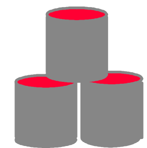
# Grundlagen der Informatik
## – Theoretische Informatik –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



# Inhalt

- Formale Sprachen und Automaten
- Berechenbarkeit und Entscheidbarkeit
- Korrektheit von Algorithmen

# Formale Sprachen und Automaten

- Wiederholung einiger Grundlagen:
  - **Alphabet A** ist eine endliche Menge von Symbolen (Zeichen)
  - **Wort** über dem Alphabet A: endliche Folge von Symbolen aus A. Leeres Wort ε (besteht aus keinem Zeichen) ist ein besonderes Wort. Die Menge aller Wörter über einem Alphabet bezeichnen wir als A*.
  - (**Formale**) **Sprache** L: Menge von Wörtern über Alphabet A, welche insbesondere das leere Wort ε enthält: $L \subseteq A^*$
- Auch die leere Menge L = { } ist eine Sprache, ebenso A*. Leere Menge L = { } nicht verwechseln mit L = {ε}!

# Formale Sprachen und Automaten

- Um komplizierte Sprachen aus einfacheren aufbauen zu können, werden die folgenden Operationen definiert
  (L und K sind Sprachen über den gemeinsamen Alphabet A)
  - $L \cdot K = \{uv \mid u \in L, v \in K\}$, das **Produkt** von L und K
  - $L^0 = \{\varepsilon\}$, $L^{n+1} = L \cdot L^n$, die **Potenzen** von L
  - $L^* = \cup \{ L^n \mid n \in \mathbb{N}_0\}$, der **Kleene-Stern** von L

# Reguläre Ausdrücke

- Sei A ein Alphabet
  - ⇨ $\emptyset$ und $\varepsilon$ sind reguläre Ausdrücke.
  - ⇨ a ist ein regulärer Ausdruck für jedes $a \in A$
  - ⇨ Sind e und f reguläre Ausdrücke, dann auch $e + f$, $e \cdot f$ und $e^*$
- Zur eindeutigen Darstellung sind Klammern „(“ und „)“ erlaubt.
  - ⇨ Zur Klammerpaarung wird vereinbart, dass * stärker bindet als · , und · stärker als +

# Lexikalische Analyse

- Verwendung regulärer Ausdrücke zur Festlegung des lexikalischen Anteil von Programmiersprachen (und somit Realisierung der ersten Stufe des Compilers)
  - ⇨ Z.B. werden Token wie Bezeichner oder Anweisungen dadurch eindeutig beschrieben
  - ⇨ Bekannte Werkzeuge aus der C-Programmierwelt: lex bzw. flex.

# Wie funktionieren Werkzeuge wie lex?

- Wie kann ein Programm erstellt werden, das zu einem beliebigen regulären Ausdruck e und einem beliebigen Text s erkennt, ob $s \in L(e)$ ist,
  - oder das alle Textstellen findet, die auf den regulären Ausdruck passen?

- Lösung dazu liefern endliche Automaten mit deren Hilfe bestimmte Sprachen erkannt werden können

# Automaten und ihre Sprachen

- Automat: sehr einfaches Modell einer zustandsorientierten Maschine, die man dazu benutzen kann, Wörter zu akzeptieren oder zurückzuweisen
  - ⇨ Im Prinzip als „Kasten“ vorstellbar, der eine Eingabe erhält und dadurch seinen internen Zustand ändert

# Automaten und ihre Sprachen

- Theorie unseres Automaten für lex
  - Eingabe: Zeichen aus unserem Alphabet A
  - Automat M kann eine Menge S von möglichen Zuständen einnehmen; Teilmennge $F \subseteq S$ bezeichnet die Menge der finalen bzw. akzeptierenden Zuständen
  - Mit Reset-Taste kann man ihn in wohldefinierten Ausgangszustand $s_0$ versetzen
  - Anschließende Eingabe eines Wortes $w \in A^*$ über dem Alphabet A
  - Jedes eingegebene Zeichen kann den Automaten M in einen neuen Zustand versetzen
  - Falls der letzte Zustande (nach vollständiger Eingabe des Wortes w) ein finaler Zustand ist, wird das Wort anerkannt, andernfalls zurückgewiesen

# Automaten und ihre Sprachen

- Insgesamt kann damit ein Automat (besser: eine Maschine) wie folgt definiert werden:
  - ⇨ $\mathbf{M = (S, A, \delta, s_0, F)}$
  - ⇨ $\delta : S \times A \longrightarrow S$ bezeichnet die Übergangsfunktion, welche den Folgezustand in Abhängigkeit der zeichenweise Eingabe beschreibt

# Darstellung von Automaten

- Automaten kann man so veranschaulichen:
  - ⇨ Jeder Zustand s wird durch einen Kreis dargestellt
  - ⇨ Jeder Übergang $\delta(s, a) = s'$ wird durch einen Pfeil von s nach s' dargestellt, welcher mit a beschriftet wird

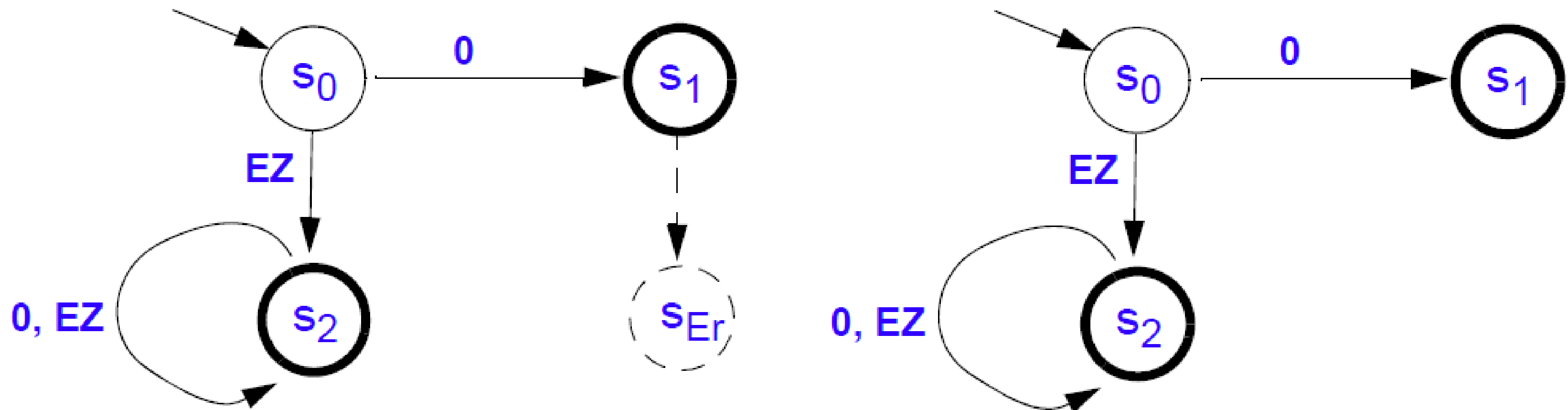
- Zwischen zwei Zuständen malt man höchstens einen Pfeil und beschriftet ihn mit allen Zeichen aus dem Alphabet, welche diesen Zustandsübergang hervorrufen

