```
function rootinsert(data:node; x: pnode):pnode;
begin
  if (x=nil) then begin
    result:=NEWnode(data); { falls nicht gefunden: neuen Knoten erzeugen und
                             anstatt nil-Blatt anhängen }
  end else if x^.key=data.key then begin result:=x  { bereits vorhanden: nichts machen }
  end else begin
    if data.key<x^.key then begin
```

```
9         x^.left:=rootinsert(data, x^.left) { links rekursiv weiter }
10        x:=rotright(x); { nach oben rotieren }
11      ende else begin
12        x^.right:=rootinsert(data, x^.right); { rechts rekursiv weiter }
13        x:=rotleft(x); { nach oben rotieren }
14      end;
15      result:=x;
16    end;
17  end;
```

Die folgende Abbildung zeigt einen Baum, in den die Knoten `B G D E A C F` in dieser Reihenfolge einfügt wurden, und zwar mit `rootinsert()`:



Löschen

Eine weitere wichtige Operation in Suchbäumen ist das Löschen eines Elementes. Dabei soll natürlich die SUchbaumeigenschaft erhalten bleiben. Hier gilt es drei Fälle zu unterscheiden:

1. Es soll ein Blatt gelöscht werden $\Rightarrow$ einfach löschen
2. Es soll ein Knoten mit nur einem Kind gelöscht werden $\Rightarrow$ das Kind einfach nach oben schieben
3. Es soll ein Knoten k_d mit zwei Blättern gelöscht werden $\Rightarrow$ such nach dem nächstgrößeren Knoten k_n im Baum und ersetze den zu löschenden Knoten durch diesen. Der nächstgrößere Knoten k_n hat garantiert nur einen rechten Nachfolge, da es keine Knoten zwischen diesem und k_d gibt. Die Kinder von k_d werden einfach die Kinder von k_n. Das alte rechte Kind von k_n muss dann einfach in den neuen SUchbaum eingefügt werden (`insert`).

Zusammenführen zweier Bäume (Join)

Um zwei Suchbäume A, B zusammenzuführen wird einer der beiden Wurzeln zur neuen Wurzel gemacht. Der Wert der zweiten Wurzel wird einfach in den neuen Suchbaum eingefügt (`rootinsert`). Die linken und rechten Teilbäume von A und dem erweiterten B werden paarweise rekursiv zusammengeführt und an die Wurzel von B angehängt:

```
1  function join(a:pnode; b:pnode):pnode;
2  begin
3    if (a=nil) then result:=b
4    else if (b=nil) then result:=a
5    else begin
6      b:=rootinsert(a^.data, b); { a in b einfügen }
7      b^.left:=join(a^.left, b^.left);
8      b^.right:=join(a^.right, b^.right);
9      free(a);
10     result:=b;
11   end;
12 end;
```

9.6.3 Randomisierte Binäre Suchbäume

Um den schlechtesten Fall zu vermeiden, in dem sich ein linearer Suchbaum ergibt, kann man folgende Strategie anwenden: Jedesmal wenn ein Knoten eingefügt werden soll, so wird er mit der Wahrscheinlichkeit $\frac{1}{N+1}$ zur Wurzel. Andernfalls wird er normal eingefügt. Dabei ist N die Größe des Baumes, dessen Wurzel gerade betrachtet wird. Ein einzelner Knoten repräsentiert einen Baum der Größe $N = 1$. Ein Knoten mit zwei Kindern, die keine Kindern besitzen hat $N = 3$ usw. Mit dieser Methode wird es wahrscheinlicher, dass ein Baum balanciert ist. Die Wahrscheinlichkeit, einen Knoten als Wurzel zu ersetzen steigt mit der Tiefe des Baumes. Auf der untersten Ebene ist die gerade $\frac{1}{2}$.

Als Erweiterung der in 9.6.2 vorgestellten Datenstruktur ist es empfehlenswert die aktuelle Größe des Unterbaumes N in jedem Knoten mit abzuspeichern.

Beim joinen zweier randomisiertes Suchbäume wird der oben bereits beschriebene Ansatz so erweitert, dass die Entscheidung, ob A (Länge N_A) oder B (Länge N_B) die neue Wurzel wird, mit den Wahrscheinlichkeiten $p_A = \frac{N_A}{N_A+N_B}$ und $p_B = \frac{N_B}{N_A+N_B}$ gewichtet ist.

9.6.4 2-3-4-Bäume und Rot-Schwarz-Bäume

2-3-4-Bäume

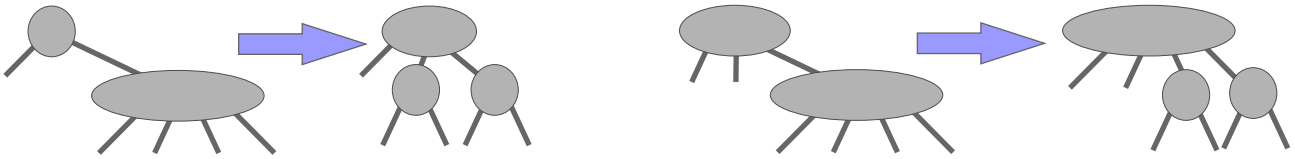
Um die Anzahl der Vergleiche und die Tiefe der Bäume zu reduzieren muss man zusätzliche Freiheiten im Baum schaffen. Ein Aufweichen der Suchbaum-Eigenschaft ist nicht zweckmäßig, da ja gesucht werden soll. Deswegen kann man die Anzahl der Kinder eines Knotens erweitern, sodass weniger Ebenen für gleich viele Knoten nötig werden. Es werden dazu mehr als ein Schlüssel/Datum-Paar pro Knoten gespeichert. Man benötigt dann zwar mehr Vergleiche in jedem Knoten, muss dafür aber die Suche weniger tief betreiben. Man erlaubt daher die folgenden Knoten:

- **2-Knoten** enthalten weiterhin einen Schlüssel. Links liegen die kleineren rechts die größeren Einträge.
- **3-Knoten** enthalten zwei Schlüssel und besitzen drei Kinder. Ganz links sind die Kinder, die kleiner als Beide und rechts die Kinder, die größer als Beide Schlüssel sind. In der Mitte befinden sich Kinder, die zwischen den Schlüsseln liegen.
- **4-Bäume** enthalten analog zu 3-Bäumen, drei Schlüssel und vier Kinder.

Die Suche läuft dann folgendermaßen ab:

1. Vergleiche den zu suchenden Schlüssel mit den Schlüsseln im Knoten.
2. Im Misserfolgsfall: Suche das passende Intervall und suche dort rekursiv weiter.

Das Suchen ist also nicht wesentlich schwieriger, als das Suchen in Binärbäumen. Allerdings wird das Einfügen in einen Binärbaum wesentlich aufwändiger. Zuerst sucht man bis zur untersten Ebene nach einem Knoten, in den man einfügen kann. Das Einfügen in 2- und 3-Knoten ist relativ einfach. Der Schlüssel wird einfach in den entsprechenden Knoten eingefügt. Aus einem 2-Knoten wird also ein 3-Knoten und aus einem 3-Knoten ein 4-Knoten. Nur beim Einfügen in einen 4-Knoten ergeben sich Probleme, da dieser nicht weiter aufgeteilt werden kann. Für 4-Knoten kann man den Knoten in zwei 2-Knoten zerteilen und das mittlere Element in den Vaterknoten einfügen. Dies funktioniert solange der Vaterknoten selbst kein 4-Knoten ist. Um den letzteren Fall zu vermeiden, kann man beim Einfügen jeden 4-Knoten, auf den man trifft in zwei 2-Knoten aufspalten. So ist garantiert, dass nie der Fall eintritt, dass ein 4-Knoten Vater eines 4-Knoten ist. Die folgende Abbildung zeigt die Operationen auf 4-Knoten:



Rot-Schwarz-Bäume

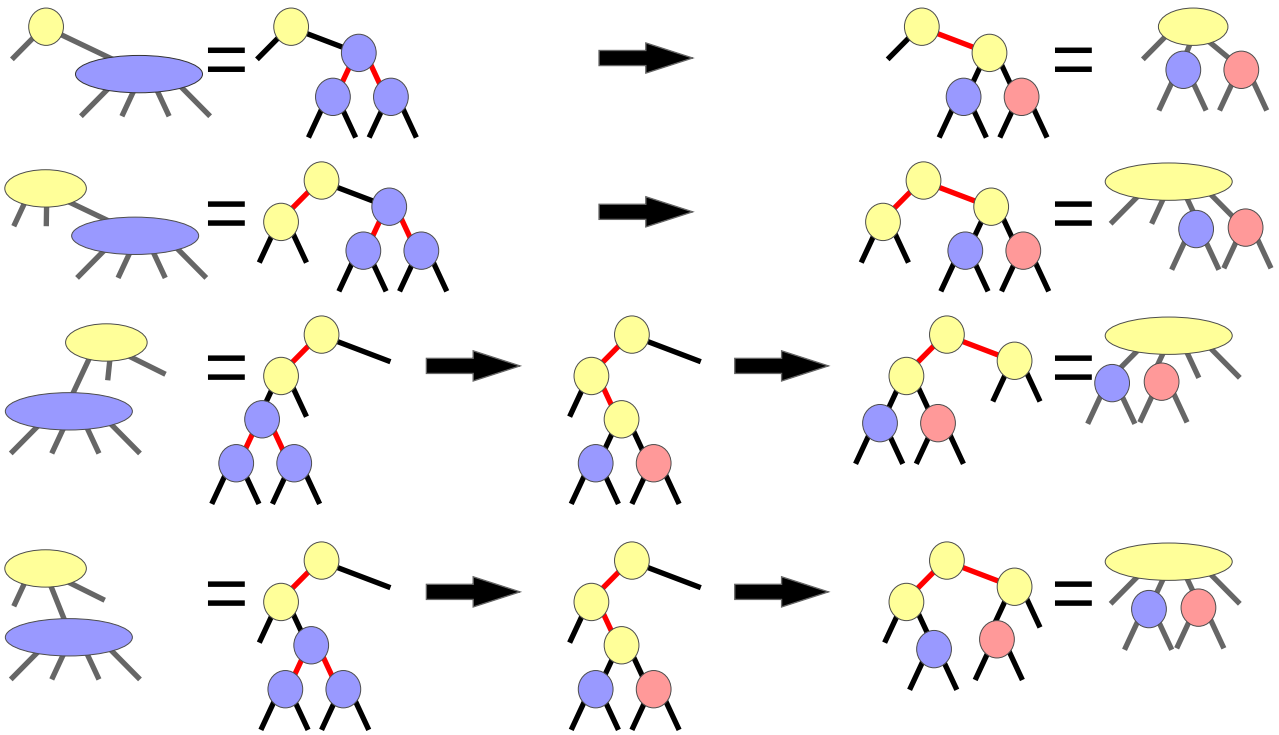
Die Implementierung von 2-3-4-Bäumen lässt sich über binäre Bäume erledigen, wenn man zusätzliche Informationen speichert. So lassen sich etwa 4-Knoten als drei 2-Knoten (also binäre Knoten) darstellen. Es gibt dann also zwei Sorten von Verbindungen zwischen Knoten:

- Verbindungen, die mehrere Knoten zu einem 3- oder 4-Knoten verbinden (rote Verbindungen)
- Verbindungen, die mehrere 2-, 3-, 4-Knoten miteinander verbinden (schwarze Verbindungen)

Die folgende Abbildung zeigt die möglichen Darstellungen:

Man muss also zu jedem Knoten nur zwei weitere Bits abspeichern (Booleans), die angeben, ob die entsprechende Verbindung rot oder schwarz ist.

Nun kann man relativ einfach die obigen Aufspaltungen von Knoten implementieren. Für die unteren beiden Operationen sind allerdings zusätzliche Vertauschungen notwendig:



Für Rot-Schwarz-Bäume gilt, dass der längste Pfad maximal doppelt so lang ist, wie der kürzeste. Außerdem ist die Tiefe des Baumes im schlimmsten Falle $2 \cdot \log N$.

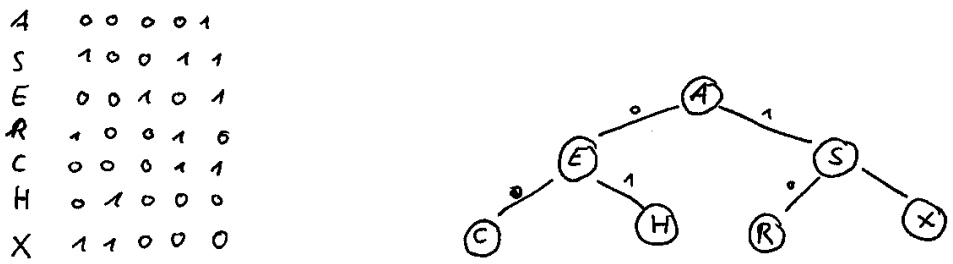
9.6.5 Digitale Suchbäume (digital search trees)

Für digitale Suchbäume werden die Bits der Binärrepräsentation der Schlüssel verwendet, um die Position im Baum zu bestimmen. Die Schlüssel werden in den Knoten gespeichert. Dabei wird das signifikanteste Bit zuerst betrachtet. Eine 1 führt zu einer Verzweigung nach rechts, eine 0 zu einer Verzweigung nach links. Damit traversiert man den bereits vorhandenen Baum, bis ein (leeres) Blatt gefunden wurde. In dieses wird dann der neue Schlüssel eingefügt. Beim suchen hangelt man sich anhand der Bits des Suchwortes durch den Baum. An jedem Knoten wird der dortige Schlüssel mit dem Suchwort verglichen. Stimmen sie überein, so kann die Suche beendet werden. Das folgende Pascal-Programm implementiert das Verfahren:

```
1  function digitalinsert(v:integer; x:pnode):pnode;
2  var p:pnode;
3      b:integer;
4  begin
5    b:=8; { max. Anzahl von Bits }
6    repeat
7      p:=x; { p zeigt immer auf den Vater des aktuellen Knotens }
8      { prüfe das b-te Bit und verzweige entsprechend }
9      if bit(v,b)=0 then x:=x^.left else x:=x^.right;
10     b:=b-1;
11   until x=nil; { bis ein Blatt erreicht ist }
12   { neuen Knoten erzeugen }
13   new(x);
14   x^.key:=v;
15   x^.left:=nil;
16   x^.right:=nil;
17   {Knoten einfügen }
18   if bit(v,b+1)=0 then p^.left:=x else p^.right:=x;
19   result:=x;
20 end;
21
22 function digitalsearch(v:integer; x:pnode):pnode;
23 var b:integer;
24     t:pnode;
25 begin
26   t^.key:=v;
27   b:=8; { max. Anzahl von Bits }
28   repeat
29     if bit(v,b)=0 then x:=x^.left else x:=x^.right;
30     b:=b-1;
31   until v=x^.key;
32   result:=x;
33 end;
```

Geht man von N Schlüsseln aus, die mit Hilfe von b Bits kodiert werden, so benötigt man beim Suchen in digitalen Suchbäumen durchschnittlich $\log N$ und schlimmstenfalls b Vergleiche.

Hier ein Beispiel für einen Suchbaum:



Es gibt noch effizientere Verfahren (siehe nächste Abschnite), die den Nachteil aufwiegen, dass in jedem Knoten ein Schlüsselvergleich nötig ist. Dieser kann bei langen Schlüsseln (z.B. strings) sehr aufwändig werden.

9.6.6 Digital Search Tries

In vielen Fällen ist die Länge der Schlüssel groß, sodass man einen Vergleich zweier Schlüssel vermeiden möchte. Alle bisherigen Suchverfahren benötigen im besten Fall $\mathcal{O}(\log N)$ Vergleiche. Digitale Tries erlauben es dies auf einen Vergleich zu beschränken, vorausgesetzt es ist billiger ein bestimmtes Bit aus dem Schlüssel zu extrahieren, als zwei Schlüssel zu vergleichen. Je nach Prozessorarchitektur kann die Extraktion eines Bits extrem billig sein, dann nämlich wenn der Prozessor den Zugriff auf einzelne Bits auf Maschinensprachen-Ebene unterstützt.

