

Grundlagen der Programmierung

Vom Programm zum Prozess:
Assembler ♦ Interpreter ♦ Compiler

Algorithmisches Denken: Vom Problem zur Lösung

1. Identifizieren des Problems
2. Formulieren des Problems
3. Entwurf des Algorithmus
4. Implementierung des Algorithmus
5. Anwendung des Algorithmus



*Vom Problem
zum Algorithmus*

→ Problemlösung

- Ausführung des Programms auf einer Maschine
- Betriebssystem (Kern) erzeugt Prozess

Recap: Wesentliche Komponenten der Hardware (von-Neumann-Architektur)

■ Zentraleinheit

■ Prozessor

mit Registern, in denen Zahlenwerte gespeichert und vom Prozessor verändert werden können

■ Arbeitsspeicher/Primärspeicher

zur Datenhaltung außerhalb der Register

■ Busse (Leitungen) zwischen beiden



■ Peripheriegeräte (zur Ein-/Ausgabe von Daten)

■ Tastatur, Maus, Bildschirm, Drucker

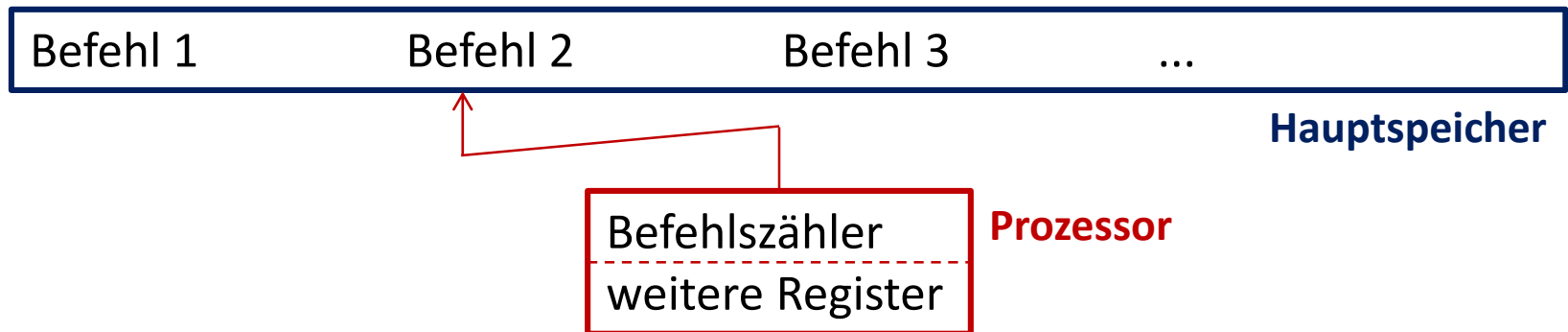
■ Sekundärspeicher (Festplatte, CD-, DVD-, Flashlaufwerke)

■ Netzwerkinterface, ...

