Compiler

编译器建设

Compilerbau

Michael Leuschel

John Witulski

Bilder

Compiler

Lehrangebot

Softwaretechnik		Programmiersprachen
Compilerbau 5 LP	Von Nand zu Tetris 5 LP	Einführung in die Logische Programmierung 5 LP
Sicherheitskritische Systeme 5 LP		Dynamische Programmier-sprachen 5 LP
Model Checking 5 LP	Funktionale Programmierung 5 LP	Vertiefung Logische Programmierung 5 LP

rheinjug
JAVA USER GROUP DÜSSELDORF

Bilder

Organisation - Übungen

Klausurzulassung:

- Übungen und Blätter freiwillig
- Bearbeitung praktischer Übungen freiwillig
- Bearbeitung Compilerprojekt **Pflicht**

Übungen:

- Ab 29.10.2021 in 25.12.02.55* um 14:30
- Material auf ilias.hhu.de

Bilder

Organisation - Benotung

- Klausur 60%-75%

  (Entscheidung im November)
- Compilerprojekt 25% - 40%
  (Entscheidung im November)
- Bachelor Informatik 5 LP

Bilder

Organisation - Sonstiges

- Kontakt Übungen:

John.Witulski@hhu.de

Termine Freitags nach Vereinbarung

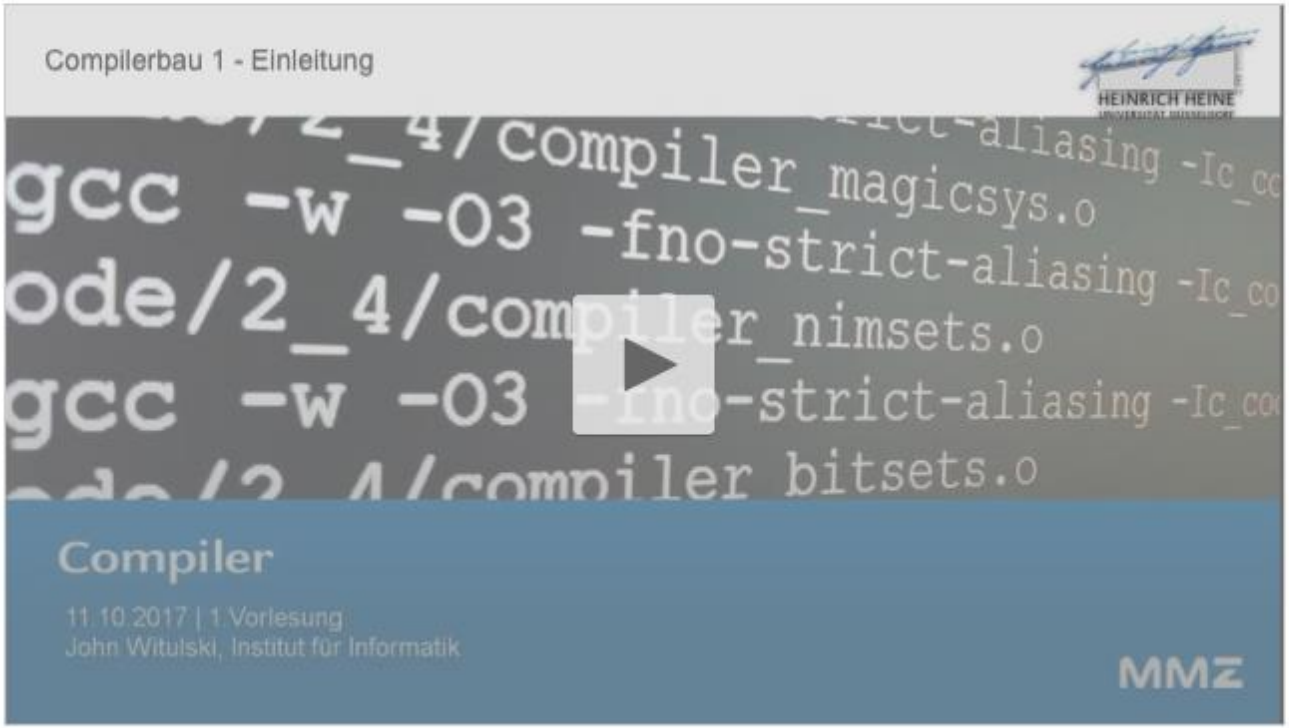
- Videoaufzeichnung

Mediathek.hhu.de

Bilder

Compilerbau 1 - Einleitung

John Witulski



Uploaded by Witulski at 10/25/2017 - recorded at 10/24/2017

Report

Information

Speaker:

John Witulski

Description:

Vorlesung Compilerbau im WS 2017/2028

Category:

