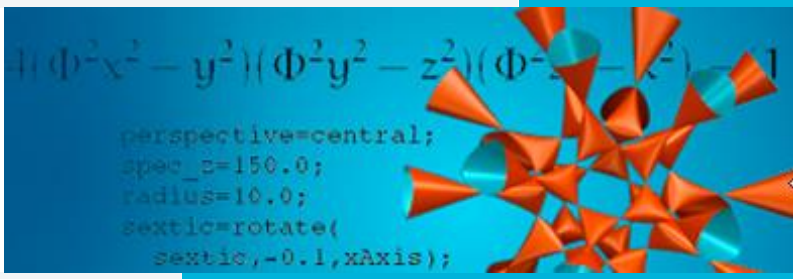
Philipps Universität Marburg

Theoretische Informatik: Überblick

Prof. Dr. Elmar Tischhauser



Basierend auf Material von © H.-P. Gumm und J. Daubert

Themen der Vorlesung

- **Spracherkennung und –verarbeitung**
  - Reguläre Sprachen
    - Lexikalische Analyse
  - Kontextfreie Sprachen
    - Syntaktische Analyse
    - Übersetzung (Compiler)
- **Berechenbarkeit**
  - Was heißt „algorithmisch lösbar“ ?
  - Welche Probleme sind nicht algorithmisch lösbar ?
- **Komplexität**
  - Für welche Probleme gibt es gute Algorithmen
  - Inhärent schwere Probleme
  - Quanten Computing ?

Spracherkennung und -verarbeitung

Sprachen sind in mehreren Schichten aufgebaut

1. **Zeichen bzw. Symbole**
   - Endliche Menge, z.B.:
     - Lateinisch = {a,b,c,… ,z,A,B,C,… ,Z}
     - Kyrillisch = {Д,Ж,Й,И,Л,Б,С, …}
     - Arabisch = {ز,ف,ي, ١,٢ , ٣, …}
     - Telugu = {ఐ,ఒ,త,మ,హా,ఱూ,ళ,ఴ, …}
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("Hallo Welt!"); }
```

Analyse von Programmtexten

- Eingabe: ***Ein Satz***
  - 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: ***Ist der Satz korrekt gebildet?*** *(Syntax)*
  - Entscheidung ob das Programm syntaktisch korrekt ist
    - Ggf: Fehlerhinweis
  - Analyse des Programms
    - Wie sind die Teile geschachtelt
    - Kontrollfluss
- Ziel: ***Was bedeutet der Satz*** *(Semantik)*
  - 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

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

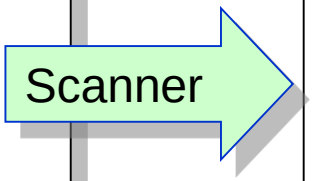
Aufgabe des Scanners

- Zerlegt Programmtext in Worte
  - nach welchen Kriterien ?
- klassifiziert Worte in einem Programmtext
  - Schlüsselwort, Zahlkonstante, Bezeichner, …
- Wortklassen werden durch Token (Gutschein) 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*

Aufgabe des Scanners

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.
```