■ Verbindungsleitungen

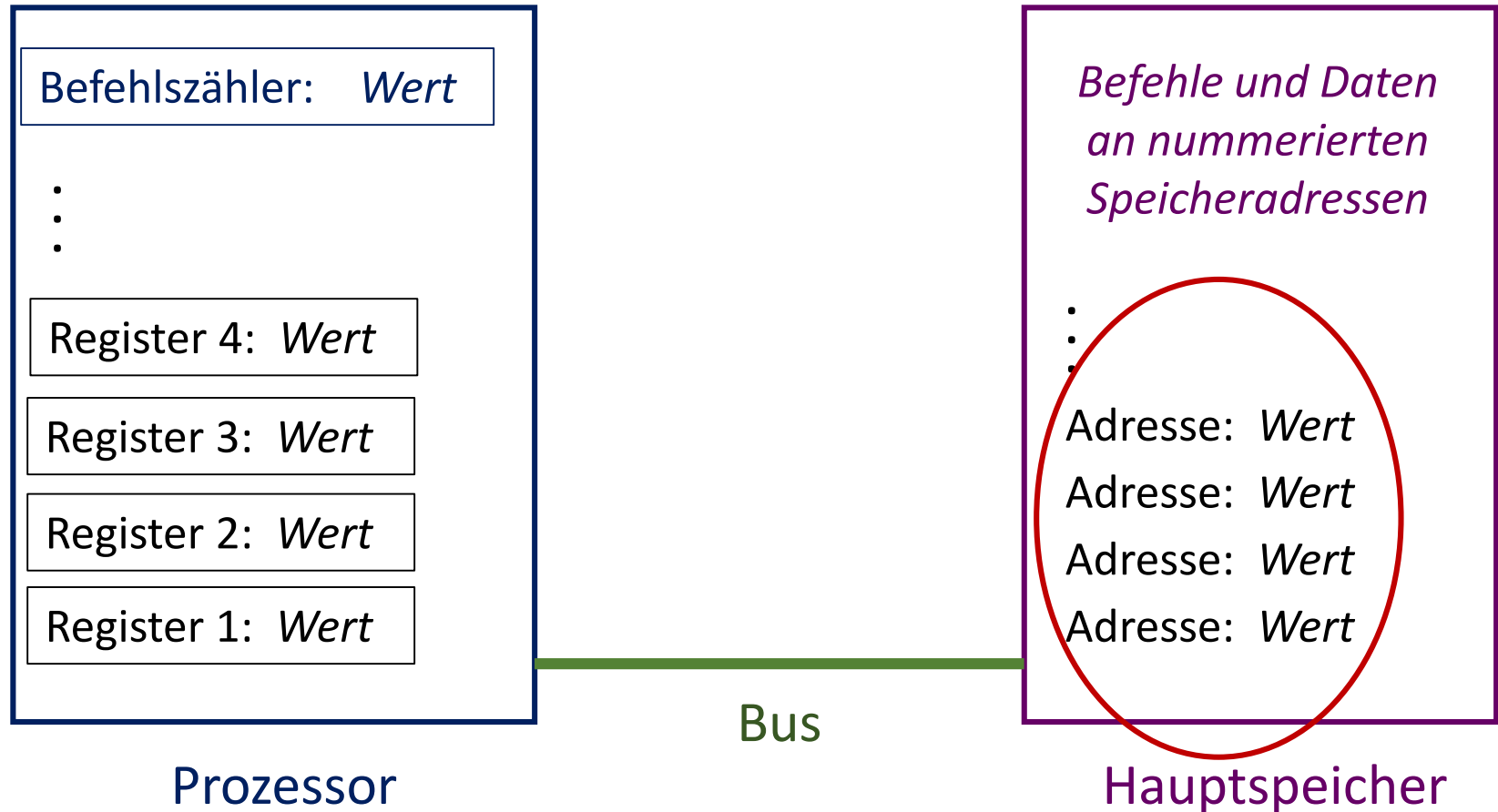
Recap: Programmausführung

Der Prozessor enthält genau einen **Befehlszähler**, der stets die Adresse der nächsten auszuführenden Anweisung enthält.



- Programm muss als **lineare Sequenz** von Befehlen vorliegen, die vom Prozessor „verstanden“ werden
- **Übersetzung** der Programmabstraktionen in Prozessorbefehle

Architektur der Zentraleinheit



Adressen und Werte im Hauptspeicher

- linear angeordnete Speicherzellen
- i.d.R. 1 Byte groß
- mit ganzen Zahlen ab 0 in Hexadezimaldarstellung (**Adressen**) durchnummeriert
- an Adresse 0 nie Programmdateien
- Speicher kann als „indizierte Liste“ **Mem** aufgefasst werden
- Adressen sind dann „Indizes“

37d4aa

37d4a9

37d4a8

	⋮
	Speicherzelle für 1 Byte
	Speicherzelle für 1 Byte
	Speicherzelle für 1 Byte
	⋮
3	Speicherzelle für 1 Byte
2	Speicherzelle für 1 Byte
1	Speicherzelle für 1 Byte
0	Speicherzelle für 1 Byte

Hauptspeicher

Adressen und Werte im Hauptspeicher

- linear angeordnete Speicherzellen
- i.d.R. 1 Byte groß
- mit ganzen Zahlen ab 0
in Hexadezimaldarstellung
(**Adressen**) durchnummeriert
- an Adresse 0 nie Programmdateien
- Speicher kann als
„indizierte Liste“ **Mem**
aufgefasst werden
- Adressen sind dann „Indizes“