Man kann dann die Digitalen Suchbäume folgendermaßen abwandeln:

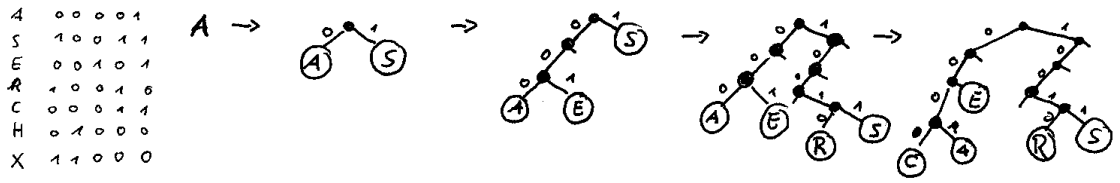
1. Die Schlüssel werden nur in Blättern gespeichert.
2. Innere Knoten enthalten keine Schlüssel. Die Verzweigung erfolgt wie bei Digitalen Suchbäumen.

Beim Einfügen eines Knotens in den Trie sind zwei Fälle möglich:

1. Die Suche endet an einem leeren Blatt $\Rightarrow$ Der Knoten kann einfach eingesetzt werden
2. Die Suche endet an einem besetzten Blatt $\Rightarrow$ Das Blatt wird durch einen inneren Knoten ersetzt, der auf die beiden neuen Knoten gemäß den entsprechenden Bits verweist. Sind die entsprechenden Bits gleich, so werden solange Unterknoten erzeugt, bis sich zwei Bits unterscheiden.

Man kann Such-Tries weiter verbessern, wenn man die Tiefe des Baumes reduziert. Dazu erlaubt man mehr als zwei Verzweigungen pro Knoten und betrachtet entsprechend auch mehr Bits pro Verzweigung. Das führt für allerdings für große Bitgruppen (Länge n) zu sehr vielen leeren Blättern, sodass man Hybrid-Verfahren benutzen kann, um auf der oberen Eben stark zu verzweigen und n mit der Tiefe des Baumes bis auf 1 abfallen lässt.

Das folgende Beispiel zeigt den Aufbau eines sehr tiefen Such-Trie:



9.6.7 Patricia-Trie

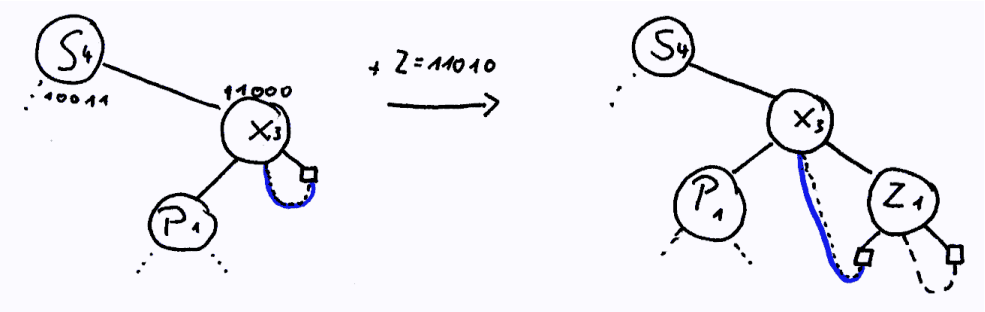
Die digitalen Such-tries haben zwei Nachteile, die folgendermaßen gelöst werden können:

1. **lange, einwegige Verzweigungskette:** Jeder Knoten erhält als zusätzliches Datum das Bit, das verglichen werden soll. Knoten auf höheren Ebenen vergleichen natürlich damit auch höherwertige Bits.
2. **zwei Arten von Knoten:** innere Knoten ohne Schlüssel und Blätter mit Schlüssel: Speichere die Schlüssel wieder in den inneren Knoten. Die Blätter enthalten dann nicht einen verweis auf `nil` (kein Nachfolger), sondern auf den inneren Knoten mit dem entsprechenden Schlüssel (Verweise nach oben im Baum).

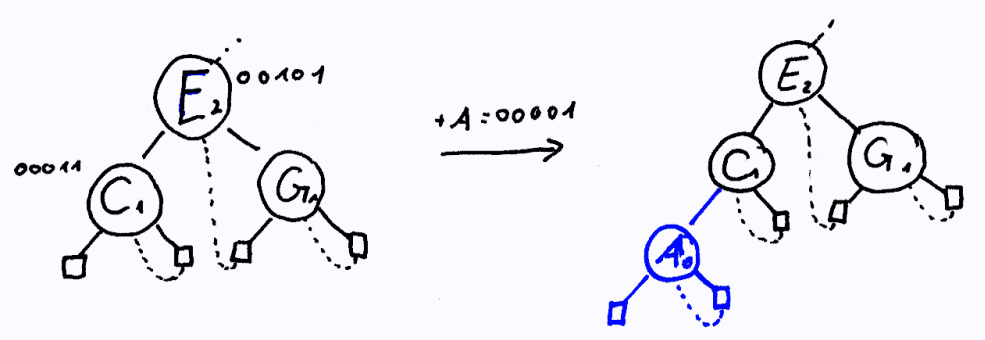
Beim Suchen bewegt man sich zunächst ohne Schlüsselvergleich anhand der Bits nach unten und vergleicht erst zum Schluss die Schlüssel. Um festzustellen, ob ein Verweis nach oben oder nach unten zeigt, muss man nur nachsehen, ob die Bit-Indizes größer werden (nach oben) oder kleiner (nach unten).

Einfügen in Patricia-Tries: Es gilt einige Fälle zu unterscheiden. Zunächst wird im Baum gesucht, bis man entweder feststellt, dass der Schlüssel bereits vorhanden ist, oder zu einem Blatt kommt.

1. Enthält das Blatt an der Stelle des Einfügens keinen Verweis nach oben, so kann man einfach den neuen Schlüssel einfügen:



2. trifft man auf ein Blatt mit Verzweigung, so wird es trotzdem durch den neuen Schlüssel ersetzt. Die Verzweigung muss aber nach folgendem Schema wieder hergestellt werden:



9.6.8 Hashing

Grundlagen

Hashing ist ebenfalls eine Suchmethode. Allerdings basiert sie nicht auf Bäumen, sondern darauf, dass man den zu suchenden Schlüssel über eine Funktion $f : K \to I$ direkt auf einen Array-Index abbildet. Eine Funktion, die dies in einer Weise macht, sodass jeder mögliche Schlüssel eineindeutig auf einen Index abgebildet wird ist nicht möglich, wenn man noch Zeit gegenüber linearer oder logarithmischer Suche gewinnen will. Darum verwaltet man zu jedem Array-Index eine Liste von zugeordneten Einträgen, die dann sehr kurz ist (vorausgesetzt es gibt genügend Indizes!). Damit ist die Komplexität des Algorithmus im Idealfall $\mathcal{O}(1)$ und im schlimmsten Falle $\mathcal{O}(N)$, aber mit sehr kleinen Konstanten.

Das Hashing verwendet also mehr Speicherplatz, um die Rechenzeit zu begrenzen. Es gibt eine mögliche perfekte Hash-Funktion für natürliche Zahlen als Schlüssel. Diese ist die Identität, es wird also der Schlüssel direkt als Index verwendet. Im Allgemeinen ist das aber nicht möglich, da die Schlüsselmenge K sehr große ist und man eine Verkleinerung dieser erreichen möchte.

Das Ziel der Erstellung einer Hash-Funktion ist es, eine möglichst gleichmäßige Abbildung der Schlüssel aus K auf die Indizes aus I zu erreichen. Eine recht erfolgreiche Funktion ist:

$$f_M(x) := x \mod M$$

Dabei ist M die Anzahl der Indizes, auf die abgebildet werden soll. M muss dabei eine Primzahl sein, da sonst zwei Gruppen von Schlüsseln auf den selben Index abgebildet werden. Für $M = 4$, würde auch ein Teil der durch zwei teilbaren Schlüssel auf den Index 0 abgebildet.

Um eine solche Operation auch für lange Schlüssel (z.B. String direkt als Schlüssel verwenden) noch effizient ausrechnen zu können, muss man das Horner-Schema ausnutzen. Wir betrachten einen Schlüssel, der sich folgendermaßen aus natürliche Zahlen $a_i \in [0, 255]$ zusammensetzt:

$$k := a_1 a_2 a_3 a_4$$

Dies könnte etwa ein String mit vier ASCII-Zeichen sein. Man kann ihn einerseits als ganze Zahl auffassen. Dies sprengt aber irgendwann alle Grenzen von Ganzzahltypen. Stattdessen kann man aber auch Komponentenweise vorgehen:

$$\begin{aligned} k \mod M &= [256 \cdot (256 \cdot (256 \cdot a_1 + a_2) + a_3) + a_4] \mod M = \\ &= [256 \cdot ([256 \cdot ([256 \cdot a_1 + a_2] \mod M) + a_3] \mod M) + a_4] \mod M \end{aligned}$$

Damit kann man die Hash-Funktion f_M in Pascal folgendermaßen notieren:

```
1 function hash(key:string; M:integer):integer;
2 const base:ineteger=117; { scramble by using 117 instead of 256 }
3 var i:longint;
4 begin
5   result:=0;
6   for i:=1 to length(key) do begin
7     result:=(a*result+integer(key[i])) mod M;
8   end;
9 end;
```

Ist vorher bekannt, wie viele Datensätze in den Hash einsortiert werden müssen, so kann man versuchen die verketteten Listen hinter jedem Array-Index (in jedem *Bin*) zu vermeiden. Geht man davon aus, dass N Datensätze gespeichert werden müssen, so verwendet man $M > N$ Bins und nutzt die leeren Indizes, um Kollisionen zu vermeiden. Die folgenden Abschnitte beschreiben einfache Methode um diese zu implementieren:

Linear Probing

Im Falle einer Kollision (Listenplatz ist bereits belegt, mit einem Schlüssel, der nicht dem einzufügenden entspricht) testet man einfach den nächsten Listenplatz. Dabei gibt es drei Möglichkeiten:

1. Der Platz ist besetzt und die Schlüssel stimmen überein, so gibt es nichts zu tun, da der Schlüssel bereits im Hash vorhanden ist.
2. Der Platz ist besetzt und die Schlüssel stimmen nicht überein. Nun wird einfach solange der nächste Platz geprüft, bis ein leerer Platz angetroffen wird.
3. Der Platz ist leer. Dann kann der Schlüssel hier eingefügt werden.

Es ist zu beachten, dass man evtl. die Suche am Anfang der Tabelle fortsetzen muss, wenn man deren Ende erreicht hat. Beim Suchen geht man dann einfach genauso vor: Ist der von der Hash-Funktion vorausgesagte Platz mit einem falschen Schlüssel besetzt, so sucht man linear im Hash weiter. Es gilt folgender Satz:

Eigenschaft des linearen Austestens: Für einen Hash, der zu weniger als $\frac{2}{3}$ gefüllt ist, sind mit linear probing durchschnittlich weniger als $x = 5$ Tests nötig. Die exakten Werte sind unter Benutzung des Auslastung $\alpha := \frac{M}{N}$:

$$x_{\text{erfolglos/insert}} = \frac{1}{2} \cdot \left[1 + \frac{1}{(1-\alpha)^2}\right] \qquad \text{und} \qquad x_{\text{erfolgreich/search}} = \frac{1}{2} \cdot \left[1 + \frac{1}{1-\alpha}\right].$$

Man verwendet typischerweise Hashes mit $M \approx 2 \cdot N$ Bins. Dieses Verfahren hat aber den Nachteil, dass im schlimmsten Falle alle Schlüssel überprüft werden müssen. Dieser Fall ist aber sehr unwahrscheinlich, da auch bei der Suche das Auftreten einer Leerstelle im Array als Misserfolg gewertet werden kann (sonst hätte ja das Einfügen die selbe Stelle gefunden). Außerdem sichert eine gute Hashfunktion eine Gleichverteilung der Schlüssel auf das Array, sodass der Abstand zur nächsten Leer stelle in der Größenordnung $\frac{1}{\alpha} = \frac{N}{M}$ ist.

Als **Clustering** bezeichnet man hier die Ansammlung von durchgehend gefüllten Bereichen im Hash.

Double Hashing

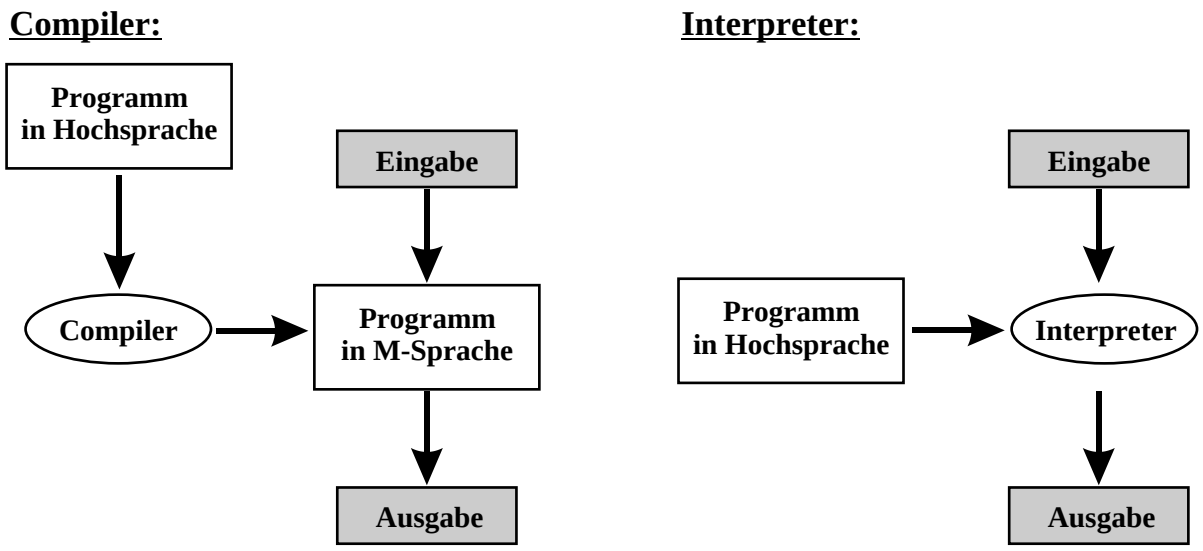
Um die Probleme des obigen Verfahrens zu lindern kann man statt einem linearen Durchsuchen des Arrays nach einem leeren Platz auch eine zweite Hashfunktion $g(k) \neq f(k)$ verwenden, die eine Suchfolge bestimmt. Die zweite Funktion bestimmt dann für jeden Index eine Sprungweite, mit der das Array abgesucht wird.