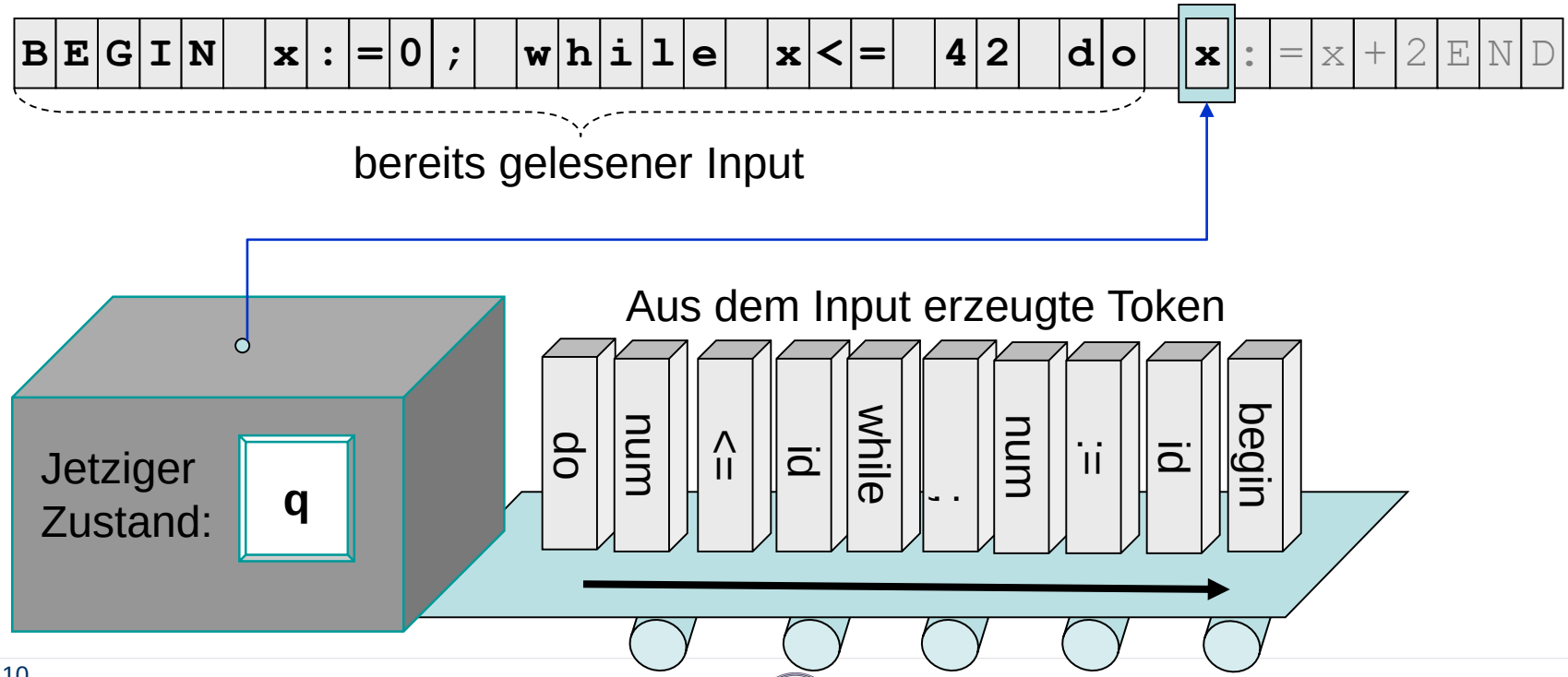


Tokenliste

```
program
id
semi
begin
id
assignOp
num
semi
id
assignOp
num
semi
while
...
```

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*
| *Block*
| …

Anweisungen :: *Anweisung Anweisungen*
| /* 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

