Algorithmisches Denken und imperative Programmierung

1. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2012/13

Wir

- Ihr Dozent: Janis Voigtländer
  Institut für Informatik, Abteilung III
  `http://www.iai.uni-bonn.de/~jv/`

  Arbeitsgebiet: Programmiersprachen

Wir

- Ihr Dozent: Janis Voigtländer
  Institut für Informatik, Abteilung III
  `http://www.iai.uni-bonn.de/~jv/`

  Arbeitsgebiet: Programmiersprachen

- Assistent: Daniel Seidel
  `http://preview.tinyurl.com/seideld`

Wir

- Ihr Dozent: Janis Voigtländer
  Institut für Informatik, Abteilung III
  `http://www.iai.uni-bonn.de/~jv/`

  Arbeitsgebiet: Programmiersprachen

- Assistent: Daniel Seidel
  `http://preview.tinyurl.com/seideld`

- Tutoren: Marcel Brandt
  Alasdair Collinson
  Gregor Ihmor
  Andy Jäger
  Julius von Kohout

Sie

1. Studierende im Bachelorstudiengang Informatik

2. Studierende im neuen Lehramtstudiengang Informatik

3. Studierende mit Nebenfach Informatik

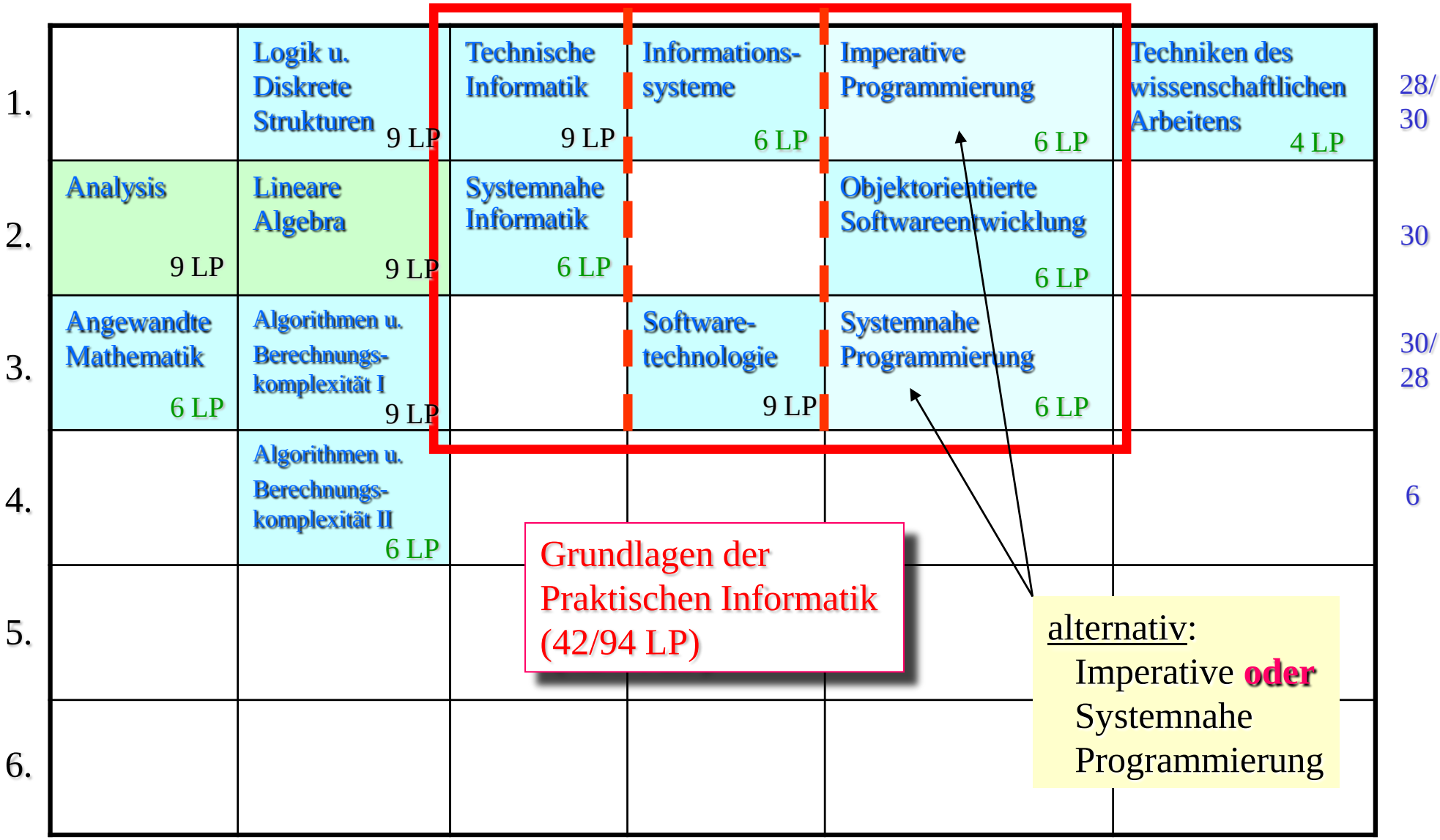
Sie

1. Studierende im Bachelorstudiengang Informatik

2. Studierende im neuen Lehramtstudiengang Informatik

3. Studierende mit Nebenfach Informatik

Zu 1., lediglich wenn Programmierung I+II
statt Programmierung II+III als
Pflichtmodule gewählt.


1.
2.
3.
4.
5.
6.
Logik u. Diskrete Strukturen 9 LP
Technische Informatik 9 LP
Informations-systeme 6 LP
Imperative Programmierung 6 LP
Techniken des wissenschaftlichen Arbeitens 4 LP
28/30
Analysis 9 LP
Lineare Algebra 9 LP
Systemnahe Informatik 6 LP
Objektorientierte Softwareentwicklung 6 LP
30
Angewandte Mathematik 6 LP
Algorithmen u. Berechnungs-komplexität I 9 LP
Software-technologie 9 LP
Systemnahe Programmierung 6 LP
30/28
Algorithmen u. Berechnungs-komplexität II 6 LP
6
Grundlagen der Praktischen Informatik (42/94 LP)
alternativ:
Imperative oder
Systemnahe
Programmierung

94

Diese Veranstaltung

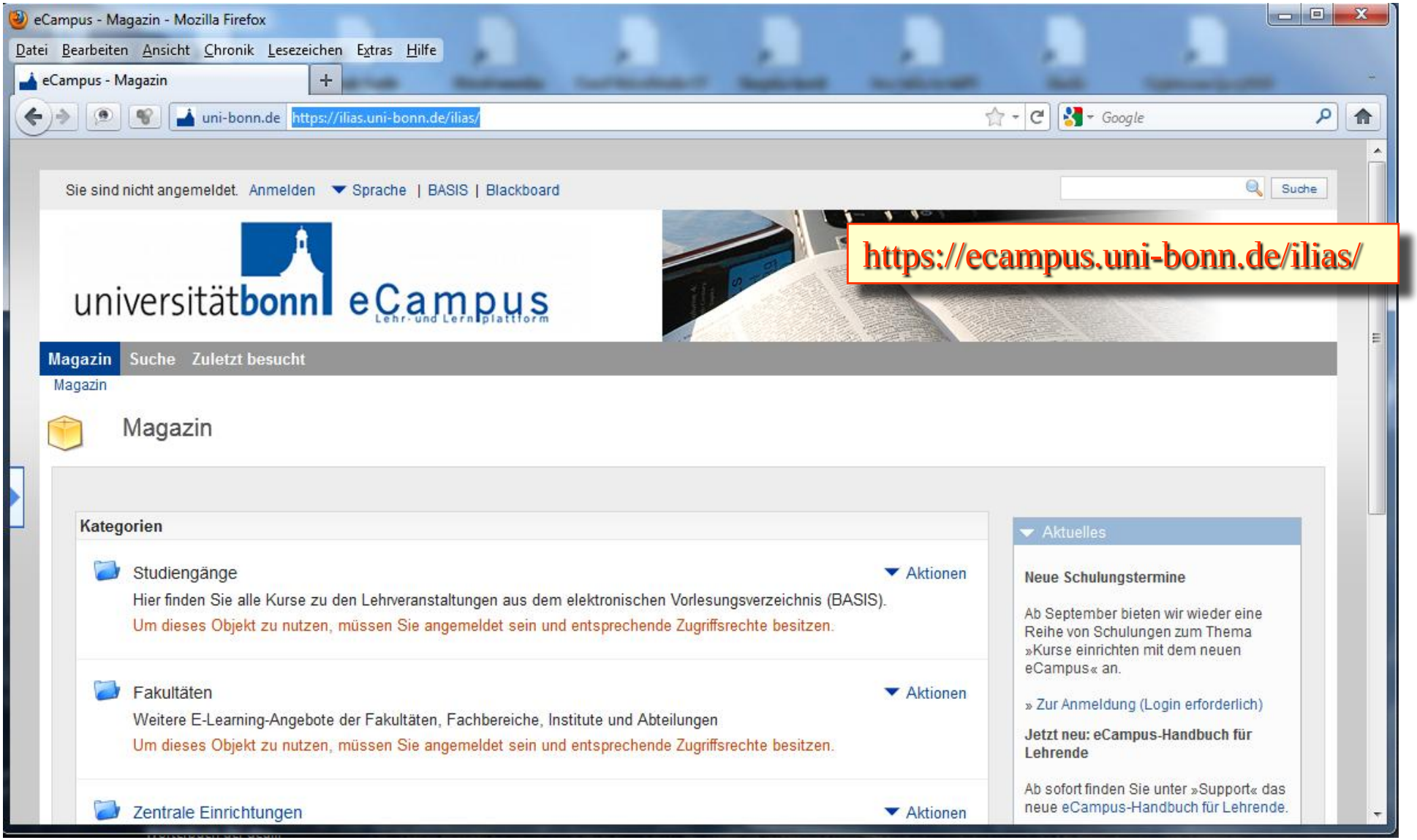
- Vorlesungstermine:
  - 2 SWS
  - montags 12:30–14:00 (Hörsaal 1+2)
  - 15.10.12–28.01.13, 14 Termine
- Übungstermine:
  - 2 SWS
  - Termine über die Woche verteilt
  - Einschreibung per URS bis spätestens übermorgen!
  - ab 23.10.12
  - Ausfall: 01.11.12 (Feiertag), 05.12.12 (Dies Academicus)

Diese Veranstaltung

- Vorlesungstermine:
  - 2 SWS
  - montags 12:30–14:00 (Hörsaal 1+2)
  - 15.10.12–28.01.13, 14 Termine
- Übungstermine:
  - 2 SWS
  - Termine über die Woche verteilt
  - Einschreibung per URS bis spätestens übermorgen!
  - ab 23.10.12
  - Ausfall: 01.11.12 (Feiertag), 05.12.12 (Dies Academicus)
- „Aufwand“: 6 LP = 180 Stunden Workload!

