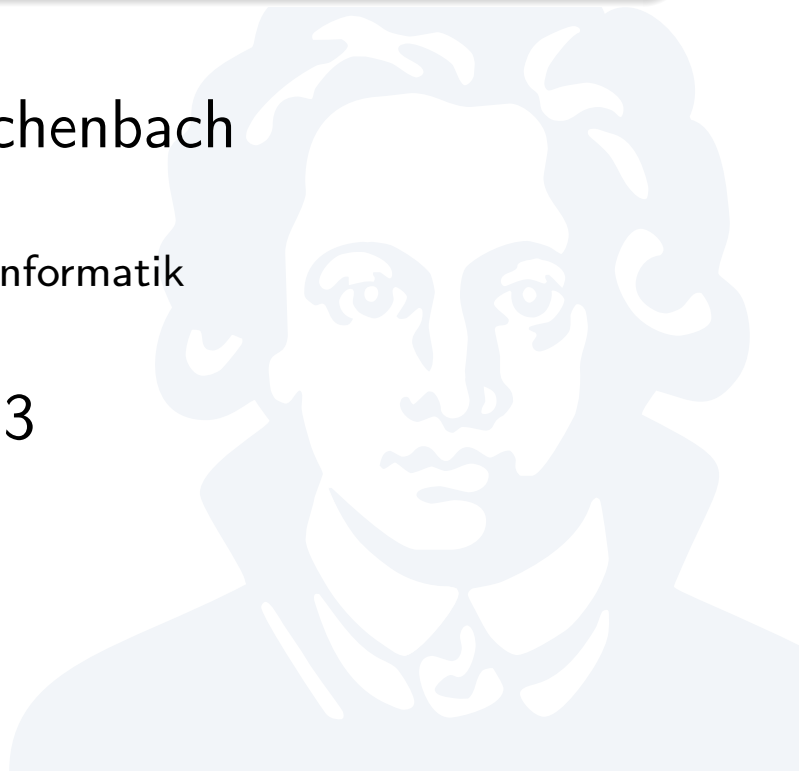
Current Topics in Software Technology
Aktuelle Themen der Softwaretechnologie

WS 2013/14

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

15. Oktober 2013



Aktuelle Themen der Softwaretechnologie

Was wir heute tun werden:

- Seminarübersicht
- Themengebiete und Themen
- Wichtiges Hintergrundmaterial
- Beispielvortrag

Teil I

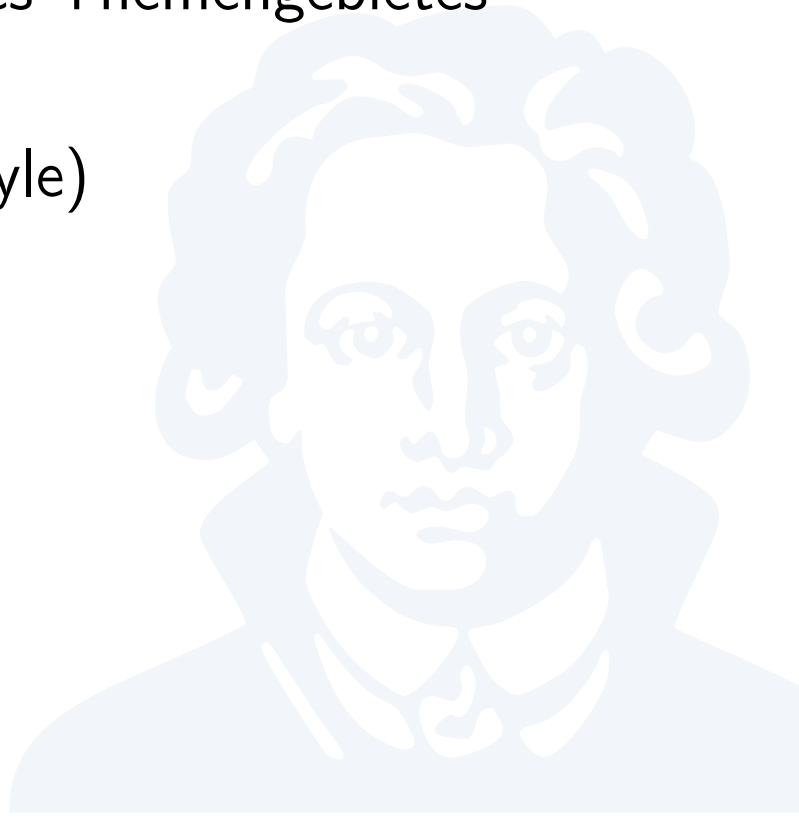
Seminarübersicht

Erwartungen

1. Ausarbeitung
2. Vortrag
3. Teilnahme mit Feedback

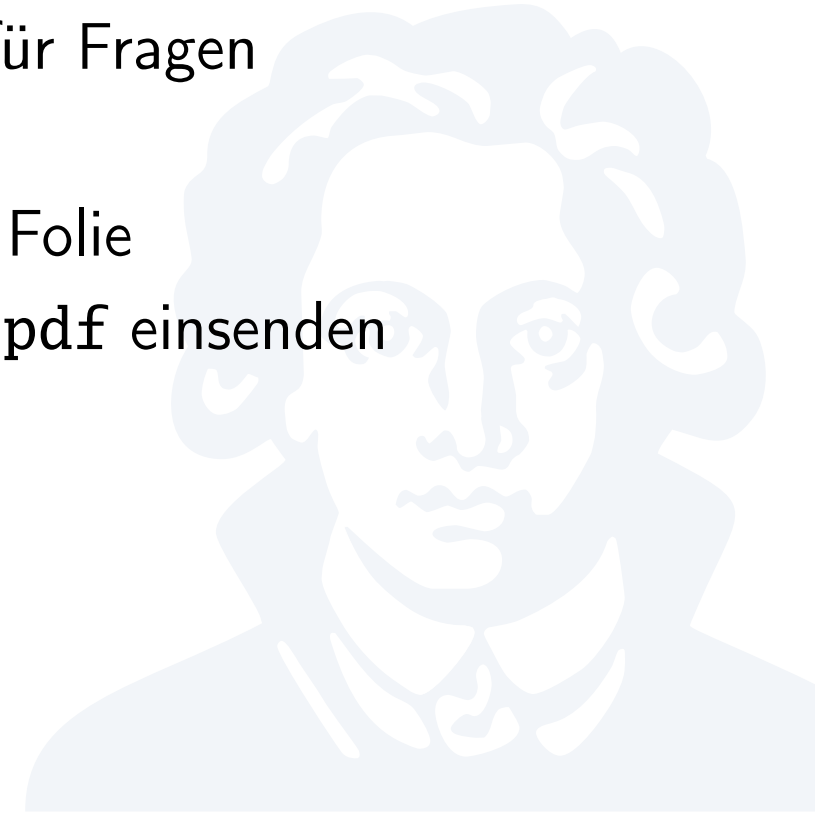
Ausarbeitung

- Einsendung: Eine Woche vor Vortrag
- Hintergrund, Zusammenfassung des Themengebietes
- Quellen angeben
- Ungefähr 15 pages (LaTeXarticle style)
- Möglichst
  - *verständlich*
  - *genau*
  - *vollständig*



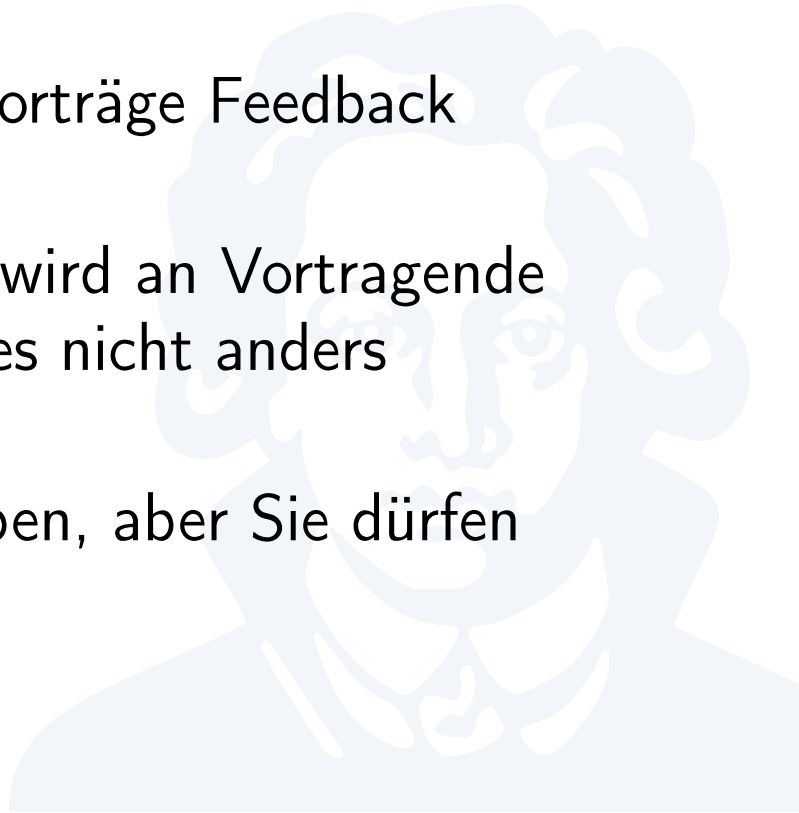
Vortrag

- Vorträge finden in Viererblocks statt
- 40-55 Minuten, inkl. 10 Minutes für Fragen
- Kurze Pause zwischen Vorträgen
- Übliche Annahme: 2 Minuten pro Folie
- Foliensatz 2 Tage vor Vortrag als `pdf` einsenden
- *Üben Sie*



Feedback (1/2)

- Erwartung: Sie sind jedes Mal anwesend
- Senden Sie Bitte zu jedem Vortrag *feedback*, innerhalb von 3 Tagen
- Sie *müssen* zu mindestens $\frac{2}{3}$ der Vorträge Feedback einsenden
- Besonders interessantes Feedback wird an Vortragende weitergeleitet (anonym, wenn Sie es nicht anders beantragen)
- Fragen für Feedback sind vorgegeben, aber Sie dürfen natürlich mehr schreiben

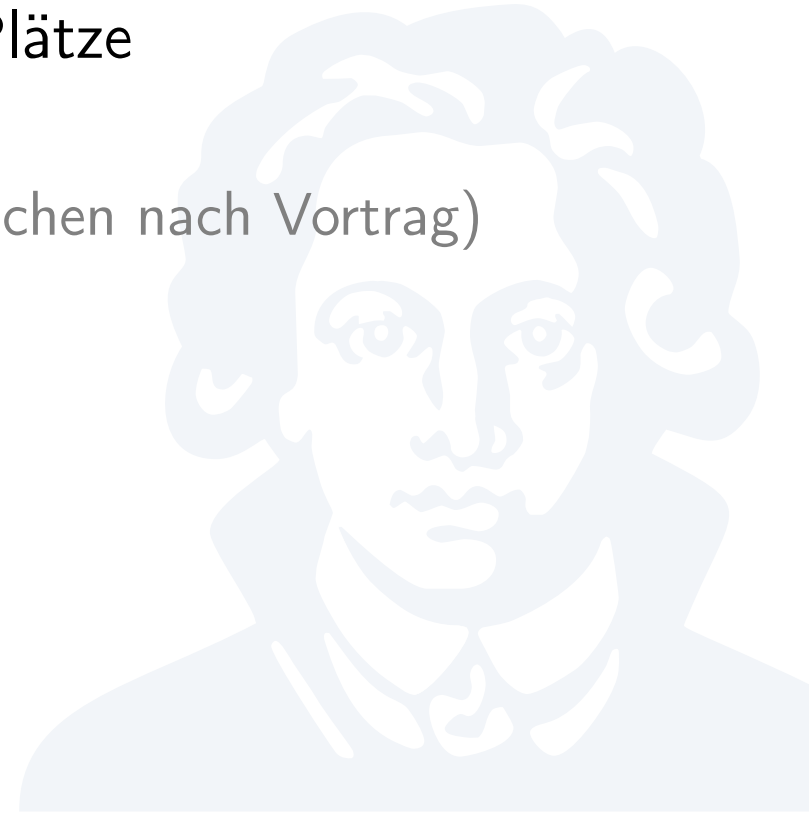


Feedback (2/2)

1. Welches grundlegende Problem wird in dem Vortrag behandelt?
2. Warum ist dieses Problem wichtig?
3. Warum haben frühere Arbeiten das Problem nicht lösen können?
4. Welche neuen Ideen und Beiträge sind ersichtlich?
5. Wie wurden die vorgetragenen Ideen und Beiträge ausgewertet?
6. Welche Resultate sind aus der Auswertung ersichtlich?
7. Was ist der stärkste Punkt der vorgestellten Arbeit?
8. Was ist der schwächste Punkt der vorgestellten Arbeit?
9. Wie würden Sie den vorgestellten Ansaz verbessern?

Vortragstermine

- Vortragssitzungen werden von hinten aufgefüllt
- Jeder Sitzungstermin hat 4 (5?) Plätze
- Daten:
  - 26.11.2013 (Ausarbeitung 2 Wochen nach Vortrag)
  - 10.12.2013
  - 21.01.2014
  - 04.02.2014

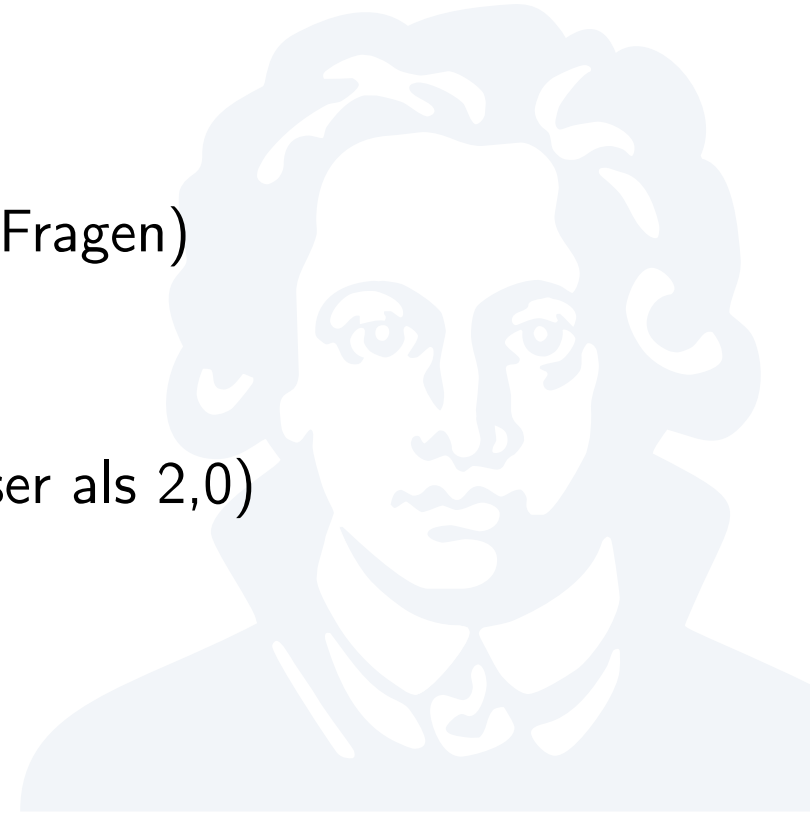


Themenauswahl

- Themen werden weiter hinten aufgeführt
- Themen sind im Seminarmerkblatt gesammelt
- Bis zum 19. (Samstag):
  - Senden Sie mir eine Liste der Themen, die Sie interessieren, in absteigender Präferenzreihenfolge
  - Wenn Sie zu zweit arbeiten möchten:
    - Senden Sie ZUSÄTZLICH Ihre beiden Namen und eine Team-Liste
    - Da die Themen begrenzt sind, kann Teamarbeit nicht garantiert werden
- Themenzuweisung ist algorithmisch
- Wenn kein Thema aus Ihrer Liste zugewiesen werden kann $\Rightarrow$ Zufall
- Themen– und Datumszuweisung am 21.

Benotung

- Beachten Sie Hinweise zur Benotung im Merkblatt
- Wesentlich: Seien Sie
  - *verständlich*
  - *genau*
  - *vollständig* (bzgl. der Feedback-Fragen)
- Andere Kriterien:
  - *Breite*
  - *Eigene Beiträge* (für Noten besser als 2,0)

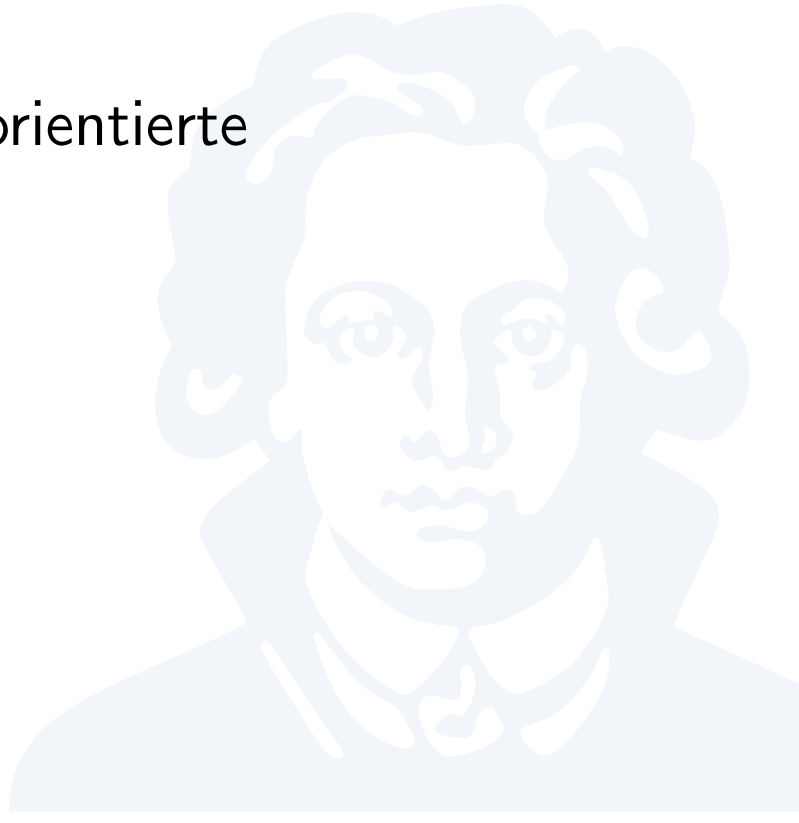


Teil II

Themengebiete und Themen

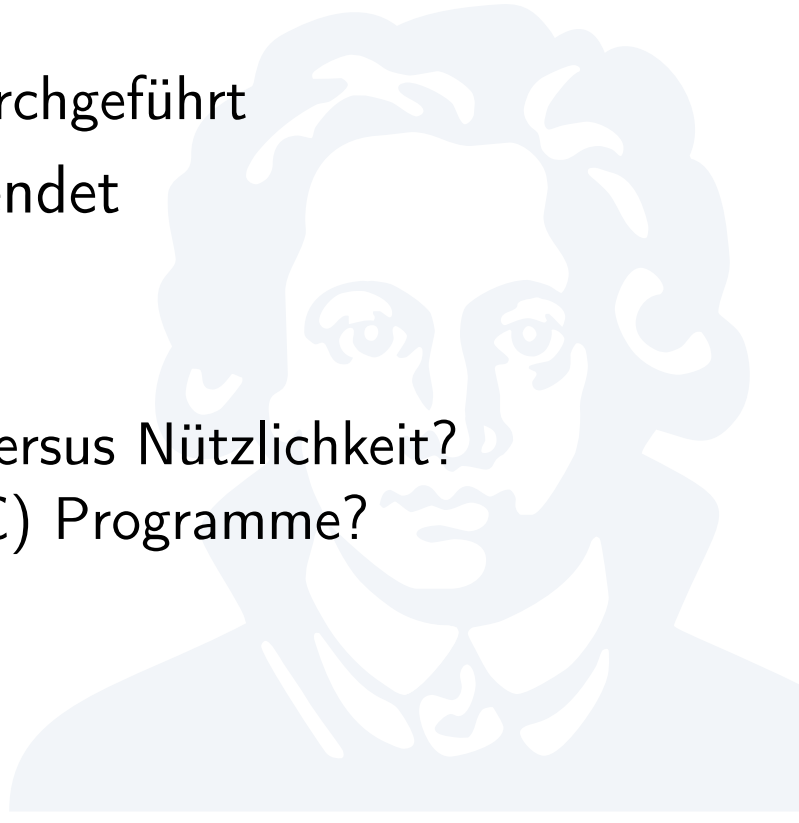
Themengebiete

1. Automatisches Refactoring
2. Metaprogrammierung und Aspektorientierte Programmierung
3. Parallele Programmiersprachen



Themengebiet 1: Automatisches Refactoring

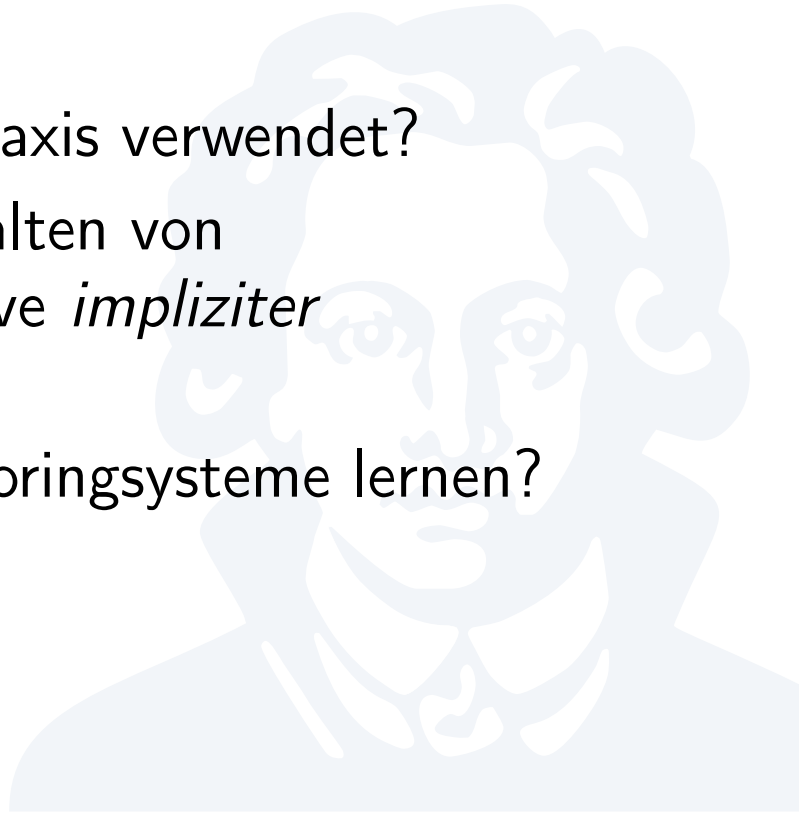
- Idee:
  - Benutzer wählt Programmtransformation
  - Softwarewerkzeug prüft, ob Transformation das Programmverhalten erhält
  - Falls ja: Transformation wird durchgeführt
- Mächtiges Konzept, vielfach verwendet
- Viele offene Fragen:
  - Gute UI?
  - Abwägungen bzgl. Korrektheit versus Nützlichkeit?
  - Skalierung auf große ($>$ 1MLOC) Programme?
  - Gleichzeitige Modifizierungen?
  - Modularität?