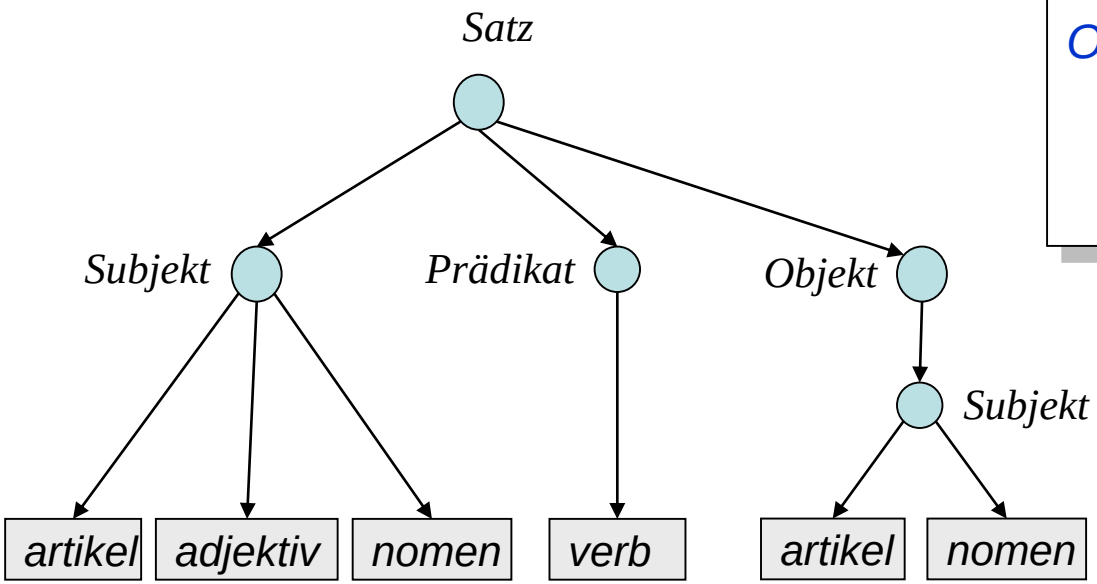
Deutsch für Anfänger

Satz :: *Subjekt Prädikat Objekt*

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

Prädikat :: *verb* | *hilfsverb*

Objekt :: *Subjekt*



Die lila Kuh legt ein Schokoladenei

: Syntaxbaum

Parser

: Tokenliste

Scanner