Diese Veranstaltung

- Vorlesungstermine:
  - 2 SWS
  - montags 12:30–14:00 (Hörsaal 1+2)
  - 15.10.12–28.01.13, 14 Termine
- Übungstermine:
  - 2 SWS
  - Termine über die Woche verteilt
  - Einschreibung per URS bis spätestens übermorgen!
  - ab 23.10.12
  - Ausfall: 01.11.12 (Feiertag), 05.12.12 (Dies Academicus)
- „Aufwand“: 6 LP = 180 Stunden Workload!
- weitere Infos (zu Terminen und anderem) auf Webseite:
  - zunächst `http://www.iai.uni-bonn.de/~jv/teaching/adip/`
  - ab nächste Woche eCampus/ILIAS


eCampus - Magazin - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
eCampus - Magazin
uni-bonn.de
https://ilias.uni-bonn.de/ilias/
Google
Sie sind nicht angemeldet. Anmelden Sprache | BASIS | Blackboard
Suche
universitätbonn eCampus
Lehr- und Lernplattform
https://ecampus.uni-bonn.de/ilias/
Magazin Suche Zuletzt besucht
Magazin
Magazin
Kategorien
Studiengänge
Aktionen
Hier finden Sie alle Kurse zu den Lehrveranstaltungen aus dem elektronischen Vorlesungsverzeichnis (BASIS).
Um dieses Objekt zu nutzen, müssen Sie angemeldet sein und entsprechende Zugriffsrechte besitzen.
Fakultäten
Aktionen
Weitere E-Learning-Angebote der Fakultäten, Fachbereiche, Institute und Abteilungen
Um dieses Objekt zu nutzen, müssen Sie angemeldet sein und entsprechende Zugriffsrechte besitzen.
Zentrale Einrichtungen
Aktionen
Aktuelles
Neue Schulungstermine
Ab September bieten wir wieder eine Reihe von Schulungen zum Thema »Kurse einrichten mit dem neuen eCampus« an.
» Zur Anmeldung (Login erforderlich)
Jetzt neu: eCampus-Handbuch für Lehrende
Ab sofort finden Sie unter »Support« das neue eCampus-Handbuch für Lehrende.

Weitere Details

Zur Übungsdurchführung:

- falls noch nicht getan, „0. Übungsblatt“ bearbeiten!
- erstes „echtes“ Übungsblatt morgen online
- zu bearbeiten jeweils bis Sonntag abend
- Achtung: elektronische Abgaben, harte Deadlines

Weitere Details

Zur Übungsdurchführung:

- falls noch nicht getan, „0. Übungsblatt“ bearbeiten!
- erstes „echtes“ Übungsblatt morgen online
- zu bearbeiten jeweils bis Sonntag abend
- Achtung: elektronische Abgaben, harte Deadlines

Zur Klausur:

- voraussichtlich zu Beginn der vorlesungsfreien Zeit

Weitere Details

Zur Übungsdurchführung:

- falls noch nicht getan, „0. Übungsblatt“ bearbeiten!
- erstes „echtes“ Übungsblatt morgen online
- zu bearbeiten jeweils bis Sonntag abend
- Achtung: elektronische Abgaben, harte Deadlines

Zur Klausur:

- voraussichtlich zu Beginn der vorlesungsfreien Zeit
- Zulassungsbedingungen:
  - regelmäßige Übungsteilnahme (Pflicht!)
  - mindestens 50% der Übungspunkte
  - „Zwischenabrechnung“ etwa zur Mitte des Semesters

Weitere Details

Zur Übungsdurchführung:

- falls noch nicht getan, „0. Übungsblatt“ bearbeiten!
- erstes „echtes“ Übungsblatt morgen online
- zu bearbeiten jeweils bis Sonntag abend
- Achtung: elektronische Abgaben, harte Deadlines

Zur Klausur:

- voraussichtlich zu Beginn der vorlesungsfreien Zeit
- Zulassungsbedingungen:
  - regelmäßige Übungsteilnahme (Pflicht!)
  - mindestens 50% der Übungspunkte
  - „Zwischenabrechnung“ etwa zur Mitte des Semesters
- Nachklausur voraussichtlich Mitte/Ende März

Ein ganz grober inhaltlicher Überblick

Algorithmen

- Algorithmusbegriff
- konkrete Algorithmen und algorithmische Prinzipien
- Aufwandsbetrachtungen
- Datenstrukturen als algorithmisches Konzept

Programmieren

- Beschreibung der Syntax von Programmiersprachen
- Kontrollstrukturen und deren Bedeutung/Semantik
- programmiersprachliche Konzepte wie: Funktionskonzept, Variablenüberdeckung, Speicherverwaltung
- Datenstrukturen aus programmiersprachlicher Sicht
- Implementation/Übersetzung von Programmiersprachen

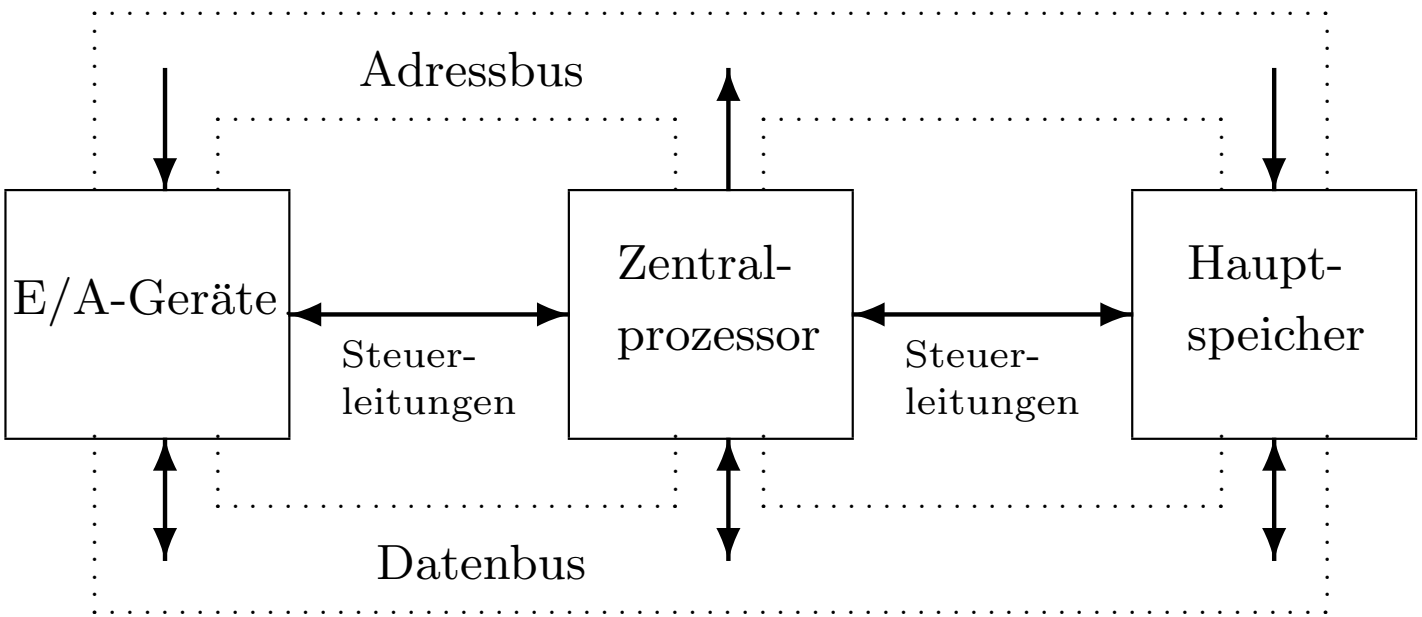
Programmiersprache(n)?

Warum Programmieren/Programmiersprachen?

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

$\Downarrow$?

Von-Neumann-Rechner:



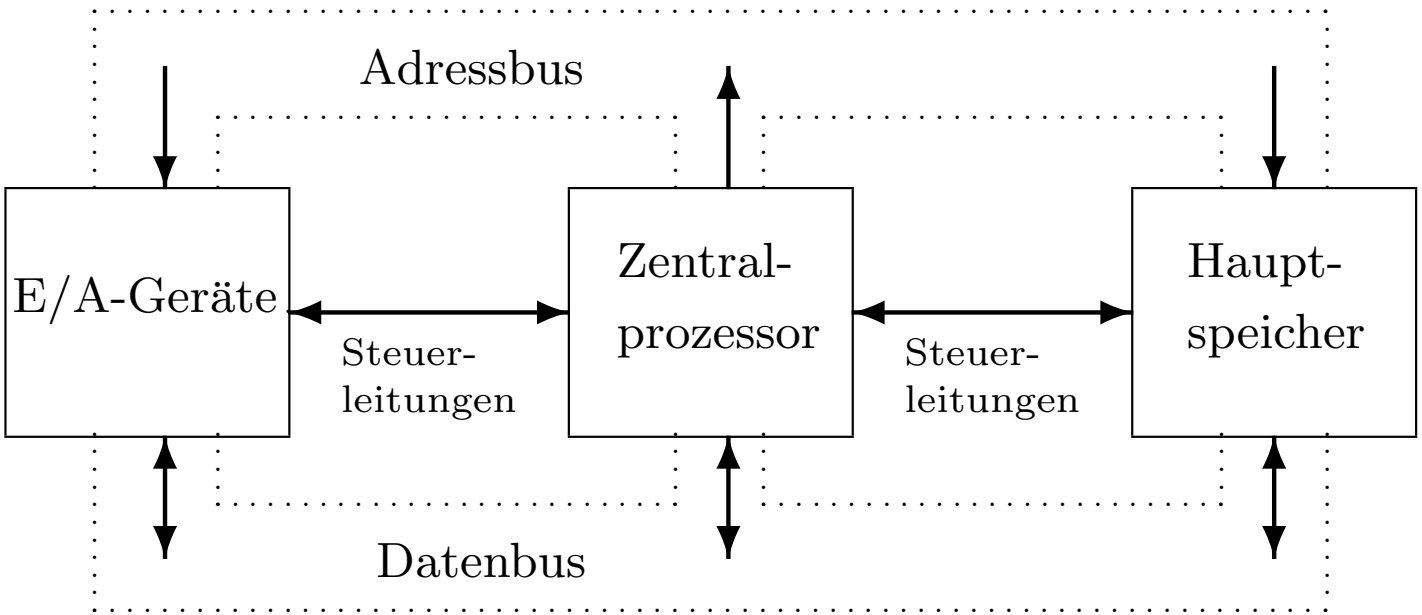
Warum Programmieren/Programmiersprachen?

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

1, 1, 2, 3, 5, 8, 13, 21, ...

$\Downarrow$?

Von-Neumann-Rechner:



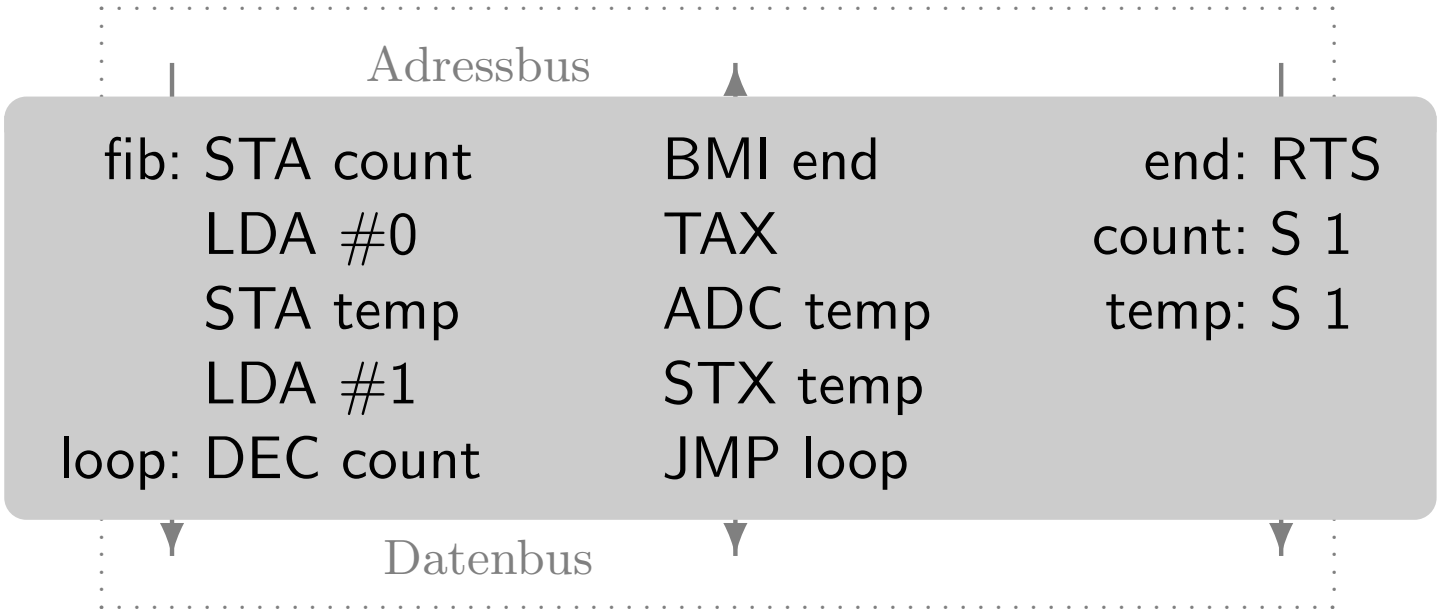
Warum Programmieren/Programmiersprachen?