37d4aa

37d4a9

37d4a8

3

2

1

0

⋮
Mem[37d4aa]
Mem[37d4a9]
Mem[37d4a8]
⋮
Mem[3]
Mem[2]
Mem[1]
Mem[0]

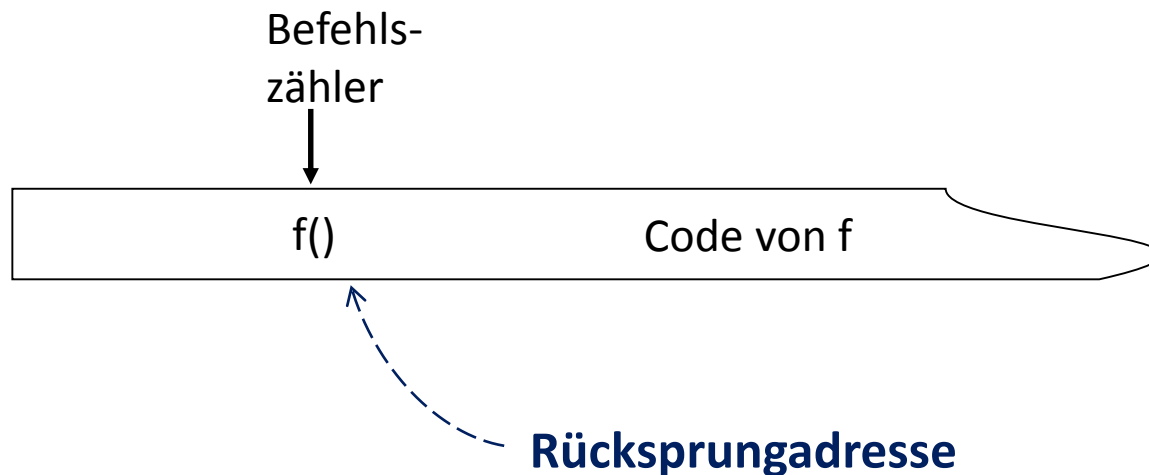
Werte

Prozessorbefehle

- Prozessor speichert Werte in Registern
- Modifikation von Werten nur in bestimmten Registern
- Befehle
 - Laden von Werten in Register
 - Speichern von Registerwerten im Hauptspeicher
 - Verändern von Registerwerten durch Anwendung (arithmetischer) Operationen
 - Befehlszähler-Operationen
 - Sprünge
 - bedingte Sprünge
- Liste derartiger Befehle, durchnummeriert

Warum Sprünge?

- z.B. Verzweigungen und Schleifen
- z.B. Funktionsaufrufe



MIPS

→ Maschinenmodelle

- Befehlssatz für bestimmte RISC-Prozessoren
- in traditionellen UNIX-Workstations und -Servern, Routern, Spielekonsolen, ...
- Befehle verwenden als Operanden
 - Werte in Registern (z.B. `$s1`, `$s2`, ..., `$ra`, `$zero`, ...)
 - Werte an *virtuellen (logischen)* Speicheradressen (z.B. `Mem[4]`, `Mem[8]`, ..., `Mem[2016]`, ...)
 - bezeichnet durch Integer-Zahl (teilbar durch 4)
 - müssen dann (vom Betriebssystem) auf *physische* Speicheradressen abgebildet werden

Beispiele MIPS-Befehle

`add $s1, $s2, $s3`

Setzt Register \$s1 auf Summe der Werte in Registern \$s2 und \$s3

`lw $s1, 20($s2)`

Lädt Wert Mem[\$s2+20] in Register \$s1

`sw $s1, 20($s2)`

Speichert Wert aus Register \$s1 an Speicheradresse \$s2+20

\$s2 - *eigentlich Wert in \$s2*

Einige MIPS Sprungbefehle

`jump 2500`

Setzt Befehlszähler auf 10000

`jump $ra`

Setzt Befehlszähler auf den Wert, der in Register `$ra` gespeichert ist

`beq $s1, $s2, 25`

Erhöht den Befehlszähler um 4+100, falls die Werte in `$s1` und `$s2` gleich sind, sonst um 4

Einige LEGv8-Befehle

- für Prozessoren mit ARM-Architektur, z.B. in Smartphones, Tablet-PCs, Routern, ...
- 32 Register X0-X31; XZR
- Beispiele für Befehle:

ADD X1, X2, X3

ADDI X1, X2, #20

LDUR X1, [X2, #40]

STUR X1, [X2, #40]

B 25

CBZ X1, 25

...

