Theoretische Informatik

Sprachen und Sprachanalyse

Berechenbarkeit

Komplexität

Der Aufbau von Sprachen

- Sprachen sind in mehreren Schichten aufgebaut

1. Zeichen bzw. Symbole
   - Endliche Menge, z.B.:
     - Lateinisch = { **a, b, c, …** }
     - Telugu = {అ,ఒ,త,మ,హా,ఱూ,ళ,ఴ,...},
     - Kyrillisch = {Д,Ж,Й,И,Л,Б,*С*,...},
     - Arabisch = {ز,ف,ي,١ , ٢, ٣, ...}

2. Worte
   - Aus Zeichen gebildet
     - endliche Länge
     - unendlich viele Möglichkeiten
   - Korrektheit meist leicht erkennbar
     - Wörterbuch, Beugung, Konkatenation

3. Sätze
   - Aus Worten gebildet
   - Regeln durch Grammatik beschrieben
   - Korrektheit schwieriger zu erkennen
     - *„Der Dativ ist dem Genitiv sein Tod.“*

Programmiersprachen

1. Zeichen
   - a,b,…, z, A,…, Z,0,1,…,9,+,-,*,/,“, $, =, <, (,), …
2. Worte
   - Schlüsselworte
     - **if, then, while, exit, case, …**
   - Bezeichner
     - **betrag, summe, fact, print, Stack, int**
   - Wertliterale
     - int-Konstante,
       - **31, -2005, +181**
     - float-Konstante,
       - **3.14, 0.5E-12, -.12**
     - String-Konstante, …
       - **“Otto“, “Das ist ein \n zweizeiliger Text“**
   - Symbole
     - **:=, ==, (,), <=, ++, +, +=, …**
3. Sätze
   - Programme
     - **“public static void main (String[] args){**
       System.out.println …. “

Analyse von Programmtexten

- Eingabe:
  - Programm als Textstring (Folge von Zeichen).
  - "**PROGRAM ggT; BEGIN x:=54;y:=30;WHILE not x=y DO IF x>y THEN x:=x-y;ELSE y:=** …"
- Ausgabe:
  - Entscheidung ob das Programm syntaktisch korrekt ist
    - Ggf: Fehlerhinweis
  - Analyse des Programms
    - Wie sind die Teile geschachtelt
    - Kontrollfluss
  - Übersetzung in einfachere Sprache
    - Maschinensprache
    - Code für VM

Drei Phasen eines Compilers

1. Lexikalische Analyse
   - *Scanner*
   - Zerlegung des Programmtextes in Worte

2. Syntaktische Analyse
   - *Parser*
   - Erkennung der Satzstruktur
   - Interne Repräsentation des Programms als Baum

3. Codeerzeugung
   - Übersetzung anhand einer Baumtraversierung

Aufgabe des Scanners

- Zerlegt Programmtext in Worte
  - nach welchen Kriterien ?
- klassifiziert Worte in einem Programmtext
  - Schlüsselwort, Zahlkonstante, Bezeichner, …
- Wortklassen werden durch *Token* repräsentiert
  - **`187, 3.14, -1.18E-12, 0.5`** ⇒ *num*
  - **`betrag, zins, x`** ⇒ *id*
  - `:=` ⇒ *assignOp*
  - **`;`** ⇒ *semi*
  - **`WHILE, while, While, wHiLe`** ⇒ *while*
- Aufgabe des Scanners also:
  - Programmtext ⇒ *Liste von Token*
- In *java.util* gibt es eine Klasse *Scanner*

Scannerlauf

Programm (repräsentiert als Text)