Teil III

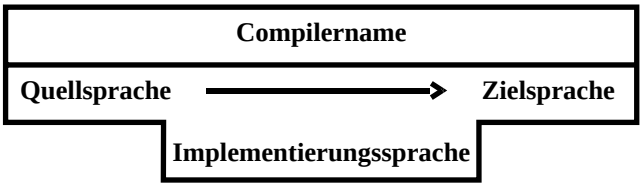
Compilerbau

10 Aufbau eines Compilers

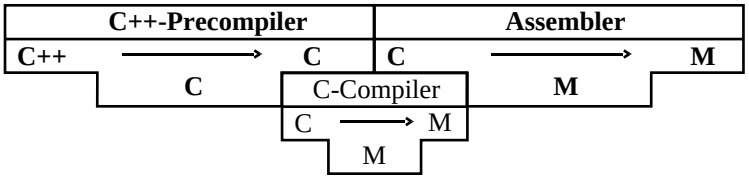
Ein Compiler oder „Übersetzer“ setzt ein in einer Hochsprache (C, C++, Java etz.) verfasstes Programm in ein Maschinenprogramm (x86-Maschinensprache, Java-Bytecode ...) um. Im Gegensatz dazu interpretiert ein Interpreter ein Hochsprachenprogramm und führt die dort aufgeführten Anweisungen direkt aus. Die folgende Grafik veranschaulicht den Unterschied:



Etwas allgemeiner übersetzt ein Compiler ein Programm in einer Quellsprache in eine Zielsprache. Der Compiler selbst muss natürlich auch in irgendeiner Sprache verfasst sein. Man kann dies in sog. T-Diagrammen ausdrücken:

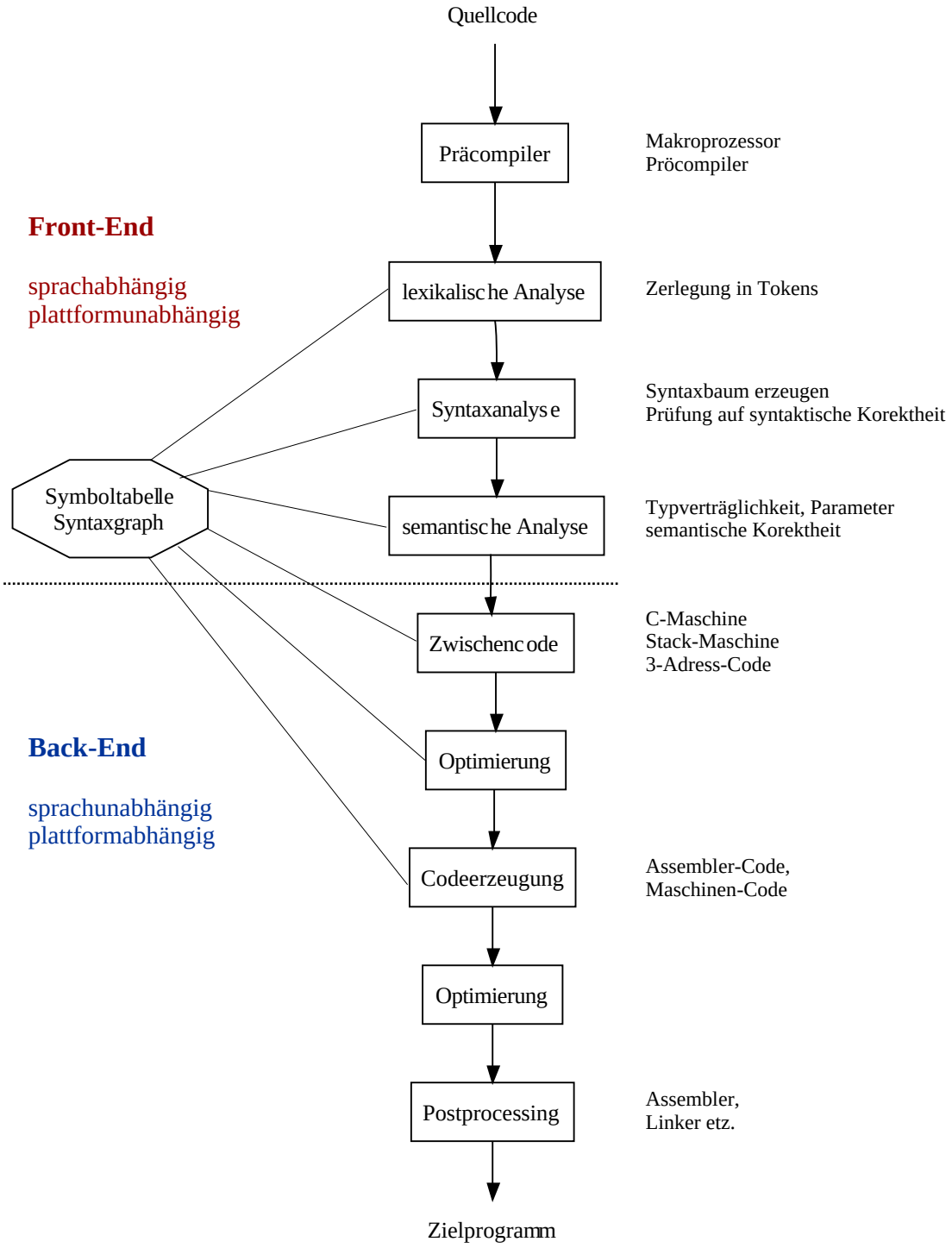


Nun kann man mehrere dieser Diagramme zusammenfügen, um den Aufbau komplexerer Übersetzer darzustellen:



Dabei übersetzt zunächst ein C++-Präprozessor den C++-Code in C-Code. Dieser wird dann von einem C-Compiler in Maschinencode übersetzt. Um den C++-Präprozessor ausführen zu können benötigt man einen C-Compiler, der ihn in Maschinensprache übersetzt.

Ein Compiler selbst ist ein recht kompliziertes Programm. Sein Ablauf kann am besten in folgendem Diagramm dargestellt werden:



Die Zeichnung führt die verschiedenen Schritte auf, die ein Programm innerhalb eines Compilers durchläuft. Als zentrale Datenstruktur hat man die Symboltabelle und den Syntaxgraphen. Die Symboltabelle verwaltet eine Liste von gültigen Bezeichnern (Variablen, Konstanten, Prozeduren etz.) im Programm. Der Syntaxgraph stellt eine Darstellung des Quellcodes im Speicher dar. Einige Sprachen bieten Präprozessoren (wie etwa C und C++). Deren Direktiven werden vor der Compilierung bearbeitet.

Zunächst wird das Programm in Tokens zerlegt. Jeder Token repräsentiert dabei ein atomares Element der Syntax (Bezeichner, Operator etz.). Aus diesem Token-Strom erzeugt die Syntaxanalyse einen Syntaxgraphen. Beim Aufbau dieses Graphen treten Syntaxfehler zu Tage und können gemeldet werden. Danach wird die Semantik des Programmes analysiert. Die verwendeten formalen Sprachen sind nicht mächtig genug, um etwa Typverträglichkeit zu codieren. Dies muss also gesondert untersucht werden. Außerdem werden hier (implizite) Typkonversionen eingefügt und evtl. die richtige überladene Methode ausgewählt.

Danach kann der Quellcode in eine Zwischensprache übersetzt werden. Diese ist schon recht nahe am Maschinencode und ermöglicht so eine einfache Optimierung. Außerdem können von der Zwischensprache ausgehend verschiedene Maschinencodes erzeugt werden. Erst bei der Übersetzung in den Maschinencode wird der SPeicher den Variablen zugewiesen (unterschiedliche Speichermodelle unterschiedlicher Prozessoren). Außerdem werden einige Variablen den internen Prozessorregistern zugewiesen. Zum Schluss müssen die erzeugten Maschinencode-Files evtl. noch gegeneinander und gegen vorcompilierte Bibliotheken gelinkt werden.

Diese gesamte Verarbeitung zerfällt in einen maschinenunabhängigen, aber sprachabhängigen Teil (Frontend) und einen maschinenabhängigen, aber sprachunabhängigen Teil (Backend). Man kann also die Ausgabe verschiedener Frontends mit ein- und demselben Backend verarbeiten. und umgekehrt die Ausgabe eines Frontends mit verschiedenen Backends für verschiedene Maschinen übersetzen (Cross-Compiler).

Die Optimierung des Codes greift üblicherweise am Zwischencode und/oder Maschinencode an. Im erstere können allgemeine Optimierungen vorgenommen werden:

- Elimination unbenutzer Variabler und nie verwendteer Programmteile
- Vereinfachung von arithmetischen Ausdrücken
- Vermeidung von Mehrfachberechnungen, Herausziehen von schleifeninvarianten Berechnungen
- Neuorganisation von Schleifen (schnellere Adressrechnung)

Auf Ebene der Maschinensprache können noch allgemeinere Eigenschaften des Programmes optimiert werden:

- Einsatz spezieller Maschinenbefehle (z.B. SSE-SSE3, MMX, 3Dnow!, spezielle Befehle auf DSPs ...)
- Optimierung der Registerbelegung.
- Elimination von überflüssigen Sprüngen

Beispiel: GNU Compiler Collection (GCC): Die GNU Compiler Collection (GCC) (`http://gcc.gnu.org/`) übersetzt den Quellcode in verschiedenen Sprachen (C, C++, Java, Objective-C, Fortran95, Treelang, Ada) mit Hilfe diverser Frontends in die Zwischensprache „GCCs Register Transfer Language (RTL)“. Diese kann von verschiedenen Backends für diverse Plattformen übersetzt werden (z.B. x86-Linux, x86-Win, x89-64, PowerPC, Atmel AVR, Motorola 68HC11, ARM, MMIX ...).

11 Lexikalische Analyse

Zur Syntaxanalyse werden die Techniken der formalen Sprachen (siehe Abschnitt 3) und der endlichen Automaten (siehe Abschnitt 6) verwendet. Man baut endliche Automaten auf, die gewisse Lexeme[1]/Tokens[2] (atomare Einheiten der Syntax) im Code erkennen. Die Automaten werden dann in realem Code simuliert. Als Ausgabe erhält man einen Strom von Tokens, die von den weiteren Programmteilen weiterverarbeitet werden können.

Man kann die einzelnen Tokens t_i über reguläre Ausdrücke definieren. Ist eine solche Definition gegeben, so kann man effektiv einen endlichen Automaten EA_i konstruieren, der solche Ausdrücke erkennt. Einen solchen baut man dann in die lexikalische Analyse zu Erkennung des Token t_i ein.

Ein Programmteil, das solches leistet wird **Tokenizer** oder **Lexer** genannt.

Beispiel: Im folgenden mathematischen Ausdruck sind die einzelnen Tokens markiert:

```
a = b*(c+23*sin(alpha))
```

```
[a] [=] [b] [*] [(] [c] [+] [23] [*] [sin(] [alpha] [)] [)]
```

Dies entspricht etwa dem folgenden Token-Strom

Token	Daten
IDENTIFIER	name=a
OP_EQUAL	
IDENTIFIER	name=b
OP_EQUAL	
OP_ASTERISK	
LBRACKET	
IDENTIFIER	name=c
OP_PLUS	
NUMBER	value=23
OP_ASTERISK	
FUNC_IDENTIFIER	name=sin
IDENTIFIER	name=alpha
RBRACKET	
RBRACKET	

Einem Token können also evtl. schon Daten zugeordnet werden.

[1] Als Lexem wird in der Linguistik üblicherweise eine Einheit des Wortschatzes einer Sprache bezeichnet, die über verschiedene grammatische Wörter abstrahiert, nach http://de.wikipedia.org/wiki/Lexem

[2] Ein Token ist ein Stück Text, dem von einer Grammatik eine Bedeutung zugewiesen wird. Dabei ist ein Token die lexikalische Grundeinheit, die ein Parser bearbeitet, nach http://de.wikipedia.org/wiki/Token_(Compilerbau)

Tokenizer für mathematische Ausdrücke: Der folgende C++-Code enthält einen Tokenizer für mathematische Ausdrücke:

```
1  jkmpTokenType jkMathParser::getToken(){
2    char ch=0;
3    while(program->get(ch) && isspace(ch)) { // eat white-spaces
4      ;
5    }
6
7    switch (ch) {  // recognize tokens
8      case 0:
9      case -1: return CurrentToken=END;
10     case ';': return CurrentToken=PRINT;
11     case '*':  return CurrentToken=MUL;
12     case '/':  return CurrentToken=DIV;
13     case '%':  return CurrentToken=MODULO;
14     case '+':  return CurrentToken=PLUS;
15     case '-':  return CurrentToken=MINUS;
16     case '(':  return CurrentToken=LBRACKET;
17     case ')':  return CurrentToken=RBRACKET;
18     case ',':  return CurrentToken=PARAMETER_DIV;
19     case '"':  return CurrentToken=STRING_DELIM;
20     case '^':  return CurrentToken=POWER;
21     case '!': { // logical NOT or factorial
22       char ch1=0;
23       if (*program) program->get(ch1);
24       if (ch1=='=') return CurrentToken=COMP_UNEQUAL;
25       // else
26       program->putback(ch1);
27       return CurrentToken=FACTORIAL_LOGIC_NOT;
28     }
29     case '&':{ // logical AND ist &&
30       char ch1=0;
31       if (*program) program->get(ch1);
32       if (ch1=='&') return CurrentToken=LOGIC_AND;
33       // else
34       program->putback(ch1);
35       jkmpError("undefined operator '&'; Did you mean LOGICAL_AND ('&&' / 'and')?");
36     }
37     case '=':{ // "' may be part of COMP_EQUAL (==) or ASSIGN (=)
38       char ch1=0;
39       if (*program) program->get(ch1);
40       if (ch1=='=') return CurrentToken=COMP_EQUAL;
41       // else
42       program->putback(ch1);
43       return CurrentToken=ASSIGN;
44     }
45     case '>':{
46       char ch1=0;
47       if (*program) program->get(ch1);
48       if (ch1=='=') return CurrentToken=COMP_GEQUAL;
49       // else
50       program->putback(ch1);
51       return CurrentToken=COMP_GREATER;
52     }
53     case '<':{
54       char ch1=0;
55       if (*program) program->get(ch1);
56       if (ch1=='=') return CurrentToken=COMP_SEQUAL;
57       // else
58       program->putback(ch1);
59       return CurrentToken=COMP_SMALLER;
60     }
61     case '0': case '1': case '2': case '3': case '4': // this is a number
62     case '5': case '6': case '7': case '8': case '9':
63     case '.': {
64       program->putback(ch);
65       (*program) >> NumberValue;
66       return CurrentToken=NUMBER;
67     }
68     default: // only few possibilities left: NAME, LOGICAL_AND (and), ..., or unknown
           token (error)
69       if (isalpha(ch) || (ch=='_')) { // try to recognize NAME, LOGIC_TRUE, LOGIC_FALSE,
             DIFF_LBRACKET
70         StringValue=ch;
71         while (program->get(ch) && (isalnum(ch) || (ch=='_')))
```

```
72            if (isalnum(ch) || (ch=='_')) StringValue.push_back(ch);
73
74          program->putback(ch); // now put the last thing read back int the stream, as it
75                      // could belong to the next token
76
77          if (StringValue=="true") return CurrentToken=LOGIC_TRUE;
78          if (StringValue=="false") return CurrentToken=LOGIC_FALSE;
79          if (StringValue=="and") return CurrentToken=LOGIC_AND;
80          if (StringValue=="or") return CurrentToken=LOGIC_OR;
81          if (StringValue=="xor") return CurrentToken=LOGIC_XOR;
82          if (StringValue=="not") return CurrentToken=LOGIC_NOT;
83          if (StringValue=="nor") return CurrentToken=LOGIC_NOR;
84          if (StringValue=="nand") return CurrentToken=LOGIC_NAND;
85
86          return CurrentToken=NAME;
87        }
88        // the parser has found an unknown token. an exception will be thrown
89        jkmpError("unknown token");
90    }
91  }
```

Dieser Programmabschnitt simuliert gerade einen Automaten, der die einzelnen Token erkennt. Zur Erkennung der meisten einfachen Token ist nur das Lesen eines einzelnen Zeichens nötig. werden mehr Zeichen benötigt, so werden diese ebenfalls gelesen. Die Zeile `(*program) >> NumberValue;` liest eine Gleitkommazahl ein. Der erkennende Automat (incl. Berechnung des Wertes der Zeichenfolge) ist hier in den Operatoren der C++-Standardbibliothek versteckt.

12 Syntaxanalyse

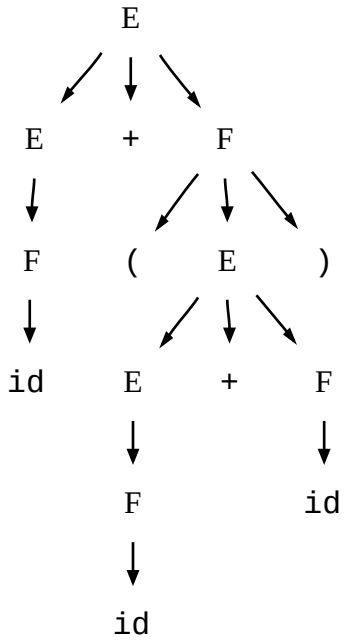
Der Parser verwendet den Token-Strom des Tokenizers, um einen Syntaxbaum aufzubauen. Ein Syntaxbaum ist im Prinzip ein Parse- oder Ableitungsbaum, der evtl. noch zusätzliche Informationen enthält. Die Syntaxanalyse löst also das **Wortproblem** und das **Analyse-Problem** einer Sprache $L(G)$ zu einer Grammatik G. Das erstere besteht darin festzustellen, ob ein Wort w zur Sprache $L(G)$ gehört. Das letztere besteht darin eine gültige Ableitung $S \overset{*}{\Rightarrow} w$ über G für ein gegebenes Wort w zu finden.