SUB X1, X2, X3

SUBI X1, X2, #20

Assembler- und Maschinensprache

■ Assemblersprache

Befehlssprache für eine Prozessor-Architektur

- **textliche** Beschreibung der Befehlsfolge
- abhängig von der Prozessor-Architektur
- *Beispiel: MIPS assembly language* (s. [MIPS Befehlssatz](#))

■ Maschinensprache ... *im Wesentlichen:*

- **binär** kodierte Beschreibung der Befehlsfolge
- Bytes oder Byte-Sequenzen kodieren sowohl Befehle eines konkreten Prozessors als auch Daten

■ **Assembler** übersetzt Assembler- in Maschinencode

AASS

- **Abstrakte Assemblersprache**
- arbeitet nur mit ganzen Zahlen
(kann durch Kodierungen immer erreicht werden)
- die wichtigsten, typischen Assembler-Operationen unabhängig von Prozessorarchitekturen
- beliebige aber feste Anzahl von Registern
 - **r1, r2, ... , rn** mit veränderlichen Werten ***r1, *r2, ... , *rn**
(kann durch Auslagern in den Speicher auf konkrete Anzahl beschränkt werden)
 - **r0** enthält den unveränderlichen Wert 0
 - **rb** enthält den Befehlszähler

Speicherbelegungsplan

- Zuordnung einer nicht negativen ganzen Zahl (**virtuelle Speicheradresse**) zu jeder Variablen:

Variable	x	y	var	k
Adresse	4	8	12	16

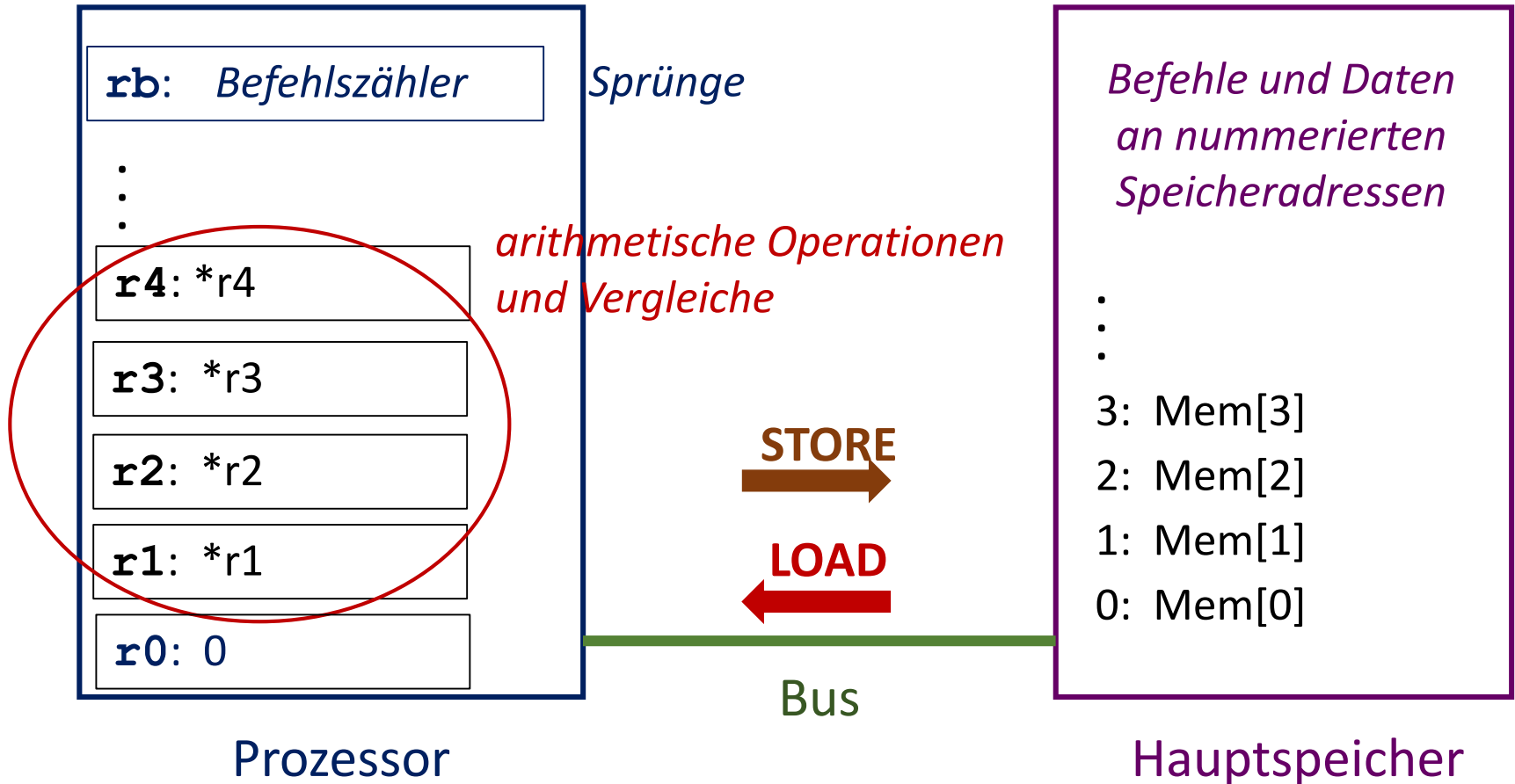
Werte: **Mem[4]** **Mem[8]** **Mem[12]** **Mem[16]**

