

Sprachtechnologie und Compiler

Prof. Dr.-Ing. Gregor Snelting
snelting@ipd.info.uni-karlsruhe.de

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe (TH)

Wintersemester 2008/09

Folien basierend auf dem Skript von
Prof. em. Dr. Dr. h.c. Gerhard Goos



Kapitel 1

Einleitung



Organisatorisches

Vorlesung

- Die Veranstaltung ist prüfbar (bis zu 4SWS)
- Homepage:
`http://pp.info.uni-karlsruhe.de/lehre/WS200809/compiler`
- **Achtung:** Am 18. + 19.11. ausnahmsweise in SR 301
- Kontakt: `snelting@ipd.info.uni-karlsruhe.de`

Übung

- Mittwochs findet um 14 Uhr eine Übung in HS-101 statt.
- **Achtung:** Am 19.11. ausnahmsweise in SR-120
- Kontakt: `braun@ipd.info.uni-karlsruhe.de`



Organisatorisches - weiterführende Veranstaltungen

Im Sommersemester bieten wir vertiefende Veranstaltungen an:

Compilerpraktikum

- Entwurf und vor allem Implementierung eines Compilers für (Mini-)Java in Kleingruppen
- lexikalische, syntaktische und semantische Analyse, einfache Optimierungen, Codeerzeugung
- 2 SWS in Prüfung anrechenbar

Compiler II

- Themen: Fortgeschrittene Programmanalysetechniken, Registerzuteilung, Cache-Optimierungen, Schleifentransformationen
- 2 SWS, prüfbar



Studien- und Diplomarbeiten

- Der Lehrstuhl bietet Studien- und Diplomarbeiten im Bereich Compiler Construction, Programmanalyse / Programmslicing, sowie maschinelle Verifikation an.
- Wir suchen Hiwis zur Mitarbeit im libFirm (<http://www.libfirm.org>) Projekt, einer modernen graphbasierten Compilerzweischensprache.



1 Motivation und Literatur

2 Übersetzungsarten

3 Architektur/Phasen

- Architektur
- Analyse
- Transformation
- Codierung
- Datenstrukturen



Warum ist Sprachtechnologie interessant?

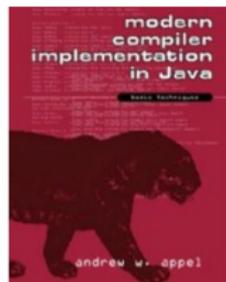
- Querschnittstechnologie mit breitem Anwendungsbereich in der Softwaretechnik
- ältestes Gebiet der praktischen Informatik
- Modell für erfolgreiche theoretische Fundierung praktischer Entwicklungen
- stabile Software-Architektur: Musterbeispiel für software engineering bei mittelgroßen sequentiellen Software-Systemen
- hohe Korrektheits- und Zuverlässigkeitsforderungen



Warum ist Sprachtechnologie interessant?

- grundlegend für Weiterentwicklungen der Programmiermethodik
- grundlegend für Weiterentwicklung der Prozessor-/Software-Schnittstelle
- liefert die Standardmethoden für die Verarbeitung textueller Eingaben und zur Generierung von Programmen aus Spezifikationen
- viele Anwendungen im Software Engineering:
Textformatierung, Programmanalyse,
Programmtransformationen, automatische Anwendung von Entwurfsmustern, Metaprogrammierung, Verarbeitung von XML, Refaktorisierung, Software-Sicherheitsanalyse, Metriken,
...





Andrew W. Appel, Jens Palsberg
Modern Compiler Implementation in Java
Cambridge University Press
ISBN: 052182060X



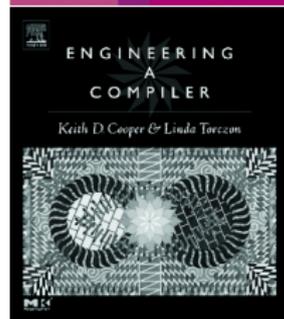
Alfred V. Aho, Monica S. Lam, Ravi Sethi
Compiler. Prinzipien, Techniken und Tools
PEARSON STUDIUM
ISBN: 3827370973



Weiterführende Literatur



Reinhard Wilhelm, Helmut Seidl
Übersetzerbau
Springer
ISBN: 9783540495963



Keith Cooper, Linda Torczon
Engineering a Compiler
Morgan Kaufmann
ISBN: 978-1-55860-699-9