Vorlesungen

Compilerbau

Compilerbau 2 - Lexing
John Witulski
10/18/2017
by Witulski

Compilerbau 3 - Parsing
John Witulski
10/26/2017
by Witulski

Compilerbau 4 - LL Parsi...
John Witulski
11/8/2017
by Witulski

Compilerbau 5 - LR Parsi...
John Witulski
11/15/2017
by Witulski

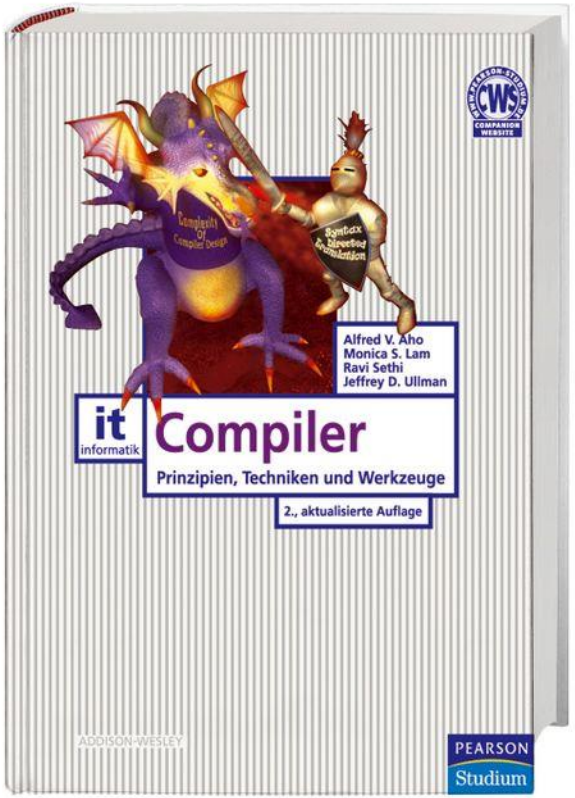
Compilerbau 6 - Semantik
John Witulski
11/22/2017
by Witulski

Compilerbau 8 - Typechec.
John Witulski
12/6/2017
by Witulski

Compiler

Bücher I

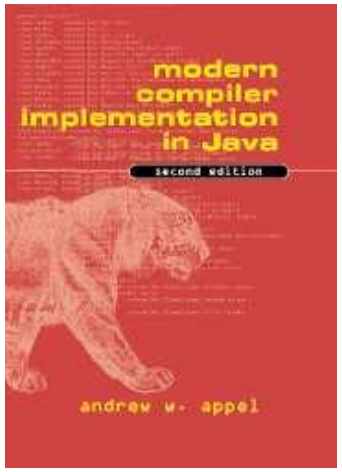
- Aho, Lam, Sethi, Ullman: Compiler, 2008, Pearson
- Errata:
  - https://www.cs.hhu.de/lehrstuehle-und-arbeitsgruppen/softwaretechnik-und-programmiersprachen/unser-team/team/leuschel/errata.html



Bilder
© Pearson Studium

Compiler

Bücher II



- Compilerbau*:*
  - Andrew W. Appel*, Modern Compiler Implementation in Java (2nd edition),* Cambridge University Press, 2002.
  - http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=052182060x

- Anderes Hintergrundmaterial:
  - Watt & Brown, *Programming Language Processors in Java,* Prentice-Hall, 2000.
  - Cooper, Torczon, *Engineering a Compiler*, Elsevier, 2004.

Compiler
Kapitel 1

Einleitung

Programmiersprachen
编程语言

- Maschinen- und Assemblersprachen
- 机器和汇编语言
- Höhere Programmiersprachen
- 高级编程语言
  - übernehmen Speicherverwaltung, Typkonsistenz, parallele Ausführung,...接管内存管理、类型一致性、并行执行
  - höhere Produktivität, weniger Fehler提高生产力，减少错误
  - gute Übersetzung wichtig um Hardware effektiv zu nutzen良好的翻译对有效使用硬件很重要

Bilder

Compiler

Maschinensprache

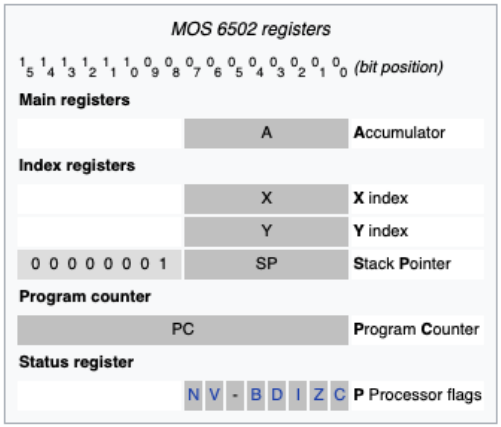
0014360	105510	137400	000003	000000	110377	000240	000000	044215
0014400	104402	130415	075410	101400	001700	002611	004260	000173
0014420	002611	004262	000173	002707	004260	000173	000000	000000
0014440	105511	044007	000213	002277	000000	177400	060220	000006
0014460	044000	002611	004240	000173	102510	072700	044423	003613
0014500	105510	030400	137377	177777	177777	110377	005640	000000
0014520	002613	004142	000173	106504	177140	144377	104510	134105
0014540	104504	146145	104514	140155	102515	072755	165424	110061
0014560	104504	042366	171211	110377	001440	000000	102515	072355
0014600	044437	042613	044400	042473	071410	044065	044215	044401

B1 80
F0 11
C9 41
90 06

Bilder

Compiler

Assembler




```
LOOP    LDA (SRC),Y ;get from source string
BEQ DONE    ;end of string ;
CMP #'A'    ;if lower than UC alphabet...
BCC SKIP    ;copy unchanged
…
SKIP    …
…
DONE    …
```

B1 80
F0 11
C9 41
90 06

https://commons.wikimedia.org/wiki/File:Apple_iieb.jpg
https://en.wikipedia.org/wiki/MOS_Technology_6502
CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=91538

http://www.6502.org/tutorials/6502opcodes.html#CMP

Bilder
© Pearson Studium

Compiler

Höhere Programmiersprachen

```
LOOP    LDA (SRC),Y ;get from source string
BEQ DONE    ;end of string ;
CMP #'A'    ;if lower than UC alphabet...
BCC SKIP    ;copy unchanged
…
SKIP    …
…
DONE    …
```

Was bekommt man im Vergleich zu Assembler?
Was verliert man?

