Berechenbarkeitstheorie

Berechenbare Funktionen

Engl. Mathematiker **Alan M. Turing** charakterisierte 1937 mit rein mathematischen Mitteln, was Computer berechnen können und wozu sie niemals in der Lage sein werden.

Berechnung von Funktionen durch Rechner ausdrückbar:

$f\colon \Sigma^* \to \Gamma^*$, wobei Σ das Eingabe- und Γ das Ausgabealphabet ist

Werden nicht alle mögl. Eingaben aus $\Sigma*$ akzeptiert, handelt es sich um eine **partielle Funktion**, die nur auf einer Teilmenge

$def(f) \subseteq \Sigma^*$ definiert ist, also: $def(f) \to \Gamma^*$

Formal wird e. solche partielle Funktion wie folgt angegeben:

$f :: \Sigma^* \to \Gamma^*$, wobei man statt $x \notin def(f)$ oft folgende Schreibweise

$f(x) = \perp$, wobei $\perp$ für *undefiniert* steht .

Partielle Funktion f :: Σ∗ →Γ∗ ist nur dann berechenbar, wenn Algorithmus gefunden werden kann, der sie berechnet.

Berechenbare Funktionen

Beispiel für berechenbare und überall def. (totale) Funktion

Folge von natürlichen Zahlen: 2, 4, 6, 8, 10, 12,…lässt sich mit sehr einfachem Algorithmus berechnen: **f (n) = 2n**

Unterschied zwischen Begriffen **Funktion** und **Algorithmus**:

- **Funktion** = eine Zuordnung oder Abbildung
  → Hier Folge, die jeder natürl. Zahl dopp. Wert zuordnet.
  → Z.B. ist Funktion **g(n) = 4n/2** gleich zur Funktion **f (n)**, da sie die gleiche Zahlenfolge erzeugt, jedoch **mit anderem Algorithmus.**
  → Solche Funktionen wie f (n) und g(n) werden als **überall definierte Funktionen** bezeichnet.

Berechenbare Funktionen

Beispiel für berechenbare, aber nur partiell def. Funktion

Eine partielle Funktion ist: **f (n) = 100 / (n–1)**

n = 1 ➔ Division durch 0 ➔ f (n) bei n = 1 nicht definiert.

Funktionen dieser Art sind **partiell definierte Funktionen.**

Nicht berechenbare Funktionen

Diagonalverfahren von Cantor

Aufstellen unendl. langer Liste von Funktionen, die alle mögl. Kombinationen von Ziffern erzeugen

→ weitere Funktion *f* (*D*)
Diagonalelemente
0 → 1 und !=0 → 0

→ neue Funktion *f* (*D*)
nicht in Liste

→ Liste kann niemals vollständig sein

$$
\begin{array}{lcl}
f(0) & = & \mathbf{1}\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ \ldots \\
f(1) & = & 2\ \mathbf{4}\ 6\ 8\ 0\ 2\ 4\ 6\ 8\ 0\ \ldots \\
f(2) & = & 0\ 0\ \mathbf{0}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ \ldots \\
f(3) & = & 4\ 2\ 4\ \mathbf{2}\ 4\ 2\ 4\ 2\ 4\ 2\ \ldots \\
f(4) & = & 5\ 5\ 5\ 5\ \mathbf{5}\ 5\ 5\ 5\ 5\ 5\ \ldots \\
f(5) & = & 0\ 1\ 0\ 0\ 1\ \mathbf{0}\ 0\ 0\ 1\ 0\ \ldots \\
f(6) & = & 1\ 4\ 2\ 8\ 5\ 7\ \mathbf{9}\ 5\ 3\ 5\ \ldots \\
 & & \ldots\ldots\ldots \\
\hline
f(D) & = & \mathbf{0}\ \mathbf{0}\ \mathbf{1}\ \mathbf{0}\ \mathbf{0}\ \mathbf{1}\ \mathbf{0}\ \ldots\ldots\ldots
\end{array}
$$

Nicht berechenbare Funktionen

Nicht durch Algorithmus berechenbare Funktionen

- Menge aller mögl. Teilmengen von N = Potenzmenge P(N).
- Satz von Cantor → mehr Teilmengen als Elemente von N
  → Potenzmenge P(N) überabzählbar
  → auch Menge aller Funktionen $f : N \rightarrow N$ überabzählbar.
- Es gibt nur abzählbar viele berechenbare Funktionen, da es zu jeder berechenbaren Funktion Algorithmus geben muss.
  → Algorithmus muss aus endl. Text (Wort aus $\Sigma*$) sein.
  → Daher gibt es nur abzählbar viele Algorithmen.
  → Auch nur abzählbar viele berechenbare Funktionen.
  → In Potenzmenge sind sehr viele Funktionen übrig, die nicht durch einen Algorithmus berechnet werden können.

Nicht berechenbare Funktionen

Funktion (nicht) berechenbar??

→ **präzisere Definition des Begriffs „Algorithmus“ erforderl.**

Church’sche Algorithmus-Definition:

Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

→ **Die Menge der berechenbaren Funktionen ist die Menge aller Funktionen, die sich mit einer Turingmaschine berechnen lassen.**

Berechenbarkeitskonzepte

Turingmaschinen

Formal kann man eine deterministische Turingmaschine als 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z0, \#, E)$ beschreiben, wobei Folg. gilt:

Z *eine endliche Menge von Zuständen,*

$\Sigma \subset \Gamma$ *ein endliches Eingabealphabet,*

Γ *ein endliches Bandalphabet,*

$\delta :: Z \times \Gamma \rightarrow Z \times \Gamma \times \{L, 0, R\}$ *eine Überführungsfunktion,*

$z_0 \in Z$ *der Startzustand,*

\# *für leeres Feld* und

$E \subseteq Z$ *eine Menge von akzeptierenden Endezuständen.*

Berechenbarkeitskonzepte

Turingmaschinen

Turingmaschinen spielen in Informatik eine wichtige Rolle:

Jedes Problem, das überhaupt maschinell lösbar ist, kann von einer Turingmaschine gelöst werden.

Tätigkeit der Turingmaschine durch 5-Tupel beschreiben:

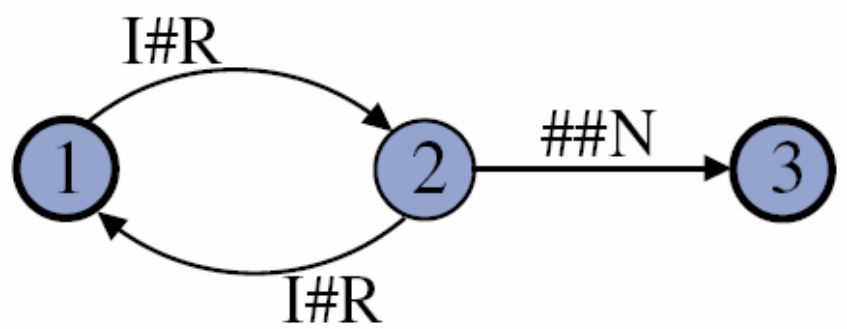
(Zustand, Eingabezeichen, Ausgabezeichen, Kopfbewegung, Folgezust.).

