



**Compiler**

**Autor:**

**Aho et al.**

**Folie: 1**



© Pearson Studium 2008

# Compiler



Compiler

Kapitel 1

Einleitung

Folie: 2

# Compiler Kapitel 1

## Einleitung



Autor:  
Aho et al.

© Pearson Studium 2008



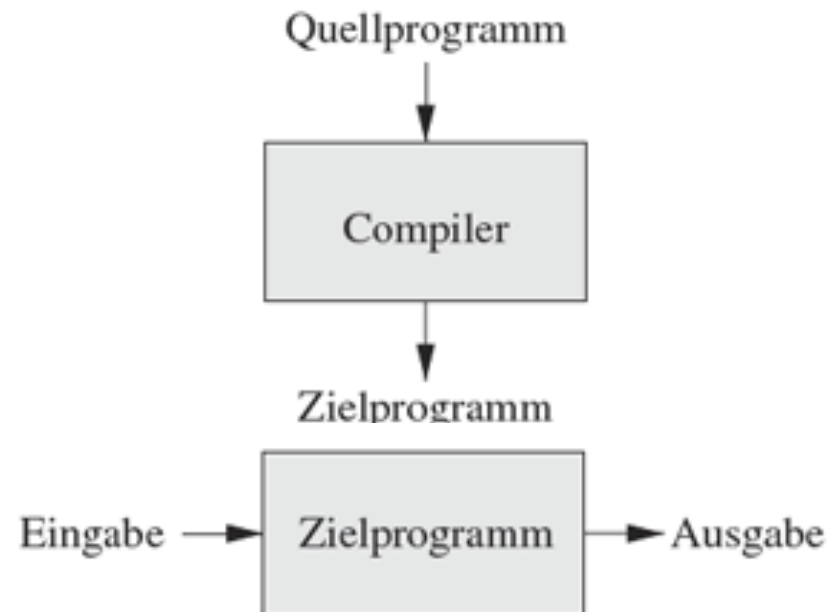
Compiler

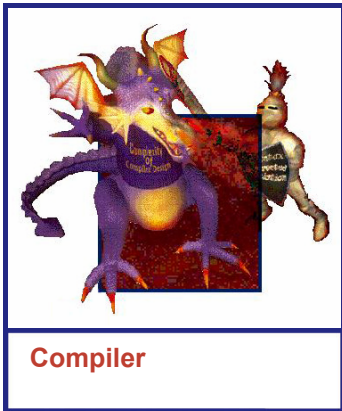
Autor:

Aho et al.

Folie: 3

# Ausführung eines Programms





Compiler

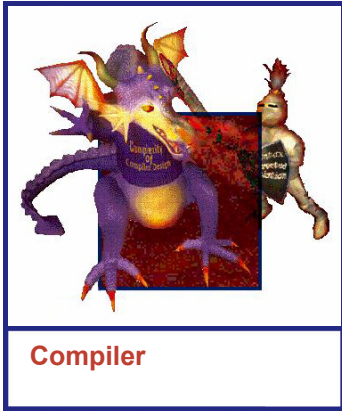
Autor:

Aho et al.

Folie: 4

# Vorteile eines Compilers (1.1.2)

- (viel) schnellere Ausführung
- Gesamter Code wird überprüft
  
- Quellcode/Interpreter muss nicht ausgeliefert werden



Compiler

Autor:

Aho et al.

Folie: 5

# Vorteile eines Interpreters (1.1.2)

- Keine Kompilationszeit
- Bessere Fehlerdiagnose (Debugging)
- Dynamische Sprachkonstrukte:
  - Introspektion/Reflection
  - Modifikation des Programms zur Laufzeit möglich