Programmiersprachen sind meist kontextfreie Sprachen (CH-2), da die Typ-3-Sprachen nicht einmal mächtig genug sind, um Klammerausdrücke darzustellen, während das Wortproblem für Typ-1-Sprachen nur mit exponentiellem Aufwand lösbar ist. Für viele Aufgaben reicht aber die Mächtigkeit der kontextfreien Sprachen nicht aus. So ist etwa die Sprache $\{w\texttt{OP=}\ \texttt{a}\ +\ w \mid w \in \Sigma\}$ nicht kontextfrei, sodass man mit diesen Sprachen nicht prüfen kann, ob eine Variable bereits deklariert wurde, bevor sie verwendet wird. Man verwendet darum „kontextsensitive Zusätze“, die aber nicht streng in der Grammatik der Sprache implementiert werden, sondern rein verbal. Sie werden dann in dem Compiler-Code integriert.

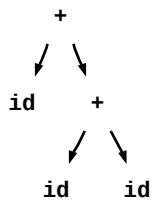
Die Syntaxanalyse erstellt also den Parsebaum. Dies soll am folgenden Beispiel verdeutlicht werden. Dazu betrachtet man folgende einfache Grammatik

$$\begin{array}{lcl} E & \rightarrow & E\ \mathbf{+}\ F \mid F \\ F & \rightarrow & \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id} \end{array}$$

Man betrachtet dann den Ausdruck **id+(id+id)** und den zugehörigen Sytaxgraphen/Parsebaum:



Dies entspricht einer bestimmten Abfolge von Regeln. Wählt man die falschen Regeln, so erhält man den falschen Parsebaum und interpretiert das Programm folglich falsch. Außerdem ist obiger Baum für eine Übersetzung fast schon zu kompliziert. Es genügt z.B. die folgende einfachere Form, da sie alle benötigten Informationen enthält:



Nun aber zur Syntaxanalyse selbst. Es gibt zwei Verfahren für die Syntaxanalyse, die im folgenden beschrieben werden sollen: Die Top-Down-Analyse und die Bottom-Up-Analyse. Die erstere generiert den Syntaxbaum von der Wurzel her. Die Bottom-Up-Analyse baut den Baum von den Blättern her auf.

$-Konvention: Im folgenden soll das Zeichen $ als Sonderzeichen eingeführt werden. Es markiert an einigen Stellen das Ende der Eingabe. Dafür muss das bisherige Axiom S durch ein neues Axiom S' ersetzt werden. Außerdem wird folgende Regel hinzugefügt:

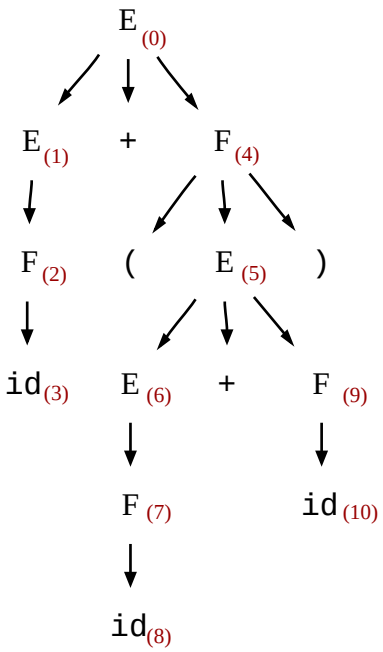
$$S' \rightarrow S\texttt{\$}$$

12.1 Top-Down-Syntaxanalyse

Wie im letzten Beispiel des vorherigen Abschnittes betrachtet man wieder die Grammatik:

$$\begin{array}{lcl} E & \rightarrow & E \texttt{ + } F \mid F \\ F & \rightarrow & \texttt{(} E \texttt{)} \mid \texttt{id} \end{array}$$

Nun wird der Parsebaum zum Satz **id+(id+id)** nummeriert, und zwar in der Reihenfolge, wie die Knoten erzeugt werden:



Dies entspricht einer Linksableitung des Satzes:

$$\begin{aligned} E &\Rightarrow E + F \Rightarrow F + F \Rightarrow \texttt{id} + F \Rightarrow \texttt{id} + (E) \Rightarrow \texttt{id} + (E + F) \\ &\Rightarrow \texttt{id} + (F + F) \Rightarrow \texttt{id} + (\texttt{id} + F) \Rightarrow \texttt{id} + (\texttt{id} + \texttt{id}) \end{aligned}$$

Allen Top-Down-Parsern ist also gemein, dass sie einen Satz in eine Linksableitung überführen. Sie finden also eine Linksableitung zu einem gegebenen Satz und einer gegebenen Grammatik G (Ableitungsproblem). Die Top-Down-Parser sind einfacher zu verstehen und einfacher in der Implementierung, als Bottom-Up-Parser (nächster Abschnitt 12.2).

12.1.1 Start- und Folgemengen

Für die Top-Down-Analyse ist es oft hilfreich die Anfänge der Regeln zu analysiere, um mit dieser Information eine Entscheidung über die anzuwendende Regel treffen zu können.

Startmengen-Abbildung: Die Startmengen-Abbildung $\text{FIRST} : 2^{(N \cup T)^*} \to 2^{T \cup \{\lambda\}}$ ist definiert durch:

1. $\text{FIRST}(\{\}) = \{\lambda\}$
2. $\text{FIRST}(\alpha) \equiv \text{FIRST}(\{\alpha\}) = \{z \in T | \alpha \overset{*}{\Rightarrow} zt,\ t \in T^*\} \cup \begin{cases} \{\lambda\} & \text{falls } \alpha \overset{*}{\Rightarrow} \lambda \\ \{\} & \text{sonst} \end{cases}$, für $\alpha \in (N \cup T)^*$
3. $\text{FIRST}(M) = \bigcup_{\alpha \in M} \text{FIRST}(\alpha)$, für $M \subseteq (N \cup T)^*$.

Eine Startmenge ist also die Menge der ersten Zeichen, die ausgehen von einem Nicht-Terminal erzeugt werden können. Es gilt außerdem $\text{FIRST}(t) = \{t\}$, falls $t \in T$ ein Terminal ist. In einem Parser kann man diese Information so benutzen: Liegt eine Produktion der Form $A \to \alpha_1 | \alpha_2 | ... | \alpha_n$ vor, und lautet die bisherigen Ableitung $S \overset{*}{\Rightarrow} w_1 ... w_l A \beta$, so kommen nur solche Regeln in Frage, für die $w_{l+1} \in \text{FIRST}\alpha_i$ gilt. Die λ-Regel $A \to \lambda$ ist anzuwenden, wenn $w_{l+1} \in \text{FIRST}\beta$ gilt.

Folgemengen-Abbildung: Die Folgemengen-Abbildung $\text{FOLLOW} : N \to 2^{T \cup \{\lambda\}}$ ist definiert durch:

$$\text{FOLLOW}(A) = \bigcup_{S \overset{*}{\Rightarrow} \alpha A \beta} \text{FIRST}(\beta) = \{z \in \text{FIRST}(\beta) | S \overset{*}{\Rightarrow} \alpha A \beta\}$$

Die Folgemengen enthalten also die führenden Terminalsymbole, die aus Wörtern folgen, die rechts von A stehen.

Beispiel: Betrachte die Grammatik mit $N = \{E, F\}, T = \{\texttt{+}, \texttt{(}, \texttt{)}, \texttt{id}\}$ und den Produktionen

$$\begin{array}{lcl} E & \to & E \texttt{ + } F \mid F \\ F & \to & \texttt{(} E \texttt{)} \mid \texttt{id} \end{array}$$

Daraus erhält man dann die First- und Follow-Tabellen:

	first	follow
E	(id	λ) +
F	(id	λ) +

Zur Ableitung dieser Tabellen betrachtet man einige repräsentative Ableitungen:

$$E \overset{*}{\Rightarrow} \texttt{(} E \texttt{)}, \quad E \overset{*}{\Rightarrow} \texttt{id + } F, \quad E \overset{*}{\Rightarrow} F \texttt{ + } F, \quad E \overset{*}{\Rightarrow} \texttt{(} F \texttt{)}$$

Algorithmus zur Bestimmung der Start- und Folgemengen: Man kann einen einfachen Algorithmus angeben, der die Startmengen zu einer Grammatik berechnet. Als Voraussetzung benötigt man nur ein Feld $d_\epsilon(A)$, das für alle Nicht-Terminale $A \in N$ angibt, ob von ihnen λ abgeleitet werden kann:

$$d_\epsilon(A) = \begin{cases} \text{true} & \text{es gibt eine Ableitung } A \overset{*}{\Rightarrow} \lambda \\ \text{false} & \text{sonst} \end{cases}$$

Der Algorithmus ist dann:

Algorithm 6 Konstruktion der FIRST-Mengen

1. für alle $A \in N$ setze $\text{FIRST}(A) = \{\lambda\}$, falls $d_\epsilon(A) = \text{true}$ und $\text{FIRST}(A) = \{\}$ sonst.
2. Für alle Terminale $a \in T$:
   a) Setze $\text{FIRST}(A) = \{a\}$
   b) Für alle Nicht-Terminale $A \in N$:
      i. Falls es eine Regel $A \rightarrow \mathbf{a}\ \alpha$ gibt, so setze $\text{FIRST}(A) = \text{FIRST}(A) \cup \{a\}$
3. Wiederhole bis sich keine Startmenge mehr ändert:
   a) berechne für alle Produktionen $A \rightarrow B\ \alpha$ $(B \in N)$ die Startmenge $\text{FIRST}(A)$, indem $B\ \alpha = w_1 w_2 ... w_k \in (N \cup T)^*$ in einzelne Symbole aus $N \cup T$ zerlegt wird, für die dann sukzessive die Startmengen $\text{FIRST}(w_i)$ berechnet werden. Die erste Startmenge, die nicht $\{\lambda\}$ oder $\{\}$ ist entspricht damit der Startmenge $\text{FIRST}(A)$.

Für die Folgemengen kann ein ähnlicher Algorithmus angegeben werden, der sich in der Standardliteratur findet.

12.1.2 LL(1)-Grammatiken

LL(1)-Grammatik: Eine kontextfreie Grammatik G heißt LL(1), wenn aus

1. $S \underset{l}{\overset{*}{\rightarrow}} xA\beta \underset{l}{\rightarrow} x\alpha\beta \overset{*}{\Rightarrow} xy$
2. $S \underset{l}{\overset{*}{\rightarrow}} xA\beta \underset{l}{\rightarrow} x\alpha'\beta \overset{*}{\Rightarrow} xy'$ und
3. $\text{FIRST}(y) = \text{FIRST}(y')$

folgt, dass $\alpha = \alpha'$. G heißt **einfach LL(1)**, falls für $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ die α_i mit paarweise unterschiedlichen Terminalsymbolen beginnen.

Eine λ-freie Typ-2-Grammatik G ist genau dann LL(1), wenn gilt: Seien $A \rightarrow \alpha_1|\alpha_2|...|\alpha_n$ alle A-Produktionen, dann sind die Startmengen $\text{FIRST}(\alpha_i)$ paarweise disjunkt.

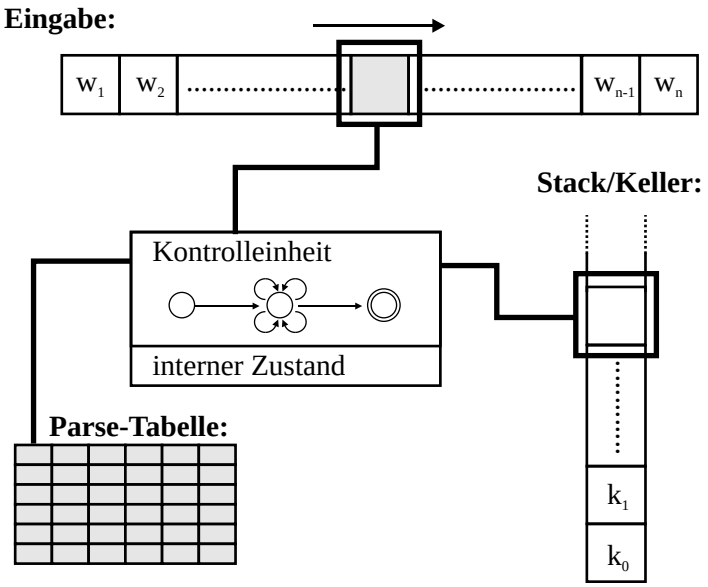
*Dies bedeutet, dass man durch lesen des nächsten Eingabesymbols eindeutig auswählen kann, welche Regel anzuwenden ist. Da man die Startmengen effektiv berechnen kann, kann man auch effektiv entscheiden, ob eine Grammatik G LL(1) ist (dieses Problem ist also entscheidbar). LL(k) steht dabei für „scanning from the **l**eft, using **l**eft productions, **k** symbols lookahead“. Im folgenden wird man sehen, dass man zu LL(1)-Grammatiken einen Parser entwerfen kann, der genau ein Symbol vorausschaut.*

Parsen einer LL(1)-Grammatik mit Kellerautomaten: Zu jeder kontextfreien Grammatik G gibt es einen nichtdeterministischen Kellerautomaten, der $L(G)$ aktzeptiert. Dieser Automat ist wie folgt definiert:

- $\Sigma_e = T$, $\Sigma_k = N \cup T \cup \{k_0\}$
- Der Automat hat nur drei Zustände $S = \{s_0, s_1, s_2\}$. Endzustand ist nur s_2, also $F = \{s_2\}$. Startzustand ist s_0.
- Es gibt vier Typen von Übergängen:
  1. **Start:** $\delta(s_0, \lambda, k_0) = \{(s_1, Sk_0)\}$ Es wird also das Xiom auf den Stack geschoben.
  2. **Ende:** $\delta(s_1, \lambda, k_0) = \{(s_2, k_0)\}$
  3. **Ableitung:** $\delta(s_1, \lambda, A) = \big\{(s_1, \alpha) \big| A \to \alpha \in P\big\}$, für alle Nicht-Terminale $A \in N$
  4. **Lesen:** $\delta(s_1, t, t) = \{(s_1, \lambda)\}$

Auf dem Stack befinden sich also immer die rechten Seiten von Regeln, die gerade gesucht werden. Steht auf dem Stack oben ein Terminal und ist dieses auch das aktuelle Eingabesymbol, so wird es erkannt (also gelöscht). Ein Nicht-Terminal A auf dem Stack wird in die rechte Seite einer A-Produktion expandiert. Der Automat simuliert also die Anwendung von Ableitungsregeln.

Diese Konstruktion ist aber noch mehrdeutig, weil nicht klar ist, welche Produktion evtl. abgeleitet werden soll. Diese Information kann der Parser aus einer sog. Parsetabelle beziehen. Man erhält dann folgende Konstruktion:



LL(1)-Parsetabelle: Die Parsetabelle $\mathcal{T}_G : N \times T \cup \{\lambda\} \to P \cup \{\textbf{error}\}$ ist eine Abbildung, die bei gegebenem Nicht-Terminal N (oberstes Kellersymbol) und einem Eingabezeichen $t \in T \cup \{\lambda\}$ die anzuwendende Produktion angibt, oder einen Fehler **error** meldet. Aus den Start- und Folgemengen lässt sich die Parse-Tabelle berechnen:

Algorithm 7 Konstruktion der LL(1)-Parsetabelle

1. Setze alle Einträge auf **error**
2. Für alle Regeln $p = (A \rightarrow \alpha) \in P$:
   a) Falls $\lambda \in \textsc{First}(\alpha)$
   $$\mathcal{F} = \big(\textsc{First}(\alpha) - \{\lambda\}\big) \cup \textsc{Follow}(A)$$
   sonst
   $$\mathcal{F} = \textsc{First}(\alpha)$$
   b) Für alle $t \in \mathcal{F}$:
      i. Falls $\mathcal{T}_G(A,t) =$ **error**, setze $\mathcal{T}_G(A,t) = p$, sonst ist die Grammatik nicht LL(1) (Fehler!)