Berechenbarkeitskonzepte

Turingmaschinen

Beispiel: Turingmaschine zur ungeraden Paritätsprüfung

→ **Ungerade Anzahl (I, III, IIIII, …) akzeptieren.**



1I#R2: Liest sie im Zustand 1 Strich, ersetzt sie ihn durch Leerz. #, bewegt Kopf R(echts) und geht in Zustand 2

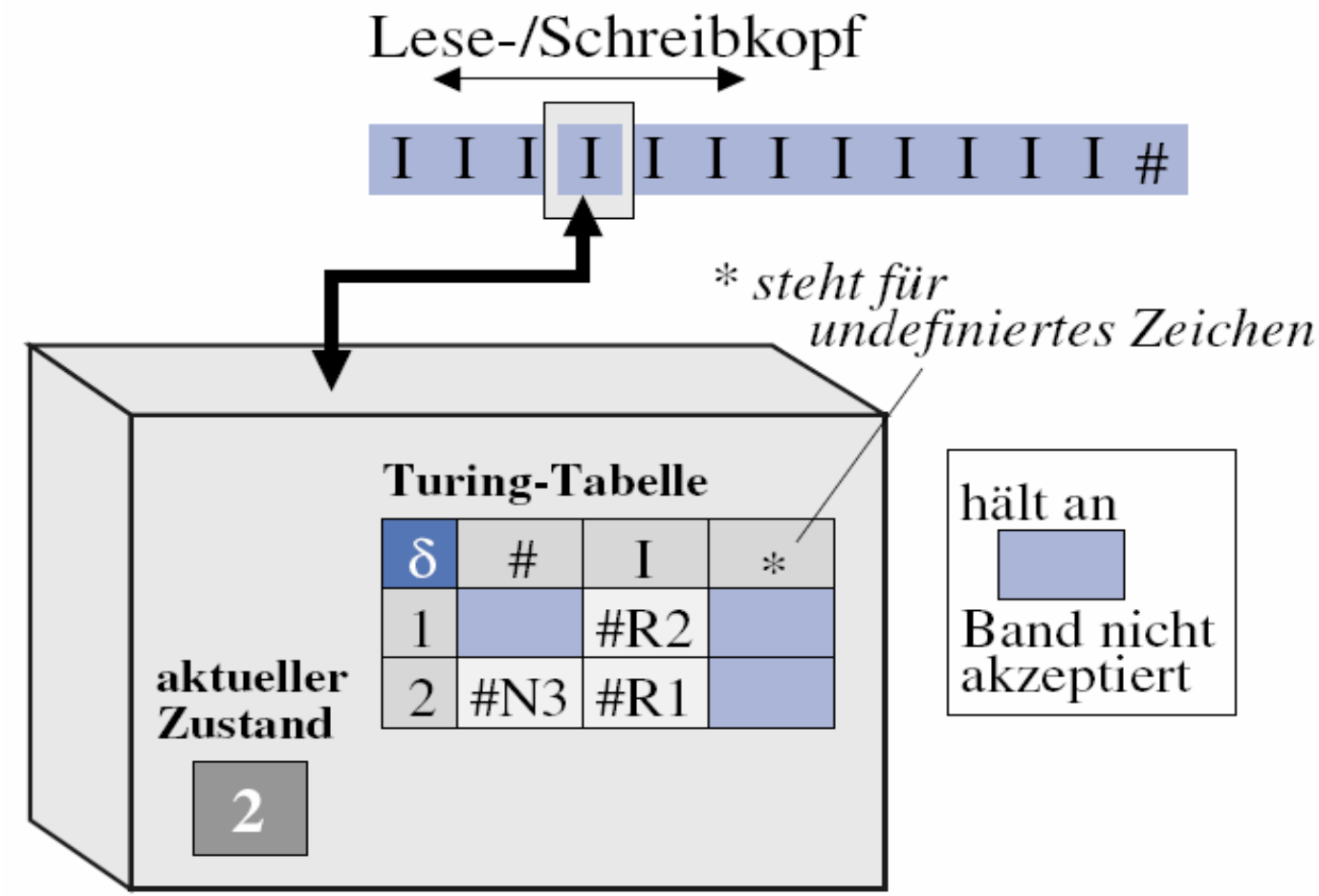
2I#R1: Liest sie im Zustand 2 Strich, ersetzt sie ihn durch Leerz. #, bewegt Kopf R(echts) und geht in Zustand 1

2##N3: Liest sie im Zustand 2 Leerz. #, macht sie N(ichts), wechselt in akzeptierenden Zustand 3 und hält an.

Berechenbarkeitskonzepte

Turingmaschinen

Beispiel: Turingmaschine zur ungeraden Paritätsprüfung

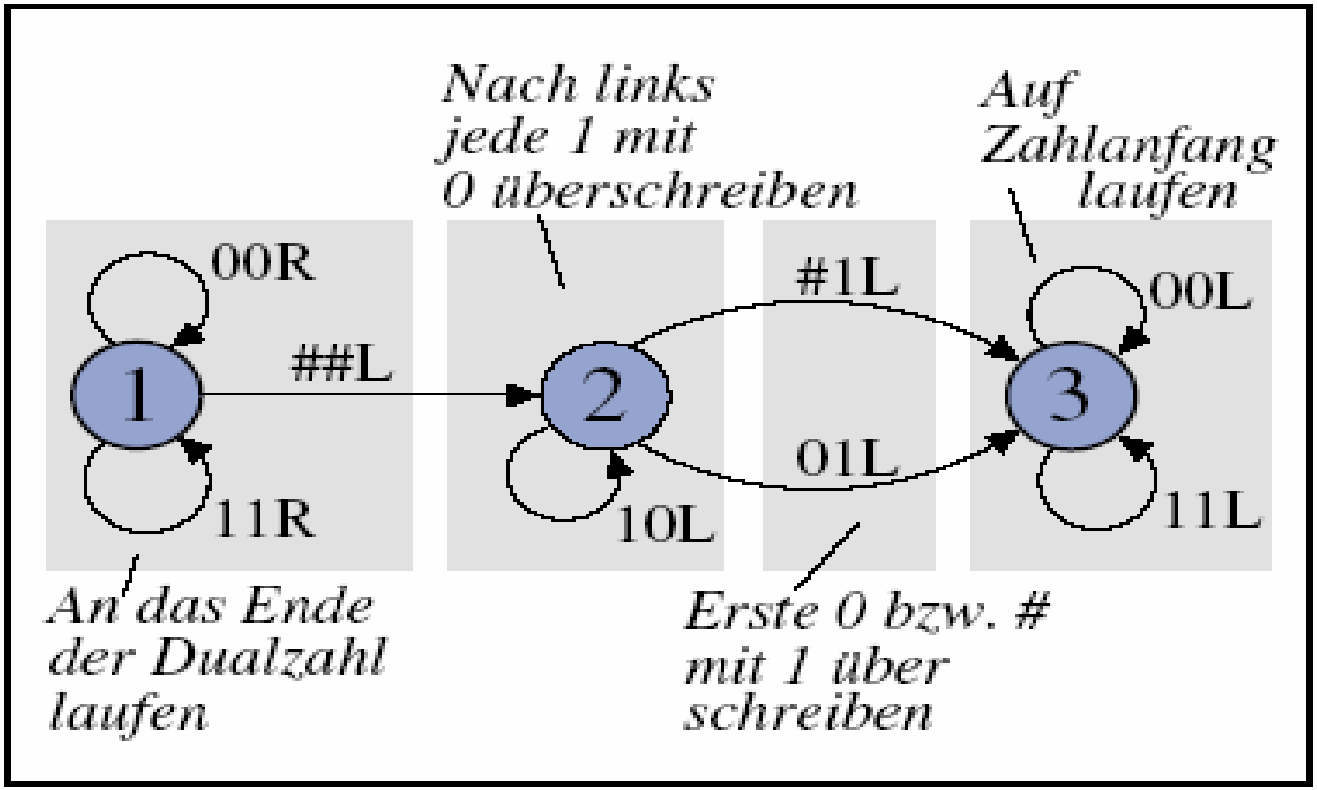


Berechenbarkeitskonzepte

Turingmaschinen

Beispiel: Turingmaschine zur dualen Addition von 1

- **aus Bandinschrift #1100# ➔ Bandschrift #1101#**
- **aus Bandinschrift #11011# ➔ Bandschrift #11100#**