Bilder

Compiler

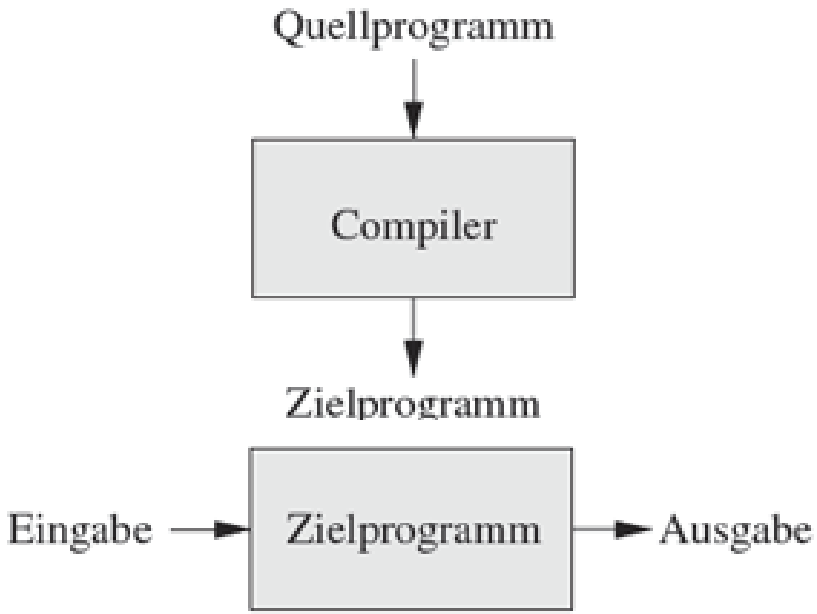
Höhere Programmiersprachen

- Ausdrücke (2*x+1)
- Datentypen
- Kontrollstrukturen (while,…)
- Deklarationen
- Abstraktion (Methoden mit Namen und Parametern,…)
- Kapselung, Sichtbarkeitsregeln

Bilder

Compiler

Ausführung eines Programms方案的执行



Bilder

Compiler

Vorteile eines Compilers
编译器的优点

- (viel) schnellere Ausführung
- 执行速度快得多
- Gesamter Code wird überprüft
  整个代码被检查

- Quellcode/Interpreter muss nicht ausgeliefert werden
  不需要交付源代码/解释器

Bilder

Vorteile eines Interpreters
翻译的优势

- Keine Kompilationszeit无编译时间
- Bessere Fehlerdiagnose更好的错误诊断 (Debugging)
- Dynamische Sprachkonstrukte:动态语言结构。
  - Introspektion/Reflection自省/反思
  - Modifikation des Programms zur Laufzeit möglich在运行时可以对程序进行修改

Bilder

Hybridcompiler混合编译器

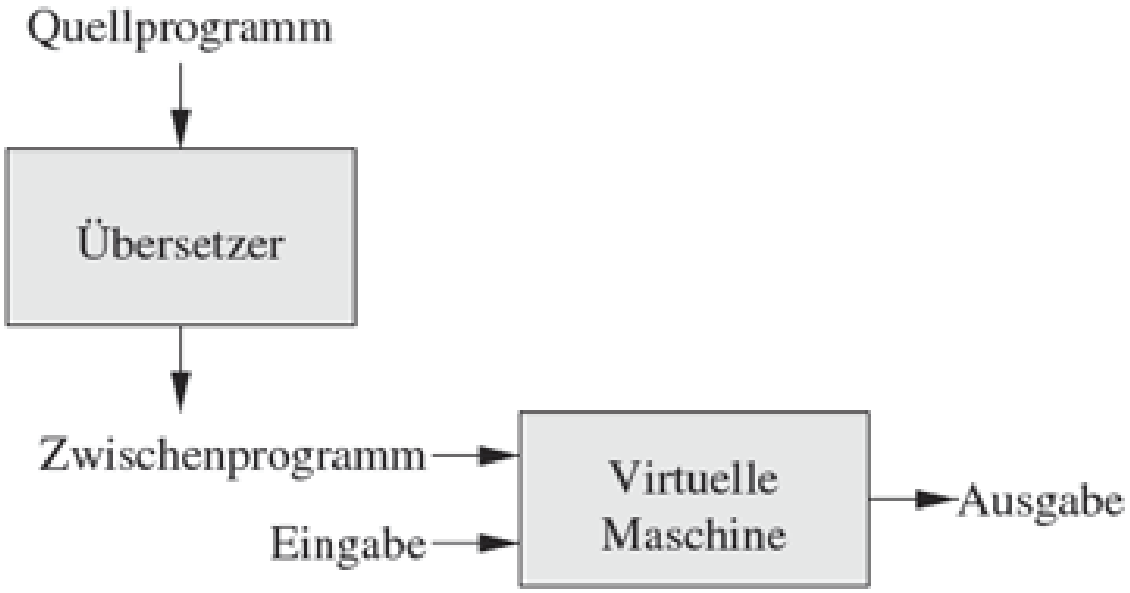


Abbildung 1.4: Ein Hybridcompiler

manchmal:
JIT Compiler

Vorteile eines Hybrid-Compilers混合编译器的优势

Compiler

- kompakter Zwischenkode紧凑的中间代码
- gegenüber Compiler:与编译器相比
  - Platformunabhängig, bessere Fehlerdiagnose, dynamische Konstrukte bedingt möglich独立于平台，更好的错误诊断，有条件的动态构造
- gegenüber Interpreter
  - Gesamter Code überprüft, schneller与解释者相比
  - - 检查整个代码，速度更快