# Darstellung von Automaten

- Auf den Anfangszustand zeigt ein Pfeil, der aus dem Nichts kommt
- Akzeptierende Zustände werden durch eine dickere Umrandung gekennzeichnet
- Für jedes Zeichen a muss aus jedem Zustand ein Pfeil ausgehen, der mit a beschriftet ist.
- Um sie immer zu erfüllen, kann man einen Error-Zustand $s_{Error}$ hinzunehmen, zu dem alle fehlenden Pfeile gerichtet sind

# Beispiel: L ($IN_0$)

- L($IN_0$): Menge der natürlichen Zahlen inkl. 0, bzw. konkret:
  - Entweder 0,
  - Einstellige Zahl ungleich 0 (EZ), oder
  - Mehrstellige Zahlen, die nicht mit 0 beginnen
- Darstellung:



# Automaten und Grammatiken

- Wie hängen Grammatiken und Automaten zusammen?
- Dazu wird zunächst eine formale Definition benötigt, welche die von einem Automaten erkannte Sprache beschreibt.
  - ⇨ Dazu muss vorher die Wirkung der Übergangsfunktion auf ein ganzes Wort sinnvoll erweitert werden:
    - $\delta^*(s, \varepsilon) = s$
    - $\delta^*(s, aw) = \delta^*(\delta(s,a), w)$ für jedes Wort w und Zeichen a.
  - ⇨ Für den Automaten **M = (S, A,** δ **,s0 , F)** heißt
    $L(M) = \{w \in A^* \mid \delta^*(s0,w) \in F\}$ die Sprache des Automaten M

# Automaten und Grammatiken

- Menge der regulären Sprachen ≙ Menge der Sprachen, die durch einen endlichen Automaten erkannt werden können, d.h.
  - ⇨ Zu jeder regulären Sprache kann ein Automat konstruiert werden, welcher diese Sprache erkennt/konstruiert und
  - ⇨ umgekehrt, d.h. jede von einem Automaten erkannte Sprache ist regulär.
- Insbesondere ist die Mächtigkeit der deterministischen und nicht-deterministischen Automaten identisch

# Formale Grammatik

- Chomsky definierte 1959 ein Regelwerk zur Beschreibung einer formalen Sprachhierarchie.
  Dazu wird der Begriff der Grammatik zusammen mit einem Ableitungsbegriff eingeführt

- In der allgemeinsten Form ist eine Chomsky-Grammatik ein 4er-Tupel **G = (N, T, P, S}** und ist wie folgt definiert:
  - ⇨ N: endliche, nichtleere Menge von nonterminalen (Symbolen),
  - ⇨ T: endliche, nichtleere Menge von Terminalen (Symbolen) mit $T \cap N = \{ \}$.
  - ⇨ $S \in N$ ist das Startsymbol.
  - ⇨ P: endliche Menge von Regeln der Form (p, q) mit $p, q \in (T \cup N)^*$
    wobei p mindestens ein nonterminales Symbol enthält.
    Eine Regel (p, q) wird auch Produktion genannt.

# Formale Grammatik

- Die von G erzeugte Sprache L(G) definiert die Menge aller gültigen Wörter, die aus dem Startsymbol ableitbar sind:
  - ⇨ $L(G) := \{w \mid S \rightarrow_G w \text{ und } w \in T^*\}$

# Chomsky-Hierarchie

- Jede Grammatik ist zunächst automatisch vom **Typ 0 (allgemein)**.
- Grammatik ist vom **Typ 1** (**kontextsensitiv**), wenn gegenüber Typ 0 einschränkend für alle Regeln $u \rightarrow v$ gilt:
  - ⇨ Worte auf der rechten Seite einer Regel sind mindestens so lang wie die auf der linken Seite, d.h. $|u| \leq |v|$.
- Eine Grammatik ist vom **Typ 2 (kontextfrei**), wenn gegenüber Typ 1 einschränkend für alle Regeln $u \rightarrow v$ gilt:
  - ⇨ u ist eine einzelne Variable.
- Eine Grammatik ist vom **Typ 3 (regulär**), wenn gegenüber Typ 2 einschränkend für alle Regeln $u \rightarrow v$ gilt:
  - ⇨ Auf der rechten Seite sind entweder einzelne Terminalzeichen oder ein Terminalzeichen gefolgt von einem Non-Terminalzeichen (linkslinear) bzw. Umgekehrt (rechtslinear)

# Chomsky-Hierarchie

- Als Tabelle dargestellt:

| Grammatik | Typ | Produktionsformen | Automatentyp |
|---|---|---|---|
| allgemein | 0 | beliebig | Turing-Maschine |
| kontextsensitiv | 1 | $\alpha A \beta \rightarrow \alpha \gamma \beta$ | Linear beschränkte Turing-Maschine |
| kontextfrei | 2 | $A \rightarrow \alpha$ | Nicht-deterministischer Kellerautomat |
| regulär | 3 | $A \rightarrow a \mid aB$ | Endlicher Automat |

- $\alpha$ $\gamma$ $\beta$ sind Symbolfolgen bestehend aus Terminalen und Non-Terminalen
- A und B sind Non-Terminale und a ist ein Terminal-Symbol

# Kontextfreie Sprachen

- Ein (Java-) Programm ist ein Wort über dem entsprechenden Alphabet A.
  - ⇨ Es besteht aus den Zeichen für Bezeichner, Anweisungen und festen Token.
  - ⇨ Die Menge aller korrekten Programme ist also eine Sprache über dem Alphabet A.
- Kann man ein Programm als regulären Ausdruck beschreiben?
  Die Antwort ist negativ, wie das folgende Beispiel zeigen wird.