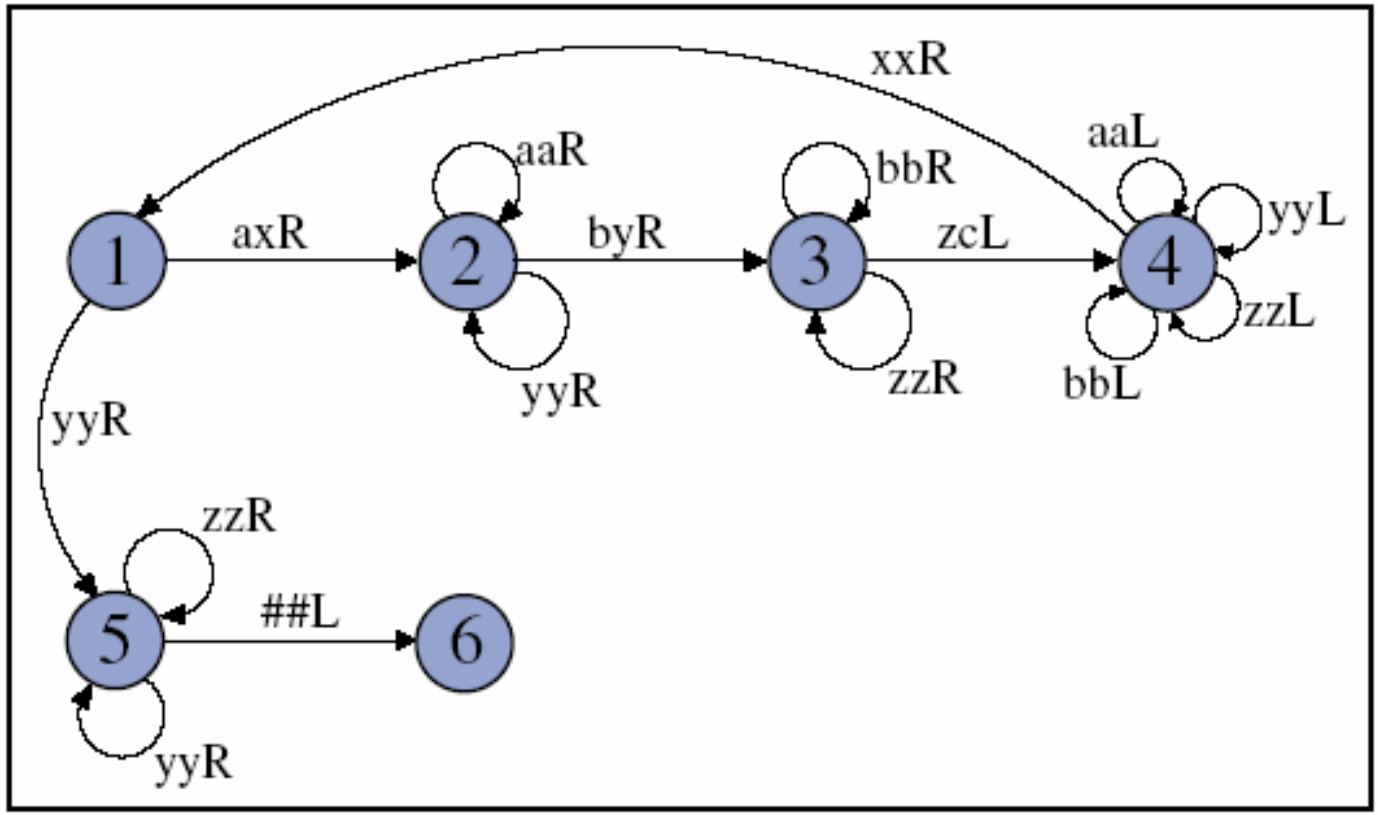


Berechenbarkeitskonzepte

Turingmaschinen

Beispiel: Turingmaschine, die Wörter aus folgender formaler Sprache erkennt: $L = \{a^n b^n c^n | n \geq 0\}$

Erlaubte Wörter sind z. B. aaaabbbbcccc, abc, aabbcc.

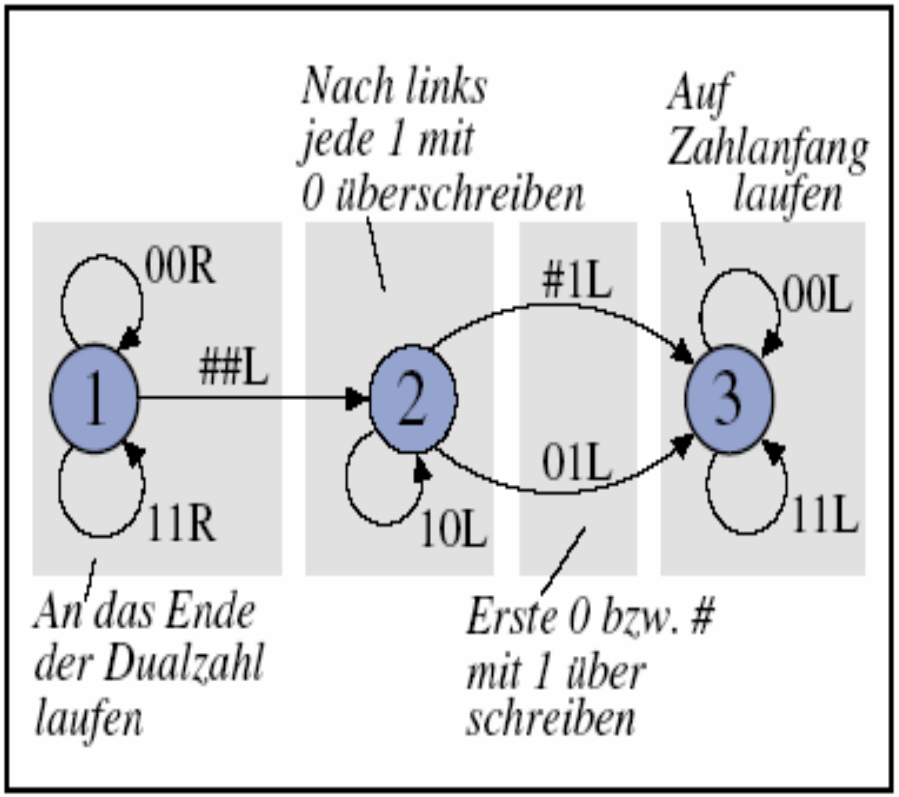


Berechenbarkeitskonzepte

Turingmaschinen

Eigentlich müsste man zw. Hardware (Turingmaschine selbst) und Software (Turingtabelle/Turingprogramm) unterscheiden

→ durch sehr einfache Programmiersprache beschreibbar, die nur LEFT, RIGHT, WRITE, IF, JUMP und HALT kennt.




```
// .....  An das Ende der Dualzahl laufen
 10 RIGHT
 20 IF 0 JUMP 10
 30 IF 1 JUMP 10
// .....  Nach links jede 1 mit Nullen ueberschreiben
 40 LEFT
 50 IF 1 JUMP 80
 60 IF 0 JUMP 100
 70 IF # JUMP 140
 80 WRITE 0
 90 IF 0 JUMP 40
100 WRITE 1
// .....  Nach links auf Zahlenanfang laufen
110 LEFT
120 IF 0 JUMP 110
130 IF 1 JUMP 110
140 HALT
```

Berechenbarkeitskonzepte

Turing-berechenbare Funktionen

- Um eine partielle Funktion $f :: N^k \rightarrow N$ zu berechnen, schreibt man die Argumente $(n_1, n_2, \ldots, n_k)$ in der Form $n_1\#n_2\#\ldots\#n_k$ auf das Band, positioniert Lese-/Schreibkopf auf das 1. Zeichen von n_1 und startet dann die Turingmaschine in ihrem Anfangszustand.

- Hält TM an → Ergebnis ab Zeichen unter Lese-/Schreibkopf nach rechts vom Band ablesbar.

  → Ist $(n_1, n_2, \ldots, n_k)$ eine von f akzeptierte Eingabe, muss Ergebnis gleich Zahl f $(n_1, n_2, \ldots, n_k)$ sein, sonst wird TM nicht anhalten oder unter Lese-/Schreibkopf beginnt keine Zahl.