Bilder

Compiler im Kontext
背景下的编译器

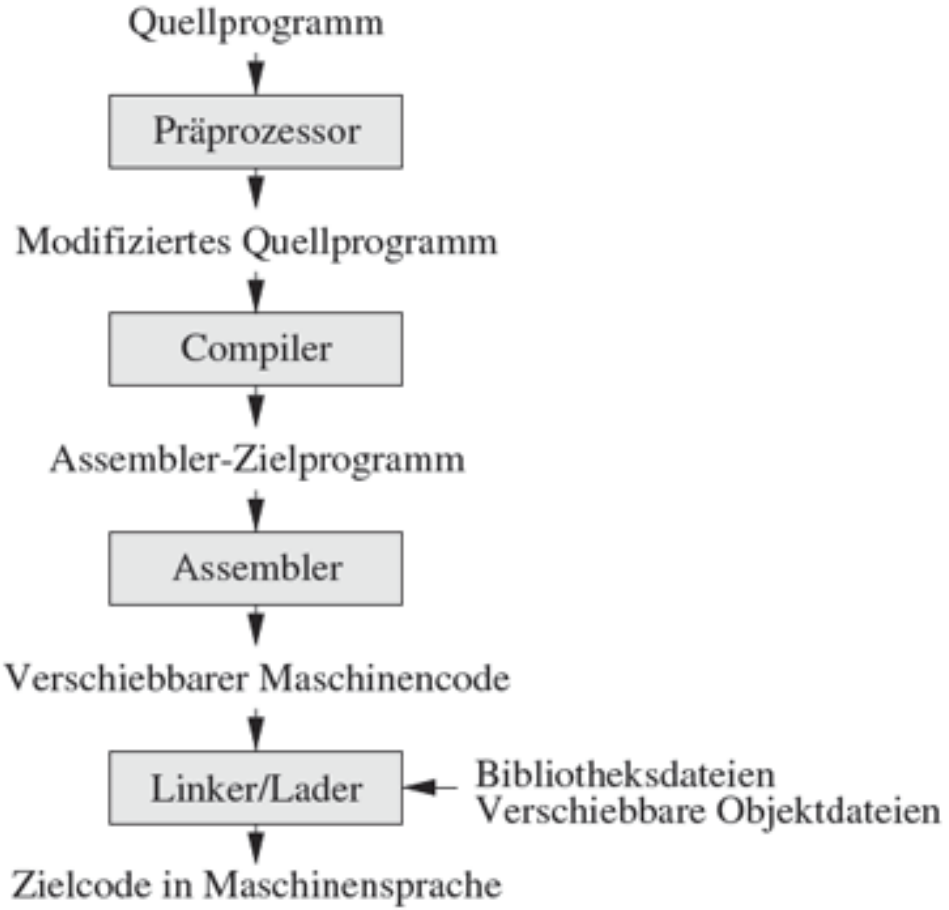


Abbildung 1.5: Ein Sprachverarbeitungssystem

Einige bekannte Compiler一些著名的编译器

https://**gcc**.gnu.org/

http://clang.llvm.org http://llvm.org

javac
Java Hotspot

Microsoft Visual C++

https://code.google.com/p/v8/

Warum Compiler studieren ?
为什么要学习编译器?

- Ein Standardwerkzeug eines Informatikers 计算机科学家的一个标准工具
- Domain specific languages (DSLs)特定领域语言
  - Vielleicht müssen Sie einen Compiler schreiben你可能需要编写一个编译器
- Studium eines mittelgroßen Softwaresystems对一个中等规模的软件系统的研究
- Interessante Theorie und Algorithmen
  - Nützlich für viele andere Anwendungen有趣的理论和算法
  - - 适用于许多其他应用

Interessante Problemen und Lösungen有趣的问题和解决方案

Compiler

- Graphalgorithmen (Dead-Code Elimination)图形算法（消除死代码）
- Suchalgorithmen, Greedy Search搜索算法，贪婪的搜索(Registerallokation,…)
- Dynamische Programmierung动态编程 (Instruction Selection)
- Automaten (Lexing)自动机
- Grammatiken (Parsing)语法(解析)
- Fixpunktalgorithmen (Datenflussanalyse)固定点算法（数据流分析

Bilder

Compiler

Einleitung

Should I learn compilers?

stackoverflow

27

If you just want to be a run-of-the-mill coder, and write stuff... you don't need to take compilers.

If you want to learn computer science and appreciate and really become a computer scientist, you *MUST* take compilers.

Compilers is a microcosm of computer science! It contains every single problem, including (but not limited to) AI (greedy algorithms & heuristic search), algorithms, theory (formal languages, automata), systems, architecture, etc.

You get to see a lot of computer science come together in an amazing way. Not only will you understand more about why programming languages work the way that they do, but you will become a better coder for having that understanding. You will learn to understand the low level, which helps at the high level.

As programmers, we very often like to talk about things being a "black box"... but things are a lot smoother when you understand a little bit about what's in the box. Even if you don't build a whole compiler, you will surely learn a lot. You will get to see the formalisms behind parsing (and realize it's not just a bunch of special cases hacked together), and a bunch of NP complete problems. You will see why the theory of computer science is so important to understand for practical things. (After all, compilers are extremely practical... and we wouldn't have the compilers we have today without formalisms).