# Kontextfreie Sprachen (Beispiel)

- Sei A = { (, )} das Alphabet, das aus einer öffnenden und schließenden Klammer besteht.
- Die Sprache KK (korrekte Klammerausdrücke) bestehe aus allen Wörtern über A, die aus einem korrekten arithmetischen Ausdruck entstehen, wenn man alles, bis auf die Klammern löscht.
  - ⇨ Es gilt z.B.: () ( ( () () ) () ) ∈ KK,
  - ⇨ aber ( () ) ) ( () ) ∉ KK

# Kontextfreie Sprachen (Beispiel)

- Beweis: (durch Widerspruch)
  - ⇨ Angenommen, man könnte KK durch einen regulären Ausdruck beschreiben, dann gäbe es auch einen endlichen deterministischen Automaten, der KK erkennt.
  - ⇨ Geben wir diesem ein Wort mit k vielen öffnenden Klammern ein, so sei der erreichte Zustand sk.
  - ⇨ Für $k \neq k'$ muss aber $sk \neq sk'$ gelten, denn aus dem ersteren Zustand muss man mit k schließenden Klammern in einem akzeptierenden Zustand, aus dem zweiten darf man das nicht.
    Der Automat müsste also unendlich viele Zustände haben (Widerspruch)

# Beispiel: while-Sprache

- Programm :: **begin** Anweisungen **end**
- Anweisungen :: ε | Anweisungen | Anweisung; Anweisungen
- Anweisung :: Zuweisung | Schleife | Alternative
- Zuweisung :: **id :=** Expr
- Schleife :: **while** Bexpr **do** Anweisung
- Alternative :: **if** Bexpr **then** Anweisung
- Expr :: Expr **+** Expr | Expr **-** Expr | Expr * Expr | **(**Expr**)** | **id** | **number**
- Bexpr :: Expr **=** Expr | Expr **<** Expr | **not** Bexpr | **true** | **false**

# Beispiel: while-Sprache

- Die entsprechende Grammatik ist ein 4er-Tupel **G = (N, T, P, S}** wobei:
  - ⇨ Non-Terminale: in der Grammatik links definierten Begriffe wie Programm oder Zuweisung, die nur zur Beschreibung der Struktur dienen und durch Terminale ersetzt werden müssen
  - ⇨ Terminale: alle Zeichen, die im Programm auftauchen, also: begin, end, ; , id, :=, while, do, if, then, +, -, *, (, ), =, >, num, not, true und false. Insbesondere die Bezeichner id und die Ziffern number werden als Terminale interpretiert, da diese mit Hilfe endlicher Automaten korrekt realisiert werden (lexikalische Analyse).
  - ⇨ Die Produktionen folgen intuitiv aus der EBNF-Beschreibung. Ein Non-Terminal-Zeichen wird ersetzt durch eine beliebige Folge von Terminalen und Non-Terminalen.
  - ⇨ Man beginnt mit dem Non-Terminal- Zeichen: S = Programm

# Kellerautomaten/Stack-Maschinen

- Ein Keller-Automat bzw. eine Stack-Maschine ist wie folgt definiert werden:
  - ⇨ **M = (S, A, B,** δ **,$s_0$ , $b_0$, F)**, wobei
  - ⇨ S die Menge der Zustände beschreibt,
  - ⇨ A das Eingabealphabet
  - ⇨ B das Kelleralphabet
  - ⇨ $\delta : S \times (A \cup \{\varepsilon\}) \times B \rightarrow 2^{S \times B^*}$ die Übergangsfunktion.
  - ⇨ Anfangszustand $\mathbf{s_0}$.
  - ⇨ Anfangswort $\mathbf{b_0}$ auf dem Kellerspeicher und
  - ⇨ F die Menge der akzeptierenden (finalen Zustände)

# Kellerautomaten/Stack-Maschinen

- Ziel: Kellerautomat akzeptiert ein Wort $w \in A^*$, wobei er jeweils das erste Zeichen a des Inputs sieht und wie folgt arbeitet:
  - ⇨ Ist der Automat im Zustand $s \in S$ und ist das oberste Kellerelement $b \in B$, so kann er ein beliebiges $(s', \alpha) \in \delta(s, \varepsilon, b)$ wählen und in den neuen Zustand s' wechseln und das oberste Kellerelement durch das Wort $\alpha \in B^*$ ersetzen.
  - ⇨ Ist das Eingabezeichen a, so kann er stattdessen auch ein $(s', \alpha) \in \delta(s, a, b)$ wählen, das Zeichen a einlesen, in den neuen Zustand s' wechseln und das oberste Kellerelement durch das Wort $\alpha \in B^*$ ersetzen.

# Beispiel: Sprache KK

- Wie könnte ein entsprechender Kellerautomat M aussehen, so dass L (M) = L (G) gilt?

- Wähle **M = (A, B,** $\delta$ **,** $\mathbf{b_0}$**)**, wobei
  - ⇨ A = { (, ) },
  - ⇨ B = {(, ), $b_0$, $b_1$},
  - ⇨ $\delta$ (ε ,$b_0$) = {$b_1$}, $\delta$ (ε ,$b_1$) = {ε , ($b_1$), $b_1 b_1$} und
    $\delta$ ( ( ,( ) = $\delta$ ( ) ,) ) = {ε },
  - ⇨ ansonsten $\delta$ (a,b) = ∅

- Wie kann dieser Automat beschrieben werden?