- beliebige Adressen sind erlaubt
- nur ganze Zahlen können gespeichert werden
- *Beispiel:* **x=17** führt zu **Mem[4]=17**

AASS-Programm

- **durchnummerierte** Liste von **AASS-Befehlen** (ab 1)
- *hier*: keine Interaktion mit dem Benutzer oder Dateien
- Eingabewerte stehen im Speicher (Variablenwerte)
- **rb** (Befehlszähler) hat den Initialwert 1
- alle anderen Register haben den Initialwert 0
- Befehlszähler wird
 - entweder durch den Befehl auf einen neuen Wert gesetzt (Sprung-Befehl)
 - oder automatisch um 1 erhöht (weiter mit nächstem Befehl)
- Programmende gdw. **STOP**-Befehl erreicht

AASS - Befehle



AASS-Befehle (1)

1. Speicherzugriffe

<i>Syntax</i>	<i>Beispiel</i>	<i>Wirkung (Semantik)</i>
LOAD <i>Register Mem</i>	LOAD r1 [4]	*r1 = Mem[4]
	LOAD r1 [r4]	*r1 = Mem[*r4]
STORE <i>Register Mem</i>	STORE r1 [8]	Mem[8] = *r1
	STORE r1 [r5]	Mem[*r5] = *r1
		<i>indirekte Adressierung</i>

2. Laden von Konstanten

SET <i>Register Wert</i>	SET r1 4	*r1 = 4
--------------------------	----------	---------

AASS-Befehle (2)

3. Arithmetische Operationen

ADD <i>Ziel Quellen</i>	ADD r3 r1 r2	$*r3 = *r1 + *r2$
SUB <i>Ziel Quellen</i>	SUB r3 r1 r2	$*r3 = *r1 - *r2$
MUL <i>Ziel Quellen</i>	MUL r3 r1 r2	$*r3 = *r1 \cdot *r2$
DIV <i>Ziel Quellen</i>	DIV r3 r1 r2	$*r3 = *r1 / *r2$

Ziel ist ein Register, *Quellen* sind Register oder positive Konstanten:

ADD r3 r1 2	$*r3 = *r1 + 2$
--------------------	-----------------

4. Unbedingter Sprung

GOTO <i>Wert</i>	GOTO 7	Befehlszähler = 7
		(weiter mit Befehl Nummer 7)

AASS-Befehle (3)