Im letzten Schritt tritt genau dann ein Fehler auf, wenn es bei den Regeln Mehrdeutigkeiten gibt, also die Startmengen nicht paarweise disjunkt sind. Die Grammatik ist dann nicht LL(1). Die Tabelle wird nach folgendem Prinzip aufgebaut: Ist $A \rightarrow \alpha$ eine Produktion, dann wird sie expandiert, wenn das aktuelle Lese-Symbol a in $\textsc{First}(\alpha)$ ist.

rekursiver Abstieg: Man kann den obigen Kellerautomaten explizit implementieren. Es gibt aber noch eine alternative Methode. Sie wird rekursiver Abstieg genannt. Dabei wird der Stack implizit in Funktionsaufrufen „versteckt“. Dieses Prinzip ist sehr einfach und man kann leicht Parser implementieren, die diesem Prinzip folgen. Der folgende Code-Ausschnitt zeigt einen Teil eines entsprechenden Parsers in C++, der die folgende Grammatik parsed:

$$\begin{array}{rcl}
\textit{mathExpression} & \rightarrow & \textit{term}\ \texttt{+}\ \textit{term} \\
 & & \mid \textit{term}\ \texttt{-}\ \textit{term} \\
\textit{term} & \rightarrow & \textit{primary}\ \texttt{*}\ \textit{term} \\
 & & \mid \textit{primary}\ \texttt{/}\ \textit{term} \\
\textit{primary} & \rightarrow & \texttt{number} \\
 & & \mid \texttt{name} \\
 & & \mid \texttt{(}\ \textit{mathExpression}\ \texttt{)} \\
 & & \mid \texttt{-}\ \textit{primary}
\end{array}$$

Hier der Quellcode in C++. Er benötigt einen Tokenizer, wie er etwa in 11 beschrieben wurde, der das nächste Token liefert, wenn `getToken()` aufgerufen wird. Das aktuelle Token ist immer in `currentToken` gespeichert:

```
1  double parse(std::string prog){
2    double res;
3    while(true) {
4      getToken(); // call Tokenizer
5      if (CurrentToken == END) { break; }
6      res= mathExpression(false);
7    }
8    return res;
9  }
10
11
12 double mathExpression(bool get){
13   double left=term(get);
14   for(;;) // forever, do until you find anything else than an expressions
15     switch(CurrentToken) {
16       case PLUS:
17         left= left + term(true);
18         break;
19       case MINUS:
20         left= left - term(true);
21         break;
```

```
22        default:
23          return left;
24      }
25  }
26
27  double term(bool get){
28    double left=primary(get);
29    for(;;) // forever, do until you find anything else than a term
30      switch(CurrentToken) {
31        case MUL:
32          left= left * term(true);
33          break;
34        case DIV:
35          left= left / term(true);
36          break;
37        default:
38          return left;
39      }
40  }
41
42  jkmpNode* jkMathParser::primary(bool get){
43    jkmpNode* res;
44    if (get) getToken();
45    switch(CurrentToken) {
46      case NUMBER: {
47        getToken();
48        res= NumberValue //returned by Tokenizer;;
49        break;
50      }
51      case NAME: {
52         std::string varname=StringValue;
53        getToken();
54         if (CurrentToken == ASSIGN) { // assign a variable name
55            res=expression(true);
56            variables[varName]=res;
57         } else {
58           res=variables[varName];
59         }
60        break;
61      }
62      case MINUS:  // found primary:  - primary
63        res= -1.0 * primary(true);
64        break;
65      case LBRACKET: { // found primary ( expression )
66        double expr=mathExpression(true);
67        if (CurrentToken != RBRACKET) jkmpError("')' expected");
68        getToken(); // swallow ")"
69        res= expr;
70        break;
71      }
72      default:
73         jkmpError("primary expected");
74    }
75    return res;
76  }
```

Eine explizite Parse-Tabelle ist hier nicht nötig. Die Entscheidung, welche Regel anzuwenden ist erfolgt aufgrund des aktuellen Tokens (Lookahed!). Der Parameter `get` weist die entsprechende Funktion an, ein neues Token zu hohlen. Jedem Nicht-Terminal wird eine Prozedur zugeordnet, die die zugehörige Regel erkennt und entsprechend umsetzt. Man kann eine der Methode auch nochmals schematisch etwas anders darstellen, um einen besseren Überblick zu gewinnen:

```
1  double term(){
2    double left=primary();         // alle Regeln beginnen mit einem 'primary'
3    switch(CurrentToken) {         // primary hat bereits das nächste Token geliefert
4      case MUL:                    // Auswahl einer Regel
5        match('*');                // nächstes Token lesen und '*' bestätigen
6        left= left * term(true);
7          break;
8        case DIV:
9          left= left / term(true);
10         break;
11       default:
12         return left;
```

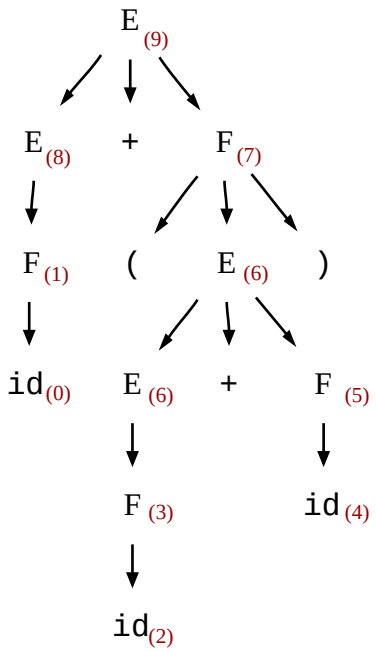
```
13          }
14    }
```

12.2 Bottom-Up-Syntaxanalyse

Wie am Anfang von Abschnitt 12.1 betrachtet man wieder die folgende Grammatik:

$$\begin{array}{lcl} E & \rightarrow & E \ \mathtt{+} \ F \mid F \\ F & \rightarrow & \mathtt{(} \ E \ \mathtt{)} \mid \mathtt{id} \end{array}$$

Nun wird wieder der Parsebaum zum Satz `id+(id+id)` nummeriert, und zwar in der Reihenfolge, wie die Knoten erzeugt werden:



Dies entspricht einer Rechtsableitung des Satzes:

$$\begin{aligned} E &\Rightarrow E + F \Rightarrow E + (E) \Rightarrow E + (E + F) \Rightarrow E + (E + \mathtt{id}\ ) \Rightarrow E + (F + \mathtt{id}\ ) \\ &\Rightarrow E + (\mathtt{id}\ + \mathtt{id}\ ) \Rightarrow F + (\mathtt{id}\ + \mathtt{id}\ ) \Rightarrow \mathtt{id}\ + (\mathtt{id}\ + \mathtt{id}\ ) \end{aligned}$$

12.2.1 Das Shift-Reduce-Prinzip

Alle Bottom-Up-Parser basieren auf dem Shift-Reduce-Prinzip. Dies bedeutet, dass sie einen Kellerautomaten simulieren, der zwei zentrale Operationen ausführt:

shift das nächste Eingabesymbol wird auf den Stack gelegt.

reduce oben auf dem Stack wird die rechte Seite einer Produktion erkannt (kann auch mehrere Stackzeilen umfassen!) und durch ihre linke Seite ersetzt.

Die Operation **shift** liest also sukzessive den Eingabestring von links nach rechts ab und schiebt ihn auf den Stack. Ist nicht zu reduzieren, so wird **shift** ausgeführt. Erkennt der Parser oben auf dem Stack eine rechte Seite einer Produktion, so führt er die Operation **reduce** aus. Dieses Vorgehen führt dazu, dass zuerst die elementarsten Produktionen erkannt werden (also die Blätter im Ableitungsbaum). Der Ableitungsbaum wird also von den Blättern her aufgebaut. Leider ist es nicht immer eindeutig, ob zu einem bestimmten Zeitpunkt **shift** oder **reduce** ausgeführt werden muss. Dies illustriert folgendes einfaches Beispiel:

Beispiel: Ablauf eines Shift-Reduce-Parsers: Man betrachte die folgende einfache Grammatik

$$\begin{array}{lcl} A & \rightarrow & A \texttt{ + } T \mid T \\ T & \rightarrow & T \texttt{ * } F \mid F \\ F & \rightarrow & \texttt{(} A \texttt{)} \mid \texttt{id} \end{array}$$

Mit dieser Grammatik soll das Wort `id * id` geparst werden. Die folgenden Tabelle zeigen zwei Möglichkeiten:

	Eingabe	Stack	Aktion
1	`id*id`	λ	shift
2	`*id`	`id`	reduce $F \rightarrow$ `id`
3	`*id`	F	reduce $T \rightarrow T$
4	`*id`	T	shift
5	`id`	T `*`	shift
6	λ	T `*id`	reduce $F \rightarrow$ `id`
7	λ	T `*` F	reduce $T \rightarrow T$ `*` F
8	λ	T	reduce $A \rightarrow T$
9	λ	A	acccept

	Eingabe	Stack	Aktion
1	`id*id`	λ	shift
2	`*id`	`id`	reduce $F \rightarrow$ `id`
3	`*id`	F	shift
4	`id`	F `*`	shift
5	λ	F `*id`	reduce $F \rightarrow$ `id`
6	λ	F `*` F	reduce $T \rightarrow F$
7	λ	F `*` F	reduce $A \rightarrow T$
8	λ	T `*` A	error

In Schritt 3 wurde also einmal reduziert und einmal geshiftet. Hier ist also nicht klar, welche Operation ausgeführt werden soll, da die Vorbedingungen für beide gegeben sind (**shift-reduce-Konflikt**). Sind an einer Stelle mehrere Regeln anwendbar, so liegt ein sog. **reduce-reduce-Konflikt** vor.

Die folgenden Abschnitte beschäftigen sich damit, wie die hier nicht-eindeutigen Entscheidungen getroffen werden können.

12.2.2 Handles und Präfixe

Handles: Sei $S \overset{*}{\underset{r}{\Rightarrow}} \beta X w \underset{r}{\Rightarrow} \beta\alpha w$ eine Rechtsableitung ($w \in T^*$, $\alpha, \beta \in (N \cup T)^*$) über einer Grammatik G. Im letzten Schritt wurde dabei die Regel $X \rightarrow \alpha$ angewendet. Es gilt dann:

1. α heißt **Handle** von $\beta\alpha w$. *Ein Handle bezeichnet damit eine Position in der Rechtsableitung, an der eine Regel angewendet wird, deren rechte Seite α ist. Es ist also in gewisser Weise ein Angriffspunkt für eine Reduktion beim Parsen.*
2. Ist G eindeutig, so ist das Handle in jedem Schritt eindeutig bestimmt. Es ist dann beim Parsen immer klar, welche Teilsymbolkette als nächstes mit welcher Regel reduziert werden muss.
3. Jedes Präfix von $\beta\alpha$ heißt **zuverlässiges Präfix** der Grammatik G. *Beim Parsen ist ein zuverlässiges Präfix ein Anfangsstück, dass sich nicht über das Handle hinaus erstreckt.*
4. $\beta\alpha$ heißt **maximales zuverlässiges Präfix** der Grammatik G. *Dies entspricht während der Ableitung gerade den Zeichen, die noch abzuleiten sind, da ja w bereits Terminal ist.*

Menge der Linkskontexte: Sei $A \in N$ ein Nicht-Terminal einer Grammatik G. Die **Menge der Linkskontexte** von A ist dann definiert als:

$$\text{left}(A) = \left\{\beta \middle| S \overset{*}{\underset{r}{\Rightarrow}} \beta A w, \;\; w \in T^*\right\}$$

Anschaulich ist dies die Menge aller Wörter über $(N \cup T)$, die links der Nicht-Terminals A stehen können, eben die Kontexte in denen A stehen kann. Es gilt offensichtlich $\text{left}(S) = \{\lambda\}$

Menge der maximal zuverlässigen Präfixe: Die Menge der maximal zuverlässigen Präfixe zu einer Produktion $A \to \alpha$ ist:

$$\mathrm{pre}(A \to \alpha) = \left\{\beta\alpha \middle| S \overset{*}{\underset{r}{\Rightarrow}} \beta X w \underset{r}{\Rightarrow} \beta\alpha w,\ \ w \in T^*\right\} = \mathrm{left}(A)\{\alpha\}$$

Im letzten Ausdruck werden die Sprachen konkateniert! Die Menge alle Präfixe einer Grammatik G ist so definiert:

$$\mathrm{pre}_G = \bigcup_{p \in P} \mathrm{pre}(p)$$

Automat zur Erkennung von Präfixen: Im folgenden soll gezeigt werden, dass man versuchen kann pre_G als regulären Ausdruck zu schreiben. Ist dies geschafft, so kann man nach Abschnitt 6.2 einen endlichen Automaten konstruieren, der die Präfixe erkennt.

Dazu betrachtet man zunächst eine Regel der Form $B \to \gamma A \delta$. Wie üblich sind $B, A \in N$ und $\gamma, \delta \in (N \cup T)^*$. Es gilt dann zunächst:

$$\mathrm{pre}(B \to \gamma A \delta) = \mathrm{left}(B)\{\gamma A \delta\}$$

Daraus folgt dann auch:

$$\mathrm{left}(A) \supseteq \mathrm{left}(B)\{\gamma\}.$$

Denn $\mathrm{left}(B)$ kann aufgrund der vorausgesetzten Regel sicher links von A stehen. Da in der Regel auch γ von A steht gilt obige Inklusionsbeziehung. Da noch andere Regeln vorhanden sein können, die A auf der rechten Seite enthalten, steht hier kein Gleichheitszeichen, sondern eine Inklusion. Man kann nun zu allen Produktionen der Grammatik Inklusionsbeziehungen herleiten. Da dieser Satz von Inklusionen vollständig ist, definiert er schließlich ein Gleichungssystem, dass man evtl. mit Hilfe von regulären Ausdrücken lösen kann. Ist dies gelungen, so lassen sich über $\mathrm{pre}(A \to \alpha) = \mathrm{left}(A)\{\alpha\}$ die maximalen Präfixe aller Produktionen berechnen und schließlich pre_G. Ist diese Gesamte Rechnung möglich (lösbar), so hat man nun pre_G in Form eines regulären Ausdruckes und kann einen entsprechenden Automaten konstruieren.

Beispiel:

	$\mathrm{left}(S) = \{\lambda\}$
$S \to A\$$	$\mathrm{left}(A) \supseteq \mathrm{left}(S)$
$A \to A+\ T$	$\mathrm{left}(T) \supseteq \mathrm{left}(A)\{A+\}$
$A \to T$	$\mathrm{left}(T) \supseteq \mathrm{left}(A)$
$T \to T*\ F$	$\mathrm{left}(F) \supseteq \mathrm{left}(T)\{T*\}$
$T \to F$	$\mathrm{left}(F) \supseteq \mathrm{left}(T)$
$F \to (\ A)$	$\mathrm{left}(A) \supseteq \mathrm{left}(F)\{(\}$

Daraus ergibt sich folgendes Gleichungssystem:

$$\begin{aligned}
\mathrm{left}(S) &= \{\lambda\} \\
\mathrm{left}(A) &= \mathrm{left}(S) \cup \mathrm{left}(F)\{(\} \\
\mathrm{left}(T) &= \mathrm{left}(A) \cup \mathrm{left}(A)\{A+\} \\
\mathrm{left}(F) &= \mathrm{left}(T) \cup \mathrm{left}(T)\{T*\}
\end{aligned}$$

Nun kann man diese Gleichungssystem lösen (mit regulären Ausdrücken, [...] ist eine Option):

$$\begin{aligned}
\text{left}(F) &= \text{left}(T) \cup \text{left}(T)\{T*\} = \text{left}(T)[T*] \\
\text{left}(T) &= \text{left}(A) \cup \text{left}(A)\{A+\} = \text{left}(A)[A+] \\
\text{left}(A) &= \text{left}(S) \cup \text{left}(F)\{\,(\} = \lambda|\,\text{left}(F)(= \lambda|\,\text{left}(T)[T*]\,(\, = \\
&= \lambda|\,\text{left}(A)[A+][T*]\,(\, = (\text{left}(A)[A+][T*]\,()^*
\end{aligned}$$

$$\begin{aligned}
\Rightarrow\ \text{left}(T) &= (\text{left}(A)[A+][T*]\,()^*[A+] \\
\Rightarrow\ \text{left}(F) &= (\text{left}(A)[A+][T*]\,()^*[A+][T*]
\end{aligned}$$

Man kann nun weiterrechnen und erhält schließlich durch Einsetzen einen regulären Ausdruck für pre_G.