# Konstruiere Kellerautomat für beliebige kontextfreie Sprache

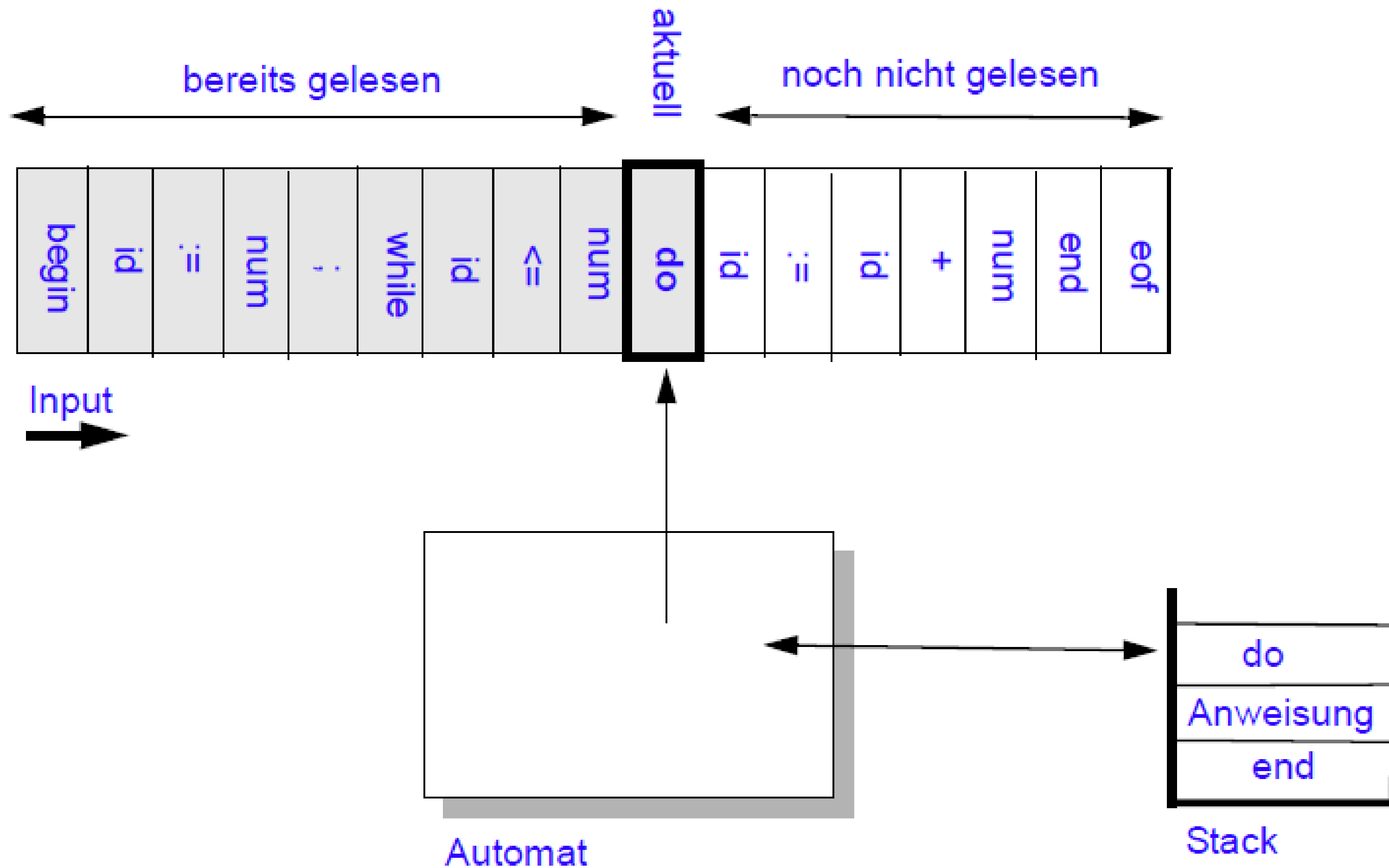
- Sei eine kontextfreie Sprache L (G), die durch Grammatik G = (N, T, P, S) mit Terminalsymbolen T und Nonterminalen N beschrieben werden kann, vorgegeben.

- Um einen Kellerautomaten M zu konstruieren, welcher die entsprechende kontextfreie Sprache akzeptiert, d.h. L (G) = L (M), wählen wir den folgenden Kellerautomaten **M = (A, B,** $\delta$ **, $b_0$)** mit:
  - ⇨ A = T,
  - ⇨ $B = T \cup N$,
  - ⇨ Für jedes Terminalsymbol $a \in A$ setzen wir: $\delta(a,a) = \varepsilon$
  - ⇨ Für jede Menge von Produktionen $(A, \alpha_i) \in P$ für i = 1 bis i = n mit $A \in N$ und $\alpha_i \in (T \cup N)^*$,
    setzen wir: $\delta(\varepsilon, A) = \{ \alpha_1, \alpha_2, \ldots, \alpha_n \}$
  - ⇨ $b_0 = S$

# Konstruiere Kellerautomat für beliebige kontextfreie Sprache (Beweis)

- Zum Beweis wird der Automat mit dem Startsymbol $b_0$ gestartet auf dem ansonsten leeren Stack, wobei man das Wort $w \in A^*$ als Input annimmt.
- Der Automat kann offensichtlich jede Linksableitung nachvollziehen. Dazu betrachten wir eine solche Linksableitung, wobei jeweils Ai die „linkesten“ Nonterminale sein sollen:
  - ⇨ $b0 \rightarrow w_1 A_1 \beta_1 \rightarrow \ldots \rightarrow w_1 \ldots w_k A_k \beta_k \rightarrow w_1 \ldots w_k \alpha_k \beta_k \rightarrow \ldots \rightarrow w$

# Beispiel: Kellerautomat für while-Sprache



# Turing-Maschinen

- Für theoretische Überlegungen interessant, möglichst einfache und überschaubare Maschinen zu haben mit allen theoretischen Fähigkeiten der großen Rechner
- Entwurf einer solchen Maschine von Alan Turing: sog. Turingmaschine.
  - ⇨ Ist die allgemeinste und damit komplexeste Maschine in der Chomsky-Hierarchie
  - ⇨ Akzeptiert alle Sprachen, die von einer beliebigen Grammatik (Typ 0) beschrieben werden können.