Thema 1.1

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, und Danny Dig, "A comparative study of manual and automated refactorings", PLDI 2012 (Peking)

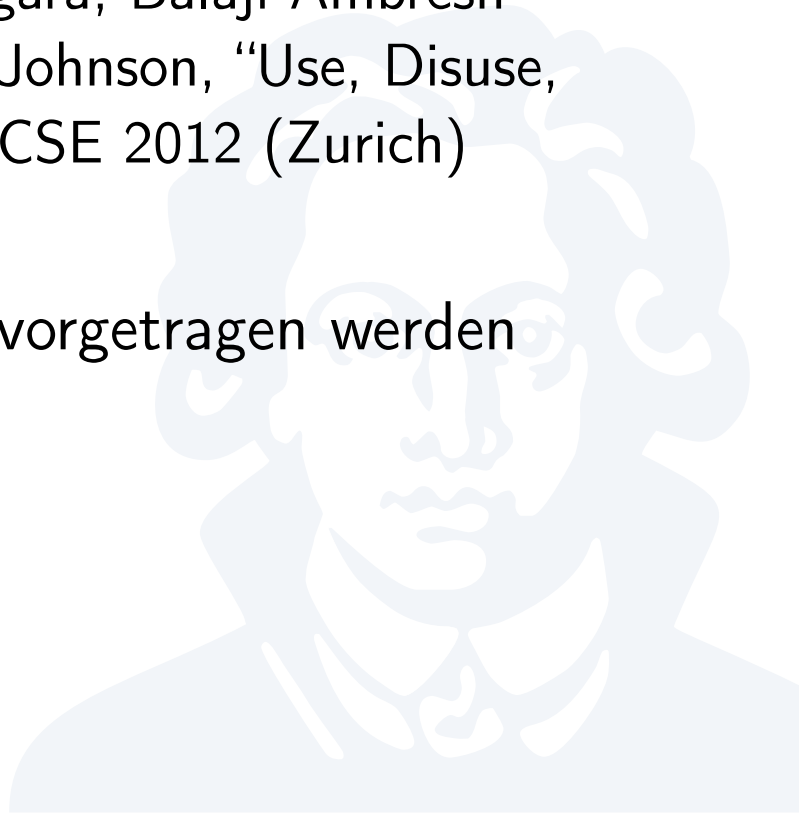
- Wie werden Refactorings in der Praxis verwendet?
- Autoren haben tatsächliches Verhalten von Programmierern analysiert, inklusive *impliziter* Refactorings
- Was können wir daraus für Refactoringsysteme lernen?



Thema 1.2

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian, P. Bailey, und Ralph E. Johnson, “Use, Disuse, and Misuse of Automated Refactorings”, ICSE 2012 (Zurich)

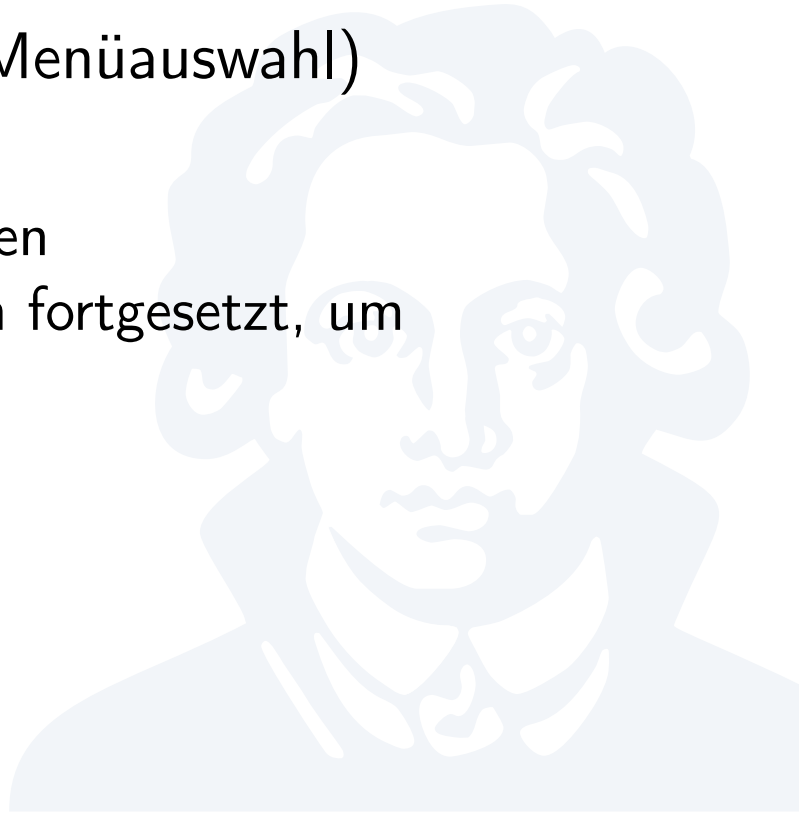
- Ähnlich dem vorherigen Papier
- Kann zusammen mit 1.1 zu zweit vorgetragen werden



Thema 1.3

Veselin Raychev, Max Schäfer, Manu Sridharan, Martin Vechev, “Refactoring with Synthesis”, OOPSLA 2013 (Indianapolis)

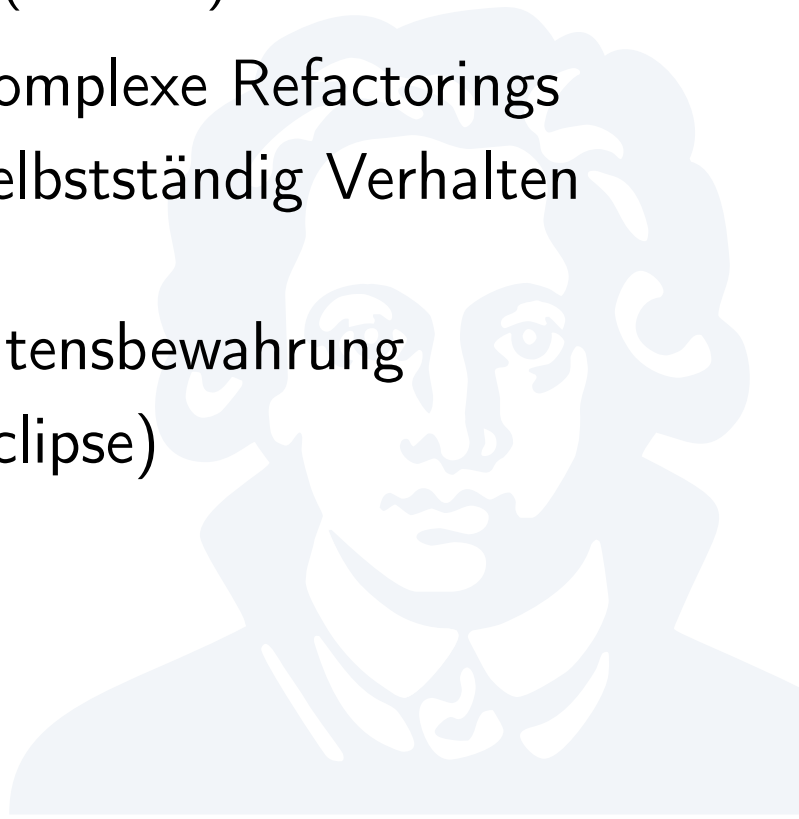
- Refactoring-UIs sind kompliziert (Menüauswahl)
- Alternative Idee:
  - Werkzeug beobachtet Änderungen
  - Änderungen werden automatisch fortgesetzt, um Verhalten zu erhalten
- Eclipse-Plugin



Thema 1.4

Christoph Reichenbach, Devin Coughlin, und Amer Diwan, "Program Metamorphosis", ECOOP 2008 (Genova)

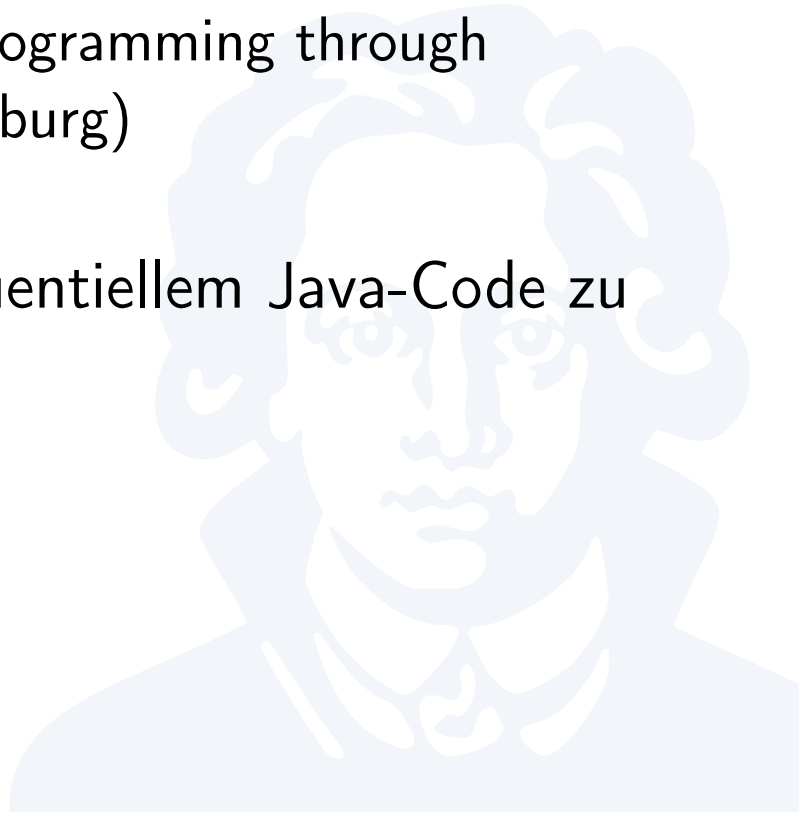
- Ein-Schritt-Refactoring erzwingt komplexe Refactorings
- Besser: kleine Schritte, die nicht selbstständig Verhalten beibehalten
- Separater Mechanismus zur Verhaltensbewahrung
- Eclipse-Plugin (alte Version von Eclipse)



Thema 1.5

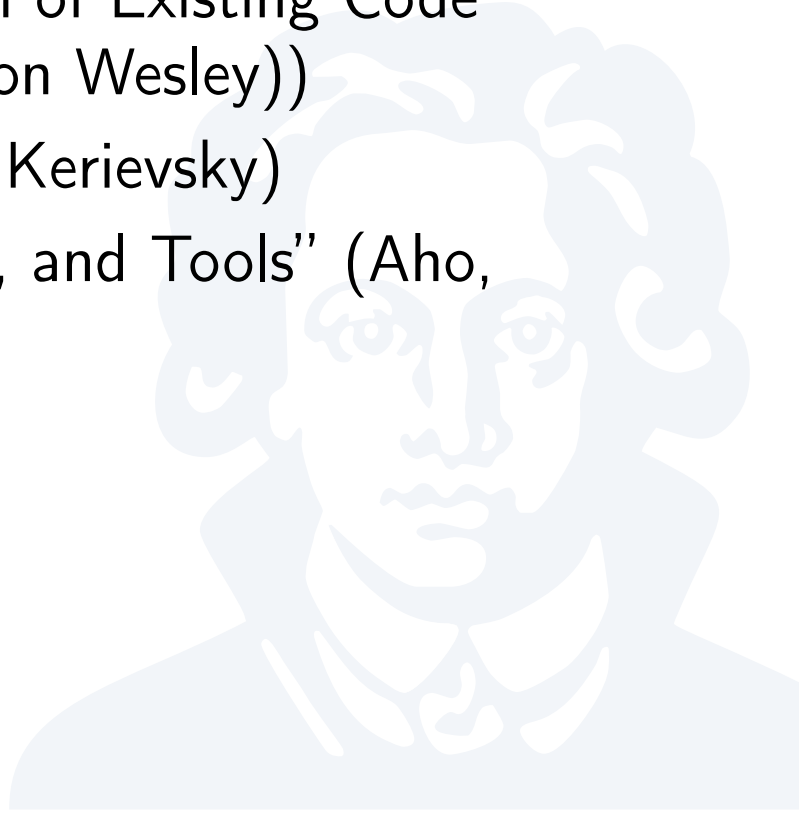
Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda, “Crossing the Gap from Imperative to Functional Programming through Refactoring”, ESEC/FSE 2013 (St. Petersburg)

- Anwendung von Refactoring:
  Programmtransformation von sequentiellem Java-Code zu parallelem Java-Code



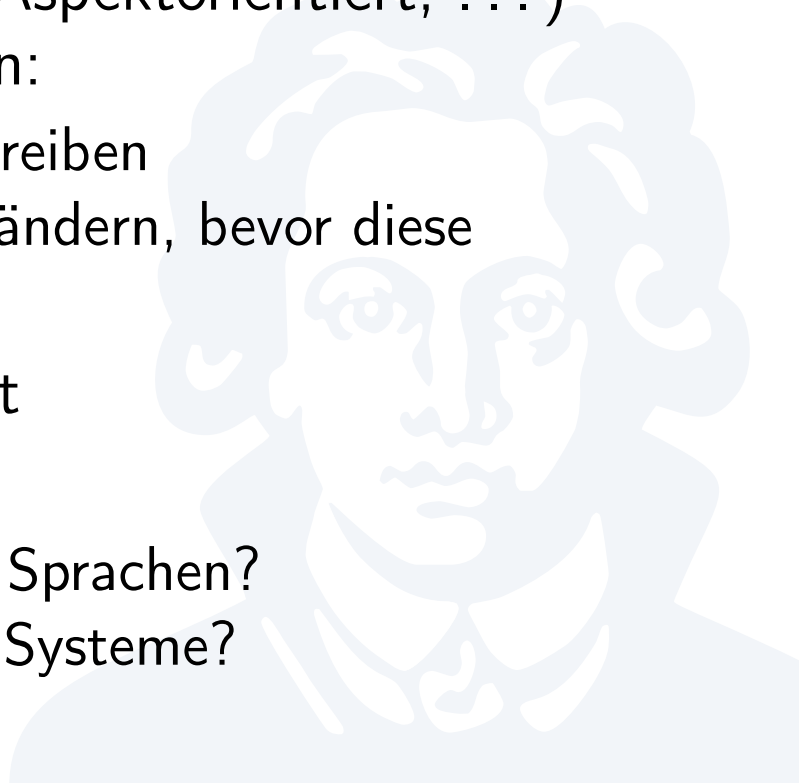
Themengebiet 1: Literatur

- "Refactoring: Improving the Design of Existing Code" (Martin Fowler et al., 1999, Addison Wesley))
- "Refactoring to Patterns" (Joshua Kerievsky)
- "Compilers: Principles, Techniques, and Tools" (Aho, Lam, Sethi, Ullman)



Themengebiet 2: Generative und Aspektorientierte Programmierung

- Abstraktionsmechanismen der meisten Programmiersprachen sind beschränkt
- Metaprogrammierung (Generativ, Aspektorientiert, ...) erweitert Abstraktionsmechanismen:
  - Programme, die Programme schreiben
  - Programme, die Programme verändern, bevor diese übersetzt werden
- Wesentliche Fragen scheinen gelöst
- Weiterhin offen:
  - Bessere Integration mit anderen Sprachen?
  - Bessere Wartbarkeit generativer Systeme?
  - Modularität?



Thema 2.1 (B)

Aspektorientierte Programmierung mit AspectJ

- AOP Erlaubt Programmverhalten jenseits traditioneller Modularitätskonzepte
- Beispiel: 'Alle Methoden, deren Namen mit `print` anfängt, werden unterdrückt'
- Aufgabe: Untersuchen Sie Aspektorientierte Programmierung in Java mit AspectJ
  - Wie wird es verwendet?
  - Wie funktioniert es intern?

Spezifische Literatur:

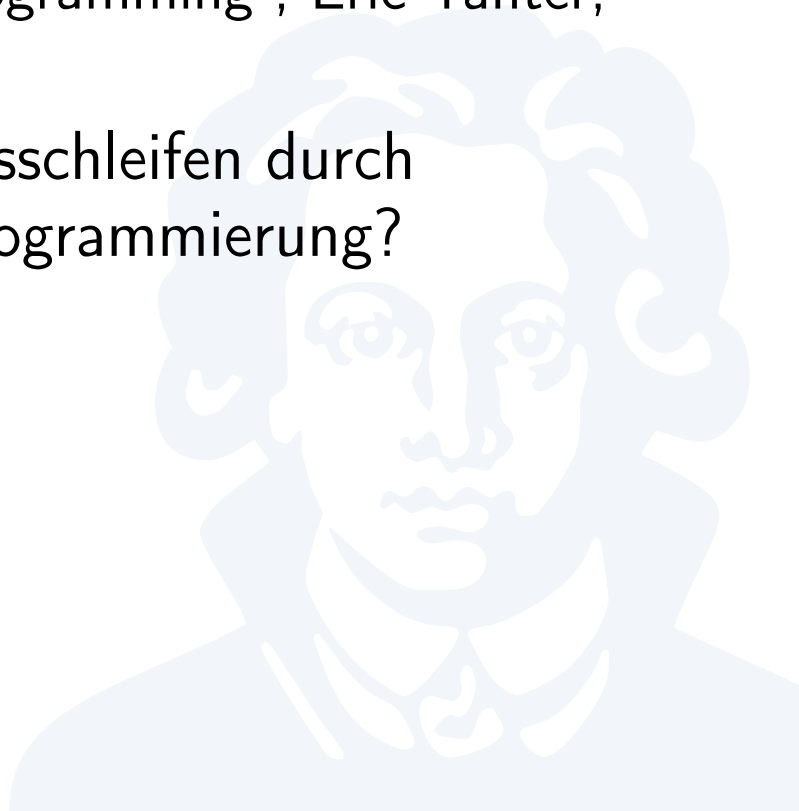
- "Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP" (Oliver Böhm)
- Kiczales et al., "Aspect-Oriented Programming" (OOPSLA 1997).
- Kiczales et al., "An overview of AspectJ" (ECOOP 2001)

Kann auch zu zweit bearbeitet werden.

Thema 2.2

"Execution Levels for Aspect-Oriented Programming", Eric Tanter, AOSD 2010 (Rennes and Saint-Malo)

- Problem: wie vermeiden wir Endlosschleifen durch Seiteneffekte aspektorientierter Programmierung?
- Ansatz: Strukturierung



Thema 2.3 (B)

Metaobjektprotokolle

- Metaobjektprotokolle erlauben das Umdefinieren von:
  - Objekterzeugung
  - Vererbung
  - Methodenaufrufe
- Aufgabe:
  - Vergleichen Sie Metaobjektprotokolle zweier Sprachen (Groovy, C++, Common LISP, . . .)
  - Entwickeln Sie geeignete Vergleichskriterien (z.B. Ausdrucksfähigkeit)

Kann auch zu zweit bearbeitet werden.

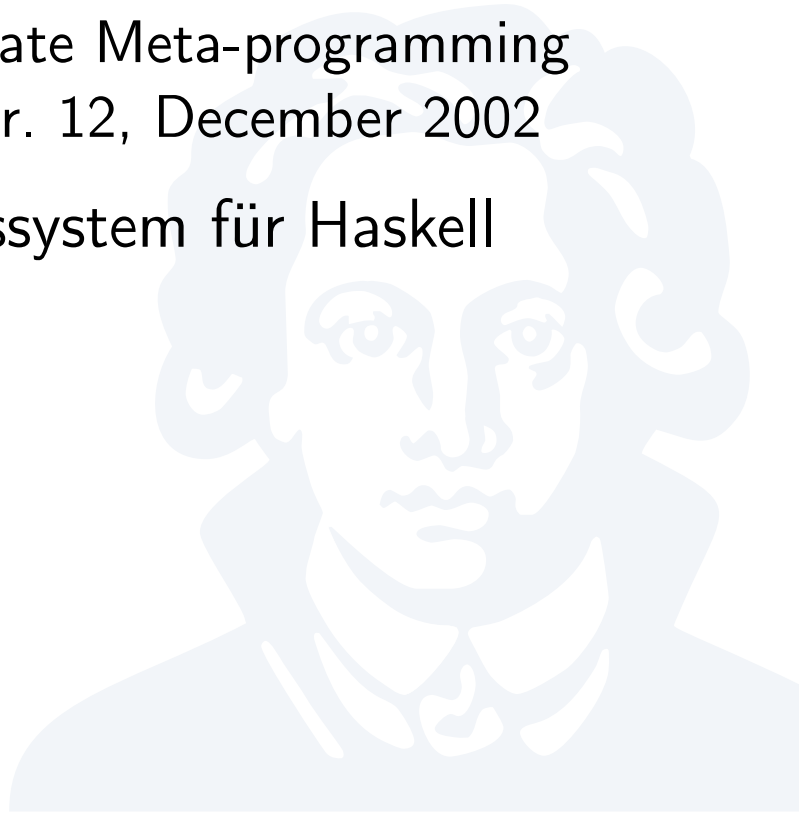
Spezifische Literatur:

- "The Art of the Metaobject Protocol", Kiczales, des Rivieres, and Bobrow, aus meinem Bürobestand
- Shigeru Chiba, "A Metaobject Protocol for C++ ", OOPSLA 1995 (Austin, TX)

Thema 2.4

Tim Sheard, Simon Peyton Jones, “Template Meta-programming for Haskell”, SIGPLAN Notices, vol. 37, Nr. 12, December 2002

- Generisches Metaprogrammierungssystem für Haskell



Thema 2.5 (B)

Language Workbenches

- Domänenspezifische Sprachen (DSLs) erlauben „paßgenaue“ Lösungssprachen
- Technische Probleme:
  - Integration mit Allzwecksprachen
  - Integration mit Werkzeugen
- Language Workbenches gehen diese Probleme durch (teil)formale Lösungsansätze an
- Aufgabe: Vergleich zweier Language Workbenches

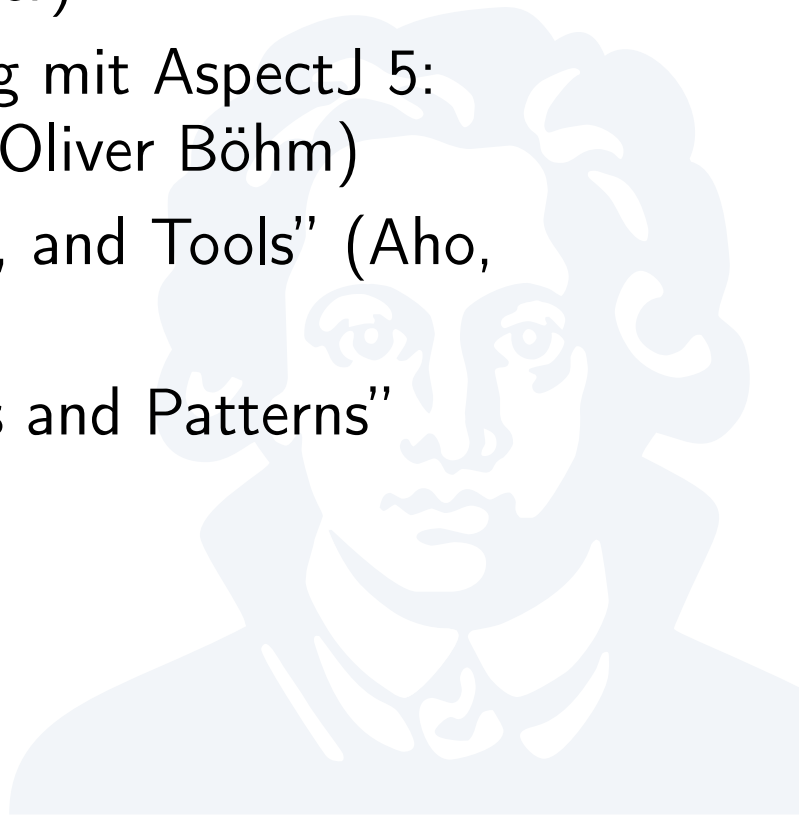
Spezifische Literatur:

- Lennart C. L. Kats, Eelco Visser. “The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs.” (OOPSLA 2010)
- Völter, “oAW xText: A framework for textual DSLs”

Kann auch zu zweit vorgetragen werden

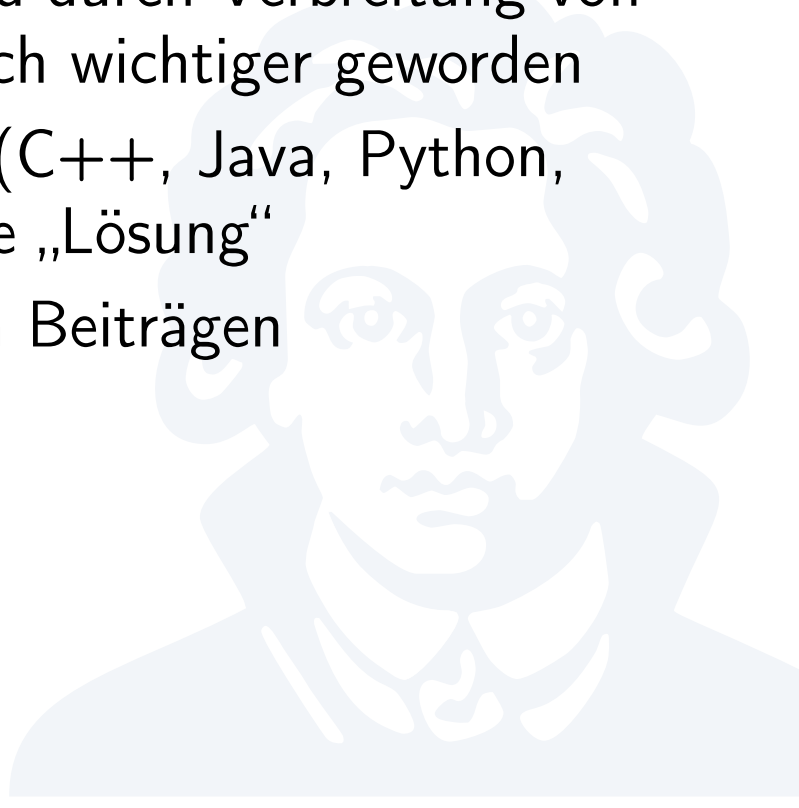
Themengebiet 2: Literatur

- "Generative Programming: Methods, Tools, and Applications" (Czarnecki, Eisenecker)
- "Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP" (Oliver Böhm)
- "Compilers: Principles, Techniques, and Tools" (Aho, Lam, Sethi, Ullman)
- "Software Product Lines: Practices and Patterns" (Clements, Northrop)



Themengebiet 3: Parallele Programmiersprachen

- Parallele Programmiersprachen sind durch Verbreitung von Multicore-Systemen und GPUs noch wichtiger geworden
- Verbreitete Programmiersprachen (C++, Java, Python, ...) immer noch keine vollständige „Lösung“
- Großer Ideenwettbewerb mit vielen Beiträgen



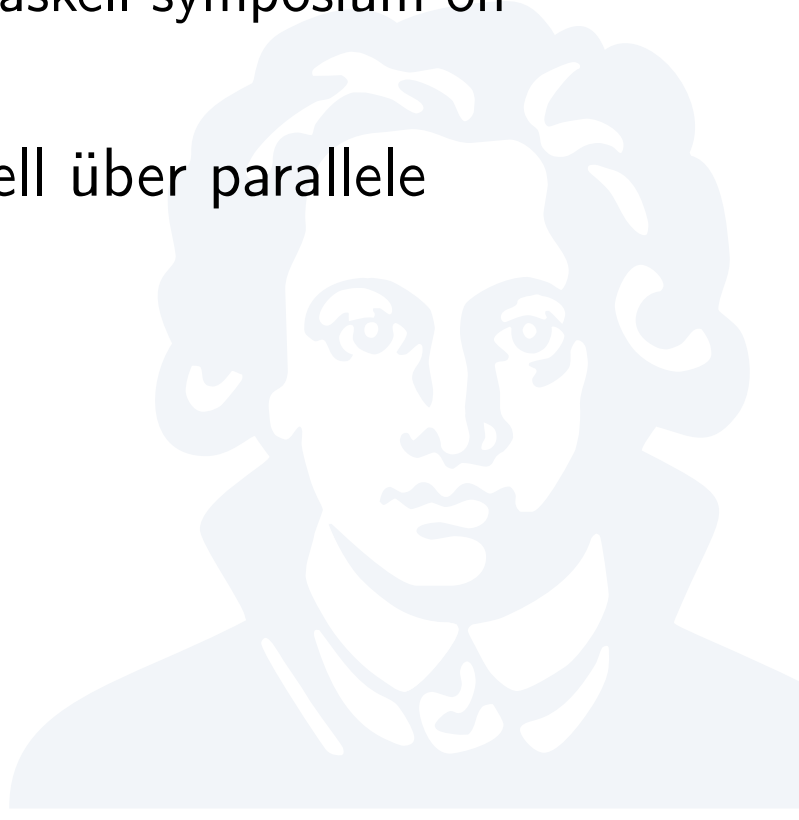
Thema 3.1

Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, Phil Trinder "Seq no more: Better Strategies for Parallel Haskell", Proceedings of the third ACM Haskell symposium on Haskell, Seiten 91–102

- Parallele Programmierung in Haskell über parallele Kombinatoren

Empfolenes Vorwissen:

- Haskell
- Monads



Thema 3.2

Viktor Vafeiadis, Chinmay Narayan, "Relaxed Separation Logic: A Program Logic for C11 Concurrency", OOPSLA 2013 (Indianapolis)

- C11 (als erste Version von C) beinhaltet ein Speichermodell
- Dieses Papier formalisiert die zugrundeliegenden Konzepte
- Erlaubt etvl. Korrektheitsbeweise, Optimierungen

Empfohlenes Vorwissen:

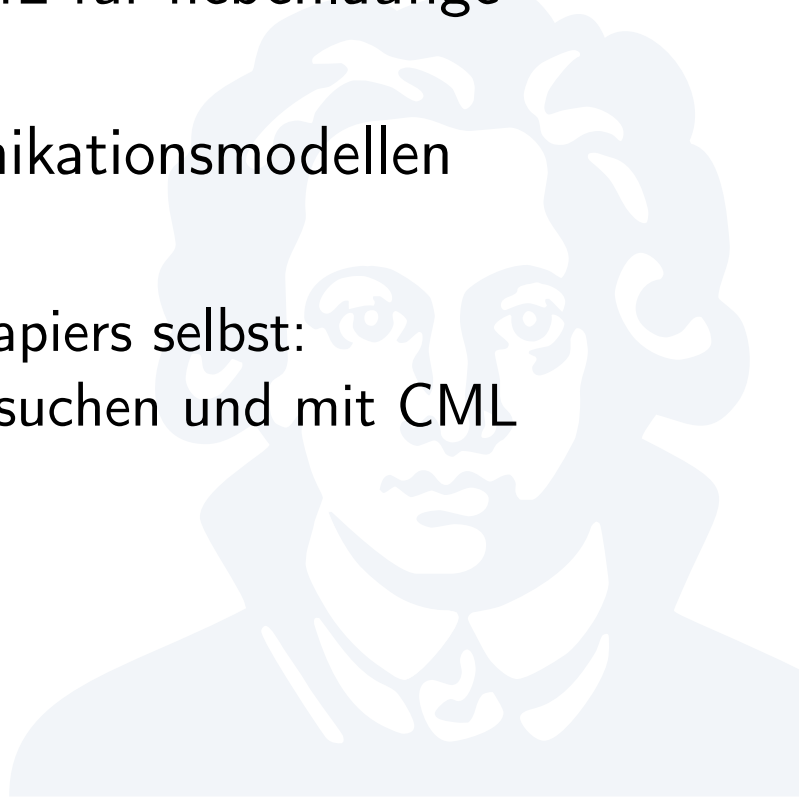
- Lambdakalkül (insbes. calculus of constructions)
- Formale Beweissysteme / Beweisassistenten (speziell Coq)

Thema 3.3 (B)

John H. Reppy, “CML: A Higher-order Concurrent Language”, Cornell

- Spracherweiterung zur Sprache SML für nebenläufige Prozesse
- Vergleichbar mit anderen Kommunikationsmodellen
- Aufgabe:
  - Zusätzlich zum Vorstellen des Papiers selbst: neuere, ähnliche Entwicklungen suchen und mit CML vergleichen

Kann zu zweit vorgetragen werden.



Thema 3.4 (B)

Aktuelle sprachbasierte Ansätze zum Datenparallelismus

- Vergleichen Sie drei aktuelle sprachbasierte Ansätze zur Parallelisierung:
  - Parallele Ausführung in Java 8 mit Lambdas (Project Lambda)
    Kombinator-basiert (strikt)
  - Parallel LINQ (PLINQ, für Microsoft's .NET-System)
    Deklarativ mit expliziter Parallelisierung
  - PQL (Reichenbach, Smaragdakis, Immerman, "PQL: A Purely-Declarative Java Extension for Parallel Programming", ECOOP 2012)
    Deklarativ mit impliziter Parallelisierung
- Entwickeln Sie Vergleichskriterien

Kann auch zu zweit vorgetragen werden

Thema 3.5 (B)

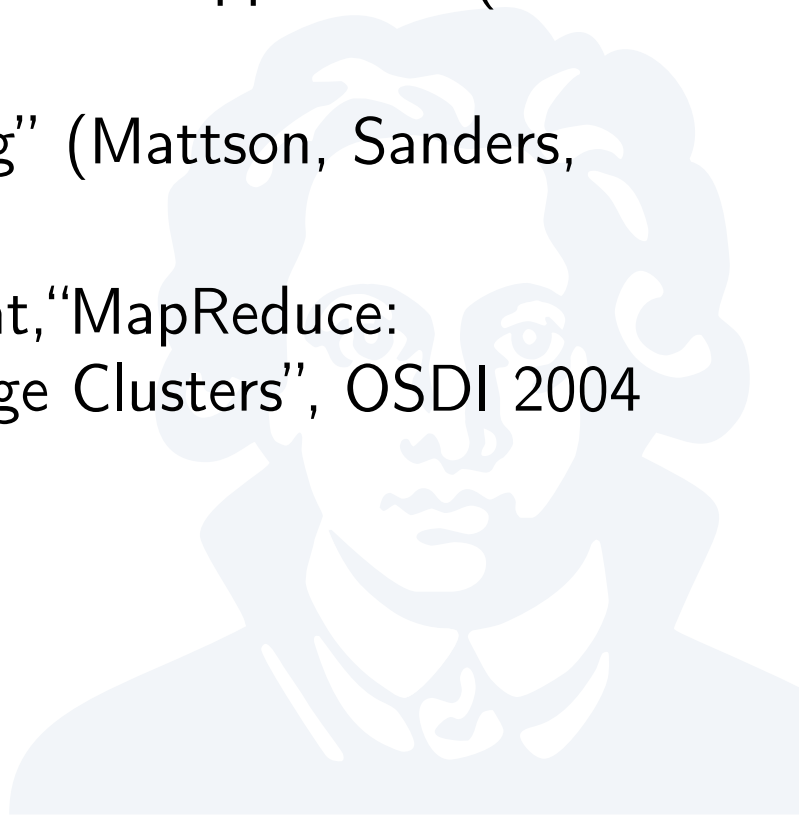
Partitionierende parallele Sprachen

- Partitionierung: Aufteilung von Problemen und Rechenressourcen
- Vergleichen Sie:
  - X10
  - Church
  - Chapel
- Wie wird Partitionierung gelöst?
- Entwickeln Sie Vergleichskriterien

Kann auch zu zweit vorgetragen werden

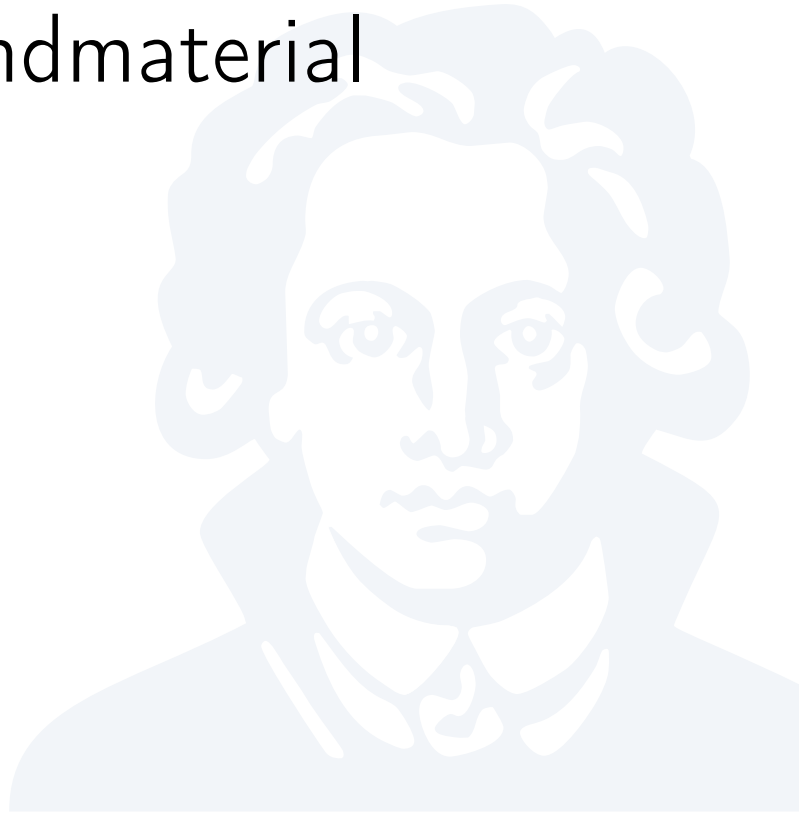
Themengebiet 3: Literatur

- “Computer Architecture: A Quantitative Approach” (John LeRoy Hennessy, David Patterson)
- “Patterns for Parallel Programming” (Mattson, Sanders, Massingill)
- Jeffrey Dean and Sanjay Ghemawat,“MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004



Teil III

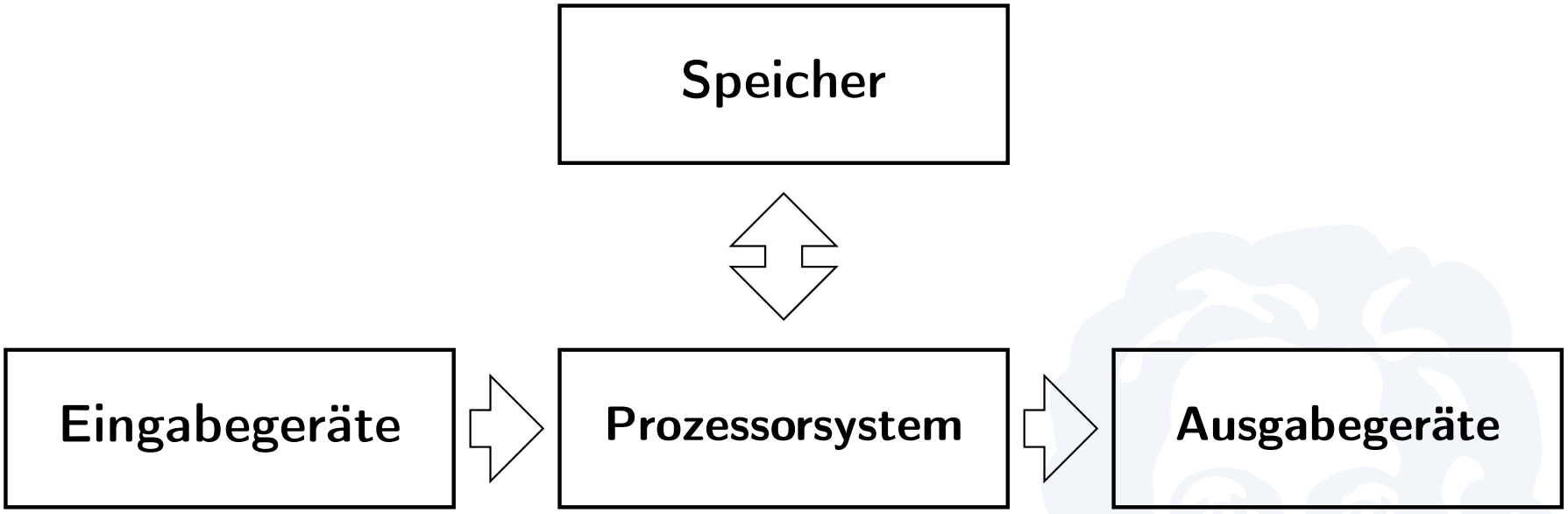
Wichtiges Hintergrundmaterial



Übersicht

1. Programmausführung
2. Programmrepräsentierung
3. Programmanalyse

Programmausführung



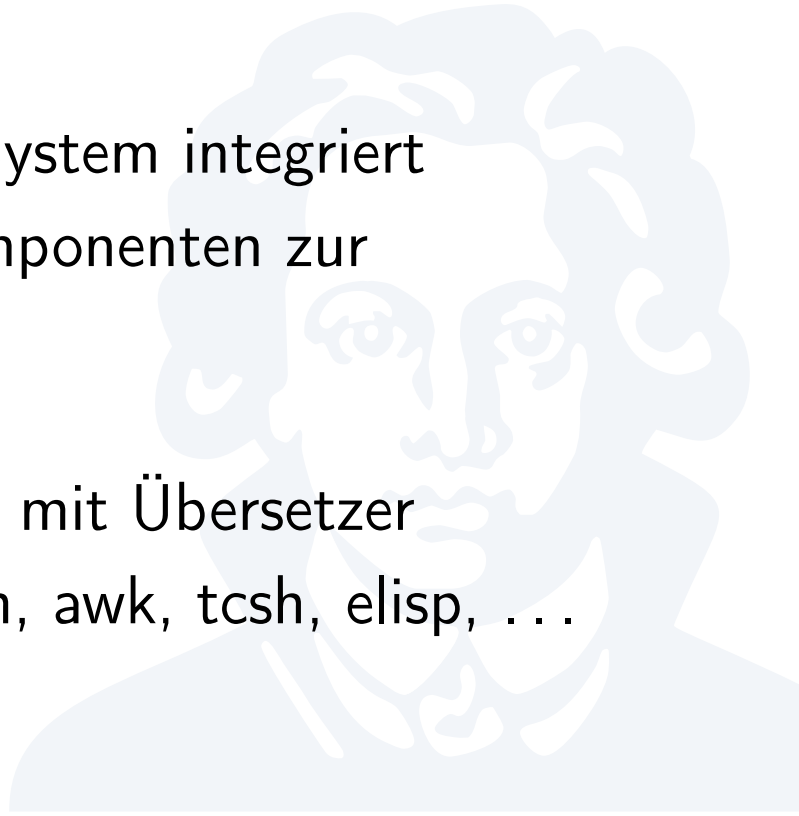
- Der *Prozessor*:
  - Liest Maschinencode aus dem Speicher
  - Führt Maschinencode aus
- Das *Betriebssystem*:
  - Kopiert Programm-Maschinencode in den Speicher
  - Richtet Prozessor auf Maschinencode

Drei Ausführungsmodelle

- Interpreter
- Übersetzer
- JIT (Just-In-Time-Übersetzer)

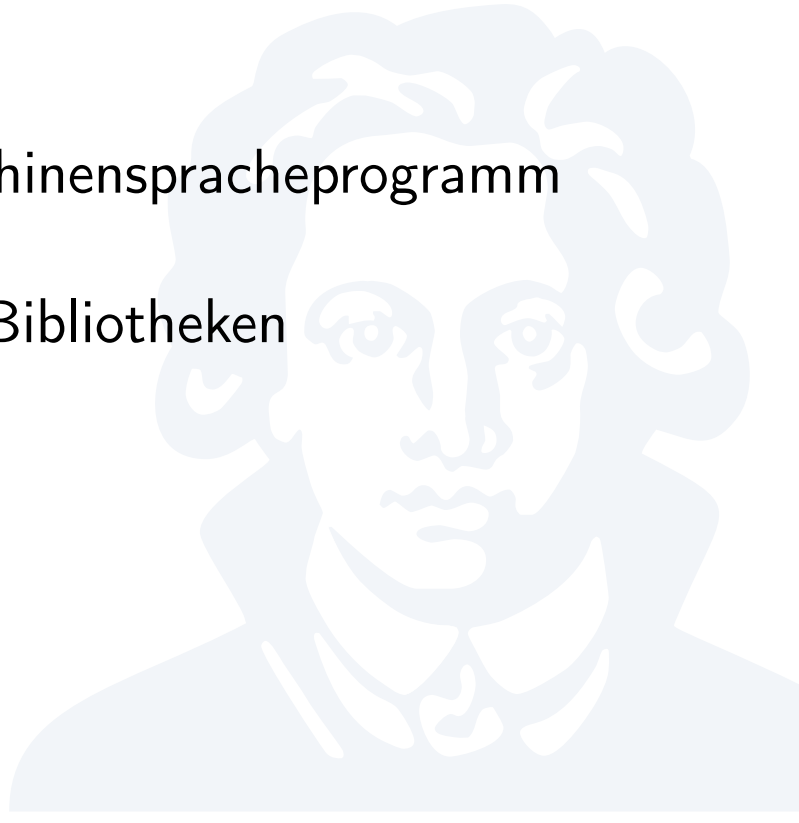
Drei Ausführungsmodelle: Interpreter

- Interpreter ist Schleife in Maschinencodeprogramm:
  - Lies nächsten Befehl
  - Dekodiere Befehl
  - Führe Befehl aus
- Alle Komponenten sind in einem System integriert
- Laufzeitsystem beinhaltet alle Komponenten zur Programmanalyse
- Relativ einfach zu implementieren
- 10–20× langsamer als Ausführung mit Übersetzer
- Beispiele: Python, Ruby, Perl, bash, awk, tcsh, elisp, ...



