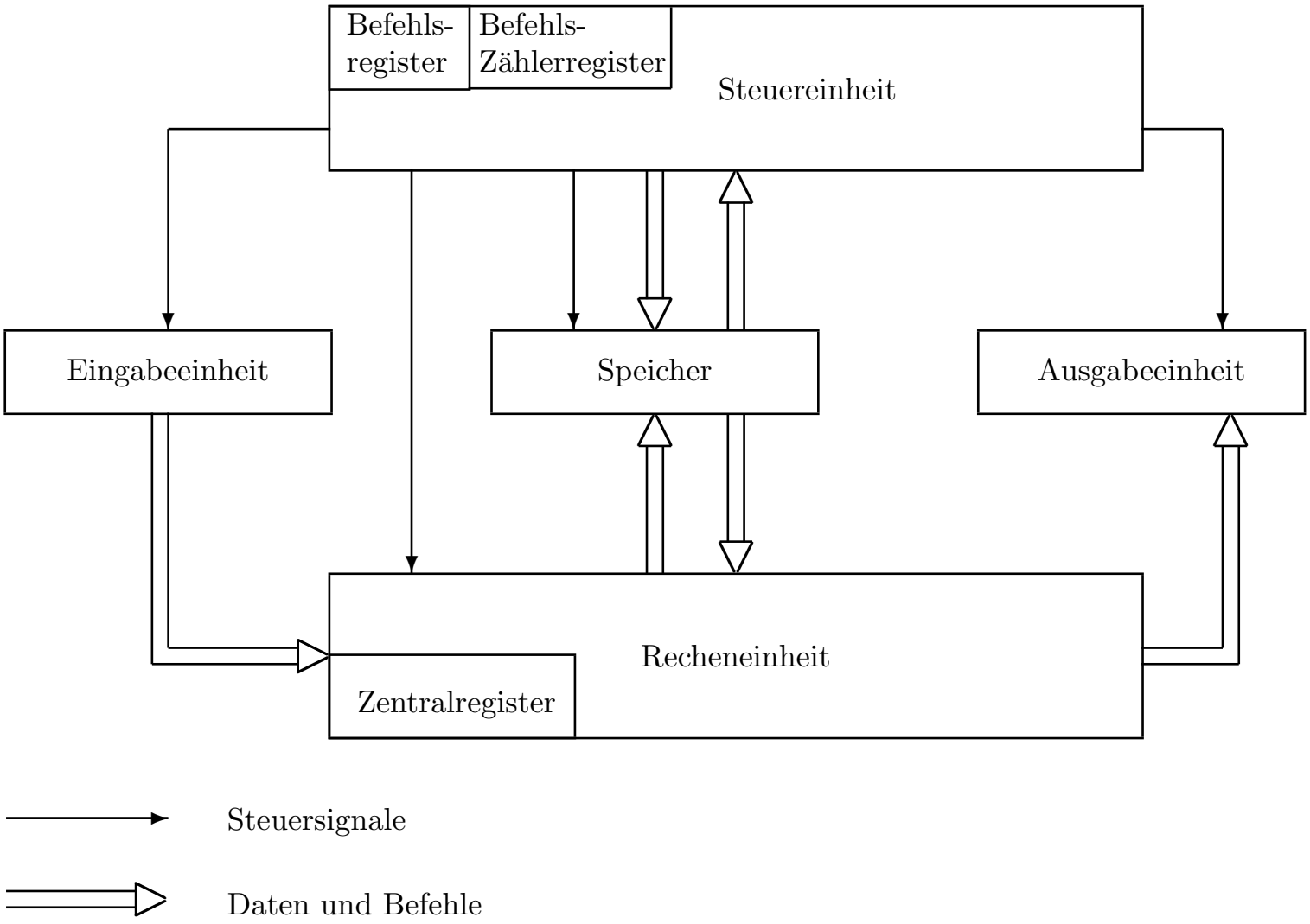
# §1 Einführung – Programmiersprachen

*Leitidee:* *Von der Maschinensprache zur höheren Programmiersprache und zurück*

- Von-Neumann-Maschine als abstraktes Computermodell
- Maschinensprache des hypothetischen Rechners M
- Höhere Programmiersprachen
  - Konzept
  - Praktisches Vorgehen: Editieren, Übersetzen, Starten
  - Begriffe: Variable, Funktion, Datentyp
  - Syntax und Semantik (Vergleich mit Schriftsprache)
  - Syntaxdiagramme zur Beschreibung der Syntax