# Turing-Maschinen

- Aufbau einer Turing-Maschine ähnelt stark dem Kellerautomat:
  - ⇨ Statt eines Stacks besitzt die Turing-Maschine ein beidseitig unbegrenztes Band, welches in einzelne Kästchen unterteilt ist
  - ⇨ Ein Leseschreibkopf kann ein Zeichen a auf dem Band lesen und abhängig von diesem und von dem internen Zustand s
    - das gelesene Zeichen mit einem neuen Zeichen b überschreiben,
    - ein Kästchen nach rechts oder links gehen und
    - in einen neuen Zustand s' wechseln

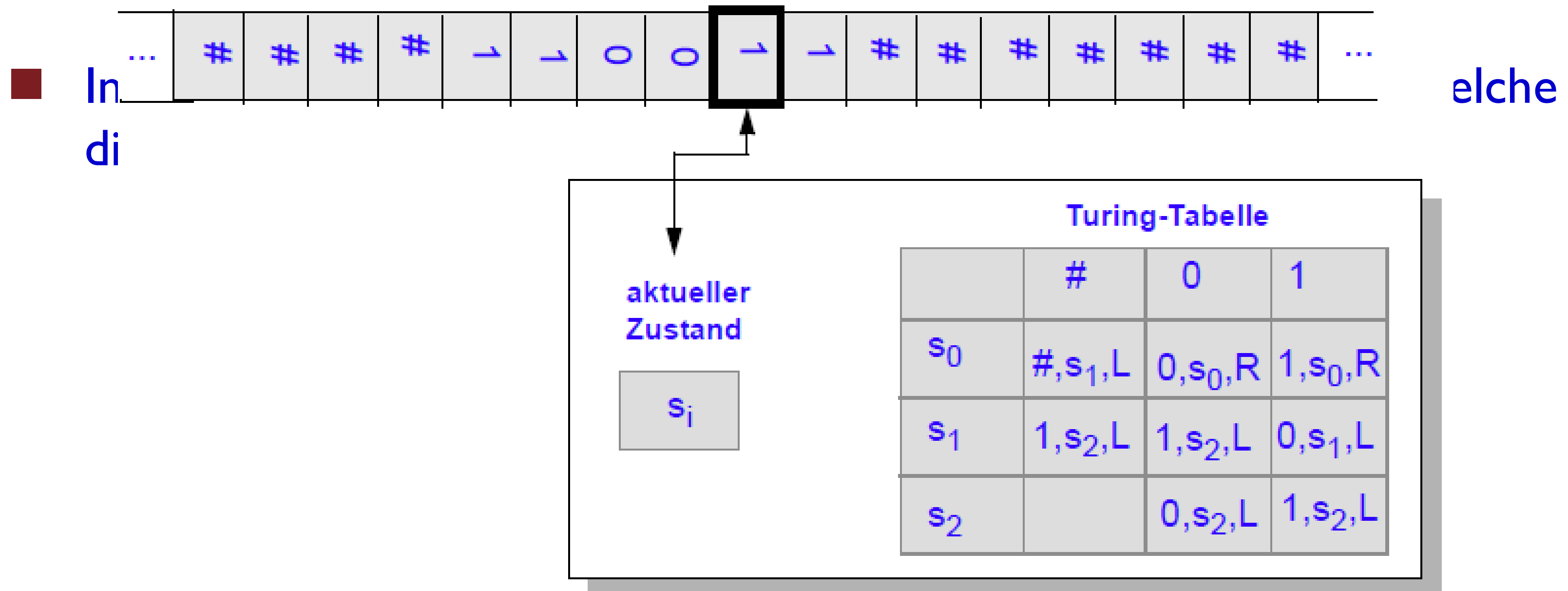
# Turing-Maschinen

- Formal kann die Turing-Maschine durch eine partielle Abbildung $\delta$ beschrieben werden:
  - $\delta : S \times A \longrightarrow S \times A \times \{L, R\}$, wobei
  - A das Alphabet und S die Menge der möglichen Zustände beschreibt und $s_0$ den eindeutigen Startzustand enthält

# Beispiel: Turing-Maschine

- Abbildung δ kann auch als (Turing-) Tabelle ausgedrückt werden, wobei die Zeilen den Zuständen und die Spalten den Eingabezeichen (Alphabet) entsprechen
  - ⇨ d.h. Zeile s und Spalte a enthält in der Turing-Tabelle das Tripel aus neuen Zeichen, neuen Zustand und Bandbewegung, also den Wert δ