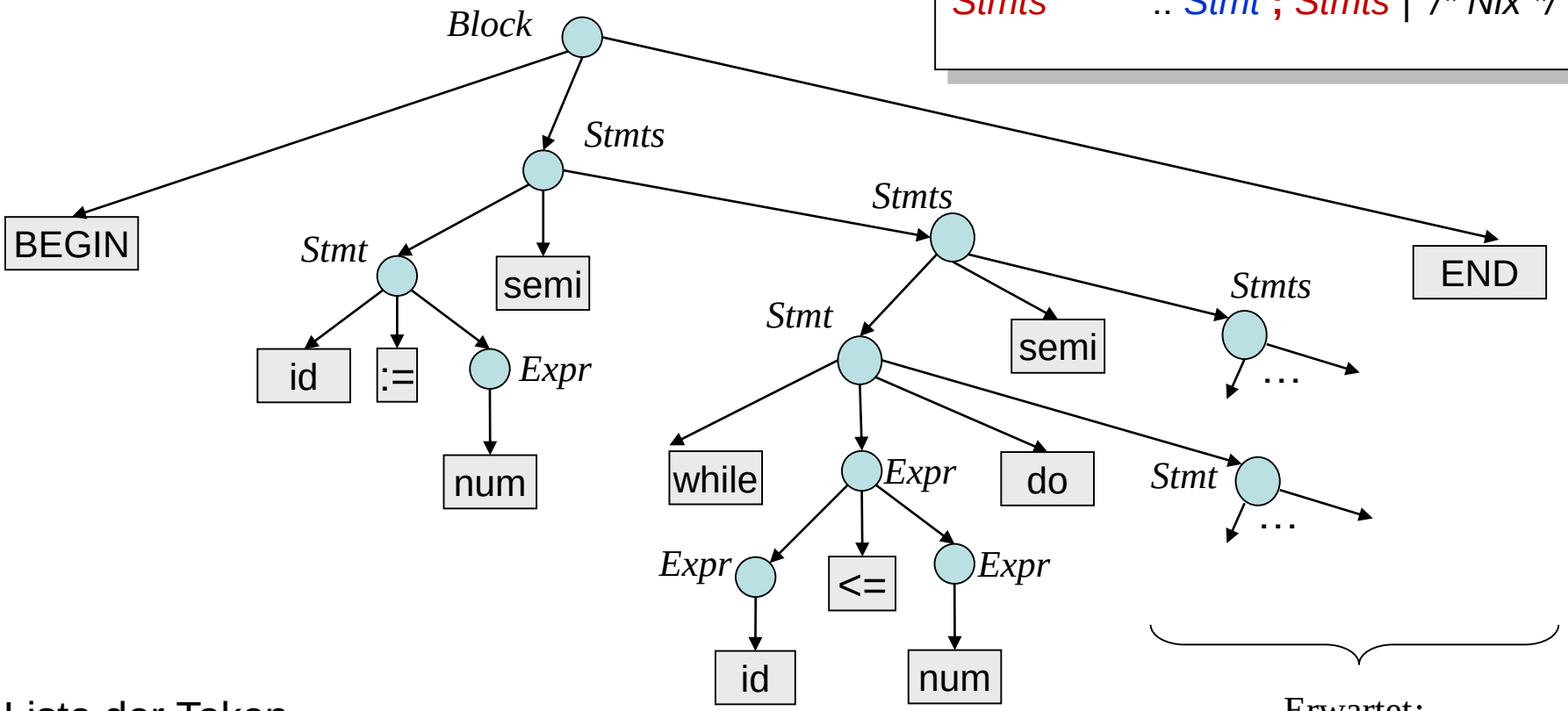
: Ein Satz

Aufgabe des Parsers

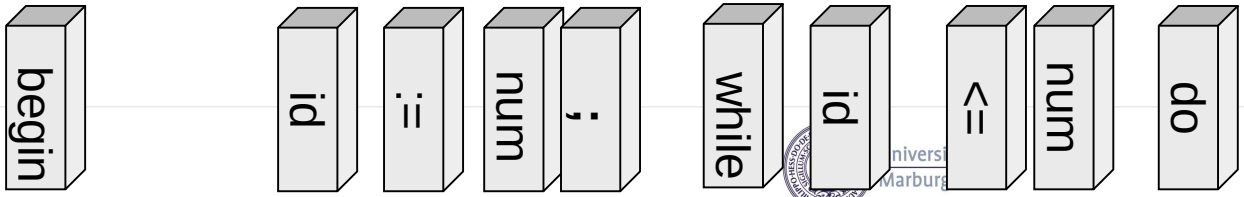
- Parser erzeugt Syntaxbaum

Pascal-Grammatik

Block	:: **begin** *Stmts* **end**
Stmt	:: **id** **:=** *Expr*
	| **while** *Expr* **do** *Stmt*
Stmts	:: *Stmt* **;** *Stmts* | /* Nix */