12.2.3 Operator-Präzedenz-Parser

Zunächst soll eine Klasse von sehr einfachen Parser besprochen werden, die speziell für (arithmetische) Ausdrücke geeignet sind. Die zugehörige Grammatik ist von folgender Form:

$$E \quad \rightarrow \quad E\ \texttt{+}\ E \mid E\ \texttt{*}\ E \mid \texttt{-}\ E \mid \texttt{(}\ E\ \texttt{)} \mid \mid \texttt{id}$$

Um hier Regeln, wie Punkt-vor-Strich zu beachten muss man den Operator noch Prioritäten geben. So soll hier etwa `*` eine höhere Priorität haben, als `+` , damit Multiplikationen vor Additionen ausgeführt werden.

Operator-Prioriäten: Man betrachtet zwei Operatoren a und b. Es gelten zusätzlich folgende Konstruktionsregeln für die Prioritäten:

1. Hat a größere Priorität, als b, so setzt man $a \gg b$ und $b \ll a$.
2. Haben a und b gleiche Priorität, so setzt man $a \ll b$ und $b \ll a$ bei rechtsassoziativen und $a \gg b$ und $b \gg a$ bei linksassoziativen Operatoren.
3. Setze für alle Operatoren O:
   - $O \ll$ `id` , `id` $\gg O$. Ein Bezeichner soll also immer zuerst ausgewertet werden.
   - $O \ll$ `(` , `(` $\gg O$ Zunächst wird der Ausdruck in den Klammern ausgewertet.
   - $O \gg$ `$` , `$` $\ll O$ Das Ende-Symbol hat immer niedrigste Priorität
4. Es gilt außerdem:
   - `$` $\ll$ `id` , `id` $\gg$ `$` ; `$` $\ll$ `(` , `)` $\gg$ `$`
   - `(` $\equiv$ `)` , `(` $\ll$ `(` , `)` $\gg$ `)`
   - `(` $\ll$ `id` , `id` $\gg$ `)`
5. Für einen einstelligen Operator E, der kein zweistelliger ist (im Zweifel kann der Tokenizer zwei unterschiedliche Tokens z.B. für unären und binäres $-$ liefern) und einen beliebigen Operator gilt (O ist ein bel. Operator):
   - $O \ll E$
   - $E \gg O$, falls E höhere Priorität hat, wie O und $E \ll O$ sonst.

Diese beziehungen stellt man am besten in einer Tabelle dar. Für obuge Beispielgrammatik ergibt sich etwa:

Diese Prioritäten notiert man in einer Tabelle (in den Zeilen steht der linke Operator a, in den Spalten der rechte b; es wird die beziehung a...b abgelesen):

	+	*	-	(	)	id	$
+	$\gg$	$\ll$	$\ll$	$\ll$	$\gg$	$\ll$	$\gg$
*	$\gg$	$\gg$	$\ll$	$\ll$	$\gg$	$\ll$	$\gg$
-	$\gg$	$\gg$		$\ll$	$\gg$	$\ll$	$\gg$
(	$\gg$	$\gg$	$\gg$	$\ll$	$\equiv$	$\ll$	
)	$\gg$	$\gg$			$\gg$	$\gg$	$\gg$
id	$\gg$	$\gg$	$\gg$		$\gg$		$\gg$
$	$\ll$	$\ll$	$\ll$	$\ll$		$\ll$	

In dieser Tabelle fehlen die Einträge, die in einem Satz/Ausdruck auf keinen Fall vorkommen können. So darf etwa **id** nie direkt vor einer öffnenden Klammer stehen. Ebenso macht ein unäres **-** als letztes Zeichen keinen Sinn.

Operator-Präzedenz-Parser: Ein Parser hat nun folgenden Ablauf:

Algorithm 8 Operator-Präzedenz-Parser

1. Initialisiere den Stack mit **$**
2. Sei x das oberste Terminalsymbol im Stack und y das aktuelle Eingabezeichen.
   a) Falls $x \ll y$ oder $x \equiv y$, schreibe y in den Stackk und lies weiter.
   b) Falls $x \gg y$, so liegt ein Handle oben auf dem Stack. Entferne den handle und ersetze ihn durch die linke Seite der zugehörigen Produktion. Gibt es keine solche Regel,liegt ein Syntaxfehler vor.
   c) Gibt es keine Relation zwischen x und y, so liegt ebenfalls ein Syntaxfehler vor.
3. Falls x =$, y =$ und der Stack außer $ nur noch das Startsymbol enthält, so akzeptiere, sonst gehe zu 2.

Anschaulich kann kann man das so durchführen: zwischen allen Terminalen des abzuleitenden Satzes werden die entsprechenden Operator-Präzedenzen gesetzt (Nicht-terminale ignorieren!). Danach werden die Terminale innerhalb eines $\ll$...$\gg$-Paares reduziert. Mit der daraus entstehenden Satzform führt man genauso fort, bis man beim Axiom angelangt ist.

Beispiel: Hier wird der Satz **$ id + id * id $** geparst.

1. *anschauliche Methode:*
   a) **$** $\ll$ **id** $\gg$ **+** $\ll$ **id** $\gg$ ***** $\ll$ **id** $\gg$ **$**
   b) **$** $\ll$ E **+** $\ll$ E ***** E $\gg$ **$**
   c) **$** $\ll$ E **+** E $\gg$ **$**
   d) **$** E **$** also akzeptiere

2. *Simulation des Automaten:*

Eingabe	$y..x$	Stack
`id + id * id $`	$\gg$	`$`
`+ id * id $`	$\ll$	`$ id`
`+ id * id $`	$\gg$	`$` E
`id * id $`	$\gg$	`$` E`+`
`* id $`	$\ll$	`$` E`+ id`
`* id $`	$\gg$	`$` E`+` E
`id $`	$\gg$	`$` E`+` E`*`
`$`	$\ll$	`$` E`+` E`* id`
`$`	$\ll$	`$` E`+` E`*` E
`$`	$\ll$	`$` E`+` E
`$`	$\ll$	`$` E $\Rightarrow$ accept!

12.2.4 Charakteristische Automaten und LR(0)-Analyse

Hier werden Zustandskellerautomaten (siehe Abschnitt 6.1.2) verwendet, um bestimmte Grammatiken zu parsen. Die Bezeichnung LR(0)-Parser steht für „**l**inks nach rechts mit **R**echtsreduktionen, mit **0** (ohne) Lookahead“.

LR(0)-Item: Ein LR(0)-Item (eine LR(0)-Konfiguration) zur kontextfreien Grammatik $G = (N, T, S, P)$ ist ein Tripel (A, α, β), so dass $A \to \alpha\beta \in P$ und $\alpha, \beta \in (N \cup T)^*$. Die Items werden als $A \to \alpha.\beta$ oder $A \to [\alpha.\beta]$ geschrieben, wobei der Punkt die Trennung zwischen den zwei Anteilen angibt. Der Teil α heißt **Geschichte des Items**. Items der Form $A \to \alpha.$, also mit $\beta = \lambda$ heißen **vollständig** oder **erkennend**. Die Menge aller Items von G wird mit $\mathcal{I}_G$ bezeichnet. Das spezielle Item $S \to .S$`$` wird als **Start-Item** bezeichnet.

Anschaulich gibt ein Item an, an welcher Stelle in einer Produktion sich ein Parser gerade befindet. Dies wird nun als Zustand für einen Zustandskellerautomaten definiert:

Item-Kellerautomat: Der Item-Kellerautomat zur Grammatik G ist der Zustandskellerautomat $K_G = \big(T, \mathcal{I}_G, \delta, [S' \to .S\texttt{\$}\ ], \{[S' \to S\texttt{\$}\ .]\}\big)$ mit den Übergängen:

- **Expansionsübergang:** $\delta([X \to \beta.Y\gamma], \lambda) = \big\{[X \to \beta.Y\gamma][Y \to .\alpha] \big| Y \to \alpha \in P\big\}$ *Ein solcher Übergang liest kein Zeichen von der Eingabe und fügt ein neues Item auf den Stack hinzu. Dieses neue Item ist die Regel, in die das oberste Item expandiert wurde.*
- **Leseübergang:** $\delta([X \to \beta.a\gamma], a) = \{[X \to \beta.a\gamma]\}$, mit $a \in T$ *Hier wandert der Markierungspunkt des oberstn Items um ein Terminal weiter.*
- **Reduktionsübergang:** $\delta([X \to \beta.Y\gamma][Y \to \alpha.], \lambda) = \{[X \to \beta Y.\gamma]\}$, mit $a \in T$ *Dieser Übergang reduziert die zwei obersten Regeln. Das oberste Item wird dabei gelöscht und im nächsten wandert der Markierungspunkt um ein Nicht-Terminal weiter.*

Der Zustand des Automaten wird also durch den Inhalt des Stacks bestimmt. Der Zustand entspricht aber nicht nur dem obersten Item auf dem Stack, sondern mehreren Stackzeilen. Dieser Ansatz ist recht unanschaulich, sodass man eine etwas andere, aber äquivalente Darstellung benutzt. Dazu verwendet man einen endlichen Automaten (charakteristischer Automat), der einen Stack gewissermaßen als zusätzliches Gerät ansteuert. Der Zustand des Automaten wird aber intern gespeichert, nicht auf dem Stack. Die Datenhaltung ist dadurch gewissermaßen doppelt, dafür gewinnt man Übersichtlichkeit und einfache Implementierbarkeit.

charakteristischer Automat char_G**:** Der charakteristische endliche Automat char_G zu einer Grammatik G ist ein (nichtdeterministischer) endlicher Automat $\mathrm{char}_G = (N \cup T, \mathcal{I}_G, \delta, s_0, S_T)$ mit:

- Startzustand $s_0 = [S' \to .S\$\ ]$
- Menge der Endzustände $S_T = \{[X \to \alpha.] \big| X \to \alpha \in P\}$
- Übergangsfunktion: $\delta : \mathcal{I}_G \times (N \cup T \cup \{\lambda\}) \to 2^{\mathcal{I}_G}$, wobei:
  - $\delta([X \to \alpha.Y\beta], Y) = \{[X \to \alpha Y.\beta]\}$, für alle $X \to \alpha Y\beta \in P$ und $Y \in (N \cup T)$.
  - $\delta([X \to \alpha.Y\beta], \epsilon) = \{[X \to .\gamma] \big| Y \to \gamma \in P\}$, für alle $X \to \alpha Y\beta \in P$

Diese abstrakte Definition hat folgende Bedeutung:

- **Leseübergang:** Befindet sich ein solcher Automat im Zustand $[X \to \alpha.t\beta]$ ($t \in T$) und gibt es einen Übergang aus diesem Zustand mit dem aktuellen Symbol t, so wird t gelesen und der Punkt wandert in diesem Item eine Position nach rechts (neuer Zustand!). Der Zustand auf dem Stack muss natürlich upgedated werden.
- **Expansionsübergang:** Wird in einem Zustand $[X \to \alpha.Y\beta]$ ein λ-Übergang gewählt, so wird ein neues Item der Form $[Y \to .\gamma]$ auf den Keller geschoben.
- **Reduktionsübergang:** Gelangt char_G in einen Endzustand $[Y \to \gamma.]$, so wird das Kellersymbol $[Y \to \gamma.]$ entfernt und das neue oberste Symbol wird durch ein Item, entsprechend einem Übergang von δ mit Y ersetzt.
- **Akzeptieren:** Im Zustand $[S' \to S\$\ .]$ beendet sich der Automat akzeptierend.

Dieser Automat ist nicht deterministisch, sodass er praktisch ohne zusätzliche Information nicht einsetzbar ist. Deswegen wird im folgenden kurz eine Methode beschrieben, die einen deterministischen charakteristischen Automaten für eine Grammatik erstellt. Dieser heißt LR(0)-Parser. Dazu sind zunächst zwei Definitionen nötig:

LR(0)-Item-Abschluss: Der LR(0)-Item-Abschluss $\mathrm{cl}_0(S)$ einer Menge $S \subseteq \mathcal{I}_G$ von Items zu einer Grammatik G ist folgendermaßen iterativ definiert:

Algorithm 9 Konstruktion eines LR(0)-Item-Abschluss

1. $S' = S$
2. Wiederhole solange sich S' noch ändert
   a) Falls $[B \to \alpha.A\beta] \in S'$ ($A \in N$), dann für zu S' alle Items der Form $[A \to .\gamma]$ hinzu.
3. man erhält $\mathrm{cl}_0(S) = S'$

Der LR(0)-Item-Abschluss ist also die Menge aller Items, die von einem Item aus durch λ-Übergänge erreicht werden können.

LR(0)-Nachfolger-Menge: Die LR(0)-Nachfolger-Menge $\mathrm{succ}_0(S, X)$ für $S \subseteq \mathcal{I}_G$ und $X \in N \cup T$ ist definiert als

$$\mathrm{succ}_0(S, X) = \mathrm{cl}_0\Big(\{[A \to \beta X.\gamma] \big| A \to \beta.X\gamma \in S\}\Big)$$

Sie gibt also alle Ableitungsmöglichkeiten aus Items in S an, wenn X verarbeitet wird.

Nun kann man den LR(0)-Automaten konstruieren:

Algorithm 10 Konstruktion eines LR(0)-Automaten

1. setze $s_0 = \mathrm{cl}_0(\{[S' \to .S\$\ ]\})$ als Startzustand und setze $\{\}$ als Fehlerzustand (jeder Übergang auf die leere Menge zeigt einen Fehler an).
2. berechne die Nachfolgermenge $s_i = \mathrm{succ}_0(s_0, A)$ für alle $A \in N \cup T$. Die dabei entstehenden Mengen sind die Zustände des Automaten. füge jeweils Übergänge von s_0 nach s_i, unter Verarbeitung von A hinzu.
3. führe nun Schritt 2 nicht für s_0 aus, sondern für die neuen Mengen s_i und wiederhole, bis sich keine neuen Zustände mehr ergeben.

LR(0)-Automat: Aus dem Aufbau der Zustände kann man die auszuführende Operationen ersehen. Enthält s_i Items der folgenden Formen, so ist die zugeordnete Aktion auszuführen:

- $[X \to \alpha.t\beta], t \in T$: lese das Zeichen t ein (**shift**).
- $[X \to \alpha.]$: reduziere die Regel $X \to \alpha$ (**reduce**)
- $[S' \to S\$\ .]$: akzeptiere das Eingabewort **accept**.

Dies ordnet also jedem Zustand ein oder mehrere Operationen zu. Ergeben sich mehr als eine Operation pro Zustand, so liegen Konflikte vor:

- **reduce-reduce-Konflikt** bei konkurierenden Regeln
- **shift-reduce-Konflikt** falls sich beide Operationen ergeben

Die entsprechende Grammatik ist dann nicht LR(0) und kann nicht mit diesem Parser erkannt werden. Die obigen Operationen werden in der sog. LR(0)-Action-Tabelle zusammengefasst. Die Übergänge werden üblicherweise in der LR(0)-Goto-Tabelle notiert und gespeichert. Sind beide Tabelle bekannt, so ist die Information über den Automaten vollständig und er kann implementiert werden.

Erweiterung durch Lookup: Simple LR(1) = SLR(1): Man kann den LR(0)-Parser erweitern, indem man ein Lookup-Zeichen einführt. Dieses Lookup-Zeichen enthält jeweils das nächste Eingabesymbol. Hiermit können reduce-reduce-Konflikte aufgelöst werden, indem man eine Regel anhand des Lookups z auswählt. Man reduziert dann $[A \to \alpha.] \in s_i$ des aktuellen Zustandes genau dann, wenn $z \in \textsc{Follow}(A)$.

LR(1)-Parser: erweitert man die Items um ein Lookahead-Symbol, so kommt man zum komplexeren LR(1)-Parser, der aber potentiell sehr viele Zustände enthält. Verschmilzt man hier nun die Zustände, die sich nur im Lookahead unterscheiden, so ergibt sich ein sog. LALR(1)-Parser.