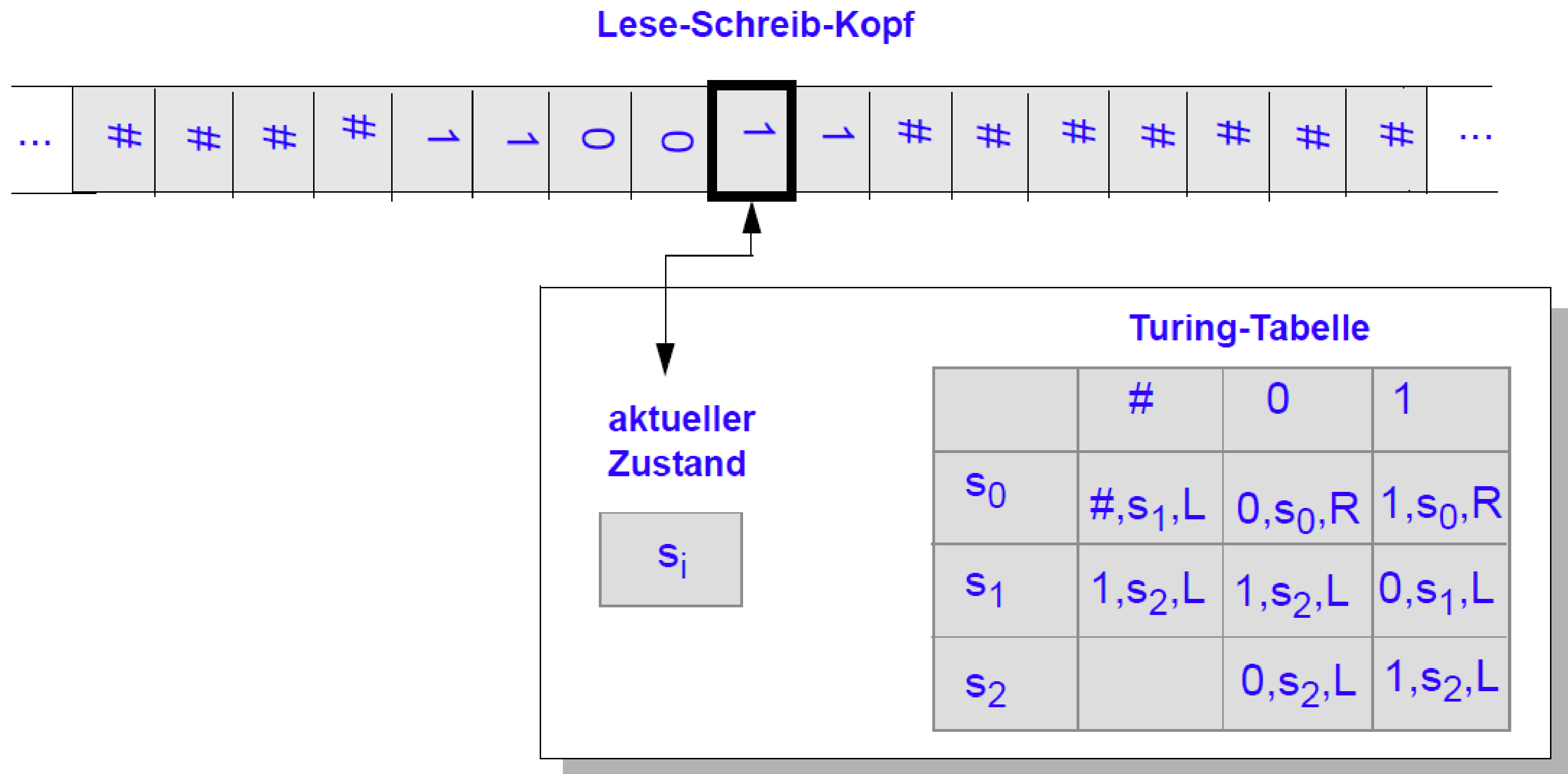
- In … elche di




| | # | 0 | 1 |
|---|---|---|---|
| $s_0$ | $\#,s_1,L$ | $0,s_0,R$ | $1,s_0,R$ |
| $s_1$ | $1,s_2,L$ | $1,s_2,L$ | $0,s_1,L$ |
| $s_2$ | | $0,s_2,L$ | $1,s_2,L$ |

# Beispiel: Turing-Maschine

- In der folgenden Abbildung wird eine Turing-Maschine gezeigt, welche die Zahl 1 zu einer Binärzahl addiert:




| | # | 0 | 1 |
|---|---|---|---|
| $s_0$ | #,$s_1$,L | 0,$s_0$,R | 1,$s_0$,R |
| $s_1$ | 1,$s_2$,L | 1,$s_2$,L | 0,$s_1$,L |
| $s_2$ | | 0,$s_2$,L | 1,$s_2$,L |

# These von Church

- Nachdem wir nun Grammatiken, Automaten und Maschinen kennengelernt haben, ist die folgende Frage interessant:
  - ⇨ Kann man den Begriff des Algorithmus präzisieren und zwar unabhängig von einer bestimmten Hardware und unabhängig von einer bestimmten Programmiersprache und
  - ⇨ wenn ja, welche Menge wird mit dieser Definition beschrieben?
- Die Antwort (These von Church) lautet:
  - ⇨ Es gibt keine eindeutige Definition für den Begriff „Algorithmus“, aber
  - ⇨ jede (vernünftige) Präzisierung des Begriffs führt auf die gleiche Menge der berechenbaren Funktionen

# These von Church

- Diese Menge der berechenbaren Funktionen
  - ⇨ entspricht einer Teilmenge der Sprachen, die durch die Turing-Maschinen beschrieben werden können, und zwar der Menge der rekursiven Sprachen.
  - ⇨ Also die Sprachen, die von mindestens einer Turing-Maschine erkannt werden, welche auf allen Eingaben stoppt.
- Menge der Sprachen, welche durch eine Turing-Maschine beschreibbar sind, definiert die Menge der rekursiv aufzählbaren Sprachen
  - ⇨ und ist eine echte Obermenge der Menge der rekursiven Sprachen, d.h.
  - ⇨ es kann Turing-Maschinen geben, die bei bestimmten Eingaben niemals stoppen

# Berechenbarkeit und Entscheidbarkeit

- Gibt es Problemstellungen, für die es keinen Algorithmus gibt, der sie löst?
  - ⇨ Antwort: Ja!
- Zum Beweise werden wir zwei Aspekte zeigen:
  - ⇨ Es gibt deutlich mehr Funktionen, als es Algorithmen gibt. Daher muss es also viele nicht (durch Algorithmen) berechenbare Funktionen geben.
  - ⇨ Wir konstruieren genau eine solche nicht-berechenbaren Funktion

# Maximale Anzahl berechenbarer Funktionen

- Jeder Algorithmus lässt sich durch einen endlichen Text über einen festen, endlichen Alphabet beschreiben.
- Zur mathematischen Beschreibung dieser (Algorithmen-) Texte verwenden wir ein Alphabet $A = \{a_1, a_2, \ldots , a_n\}$ mit der alphabetischen Ordnung $a_1 < a_2 < \ldots < a_n$.
- Jeder Algorithmus entspricht dann genau einem Text (Wort) w ∈ von A

