

# Berechenbarkeit

# Motivation

- ▶ Endliche Automaten erkennen nicht alle algorithmisch erkennbaren Sprachen.
  - ▶ Kontextfreie Grammatiken erzeugen nicht alle algorithmisch erzeugbaren Sprachen.
- 
- ▶ Welche Berechnungsmodelle erlauben die Berechnung aller algorithmisch beschreibbaren Sprachen?
  - ▶ Was können allgemeinere Berechnungsmodelle berechnen?
  - ▶ Wo sind die Grenzen?
  - ▶ Was heisst überhaupt Berechenbarkeit?

# Berechenbarkeitsmodelle

- ▶ präzisieren den Begriff der **berechenbaren Funktionen**
- ▶ **Syntax**: Sprache zum Beschreiben von Algorithmen
- ▶ **Semantik**: Kalkül zur Ausführung der Algorithmen

---

## Beispiele für Berechenbarkeitsmodelle

- ▶ Spezielle Programmier- oder Spezifikationssprachen
- ▶ Turingmaschinen
- ▶ Typ-0-Grammatiken
- ▶ Registermaschinen,  $\lambda$ -Kalkül,  $\mu$ -rekursive Funktionen

# PASCALchen

- ▶ Kleine imperative Programmierspache
- ▶ Besteht aus *while*-Programmen
- ▶ Einziger Datentyp: natürliche Zahlen
- ▶ Keine Rekursion

# Syntax (1)

## Zeichen in PASCALchen

- ▶ Variablen:  $X_n$  mit  $n \in \mathbb{N}$
- ▶ Operatoren:  $succ$ ,  $pred$ ,  $0$
- ▶ Zuweisungssymbol:  $:=$
- ▶ Schlüsselwörter:  $begin$ ,  $end$ ,  $while$ ,  $do$
- ▶ Hilfssymbole:  $(, ), ;$

## Syntax (2)

- Ein *while-Programm* ist eine Reihung.

$$\langle prog \rangle ::= \langle comp \rangle$$

- Eine *Reihung* hat die Form  $begin\ S_1; S_2; \dots; S_m\ end$ , wobei  $S_1, \dots, S_m$  für  $m \geq 0$  Anweisungen sind.

$$\begin{aligned} \langle comp \rangle &::= begin\ \langle stmtlist \rangle\ end \mid begin\ end \\ \langle stmtlist \rangle &::= \langle stmt \rangle \mid \langle stmt \rangle; \langle stmtlist \rangle \end{aligned}$$

- Eine **Anweisung** ist eine Reihung, eine Zuweisung oder eine Wiederholung

$$\langle stmt \rangle ::= \langle comp \rangle \mid \langle assign \rangle \mid \langle while \rangle$$

- Eine **Zuweisung** hat die Form  $X_i := 0$ , oder  $X_i := succ(X_j)$  oder  $X_i := pred(X_j)$ , wobei  $X_i, X_j$  zwei Variablen sind.

$$\begin{aligned} \langle assign \rangle &::= \langle var \rangle := \langle expr \rangle \\ \langle expr \rangle &::= 0 \mid succ(\langle var \rangle) \mid pred(\langle var \rangle) \end{aligned}$$

- Eine **Wiederholung** hat die Form  $\text{while } X_i \neq X_j \text{ do } S$ , wobei  $S$  eine Anweisung ist und  $X_i, X_j$  zwei Variablen.

$$\langle \text{while} \rangle ::= \text{while } \langle \text{var} \rangle \neq \langle \text{var} \rangle \text{ do } \langle \text{stmt} \rangle$$

- **Variablen** haben die Form  $X_n$  mit  $n \in \mathbb{N}$ .

$$\langle \text{var} \rangle ::= X \langle \text{nat} \rangle$$
$$\langle \text{nat} \rangle ::= 0 \mid \langle \text{one} - \text{nine} \rangle \langle \text{cipherseq} \rangle$$
$$\langle \text{cipherseq} \rangle ::= \lambda \mid \langle \text{cipher} \rangle \langle \text{cipherseq} \rangle$$
$$\langle \text{cipher} \rangle ::= 0 \mid \langle \text{one} - \text{nine} \rangle$$
$$\langle \text{one} - \text{nine} \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$



# Beispiel

```
begin  
   $X1 := succ(X2);$   
   $X1 := pred(X1)$   
end
```

Weist der Variable  $X1$  den Wert von  $X2$  zu.