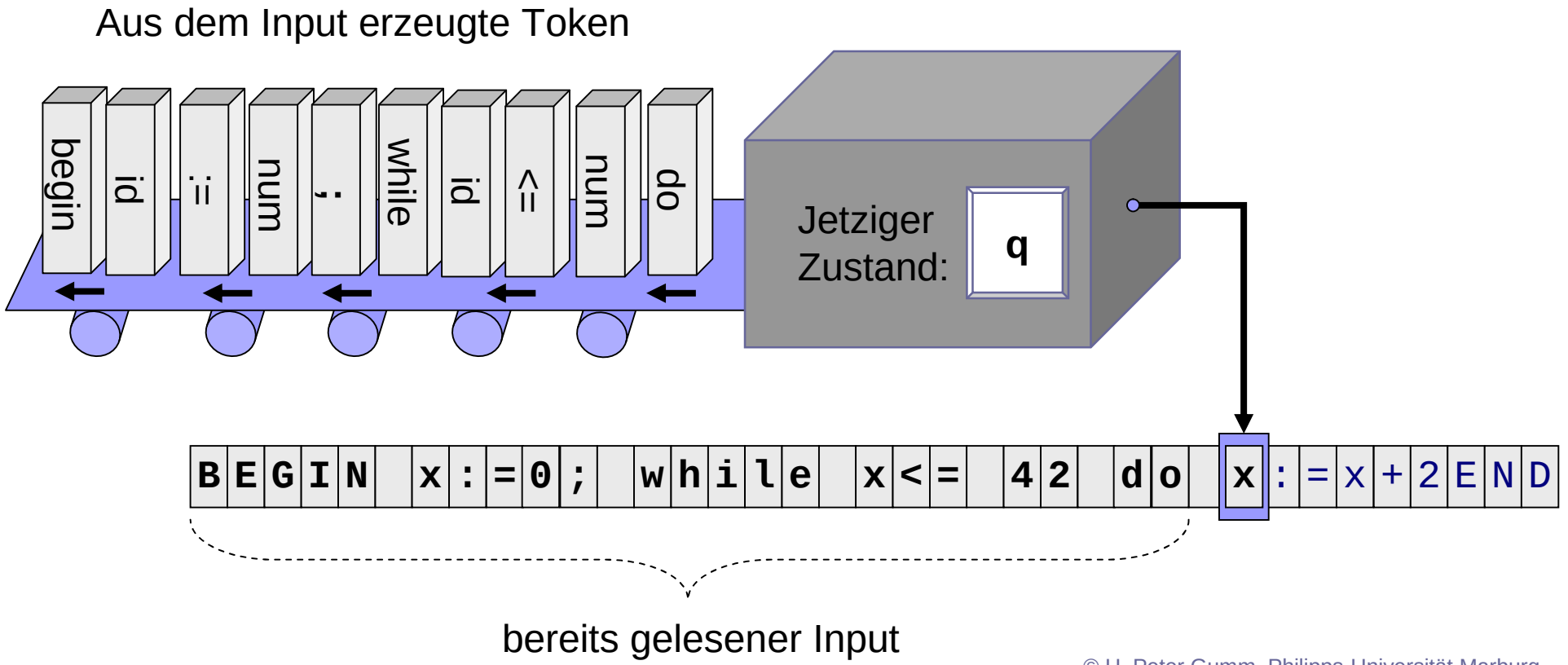
```
"PROGRAM ggT;
 BEGIN
   x:=54;
   y:=30;
       WHILE not x=y DO
           IF x>y THEN x:=x-y
               ELSE y:=y-x;
       writeln(´Das Ergebnis ist: ´,x)
 END."
```

Scanner

Tokenliste	repräsentiert z.B. als
program	17
id	23
semi	54
begin	19
id	23
assignOp	37
num	18
semi	54
id	23
assignOp	37
num	18
semi	54
while	74
…	…

Scanner als Automat

- *Scanner* ist ein Automat
  - Abhängig von Input und Zustand
    - geht er in einen neuen Zustand
    - klassifiziert einen Teilstring



Drei Phasen eines Compilers

1. Lexikalische Analyse
   - *Scanner*
   - Zerlegung des Programmtextes in Worte

2. Syntaktische Analyse
   - *Parser*
   - Erkennung der Satzstruktur
   - Interne Repräsentation des Programms als Baum

3. Codeerzeugung
   - Übersetzung anhand einer Baumtraversierung

Grammatik

- Grammatiken legen mögliche Satzstrukturen fest
- Rekursion ist erlaubt

Grammatik für Java:

block :: *{ Anweisungen }*
Anweisung :: *if (Expr) Anweisung*
| *while(Expr) Anweisung*
| *{ Anweisungen }*
| *…*

Anweisungen :: *Anweisungen Anweisung*
| */* Nix */*

Grammatik für Anfänger:

Satz :: *Subjekt Prädikat Objekt*

Subjekt :: ***artikel nomen***
| ***artikel adjektiv nomen***

Prädikat :: ***verb*** | ***hilfsverb***

Objekt :: *Subjekt*

Syntax nat. Sprache

- Gruppierung von Worten anhand einer *Grammatik*

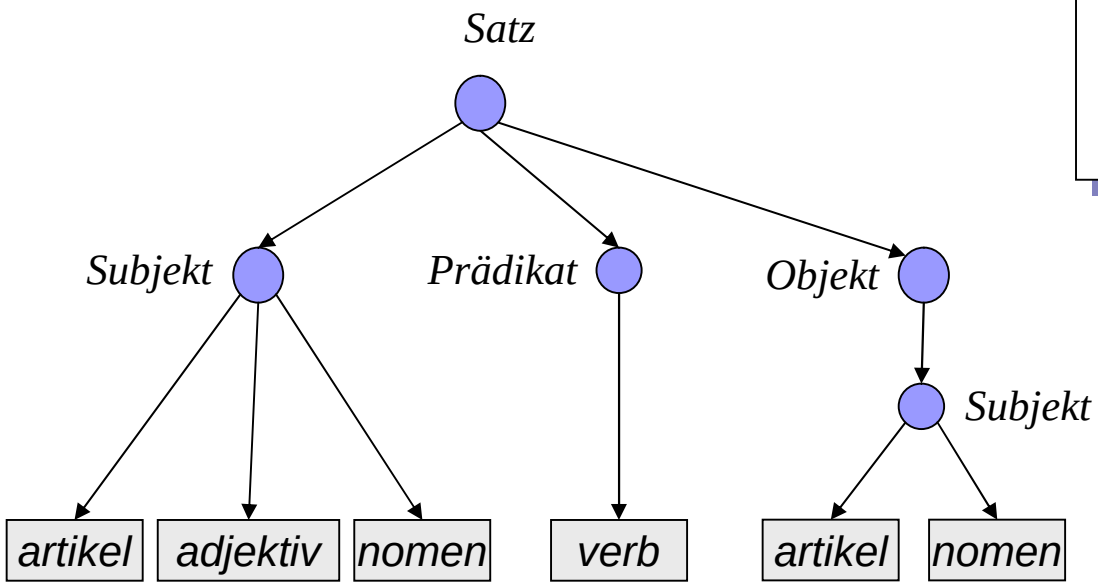
Deutsche Grammatik:

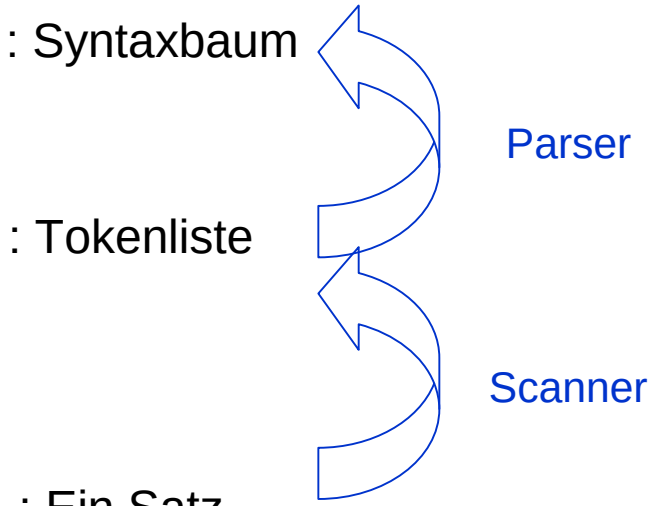
Satz :: *Subjekt Prädikat Objekt*

Subjekt :: ***artikel nomen***
| ***artikel adjektiv nomen***

Prädikat :: ***verb*** | ***hilfsverb***

Objekt :: *Subjekt*





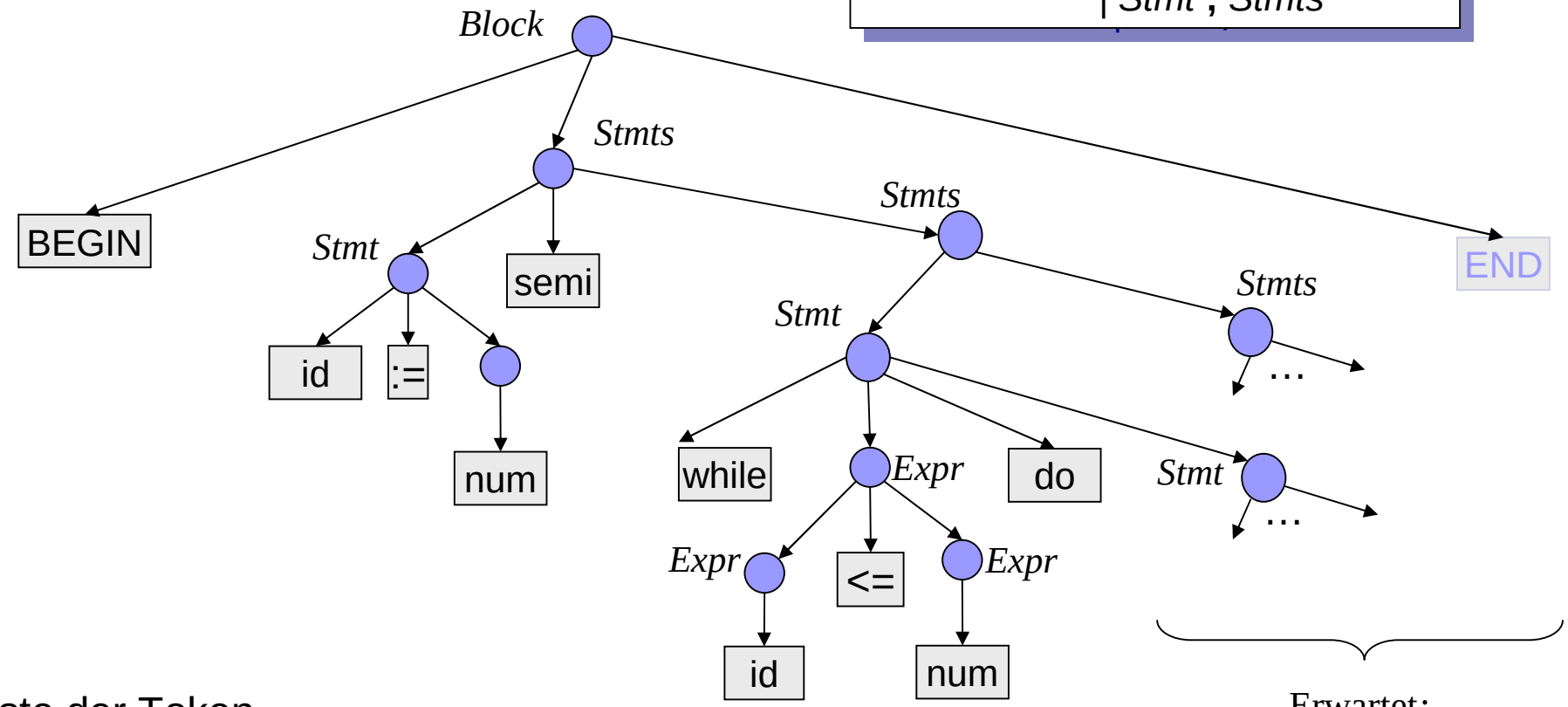
Die lila Kuh legt ein Schokoladenei

Aufgabe des Parsers

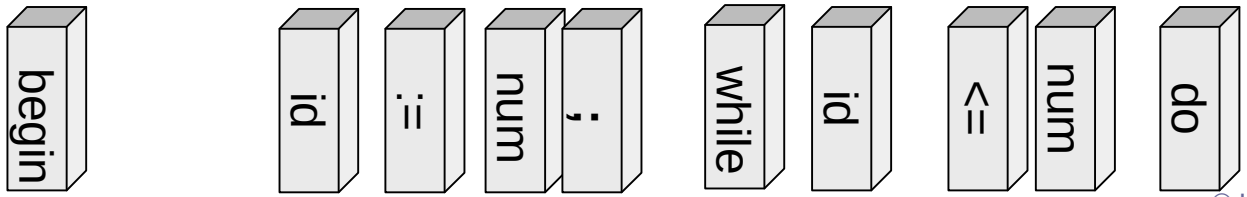
- Parser erzeugt Syntaxbaum

Grammatik für Pascal:

block	:: **Begin** *Stmts* **End**
Stmt	:: **id** := *Expr*
	| **while** *Expr* **do** *Stmt*
	| *Stmt* **;** *Stmts*