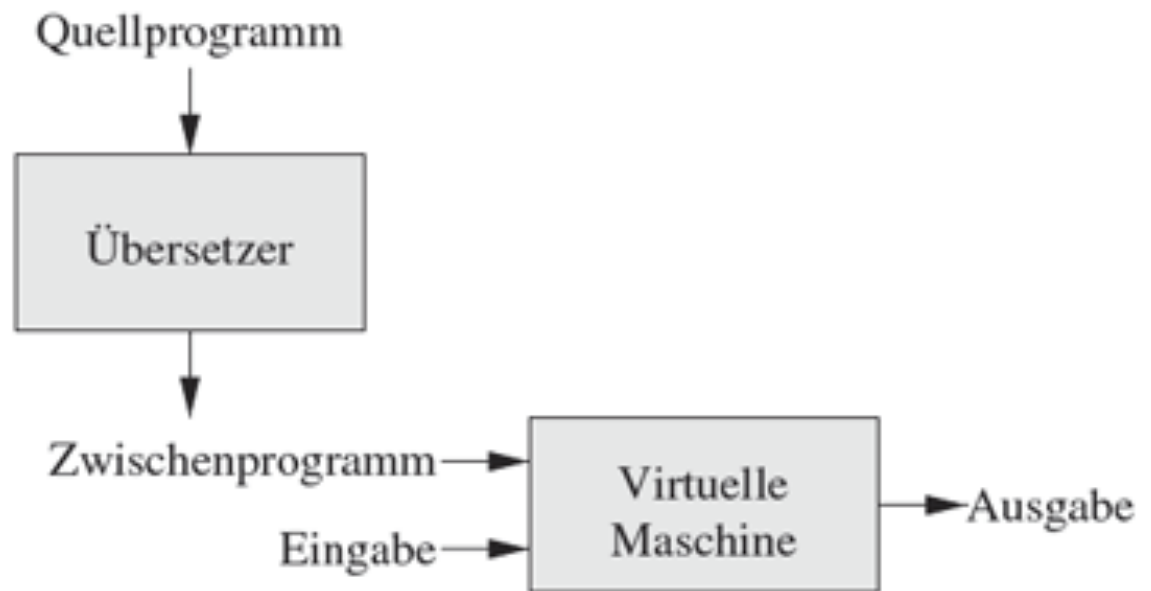
Compiler

Kapitel 1

Einleitung

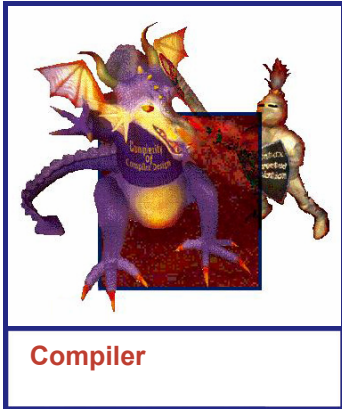
Folie: 6

# Hybridcompiler



manchmal:  
JIT Compiler

Abbildung 1.4: Ein Hybridcompiler



Autor:

Aho et al.

Folie: 7

# Vorteile eines Hybrid-Compilers

- kompakter Zwischenkode
- gegenüber Compiler:
  - Plattformunabhängig, bessere Fehlerdiagnose, dynamische Konstrukte bedingt möglich
- gegenüber Interpreter
  - Gesamter Code überprüft, schneller



Compiler

Kapitel 1

Einleitung

Folie: 8

PEARSON  
Studium **it**  
informatik

Autor:  
Aho et al.