5. Bedingte Sprünge

GOEQ *Vergleichsregister Wert* **GOEQ** **r1** **r2** **7**
→ falls ***r1** == ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

GOLS *Vergleichsregister Wert* **GOLS** **r1** **r2** **7**
→ falls ***r1** < ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

GOGP *Vergleichsregister Wert* **GOGP** **r1** **r2** **7**
→ falls ***r1** > ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

AASS-Befehle (4)

GOLE *Vergleichsregister Wert* **GOLE** **r1** **r2** **7**
→ falls ***r1** **<=** ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

GOG *Vergleichsregister Wert* **GOG** **r1** **r2** **7**
→ falls ***r1** **>=** ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

GONE *Vergleichsregister Wert* **GONE** **r1** **r2** **7**
→ falls ***r1** **!=** ***r2**, dann Befehlszähler = 7,
sonst um 1 erhöhen

STOP Programmende

*Text* Kommentar

Sprünge mit indirekter Adressierung

GOTO r1 Setzt Befehlszähler auf *r1

GOEQ r1 r2 r3 Setzt bei *r1==*r2 Befehlszähler auf *r3,
sonst wird er um 1 erhöht

GOLS, GOGR, GOLE, GOGF und GONE analog

AASS – Beispiel (1)

Speicherbelegung:

<i>Variable</i>	x	y
<i>Adresse</i>	4	6

Befehlsnummer

1 LOAD r1 [4]
2 LOAD r2 [6]
3 SET r3 1

rb	1	2	3	
r1	0	2	2	2
r2	0	0	3	3
r3	0	0	0	1

x = 2, y = 3

AASS – Beispiel (1)

Speicherbelegung:

Variable	x	y
Adresse	4	6

Befehlsnummer

```

1 LOAD r1 [4]
2 LOAD r2 [6]
3 SET r3 1
4 GOEQ r2 r0 8      # falls y = 0 Sprung zu 8
5 MUL r3 r3 r1      # Multiplikation mit x
6 SUB r2 r2 1       # *r2 um 1 vermindern
7 GOTO 4
8 STORE r3 [4]
9 STOP
  
```

realisiert $x = x * y$ für $y \geq 0$ sonst Endlosschleife!!!

AASS - Beispiel indirekte Adressierung

- Berechnung der Summe der Elemente einer Liste mit Länge $n > 0$

<i>Variable</i>	n	L[0]	L[1]	...	L[n-1]	sum
<i>Adresse</i>	2	4	8	...	4n	4n+4

- *Entwurf:*
 - in einem Register für alle i ($1 \leq i \leq n$) $4i$ berechnen
 - Laden von Mem[4i]
 - Iteriert diese Werte addieren, beginnen mit 0
 - ... bis Anzahl der addierten Werte gleich Länge der Liste

AASS - Beispiel indirekte Adressierung

```
1 SET r1 4           # initiale Speicheradresse
2 SET r2 0           # Anzahl addierter Elemente
3 LOAD r3 [2]        # Laenge der Liste
4 GOEQ r2 r3 10       # alle Elemente addiert?
5 LOAD r4 [r1]        # aktuelles Listenelement
6 ADD r5 r5 r4        # Summe wird in r5 gebildet
7 ADD r2 r2 1         # ein Element mehr addiert
8 ADD r1 r1 4         # naechste Speicheradresse
9 GOTO 4
10 STORE r5 [r1]      # speichern in Variable sum
11 STOP
```

Hoch- *versus* Assemblersprachen

■ Hochsprachen

- benutzen Abstraktionen, die dem algorithmischen Denken entsprechen, z.B.:
Schleifen, Prozeduren, Funktionen, Listen, ...
- *Beispiele:* Python, C, C++, Java, C#, PASCAL, ...