I really hope you consider learning about them... it will help you get to the next level as a computer scientist :-).

share improve this answer

answered Apr 9 '09 at 7:31

Tom

8,455 • 3 • 26 • 39

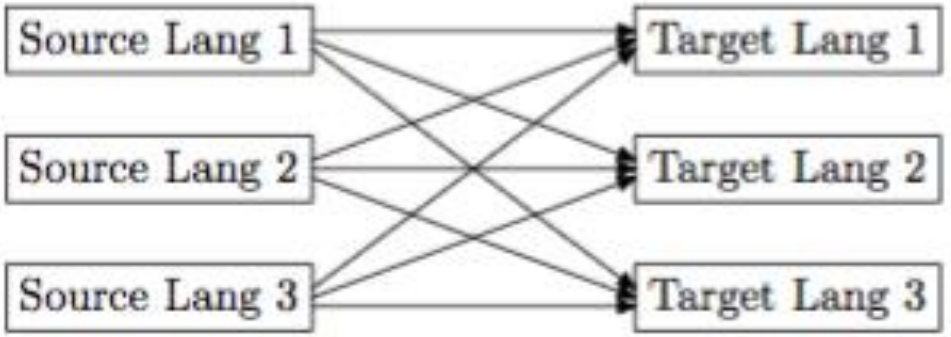
http://stackoverflow.com/questions/733093/when-should-i-learn-compilers

Compilerbau:编译器建设。
一项复杂的工作
Ein komplexes Unterfangen


```
position = initial + rate*60
```

?

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



Compiler

Beispiel

```
#define CUBE(x) (x)*(x)*(x)
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += CUBE(x);
  }
  printf("The sum is %d\n", sum);
}
```

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Compiled_Programs

Bilder

Compiler

```
#define CUBE(x) (x)*(x)*(x)
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += CUBE(x);
  }
  printf("The sum is %d\n", sum);
}
```

```
# Assembly of x86
  .cstring
LC0:
  .ascii "The sum is %d\12\0"
  .text
.globl _main
_main:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $40, %esp
  movl    $0, -20(%ebp)
  movl    $2, -16(%ebp)
  movl    $0, -12(%ebp)
  jmp     L2
L3:
  movl    -
  imull   -
  imull   -
  addl    %
L2:
  cmpl    $
  setle   %
  addl    $
  testb   %
  jne     L
  movl    -
  movl    %
  movl    $
  call    _
  leave
  ret
```

```
# Assembly of ARM
_main:
@ BB#0:
  push      {r7, lr}
  mov       r7, sp
  sub       sp, sp, #16
  mov       r1, #2
  mov  r0, #0
  str      r0, [r7, #-4]
  str  r0, [sp, #8]
  stm       sp, {r0, r1}
  b LBB0_2
LBB0_1:
  ldr       r0, [sp, #4]
            r3, [sp]
            r0, r0
            r1, r0, r3
            [sp]
            r0, [sp, #8]
            r1, r0, #1
            #99
            r1, [sp, #8]
            BB0_1
            LCPI0_0
            [sp]
            r0, pc, r0
            _printf
            r0, [r7, #-4]
            sp, r7
            {r7, lr}
  mov       pc, lr
```

```
  .cstring
LC0:
  .ascii "The sum is %d\12\0"
  .text
.globl _main
_main:
  pushl %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  movl  $800, 4(%esp)
  movl  $LC0, (%esp)
  call  _printf
  leave
  ret
```

Bilder

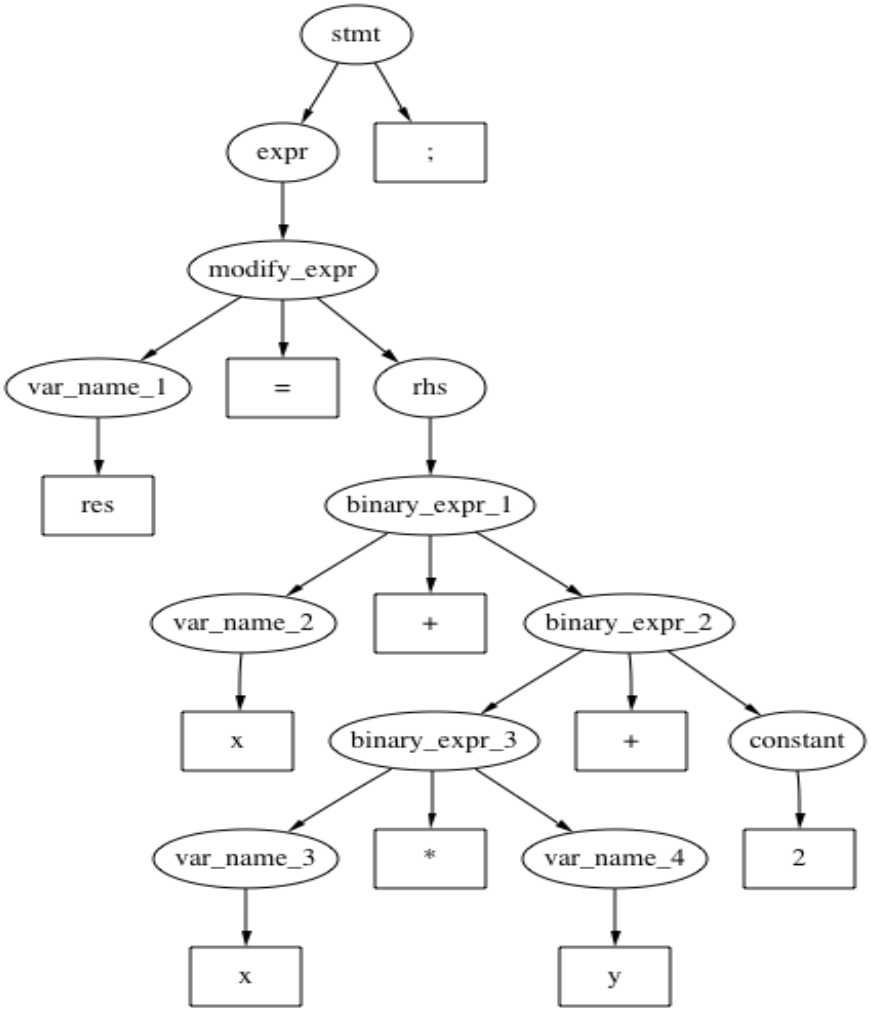
Simple1.java

```
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);
    }
}
```