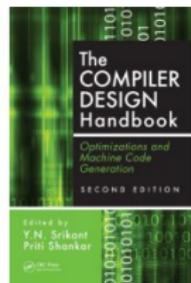
Weiterführende Literatur



Robert Morgan
Building an Optimizing Compiler
Digital Press
ISBN: 155558179X

Steven Muchnick
Advanced Compiler Design and Implementation
Morgan Kaufmann
ISBN: 978-1-55860-320-2





Y. N. Srikant, Priti Shankar
**The Compiler Design Handbook:
Optimizations and Machine Code
Generation**
CRC Press Inc
ISBN: 142004382X

Anwendungsfelder

- Text → Information
 - Textanalyse (Beispiel: \LaTeX , Word, Konfigurationsdateien)
 - Programmanalyse (Beispiel: Software-Sanierung)
- Text → Text
 - Quell-Quell-Transformation (Beispiel: Präprozessor, UML nach C)
- Programm → Stackcode
 - Beispiel: Java Byte Code, .NET CLI
- Programm → Maschinencode
 - Beispiel: Alle C-Übersetzer



Übersetzen

Text aus *Quellsprache* in *Zielsprache* übertragen **unter Erhaltung der Bedeutung** (bedeutungsäquivalent, semantikerhaltend)

- Quell-/Zielsprache können identisch sein
- Ziel kann maschinen-interpretierbare Sprache sein.
- Bei Programmiersprachen: Semantik durch Ausführung - **Interpretation** - des Quell-/Zieltexts gegeben



Zielkriterien

- Korrektheit
- Minimaler Betriebsmittelaufwand zur Laufzeit (Rechenzeit, Speicher, Energie)
- Kompatibilität mit anderen Übersetzern
 - gleicher Sprachumfang akzeptiert
 - Interaktion mit Programmen
 - anderer Sprachen
 - anderer Übersetzer
 - auf anderen Rechnern
 - dynamische Verknüpfung
- Übersetzungsgeschwindigkeit
- *Diese Ziele widersprechen sich teilweise!*



Bedeutung (Semantik)

definiert durch Komposition:

- Programmtext hat Struktur (Syntax)
- Bedeutung den Strukturelementen zugeordnet
- Gesamtbedeutung durch Komposition

Strukturelemente imperativer Sprachen

- Datentypen, Objekte (Variable, benannte Konstante, Literale) und Operationen
- ablaufsteuernde Elemente
- statische Strukturelemente (definieren Gültigkeitsbereiche)

also:

- Programmtext hat keine Bedeutung als Text
- Bedeutung erst nach Identifikation der Strukturelemente erfaßbar



Spezifikation von Übersetzungen

Aufzählung:

- Gib zu jedem Quellprogramm ein (oder mehrere) Zielprogramme an
- **Aber:** Quellsprachen erlauben i.A. abzählbar unendlich viele Elemente (Programme)

Intensionale Definition:

- Kompositionalität der Sprachen nutzen
- Sprachelemente definieren Bedeutung
- Übersetzungsdefinition für alle Sprachelemente definiert
- Übersetzung unter Nutzung der Kompositionalität



Anforderungsanalyse bei Übersetzern

- Festlegung der Quellsprache (Norm, Erweiterungen, Einschränkungen, ...)
- Festlegung des Niveaus der Interpretation und der Zielsprache:
 - abstrakte Maschine definieren
 - Abgrenzung Hardware/Laufzeitsystem
 - Entwurf Laufzeitsystem
- Systemeinbettung: andere Übersetzer, andere Sprachen, BS-Anschluß, ... berücksichtigen
- formales Modell der Semantik zur genauen Definition von „Korrektheit“ der Bedeutungsäquivalenz (wird leider meist unterlassen)



Abstrakte Maschinen

gegeben durch Typen, Objekte und Operationen, Ablaufsteuerung

- Sprache definiert eine abstrakte Maschine
 - Objekte bestimmen die Speicherelemente
 - Operationen und Ablaufsteuerung die Befehle

Beispiele: virtuelle Java-Maschine, .NET CLR

- Interpretation: Ausführung des Programms auf der Ebene der abstrakten Maschine
- Umkehrung: Jede abstrakte Maschine definiert Sprache