# Maximale Anzahl berechenbarer Funktionen

- Um die Kardinalität der Menge A* abschätzen zu können, erweitern wir die alphabetischen Ordnung für einzelne Zeichen aus A auf Texte beliebige Länge und erhalten so eine (eindeutige) lexikographische Ordnung auf A*:
  - ⇨ Zu jeder Länge l gibt es $n^l$ verschiedene Texte über A, also endliche viele. Texte kleinerer Länge stehen in der Ordnung vor Texten mit einer größeren Länge.
  - ⇨ Lexikographisch innerhalb der Texte gleicher Länge gilt
    - $z_1 z_2 \ldots z_l < u_1 u_2 \ldots u_l$ genau dann, wenn:
    - $z_1 < u_1 \lor (z_1 = u_1 \land z_2 < u_2) \lor \ldots \lor (z_1 = u_1 \land \ldots \land z_{l-1} = u_{l-1} \land z_l < u_l)$
- Aus der Existenz dieser lexikographischen Ordnung folgt sofort, dass die Menge A* aller Texte abzählbar ist. Daraus folgt sofort:
  - ⇨ Die Menge der durch Algorithmen definierten berechenbaren Funktionen ist abzählbar

# Existenz nicht-berechenbarer Funktionen

- Die Menge F = { f | f: **Z** --> **Z**} der einstelligen Funktionen auf **Z** ist überabzählbar
- Zum Beweis nehmen wir an, F sei abzählbar:
  - ⇨ Dann können wir alle $f \in F$ auflisten, etwa $F = \{f_0, f_1, ... \}$, d.h. jede Funktion f hat eine Nummer i in dieser Folge.
  - ⇨ Wir konstruieren nun eine Funktion g: **Z** --> **Z** mit $g(x) = f_{abs(x)}(x) + 1$.
  - ⇨ Dann gilt für alle i = 1, 2, ... : $g(i) \neq f_i(i)$.
  - ⇨ Also ist für alle i = 1, 2, ... : $g \neq f_i$ .
  - ⇨ Die Funktion g kann also nicht in der obigen Folge vorkommen, obwohl sie eine einstellige Funktion auf **Z** ist.
  - ⇨ Das führt zum Widerspruch, also muss die Annahme, F sei abzählbar, falsch sein

# Existenz nicht-berechenbarer Funktionen

- Berechenbare  Funktionen sind also unter allen Funktionen genauso „seltene“ Ausnahmen wie ganze (oder natürliche oder rationale) Zahlen unter den reellen.
- Auf ähnliche Weise hat Cantor erstmals bewiesen, dass die reellen Zahlen überabzählbar sind

# Konkrete nicht-berechenbare Funktion

- Um mit wenig Darstellungsaufwand eine nichtberechenbare Funktion anzugeben, wird das folgende Algorithmenmodell verwendet:
  - ⇨ Berechnet werden partielle Funktionen f: A* --> A* über einen festen Alphabet A
  - ⇨ Auch die Algorithmen selbst lassen sich als Text über A darstellen.
- Des weiteren werden noch die folgenden Definitionen benötigt:
  - ⇨ Sei $x \in A^*$ ein beliebiger Text. Dann bezeichnet $\phi_x$ die vom Algorithmus mit Text x berechnete Funktion.
  - ⇨ Ist x kein sinnvoller Algorithmentext, so sei $\phi_x$ überall undefiniert.
  - ⇨ Sei f: A* --> A* eine partielle Funktion, Dann ist $D(f) = \{x \in A^* \mid f(x) \text{ ist definiert } \}$ Urbildbereich von f oder auch Definitionsbereich

# Konkrete nicht-berechenbare Funktion

- Der Urbildbereich definiert genau die Menge aller Eingabewerte, bei der die Funktion einen Wert berechnet oder anders formuliert:
  - ⇨ Für alle Werte außerhalb von D(f) hält der Algorithmus nicht.
- Nun sind wir in der Lage eine nichtberechenbare totale Funktion h: A* --> A* konkret anzugeben. Dazu sei a ∈ A ein fest gewählter Buchstabe
  - ⇨ $h(x) = \begin{cases} \varepsilon & \text{falls } x \in D(\phi_x) \\ a & \text{sonst} \end{cases}$

# Beweis: Halteproblem

- Wir nehmen dazu wieder an, die Funktion h: A* --> A* sei berechenbar mit h:

  ⇨ $$h(x) = \begin{cases} \varepsilon & \text{falls } x \in D(\phi_x) \\ a & \text{sonst} \end{cases}$$

- Dann ist aufgrund der Church'schen These auch die folgende Funktion g: A* --> A* berechenbar:

  ⇨ $$g(x) = \begin{cases} \phi_x(x)a & \text{falls } h(x) = \varepsilon \\ \varepsilon & \text{sonst} \end{cases}$$

# Beweis: Halteproblem

- Auf Grund der Definition gilt sofort $g(x) \neq \phi_x(x)$ für alle $x \in A^*$.
- Da g berechenbar ist, gibt es insbesondere einen Algorithmus mit einem Text $y \in A^*$, der g berechnet. D.h. es gibt ein y mit $g = \phi_y$ .
- Damit folgt insgesamt: $\phi_y(y) = g(y) \neq \phi_y(y)$ und damit ein Widerspruch zur Annahme, h sei berechenbar.

# Nicht-entscheidbare Probleme

- Das Halteproblem ist ein Beispiel für ein semantisches Problem von Algorithmen der folgenden allgemeinen Art:
  - ⇨ Kann anhand des Programmtextes (Syntax) entschieden werden, ob die berechnete Funktion (Semantik) eine bestimmte Eigenschaft hat?
  - ⇨ Beim Halteproblem ist dies die Eigenschaft, ob die berechnete Funktion an einer bestimmten Stelle definiert ist, d.h. für eine bestimmte Eingabe terminiert.

# Nicht-entscheidbare Probleme

- Wie steht es nun mit anderen semantischen Eigenschaften von Algorithmen?
  - ⇨ Der Satz von Rice (grundlegender Satz der Algorithmentheorie) besagt, dass jede nicht-triviale semantische Eigenschaft von Algorithmen nicht-entscheidbar ist.
  - ⇨ Eine Eigenschaft ist genau dann trivial, wenn entweder jede oder keine berechnete Funktion sie hat.
  - ⇨ Nicht-triviale Eigenschaften sind demnach solche, die manche berechnete Funktion hat und manche nicht.