Berechenbarkeitskonzepte

Turing-berechenbare Funktionen

- Hat man Turingtabelle zur Berechnung von f, kann man sie „programmieren“, dass sie am Ende Zwischenergebnisse auf Band löscht und dort nur noch Ergebnis f (n_1, n_2,…, n_k) und Lese-/Schreibkopf auf 1.Zeichen des Ergebnisses steht.

→ So lässt sich sofort mit diesem Ergebnis weiterrechnen, indem man dieses als Eingabe für eine nachfolgende Turingmaschine verwendet.

→ Es gilt nämlich, dass sequenzielle Ausführung zweier Turing-berechenbarer partieller Funktionen f, g :: N → N wieder Turing-berechenbar ist.

Berechenbarkeitskonzepte

Registermaschinen

Eine Registermaschine kann alles berechnen, was auch ein realer Rechner berechnen kann, da man mit Registermaschinen Turingmaschinen simulieren kann.

→ **Z.B. könnte man die Speicherzellen mit geraden Nummern zur Speicherung der Daten auf dem Eingabeband der Turingmaschine verwenden, und dann nur noch die Speicherzellen mit ungeraden Nummern für die anderen Daten verwenden.**

Berechenbarkeitskonzepte

GOTO-Programme

- GOTO-Sprache verwendet statt Registern Variablennamen
  → ist somit höherer Programmiersprache schon näher.
  → Jede Zeile eines GOTO-Programms enthält entweder eine Zuweisung oder einen bedingten Sprung, wobei am Anfang jeder Zeile die Zeilennummer steht:

znr: *id* := *ausdruck*
znr: **IF** *bed* **GOTO** *znr*
Für arithmetische und boolesche Ausdrücke gilt folgende Backus-Naur-Form:
ausdruck : *id*
| **0**
| **succ**(*ausdruck*) /* liefert Nachfolgezahl zur natürlichen Zahl x, als $x+1$ */
bed : *ausdruck* <= *ausdruck*

Berechenbarkeitskonzepte

GOTO-Programme

Realisierung der nicht in der GOTO-Programmiersprache vorhandenen Addition von zwei natürlichen Zahlen x := a + b:

```
100:   ....
101:  x := a
102:  z := 0
103:  IF b <= z GOTO 107
104:  x := succ(x)
105:  z := succ(z)
106:  GOTO 103
107:   ....   ; Ergebnis steht in x
```

Jedes GOTO-Progr. kann jede Registermaschine simulieren
→ jede Turing-berechenbare Funktion ist auch GOTO-berechenbar.

Berechenbarkeitskonzepte

GOTO-Programme

Umgekehrt: Jedes GOTO-Programm ist durch Reg.maschine simulierbar, indem man jeder Variablen eine Speicherzelle im Datenspeicher zuordnet, und dann jede Zuweisung durch entsprechende Load/Store-Befehle nachbildet.

Z.B. kann man die GOTO-Zuweisung: a := succ(b) mit folgenden Befehlen einer Registermaschine realisieren:

```
LDK 1
ADD 02   ; wenn 02 Adresse von b ist
STA 03   ; wenn 03 Adresse von a ist
```

Somit sind beide Berechnungskonzepte äquivalent und man kann mit beiden die gleichen partiellen Funktionen berechnen

Berechenbarkeitskonzepte

WHILE-Programme

Ein WHILE-Programm besteht aus einer Zuweisung, einer bedingten Anweisung, einer WHILE-Schleife oder einer Sequenz von Programmen:

$$
\begin{array}{lll}
\mathit{Prog} & : & \mathit{id} := \mathit{ausdruck} \\
 & | & \textbf{IF } \mathit{bed} \textbf{ THEN } \mathit{Prog} \textbf{ ELSE } \mathit{Prog} \\
 & | & \textbf{WHILE } \mathit{bed} \textbf{ DO } \mathit{Prog} \textbf{ DONE} \\
 & | & \mathit{Prog} \ ; \ \mathit{Prog} \\
\mathit{bed} & : & \mathit{ausdruck} <= \mathit{ausdruck} \\
\mathit{ausdruck} & : & \mathit{id} \\
 & | & \mathbf{0} \\
 & | & \mathbf{succ}(\mathit{ausdruck})
\end{array}
$$

Berechenbarkeitskonzepte

WHILE-Programme

Folg. WHILE-Programm realisiert die in der WHILE-Sprache nicht vorhandene Addition von zwei Zahlen x := a+b und die Multiplikation zweier Zahlen e := a ∗ b:

```
//  ..............  Addition: x := a + b
z := 0;
x := a;
WHILE succ(z) <= b DO
    x := succ(x);
    z := succ(z)
DONE
```

```
//  ..............  Multiplikation: e := a * b
m := 0;
e := 0;
WHILE succ(m) <= a DO
    z := 0;
    WHILE succ(z) <= b DO
        e := succ(e);
        z := succ(z)
    DONE
    m := succ(m)
DONE
```

Berechenbarkeitskonzepte

WHILE-Programme

Jede WHILE-Schleife ist durch bedingten Sprung nachbildbar:

WHILE-Programm

WHILE $x \leq y$ **do**
Prog

ist identisch zu:

GOTO-Programm

znr1: **IF** *succ*(y) $\leq x$ **GOTO** *znrX*
... *Prog*
... **IF** $x \leq y$ **GOTO** *znr1*
znrX:

→ **also ist es möglich, dass man ohne GOTOs programmieren kann**

Berechenbarkeitskonzepte

WHILE-Programme

Umgekehrt kann man mit WHILE- auch GOTO-Programme und damit auch Registermaschinen simulieren.

→ Programmzähler pc als zusätzl. Variable und ersetzt jede GOTO-Anweisung wie folgt durch eine WHILE-Anweisung:

GOTO-Anweisung		WHILE-Anweisung
znr: *id* := *ausdruck*	durch	*id* := *ausdruck*; pc := pc + 1;
znr: **IF** *bed* **GOTO** *znr*	durch	**IF** *bed* pc := *znr*

Ein beliebiges GOTO-Programm

```
1: gotoAnw1
2: gotoAnw2
  .....
n: gotoAnwN
```

kann durch ein WHILE-Programm der folgenden Form simuliert werden:

```
pc = 1
WHILE PC <= n DO
    IF pc == 1 THEN whileAnw1;
    IF pc == 2 THEN whileAnw2;
      ........
    IF pc == n THEN whileAnwN;
DONE
```

Berechenbarkeitskonzepte

WHILE-Programme

Es gilt somit: Jedes GOTO-Programm lässt sich in ein äquivalentes WHILE-Programm umwandeln und umgekehrt.

Da WHILE-berechenbare Funktionen zugleich auch GOTO-berechenbar und damit auch Turing-berechenbar sind, stellte Kleene sein Kleene'sches Normalformentheorem auf:

Jede berechenbare Funktion kann mit einer WHILE-Schleife programmiert werden.

Berechenbarkeitskonzepte

LOOP-Programme

Bei LOOP-Programmen wird WHILE-Schleife durch FOR-Schleife ersetzt, wobei diese FOR-Schleife aber etwas anders ausgelegt ist als die for-Schleife in C/C++ oder Java:

$$
\begin{aligned}
Prog \;:\; & \; id := ausdruck \\
& \mid \textbf{IF } bed \textbf{ THEN } Prog \textbf{ ELSE } Prog \\
& \mid \textbf{FOR } i := ausdruck \textbf{ TO } ausdruck \textbf{ DO} \\
& \mid Prog \;;\; Prog \\
bed \;:\; & \; ausdruck <= ausdruck \\
ausdruck \;:\; & \; id \\
& \mid \mathbf{0} \\
& \mid \mathbf{succ}(ausdruck)
\end{aligned}
$$

Berechenbarkeitskonzepte

LOOP-Programme

Folg. LOOP-Programm addiert alle Werte zwischen 1 und 10:

```
sum := 0;
FOR i:=1 TO 10 DO
    FOR j := 1 TO i DO
        sum := succ(sum)
    DONE
DONE
```

Da bei LOOP-Programmen immer ein Start- und ein Endwert angegeben ist, gilt Folgendes:

Jedes LOOP-Programm terminiert.