© Pearson Studium 2008

# Compiler im Kontext

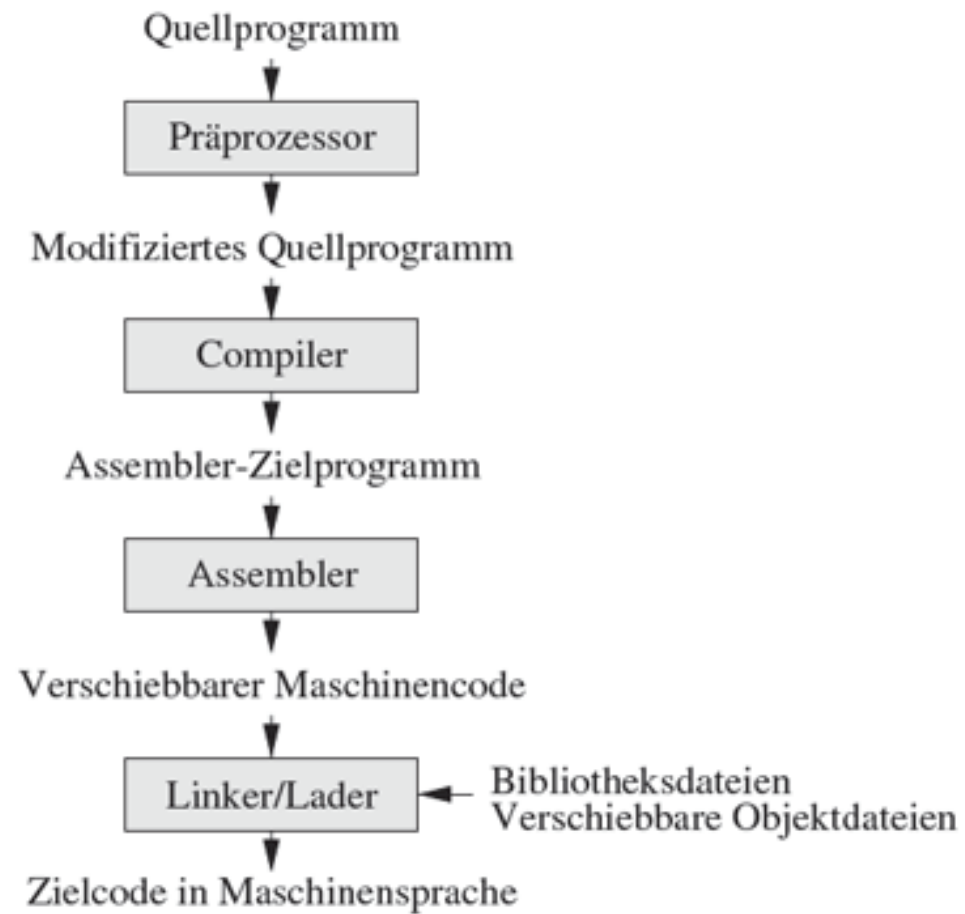
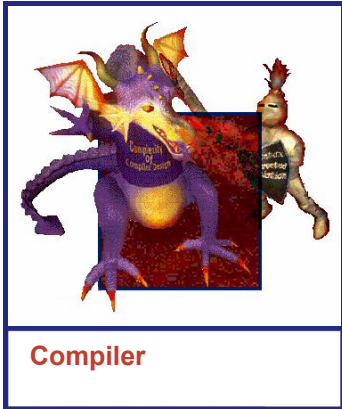


Abbildung 1.5: Ein Sprachverarbeitungssystem



Compiler

Kapitel 1

Einleitung

Folie: 9

# Warum Compiler studieren ?

- Ein Standardwerkzeug eines Informatikers
- Domain specific languages (DSLs)
  - Vielleicht müssen Sie einen Compiler schreiben
- Studium eines mittelgroßen Softwaresystems
- Interessante Theorie und Algorithmen
  - Nützlich für viele andere Anwendungen



Compiler

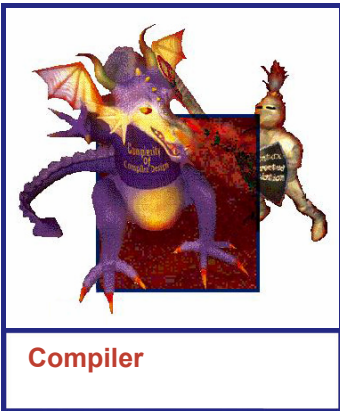
Kapitel 1

Einleitung

Folie: 10

# Simple1.java

```
public class simple1 {  
  
    public static void main(String args[])  
    {  
  
        int x = 3;  
        int y = 5;  
        int res = x + x*y + 2;  
        System.out.print("Result: ");  
        System.out.println(res);  
    }  
}
```