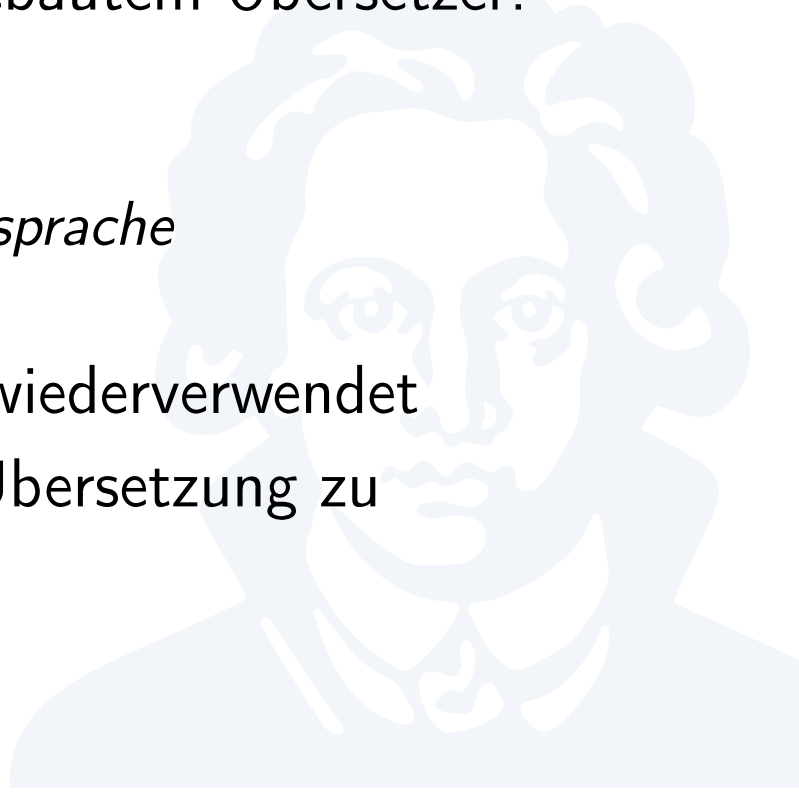
Drei Ausführungsmodelle: Übersetzer

- Strikte Trennung:
  - Übersetzung:
    Quellspracheprogramm $\rightarrow$ Maschinenspracheprogramm
  - Ausführung:
    Maschinenspracheprogramm + Bibliotheken



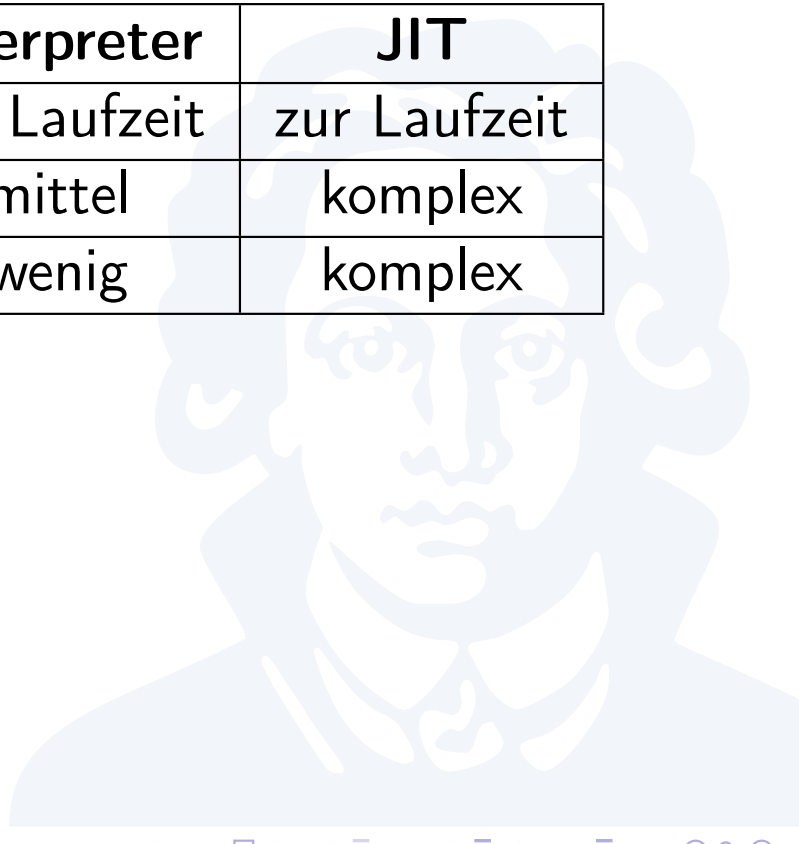
Drei Ausführungsmodelle: JIT

- Just-In-Time–Übersetzung: Übersetzt “gerade noch rechtzeitig”
- Ähnlich Interpreter, aber mit eingebautem Übersetzer:
  - Lies nächste Befehle
  - Dekodiere Befehle
  - *Übersetze Befehle in Maschinensprache*
  - *Führe Maschinensprache aus*
- Maschinencode wird gespeichert, wiederverwendet
- Kann Laufzeitwissen nutzen, um Übersetzung zu verbessern



Drei Ausführungsmodelle: Zusammenfassung

	Übersetzer	**Interpreter**	**JIT**
Programmanalyse	vor Laufzeit	zur Laufzeit	zur Laufzeit
Laufzeitsystem	minimal	mittel	komplex
Optimierungen	komplex	wenig	komplex

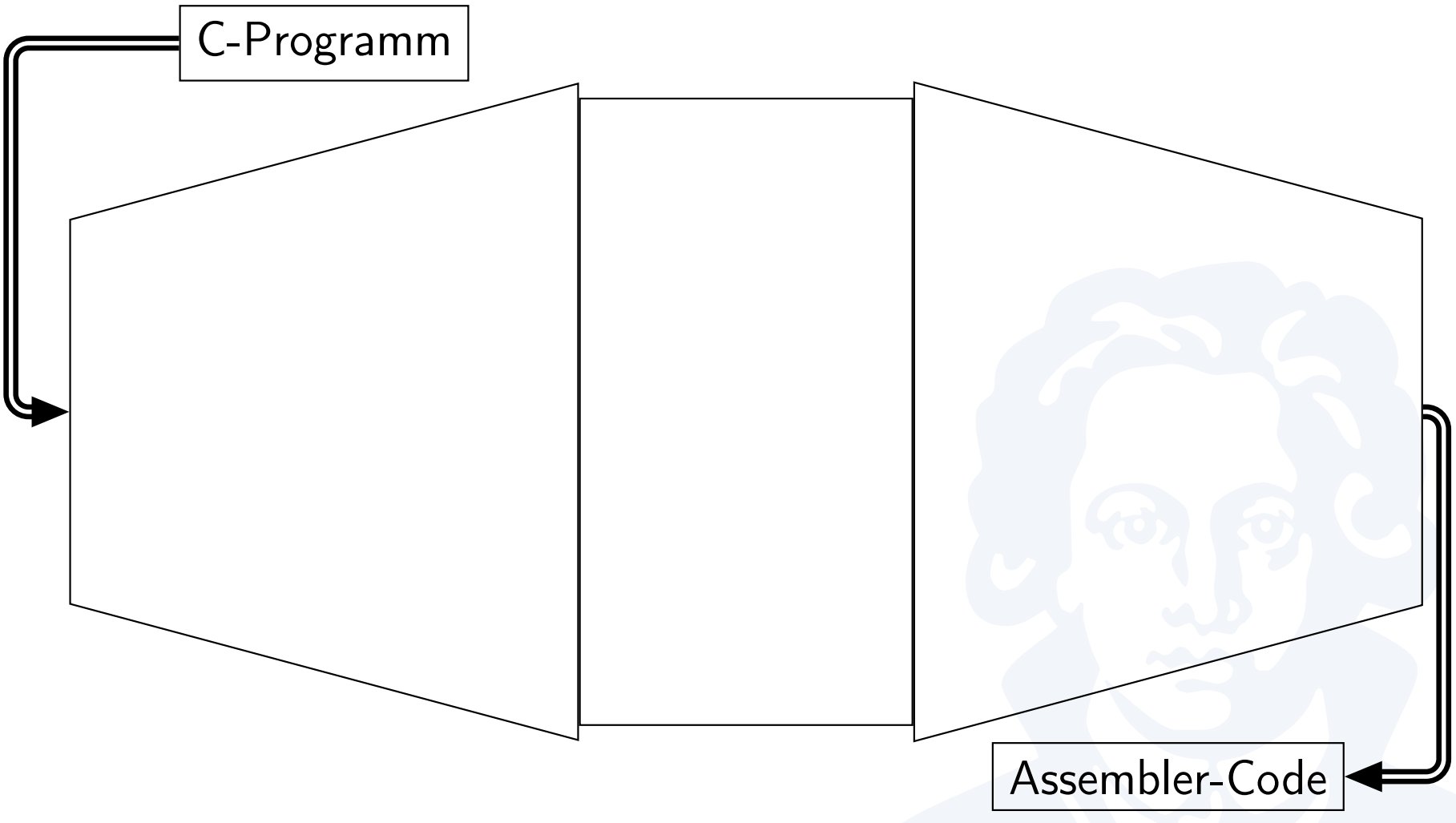


Frontend

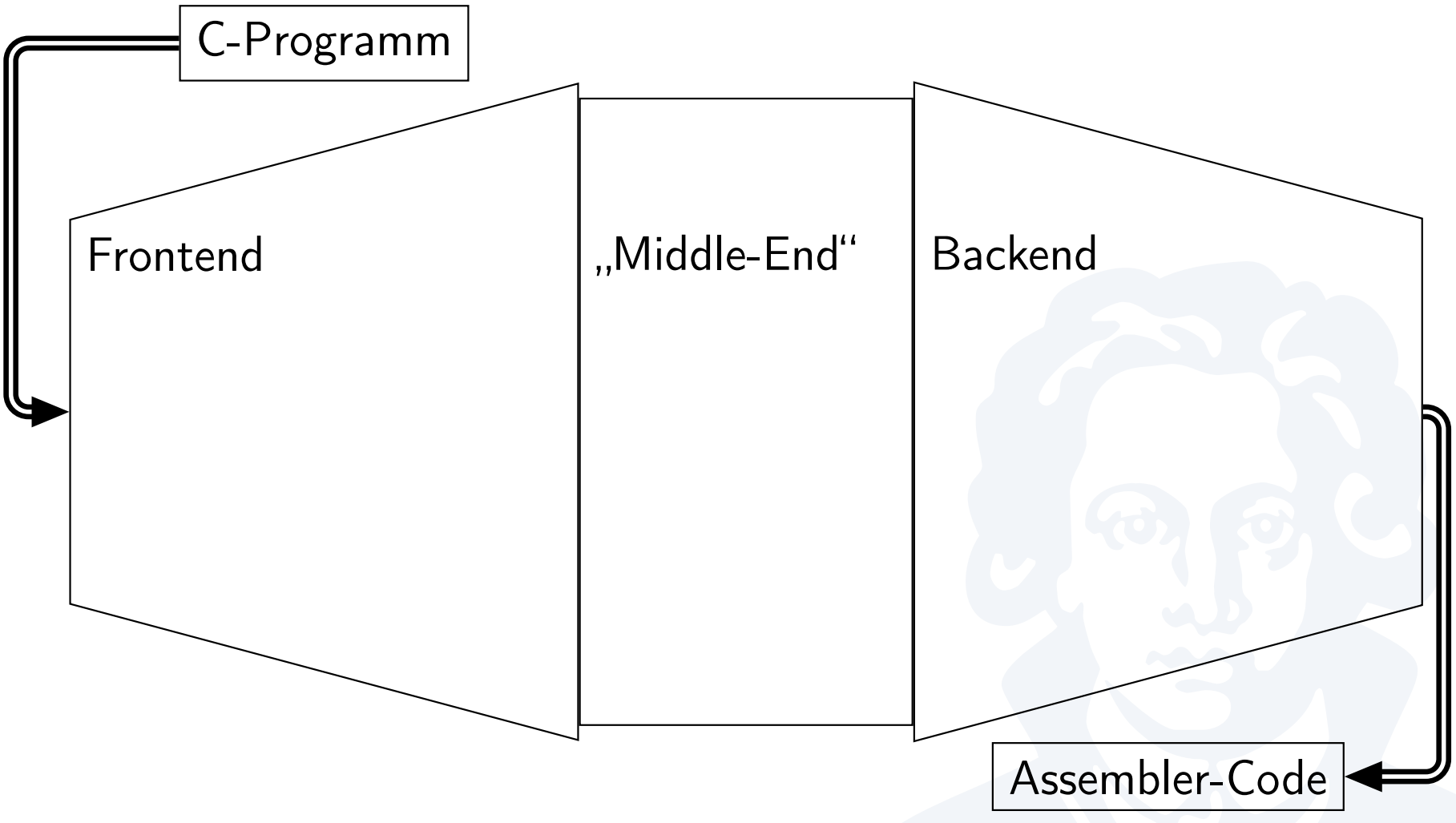
Gemeinsamkeit aller Ausführungsmodelle:

- Jedes Ausführungsmodell muß:
  - Programm einlesen
  - Programm analysieren
  - Fehler finden

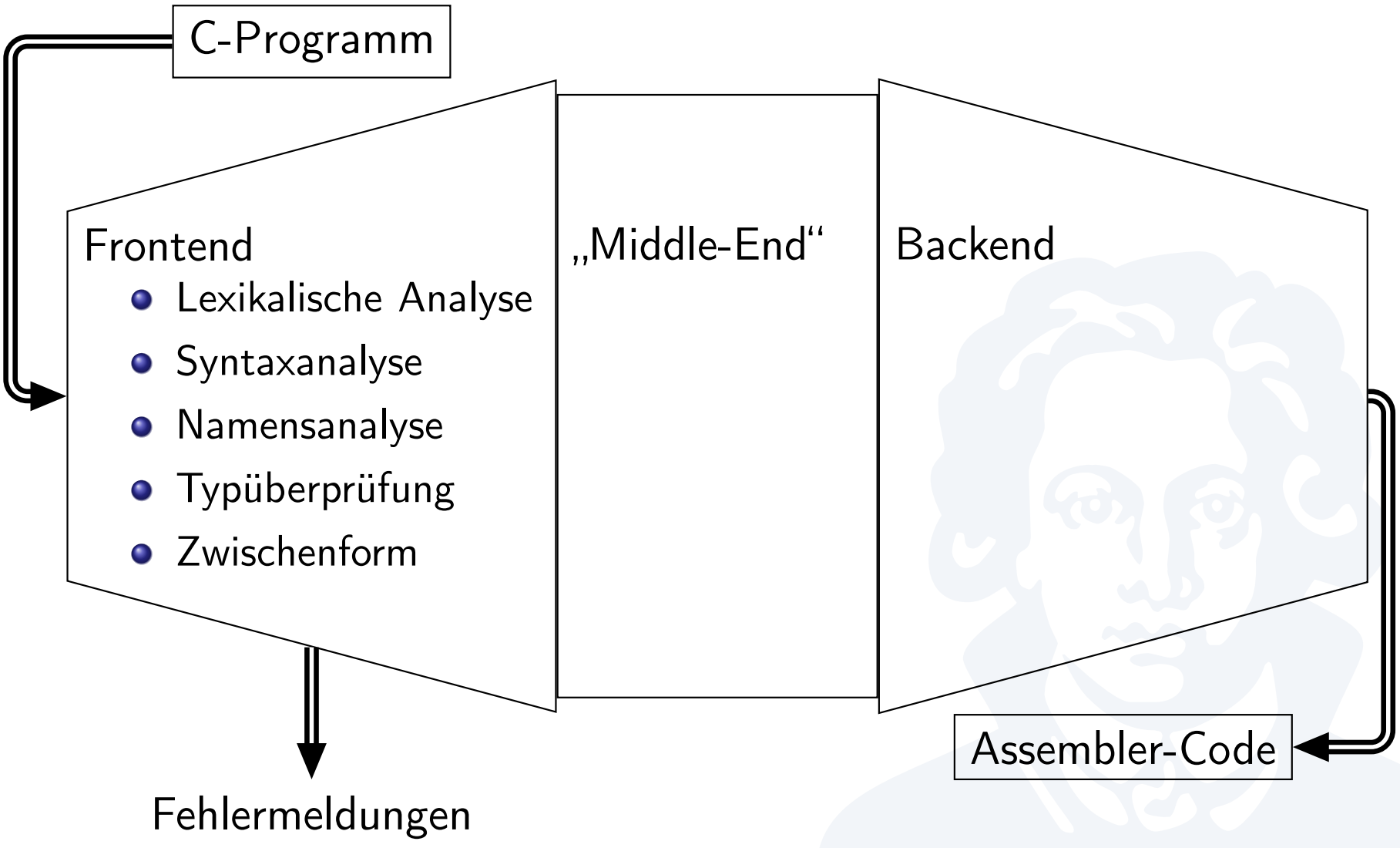
Der Übersetzer



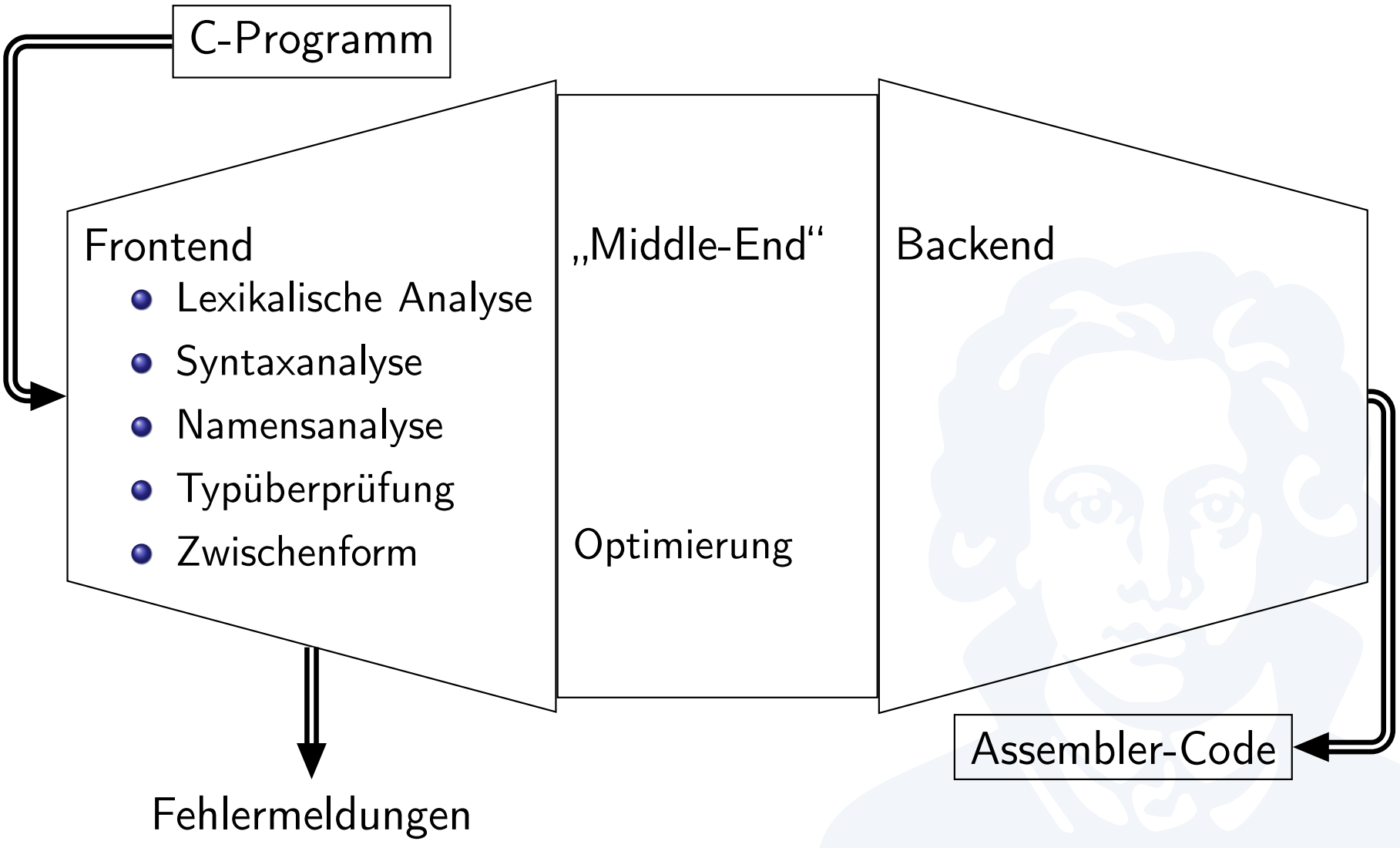
Der Übersetzer



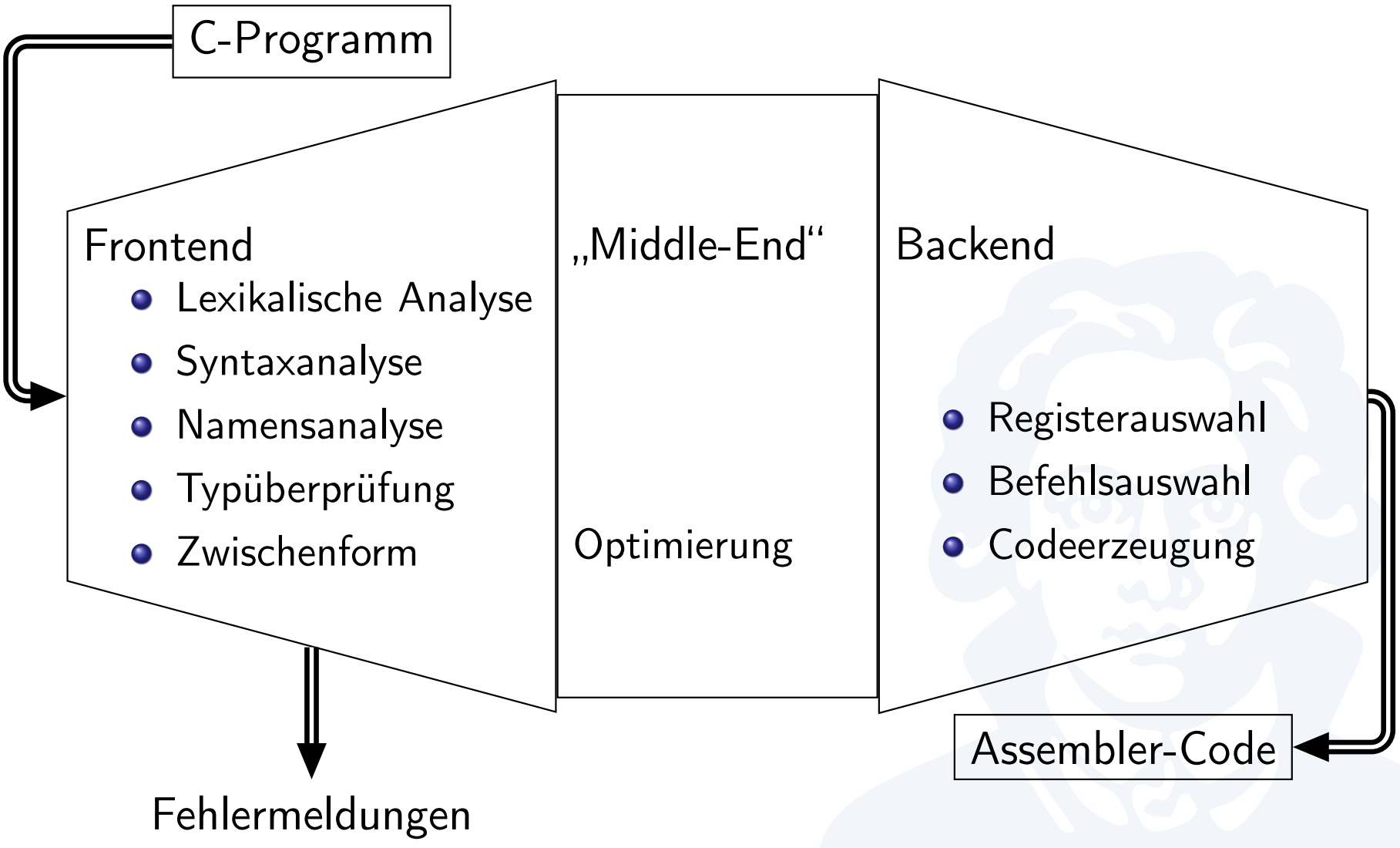
Der Übersetzer



Der Übersetzer



Der Übersetzer



Der Übersetzer: Frontend: Lexikalische Analyse

Eingabe:

```
{
  int x = 17 + 3L;
  char *c = x;
}
```

Der Übersetzer: Frontend: Lexikalische Analyse

Eingabe:

```
{
  int x = 17 + 3L;
  char *c = x;
}
```

- Lexikalische Analyse zerlegt Eingabe in Tokens/Lexeme
- Erzeugt vom Lexer (*Tokenizer*, *Scanner*)
- Ignoriert Leerzeichen, Zeilenumbrüche etc. (in C und C-artigen Sprachen)