Die LOOP-Sprache ist schwächer als die WHILE-Sprache, da man weder einen Turing- noch einen Registermaschinen-Simulator in der LOOP-Sprache schreiben kann.

Berechenbarkeitskonzepte

Primitive Rekursion

Funktionale Programmiersprachen kommen auch völlig ohne Zuweisung und Schleifen aus, denn jede Funktion kann man direkt durch Rekursion definieren.

Primitive Rekursion lässt sich wie folgt definieren:

Zu einer k-stelligen Funktion f und einer (k+2)-stelligen Funktion g ist eine (k+1)-stellige Funktion R_{fg} wie folgt definiert:

$$R_{fg}(0, x_1, \ldots, x_k) = f(x_1, \ldots, x_k)$$
$$R_{fg}(n+1, x_1, \ldots, x_k) = g(n, R_{fg}(n, x_1, \ldots, x_k), x_1, \ldots, x_k)$$

In 2. Gleichung tritt Funktion R_{fg} links und rechts von = auf.
→ Trotzdem keine zirkulare Definition, denn auf rechter Seite wird Parameter n im 1. Argument um 1 erniedrigt.

Berechenbarkeitskonzepte

Primitive Rekursion

Bei primitiver Rekursion ist noch erlaubt:

→ Komposition (Einsetzen schon vorh. Funktionen) und

→ Vertauschen von Argumentpositionen.

Definition:

Eine Funktion $f : N^k \rightarrow N$ heißt primitiv rekursiv, wenn sie aus 0, succ() und den Projektionen von Verkettung und primitiver Rekursion definiert werden kann, wobei Folgendes gilt:

- Projektion: $p_{k,i} : N^k \rightarrow N$ definiert durch $p_{k,i}(x_1,\ldots, x_k) = x_i$
- Verkettung: Ist f n-stellige und $g_1,\ldots, g_n$ k-stellige Funktion, so ist Verkettung $f \circ (g_1,\ldots, g_n) : N^k \rightarrow N$ definiert durch:

$$(f \circ (g_1,\ldots, g_n))(x_1,\ldots, x_k) = f\,(g_1(x_1,\ldots, x_k),\ldots, g_n(x_1,\ldots, x_k))$$

Berechenbarkeitskonzepte

Grundoperationen der Arithmetik mit primitiver Rekursion

Addition durch $k = 1$, $f = p_{1,1}$, $g = succ \circ (p_{3,2})$:

$$add(0, y) = p_{1,1}(y) = y$$
$$add(x + 1, y) = succ(p_{3,2}(x, add(x, y), y)) = succ(add(x, y))$$

Multiplikation durch nochmalige primitive Rek. auf add:

$$mult(0, y) = 0 \circ () = 0$$
$$mult(x+1,y) = (add \circ (p_{3,2}, p_{3,3}))(x,mult(x,y),y)=add(mult(x,y),y)$$

Vorgängerfunktion pred():

$$pred(0) = 0$$
$$pred(x + 1) = p_{2,2}(pred(x), x) = x$$

Subtraktion durch $k = 1$, $f = p_{1,1}$, $g = pred \circ (p_{3,2})$:

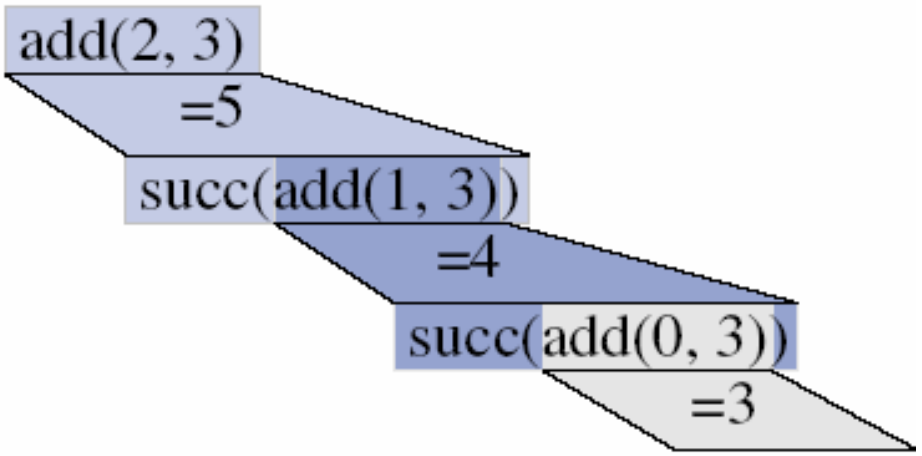
$$sub(0, y) = p_{1,1}(y) = y$$
$$sub(x + 1, y) = pred(p_{3,2}(x, sub(x, y), y)) = pred(sub(x, y))$$

Berechenbarkeitskonzepte

Grundoperationen der Arithmetik mit primitiver Rekursion

C-Programm zu add() und mult() mit primitiver Rekursion

```
int succ(int x) { return x + 1; }
int pred(int x) { return x==0 ? 0 : x−1; }
int add (int x, int y) { return (x==0) ? y : succ(add(pred(x), y)); }
int mult(int x, int y) { return (x==0) ? 0 : add(mult(pred(x), y), y); }
```



$$add(2,3) = succ(succ(3)) = 5$$

Berechenbarkeitskonzepte

Primitive Rekursion

Jede primitiv rekursive Funktion ist LOOP-berechenbar.

Oder anders ausgedrückt:

Jede „vernünftig“ total berechenbare Funktion ist primitiv rekursiv, wobei „vernünftig“ bedeutet, dass man die maximal möglichen Schleifendurchläufe vorab bestimmen kann.

add()- und mult()-Realisierung mit LOOP-Programm:

```
int add(int x, int y) {
    int i, wert = x;

    for (i=1; i <= y; i++)
        wert = succ(wert);
    return wert;
}
```

```
int mult(int x, int y) {
    int i, wert = 0;
    for (i=1; i <= y; i++)
        wert = add(wert, x);
    return wert;
}
```

Berechenbarkeitskonzepte

μ-Rekursion

Probleme, die nicht mit primitiv rek. Funktionen lösbar sind. Z.B. Suchprobleme, bei denen für (k+1)-stellige bool. Funktion f (z, x_1,…, x_k) kleinstes z zu finden ist, für das Funktion true liefert. Hier mit z = 0 beginnen, dann z = 1 probieren usw.

Allgemein lässt sich dies wie folgt rekursiv beschreiben:

$$\mu f(z, x_1, \ldots, x_k) = \begin{cases} z & , \quad \text{wenn } f(z, x_1, \ldots, x_k) = \text{true} \\ \mu f(z+1, x_1, \ldots, x_k) & , \quad \text{sonst} \end{cases}$$

Solche Funktion = μ-rekursiv (nicht mehr primitiv rekursiv), da beim rekursiven Aufruf ein Argument nicht mehr kleiner, sondern größer wird:

→ **Gefahr, dass Funktion zu keinem Ende führt, wenn kein z existiert, für das f(z, x_1,…, x_k) Wahrheitswert true liefert.**

Berechenbarkeitskonzepte

μ-Rekursion

- Ist die Funktion f eine totale Funktion
  → besteht diese Gefahr nicht, denn dann liefert diese Funktion ja für alle möglichen Werte von z Wert true und somit würde sie beim ersten Aufruf stoppen.

- Handelt es sich bei f aber um partielle Funktion
  → besteht diese Gefahr des Nichtterminierens sehr wohl.