Compiler

Kapitel 1

Einleitung

Folie: 11

# Tokenized Simple1.java

LINE 2	SEMICOLON	IDENTIFIER: out
PUBLIC_SYMBOL	LINE 7	DOT
CLASS_SYMBOL	INT_SYMBOL	IDENTIFIER: print
IDENTIFIER: Simple1	IDENTIFIER: y	LEFT_PARENTHESIS
LEFT_BRACE	EQUALS	STRINGCONST: "Result:"
LINE 4	INTCONST: 5	"
PUBLIC_SYMBOL	SEMICOLON	RIGHT_PARENTHESIS
STATIC_SYMBOL	LINE 8	SEMICOLON
VOID_SYMBOL	INT_SYMBOL	LINE 10
IDENTIFIER: main	IDENTIFIER: res	IDENTIFIER: System
LEFT_PARENTHESIS	EQUALS	DOT
IDENTIFIER: String	IDENTIFIER: x	IDENTIFIER: out
IDENTIFIER: args	PLUS	DOT
LEFT_BRACKET	IDENTIFIER: x	IDENTIFIER: println
RIGHT_BRACKET	STAR	LEFT_PARENTHESIS
RIGHT_PARENTHESIS	IDENTIFIER: y	IDENTIFIER: res
LEFT_BRACE	PLUS	RIGHT_PARENTHESIS
LINE 6	INTCONST: 2	SEMICOLON
INT_SYMBOL	SEMICOLON	LINE 11
IDENTIFIER: x	LINE 9	RIGHT_BRACE
EQUALS	IDENTIFIER: System	LINE 12
INTCONST: 3	DOT	RIGHT_BRACE