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

1, 1, 2, 3, 5, 8, 13, 21, ...

$\Downarrow$?

Von-Neumann-Rechner:



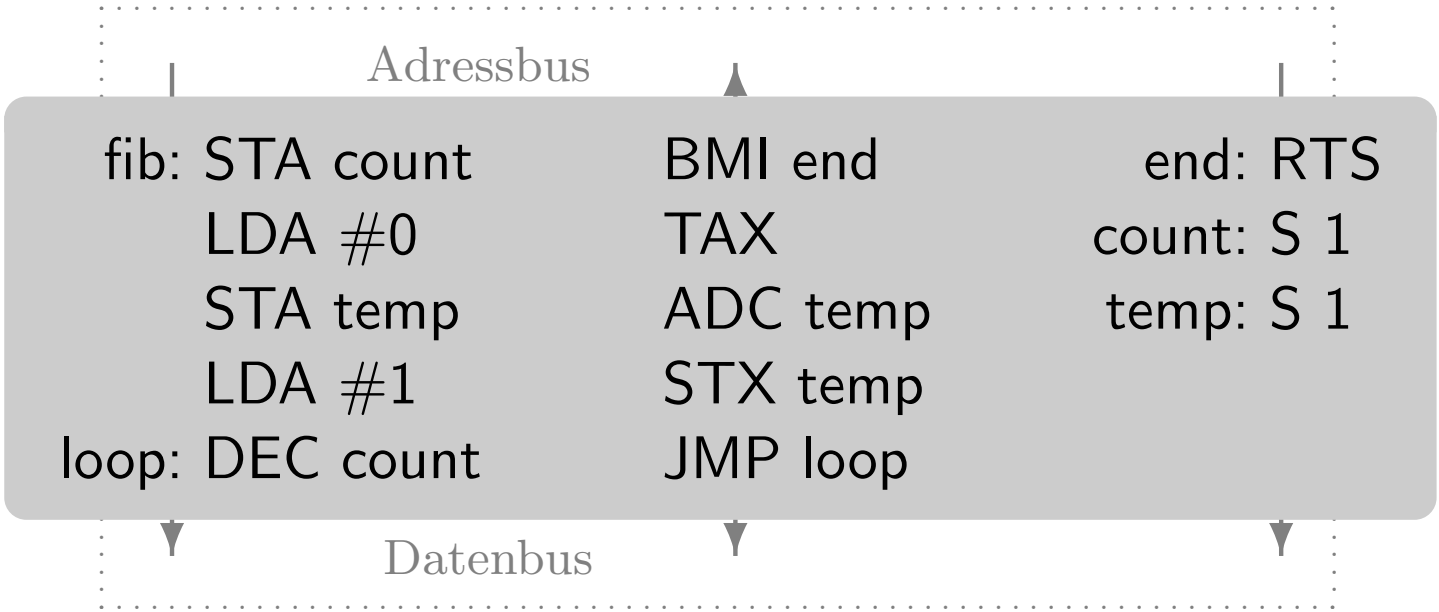
Warum Programmieren/Programmiersprachen?

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

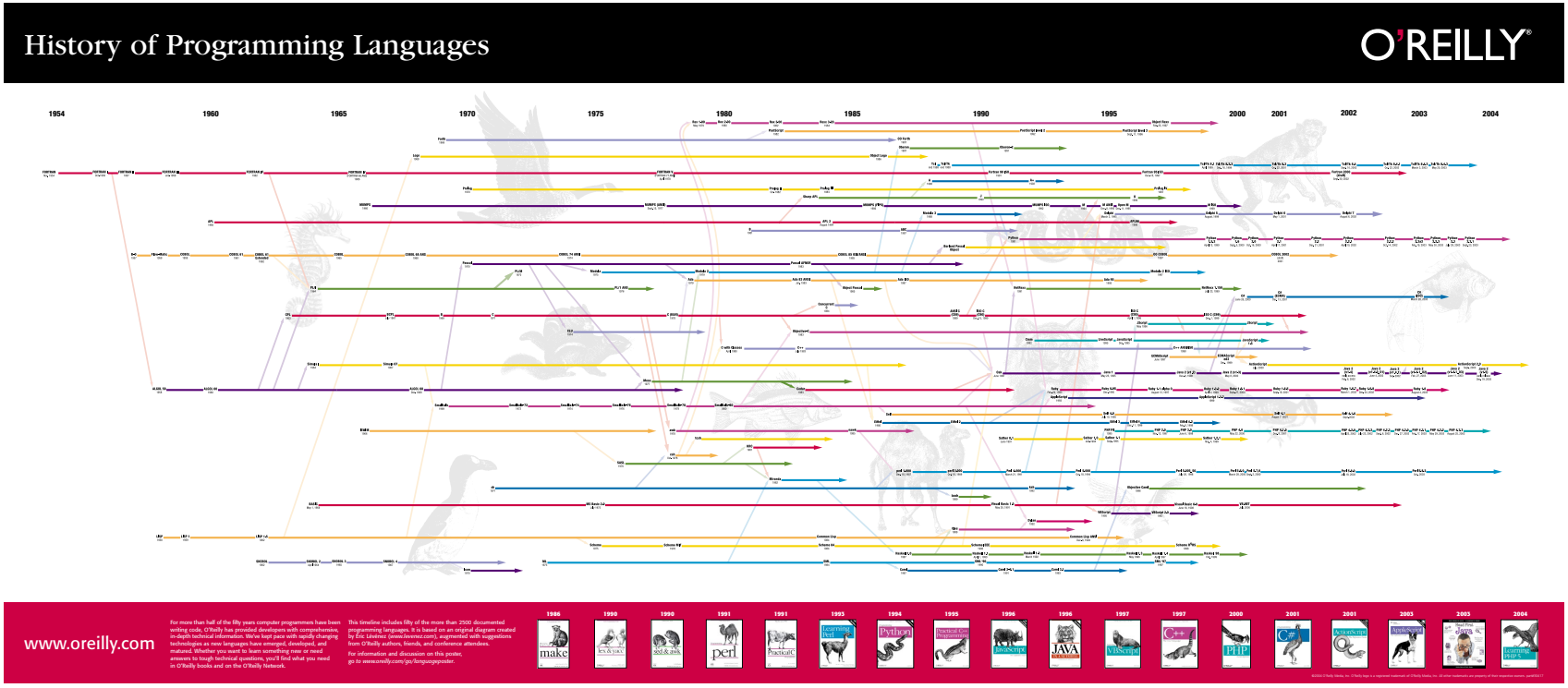
1, 1, 2, 3, 5, 8, 13, 21, ...

$\Downarrow$? mittels einer Hochsprache!

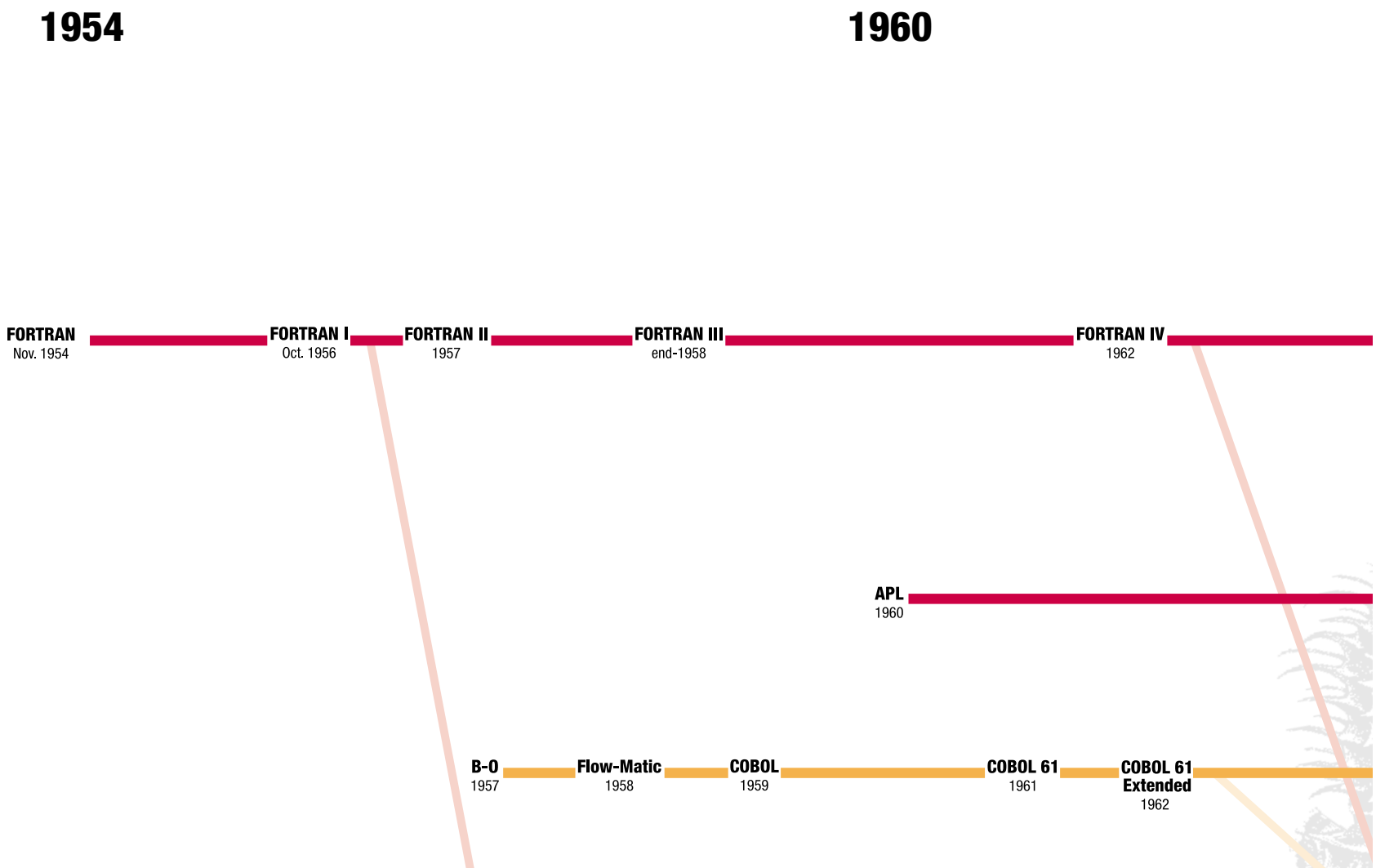
Von-Neumann-Rechner:



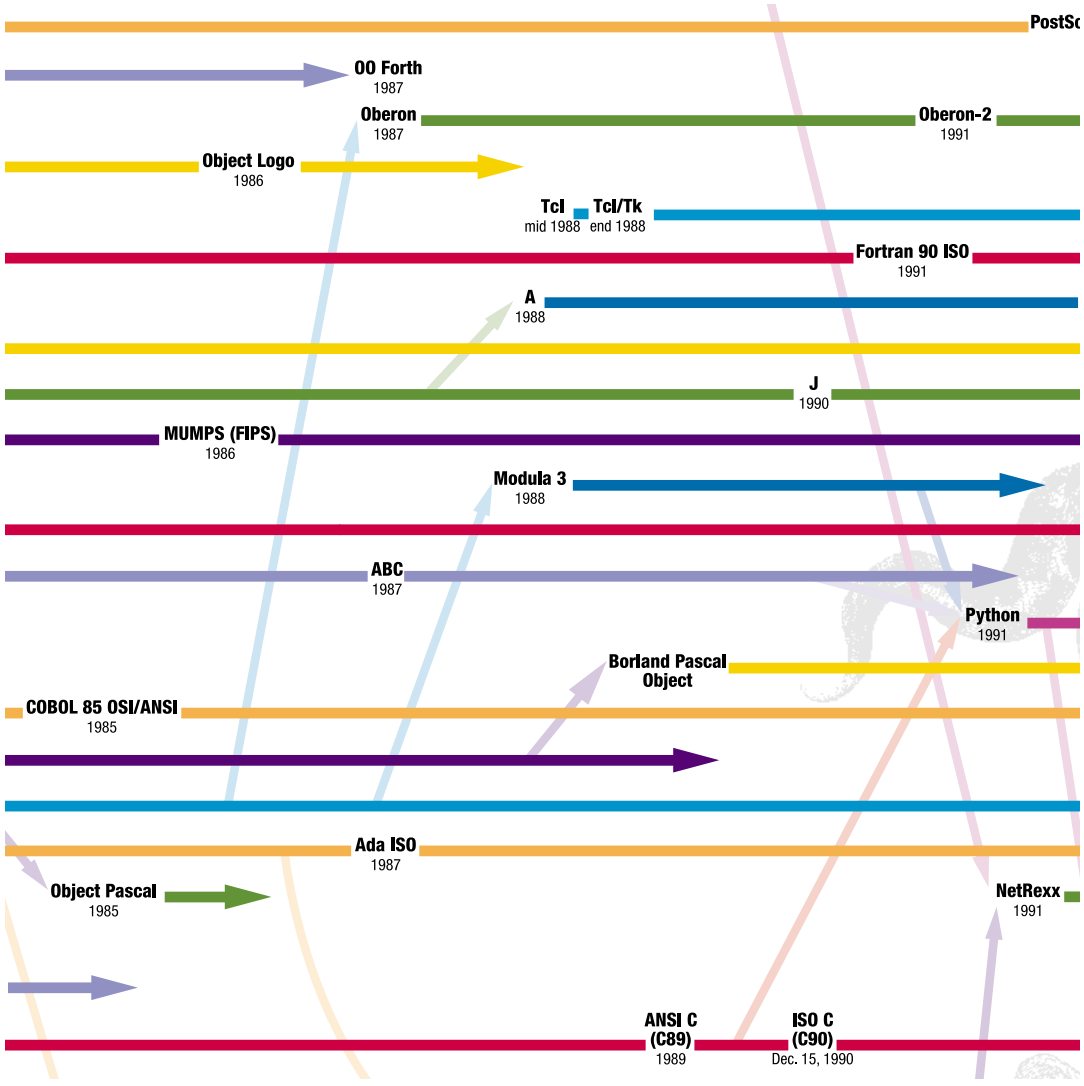
Eine Vielfalt an Höheren Programmiersprachen



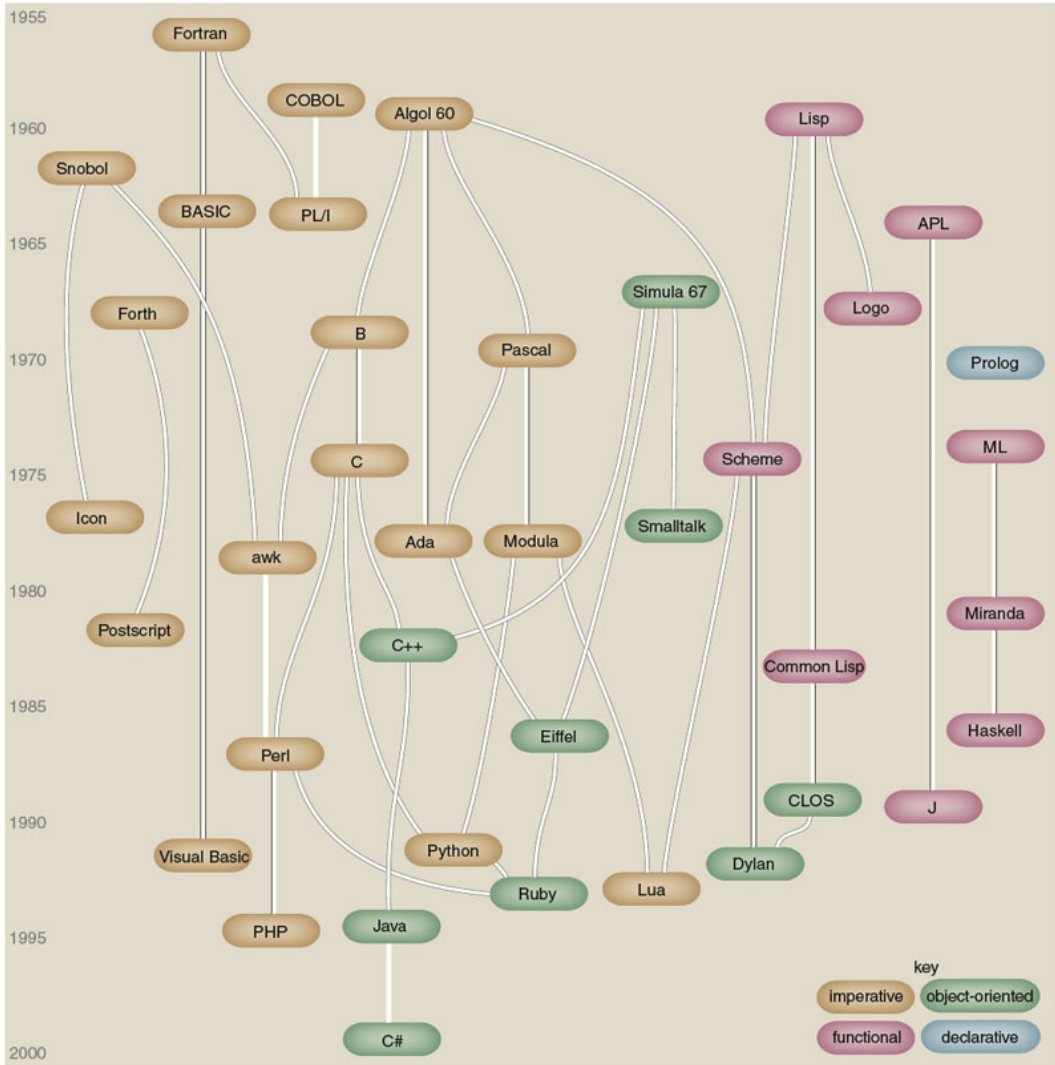
Eine Vielfalt an Höheren Programmiersprachen