Liste der Token



Drei Phasen eines Compilers

1. Lexikalische Analyse
   - *Scanner*
   - Zerlegung des Programmtextes in Worte

2. Syntaktische Analyse
   - *Parser*
   - Erkennung der Satzstruktur
   - Interne Repräsentation des Programms als Baum

3. Codeerzeugung
   - Übersetzung anhand einer Baumtraversierung

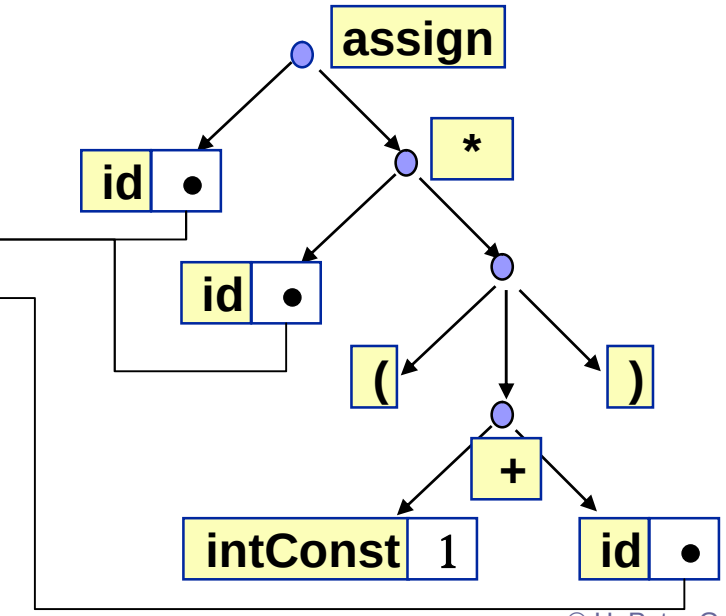
Codeerzeugung

- Aus Syntaxbaum
  - erzeuge Maschinencode
  - führe Buch über Speicherplatz für Variablen
    - Symboltabelle
- Kein Thema in dieser Vorlesung

```
betrag := betrag*(1+zins)
```

Name	Typ	Sp.Platz
betrag	float	17F4
zins	float	17F8
x	int	201C
test	boolean	C011

Parse Tree:

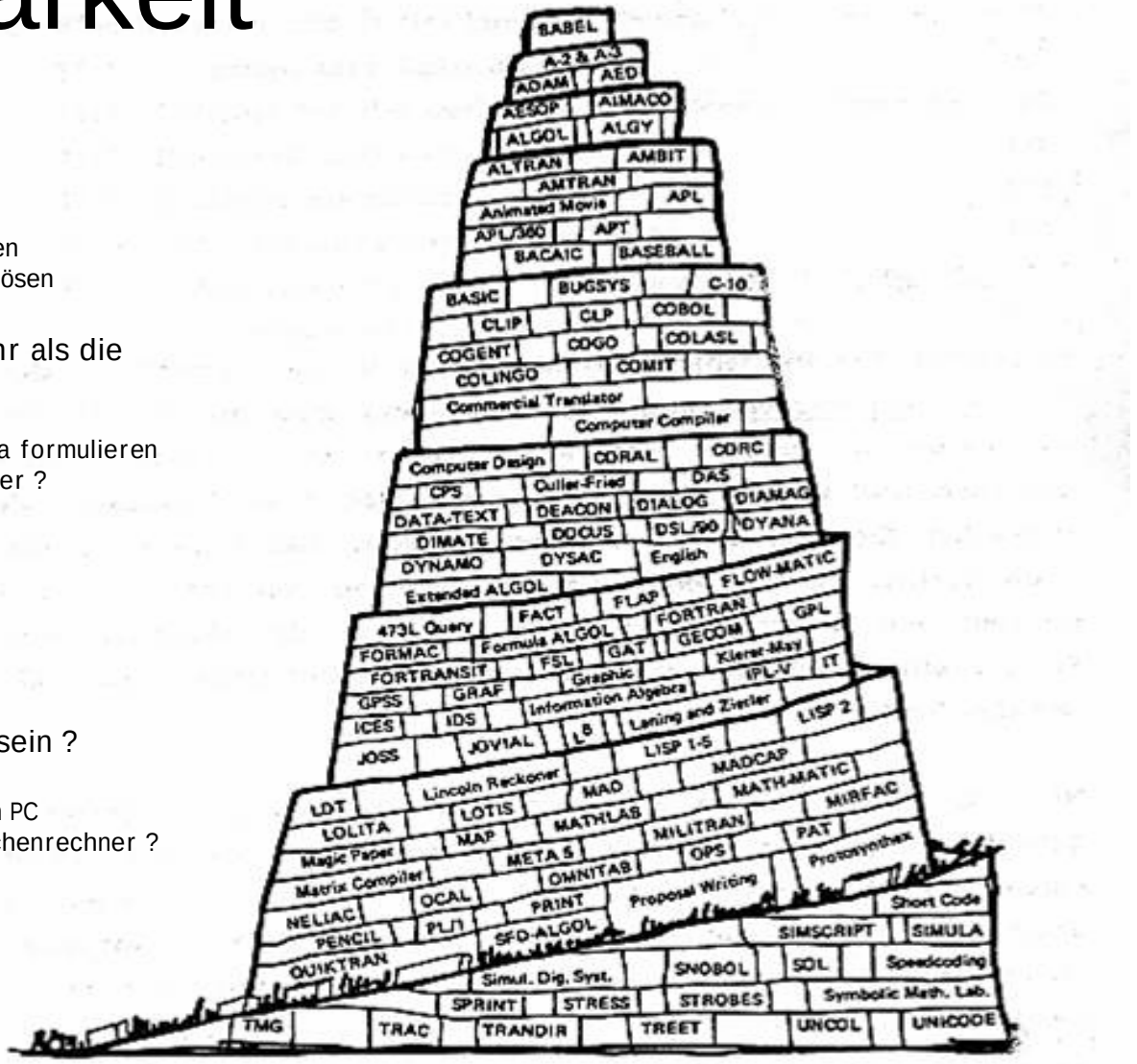


Code für einen Stackprozessor

```
LOAD*  17F4
LOAD   17F4
LOADC  1
LOAD   17F8
ADD
MULT
STORE
```

Berechenbarkeit

- Was ist ein Algorithmus
  - Was kann man mit Algorithmen lösen
  - Was kann man nicht algorithmisch lösen
- Kann eine Programmiersprache mehr als die andere ?
  - Kann man jeden Algorithmus in Java formulieren
  - in Pascal, C, Prolog, C, C++, Assembler ?
- Kann man jeden Algorithmus
  - rekursiv formulieren ?
  - braucht man Zuweisungen ?
  - braucht man Variablen ?
- Wie kompliziert muss ein Rechner sein ?
  - Kann eine Workstation mehr als ein PC
  - mehr als ein progrmmierbarer Taschenrechner ?

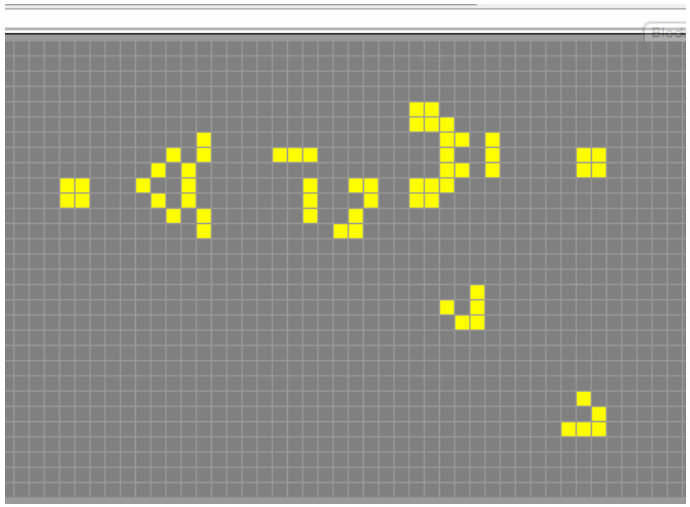


Was können Algorithmen nicht

- Gibt es Algorithmen, um zu entscheiden, ob ein Programm
  - syntaktisch korrekt ist ?
    - ja, Parser
  - semantisch korrekt ist
    - kein Laufzeitfehler ?
    - keine Endlosschleife ?
    - nein, nicht algorithmisch lösbar