13 Syntaxgesteuerte Übersetzung

attributierte Grammatik, semantische Regel: Sei $G = (N, T; S; P)$ eine Typ-2-Grammatik (kontextfrei). G heißt **attributierte Grammatik**, wenn jedes Symbol $X \in (N \cup T)$ eine Menge von Attributen $\mathrm{attrib}(X)$ besitzt und den Produktionen **semantische Regeln** wie folgt zugeordnet werden: Ist $p = X_0 \to X_1 ... x_m, \quad X_i \in N \cup T$ eine Produktion, so ist ihr ein Menge $R(p)$ von semantischen Regeln der Form

$$X_i.a = f(X_j.a_j, ..., X_k.a_k)$$

zugeordnet, wobei die X_i in der Produktion Symbole und die a_j Attribute zu X_j sind. Für jedes Auftreten von X in einem Ableitungsbaum ist höchstens eine Regel anwendbar, um $X.a$ zu berechnen. Die Art der Attribute ist nicht näher spezifiziert. Diese können beliebige Informationen enthalten (sofern diese sinnvoll kodierbar sind).

syntaxgesteuerte Definition: Diese erweitern das Konzept der attributierten Grammatik. Es dürfen nun beliebige Anweisungen in den semantischen regeln auftauchen, z.B. `print(...)` zur Ausgabe.

synthetisierte und vererbte Attribute: Sei $b = f(c_1, ..., c_k)$ eine semantische Regel zu einer Produktion $X_0 \to X_1 X_2 ... X_m$. $b, c_1, ..., c_k$ sind Attribute der Symbole in der regel.

- Das Attribut b heißt **synthetisiert**, wenn es ein Attribut von X_0 ist.
- Das Attribut b heißt **vererbt**, wenn es ein Attribut eines der X_i der rechten Seite der Produktion ist.

synthetisierte Attribute hängen damit nur von den Attributen der Nachfolger des Knotens im Parsebaum ab. Vererbte Attribute hängen von seinen Geschwistern und Vorgängern ab.

S-attributiert Definition: Eine syntaxgesteuerte Definition heißt **S-attributiert**, wenn sie nur synthetisierte Attribute enthält.
Die Attribute eines S-attributierten Ableitungsbaumes können durch Berechnung von den Blättern zur Wurzel ausgewertet werden. Sie eignen sich daher gut für die Bottom-Up-Syntaxanalyse. Dabei kann dann u.U. sogar eine Auswertung erfolgen, ohne den Ableitungsbaum explizit zu berechnen und im Speicher anzulegen.

L-attributiert Definition: Eine syntaxgesteuerte Definition heißt **L-attributiert**, wenn jedes vererbte Attribut eines Symbols X_j auf der rechten Seite einer Produktion $X_0 \to X_1 ... X_m$ nur abhängig ist von den links von ihm stehenden Attributen, genauer:

1. von Attributen der Symbole $X_1, ..., X_{j-1}$
2. vererbten Attributen von X_0

Die Attribute eines L-attributierten Ableitungsbaumes können durch einen Tiefendurchlauf des Baumes berechnet werden (Postorder).

Beispiel (synthetisiert/vererbt): Man betrachte folgende Grammatik für Gleitkommazahlen. Die semantischen Regeln berechnen den wert der Gleitkommazahl:

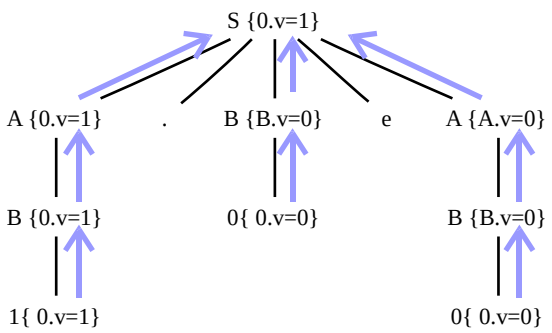
$$
\begin{array}{lcll}
S & \rightarrow & A_1 \ \mathbf{.} \ B \ \mathbf{e} \ A_2 & \{ S.v = A_1.z \times (A_1.v + B.v \div 10^{B.l}) \times 10^{A_2.z \times A_2.v} \} \\
A & \rightarrow & V \ B & \{ A.v = B.v; \quad A.z = V.z \} \\
V & \rightarrow & \mathbf{+} & \{ V.z = +1 \} \\
 & & \mid \mathbf{-} & \{ V.z = -1 \} \\
B & \rightarrow & B_1 \ \mathbf{ziffer} & \{ B.v = B_1.v \times 10 + \mathbf{ziffer.lexval} \ ; \quad B.l = B_1.l + 1 \} \\
 & & \mid \mathbf{ziffer} & \{ B.v = \mathbf{ziffer.lexval} \ ; \quad B.l = 1 \}
\end{array}
$$

Das Attribut **`ziffer.lexval`** enthält den Zahlwert einer Ziffer. Es handelt sich um eine Grammatik mit synthetisierten Attributen.
Die folgende Grammatik erlaubt die Deklaration von Typen:

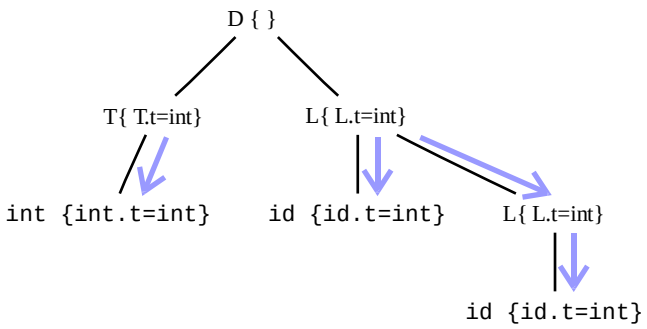
$$
\begin{array}{lcll}
D & \rightarrow & T \ L & \{ L.t = T.t \} \\
T & \rightarrow & \mathbf{int} & \{ T.t = \mathbf{int} \} \\
 & & \mid \mathbf{float} & \{ T.t = \mathbf{float} \} \\
L & \rightarrow & L_1 \ \mathbf{,} \ \mathbf{id} & \{ L_1.t = L.t; \quad \mathbf{id}.t = L.t \} \\
 & & \mid \mathbf{id} & \{ \mathbf{id}.t = L.t \}
\end{array}
$$

Die Attribute dieser Grammatik sind sämtlich vererbt. Die folgende Abbildung zeigt zwei Ableitungsbäume für einfache Ausdrücke der eben genannten Grammatiken:

synthetisierte Attribute
(Auswertung der Gleitkommazahl `1.0e0`)



vererbte Attribute
(Auswertung der Typdeklaration `int id, id`)



Die blauen Pfeile geben die Richtung der Zuweisungen bei der Auswertung der Attribute an.

Beispiel (Infix → Postfix): Das folgende Übersetzungsschema leistet die Übersetzung arithmetischer Ausdrücke von Infix- in Postfix-Notation:

$$
\begin{array}{lcl}
E & \rightarrow & T \ R \\
R & \rightarrow & \mathbf{+} \ T \ \{ \texttt{print("+")} \} \ R \\
 & & \mid \mathbf{-} \ T \ \{ \texttt{print("-")} \} \ R \\
 & & \mid \lambda \\
T & \rightarrow & \mathbf{num} \ \{ \texttt{print(num.value)} \}
\end{array}
$$

14 Semantische Analyse

Die semantische Analyse unterteilt sich in zwei Gebiete:

1. **statische Semantik:** Einige semantische Anforderungen können zur Compilzeit geprüft werden
   - typgerechte Verwendung von Bezeichnern
   - Gültigkeit von Bezeichnern
   - Zuordnung von `break`-Anweisungen zu den entsprechenden Schleifen und `case`-Anweisungen u.ä.
2. **dynamische Semantik:** Diese bedingungen können erst zur Laufzeit geprüft werden:
   - Arithmetische Ausnahmebedingungen, wie $1/0$, `NaN`, ...
   - Arraygrenzen-Überwachung (z.B. in Pascal, in C nicht implementiert!)
   - Wertebereichsverletzungen bei Zuweisungen

14.1 Typausdrücke

In vielen Programmiersprachen können Variablen mit einem gegebenen Typ deklariert werden. Dies schließt etwas verallgemeinert auch Funktionsdeklarationen ein. Man kann solche Typ-Beziehungen über sog. Typausdrücke darstellen.

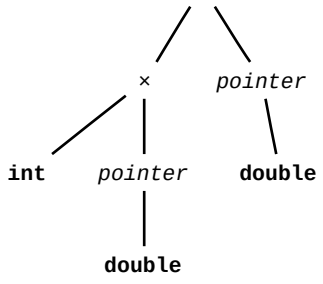
Typausdrücke: Ein Typausdruck ist:

1. **einfache Typen**, wie `int`, `double`, `void`, ... und der spezielle Typ `typeError` (um Fehler anzuzeigen).
2. der Name eines **selbstdefinierten Typs**
3. **Konstruktionen**:
   - *Arrays*: Ist T ein Typausdruck und I eine Indexmenge, so ist $\mathrm{array}(I, T)$ ein Typausdruck
   - *Produkte*: Sind T_1, T_2 Typausdrücke, so ist auch das kartesische Produkt $T_1 \times T_2$ ein Typausdruck.
   - *Zeiger (Pointer)*: Ist T ein Typausdruck, so ist auch $\mathrm{pointer}(T)$ ein Typausdruck, der einen Zeiger auf T darstellt.
   - *Records*: Die Felder des records haben die Bezeichner $\mathtt{id}_i$ und die Typen T_i. Dann ist auch $\mathrm{record}(\mathtt{id}_1 \times T_1, ..., \mathtt{id}_n \times T_n)$ ein
   - *Funktionen*: Ist D ein Typausdruck, der den Definitionsbereich einer Funktion bescchreibt und R ein Typausdruck, der ihren Wertebereich beschreibt, so ist $D \Rightarrow R$ ein Typausdruck.

Beispiel: Die Funktionsdeklaration `int* f(int a, double* b)` hat den Typ

$$(\texttt{int} \times \text{pointer}(\texttt{double}) \Rightarrow \text{pointer}(\texttt{int}))$$

Dies lässt sich auch als Baum darstellen:



14.2 Symboltabelle

Die Symboltabelle wurde in Abschnitt 10 bereits eingeführt. Sie ist die zentrale Datenstruktur des Compilers und verwaltet alle definierten Symbole (Variablen, Funktionen, ...) eines Programmes. Während der semantischen Analyse werden mit ihr folgende Operationen ausgeführt:

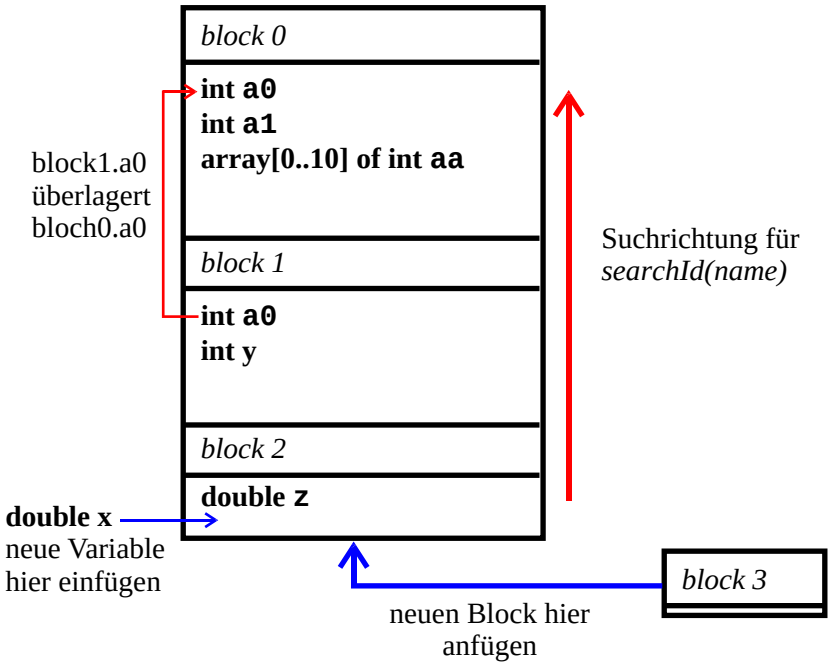
- Speicherung von Attributen der Bezeichner (Typ, Größe etz.)
- Analyse der Definition von Bezeichnern
- Prüfen auf Korrektheit der Definitionen
- Behandlung externen Deklarationen

Definition: Die Symboltabelle ist eine Datenstruktur, auf der eine spezielle Menge von Operationen definiert sind:

- *create()*: Anlegen einer neuen, leeren Tabelle
- *delete()*: Auflösen der Tabelle
- *enterBlock()*: Betreten eines neuen Gültigkeitsbereiches
- *exitBlock()*: verlassen des aktuellen Gültigkeitsbereiches
- *enterID(name, info)*: einen Bezeichner in den aktuellen Gültigkeitsbereich einfügen (mit Information)
- *searchId(name)*: sucht nach einem Bezeichner und gibt die Informationen zu diesem Bezeichner zurück (oder einen Fehler).

Die Symboltabelle ist also in Blöcke gegliedert, die die verschiedenen Gültigkeitsbereiche im Programm wiederspiegeln.

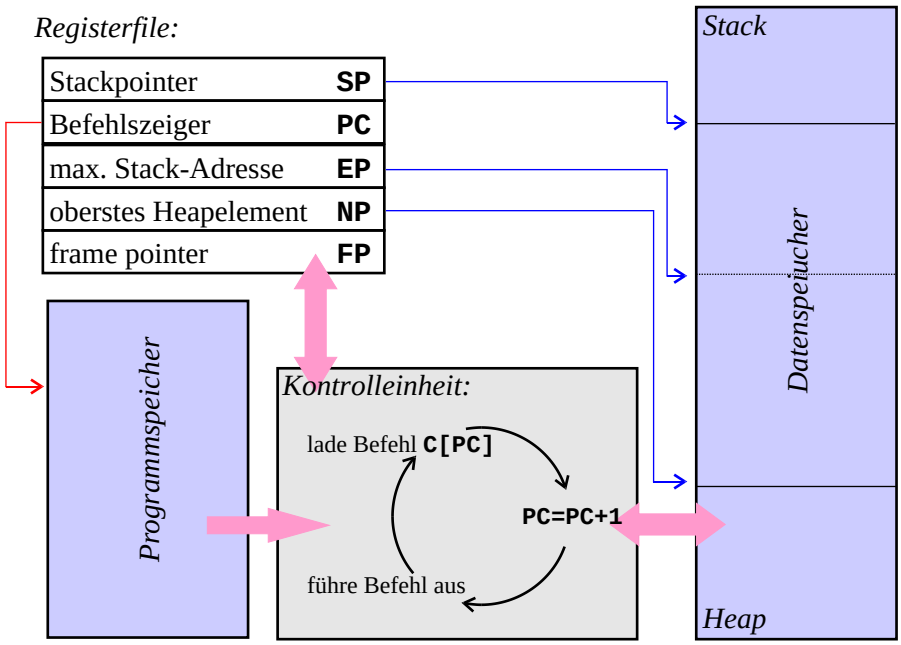
Die folgende Abbildung veranschaulicht das Prinzip


block 0
int a0
int a1
array[0..10] of int aa
block1.a0
überlagert
bloch0.a0
block 1
int a0
int y
Suchrichtung für
searchId(name)
block 2
double z
double x
neue Variable
hier einfügen
block 3
neuen Block hier
anfügen

15 Zwischencodeerzeugung

Wie bereits in Kapitel 10 eingeführt, wird er Programmquelltext zunächst in einen Zwischencode übersetzt. Dieser ist noch Maschinenunabhängig, aber näher an realen Prozessoren, wie eine Hochsprache. Hier sollen nur kurz die Prinzipien einiger Zwischensprachen erläutert werden:

15.1 C-Maschine



L-Wert: Der L-Wert $\texttt{code}_L\ v\ \varphi$ einer Variable v in der Speicherumgebung φ bezeichnet die Adresse von v.

R-Wert: Der R-Wert $\texttt{code}_R\ v\ \varphi$ einer Variable v in der Speicherumgebung φ bezeichnet den Wert der Variable v.