Berechenbarkeitskonzepte

μ-Rekursion

Es gilt nun: Klasse der μ-rekursiven Funktionen stimmt mit Klasse der WHILE-berechenbaren Funktionen überein.

```
z := 0;
WHILE NOT f(z, x1, ..., xK) DO
    z := succ(z);
DONE
```

Da μ-rekursive Funktionen zugleich auch WHILE-berechenbar und damit auch GOTO- und Turing-berechenbar sind, gilt:

Jede berechenbare Funktion lässt sich mit μ-rekursiven Funktionen definieren.

Berechenbarkeitskonzepte

Ackermann-Funktion

Bisher haben wir Folgendes festgestellt:

- **Jede LOOP-berechenbare Funktion ist total.**
- **Jede total berechenbare Funktion, für die mit LOOP-Prog. eine Abschätzung der maximal möglichen Schleifendurchläufe möglich ist, ist auch LOOP-berechenbar.**
- **LOOP-berechenbare Funktionen sind genau die primitiv rekursiven Funktionen.**

Nun stellt sich die Frage, ob jede total berechenbare Funktion grundsätzlich immer durch ein LOOP-Programm berechenbar ist.

Berechenbarkeitskonzepte

Ackermann-Funktion

Die Ackermann-Funktion ist ein Beispiel für eine totale und berechenbare Funktion, die nicht primitiv rekursiv ist:

$$a(n,m) = \begin{cases} m+1 & : \text{ wenn } n = 0 \\ a(n-1,1) & : \text{ wenn } m = 0 \\ a(n-1,a(n,m-1)) & : \text{ sonst} \end{cases}$$

n: **0**
m: **10**
a(0,10) = 11

n: **1**
m: **10**
a(1,10) = 12

n: **2**
m: **2**
a(2,2) = 7

Berechenbarkeitskonzepte

Ackermann-Funktion

$$a(n,m) = \begin{cases} m+1 & : \text{ wenn } n = 0 \\ a(n-1,1) & : \text{ wenn } m = 0 \\ a(n-1,a(n,m-1)) & : \text{ sonst} \end{cases}$$

- Wächst sehr stark an und kann von keiner primitiv rekursiven Funktion nach oben beschränkt werden.
- Entworfen als Verallgemeinerung der Grundrechenarten:

$$a(0,y) = y+1$$
$$a(1,y) = y+2$$
$$a(2,y) = 2y+3$$
$$a(3,y) = 2^{y+3} - 3$$
$$a(4,y) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$$

$$a(4,0) = 2^4 - 3 = 13$$
$$a(4,1) = 2^{16} - 3 = 65533$$
$$a(4,2) = 2^{65536} - 3$$

Wegen $[65536 * \log_{10} 2]$ hat Zahl a(4, 2) nicht weniger als 19728 Dezimalstellen.

Berechenbarkeitskonzepte

Ackermann-Funktion

Die Zahl a(4, 4) ist bereits unvorstellbar.
Schreibt man 4 Stellen dieser Zahl je Sekunde, würde man

$$10^{10^{19727}} \text{ Sekunden benötigen.}$$

Als Vergleich: Das Alter der Erde beträgt 10^{17} Sekunden.

Eine weitere mehrfach rekursive Funktion, ähnlich der Ackermann-Funktion, ist die **Hofstadter-Funktion:**

$$hof(n) = \begin{cases} 1 & : \text{ wenn } n \leq 2 \\ hof(n - hof(n-1)) + hof(n - hof(n-2)) & : \text{ wenn } n > 2 \end{cases}$$

Diese Beispiele zeigen, dass nicht jede berechenbare Funktion auch durch ein LOOP-Programm berechenbar ist.

Berechenbarkeitskonzepte

Church'sche These und Chomsky-Hierarchie

Möglichkeiten für math. Definition des Begriffs „Algorithmus“:

$$\begin{aligned}\text{Algorithmus} &= \text{Turing-Programm} &&= \text{Registermaschinenprogramm} = \\ &= \text{WHILE-Programm} &&= \text{Definition durch } \mu\text{-Rekursion}\end{aligned}$$

- **Somit gilt, dass alle Programmiersprachen, die über eine WHILE-Schleife verfügen, wie C/C++, Java, PASCAL usw. zu dieser Äquivalenz-Liste hinzugefügt werden können.**
- **Die Klasse von Funktionen, die auf diese Weise berechenbar sind = berechenbaren Funktionen.**

These von Church

Klasse der intuitiv berechenbaren Funktionen = Klasse der (Turing-) berechenbaren Funktionen

Jedes fundierte Präzisieren des Algorithmus-Begriffs führt zur gleichen Klasse berechenbarer Funktionen

Berechenbarkeitskonzepte

Chomsky-Hierarchie

Chomsky-Hierarchie, wobei die Allgemeingültigkeit in Tabelle von oben nach unten abnimmt:

Klasse	Grammatiken	Sprachen	Minimaler Automat
Typ 0	uneingeschränkt	rekursiv (Seite 664)	Turingmaschine (Seite 654)
Typ 1	kontextsensitiv	kontextsensitiv	Linear beschränkt
Typ 2	kontextfrei	kontextfrei (Seite 634ff)	Kellerautomat (Seite 634ff)
Typ 3	regulär	regulär (Seite 624ff)	Endlicher Automat (Seite 624ff)
	Jeder Typ ist eine Teilmenge des darüberliegenden Typs		

Prinzipiell unlösbare Probleme

Prinzipiell unlösbare Probleme:

- **Es lässt sich zwar ein Algorithmus angeben, aber dennoch sind diese nicht mit einem Computer lösbar.**

- **Probleme, die niemals durch menschliche Erfindungskraft, wissenschaftlichen und technischen Fortschritt lösbar sind.**

Prinzipiell unlösbare Probleme

Entscheidbare Mengen

Menge M heißt rekursiv entscheidbar oder entscheidbar, wenn es Algorithmus gibt, der nach endlicher Zeit terminiert und entscheidet, ob die Eingabe zur Menge gehört oder nicht.

Formal lässt sich dies wie folgt ausdrücken:

Eine Teilmenge der natürlichen Zahlen $M \subseteq \mathcal{N}$ heißt entscheidbar, wenn es eine total berechenbare Funktion f gibt mit:

$$f: \mathcal{N} \to \{0,1\}, \qquad f(x) = \begin{cases} 1, & \text{wenn } x \in M \\ 0, & \text{wenn } x \notin M \end{cases}$$

Entscheidbare Mengen sind z.B.: alle endlichen Mengen, Komplemente endlicher Mengen, die Menge aller geraden Zahlen und die Menge aller Primzahlen.

Prinzipiell unlösbare Probleme

Entscheidbare Mengen

Die Goldbach-Vermutung lautet z.B.:

„Jede gerade Zahl größer als 2 lässt sich als Summe zweier Primzahlen darstellen.“

- **Für Informatiker ist Goldbach-Problem gelöst, denn Frage: „Gibt es einen Algorithmus, der bezüglich einer beliebigen geraden natürlichen Zahl entscheidet, ob sie die Goldbach-Eigenschaft besitzt?“ ist mit „ja“ beantwortbar.**
- **Für Mathematiker nicht, denn er hätte gerne die Frage beantwortet, ob es gerade Zahlen gibt, die die Goldbach-Eigenschaft nicht besitzen.**

Prinzipiell unlösbare Probleme

Semi-entscheidbare Mengen

- Allgemeinere Klasse als die entscheidbaren Mengen sind **rekursiv aufzählbare bzw. semi-entscheidbare Mengen**, bei denen in endl. Zeit Frage, ob ein Element zu diesen Mengen gehört, nur mit „ja“ oder nur mit „nein“ zu beantworten ist.
- **Eine dieser beiden Antworten kann hierbei für bestimmte Eingaben nicht in endlicher Zeit gegeben werden.**

Beispiel sind die "wundersamen" Zahlen (3n+1-Problem):

$$n_{i+1} = \begin{cases} 3n_i + 1\,, & \text{wenn } n_i \text{ ungerade} \\ \frac{n_i}{2}\,, & \text{wenn } n_i \text{ gerade} \end{cases}$$

Startzahl heißt wundersam, wenn diese Folge (Collatz-Folge) mit Zahl 1 endet, andernfalls ist die Startzahl unwundersam.