# Rechnermodell nach J. von Neumann



# Rechnermodell nach J. von Neumann - Fortsetzung

**Komponenten**

- *Rechnerkern (CPU)* – bestehend aus

  *Recheneinheit (ALU, arithmetic logical unit)*
  führt Arithmetik- und Logikbefehle aus

  *Steuereinheit (control unit)*
  dekodiert Programmbefehle
  steuert Programmausführung

- *Speicher* – enthält die Daten (binär kodiert) in Speicherzellen, die jeweils nummeriert (addressierbar) sind
  NEU: Programm befindet sich im Speicher
  (Unterschied zur Zuse-Maschine!)

- *Ein- und Ausgabeeinheit (Peripherie)* – z.B. Tastatur, Bildschirm, Drucker u.ä.

# Programmausführung

- Befehlszählerregister der Steuereinheit enthält die Adresse des Speicherplatzes mit dem nächsten Befehl in binär codierter Form.
- Steuereinheit lädt Befehl in das Befehlsregister, entschlüsselt ihn und führt ihn aus.
- Befehlszählerregister wird um 1 erhöht oder bei Sprungbefehlen auf die gewünschte Adresse gesetzt.

  Usw.

  *Fetch-Decode-Execute Cycle*

# Hypothetische Maschine, Maschinensprache

- Hypothetische Maschine M und ihre Maschinensprache als stark idealisiertes Modell eines Prozessors und seines Befehlssatzes
- Prinzip wichtig für das Verständnis von Progr.sprachen (Programme in höheren Programmiersprachen werden in mehreren Schritten in Maschinenprogramme „übersetzt“.)
- [$*$] Realistischere Modelle: `CPUsim`, `SPIM` [MIPS-Emulator]
- [$*$] Weiterführende Literatur:
  Tanenbaum, Goodman: Computerarchitektur
  Patterson, Hennesy: Computer Organisation & Design

# Höhere Programmiersprachen

**Nachteile von Programmen in Maschinensprache**

- sehr aufwendig bei komplexeren Aufgabenstellungen
- fehleranfällig
- schwer verständlich
- nicht portabel, d.h. nicht auf Rechner anderer Bauart übertragbar, insbesondere prozessorabhängig

**Abhilfe durch höhere Programmiersprachen**

- *Konzept:* Quellprogramm $\overset{\text{Übersetzer}}{\Longrightarrow}$ Maschinenprogramm
  - Quellprogramm: geschrieben in höherer Prog.sprache
  - Übersetzer (Compiler): Maschinenprogramm, das das Quellprogramm einliest und daraus ein Maschinenprogramm erzeugt
- Von Neumannsche Rechnerarchitektur hier vorteilhaft!

# Höhere Programmiersprachen II

**Typische Arbeitsschritte** (am Beispiel Linux)

1. Erstellen des Quellprogramms mit einem Editor in einer Datei (mit fester Endung), z.B. `kwrite datei.cpp`
2. Übersetzen (Compilieren), z.B. `c++ datei.cpp`
3. Ausführen des Maschinenprogramms (in der Regel durch Namensaufruf), z.B. `./a.out`

**Variablenbegriff und Ausdrucksanw. in C++** *(vorläufig)*

- Eine Variable ist ein Name für einen Speicherplatz.
- Compiler legt konkrete Speicherplätze für Variablen fest.
- Variablen müssen vor Verwendung vereinbart werden.
- Syntax von Programmiersprachen ist angelehnt an math. Notation, jedoch gibt es Bedeutungsunterschiede:
  `s = (a+b+c)/2;` Speichere auf $s$ den Wert von $\frac{a+b+c}{2}$
  `i = i+1;` Erhöhe den Wert von $i$ um 1