Tokenized Simple1.java符号化

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

Syntax tree for Simple1.java (part)语法树



Hinweis: “+” ist hier rechts-assoziativ (normalerweise ist es links-assoziativ!)

javap -c Simple1

Bytecode

Compiler

Einleitung

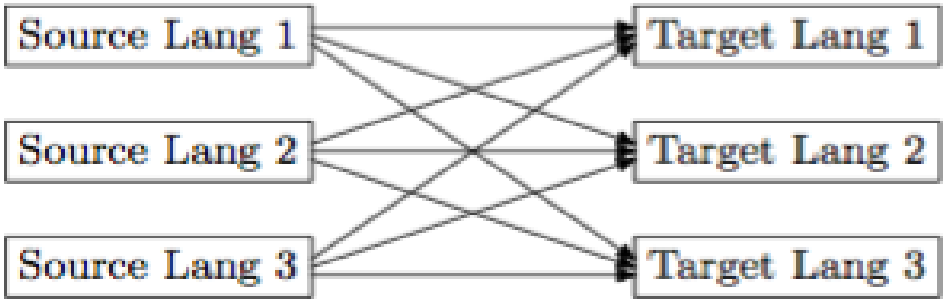
```
public class Simple1 extends java.lang.Object{
public Simple1();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

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

}
```

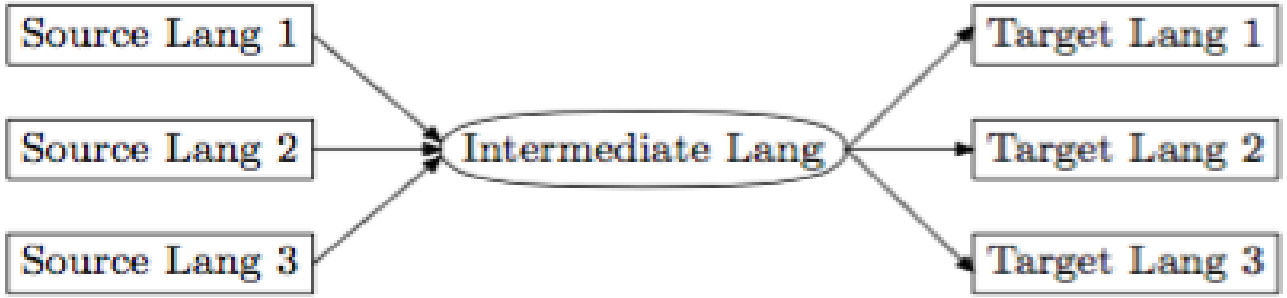
Compiler Architektur编译器结构

```
position = initial + rate*60
```

?

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





Front End
(Analyse)

Back End
(Synthese)