- Gibt es einen Algorithmus, um zu entscheiden, ob
  - zwei Programme äquivalent sind ?
  - ein Rechner virenverseucht ist
  - ein Programm einen Virus hat

Game of Life

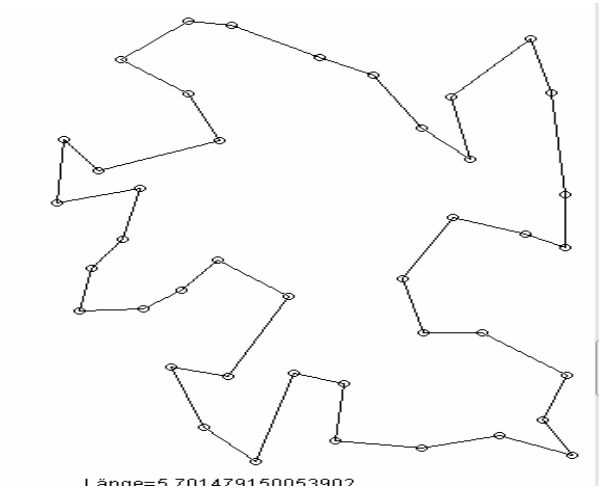


- Gibt es einen Algorithmus um festzustellen, ob eine Konfiguration
  - ausstirbt
    - nein
  - sich reproduziert
    - nein
  - periodisch wird
    - nein
  - sich nach höchstens k Schritten wiederholt
    - ja

- Regeln (John Conway)
  - Eine lebendes Feld
    - mit 2 oder 3 Nachbarn überlebt
    - mit weniger als 2 oder mehr als 3 Nachbarn stirbt
      - vor Einsamkeit
      - wegen Überbevölkerung
  - Ein leeres Feld
    - mit 3 Nachbarn erweckt zum Leben

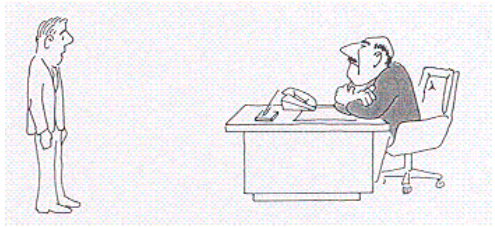
Komplexität

- Inhärent schwere Probleme
  - Travelling Salesman
  - Bin Packing
  - Satisfiability
  - Hamiltonscher Kreis
- Lösbar, aber kein effizienter Algorithmus bekannt
  - Könnte man eines effizient lösen, dann auch alle anderen
- Lösungsalgorithmus im wesentlichen
  - systematisches Ausprobieren
- Oft behilft man sich mit Näherungslösungen
  - z.B.: simulated Annealing
  - siehe: Algorithmus der Woche
    - » Simulated Annealing



http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo41.php

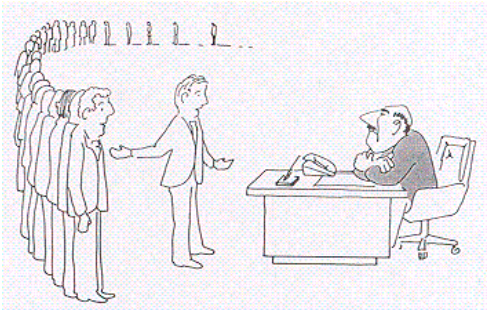
NP-Vollständigkeit



I can't find an efficient algorithm, I guess I'm just too dumb.

- Problem P ist NP-vollständig, wenn es genau so schwer ist, wie eines (= alle) dieser Probleme
  - Könnte man eines der schweren Probleme effizient lösen, dann auch P
  - Könnte man P effizient lösen, so könnte man auch die berühmten schweren Probleme lösen
- Viele Probleme sind NP-vollständig
  - trotzdem nicht die Flinte ins Korn werfen
  - Komplexitätsaussagen gelten nur asymptotisch
  - für „kleine“ Inputgrößen gibt es viel Raum für Kreativität
  - SAT-Probleme mit tausend Variablen heute lösbar
    - Binary Decision Diagrams
    - Stålmark-Algorithmus®
    - industriell relevant
      - z.B. für Model checking

I can't find an efficient algorithm,
because no such algorithm is possible.



I can't find an efficient algorithm, but

neither can all these famous people.