# Makros

*while*-Programme sind Anweisungen, die in anderen *while*-Programmen verwendet werden können.

$$\left. \begin{array}{l} \textit{begin} \\ \quad X1 := \textit{succ}(X2); \\ \quad X1 := \textit{pred}(X1) \\ \textit{end} \end{array} \right\} X1 := X2$$

*begin*  
     $X4 := 0$ ;  $X1 := X2$ ;  
    *while*  $X4 \neq X3$  *do begin*  
         $X4 := succ(X4)$ ;  
         $X1 := succ(X1)$   
    *end*  
*end*

}  $X1 := X2 + X3$

$$\left. \begin{array}{l} \textit{begin} \\ \quad Z := 0; V := 0; \\ \quad \textit{while } V \neq Y \textit{ do begin} \\ \qquad \quad Z := Z + X; \\ \qquad \quad V := \textit{succ}(V) \\ \quad \textit{end} \\ \textit{end} \end{array} \right\} Z := X * Y$$

**Beachte:** Wir benennen Variablen durch beliebige Großbuchstaben, wenn die spezielle Form  $Xn$  nicht gebraucht wird.

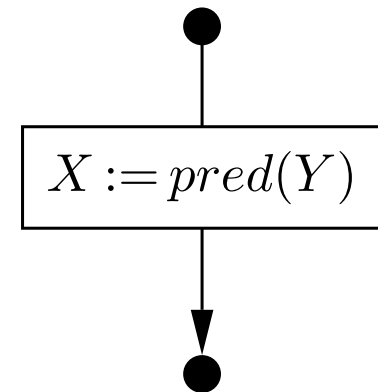
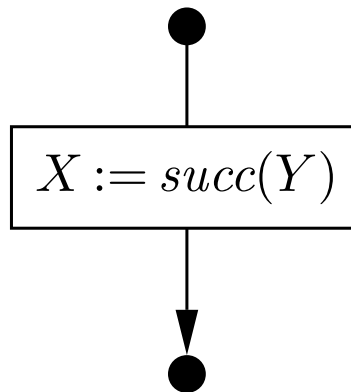
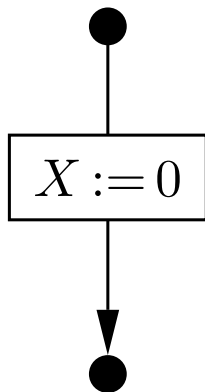
## Weitere mögliche Makros

- ▶  $X1 := n,$
- ▶  $X1 := X2 \div X3,$
- ▶  $X1 := 2^{X2},$
- ▶ *while*  $B$  *do*  $S,$
- ▶ *if-then-else,*
- ▶ *repeat-until. . .*