Der Übersetzer: Frontend: Lexikalische Analyse

Eingabe:

```
{
  int x = 17 + 3L;
  char *c = x;
}
```

- Lexikalische Analyse zerlegt Eingabe in Tokens/Lexeme
- Erzeugt vom Lexer (*Tokenizer*, *Scanner*)
- Ignoriert Leerzeichen, Zeilenumbrüche etc. (in C und C-artigen Sprachen)

Tokens und Lexeme:

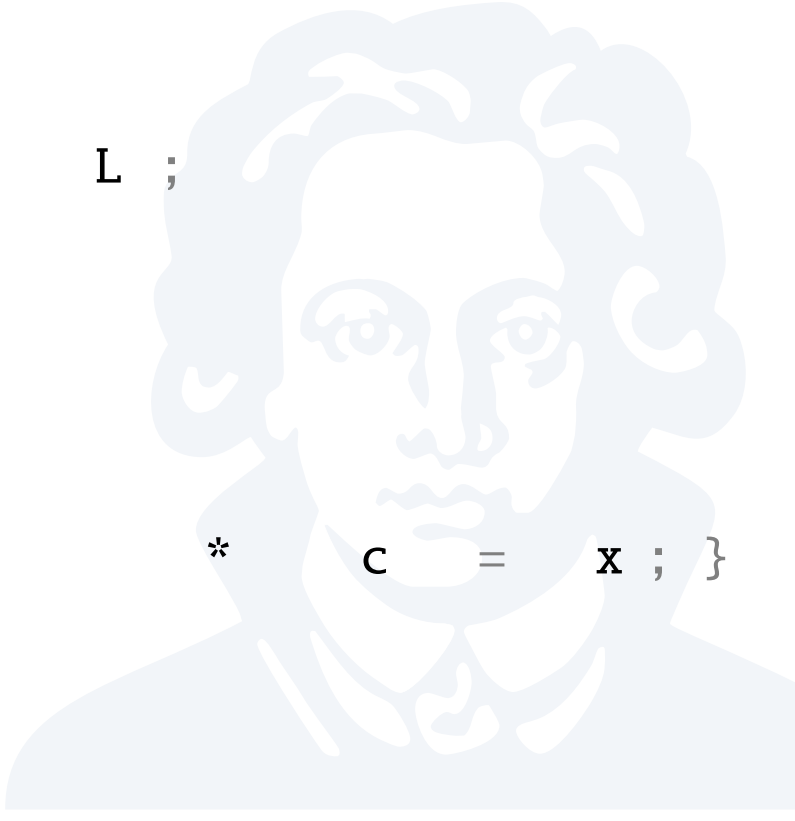
punctuator	{
int	int
identifier	x
punctuator	=
constant	17
punctuator	+
constant	3
long-suffix	L
punctuator	;
char	char
punctuator	*
identifier	c
punctuator	=
identifier	x
punctuator	;
punctuator	}

Der Übersetzer: Frontend: Parser

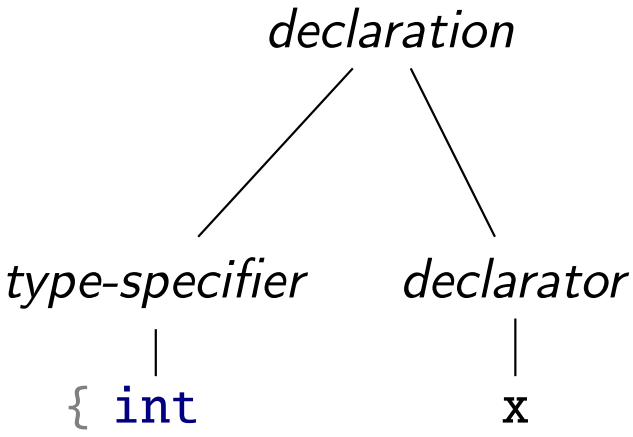
```
{ int x = 17 + 3 L ;
```

```
{
  int x = 17 + 3L;
  char *c = x;
}
```

```
char * c = x ; }
```



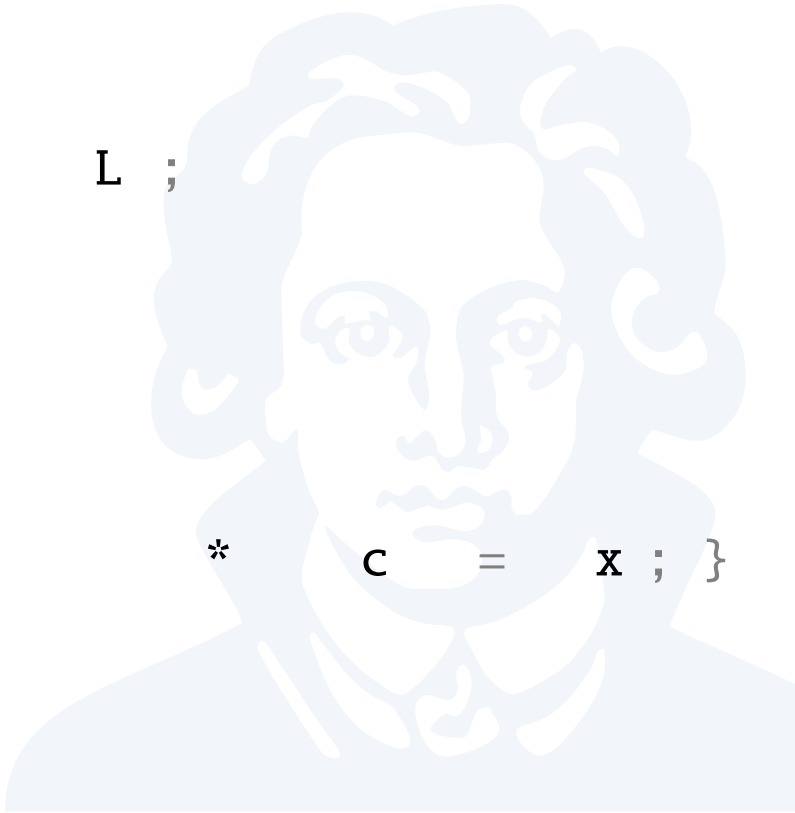
Der Übersetzer: Frontend: Parser



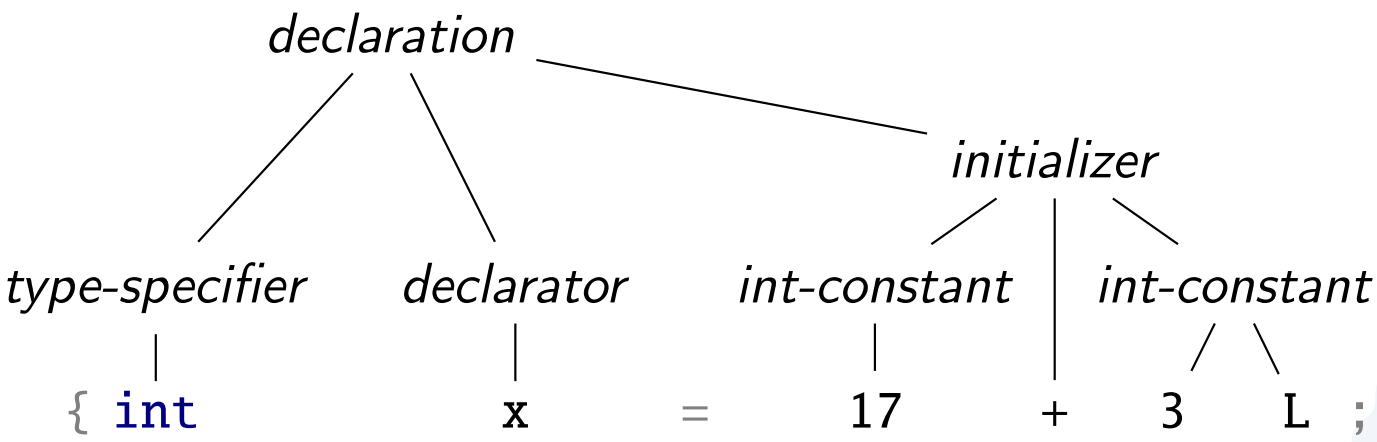

```
{ int x = 17 + 3L;
```

```
{
  int x = 17 + 3L;
  char *c = x;
}
```

```
char * c = x; }
```

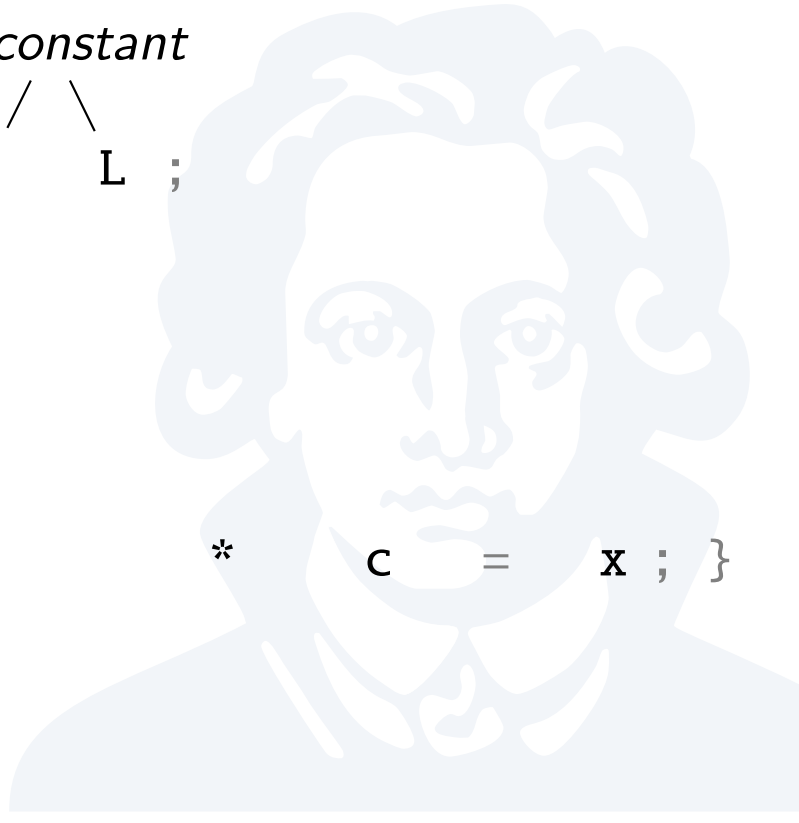


Der Übersetzer: Frontend: Parser

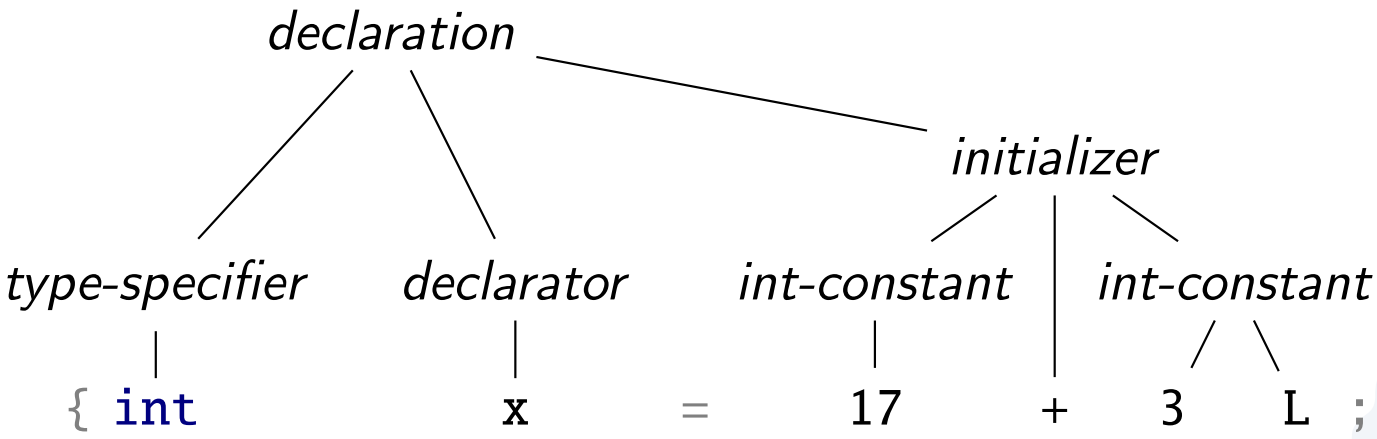



```
{
  int x = 17 + 3L;
  char *c = x;
}
```

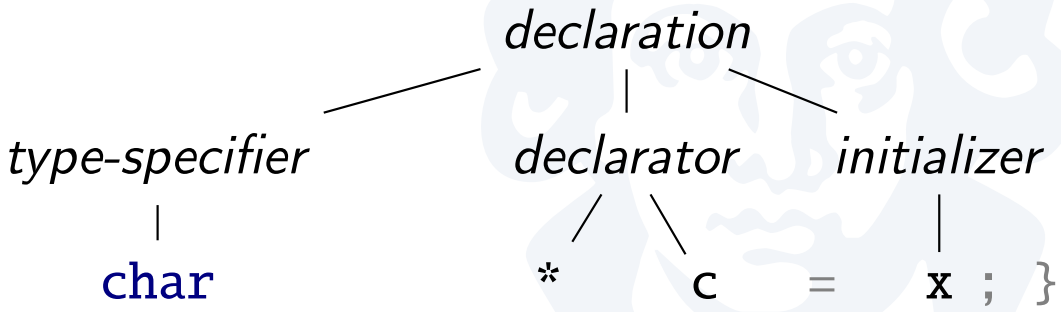
```
{ int x = 17 + 3 L ; char * c = x ; }
```



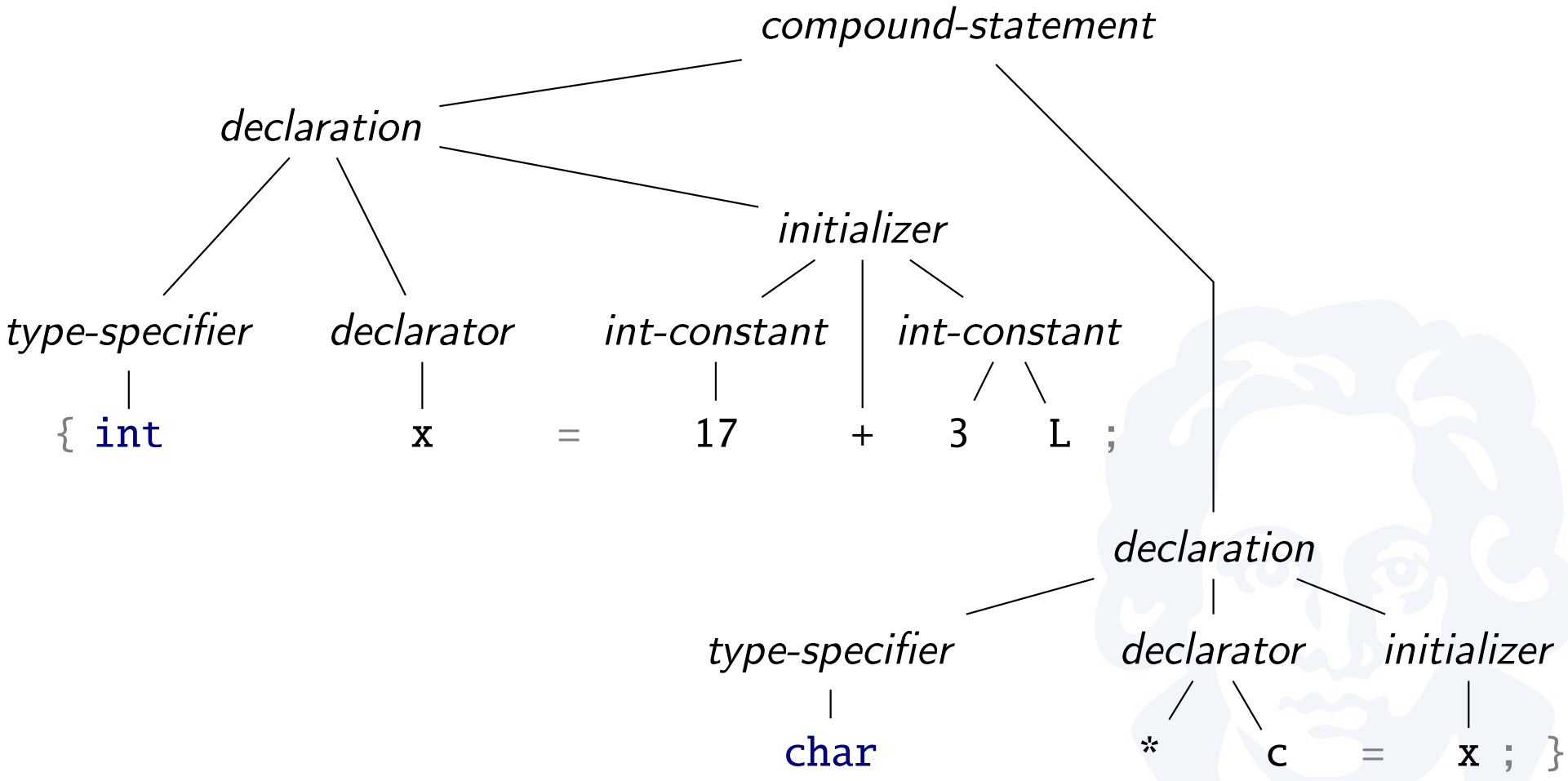
Der Übersetzer: Frontend: Parser



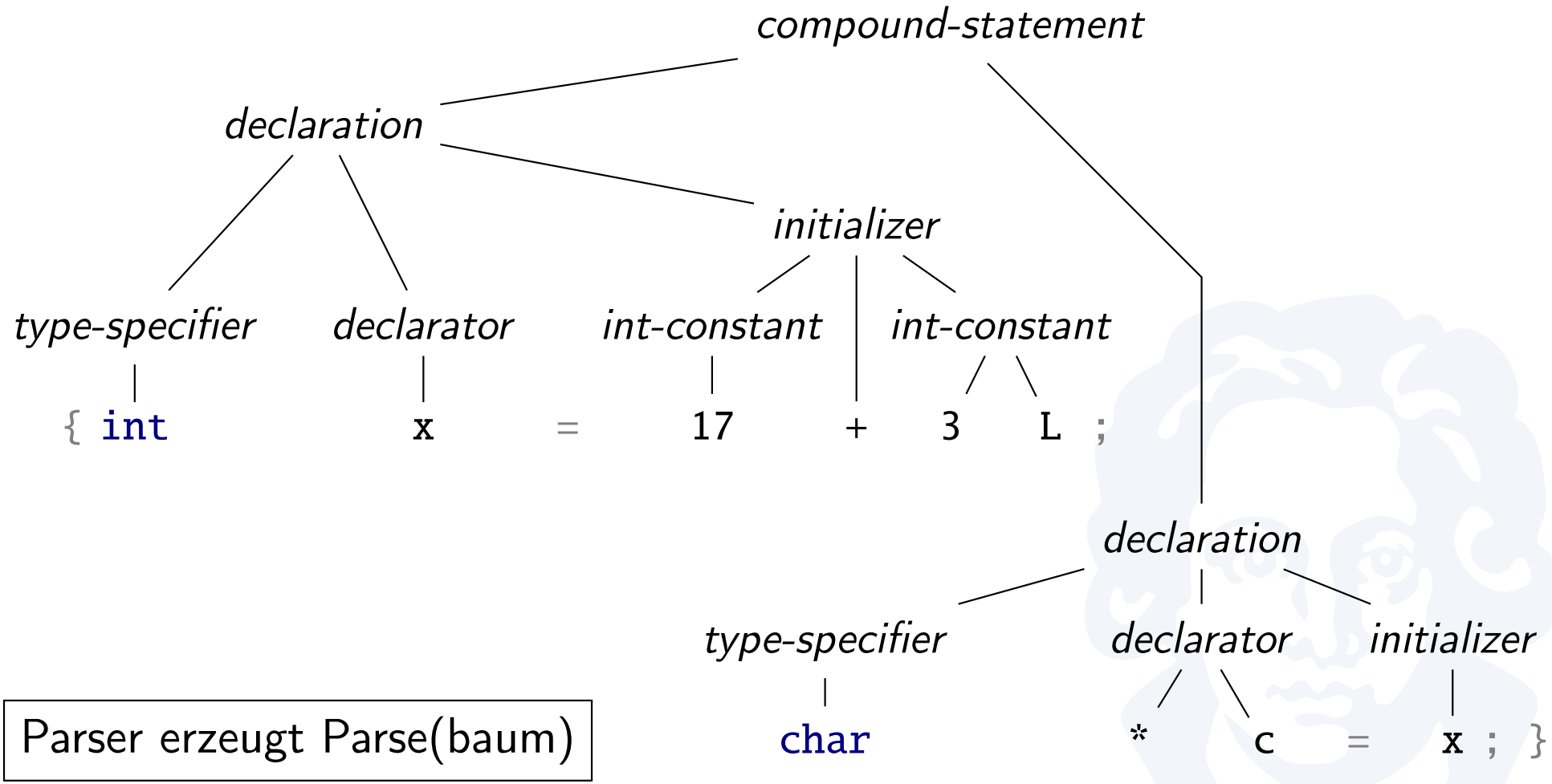

```
{
  int x = 17 + 3L;
  char *c = x;
}
```