Prinzipiell unlösbare Probleme

Semi-entscheidbare Mengen

Wundersame Zahlen (3n+1-Problem):

6: 3, 10, 5, 16, 8, 4, 2, 1
7: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8: 4, 2, 1
9: 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
100009: 300028, 150014, 75007, 225022, 112511, 337534, ... 4, 2, 1

→ semi-entscheidbar: antwortet Algorithmus nach endl. Zahl von Schritten mit „ja“ → wundersame Zahl
unwundersame Zahl → kein Ende (keine Entscheidung)

→ Bis heute wurde noch keine unwundersame Zahl gefunden.

Es gilt jedenfalls:

Jede entscheidbare Menge ist auch semi-entscheidbar.

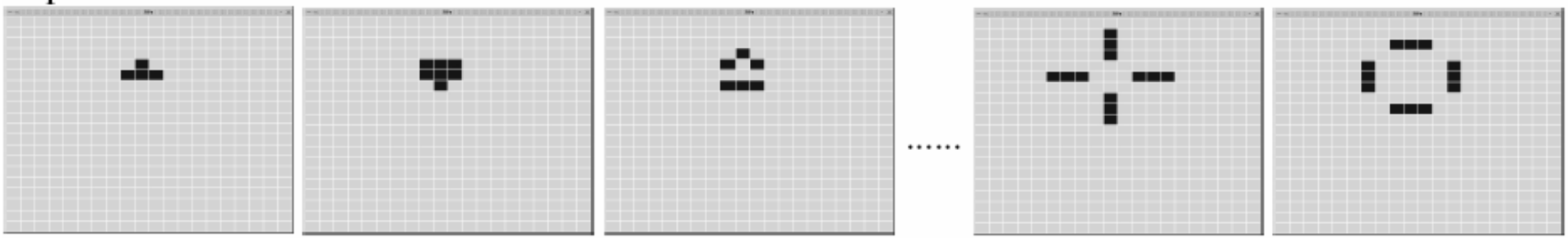
Prinzipiell unlösbare Probleme

Semi-entscheidbare Mengen

Beispiel für semi-entscheidbare, aber nicht entscheidbare Menge → Game of Life

- **Geburt**
  Tote Zelle wird geboren, wenn 3 ihrer acht Nachbarn leben.
- **Tod durch Überbevölkerung**
  Zelle stirbt, wenn vier oder mehr Nachbarn leben.
- **Tod durch Vereinsamung**
  Zelle stirbt, wenn sie keinen oder nur 1 lebenden Nachbar.

Populationen



zu Beginn | nach 1. Schritt | nach zwei Schritten | später sich ständig wiederholende Muster

Prinzipiell unlösbare Probleme

Semi-entscheidbare Mengen

Beispiel für semi-entscheidbare, aber nicht entscheidbare Menge → Game of Life

Algorithmus gesucht, der für beliebige Figur (Population) entscheidet, ob diese unsterblich ist oder nicht.

- **Algorithmus, der Schritt für Schritt nachvollzieht, kann Aussterben oder period. Wiederh. feststellen (entscheiden).**
- **Tritt keiner dieser Fälle ein → kein Ende (k. Entscheidung).**
- **Abstrakterer Algorithmus ??, der Ergebnis vieler Schritte ermittelt, ohne dass er alle Einzelschritte durchlaufen muss**
  → wurde bewiesen, dass es solchen Algorithmus nicht gibt
  → nicht entscheidbar, ob Population ausstirbt bzw. wegen sich ständig wiederholender Muster ewig lebt.

➔Es gibt semi-entscheidbare Mengen, die nicht entscheidbar

Prinzipiell unlösbare Probleme

Das Halteproblem

Halteproblem = Beispiel für semi-entscheidbares, aber nicht entscheidbares Problem, das auf Alan M. Turing zurückgeht.

- **Gibt es ein Programm, das als Eingabe den Quelltext eines zweiten Programms sowie dessen Eingabewerte erhält, und dann entscheiden kann, ob das zweite Programm terminiert, d.h. nicht endlos weiterläuft?**

→ Durch Widerspruchsbeweis lässt sich eindeutig zeigen, dass eine solche Turingmaschine und damit auch folglich kein solches Programm jemals existieren kann.

Prinzipiell unlösbare Probleme

Das Halteproblem

Angenommen, es gibt eine Funktion haltetest():

```
string haltetest(Programm,Eingabe) {
    if (Programm(Eingabe) terminiert)
        return JA;
    else
        return NEIN;
}
```

dann lässt sich daraus folgende Funktion test() bilden:

```
void test(Programm) {
    while (haltetest(Programm, Programm) == JA)
        ; /* leere Anweisung */
    /* Solange das Programm bei Eingabe seiner eigenen Kodierung */
    /* terminiert, terminiert die Funktion test() nicht.        */
}
```

Prinzipiell unlösbare Probleme

Das Halteproblem

Übergibt man d. Funktion **test()** sich selbst als Eingabedaten, um sie sich somit von **haltetest()** auf Terminierung zu prüfen:

```
test ( test );  /* Dieser Aufruf terminiert nun genau dann, */
                /* wenn er nicht terminiert. (Widerspruch!) */
```

kann diese kein richtiges Ergebnis liefern, denn nun gilt:

- Liefert **haltetest(test, test)** JA → **test(test)** terminiert, aber dann ist Bedingung **haltetest(Programm, Programm)** in **test()** immer wahr, so dass **test(test)** eben nicht terminiert, weil while-Schleife nie beendet wird → Widerspruch!
- Liefert **haltetest(test, test)** NEIN → Bed. der while- Schleife nie wahr und **test(test)** terminiert sofort →auch Widerspruch!

Prinzipiell unlösbare Probleme

Das Halteproblem

→ Bedeutet, dass es keine Turingmaschine geben kann, die, wenn sie als Eingabe Kodierung einer Turingmaschine M und zugehörige Eingabe w erhält, JA ausgibt, wenn M auf w hält, und NEIN ausgibt, wenn M nicht auf w hält.

→ Es gibt aber Turingmaschine, die immer dann JA ausgibt, wenn M auf w hält, aber endlos arbeitet, wenn M nicht hält. Sie muss nur Berechnung der Turingmaschine simulieren und JA ausgeben, nachdem Simulation eventuell hält.
Halteproblem deshalb semi-entscheidbares Problem.

In jedem turingmächtigen System (z.B. Arithmetik) sind Aussagen formulierbar, die weder beweis- noch widerlegbar sind. Solche Systeme sind grundsätzlich unvollständig.

Prinzipiell unlösbare Probleme

Das Halteproblem

Es gibt also Funktionen, die zwar wohldefiniert sind, deren Wert sich aber trotzdem i.A. nicht berechnen lässt.

Setzt man nun die Church'sche These als wahr voraus, so können Maschinen und letztlich Menschen das Halteproblem (und viele andere Probleme) grundsätzlich nicht lösen.

→ Für die SW-Entwicklung bedeutet das Halteproblem, dass i.A. eine automat. Überprüfung einer Programmlogik nicht möglich ist.

→ Insbesondere ist es nicht immer möglich durch Rechner festzustellen, ob ein gegebenes Programm jemals anhalten wird.

Prinzipiell unlösbare Probleme

Unberechenbarkeit (Fleißiger Biber)

Fleißiger Biber (busy beaver) ist eine Turingmaschine mit n Zuständen, einem Endzustand und einem Alphabet mit zwei Werten {0, 1}, die eine max. Anzahl k_n von Einsen auf leeres (mit Nullen beschriebenes) Band schreibt und dann anhält.