# Höhere Programmiersprachen III

## Funktionen in C++

- Eine Reihe vordefinierter Funktionen ist vorhanden, etwa mathematische Funktionen, z.B. `sqrt`, `abs`
- Funktionsbegriff allgemeiner als in der Mathematik:
  - *Funktionen* sind abgegrenzte Codebereiche, in die beim Funktionsaufruf gesprungen wird.
  - Bei Beendigung der Funktion Rücksprung hinter die aufrufende Stelle.
  - Kommunikation mit der Funktion (u.a.) durch Argumentübergabe und Rückgabe des Funktionswerts.
  - C++-Funktionen ohne Funktionswert zulässig.
  - Funktionen mit leerer Argumentliste möglich.
  - Funktion `main` in C++-Programm erforderlich, automatischer Aufruf beim Programmstart, Rückgabewert von `main` geht an Kommandointerpreter (nicht an das Programm!)
  - Verwendung vordefinierter Funktionen muss angegeben werden, in der Regel mit include-Anweisungen, z.B. `#include <cmath>`

# Höhere Programmiersprachen IV

## Datentypen in C++

- *Datentypen* geben an, wie z.B. in Variablen gespeicherte Bitmuster zu interpetieren sind und welche Operationen zulässig sind
- Große Anzahl unterschiedlicher Datentypen, z.B für Zahlen, Zeichen, Zeichenketten, Adressen, Vektoren usw.:
  `string` Zeichenkettentyp von C++
- Unterschiedliche Datentypen und arithmetische Operatoren für Zahlen:

  `int` Ganze Zahlen (exakte Arithmetik)
  `double` Gleitkommazahlen (ggf. Rundung)

  Zahlbereiche immer beschränkt, Größenbereich und interne Darstellung sehr unterschiedlich *(später!)*

# Höhere Programmiersprachen V

## Syntax und Semantik

**Syntax** beschreibt, aus welchen Zeichenketten ein *strukturell* korrektes Programm aufgebaut ist.
Syntax von Programmiersprachen $\approx$ Grammatik und Orthographie geschriebener Sprachen
Compiler meldet Syntaxfehler!

**Semantik** gibt die *inhaltliche* Bedeutung eines Programms an.
Compiler kann bei bestimmten Konstrukten warnen, die syntaktisch korrekt, aber inhaltlich vermutlich falsch sind.

- Zweistufige Beschreibung der Syntax von C++ :
  1. Bildung von *Eingabesymbolen* aus Zeichen $\approx$ Bildung von Worten und Satzzeichen aus Schriftzeichen in natürlichen Sprachen (lexikalische Struktur)
  2. Bildung strukturell zulässiger Programme aus Eingabesymbolen $\approx$ Bildung grammatikalisch und orthographisch korrekter Texte in natürlichen Sprachen

# Höhere Programmiersprachen VI

## Syntaxdiagramme

- *Syntaxdiagramme* visualisieren die Bildung strukturell zulässiger Programme aus Eingabesymbolen
- Durchlauf der Syntaxdiagramme in Pfeilrichtung, jeder zulässige Pfad erzeugt eine syntaktisch korrekte Aufeinanderfolge von Eingabesymbolen
- Kennzeichnung von Eingabesymbolen mit *abgerundeten* Blöcken, falls nicht durch Syntaxdiagramme sinnvoll weiter zerlegbar („Endsymbole“)
- *Eckige* Blöcke werden für Syntaxdiagramme verwendet, die keine Endsymbole sind.
- Syntaxdiagramme sind eine graphische Darstellung der Extended Backus Naur Form (EBNF)
- Syntaxdiagramme in C++ beziehen sich auf die Situation *nach* Ausführung der Präprozessoranweisungen