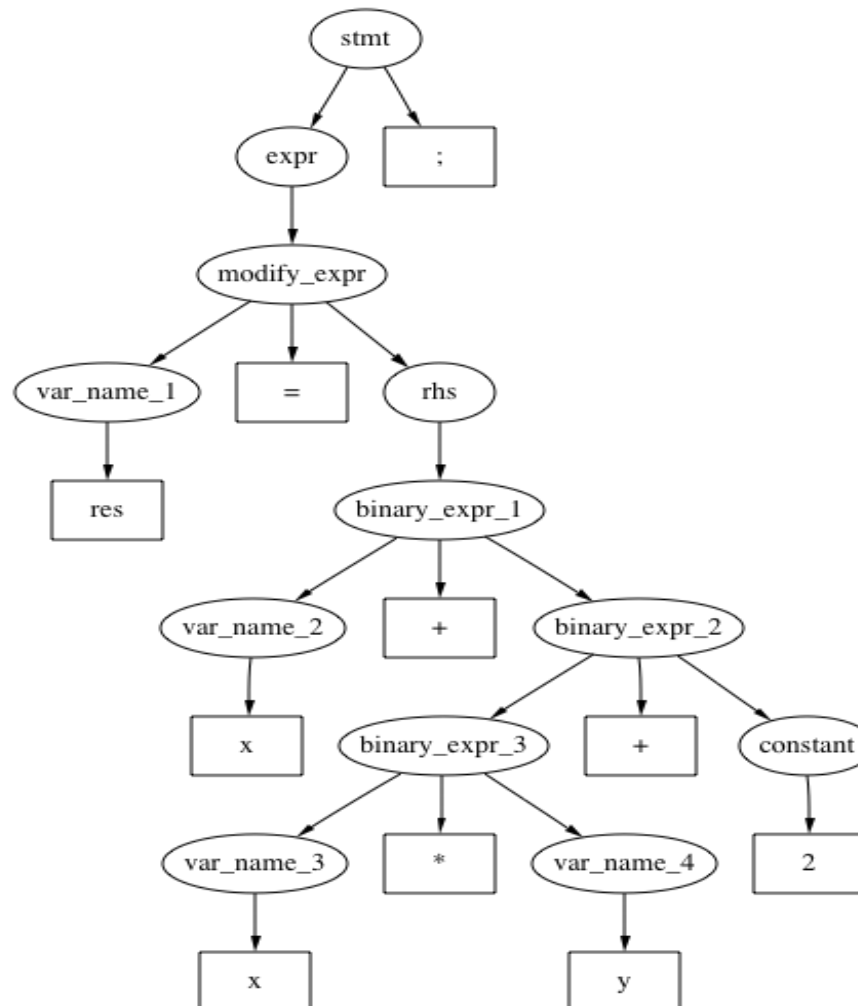
Compiler

Kapitel 1

Einleitung

Folie: 12

# Syntax tree for Simple1.java (part)



Note: “+” is treated  
as right associative  
(normally it is left  
associative!)



Compiler

Kapitel 1

Einleitung

Folie: 13

PEARSON  
Studium **it**  
informatik

Autor:  
Aho et al.

© Pearson Studium 2008

# Bytecode

```
public class Simple1 extends java.lang.Object{  
public Simple1();
```

Code:

```
0:   aload_0  
1:   invokespecial   #1; //Method java/lang/Object."<init>":()V  
4:   return
```

```
public static void main(java.lang.String[]);
```

Code:

```
0:   iconst_3  
1:   istore_1  
2:   iconst_5  
3:   istore_2  
4:   iload_1  
5:   iload_1  
6:   iload_2  
7:   imul  
8:   iadd  
9:   iconst_2  
10:  iadd  
11:  istore_3  
12:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;  
15:  ldc           #3; //String Result:  
17:  invokevirtual  #4; //Method java/io/PrintStream.print:(Ljava/lang/  
String;)V  
20:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;  
23:  iload_3  
24:  invokevirtual  #5; //Method java/io/PrintStream.println:(I)V  
27:  return
```

```
}
```



Compiler

Kapitel 1

Einleitung

Folie: 14



Autor:  
Aho et al.

© Pearson Studium 2008

Front End  
(Analyse)

Symboltabelle

Back End  
(Synthese)

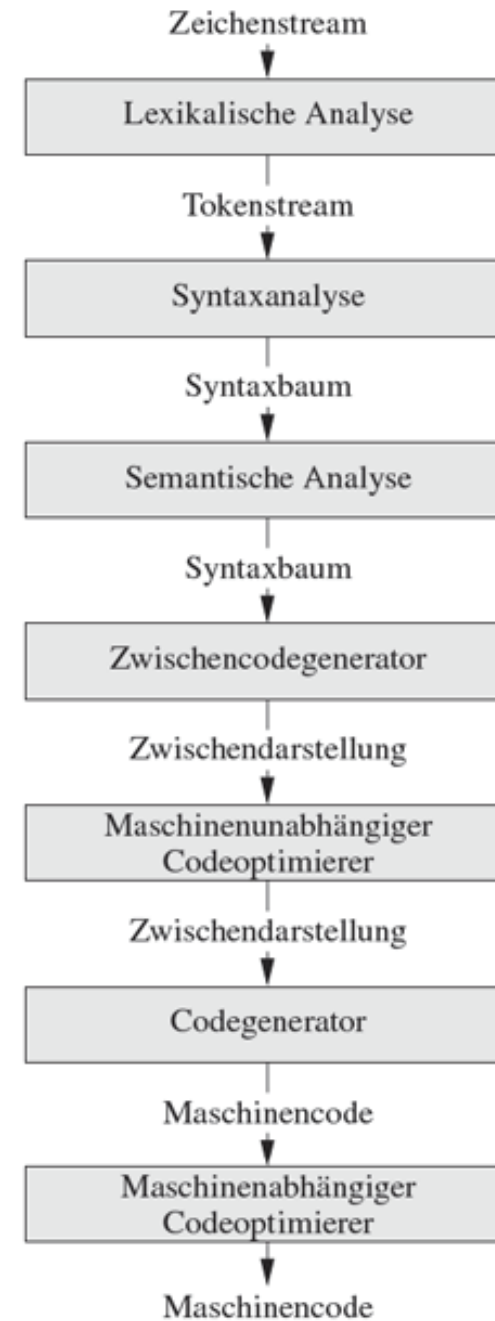


Abbildung 1.6: Phasen eines Compilers



Compiler

Kapitel 1

Einleitung

Folie: 15



Autor:  
Aho et al.