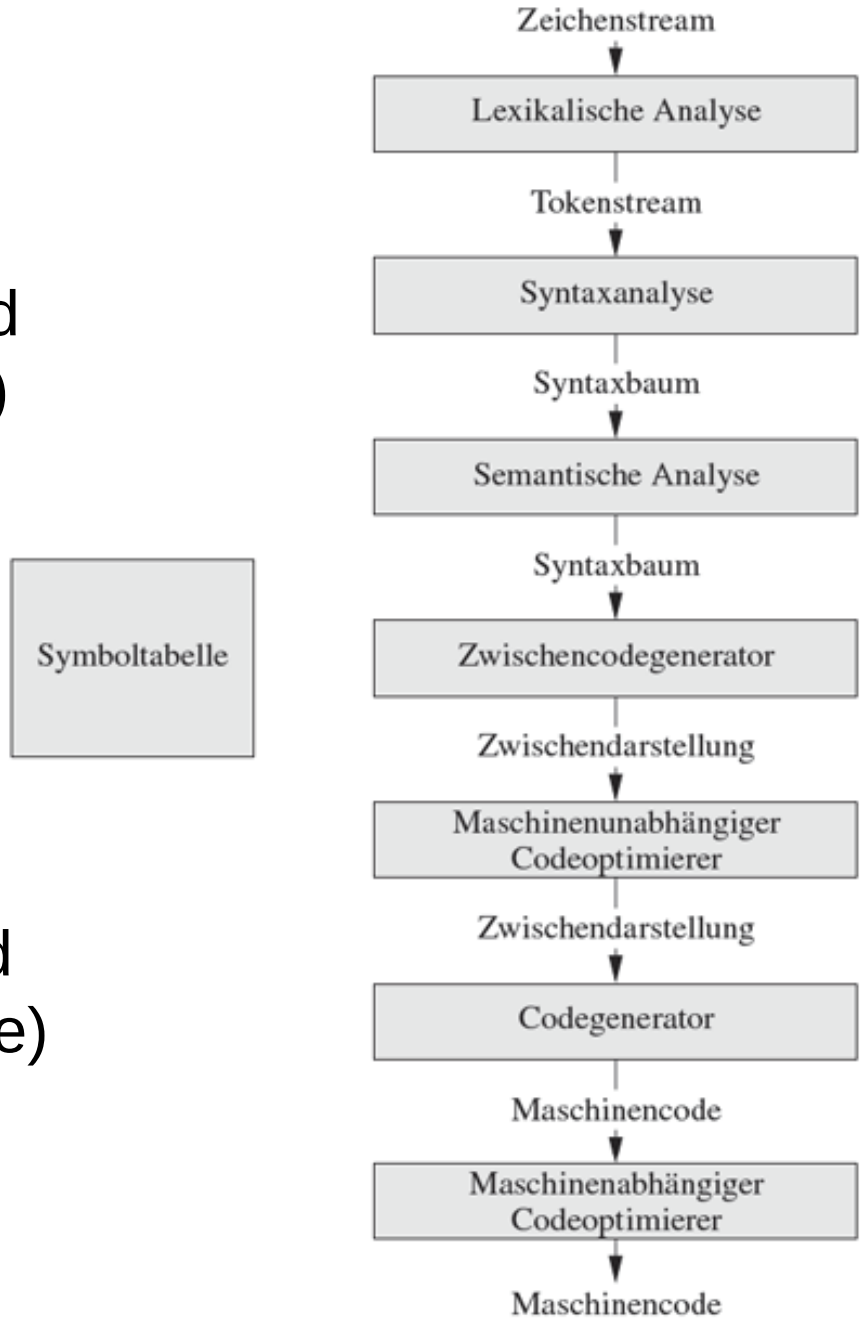


Abbildung 1.6: Phasen eines Compilers

Beispiel

- Übersetzung von:

position = initial + rate*60

Compiler

Einleitung

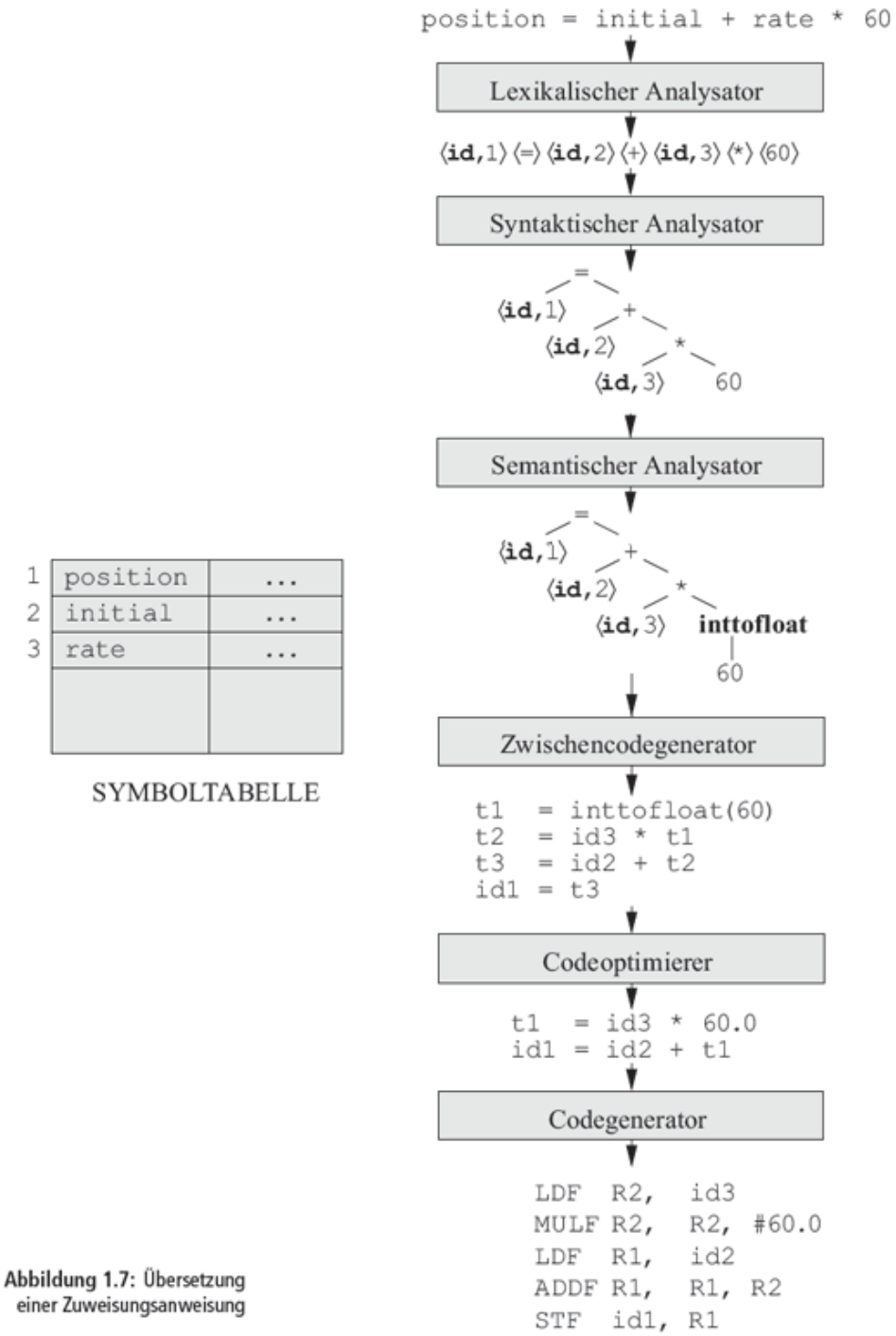


Abbildung 1.7: Übersetzung einer Zuweisungsanweisung

Programmiersprachen: Einige Konzepte (1.6)编程语言。一些概念

- Statisch: zur Kompilierzeit 静态
- Dynamisch: zur Laufzeit
- **Gültigkeitsbereich** (scope) eines Namens
  - **Statisch** (lexikalisch) oder dynamisch
- **Speicherort** eines Namens名称的存储位置
  - Statisch oder dynamisch bestimmbar
  - **Umgebung** verknüpft Namen mit Speicherort
- **Wert** eines Speicherorts
  - Statisch oder **dynamisch**