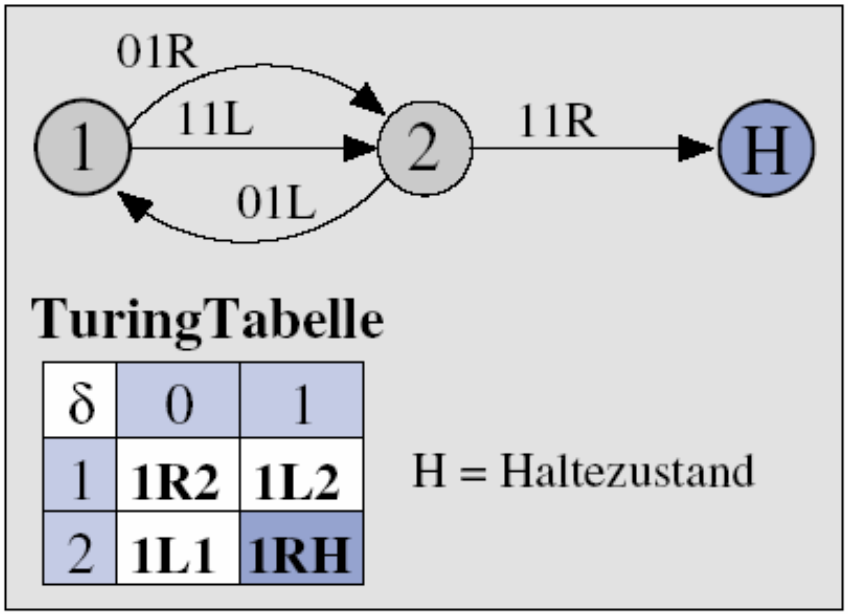
Keine andere Turingmaschine mit gleicher Anzahl von Zuständen und gleichem Alphabet kann mit mehr Einsen als ein fleißiger Biber auf dem Band anhalten.

Über Zahl k_n von Einsen, die fleißiger Biber mit n Zuständen schreibt, definiert man den Wert der **Busy-Beaver-Funktion** (auch Rado-Funktion genannt) an der Stelle n : $\Sigma(n) = k_n$

Prinzipiell unlösbare Probleme

Unberechenbarkeit (Fleißiger Biber)

Turingmaschine mit 2 Zuständen, welche max. mögliche Zahl von vier Einsen schreibt. Das Schreiben der vier Einsen der Turingmaschine geschieht dabei wie rechts gezeigt:



TuringTabelle

δ	0	1
1	**1R2**	**1L2**
2	**1L1**	**1RH**

H = Haltezustand

Schritt	Zustand	Bandinhalt (*Kopfpos.*)					
0	1	.	.	.		0	
1	2	.	.	.		1	0
2	1	.	.	.		1	1
3	2	.	.	.	0	1	1
4	1	.	.	0	1	1	1
5	2	.	.	1	1	1	1
6	H	.	.	1	1	1	1

Prinzipiell unlösbare Probleme

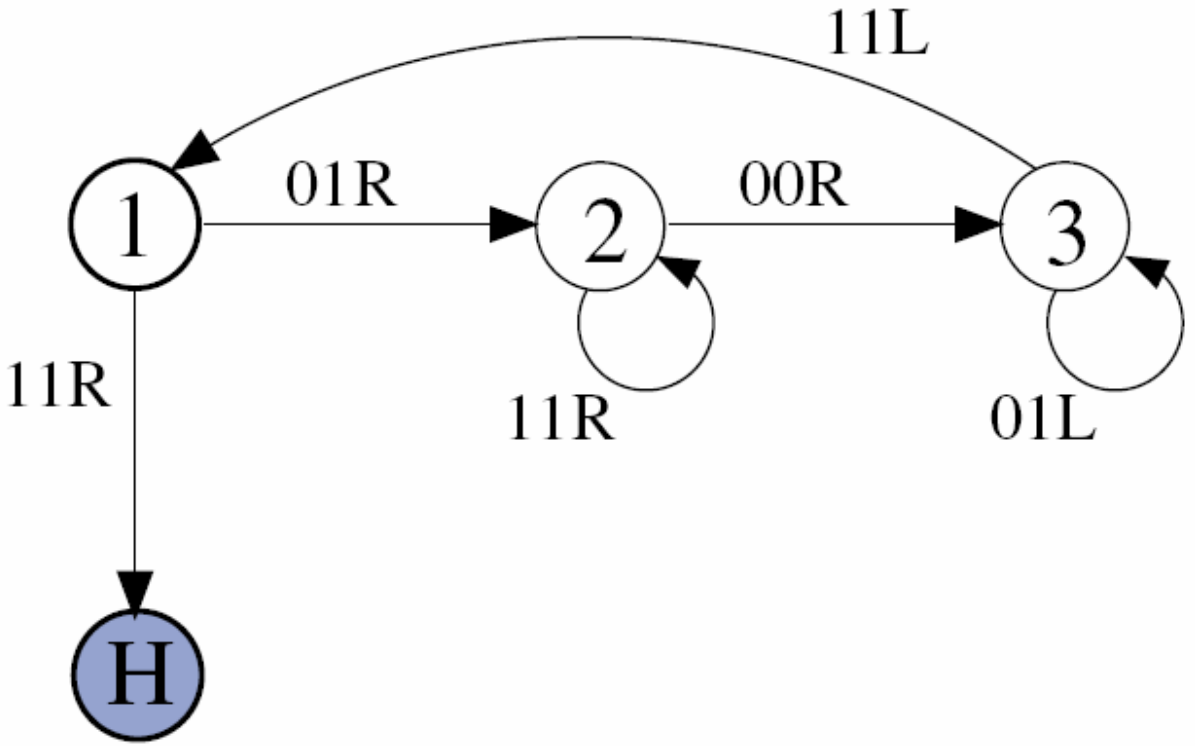
Unberechenbarkeit (Fleißiger Biber)

Turingmaschine mit drei Zuständen, welche die maximal mögliche Zahl von sechs Einsen schreibt:

TuringTabelle

δ	0	1
1	**1R2**	**1RH**
2	**0R3**	**1R2**
3	**1L3**	**1L1**

H = Haltezustand



Prinzipiell unlösbare Probleme

Unberechenbarkeit (Fleißiger Biber)

Busy beaver → nicht lösbares Problem, da nicht allgemein entscheidbar, ob gegebene Turingmaschine mit n Zuständen tatsächlich Folge von 1 mit max. Länge schreibt oder nicht.

→ **Menge der Werte von Σ(n) ist nicht entscheidbar, obwohl Σ(n) wohldefiniert ist → Funktion Σ(n) ist nicht berechenbar.**

→ **Zudem wächst Σ(n) stärker als jede berechenbare Funktion, sogar stärker als jede Exponentialfunktion. Für n > 5 sind keine Aussagen über den Wert von Σ(n) mehr möglich.**

→ **Man müsste für jede einzelne Turingmaschine mit n Zuständen herausfinden, nach wie vielen Schritten sie hält und evtl. beweisen, dass sie nicht anhält.**
Aus Gödels Unvollständigkeitssatz folgt, dass kein Beweis für alle solchen Turingmaschinen existiert.

Prinzipiell unlösbare Probleme

Unberechenbarkeit (Fleißiger Biber)

Anwachsen der Beasy-Beaver-Funktion $\Sigma(n)$

n	Anzahl der Maschinen	$\Sigma(n)$, (Jahr)
1	144	1
2	104976	4
3	$> 10^8$	6
4	$> 10^{11}$	13
5	$> 10^{15}$	$\geq$ 1915 (1984), $\geq$ 4098 (1989)
6	$> 10^{19}$	$> 1,29 \cdot 10^{865}$
7	$> 10^{23}$	keine realistische Abschätzung mehr möglich