Der Übersetzer: Frontend: Parser

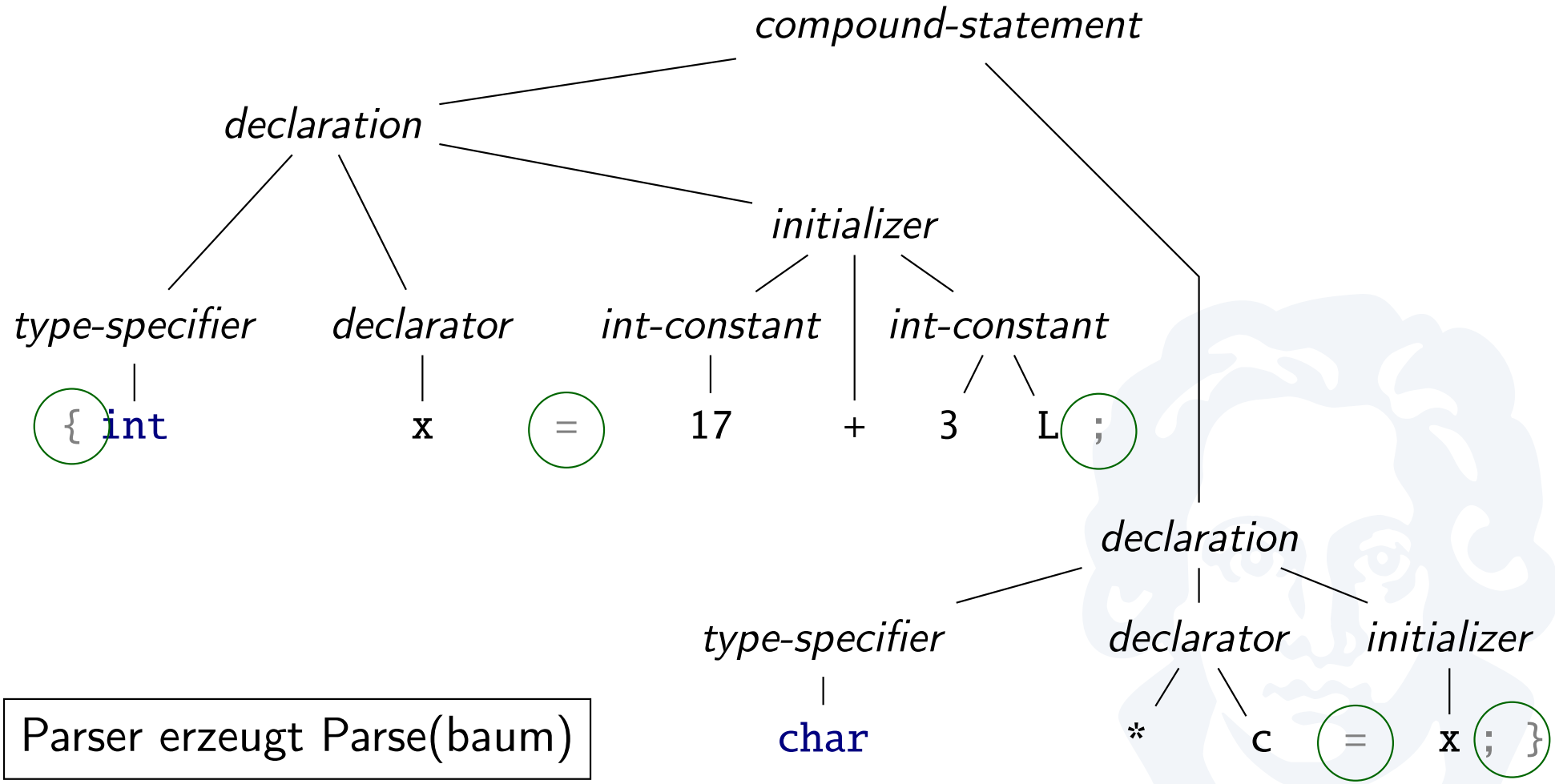


Der Übersetzer: Frontend: Parser



Parser erzeugt Parse(baum)

Der Übersetzer: Frontend: Parser

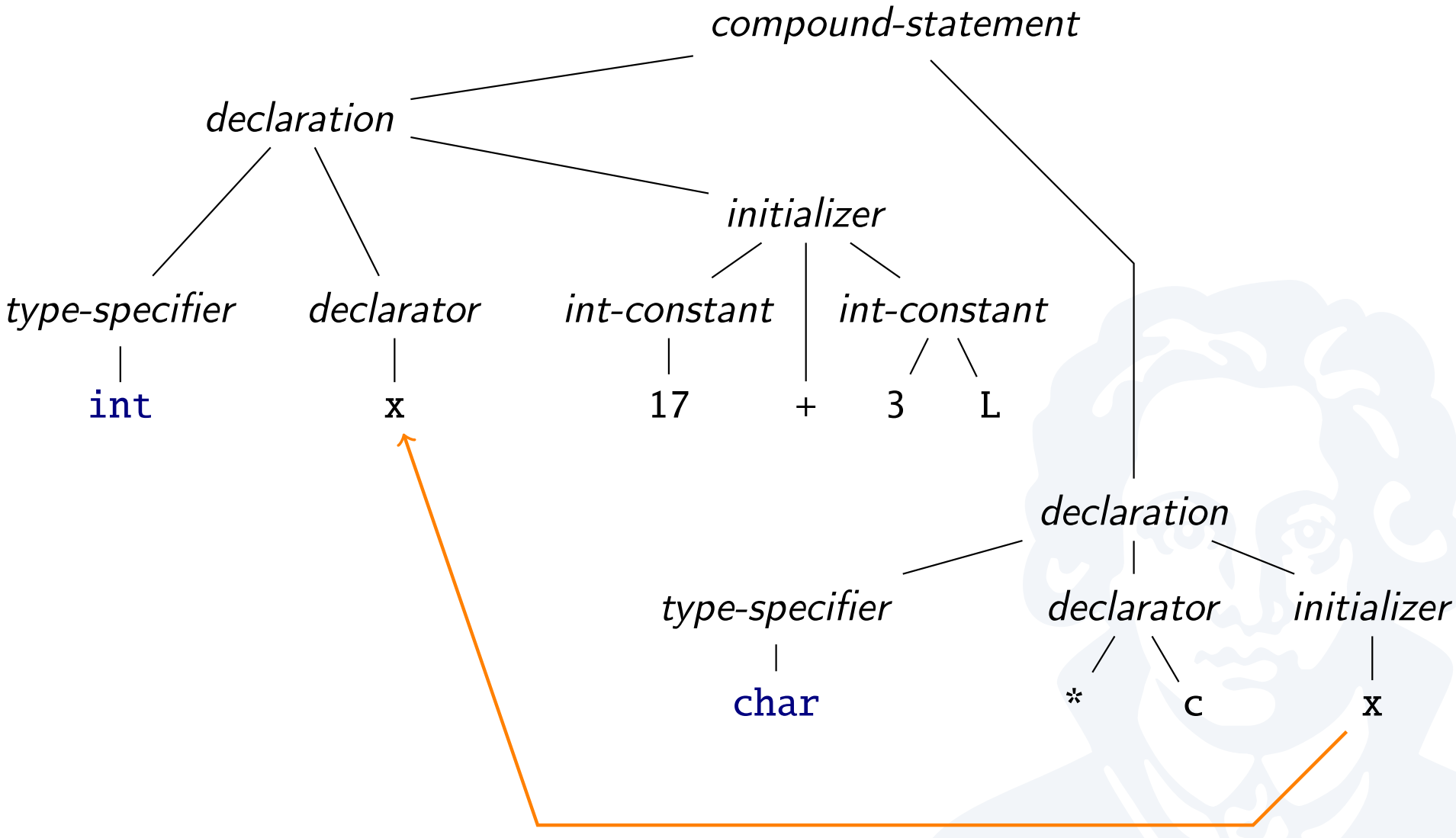


Parser erzeugt Parse(baum)

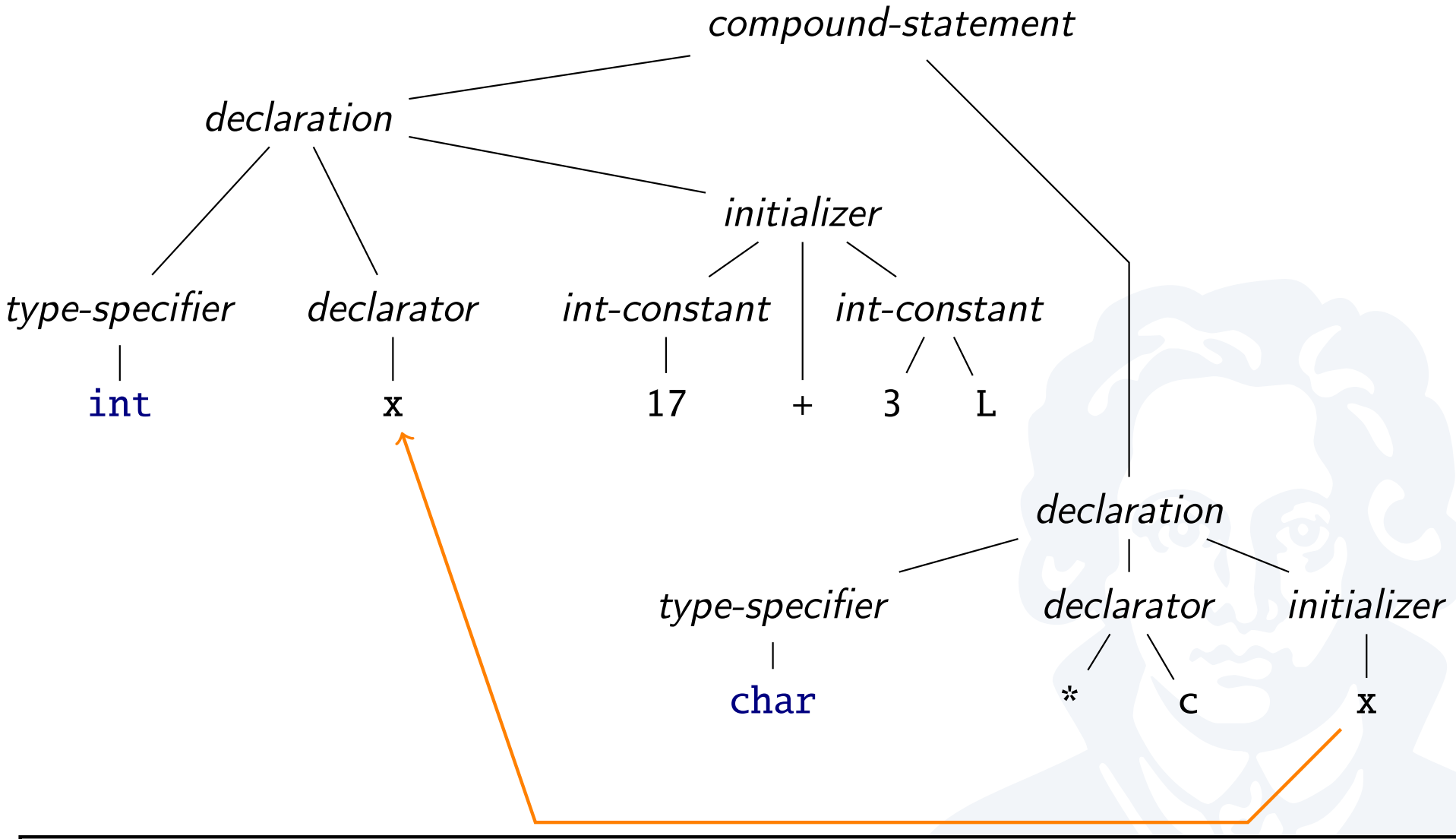
Entfernung unnötiger Details

Parse wird abstrahiert zu AST (*Abstract Syntax Tree*)

Frontend: Namensanalyse

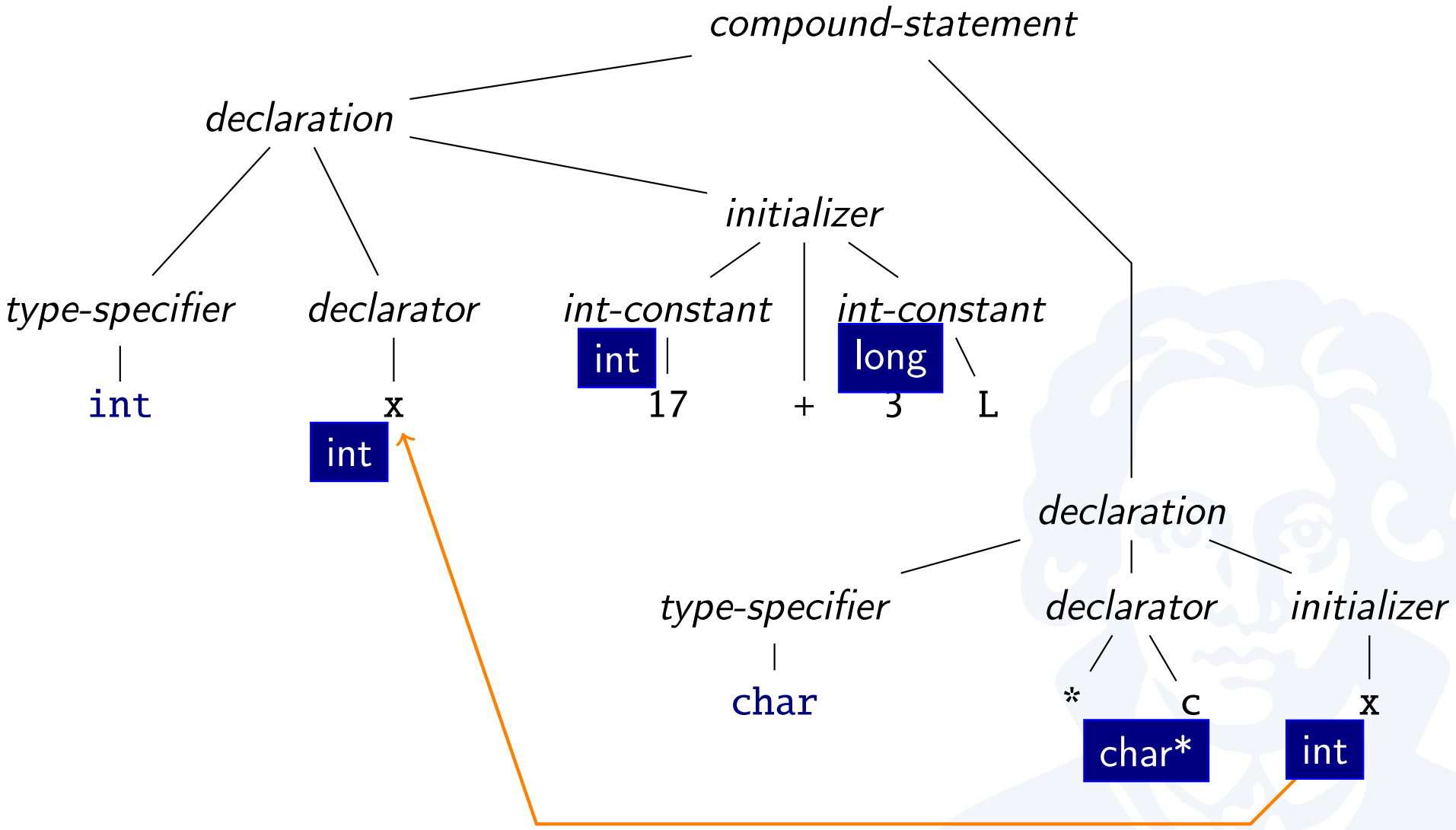


Frontend: Namensanalyse

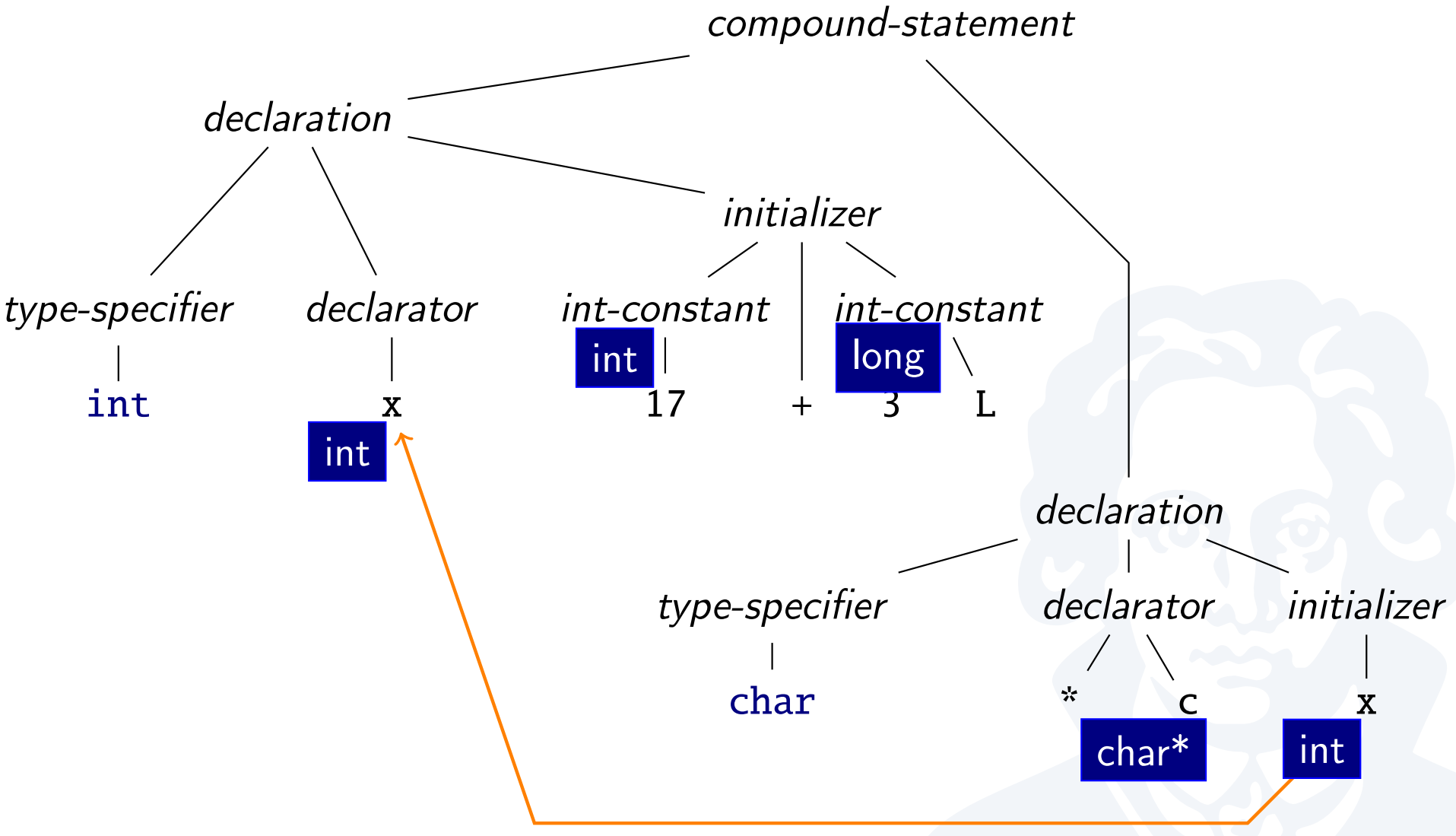


Verwendungen von Namen werden an ihre Definitionen gebunden

Frontend: Typanalyse (1/2)

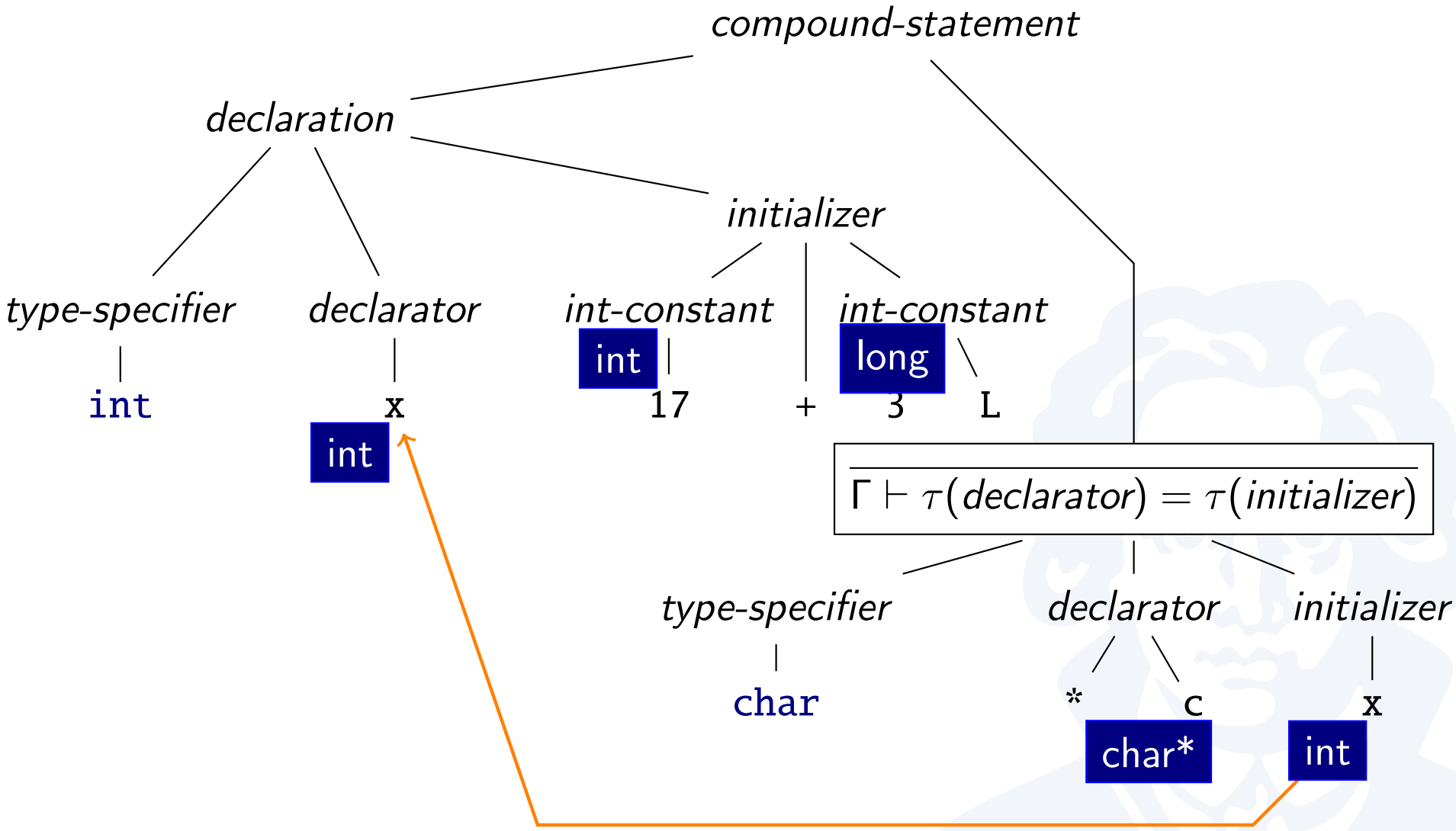


Frontend: Typanalyse (1/2)