Eine Vielfalt an Höheren Programmiersprachen

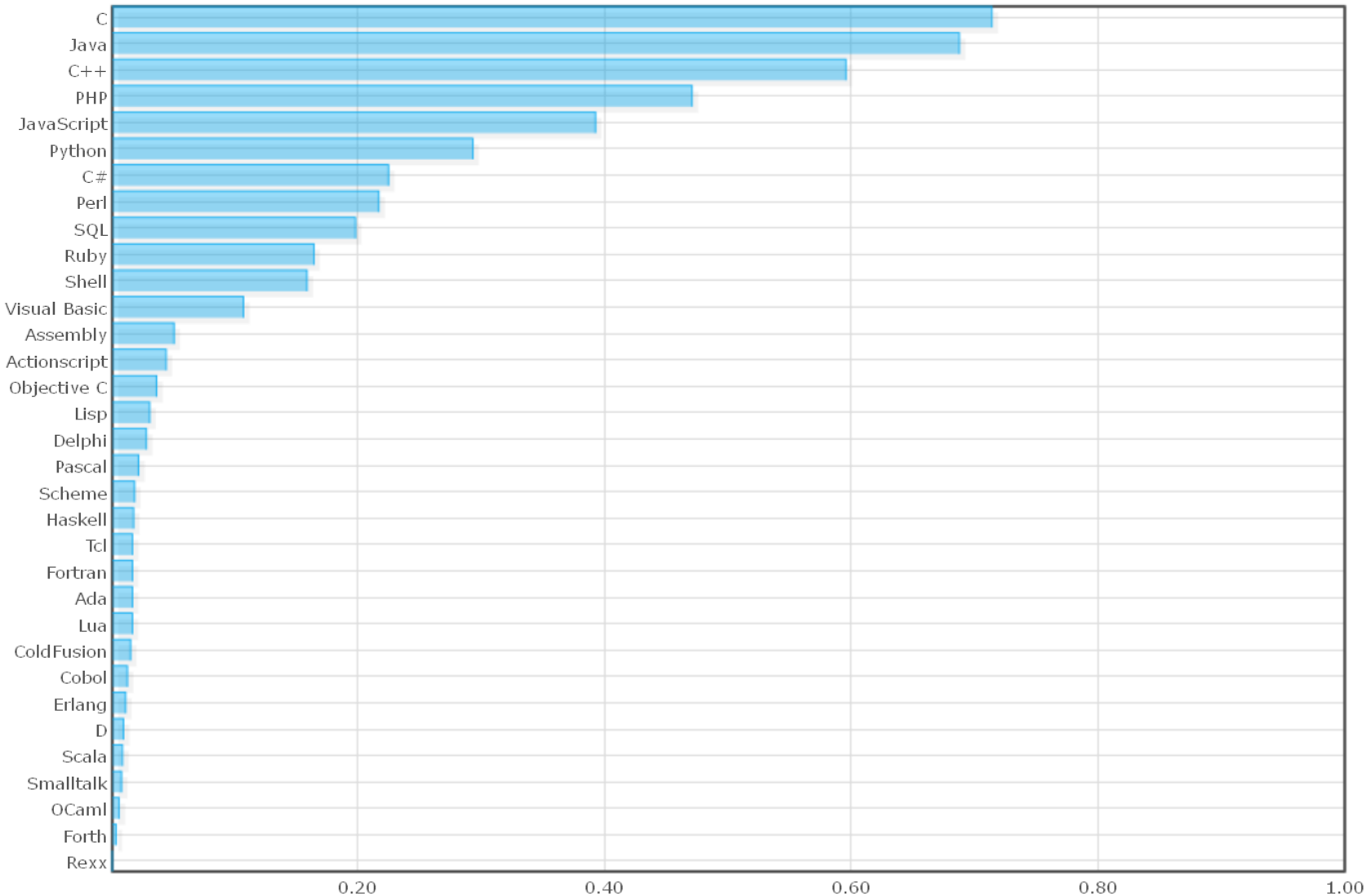


Eine weitere Perspektive

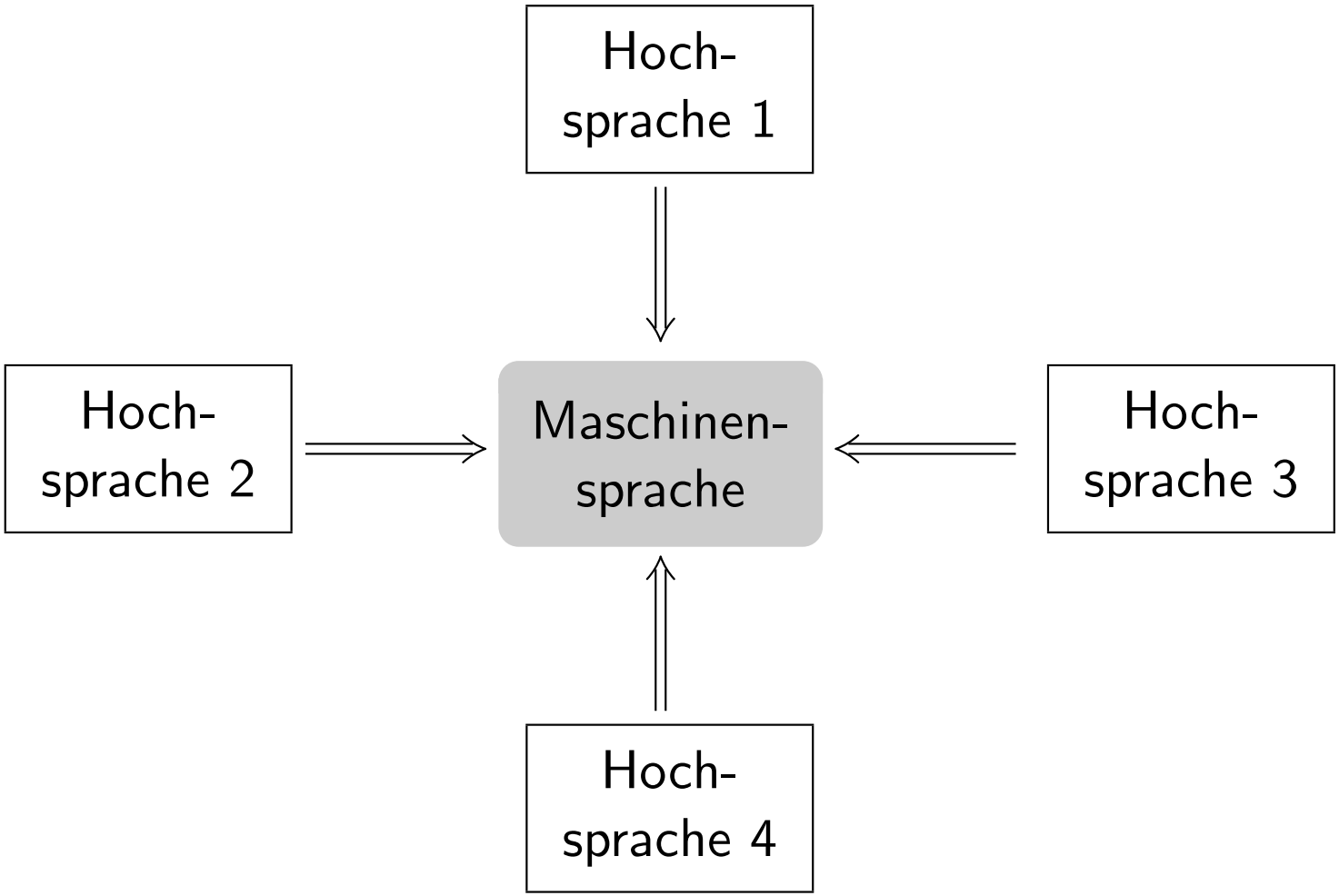


Aus dem „American Scientiest“: The Semicolon Wars

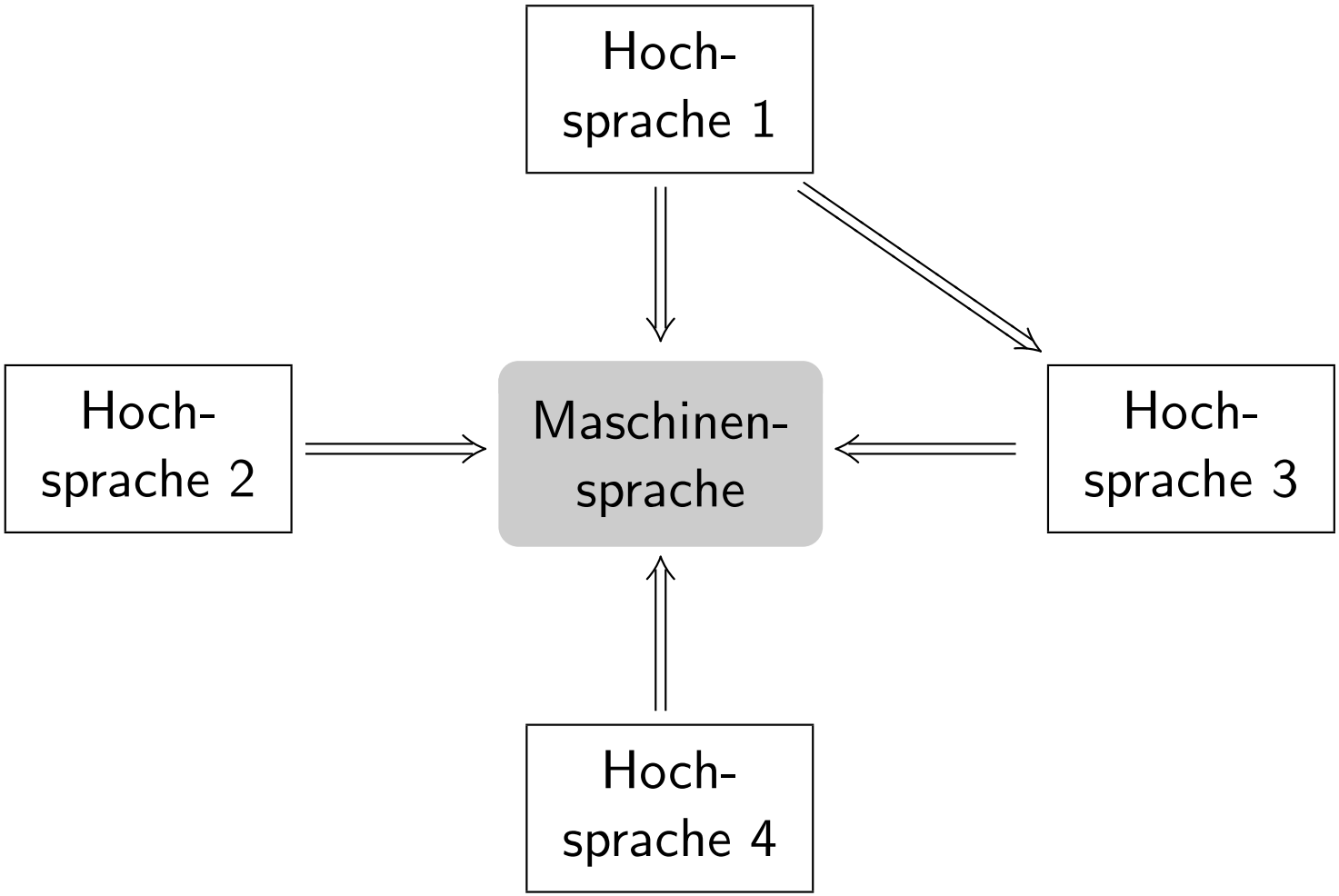
http://langpop.com/



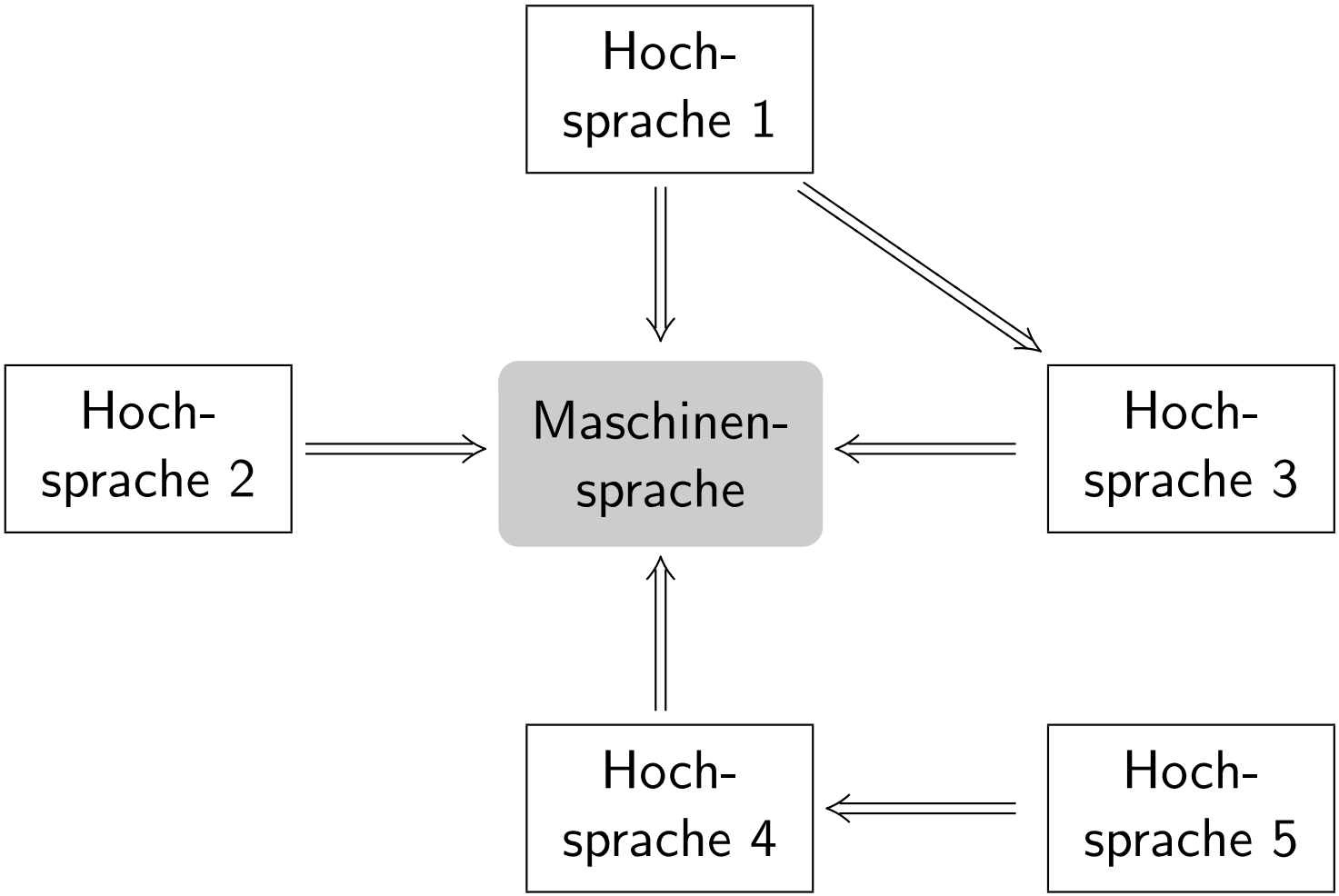
Übersetzung von Programmiersprachen



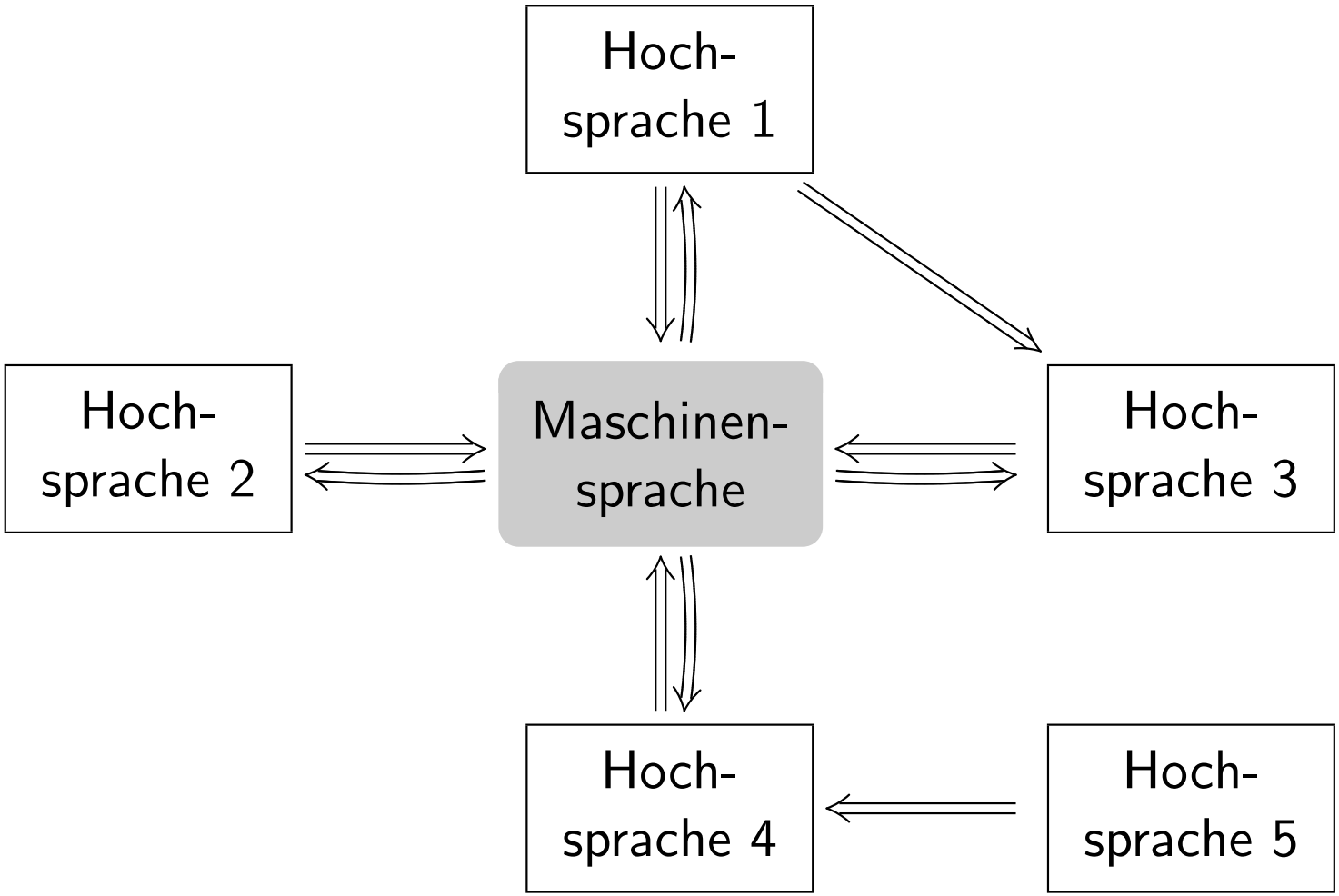
Übersetzung von Programmiersprachen



Übersetzung von Programmiersprachen



Übersetzung von Programmiersprachen



Algorithmen?

Warum Algorithmisches Denken?

Analogie zu mathematischer Modellbildung:

„Textaufgabe“

Wir beobachten das Wachstum einer Kaninchenpopulation. Zu Beginn gibt es ein neugeborenes Paar Kaninchen. Jedes Paar Kaninchen wirft pro Monat ein weiteres Paar Kaninchen. Allerdings bekommt ein neugeborenes Paar erst im zweiten Lebensmonat Nachwuchs. Kaninchen sind unsterblich.
Wie entwickelt sich die Population?

Warum Algorithmisches Denken?

Analogie zu mathematischer Modellbildung:

„Textaufgabe“

Wir beobachten das Wachstum einer Kaninchenpopulation. Zu Beginn gibt es ein neugeborenes Paar Kaninchen. Jedes Paar Kaninchen wirft pro Monat ein weiteres Paar Kaninchen. Allerdings bekommt ein neugeborenes Paar erst im zweiten Lebensmonat Nachwuchs. Kaninchen sind unsterblich.
Wie entwickelt sich die Population?

$$\Downarrow$$

„Rechenvorschrift“

$$\begin{aligned} fib_0 &= 1 \\ fib_1 &= 1 \\ fib_{n+2} &= fib_n + fib_{n+1} \end{aligned}$$

Warum Algorithmisches Denken?

Zur mechanischen Berechnung, weitere Betrachtungen notwendig:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

Warum Algorithmisches Denken?

Zur mechanischen Berechnung, weitere Betrachtungen notwendig:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

$$\Downarrow$$

Beobachtung: Man braucht jeweils nur die Werte der vorangegangenen zwei Monate zu kennen.

Warum Algorithmisches Denken?

Zur mechanischen Berechnung, weitere Betrachtungen notwendig:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

Prinzipielles Vorgehen

1. Man merke sich jeweils den Wert des vorletzten und des letzten Monats.

Warum Algorithmisches Denken?

Zur mechanischen Berechnung, weitere Betrachtungen notwendig:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

$$\Downarrow$$

Prinzipielles Vorgehen

1. Man merke sich jeweils den Wert des vorletzten und des letzten Monats.
2. Für jeden neuen Monat bilde man einen neuen Wert durch Addition der beiden bisher gemerkten.

Warum Algorithmisches Denken?

Zur mechanischen Berechnung, weitere Betrachtungen notwendig:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

Prinzipielles Vorgehen

1. Man merke sich jeweils den Wert des vorletzten und des letzten Monats.
2. Für jeden neuen Monat bilde man einen neuen Wert durch Addition der beiden bisher gemerkten.
3. Man „vergesse“ den Wert des vorletzten Monats und merke sich dafür den neuen Wert.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.
2. Wenn $n < 2$, dann bereits fertig.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.
4. Das Ergebnis liegt zum Schluss in y bereit.

Warum Algorithmisches Denken?

Weiter konkretisierte Anweisungen

Um den Wert des n. Monats zu berechnen:

1. Verwende zwei „Variablen“ x und y, zu Beginn beide gleich 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.
4. Das Ergebnis liegt zum Schluss in y bereit.

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Warum Algorithmisches Denken?

Es wären auch andere Realisierungen möglich gewesen, etwa:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