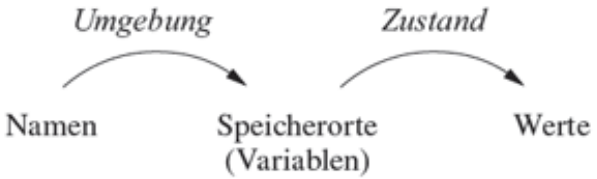


Abbildung 1.8: Abbildung von Namen auf Werte in zwei Stufen

Gültigkeitsbereich : Blockstruktur (1.6)范围：块状结构

- Programmiersprachen mit Blockstruktur:
  - erlauben Verschachtelung von Blöcken
    具有块状结构的编程语言。
  - - 允许区块嵌套

```
...
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*

Programmiersprachen: Einige Konzepte II

Compiler

Einleitung

- **Bezeichner**识别器
  - Zeichenstring der auf eine Entität verweist (zB Datenobjekt, Prozedur, Klasse oder Type)指向一个实体（如数据对象、程序、类或类型）的字符串。
  - Bezeichner sind Namen (Namen sind nicht immer Bezeichner, zB x.y)标识符是名称
- **Variable** eines Speicherortsv- 储存地点的变量
  - Verweist auf einen bestimmten Speicherort
  - 指的是一个特定的内存位置

Programmiersprachen: Einige Konzepte III

- **Deklaration** 声明
  - Geben Typ an指定类型
  - int i
- **Definition**定义
  - Geben Wert an指定值
  - i = 7

Gültigkeitsbereich有效性范围

- 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*

- *x的声明的范围是指x的使用指向这个声明的上下文。如果声明的范围可以从程序文本中读出，那么一种语言就会使用静态或词法范围。否则，该语言会使用一个动态的有效性范围*

- Lebensdauer:
  - muss mindestens den Gültigkeitsbereich abdecken必须至少涵盖范围
  - - 但可以更长（例如静态变量）。
  - kann aber länger sein (zB static Variablen)

Statischer Gültigkeitsbereich
Blockstruktur
静态范围 块状结构

```
main() {
    int a = 1;                          B1
    int b = 1;
    {
        int b = 2;                      B2
        {
            int a = 3;                  B3
            cout << a << b;
        }
        {
            int b = 4;                  B4
            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

Dynamischer Gültigkeitsbereich

- **Objekt-Orientierte Programmierung: Vererbung** 面向对象的编程：继承性
  - 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动态编程语言

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

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

```
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 ?

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?

Andere Themen in Drachenbuch 1.6
龙之书1.6的其他主题

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

Zusammenfassung

- Compiler, Interpreter, Hybridcompiler
- Compilerphasen
  - Compiler: Abfolge von Phasen
  - Trennung: Back-End, Front-End
- Theorie und Praxis: Modellierung im Compilerdesign (Automaten, Grammatiken, reguläre Ausdrücke, Bäume)
- Einige Programmiersprachenkonzepte
  - Statisch/dynamisch, “scope”, …

Compiler

Rest vom Kurs

- Lexing
- Parsing
- Semantische Analyse
- Zwischencode
- Codegenerierung

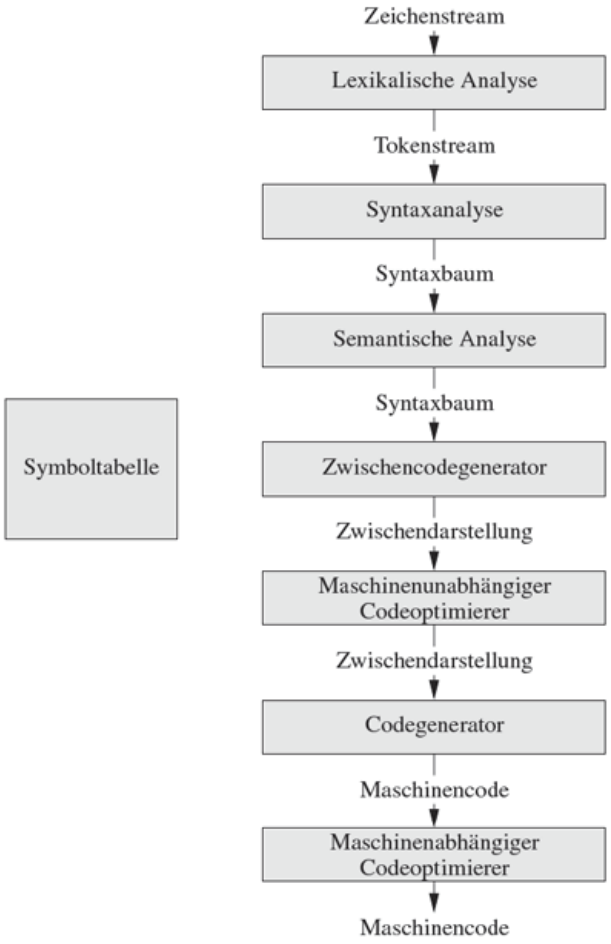


Abbildung 1.6: Phasen eines Compilers

Bilder
© Pearson Studium