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

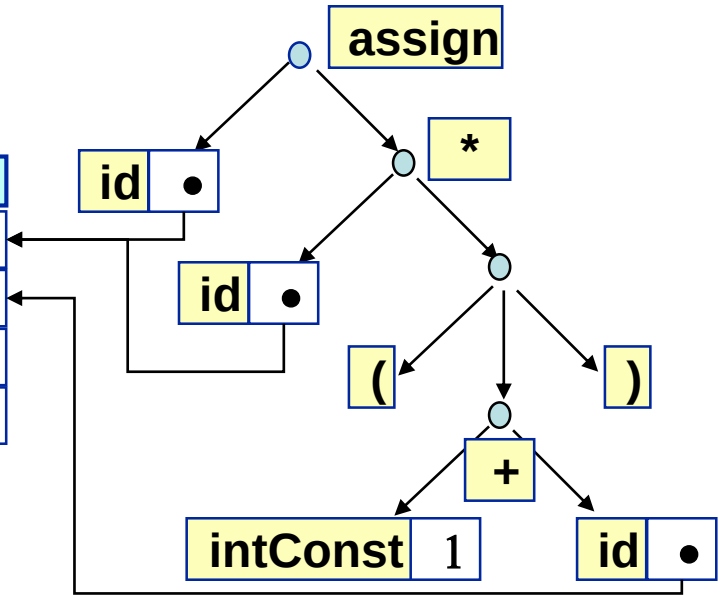
Codeerzeugung

- 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 hypoth. Stackprozessor

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

Themen der Vorlesung

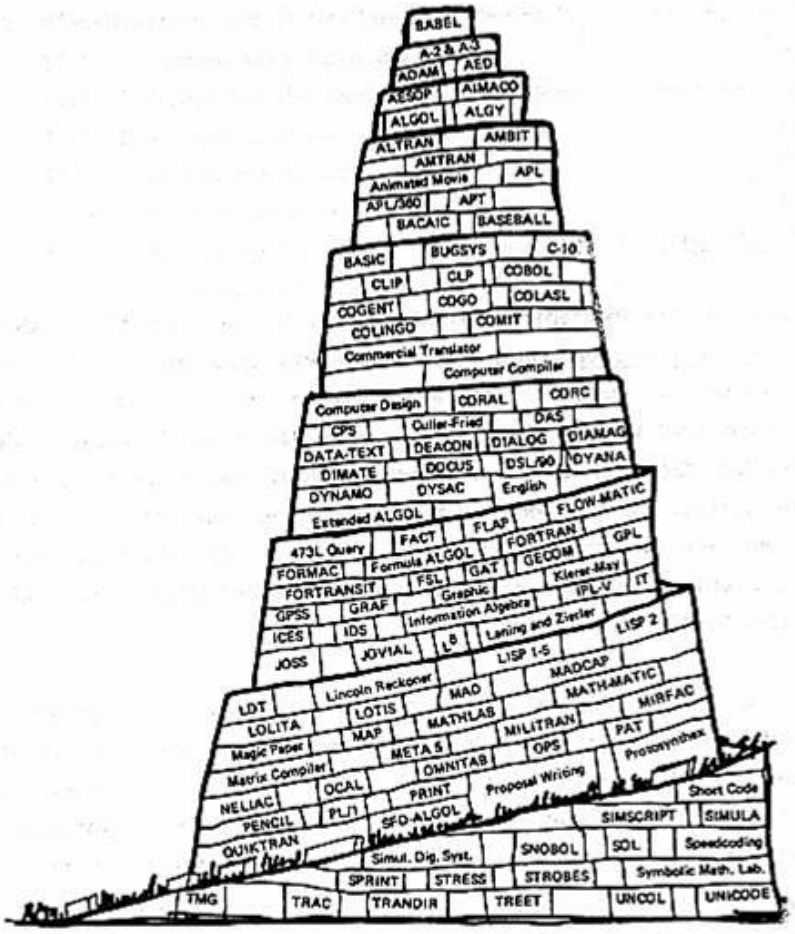
- Spracherkennung und –verarbeitung
  - Reguläre Sprachen
    - Lexikalische Analyse
  - Kontextfreie Sprachen
    - Syntaktische Analyse
    - Übersetzung (Compiler)

- Berechenbarkeit
  - Was heißt „algorithmisch lösbar“ ?
  - Welche Probleme sind nicht algorithmisch lösbar ?

- Komplexität
  - Für welche Probleme gibt es gute Algorithmen
  - Inhärent schwere Probleme

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 programmierbarer Taschenrechner ?



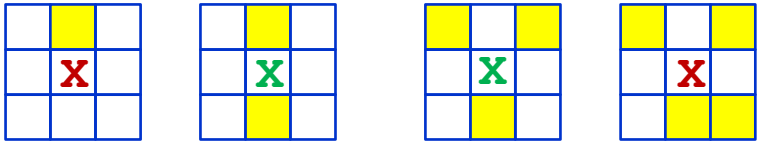
Was können Algorithmen nicht

- 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 Malware enthält
  - ein Nutzer potenziell auf ein Objekt Zugriff erhalten kann?