Warum Algorithmisches Denken?

Es wären auch andere Realisierungen möglich gewesen, etwa:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.

Warum Algorithmisches Denken?

Es wären auch andere Realisierungen möglich gewesen, etwa:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.
2. Andernfalls:
   - 2.1 Berechne separat den Wert des vorletzten Monats.

Warum Algorithmisches Denken?

Es wären auch andere Realisierungen möglich gewesen, etwa:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.
2. Andernfalls:
   - 2.1 Berechne separat den Wert des vorletzten Monats.
   - 2.2 Berechne separat den Wert des letzten Monats.

Warum Algorithmisches Denken?

Es wären auch andere Realisierungen möglich gewesen, etwa:

„Rechenvorschrift“

$$
\begin{aligned}
fib_0 &= 1 \\
fib_1 &= 1 \\
fib_{n+2} &= fib_n + fib_{n+1}
\end{aligned}
$$

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.
2. Andernfalls:
   - 2.1 Berechne separat den Wert des vorletzten Monats.
   - 2.2 Berechne separat den Wert des letzten Monats.
   - 2.3 Das Ergebnis ist die Summe dieser beiden Werte.

Warum Algorithmisches Denken?

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.
2. Andernfalls:
   - 2.1 Berechne separat den Wert des vorletzten Monats.
   - 2.2 Berechne separat den Wert des letzten Monats.
   - 2.3 Das Ergebnis ist die Summe dieser beiden Werte.

Warum Algorithmisches Denken?

„Algorithmus“

Um den Wert des n. Monats zu berechnen:

1. Wenn $n < 2$, dann bereits fertig; das Ergebnis ist 1.
2. Andernfalls:
   - 2.1 Berechne separat den Wert des vorletzten Monats.
   - 2.2 Berechne separat den Wert des letzten Monats.
   - 2.3 Das Ergebnis ist die Summe dieser beiden Werte.

```
int fib(int n){
  if (n<2) return 1;
  int x = fib(n-2);
  int y = fib(n-1);
  return x+y;}
```

Einige Beobachtungen

- Die erste Variante zur Berechnung der Fibonacci-Zahlen ist deutlich effizienter.

Einige Beobachtungen

- Die erste Variante zur Berechnung der Fibonacci-Zahlen ist deutlich effizienter.
- Der Unterschied liegt bereits in der algorithmischen Idee, nicht erst in der letztlichen Umsetzung als Programm begründet.

Einige Beobachtungen

- Die erste Variante zur Berechnung der Fibonacci-Zahlen ist deutlich effizienter.
- Der Unterschied liegt bereits in der algorithmischen Idee, nicht erst in der letztlichen Umsetzung als Programm begründet.

Algorithmenentwurf

- kreativer Akt
- „Kunst“
- schwer zu erlernen

vs.

Programmkonstruktion

- zum Teil „mechanisch“
- „Handwerk“
- leichter zu erlernen

Einige Beobachtungen

- Die erste Variante zur Berechnung der Fibonacci-Zahlen ist deutlich effizienter.
- Der Unterschied liegt bereits in der algorithmischen Idee, nicht erst in der letztlichen Umsetzung als Programm begründet.

Algorithmenentwurf

- kreativer Akt
- „Kunst“
- schwer zu erlernen

vs.

Programmkonstruktion

- zum Teil „mechanisch“
- „Handwerk“
- leichter zu erlernen

- Jedes Problem lässt sich (wenn überhaupt) durch verschiedene Algorithmen lösen.

Einige Beobachtungen

- Die erste Variante zur Berechnung der Fibonacci-Zahlen ist deutlich effizienter.
- Der Unterschied liegt bereits in der algorithmischen Idee, nicht erst in der letztlichen Umsetzung als Programm begründet.

Algorithmenentwurf

- kreativer Akt
- „Kunst“
- schwer zu erlernen

vs.

Programmkonstruktion

- zum Teil „mechanisch“
- „Handwerk“
- leichter zu erlernen

- Jedes Problem lässt sich (wenn überhaupt) durch verschiedene Algorithmen lösen.
- Jeder Algorithmus lässt sich durch verschiedene Programme umsetzen.