# Beispiele nicht-entscheidbare Probleme

- Beispiele für nicht-entscheidbare Probleme sind unter anderem die folgenden Probleme:
  - ⇨ Ist die berechnete Funktion total? Überall undefiniert?
    Injektiv, Surjektiv, Bijektiv? etc.
  - ⇨ Berechnen zwei vorgegebene Algorithmen dieselbe Funktion?
  - ⇨ Ist ein gegebener Algorithmus korrekt, d.h. berechnet er die gewünschte Funktion?

# Korrektheit von Algorithmen

- Es ist wesentlich, dass Algorithmen, die für eine bestimmte Aufgabe entworfen wurden, korrekt sind, d.h. dass sie genau so verhalten, wie beabsichtigt.
- Bei kritischen und zugleich hochsensiblen Anwendungen können durch nicht korrekte Algorithmen katastrophale Fehlfunktionen bzw. Schäden ausgelöst werden.
- Es wurde ja erwähnt, dass die Korrektheit im Allgemeinen nicht entscheidbar ist.
  - ⇨ In vielen Fällen wird man mit pragmatischen Austesten eine hinreichende Sicherheit erreichen können.
  - ⇨ Falls man jedoch an einer mathematischen exakten Korrektheit interessiert ist, muss man in jedem Einzelfall eine Beweis durchführen

# Korrektheit von Algorithmen

- Der Nachweis der Korrektheit eines Algorithmus/Programms bezieht sich immer auf eine Spezifikation dessen, was er tun soll.
  - ⇨ Es handelt sich also immer um eine relative Korrektheit. Ein Algorithmus kann also nicht an sich bewiesen werden, sondern nur in Bezug auf seine Spezifikation.
  - ⇨ Unter Verifikation meint man einen formalen Beweis der Korrektheit bezüglich einer formalen Spezifikation.
  - ⇨ Unter Validation meint man einen nicht-formalen Nachweis der Korrektheit bezüglich einer informellen oder formalen Spezifikation, etwa durch systematischen Testen

# Verifikation

- Verifikation erfordert immer eine Formalisierung der Algorithmen- bzw. Programmiersprache sowie eine Spezifikation in einem mathematischen exakten Formalismus.
- Große Teilgebiete der Softwaretechnik beschäftigen sich mit Spezifikationssprachen, semiautomatischer und automatischer Verifikation und mit Testverfahren.
  - ⇨ Dabei werden unter anderem Ansätze aus der Logik (Kalkül-Systeme) aber auch algebraische und axiomatische Verfahren unterschieden.
  - ⇨ Die meisten dieser Ansätze sind allerdings auf Grund ihrer hohen Komplexität nur für „kleine" Problemstellungen geeignet und daher nicht praktikabel

# Verifikation

- Im folgenden wird ein kurzer Einblick in eine Methode zum Nachweis der Korrektheit bei imperativen Algorithmen:
  - ⇨ Imperative Algorithmen liegen den meisten Programmiersprachen zugrunde
  - ⇨ Die Methode verwendet Vor- und Nachbedingungen bzgl. jeder Anweisung.
  - ⇨ In einer abgeschwächten (weniger formalen Variante) können „gröbere“ Vor- und Nachbedingungen verwendet werden, die sich nun auf größere Anweisungs- Blöcke beziehen

# Vor- und Nachbedingungen

- Falls VOR und NACH (logische) Aussagen über den Zustand vor bzw. nach Ausführung der Anweisung ANW darstellen, kann die Vor- und Nachbedingung wie folgt formalisiert werden:
  - ⇨ {VOR} ANW {NACH}
- Die Semantik der Vor- und Nachbedingungen ist naheliegend und wie folgt definiert:
  - ⇨ Gilt die Bedingung VOR unmittelbar vor Ausführung der Anweisung ANW und terminiert ANW, so muss auch die Bedingung NACH unmittelbar nach Ausführung der Anweisung ANW gelten.
  - ⇨ Terminiert die Anweisung bzw. Anweisungsfolge ANW nicht, so ist die Bedingung trivialerweise wahr, wie auch immer VOR und NACH aussehen.
  - ⇨ Die Bedingung ist auch dann trivialerweise wahr, wenn die Vorbedingung VOR nicht gilt, gleichgültig, ob ANW terminiert oder nicht und ob NACH gilt oder nicht

# Vor- und Nachbedingungen

- Vor- und Nachbedingungen basieren auf dem Konzept eines Zustands, der durch eine Anweisung verändert wird.
  - ⇨ Der Zustand ist bei imperativen Algorithmen über die Werte der Variablen bestimmt.
  - ⇨ Vor- und Nachbedingungen können somit als logische Formeln der Prädikatenlogik unter Verwendung dieser Variablen notiert werden

# Beispiele

- Die Bedingung
  - ⇨ $\{X = 0\}\ X := X + 1\ \{X = 1\}$
  - ⇨ ist wahr, wobei die Variable X Werte aus dem Bereich der ganzen Zahlen **Z** annehmen kann.
- Die Bedingung
  - ⇨ $\{X = i \wedge Y = j \wedge i \neq j\}\ X := Y;\ Y := X\ \{X = j \wedge Y = i\}$
  - ⇨ ist falsch für alle Werte $i, j \in \mathbf{Z}$,
  - ⇨ da nach dem ersten Schritt beide Variablen den Wert j angenommen haben.
  - ⇨ Typischer Anfänger-Programmierfehler beim Wertetausch.

# Beispiele

- Wie sehe eine entsprechende korrekte Anweisungsfolge für den Wertetausch aus?
- Die Bedingung
  - ⇨ $\{X = a\}$ while $X \neq 0$ do $X := X - 1$ od $\{X = 0\}$
  - ⇨ ist wahr für alle Werte $a \in \mathbf{Z}$, insbesondere auch für Werte $a < 0$, da dann die while-Schleife nicht terminiert

# Korrektheit von Programmen

- Ein Programm α (Folge von Anweisungen) heißt **partiell korrekt** bzgl. der Bedingungen VOR und NACH gdw.
  - ⇨ {VOR} α {NACH} wahr ist.
- Das Programm α heißt **total korrekt** bzgl. der Bedingungen VOR und NACH gdw.
  - ⇨ α partiell korrekt ist bzgl. VOR und NACH, und wenn α darüber hinaus immer dann terminiert, wenn vorher VOR gilt