Game of Life

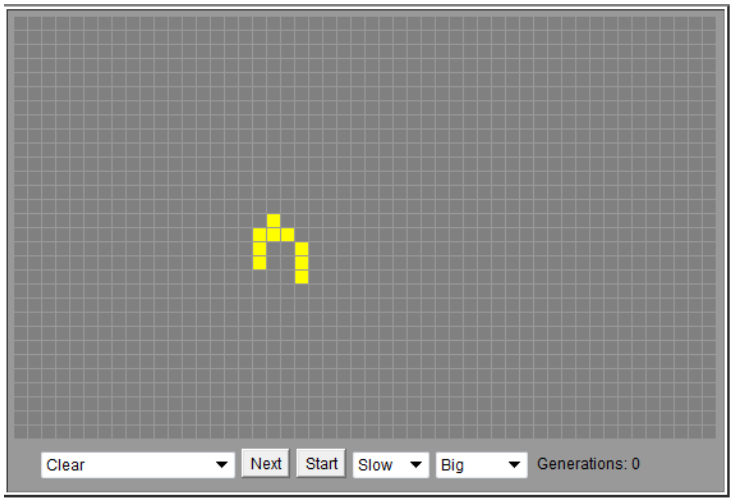
- Gibt es einen Algorithmus, um festzustellen, ob eine Konfiguration...
  - ...ausstirbt?
  - ...sich reproduziert?
  - ...periodisch wird?
  - ...sich nach höchstens k Schritten wiederholt?

- **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

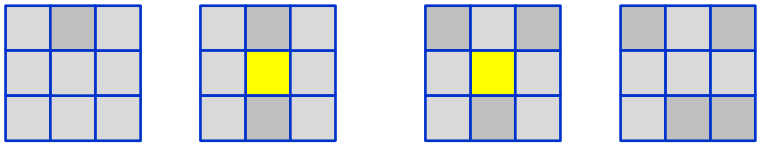


Game of Life

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



- **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



Game of Life

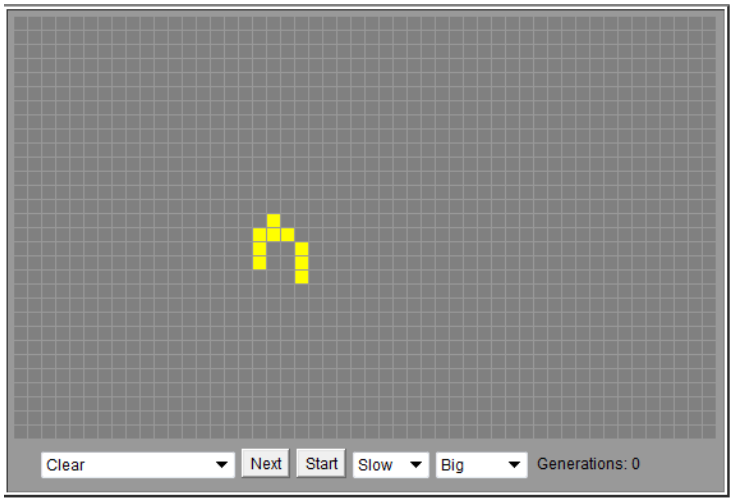
- Gibt es einen Algorithmus, um festzustellen, ob eine Konfiguration...
  - ...ausstirbt?

    Nein
  - ...sich reproduziert?

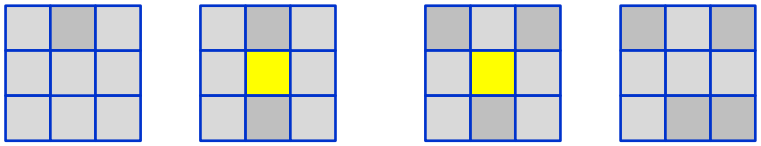
    Nein
  - ...periodisch wird?