16 3-Adress-Code

Dieser Zwischencode liegt sehr nahe an Maschinenbefehlen. Jede Instruktion darf maximal auf drei SPeicheradresse zugreifen. Beispiele sind:

- `x := y op z`: Führe binäre Operation mit y und z aus und speichere das Ergebnis in x.
- `x := uop z`: Führe unäre Operation mit z aus und speichere das Ergebnis in x.
- `x := y`: einfache Zuweisung
- `x := y[i]` / `x[i] := y`: Array-Zugriffe
- `if (x cop y) goto label`: vergleiche x und y mit Vergleichsoperation `cop` und springe falls true zu Marke `label`
- `goto label`: Sprung zur Marke `label`
- `noop`: keine Operation
- usw.

Beispiel: Die folgende while-Schleife wird übersetzt:

$$\textbf{while } (a \textbf{ and } b) \textbf{ do } c := d + e \textbf{ end}$$

3-Adress-Code zu dieser Schleife:

```
L1: noop                  // Überbleibsel des Compilers
    if (a==0) goto L3     // Schleife beenden
    goto L4               // a=true => prüfe b (bei L4)
L4: noop
    if (b==0) goto L3     // Schleife beenden
    goto L2
L2: noop
    c := d + e            // Schleife ausführen
    goto L1
L3: noop
```

A Glossar

$|x|$**:** Anzahl der Buchstaben im Wort W

$\sharp_a(w)$**:** Anzahl der Vorkommen des Buchstaben a im Wort w

$\mathrm{bin}(n)$**:** Binärdarstellung der Zahl $n \in \mathbb{N}$

λ**:** leeres Wort einer Sprache

$A \times B$**:** kartesisches produkt zweier Mengen A und B, also: $A \times B = \{(a,b) \big| a \in A, b \in B\}$

G_f**:** Graph der Funktion f, also: $G_f = \{(x, f(x)) \big| x \in \mathrm{Db}(f)\}$

Σ**:** ein Alphabet, also eine Menge von Buchstaben

Σ^***:** Die Menge aller Wörter über Σ

$\Sigma^+ = \Sigma^* - \{\lambda\}$

$\Sigma^{=n} \equiv \Sigma^n$**:** Menge aller Wörter über Σ der Länge n.

$\leq_{\textbf{LEX}}$**:** Ordnungsrelation der lexikographischen Ordnung

$\leq_{\textbf{ll}}$**:** Ordnungsrelation der längen-lexikographischen Ordnung

w^R**:** Spiegelwort zum Wort w

$\mathrm{res}_A(x)$**:** Resultatsfunktion eines Algorithmujs A

$\uparrow$**:** undefiniert

$\downarrow$**:** definiert

$\mathrm{Db}(f)$**:** Definitionsbereich einer Funktion f

$\mathrm{Wb}(f)$**:** Wertebereich einer Funktion f

L_A**:** Menge/Sprache, der vom Algorithmus A akzeptierten/aufgezählten Wörter

$\overline{L}$**:** Komplement der Menge/Sprache L

c_M**:** charakteristische Funktion der Menge M

χ_M**:** partielle charakteristische Funktion der Menge M

KON: Menge von Konfigurationen

$U = (\textbf{KON}, \rightarrow)$**:** Umformungssystem

$c \xrightarrow{n} c'$**:** endliche Umformung von c nach c' in n Schritten

$c \xrightarrow{*} c'$**:** endliche Umformung von c nach c' in endlich vielen Schritten

$\mathrm{In}(x)$**:** Eingabefunktion $\mathrm{In} : I \mapsto \textsc{Kon}$

$\mathrm{Out}(c)$**:** Ausgabefunktion $\mathrm{Out} : \textsc{Kon} \mapsto O$

$R = (I, O, \textbf{KON}, \mathrm{In}, \mathrm{Out}, \rightarrow)$**:** Rechensystem

$R(M)$**:** Rechensystem zu einer mathematischen Maschine M

RES_R**:** Resultatsrelation des Rechensystems R

OPER: Menge elementarer (Speicher-)Operationen

TEST: Menge von Testoperationen

$M = (B, P)$**:** mathematische Maschine

$\mathrm{TB}(\Sigma, T, \Gamma, n)$**:** Basis-Turingmaschine

$\mathrm{BI} = \left\{ f : \mathbb{Z} \to \Gamma \middle| f(z) = b \right.$ **für fast alle** $\left. z \right\}$ **(bei Turing-Maschinen):** Menge der Zuordnungsvorschriften, die einer Adresse $z \in \mathbb{Z}$ eine Beschriftung aus dem Bandalphabet Γ zuordnet, also im Prinzip die Menge der möglichen Beschriftungen eines Turing-Bandes.

$F(\mathrm{TM})$**:** Klasse der Turing-berechenbaren Funktionen

$F(\mathrm{TO})$**:** Klasse der mit Turing-Operatoren berechenbaren Funktionen

$\underline{n}, \quad n \in \mathbb{N}$**:** Unärdarstellung der Zahl n

$\underline{\vec{n}} = \underline{n_1} b \underline{n_2} b ... b \underline{n_k}, \quad \vec{n} \in \mathbb{N}^k$**:** Unärdarstellung des Vektors $\vec{n}$ mit dem Trennzeichen 'b'.

$\mathrm{Iter}(f, n)(s) := f(f(...f(s))) = f^n(s)$**:** n-fache iteration der FUnktion f

$\mathrm{Iter}_t(f)(s)$**:** bedingte Iteration (while-Schleife) der FUnktion f mit dem Test t.

:

:

:

Index

$\sharp_a(w)$, 14
EA, 60
KON, 34
Σ, 14
$|x|$, 14
λEA, 60
λ-Automat, 60
λ-Übergänge, 60
λ-frei, 24, 29
λ-treu, 24
Äquivalenz von mathematische Maschinen, 37
Äquivalenz vonBasismaschinen, 37
Äquivalenzproblem, 26
Übergangsfunktion, 60, 61
Übersetzer, 96
$-Konvention, 103
2-3-4-Bäume, 90
3-Adress-Code, 123
3AC, 123
3Dnow, 98

Ableitung, 16
Ableitungsbaum, 18
Ableitungsproblem, 104
Action-Tabelle, 116
Addition
    unäre, 36, 39, 42, 44
Adjazenzmatrix, 73
Algorithmen
    Binäre Suche, 86
    Heapsort, 85
    Suchen, 86
Algorithmus, 32
    Baum-Isomorphie, 77
    Baumtiefe, 76
    Bubble-Sort, 81
    Eigenschaften, 32
    Elemente in Baum zählen, 76
    Hybrid-Quicksort, 83
    Insertion-Sort, 81
    inuitive Definition, 32
    Kompression, 77
    Lempel-Ziv-Welch, 78
    Mergesort, 83
    QUICK-FIND, 75
    QUICK-UNION, 75
    Quicksort, 82
    Selection-Sort, 80
    ShellSort, 81
    Sortierverfahren, 80
    Traversieren von Binärbäumen, 76
    Union-Find, 75
    WEIGHTED-QUICK-UNION, 75
Alphabet, 14
Analyse-Problem, 102
Array, 119
Attribut, 117
attributierte Grammatik, 117
Aufzählbarkeit, 25, 33
    Turing-aufzählbar, 40
Aufzählungsfunktion, 33
Aufzählungsverfahen, 33
Ausgabefunktion, 61
Automat, 60
Automat mit Ausgabe, 61
Automaten, 99

B-berechenbar, 37
B-Programm, 35
Backend, 98
Basismaschine, mathematische, 35
Basismaschinen-Programm, 35
Baum, 68, 76
    2-3-4-Suchbaum, 90
    binärer Suchbaum, 86
    digital search trie, 92
    digitaler Suchbaum, 92
    Durchläufe, 69
    Ebene, 68
    Eigenschaften, 68
    Elemenete zählen, 76
    Höhe, 68
    induktive Definition, 68
    n-ärer Baum, 68
    Pfad, 68
    Pfadlänge, 68
    Rot-Schwarz-Suchbaum, 91
    Terminologie, 68
    Tiefe bestimmen, 76
    voller Binärbaum, 68
    vollständig Binärbaum, 68
Baum, binomialer, 71
Baum-Isomorphie, 77
bedingte Instruktion
    mathematische Maschinen, 37
Berechenbarkeit, 32
    B-berechenbar, 37
    Turing-berechenbar, 40
Berechnungsverfahren, 32
Binäraddition, 42
Binäre Suche, 86
Binärer Suchbaum, 86
Binomialer Baum, 71
Binomialer Heap, 71

Bottom-Up-Syntaxanalyse, 109
Bubble-Sort, 81
Buchstabe, 14

C++-Präprozessor, 97
C-Maschine, 122
charakteristischer Automat, 115
Chomsky, Noam, 15
Chomsky-Grammatik, 16
Chomsky-Hierarchiesatz, 21
Church-Turing-These, 40
Compiler, 96
Comsky-Normalform, 21
Cross-Compiler, 98
CYK-Algorithmus, 26

dangling else, 23
Definitionsbreich
    Aufzählbarkeit, 34
deterministischer Automat, 60
deterministischer, linear beschränkter Automat (NLBA), 60
Digital Search Trie, 92
Digitale Suchbäume, 92
Double Hashing, 94
dynamische Semantik, 119

EBNF, 17
echte Satzform, 17
echter Präfix, 15
echter Suffix, 15
echtes Teilwort, 15
eindeutige Grammatik, 24
einfach LL(1), 105
einfache Typen, 119
Endlicher Automat (EA), 60
Entropie, 66
Entscheidbarkeit, 32
    Turing-entscheidbar, 40
Entscheidungsverfahren, (totales/partielles), 32
Ergebnisfunktion
    Turing-Operatoren, 41
Ergebnisrelation, 35
erkennendes Item, 114
Erweiterte Backus-Naur-Form (EBNF), 17
Expansionsübergang, 114

Folgemenge, 106
Folgemengen-Abbildung, 104
Frontend, 98
Funktionen, 119

Gödelisierung, 9
GCC, 98
Geschichte des Items, 114
Gleitkommazahl, 118
GNU Compiler Collection, 98
Goto-Tabelle, 116
Graph, 73
Graph einer Funktion, 8

Halbband-Turingmaschine, 43
Handle, 110
Hash-Funktion, 93
Hashing, 93
    Double Hashing, 94
    Linear Probing, 94
Heap, 71, 85
Heapsort, 85
Herleitung, 16
Hierarchiesatz, 21
Huffman Kodierung, 77
Hybrid-Quicksort, 83

Informationsgehalt, 66
Informationstheorie, 66
inhärent mehrdeutige Sprache, 24
Inorder, 69
Insertion-Sort, 81
Instruktion
    bedingte (mathematische Maschinen), 37
    Operationsinstruktion, 36
    Testinstruktion, 36
Interpreter, 96
irregulärer Endzustand, 60
Item, 114
Item-Kellerautomat, 114

Kellerautomat, 62, 106, 109
Kellerspeicher, 61
Klammerausdruck, 22
Kompression, 77
Kompressions-Algorithmus
    Lempel-Ziv-Welch, 78
Konfiguration, 34, 114
Konstruktionen, 119
kontextsensitive Grammatik, 27
Kopieroperator, 42

L-attributiert Definition, 117
L-Wert, 122
Löschperator, 42
LALR(1)-Parser, 116
Landau-Notation, 9
leeres Wort, 14
Leerheitsproblem, 26
Lempel-Ziv-Welch-Kompression, 78
Leseübergang, 114
Lexem, 99
Lexikalische Analyse, 99
Linear Probing, 94
Linksableitung, 103
Linksbleitung, 16
Linksfaktorisierung, 30
Linkskontext, 110
Linksoperator, 42
linksrekursiv, 25, 29
linksrekursive Grammatik, 25
LL(1)-Grammatik, 105
LL(1)-Parsetabelle, 106
Lookup, 116
LR(0)-Item, 114
LR(0)-Item-Abschluss, 115

LR(0)-Konfiguration, 114
LR(0)-Nachfolger-Menge, 115
LR(0)-Parser, 114, 115
LR(1)-Parser, 116

Maschine
    Basismaschine, mathematische, 35
    Basismaschinen-Programm, 35
    mathematische, 34, 35
    Turingmaschine, 37
mathematische Maschine, 34, 35
maximales zuverlässiges Präfix, 110
Mehrband-Turingmaschine, 43
mehrdeutige Grammatik, 24
mehrdeutige Sprache, 24
Mergesort, 83
Meta-Zeichen, 19
MMX, 98

Nachricht, 66
nicht-rekursive Chomsky-Sprachen, 25

Operator-Präzedenz-Parser, 112, 113
Operator-Prioriät, 112
Ordnung, 14
    längen-lexikographische, 14
    lexikographische, 14

Paarfunktion, 9
Parsebaum, 18
Parser, 30
Parsetabelle, 106
Pfadkompression, 76
Pointer, 119
Postorder, 69
Potentzmenge, 8
Präfix, 15
Präfixcode, 78
Preorder, 69
Priorität, 112
Produktion, 15
Pumping-Lemma, 25
push down automaton (PDA), 62

QUICK-FIND-Algorithmus, 75
QUICK-UNION-Algorithmus, 75
Quicksort, 82

R-Wert, 122
Rechensystem, 35
Rechtsableitung, 16, 109
Rechtsoperator, 42
Record, 119
reduce-reduce-Konflikt, 110, 116
reduce: shift-reduce-Prinzip, 109
Reduktionsübergang, 114
reduzierte Grammatik, 24
Registerbelegung, 98
reguläre Menge, 19
regulärer Ausdruck, 19, 64
rekursiver Abstieg, 107
Relation, 8
Resultatsfunktion
    mathematische Maschine, 37
Rot-Schwarz-Baum, 91

S-attributiert Definition, 117
Säparierte Grammatik, 21
Satz, 17
Satzform, 17
Selection-Sort, 80
Semantische Analyse, 119
semantische Regel, 117
Shannon-Informationstheorie, 66
ShellSort, 81
shift-reduce-Konflikt, 110, 116
shift-reduce-Prinzip, 109
Simple LR(1) = SLR(1), 116
Sortieren, 80
Sprache, 15
    Iteration, 15
    Konkatenation, 15
    Verkettung, 15
SSE, 98
Stack, 61
Start-Item, 114
Startmenge, 106
Startmengen-Abbildung, 104
statische Semantik, 119
Stelligkeit von Funktionen, 8
Substitutionsregel, 21
Suchbaum, binärer, 86
Suchen, 86
Suffix, 15
Symboltabelle, 120
Syntaxanalyse, 102
Syntaxdiagramm, 17
Syntaxgesteuerte Übersetzung, 117
syntaxgesteuerte Definition, 117
synthetisiertes Attribut, 117

T-Diagrammen, 96
Teilwort, 15
Termersetzungssystem, 16
Terminalwort, 17
Token, 99
Tokenizer, 100
Top-Down-Parser, 29
Top-Down-Syntaxanalyse, 103
Transportperator, 42
Traversieren von Bäumen, 76
Turing-aufzählbar, 40
Turing-Berechenbarkeit, 40
Turing-entscheidbar, 40
Turing-Operator, 41
    Kopieroperator, 42
    Löschoperator, 42
    Linksoperator, 42
    Rechtsoperator, 42
    Transportoperator, 42
    unäre Addition, 42
Turing-These, 40

Turingmaschine, 25, 37
    Halbband-TM, 43
    Mehrband-TM, 43
Typausdruck, 119

U-Rechnung
    endliche, 34
    maximale, 34
Umformungsregel, 21
Umformungssystem, 16, 34
    deterministisch, 34
Unäraddition, 42
Unärdarstellung, 11
unäre Addition, 36, 39, 44
Unendlichkeitsproblem, 26
unerreichbares Symbol, 24, 27
Union-Find-Algorithmus, 75
universelle Funktion, 11
unproduktives Symbol, 24, 27

vererbtes Attribut, 117
vollständiges Item, 114

Wörter
    Iteration, 15
    Konkatenation, 15
    Ordnung, 14
    Spiegelwort, 15
WEIGHTED-QUICK-UNION-Algorithmus, 75
Wildcard, 19
Wortproblem, 102

Zahlendarstellungen, 11
Zeiger, 119
Zustandskellerautomat, 62
Zustandskellerautomaten, 114
Zustandstabelle, 61
zuverlässiges Präfix, 110
Zwischencode, 122
zykelfrei, 25
zykelfreie Grammatik, 25