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Algorithmus:

- möglichst präzise beschriebener Plan/Prozess

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Algorithmus:

- möglichst präzise beschriebener Plan/Prozess
- effektiv ausführbare Einzelschritte

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Algorithmus:

- möglichst präzise beschriebener Plan/Prozess
- effektiv ausführbare Einzelschritte
- möglichst effizient

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Algorithmus:

- möglichst präzise beschriebener Plan/Prozess
- effektiv ausführbare Einzelschritte
- möglichst effizient
- oft informell repräsentiert
  (Diagramm, natürliche Sprache, Pseudocode)

Vom Problem zum Programm — Begrifflichkeiten

Problem:

- mit Rechnerhilfe zu lösende Aufgabenstellung
- abstrakte Spezifikation des zu erreichenden Ziels
- Spezifikation der Voraussetzungen

Algorithmus:

- möglichst präzise beschriebener Plan/Prozess
- effektiv ausführbare Einzelschritte
- möglichst effizient
- oft informell repräsentiert
  (Diagramm, natürliche Sprache, Pseudocode)

Programm:

- textuelle Repräsentation eines Algorithmus
- präzisiert alle Details des Algorithmus
  (so dass maschinell ausführbar)
- zwingend in einer formalen Sprache abgefasst

Genauere Klärung des Begriffs „Algorithmus“

Algorithmen:

- . . . sind durch endlichen Text vollständig beschrieben (Finitheit)

Genauere Klärung des Begriffs „Algorithmus“

Algorithmen:

- . . . sind durch endlichen Text vollständig beschrieben (Finitheit)
- . . . laufen in einzelnen, wohldefinierten Schritten ab (Effektivität)

Genauere Klärung des Begriffs „Algorithmus“

Algorithmen:

- . . . sind durch endlichen Text vollständig beschrieben (Finitheit)
- . . . laufen in einzelnen, wohldefinierten Schritten ab (Effektivität)
- . . . legen eindeutig fest, welcher Schritt auf einen vorangegangenen folgt (Determinismus)

Genauere Klärung des Begriffs „Algorithmus“

Algorithmen:

- . . . sind durch endlichen Text vollständig beschrieben (Finitheit)
- . . . laufen in einzelnen, wohldefinierten Schritten ab (Effektivität)
- . . . legen eindeutig fest, welcher Schritt auf einen vorangegangenen folgt (Determinismus)
- . . . kommen nach endlich vielen Schritten zum Schluss (Terminierung)

Genauere Klärung des Begriffs „Algorithmus“

Algorithmen:

- . . . sind durch endlichen Text vollständig beschrieben (Finitheit)
- . . . laufen in einzelnen, wohldefinierten Schritten ab (Effektivität)
- . . . legen eindeutig fest, welcher Schritt auf einen vorangegangenen folgt (Determinismus)
- . . . kommen nach endlich vielen Schritten zum Schluss (Terminierung)

Jedoch weicht man einige dieser Bedingungen mitunter auf und spricht dann von:

- nicht-deterministischen Algorithmen
- nicht-terminierenden Algorithmen

Jenseits der formalen Definition von „Algorithmus“

Weitere, „wünschenswerte“, Eigenschaften von Algorithmen:

- Korrektheit (erfüllt vorgegebene Ein-/Ausgabespezifikation)

Jenseits der formalen Definition von „Algorithmus“

Weitere, „wünschenswerte“, Eigenschaften von Algorithmen:

- Korrektheit (erfüllt vorgegebene Ein-/Ausgabespezifikation)
- Effizienz (verursacht möglichst wenig Aufwand)

Jenseits der formalen Definition von „Algorithmus“

Weitere, „wünschenswerte“, Eigenschaften von Algorithmen:

- Korrektheit (erfüllt vorgegebene Ein-/Ausgabespezifikation)
- Effizienz (verursacht möglichst wenig Aufwand)
- Verständlichkeit (kurzer Text, übersichtlich)

Jenseits der formalen Definition von „Algorithmus“

Weitere, „wünschenswerte“, Eigenschaften von Algorithmen:

- Korrektheit (erfüllt vorgegebene Ein-/Ausgabespezifikation)
- Effizienz (verursacht möglichst wenig Aufwand)
- Verständlichkeit (kurzer Text, übersichtlich)
- Anpassbarkeit (leicht veränderbar)

Jenseits der formalen Definition von „Algorithmus“

Weitere, „wünschenswerte“, Eigenschaften von Algorithmen:

- Korrektheit (erfüllt vorgegebene Ein-/Ausgabespezifikation)
- Effizienz (verursacht möglichst wenig Aufwand)
- Verständlichkeit (kurzer Text, übersichtlich)
- Anpassbarkeit (leicht veränderbar)

Mitunter widersprechen sich diese erwünschten Eigenschaften gegenseitig, bzw. müssen gegeneinander abgewogen werden.

Beschreiben von Algorithmen

—

Wahl einer Programmiersprache

Verschiedene Beschreibungsmöglichkeiten

Natürlichsprachlicher Ablaufplan:

Um den Wert des n. Monats zu berechnen:

1. Belege zwei Variablen x und y jeweils mit 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.
4. Das Ergebnis liegt zum Schluss in y bereit.

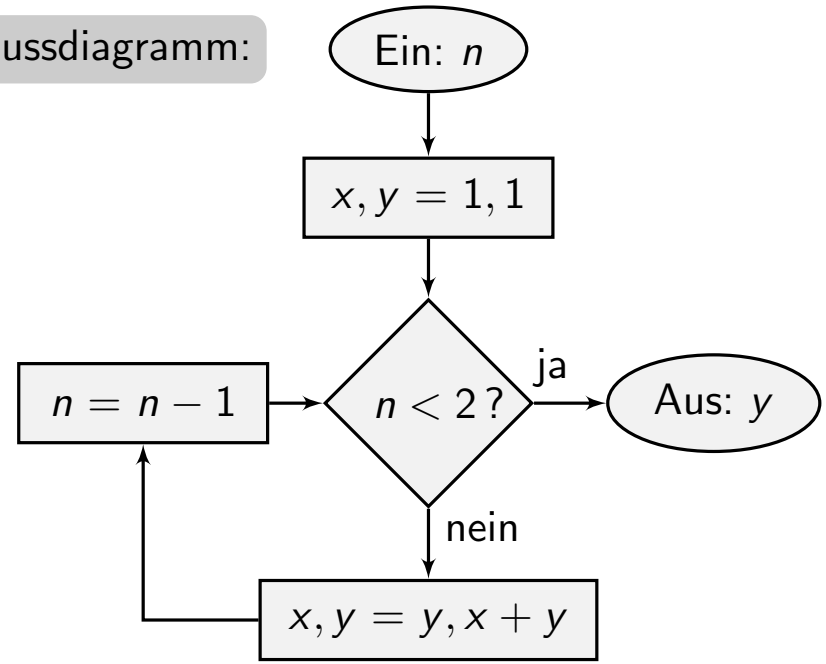
Verschiedene Beschreibungsmöglichkeiten

Natürlichsprachlicher Ablaufplan:

Um den Wert des n. Monats zu berechnen:

1. Belege zwei Variablen x und y jeweils mit 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.
4. Das Ergebnis liegt zum Schluss in y bereit.

Flussdiagramm:



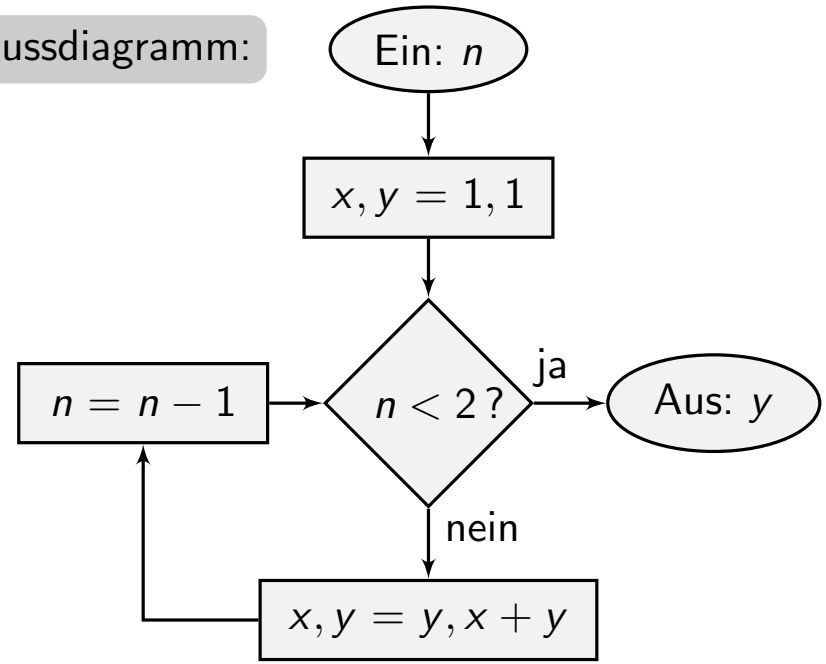
Verschiedene Beschreibungsmöglichkeiten

Natürlichsprachlicher Ablaufplan:

Um den Wert des n. Monats zu berechnen:

1. Belege zwei Variablen x und y jeweils mit 1.
2. Wenn $n < 2$, dann bereits fertig.
3. Andernfalls:
   - 3.1 Belege x und y neu mit y bzw. $x + y$.
   - 3.2 Das Ganze $n - 1$ mal.
4. Das Ergebnis liegt zum Schluss in y bereit.

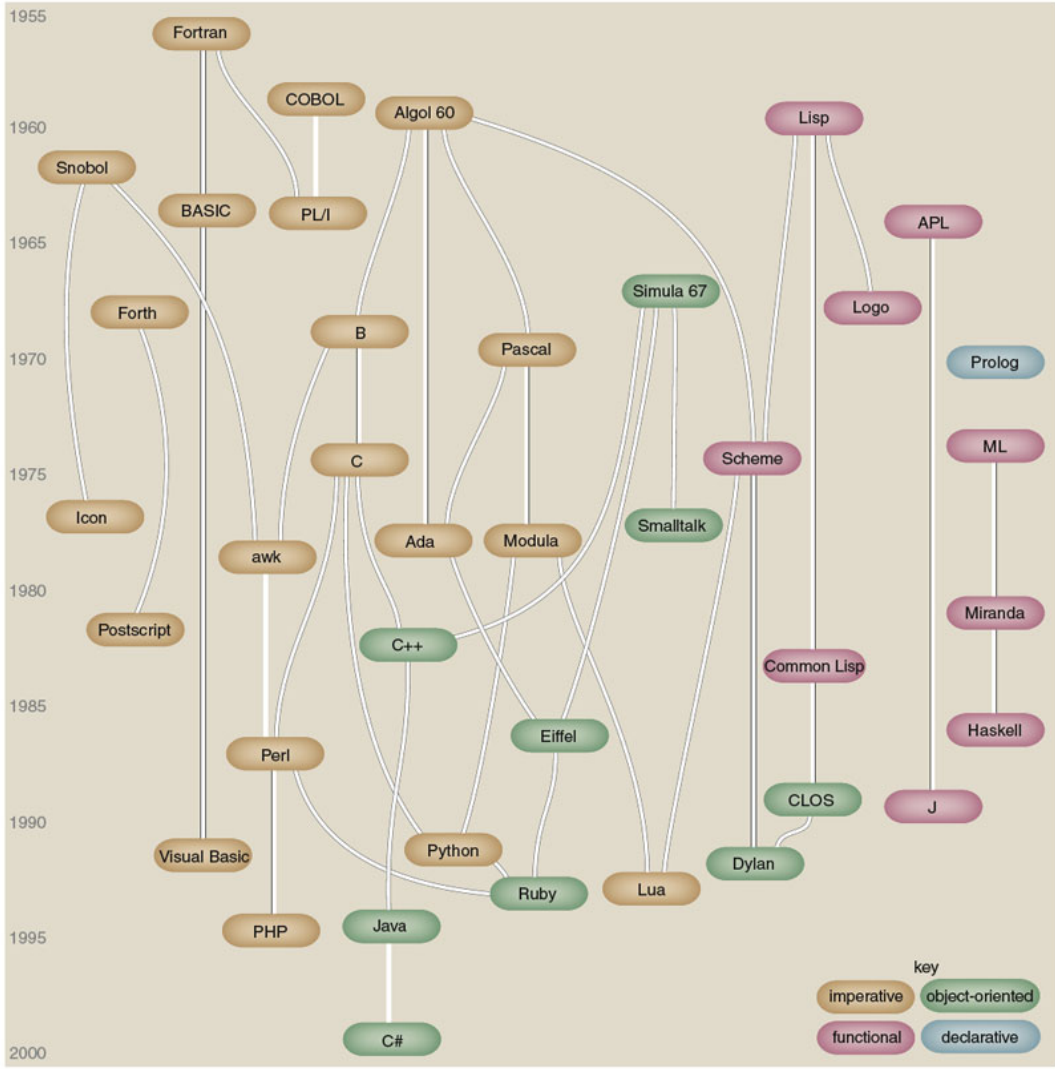
Flussdiagramm:



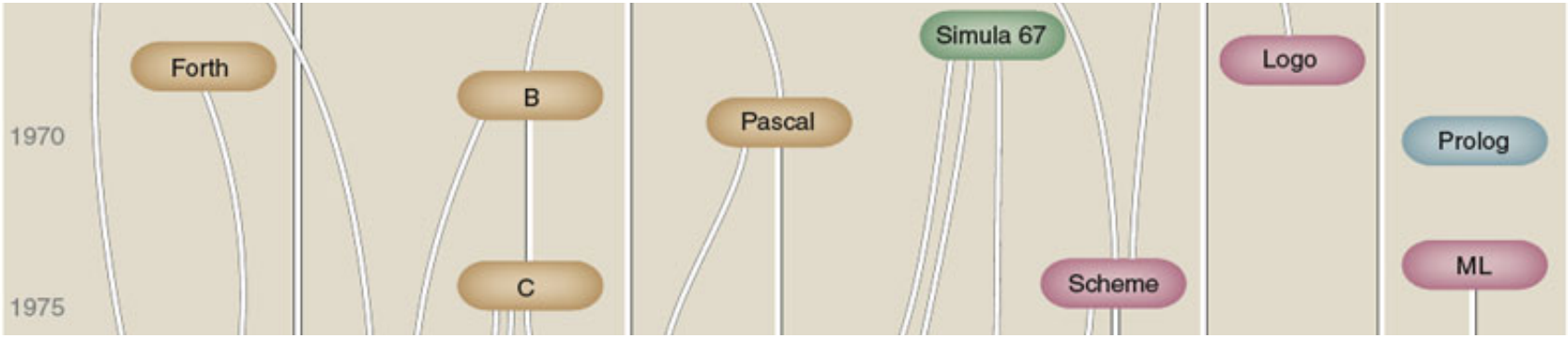
Programm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

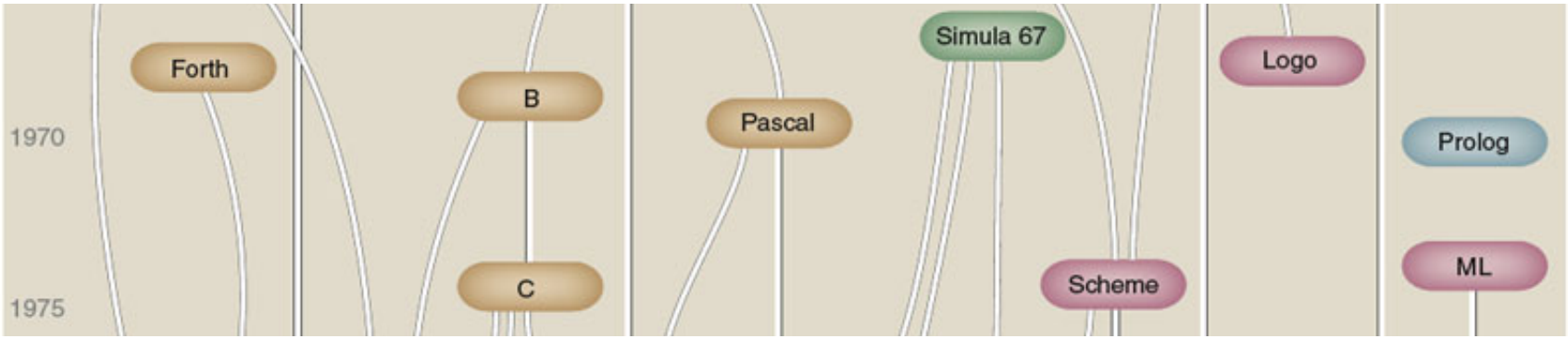
Eine konkrete Programmiersprache: C



Eine konkrete Programmiersprache: C



Eine konkrete Programmiersprache: C



- 1972 entwickelt von Dennis Ritchie an den Bell Labs
- entscheidend für die Entwicklung von UNIX
- maschinennah, explizite Speicherverwaltung
- trotzdem portabel, weit verbreitet
- einfacher Aufbau, geringe Anzahl verschiedener Konzepte
- imperativ!

Genaugenommen: „Disciplined C“

Vereinfachungen/Einschränkungen für die Vorlesung:

- Verzicht auf einige Kontrollstrukturen
- keine Pointer-Arithmetik
- etwas strengere Typdisziplin
- keine Makros/Präprozessor

Genaugenommen: „Disciplined C“

Vereinfachungen/Einschränkungen für die Vorlesung:

- Verzicht auf einige Kontrollstrukturen
- keine Pointer-Arithmetik
- etwas strengere Typdisziplin
- keine Makros/Präprozessor

Fokus der Vorlesung auf:

- Prinzip imperativer Kontrollstrukturen
- allgemeine Programmierkonzepte
- algorithmische Prinzipien
- wichtige Datenstrukturen

Genaugenommen: „Disciplined C“

Vereinfachungen/Einschränkungen für die Vorlesung:

- Verzicht auf einige Kontrollstrukturen
- keine Pointer-Arithmetik
- etwas strengere Typdisziplin
- keine Makros/Präprozessor

Fokus der Vorlesung auf:

- Prinzip imperativer Kontrollstrukturen
- allgemeine Programmierkonzepte
- algorithmische Prinzipien
- wichtige Datenstrukturen

Nicht, zum Beispiel:

- Modularisierung
- Perlen der Algorithmik

1.		Logik u. Diskrete Strukturen	Technische Informatik	Informations-systeme	Imperative Programmierung	Techniken des wissenschaftlichen Arbeitens
2.	Analysis	Lineare Algebra	Systemnahe Informatik		Objektorientierte Softwareentwicklung	
3.	Angewandte Mathematik	Algorithmen u. Berechnungs-komplexität I		Software-technologie	Systemnahe Programmierung	
4.		Algorithmen u. Berechnungs-komplexität II			Wahlpflicht: Deskriptive Programmierung	
5.						**Projektgruppe** - Seminar - Praktikum
6.						**Bachelormodul** - B.arbeit - B.seminar

28/30

30

30/28

6

+ mehrere Abhängigkeiten bei Nebenfach- und Wahlpflichtmodulen

Syntax von Programmiersprachen

Syntaxbeschreibung

Zur Erinnerung, (Teil von einem) C-Beispielprogramm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Syntaxbeschreibung

Zur Erinnerung, (Teil von einem) C-Beispielprogramm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Wichtige Fragen:

- Aus welchen „Bausteinen“ wird ein Programm zusammengesetzt?

Syntaxbeschreibung

Zur Erinnerung, (Teil von einem) C-Beispielprogramm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Wichtige Fragen:

- Aus welchen „Bausteinen“ wird ein Programm zusammengesetzt?
- Nach welchen Regeln werden diese systematisch kombiniert?

Syntaxbeschreibung

Zur Erinnerung, (Teil von einem) C-Beispielprogramm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Wichtige Fragen:

- Aus welchen „Bausteinen“ wird ein Programm zusammengesetzt?
- Nach welchen Regeln werden diese systematisch kombiniert?
- Wie lässt sich dies formal beschreiben (und intuitiv verstehen)?

Syntaxbeschreibung

Zur Erinnerung, (Teil von einem) C-Beispielprogramm:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Wichtige Fragen:

- Aus welchen „Bausteinen“ wird ein Programm zusammengesetzt?
- Nach welchen Regeln werden diese systematisch kombiniert?
- Wie lässt sich dies formal beschreiben (und intuitiv verstehen)?
- Wie kann ein Compiler an Hand einer solchen Beschreibung feststellen, ob ein Programm überhaupt legal ist?

Etwas pragmatischer ausgedrückt:

Wir suchen eine Möglichkeit, um etwa systematisch sinnvolle C-Programmteile wie:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

Etwas pragmatischer ausgedrückt:

Wir suchen eine Möglichkeit, um etwa systematisch sinnvolle C-Programmteile wie:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) {z=x; x=y; y=z+y;}
  return y;}
```

zu unterscheiden von „irgendwelchen“ Zeichenketten wie:

```
int fib(int n){
  int x=1, y=1, i, z;
  if (n>=2; i=i+1)
    for (i=1; i<n) {z=x; x=y; y=z+y;}
  return y;}
```

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Sprache: Teilmenge von Σ^*

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Sprache: Teilmenge von Σ^*

Metasprache: Sprache zur Beschreibung einer anderen Sprache

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Sprache: Teilmenge von Σ^*

Metasprache: Sprache zur Beschreibung einer anderen Sprache;
in der Regel über verschiedenen Alphabeten

Klärung einiger Begriffe / Definitionen

Alphabet: eine nichtleere, endliche Menge Σ;
Elemente heißen Symbole

Wort: eine endliche Folge von Symbolen aus Σ

- leeres Wort: Folge der Länge 0; Notation: ε

Σ^*: Menge aller Worte (über Alphabet Σ)

- $\varepsilon \in \Sigma^*$
- wenn $w \in \Sigma^*$ und $\sigma \in \Sigma$, dann $w\sigma \in \Sigma^*$
- keine weiteren Worte in Σ^*

Konkatenation: Operation, die zwei Worte aneinanderhängt

Sprache: Teilmenge von Σ^*

Metasprache: Sprache zur Beschreibung einer anderen Sprache;
in der Regel über verschiedenen Alphabeten

Objektsprache: durch eine andere Sprache beschriebene Sprache

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort:

Σ^*:

Konkatenation:

Sprache:

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*:

Konkatenation:

Sprache:

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Konkatenation:

Sprache:

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Konkatenation: $ab \cdot cca = abcca$

Sprache:

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Konkatenation: $ab \cdot cca = abcca$

Sprache: $\{a^n bc^n \mid n \geq 0\}$

Klärung einiger Begriffe / Beispiele

Alphabet: $\Sigma = \{a, b, c\}$

Wort: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Konkatenation: $ab \cdot cca = abcca$

Sprache: $\{a^n bc^n \mid n \geq 0\}$

Metasprache: ???