© Pearson Studium 2008

# Beispiel

- Übersetzung von:

$$\text{position} = \text{initial} + \text{rate} * 60$$



## Compiler

### Kapitel 1

### Einleitung

### Folie: 16

1	position	...
2	initial	...
3	rate	...

SYMBOLTABELLE

position = initial + rate \* 60

Lexikalischer Analysator

$\langle id, 1 \rangle (=) \langle id, 2 \rangle (+) \langle id, 3 \rangle (*) \langle 60 \rangle$

Syntaktischer Analysator

```

      =
     / \
  <id,1> +
         / \
        <id,2> *
              / \
             <id,3> 60
  
```

Semantischer Analysator

```

      =
     / \
  <id,1> +
         / \
        <id,2> *
              / \
             <id,3> inttfloat
                   |
                   60
  
```

Zwischengeneratore

```

t1 = inttfloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
  
```

Codeoptimierer

```

t1 = id3 * 60.0
id1 = id2 + t1
  
```

Codegenerator

```

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
  
```

Abbildung 1.7: Übersetzung  
einer Zuweisungsanweisung



Compiler

Kapitel 1

Einleitung

Folie: 17



Autor:  
Aho et al.

© Pearson Studium 2008

# Programmiersprachen: Einige Konzepte (1.6)

- **Statisch**: zur Kompilierzeit
- **Dynamisch**: zur Laufzeit
- **Gültigkeitsbereich** (scope) eines Namens
  - **Statisch** (lexikalisch) oder dynamisch
- **Speicherort** eines Namens
  - Statisch oder dynamisch bestimmbar
- **Wert** eines Speicherorts
  - Statisch oder **dynamisch**



Compiler

Kapitel 1

Einleitung

Folie: 18

PEARSON  
Studium **it**  
informatik

Autor:  
Aho et al.

© Pearson Studium 2008

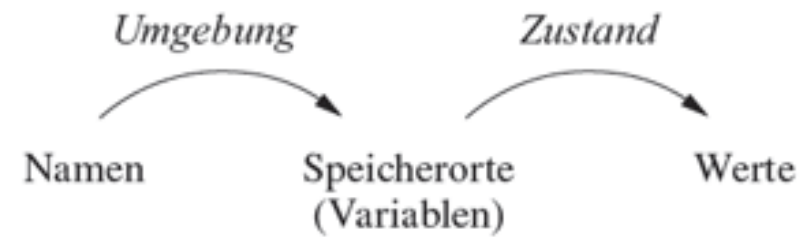


Abbildung 1.8: Abbildung von Namen auf Werte in zwei Stufen

```
...  
int i;          /* globales i */  
...  
void f(...) {  
    int i;      /* lokales i */  
    ...  
    i = 3;      /* Verwendung des lokalen i */  
    ...  
}  
...  
x = i + 1;     /* Verwendung des globalen i */
```

Abbildung 1.9: Zwei Deklarationen des Namens *i*



Compiler

Kapitel 1

Einleitung

Folie: 19

# Programmiersprachen: Einige Konzepte II

- **Bezeichner**
  - Zeichenstring der auf eine Entität verweist (zB Datenobjekt, Prozedur, Klasse oder Type)
  - Bezeichner sind Namen (Namen sind nicht immer Bezeichner, zB x.y)
- **Variable** eines Speicherorts
  - Verweist auf einen bestimmten Speicherort



Compiler

Kapitel 1

Einleitung

Folie: 20

# Programmiersprachen: Einige Konzepte III

- **Deklaration**
  - Geben Typ an
  - `int i`
- **Definition**
  - Geben Wert an
  - `i = 7`