1 Motivation und Literatur

2 Übersetzungsarten

3 Architektur/Phasen

- Architektur
- Analyse
- Transformation
- Codierung
- Datenstrukturen



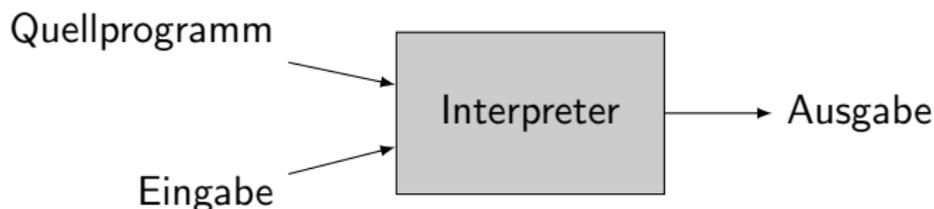
Ebenen der Interpretation bzw. Übersetzung

- Reiner Interpretierer
- Vorübersetzung
- Laufzeitübersetzer
- Vollständige Übersetzung
- Makrosprachen



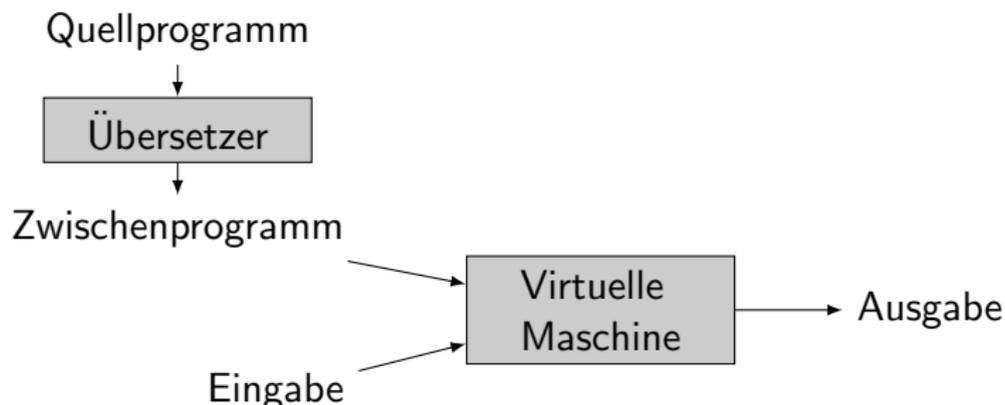
Reiner Interpretierer

- liest Quelltext jeder Anweisung bei jeder Ausführung und interpretiert sie
- billig, wenn nur einmal ausgeführt
- sinnvoll bei Kommandosprachen



Interpretation nach Vorübersetzung

- Analyse der Quelle und Transformation in eine für den Interpretierer günstigere Form, z.B. durch
 - Zuordnung Bezeichnergebrauch - Vereinbarung
 - Transformation in Postfixform
- nicht unbedingt Maschinenniveau
- Beispiel: Java-Bytecode, Smalltalk-Bytecode, Pascal P-Code



Laufzeitübersetzer (JIT-Compiler)

- Übersetzung während Programmausführung¹
- Auszuführender Code wird übersetzt und eingesetzt
 - schneller als reine Interpretation
 - Speichergewinn: Quelle kompakter als Zielprogramm
 - langsamer als vollständige Übersetzung, da nur lokaler Kontext benutzt (es ist auch Besseres denkbar, aber schwierig)
 - gut im Test (nur kleiner Teil des Codes ausgeführt)
 - kann dynamisch ermittelte Laufzeiteigenschaften berücksichtigen (dynamische Optimierung)
- Übersetzung mit Substitution
- Beispiel: .NET, das sich wegen nicht direkt greifbarer Typinformation nicht für Interpretation eignet

¹Laufzeitübersetzung erfunden 1974, CMU

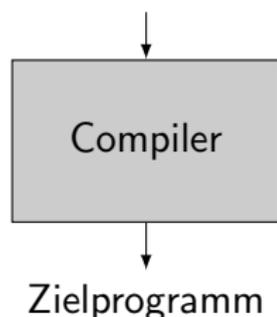


Vollständige Übersetzung

Auch (maschinennahe) Zielsprache definiert eine abstrakte Maschine. Interpretierer definiert durch:

- die Hardware der Zielmaschine
- Betriebssystem
- Laufzeitsystem (E/A-Unterprogramme, Speicherverwaltung, -bereinigung, ...)