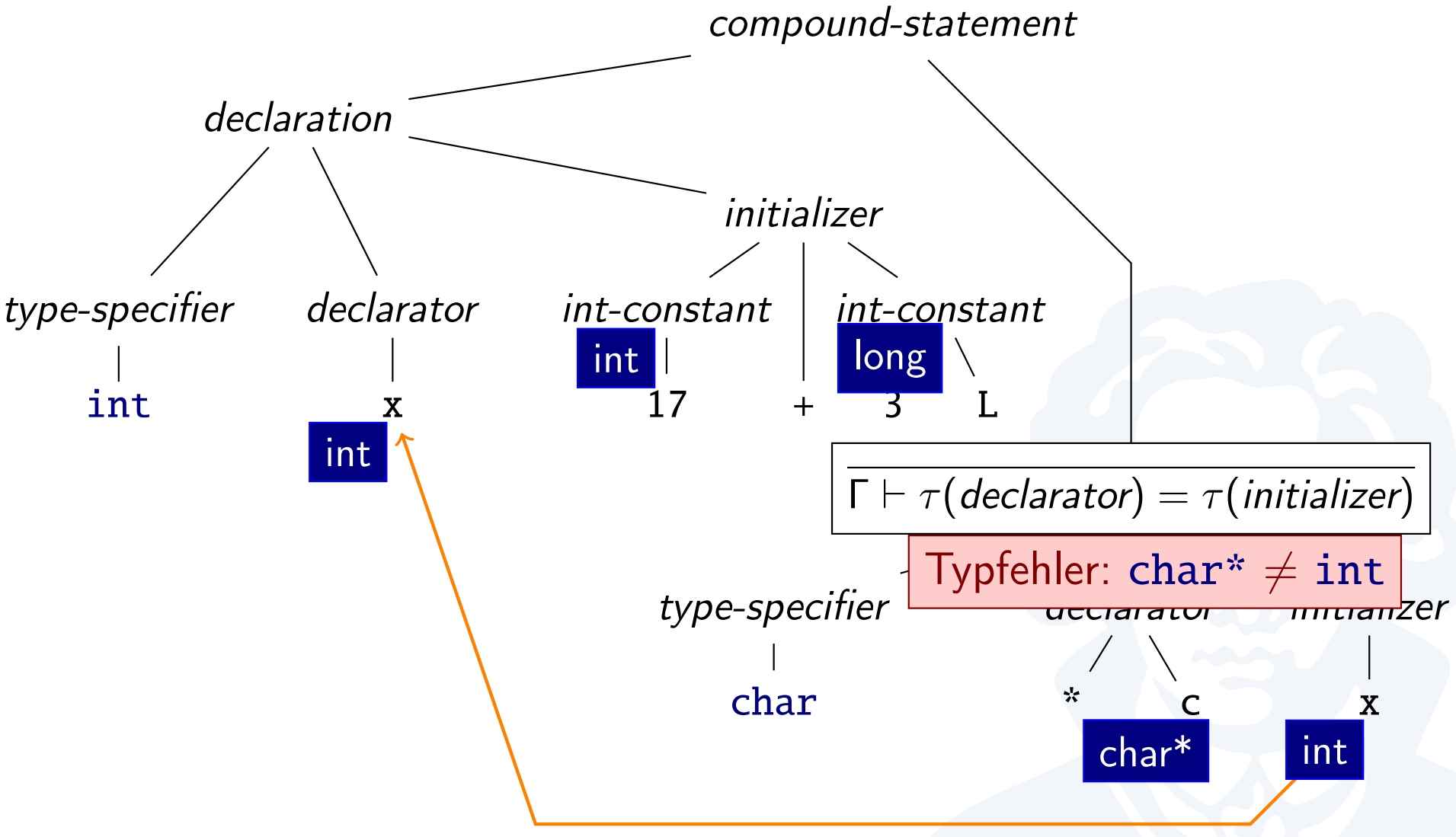


Typen werden an Namen gebunden

Frontend: Typanalyse (2/2)



Frontend: Typanalyse (2/2)



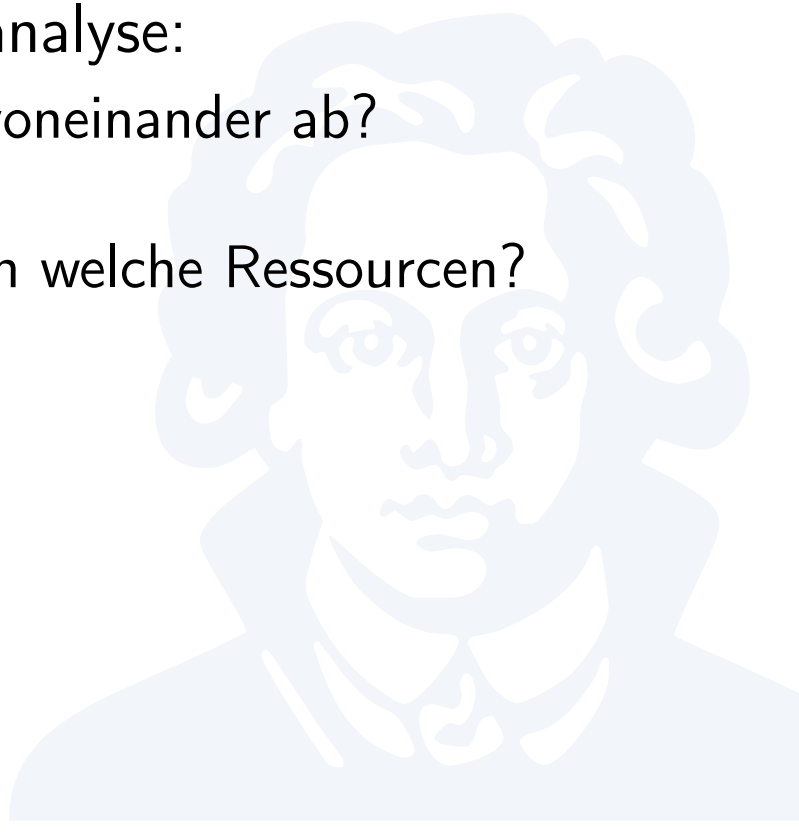
Typregeln erzwingen Typkorrektheit

Der Übersetzer: „Middle-End“

- Frontend erzeugt Zwischenform des Programmes

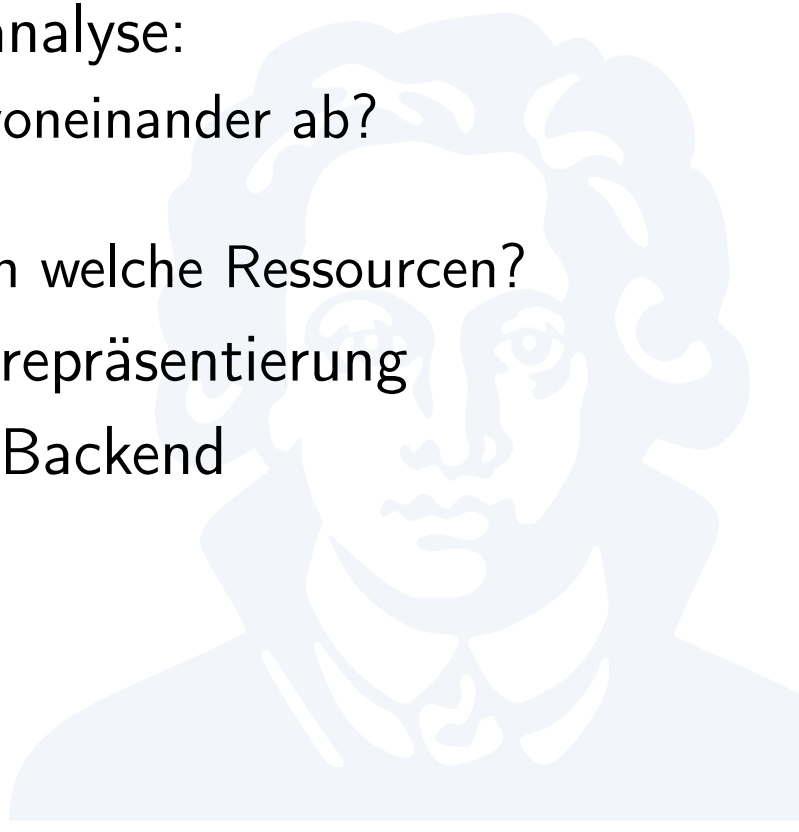
Der Übersetzer: „Middle-End“

- Frontend erzeugt Zwischenform des Programmes
- Zwischenform hilft bei Programmanalyse:
  - Welche Programmteile hängen voneinander ab? (Kontrollfluß/Daten)
  - Welche Programmteile benötigen welche Ressourcen?



Der Übersetzer: „Middle-End“

- Frontend erzeugt Zwischenform des Programmes
- Zwischenform hilft bei Programmanalyse:
  - Welche Programmteile hängen voneinander ab? (Kontrollfluß/Daten)
  - Welche Programmteile benötigen welche Ressourcen?
- „Middle-End“ optimiert Programmrepräsentierung
- Optimierte Zwischenform geht an Backend

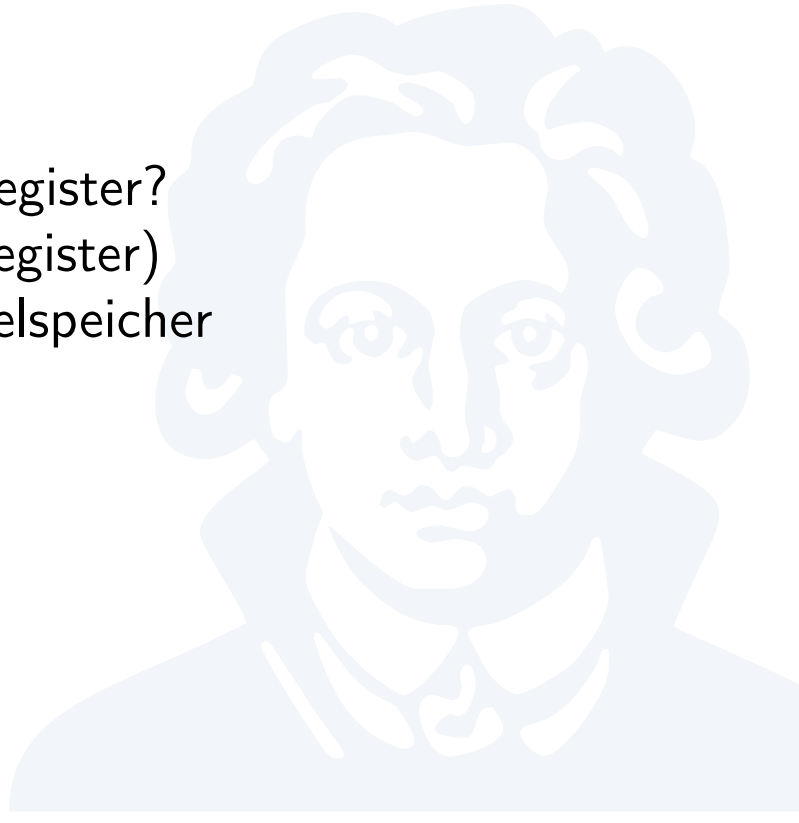


Der Übersetzer: Backend

- Backend erzeugt Assembler- oder Maschinencode
- Unterstützt oft verschiedene Prozessoren und Aufrufkonventionen
- Wesentliche Aufgaben:

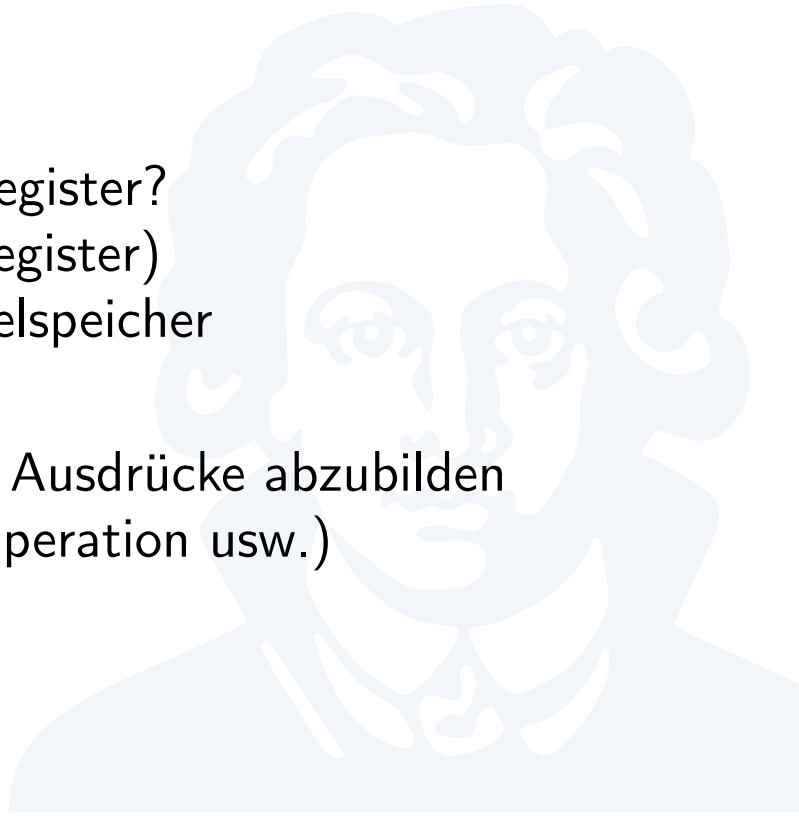
Der Übersetzer: Backend

- Backend erzeugt Assembler- oder Maschinencode
- Unterstützt oft verschiedene Prozessoren und Aufrufkonventionen
- Wesentliche Aufgaben:
  - Registerauswahl:
    - Welche Variablen in welche Register? (gesichert, temporär, Spezialregister)
    - Andere Variablen in den Stapelspeicher



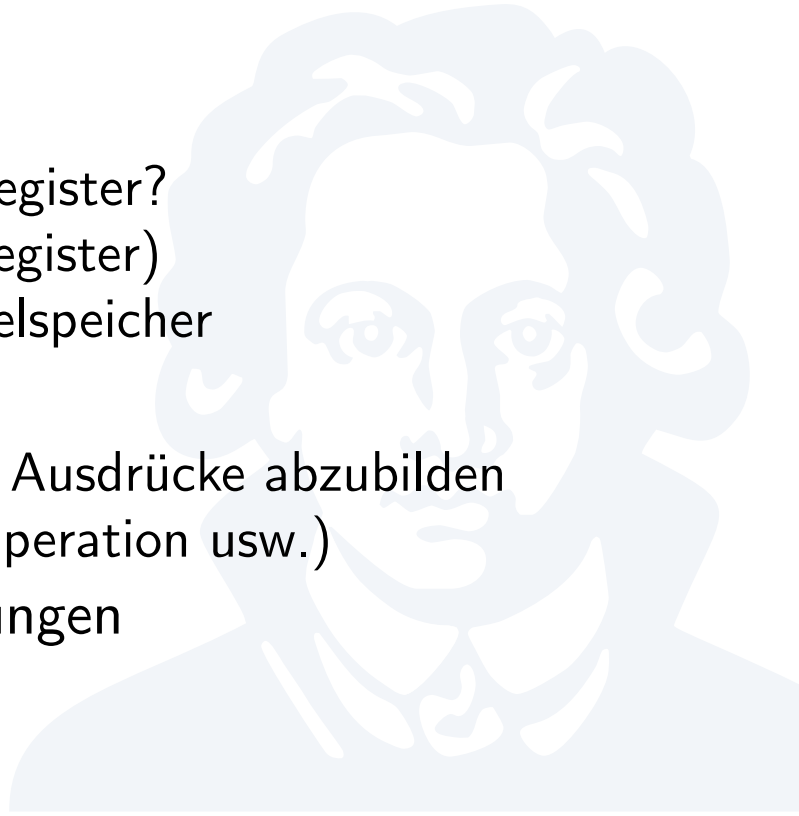
Der Übersetzer: Backend

- Backend erzeugt Assembler- oder Maschinencode
- Unterstützt oft verschiedene Prozessoren und Aufrufkonventionen
- Wesentliche Aufgaben:
  - Registerauswahl:
    - Welche Variablen in welche Register? (gesichert, temporär, Spezialregister)
    - Andere Variablen in den Stapelspeicher
  - Befehlsauswahl:
    - Effiziente Befehle suchen, um Ausdrücke abzubilden (Direkt-Operation, Register-Operation usw.)



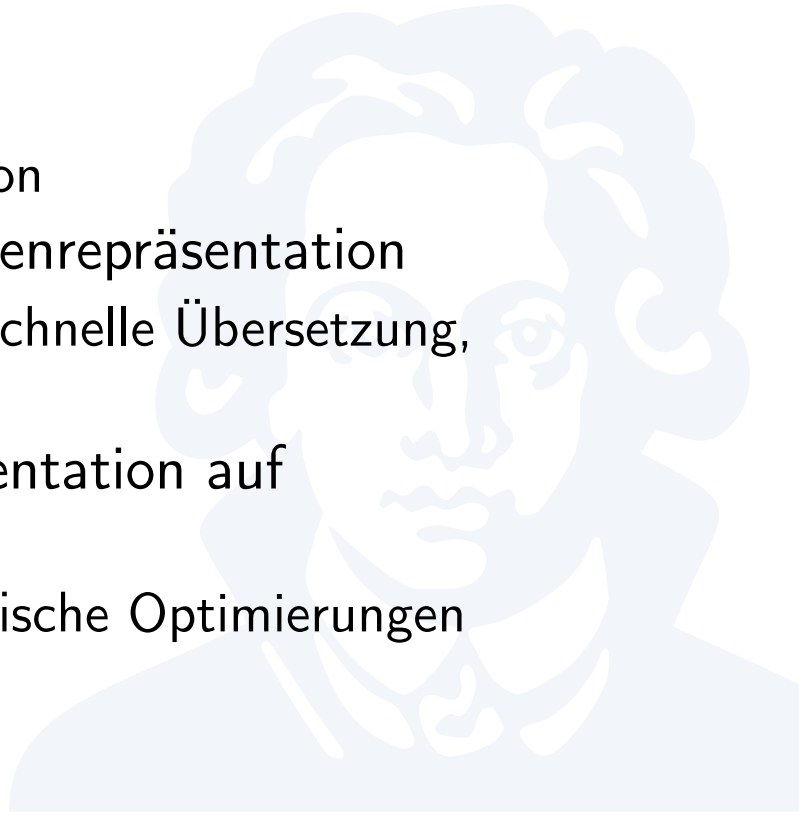
Der Übersetzer: Backend

- Backend erzeugt Assembler- oder Maschinencode
- Unterstützt oft verschiedene Prozessoren und Aufrufkonventionen
- Wesentliche Aufgaben:
  - Registerauswahl:
    - Welche Variablen in welche Register? (gesichert, temporär, Spezialregister)
    - Andere Variablen in den Stapelspeicher
  - Befehlsauswahl:
    - Effiziente Befehle suchen, um Ausdrücke abzubilden (Direkt-Operation, Register-Operation usw.)
  - Architekturspezifische Optimierungen
    - Vektoroperationen



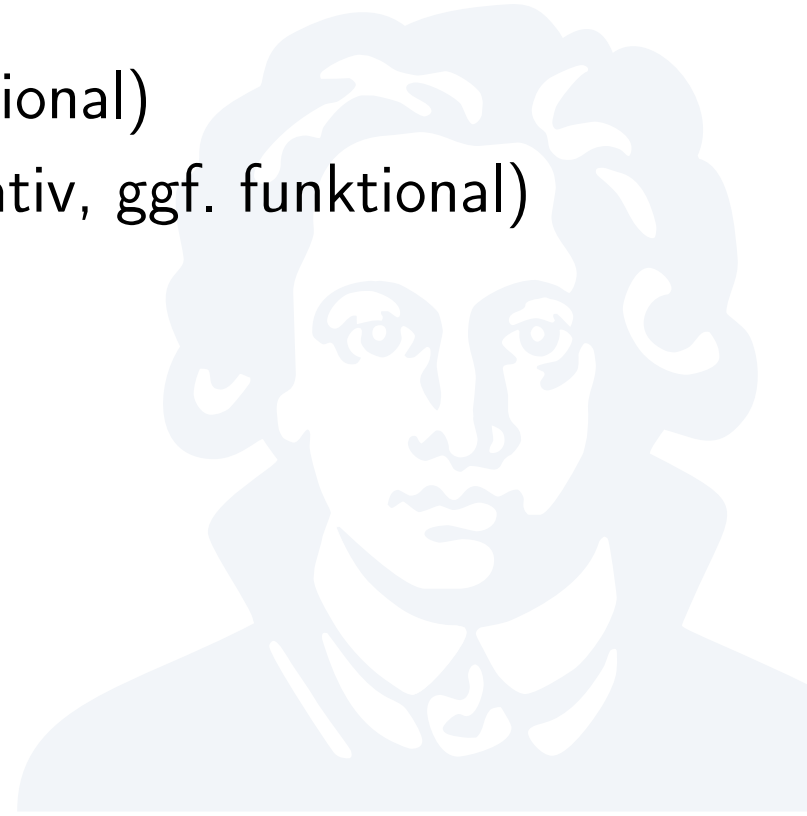
Zusammenfassung: Der Übersetzer

- Drei Übersetzerphasen:
  - **Frontend** liest Programm ein
    - Lexikalische Analyse
    - Parser: Parse-Baum $\rightarrow$ AST
    - Typanalyse und Fehlersuche
    - Erzeugt Zwischenrepräsentation
  - **„Middle-End“** optimiert Zwischenrepräsentation
    - Kann übersprungen werden (schnelle Übersetzung, langsamer Code)
  - **Backend** bildet Zwischenrepräsentation auf Assembler/Maschinencode ab
    - Evtl. kleinere maschinenspezifische Optimierungen



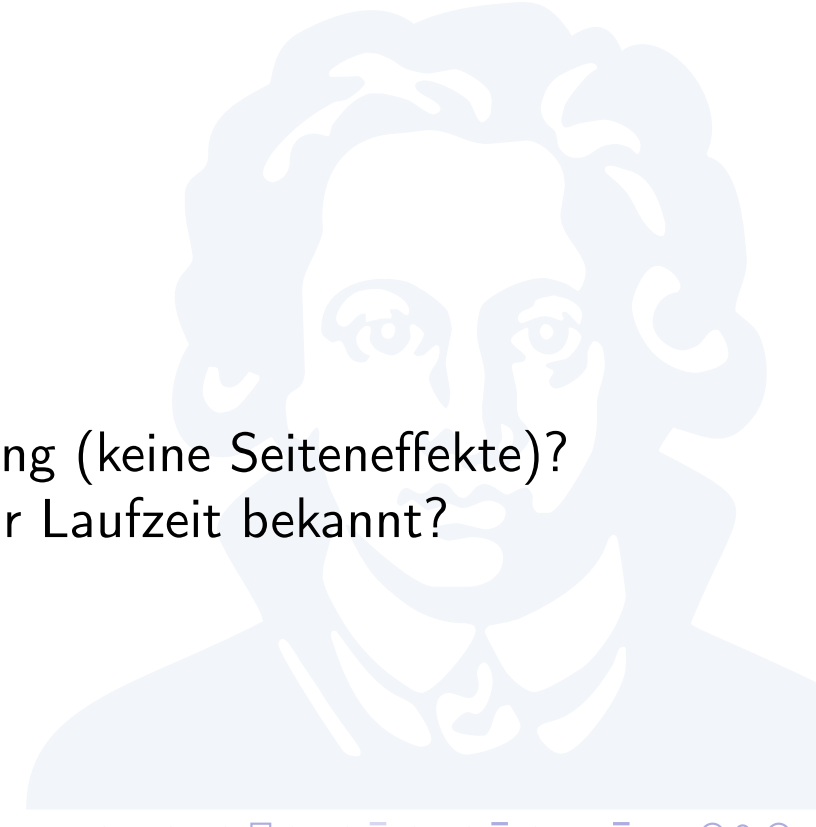
Programmanalyse

- Analyse durch Übersetzung (funktional)
- Analyse durch Annotation (imperativ, ggf. funktional)



Programmanalyse durch Annotation

- AST-Knoten halten Attribute, z.B.:
  - Für Variablen (`v`):
    - Wo deklariert?
    - Typ?
    - Wo wird zugewiesen?
    - Wo wird gelesen?
  - Für Ausdrücke (`1 + 2 + 3`):
    - Typ?
    - Ist hier eine „reine“ Berechnung (keine Seiteneffekte)?
    - Berechnungsergebnis, falls zur Laufzeit bekannt?