    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



Themen der Vorlesung

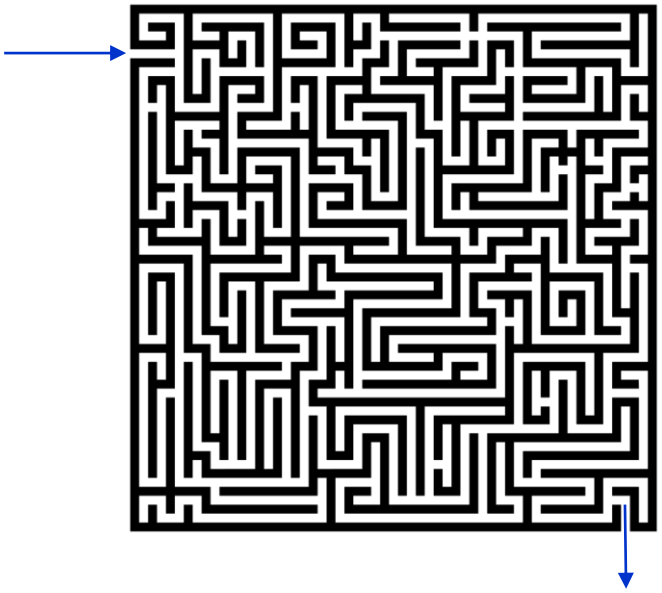
- Spracherkennung und –verarbeitung
  - Reguläre Sprachen
    - Lexikalische Analyse
  - Kontextfreie Sprachen
    - Syntaktische Analyse
    - Übersetzung (Compiler)

- Berechenbarkeit
  - Was heißt „algorithmisch lösbar“ ?
  - Welche Probleme sind nicht algorithmisch lösbar ?

- Komplexität
  - Für welche Probleme gibt es gute Algorithmen
  - Inhärent schwere Probleme

Komplexität

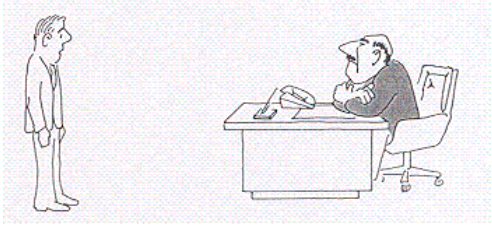
- Inhärent schwere Probleme
  - Travelling Salesman (TSP)
  - Bin Packing
  - Erfüllbarkeit einer Boole‘schen Formel
  - Existenz eines Hamiltonschen Kreises
  - ...
- Bisher bekannte Lösungsalgorithmen:
  - systematisches Ausprobieren
- Also lösbar, aber
  - kein effizienter Algorithmus bekannt
  - unklar, ob effizienter Algorithmus existiert
- Manchmal recht gute Näherungslösungen
  - z.B.: Christofides-Heuristik für metrisches TSP (Approximationsfaktor <= 3/2)
  - z.B.: Packing von Steinerbäumen (VLSI-Design)



http://4vector.com/i/free-vector-labyrinth_098257_Labyrinth.png

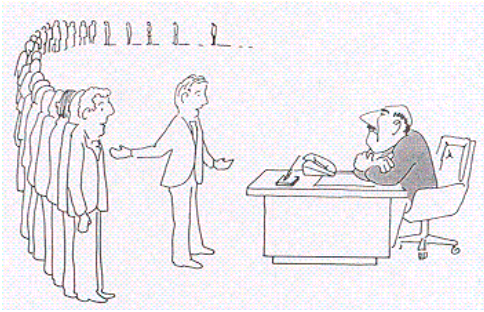
NP-Vollständigkeit

- NP-vollständige Probleme:
  - Wichtige Klasse schwerer Probleme
  - Könnte man ein Problem aus NP effizient lösen, dann auch alle anderen
- Viele relevante Probleme sind NP-vollständig
  - trotzdem nicht die Flinte ins Korn werfen
    - Komplexitätsaussagen nur asymptotisch
    - „kleine“ Inputgrößen bieten Raum für Kreativität
  - SAT-Probleme mit tausenden Variablen heute lösbar
    - Binary Decision Diagrams
    - Stålmark-Algorithmus®
    - industriell relevant
      - z.B. für Model checking
      - Z.B. Für automatisierte symmetrische Kryptoanalyse



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

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.