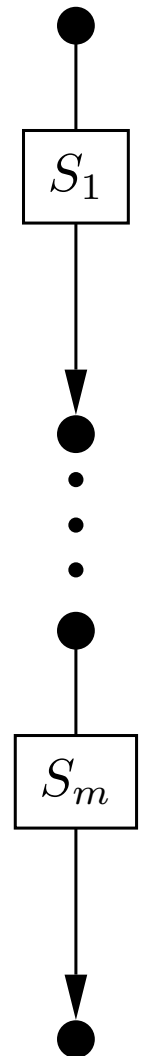
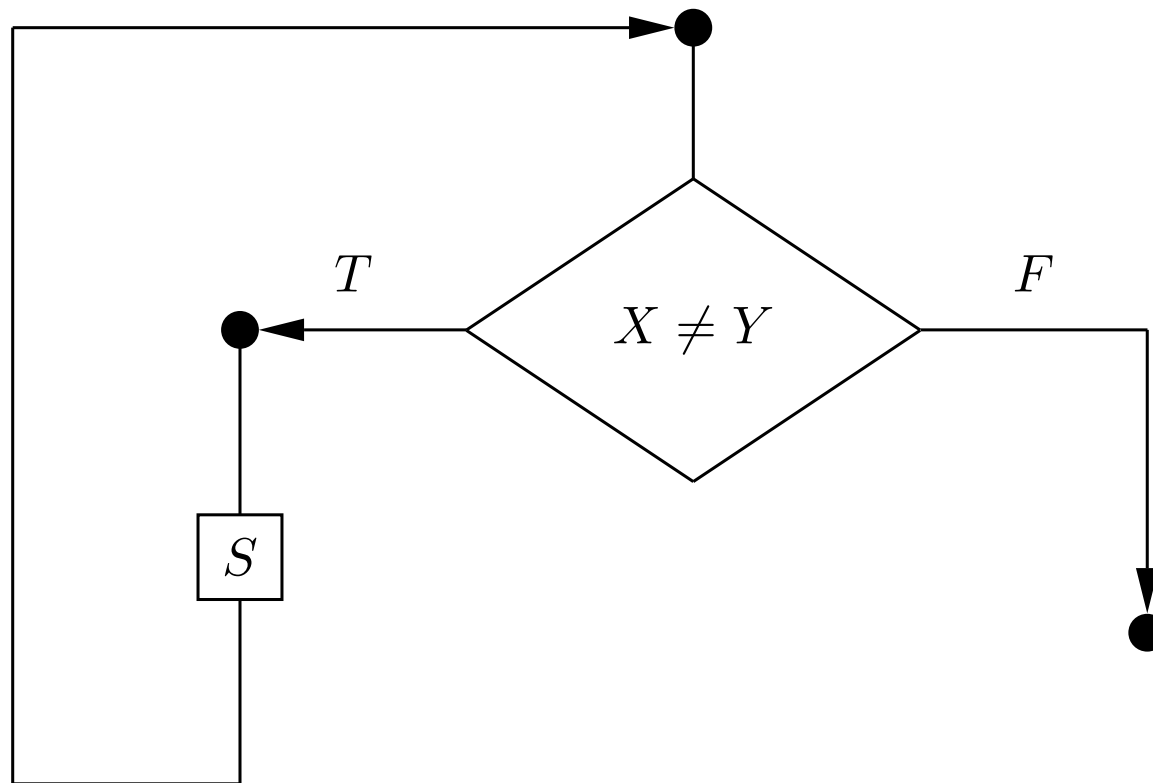
mit  $B$ : Boolescher Ausdruck, der aus Variablen, Konstanten,  $=$ ,  $\neq$ ,  $<$ , *and*, *or*, *not* aufgebaut ist.

# Flussdiagramme

- Zuweisungen



- Wiederholung und Reihung



# Semantik (operationell)

Ein *while*-Programm wird ***k*-variabel** genannt, wenn höchstens die Variablen  $X_1, \dots, X_k$  darin vorkommen.

Ein **Berechnungszustand** eines *k*-variablen *while*-Programms ist ein *k*-dimensionaler Vektor über den natürlichen Zahlen

$$a = (x_1, \dots, x_k) \in \mathbb{N}^k,$$

der auch **Zustand** oder **Zustandsvektor** genannt wird.



# Berechnung

$$a_0 A_1 a_1 A_2 a_2 \cdots a_{i-1} A_i a_i A_{i+1} a_{i+1} \cdots$$

- $a_i$ : Berechnungszustände
- $a_0$  wird **Eingabe** genannt
- $A_i$ : **Instruktionen** (d.h. Tests oder Zuweisungen)
- die im Folgenden genannten Bedingungen müssen erfüllt sein.

## Bedingungen für $a_0A_1a_1A_2a_2\cdots$ (1)

- Es gibt einen Weg durch das zugehörige Flussdiagramm, der beim Eingang beginnt und genau die Instruktionen  $A_1, A_2, \dots, A_i, \dots$  in dieser Reihenfolge durchläuft.
- $a_0$  ist frei wählbar.
- $A_i = Xu \neq Xv \implies a_i = a_{i-1}$ .

Bedingungen für  $a_0 A_1 a_1 A_2 a_2 \cdots$  (2)

$$A_i = \left\{ \begin{array}{l} Xu := 0 \\ Xu := succ(Xv) \\ Xu := pred(Xv) \end{array} \right\} \text{ und } a_{i-1} = (x_1, \dots, x_k)$$

$\Rightarrow$

$$a_i = (x_1, \dots, x_{u-1}, \left\{ \begin{array}{c} 0 \\ succ(Xv) \\ pred(Xv) \end{array} \right\}, x_{u+1}, \dots, x_k).$$

## Bedingungen für $a_0A_1a_1A_2a_2\cdots$ (3)

- $A_i$ : letzte Instruktion der Berechnung  $\implies$  hinter  $A_i$  ist der Ausgang.
- $A_i = Xu \neq Xv$  und  $a_{i-1} = (x_1, \dots, x_k)$ 
  - ▶  $x_u \neq x_v \implies A_{i+1}$ : erste Instruktion nach der herausführenden  $T$ -Kante.
  - ▶  $x_u = x_v, \implies A_i$  ist die letzte Instruktion der Berechnung, oder  $A_{i+1}$  ist die erste Instruktion nach der herausführenden  $F$ -Kante.