Beliebte Programmanalyseverfahren

- Flußanalyse
- Typanalyse
- Abstrakte Interpretierung

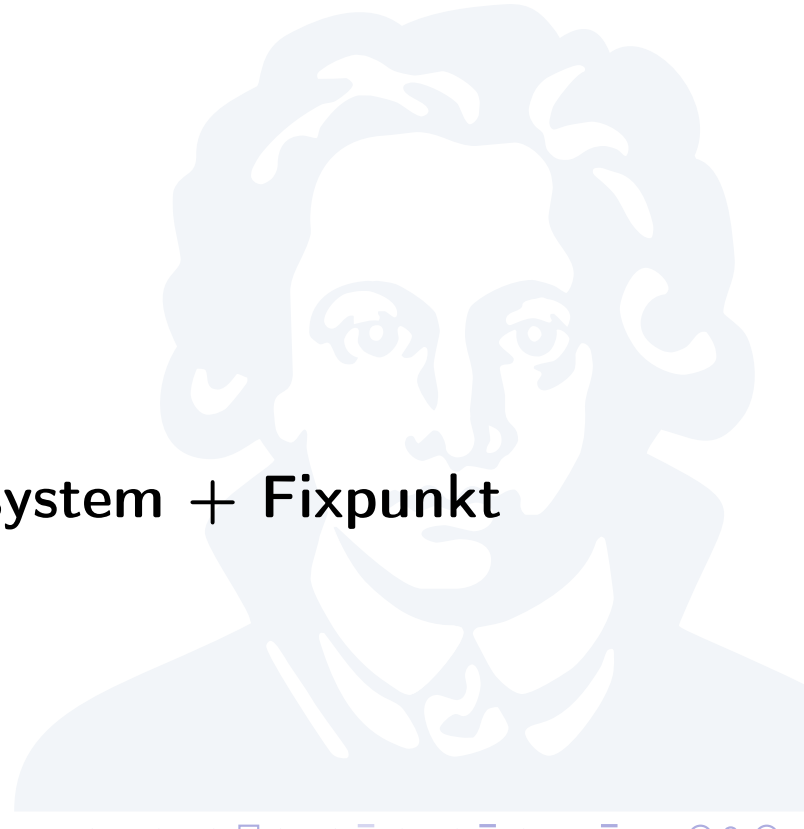
Flußanalyse

Fragestellung z.B.: welche Variablen haben konstante Werte?

```
x := 3;
y := 2;
while (y > 2) {
  y := y - 1;
  z := x;
}
```

(s. Tafel)

Basis: Monotones Ungleichungssystem + Fixpunkt



Typanalyse

Fragestellung z.B.: Wo weden Seiteneffekte ausgelöst?

- Effektregel: $\dfrac{a : \tau \xrightarrow{\eta} \sigma \quad b : \tau \quad \text{Effekt } \eta}{a(b) : \sigma}$ funapp
- Typwissen:
  - *inc* : int $\to$ int
  - *print* : int $\xrightarrow{\text{wr}}$ int

let f = if x > 0 then print else inc
in let v = inc(3)
in f(v)

Basis: Termunifikation

Typanalyse

Fragestellung z.B.: Wo weden Seiteneffekte ausgelöst?

- Effektregel: $\dfrac{a : \tau \xrightarrow{\eta} \sigma \quad b : \tau \quad \text{Effekt } \eta}{a(b) : \sigma}$ funapp
- Typwissen:
  - *inc* : int $\to$ int
  - *print* : int $\xrightarrow{\text{wr}}$ int

let f = if x > 0 then print else inc f : int $\xrightarrow{\text{wr}}$ int
in let v = inc(3)
in f(v)

Basis: Termunifikation

Typanalyse

Fragestellung z.B.: Wo weden Seiteneffekte ausgelöst?

- Effektregel: $\dfrac{a : \tau \xrightarrow{\eta} \sigma \quad b : \tau \quad \text{Effekt } \eta}{a(b) : \sigma}$ funapp
- Typwissen:
  - *inc* : int $\to$ int
  - *print* : int $\xrightarrow{\text{wr}}$ int

let f = if x > 0 then print else inc
in let v = inc(3)
in f(v)

f : int $\xrightarrow{\text{wr}}$ int
v : int (kein Effekt)

Basis: Termunifikation

Typanalyse

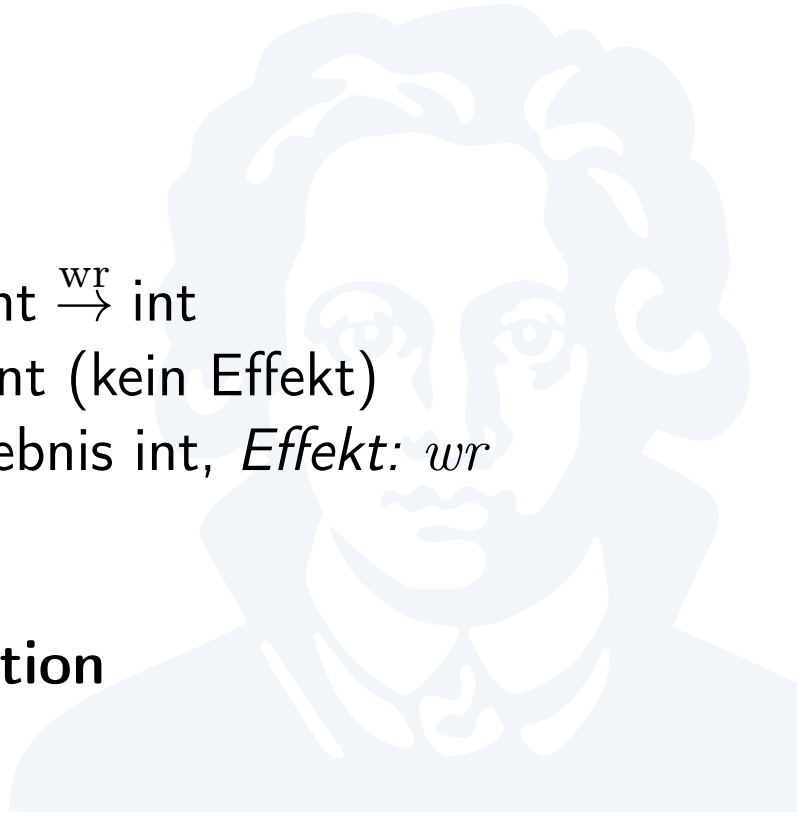
Fragestellung z.B.: Wo weden Seiteneffekte ausgelöst?

- Effektregel: $\dfrac{a : \tau \xrightarrow{\eta} \sigma \quad b : \tau \quad \text{Effekt } \eta}{a(b) : \sigma}$ funapp
- Typwissen:
  - *inc* : int $\to$ int
  - *print* : int $\xrightarrow{\text{wr}}$ int

let f = if x > 0 then print else inc
in let v = inc(3)
in f(v)

f : int $\xrightarrow{\text{wr}}$ int
v : int (kein Effekt)
Ergebnis int, *Effekt:* wr

Basis: Termunifikation



Typanalyse

Fragestellung z.B.: Wo weden Seiteneffekte ausgelöst?

- Effektregel: $\dfrac{a : \tau \xrightarrow{\eta} \sigma \quad b : \tau \quad \text{Effekt } \eta}{a(b) : \sigma}$ funapp
- Typwissen:
  - *inc* : int $\rightarrow$ int
  - *print* : int $\xrightarrow{\text{wr}}$ int

let f = if x > 0 then print else inc
in let v = inc(3)
in f(v)

f : int $\xrightarrow{\text{wr}}$ int
v : int (kein Effekt)
Ergebnis int, *Effekt:* wr

Kombinierte Typ/Datenflußanalyse

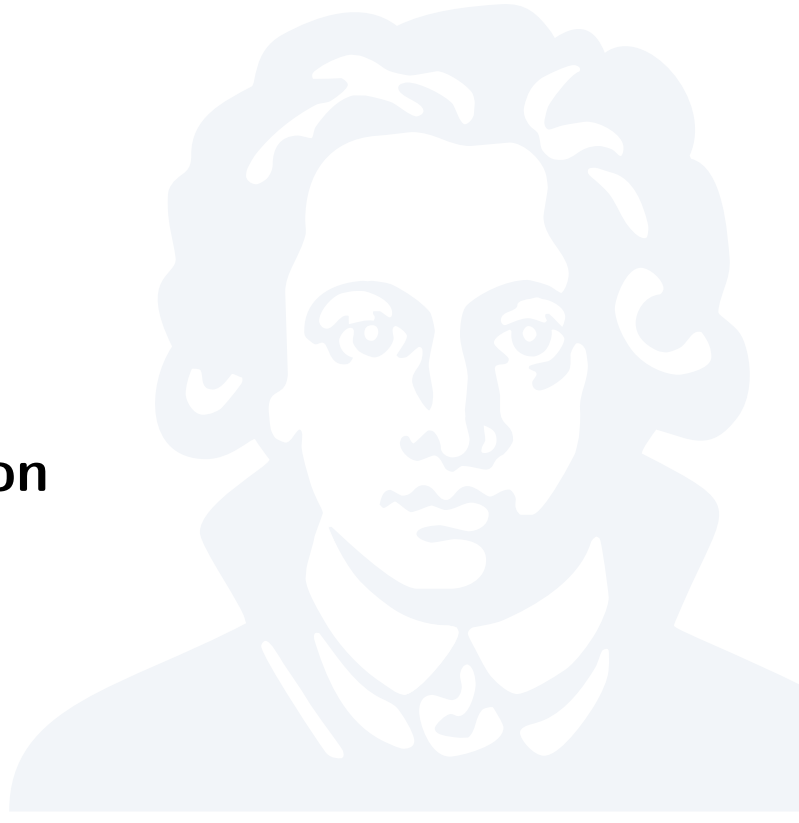
Basis: Termunifikation

Abstrakte Interpretierung

Fragestellung z.B.: Was ist das Vorzeichen der Variableninhalte?

Konkret	**Abstrakt**
`x := 42;`	
`y := -3;`	
`a := x * y;`	
`b := x - y;`	
`c := x + y;`	

Basis: Abstraktion

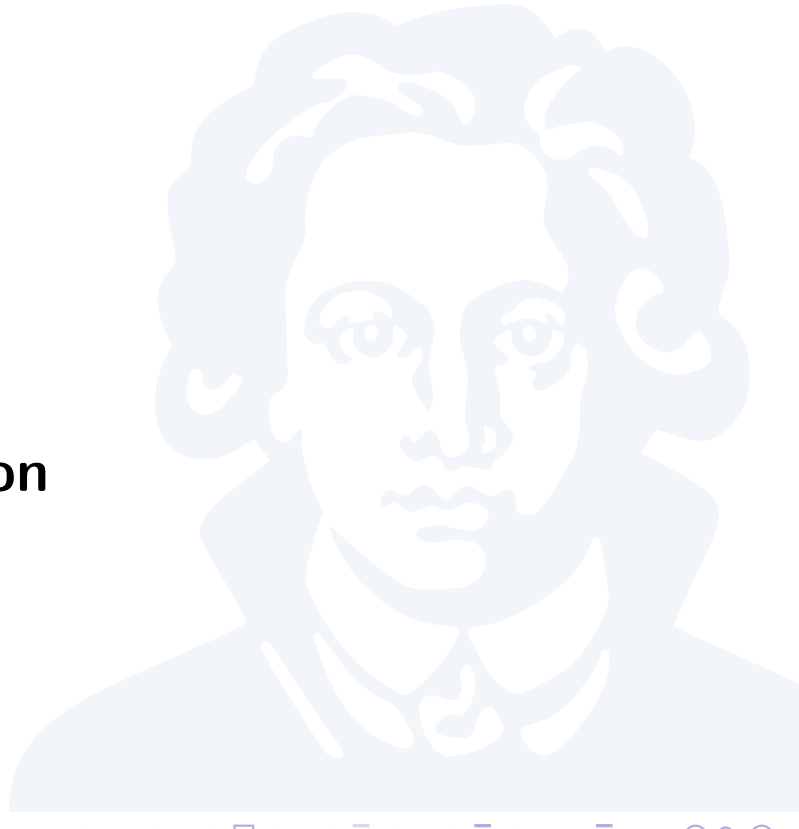


Abstrakte Interpretierung

Fragestellung z.B.: Was ist das Vorzeichen der Variableninhalte?

Konkret	**Abstrakt**
`x := 42;`	$x = +$
`y := -3;`	$y = -$
`a := x * y;`	
`b := x - y;`	
`c := x + y;`	

Basis: Abstraktion

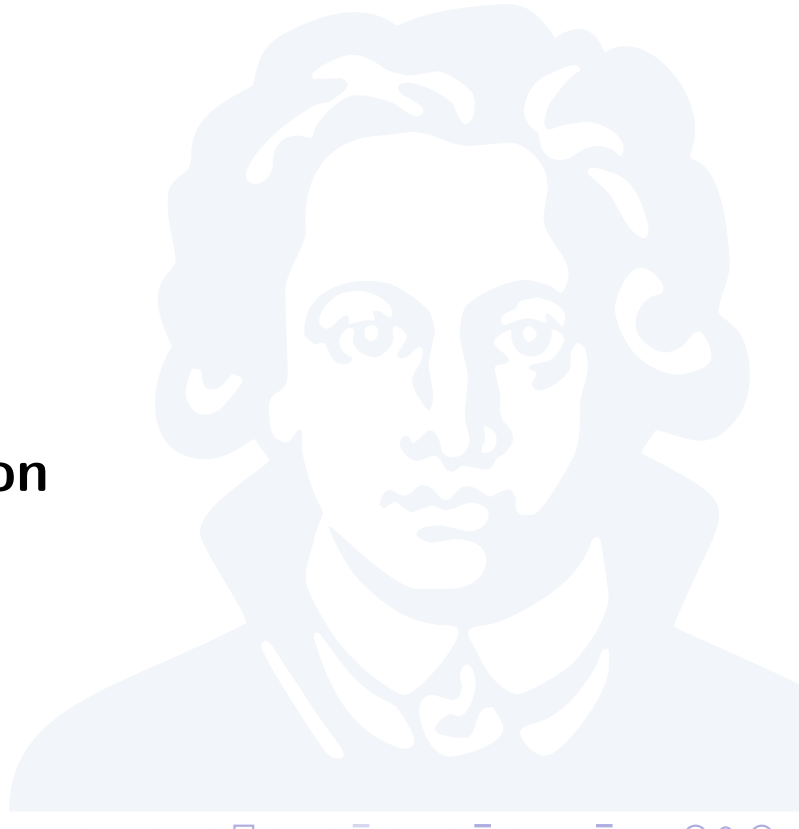


Abstrakte Interpretierung

Fragestellung z.B.: Was ist das Vorzeichen der Variableninhalte?

Konkret	**Abstrakt**
`x := 42;`	$x = +$
`y := -3;`	$y = -$
`a := x * y;`	$a = x \otimes y = -$
`b := x - y;`	
`c := x + y;`	

Basis: Abstraktion

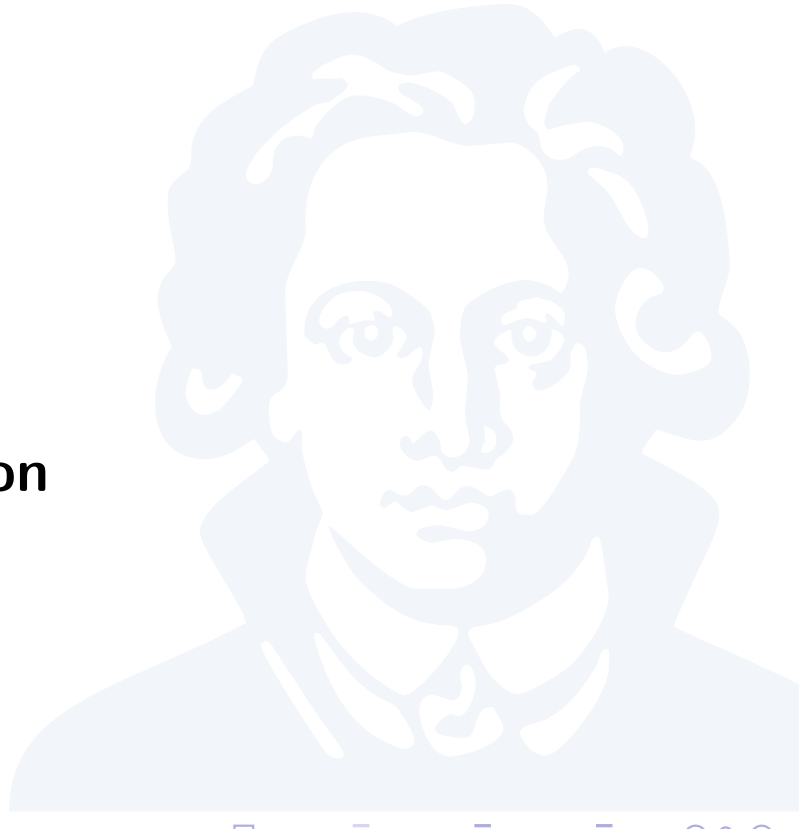


Abstrakte Interpretierung

Fragestellung z.B.: Was ist das Vorzeichen der Variableninhalte?

Konkret	**Abstrakt**
`x := 42;`	$x = +$
`y := -3;`	$y = -$
`a := x * y;`	$a = x \otimes y = -$
`b := x - y;`	$a = x \ominus y = +$
`c := x + y;`	

Basis: Abstraktion

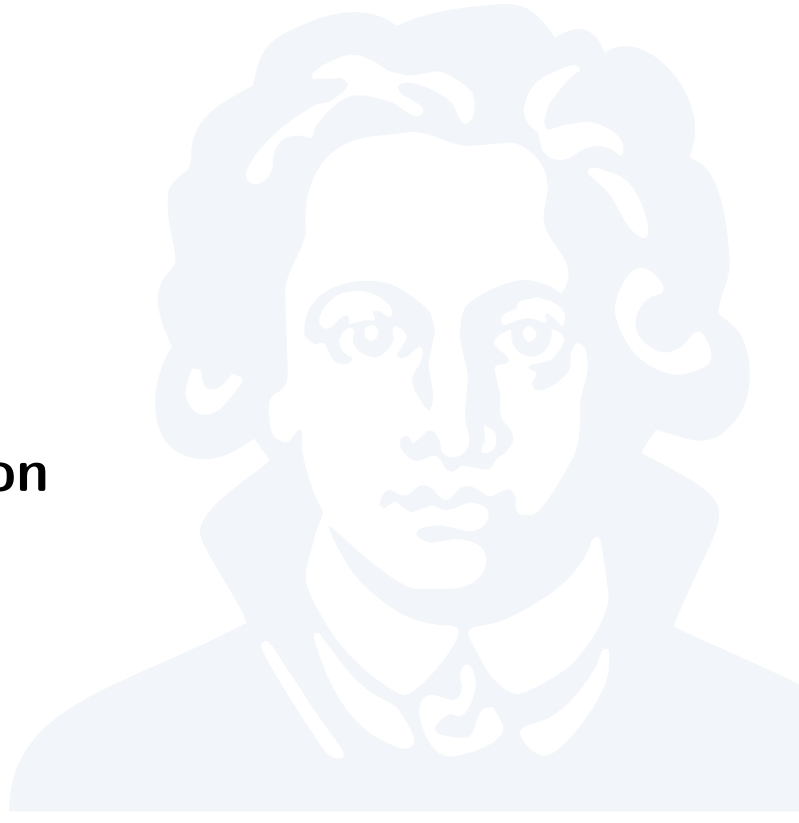


Abstrakte Interpretierung

Fragestellung z.B.: Was ist das Vorzeichen der Variableninhalte?

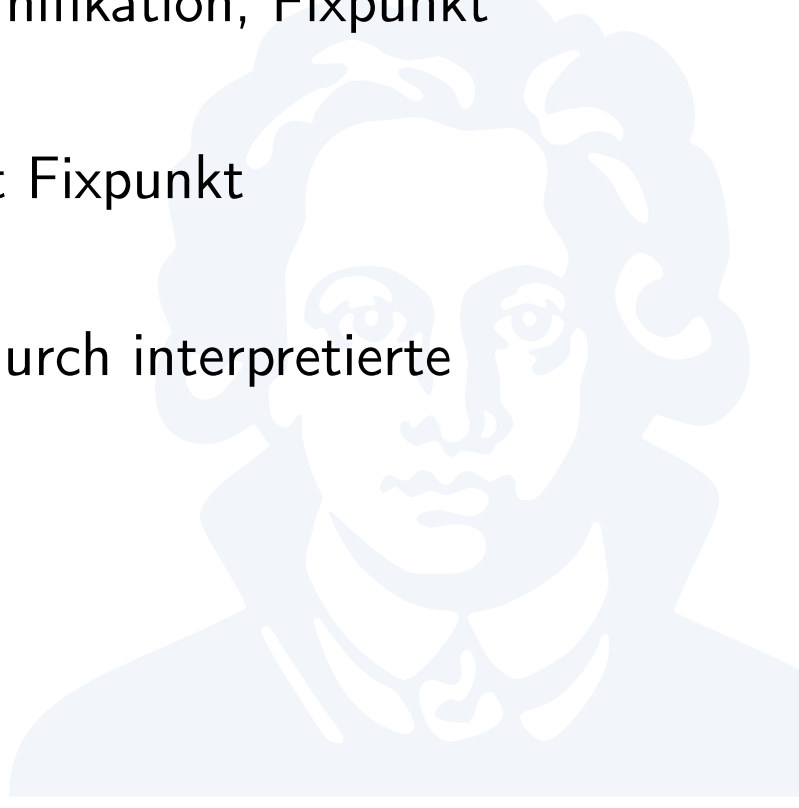
Konkret	**Abstrakt**
`x := 42;`	$x = +$
`y := -3;`	$y = -$
`a := x * y;`	$a = x \otimes y = -$
`b := x - y;`	$a = x \ominus y = +$
`c := x + y;`	$a = x \oplus y = \overset{+}{-}$

Basis: Abstraktion



Beliebte Programmanalyseverfahren

- Typanalyse
  - Gleichungssystem: Lösung mit Unifikation, Fixpunkt
- Flußanalyse
  - Ungleichungssystem: Lösung mit Fixpunkt
- Abstrakte Interpretierung
  - Abstraktes Programm: Lösung durch interpretierte Ausführung



Teil IV

Beispielvortrag