■ Assemblersprachen

- benutzen Befehle, die der Prozessorsteuerung dienen
- weniger abstrakt, näher an der Technik
- unübersichtlicher, größere Fehlergefahr

Die Übersetzer

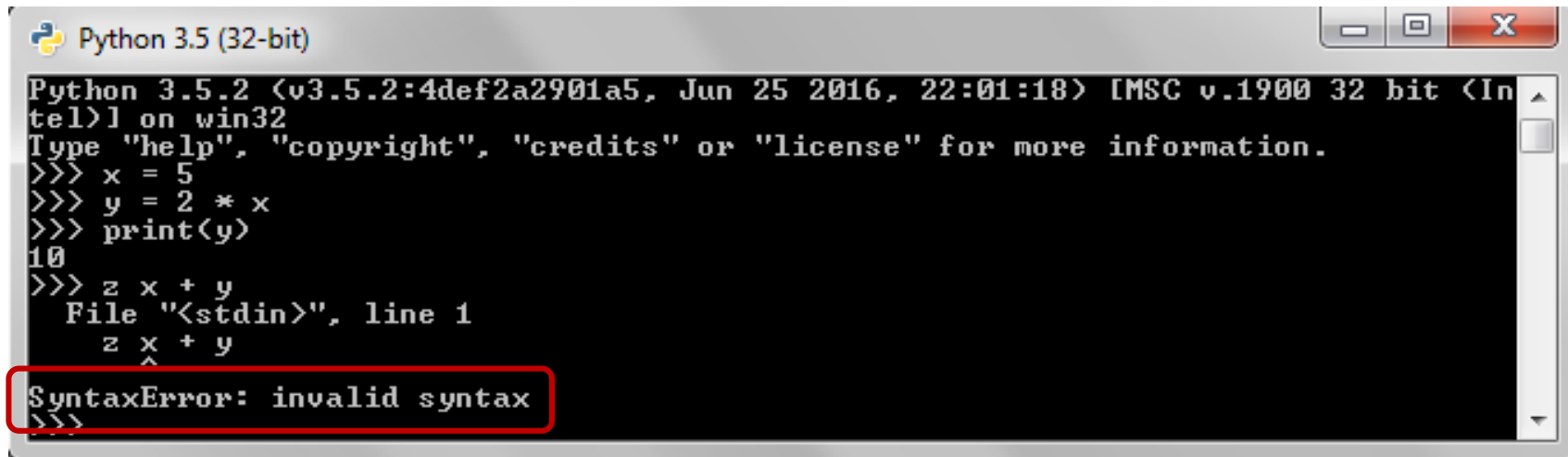
- Lösung: automatische Übersetzung von Hochsprache in Assembler- oder Maschinensprache
- **Interpreter**
 - Übersetzung **zeilenweise**
 - **unmittelbare Ausführung** nach der Übersetzung (*Zeile für Zeile*)
 - für jeden Programmlauf erneut übersetzen
- **Compiler**
 1. Übersetzung des gesamten Programms vor der Ausführung und (meist) Speichern der Übersetzung in neuer Datei
 2. Ausführung des übersetzten Programms

Interpreter

- **Interpreter-Sprachen** sind z.B.
Python, Basic, Ruby, PHP, ...
- **Vorteil:** unmittelbare Ausführung führt zu direktem Feedback, wie im interaktiven Python-Editor
→ **Debugging** (Nachvollziehen des Programmverlaufs durch zeilenweise Ausführung, zur Fehlersuche)
- **Nachteile:**
 - Zusammenhänge von Programmteilen sind bei der Übersetzung oft noch nicht bekannt
 - Langsamere Programmausführung
 - Fehler führen oft zu spätem Programmabbruch

Syntaxfehler in Interpreter-Sprachen

- Gleichheitszeichen fehlt in Python-Code



```
Python 3.5 (32-bit)
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 5
>>> y = 2 * x
>>> print(y)
10
>>> z x + y
      File "<stdin>", line 1
        z x + y
          ^
SyntaxError: invalid syntax
>>>
```

- **Syntax:** Ist das Programm/sind alle Kommandos korrekt aufgebaut?
(Regeln sind durch die **Grammatik** der Sprache vorgegeben)