Compiler

Kapitel 1

Einleitung

Folie: 21

# Gültigkeitsbereich

- Der **Gültigkeitsbereich (scope)** einer **Deklaration** von  $x$  ist der Kontext, in dem Verwendungen von  $x$  auf diese Deklaration verweisen. Eine Sprache verwendet einen **statischen** oder **lexikalischen Gültigkeitsbereich**, wenn der Gültigkeitsbereich einer Deklaration aus dem Programmtext abzulesen ist. Anderenfalls nutzt die Sprache einen **dynamischen Gültigkeitsbereich**
- Lebensdauer:
  - muss mindestens den Gültigkeitsbereich abdecken
  - kann aber länger sein (zB static Variablen)



Compiler

Kapitel 1

Einleitung

Folie: 22

# Statischer Gültigkeitsbereich Blockstruktur

```
main() {  
  int a = 1;  $B_1$   
  int b = 1;  
  {  
    int b = 2;  $B_2$   
    {  
      int a = 3;  $B_3$   
      cout << a << b;  
    }  
    {  
      int b = 4;  $B_4$   
      cout << a << b;  
    }  
    cout << a << b;  
  }  
  cout << a << b;  
}
```

Abbildung 1.10: Blöcke in einem C++-Programm

Deklaration	Gültigkeitsbereich
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	$B_3$
int b = 4;	$B_4$



Compiler

Kapitel 1

Einleitung

Folie: 23

# Dynamischer Gültigkeitsbereich

- **Objekt-Orientierte Programmierung: Vererbung**
  - Klasse C mit methode m()
  - Unterklasse D von C mit eigenem m()
  - Objekt x der Klasse C, Aufruf x.m()
- **Dynamische Programmiersprachen**
  - Tcl, Python, Ruby



Compiler

Kapitel 1

Einleitung

Folie: 24

w=13  
x=11  
y=13  
z=11

# Übung

```
int w, x, y, z;  
int i = 4; int j = 5;  
{  
  int j = 7;  
  i = 6;  
  w = i + j;  
}  
x = i + j;  
{  
  int i = 8;  
  y = i + j;  
}  
z = i + j;
```

**a** Code für Übung 1.6.1

```
int w, x, y, z;  
int i = 3; int j = 4;  
{  
  int i = 5;  
  w = i + j;  
}  
x = i + j;  
{  
  int j = 6;  
  i = 7;  
  y = i + j;  
}  
z = i + j;
```

**b** Code für Übung 1.6.2

w=9  
x=7  
y=13  
z=11

Abbildung 1.13: Code in Blockstruktur

Werte für w,x,y,z ?



Compiler

Kapitel 1

Einleitung

Folie: 25

PEARSON  
Studium **it**  
informatik

Autor:  
Aho et al.

© Pearson Studium 2008

# Übung

int w: B1 - B3 - B4

x: B1-B2-B4

y: B1-B5

z: B1-B2-B5

int x: B2-B3

z: B2

int w,x: B3

int w,x: B4

int y,z: B5

```
{ int w, x, y, z;      /* Block B1 */  
  { int x, z;         /* Block B2 */  
    { int w, x;       /* Block B3 */  }  
  }  
  { int w, x;         /* Block B4 */  
    { int y, z;       /* Block B5 */  }  
  }  
}
```

Abbildung 1.14: Code in Blockstruktur für Übung 1.6.3

Gültigkeitsbereiche der zwölf Deklarationen?



Compiler

Kapitel 1

Einleitung

Folie: 26



Autor:  
Aho et al.

© Pearson Studium 2008

# Andere Themen in 1.6

- Zugriffskontrolle
  - public
  - private
  - protected
- Parameterübergabe
  - call by value
  - call by reference
  - call by name
  - Aliasing
  - Selber durchlesen



Compiler

Kapitel 1

Einleitung

Folie: 27

# Zusammenfassung

- Compiler, Interpreter, Hybridcompiler
- Compilerphasen
  - Back-End, Front-End
- Einige Programmiersprachenkonzepte
  - Statisch / Dynamisch
  - Gültigkeitsbereich (scope)
  - Umgebung