# Endliche Berechnung

$$a_0 A_1 a_1 \cdots a_{n-1} A_n a_n \quad (n \geq 0)$$

- ▶  $n$ : Länge der Berechnung
- ▶  $a_n$ : Ausgabe
- ▶  $n = 0 \implies$  Flussdiagramm enthält keine Instruktion

# Beobachtung

Zu jedem  $k$ -variablen *while*-Programm und jedem Berechnungszustand  $a_0$  gibt es genau eine Berechnung mit  $a_0$  als Anfang.

# Semantikfunktion für *while*-Programme

Sei  $P$  ein  $k$ -variables *while*-Programm und  $j \in \mathbb{N}$ . Dann ist die  $j$ -stellige **Semantikfunktion** von  $P$

$$SEM_P: \mathbb{N}^j \rightarrow \mathbb{N}$$

für die Argumente  $(x_1, \dots, x_j) \in \mathbb{N}^j$  nach folgenden Regeln definiert:

**Regel 1:** Aus den Argumenten  $(x_1, \dots, x_j)$  wird eine Eingabe  $a \in \mathbb{N}^k$  hergestellt, wobei  $a = (x_1, \dots, x_k)$  ist für  $j \geq k$  und  $a = (x_1, \dots, x_j, 0, \dots, 0)$  mit  $k - j$  Nullen für  $j < k$ .

**Regel 2:**  $P$  wird mit Eingabe  $a$  berechnet.

**Regel 3:** Terminiert die Berechnung mit der Ausgabe  $(y_1, \dots, y_k)$ , so ist  $SEM_P(x_1, \dots, x_j) = y_1$ .

**Regel 4:** Terminiert sie nicht, ist  $SEM_P(x_1, \dots, x_j)$  undefiniert.



## Bemerkungen

1. Semantikfunktion total, falls jede Berechnung terminiert.
2. Wahlfreiheit von  $j \implies$  Jedes Programm berechnet unendlich viele Funktionen.
3. Statt  $SEM_P$  kann auch  $SEM_P^{(j)}$  geschrieben werden.

# Berechenbare Funktion

Eine partielle Funktion  $f: \mathbb{N}^j \rightarrow \mathbb{N}$  heißt **berechenbar**, wenn ein *while*-Programm existiert mit

$$f = SEM_P^{(j)}.$$

# Churchsche These

Jede partielle Funktion  $f: \mathbb{N}^j \rightarrow \mathbb{N}$ , die durch irgendeinen Mechanismus oder auf Grund irgendeiner Überlegung algorithmisch berechnet werden kann, ist bereits berechenbar (durch ein *while*-Programm).

# Programme als Eingaben für Programme

- ▶ Verhalten eines Programms hängt vom eingegebenen Programm ab.

Beispiel: Interpreter für PASCALchen



# Repräsentation von *while*-Programmen als natürliche Zahlen (Gödelnumerierung)

1. Umwandlung der Zeichen, aus denen *while*-Programme bestehen, in Bitmuster
  - ▶ Der Zeichensatz  $A$  von PASCALchen besteht aus 22 Zeichen.
  - ▶ Sei  $code: A \rightarrow \{0, 1\}^*$  eine injektive Abbildung mit  $code(a) = 1b_1 \cdots b_5$

$code(a)$  liefert eine eindeutige 6-Bit-Darstellung von  $a$ , die mit 1 beginnt.

## 2. Repräsentation von *while*-Programmen als Bitmuster

$code^*: A^* \rightarrow \{0, 1\}^*$  mit

(i)  $code^*(\lambda) = \lambda$  und

(ii)  $code^*(av) = code(a)code^*(v)$  für  $a \in A$ ,  $v \in A^*$ .

Für jedes *while*-Programm  $P$  liefert  $code^*(P)$  ein Bitmuster.

## 3. Umwandlung von *while*-Programmen in natürliche Zahlen

Jedes Bitmuster lässt sich eindeutig als Binärdarstellung einer natürlichen Zahl auffassen.

# Injektivität der Gödelnumerierung

## Lemma

Die Abbildung  $code^*$  ist injektiv.