Syntax *versus* Semantik

■ Syntax

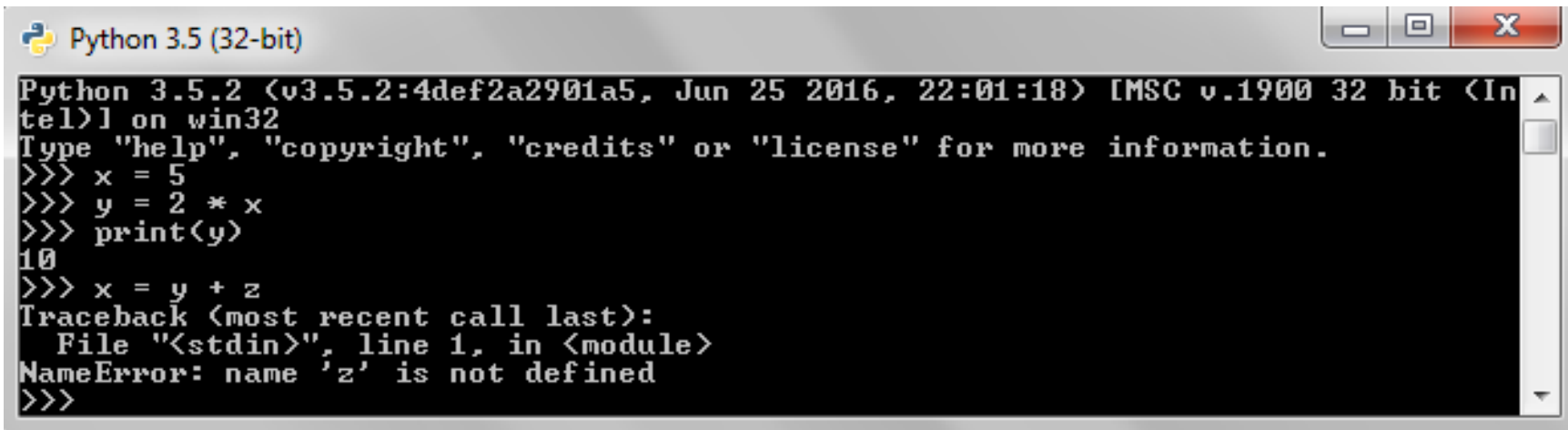
- Welche sprachlichen Elemente gibt es?
Wie sind diese aufgebaut?
- **Grammatik** für Python enthält z.B. eine Regel,
die die Syntax von Zuweisungen definiert, *etwa*
<Zuweisung> ::= <Variable> '=' <Ausdruck>
- Regeln (meist) in dieser **Backus-Naur-Form**

■ Semantik

- legt die **Bedeutung** der sprachlichen Elemente fest
- z.B. **<Zuweisung>** ermittelt den Wert von **<Ausdruck>**
und weist diesen **<Variable>** zu

Fehler in Interpreter-Sprache Python

- Zugriff auf undefinierte Variable (Programm nicht wohlgeformt)



```
Python 3.5 (32-bit)
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit <Intel>] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> x = 5
>>> y = 2 * x
>>> print(y)
10
>>> x = y + z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>>
```

- **Wohlgeformtheit:**

zusätzliche Regeln ergänzen die Syntaxdefinition

(z.B.: Man kann nur den Wert einer Variablen bestimmen, wenn ihr vorher mindestens einmal ein Wert zugewiesen wurde.)

Compiler

- **Compilersprachen** sind z.B. Pascal, **C**, C++, ... → **Praxis der Programmierung**
- zwei Dateien: Quellcode und Zielcode
- Übersetzung nur einmal erforderlich
- Vermeidung von Programmabbrüchen wegen einer Vielzahl von Fehlern durch **statische Prüfungen**
 1. **Syntaxanalyse** (Grammatikregeln eingehalten?)
 2. **Statische semantische Analyse** (Programm wohlgeformt?)
 - **Namen/Symboletabelle**
 - **Typprüfung** (bei Sprachen mit Typsystem)
 3. **Codeerzeugung** (Übersetzung)

Virtuelle Maschinen

- Einige Compiler übersetzen in Zwischencode (eine Art Assembler-Code).
- Dieser wird bei der Programmausführung interpretiert (durch eine *Virtuelle Maschine (VM)* weiter in Maschinencode übersetzt).
- *Beispiel:* **Java** → **Praxis der Programmierung**
 - Java-Compiler übersetzt in Byte-Code
 - Java VM interpretiert diesen Byte-Code
 - Vorteil: einheitliche Zielsprache des Compilers
 - nur JVM ist plattformabhängig