Quellprogramm



Eingabe → Zielprogramm → Ausgabe

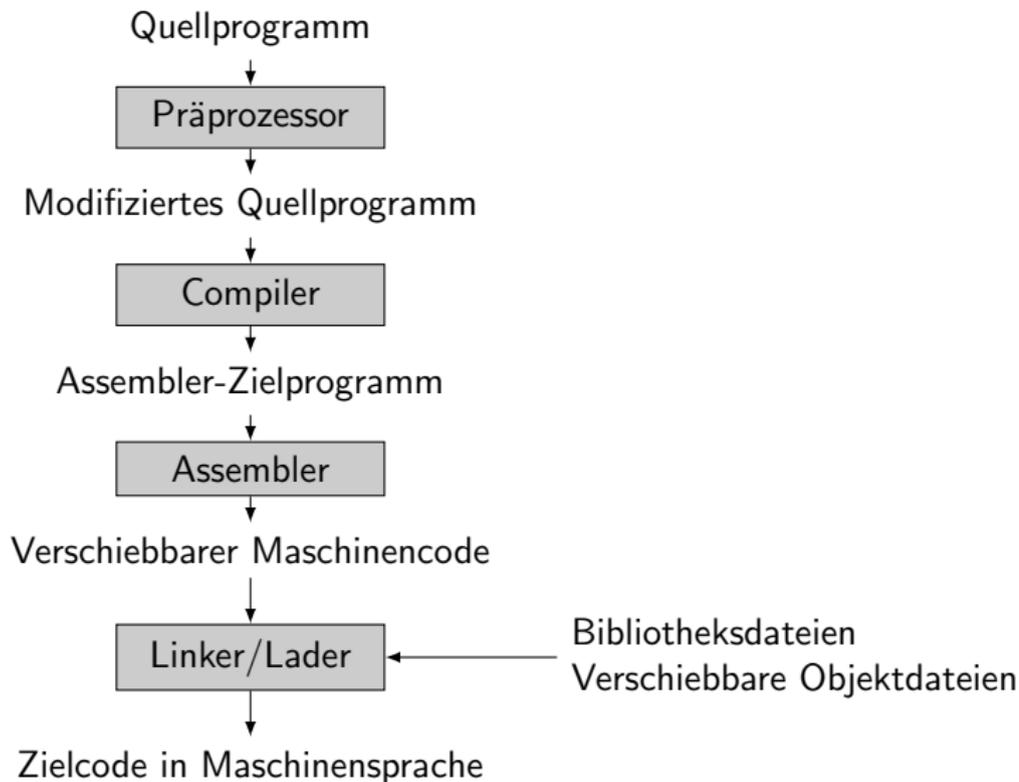


Makrosprachen

- Zielsprache identisch mit (Ausschnitt der) Quellsprache
- Makros werden nach vorgegebenen Vorschriften ersetzt
- Beispiele: Textformatierung, Textverarbeitung, Vorverarbeitung (Präprozessor) von C, C++, C#



Ein Sprachverarbeitungssystem



1 Motivation und Literatur

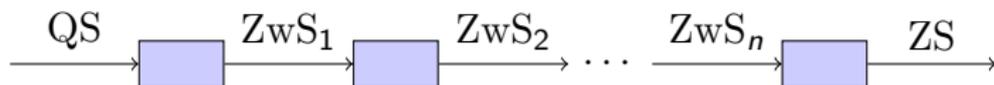
2 Übersetzungsarten

3 Architektur/Phasen

- Architektur
- Analyse
- Transformation
- Codierung
- Datenstrukturen



Fließbandarchitektur eines Übersetzer



QS: Quellsprache ZwS_i : Zwischensprache i ZS: Zielsprache

Die Fließbandarchitektur wurde auf Grund von Speicherbeschränkungen eingeführt (pragmatische Entscheidung), nicht auf Grund einer mathematischen Notwendigkeit.



Komponenten eines Übersetzters

Analyse	lexikalische Analyse	Lesen, Tokens (Symbole) erkennen
	syntaktische Analyse	Strukturbaum erzeugen
	semantische Analyse	Namens-, Typ-, Operatoranalyse, Konsistenzprüfung
Abbildung	Transformation	Daten abbilden, Operationen abbilden
	globale Optimierung	Konstantenfaltung, gemeinsame Teilausdrücke erkennen, globale Zusammenhänge erkennen und nutzen, Programmreorganisation
Codierung	Codeerzeugung	Ausführungsreihenfolge bestimmen, Befehlsauswahl, Registerzuweisung, Nachoptimierung
	Assemblieren und Binden	interne und externe Adressen auflösen, Befehle, Adressen, Daten codieren



Übersetzung einer Zuweisungsanweisung

position = initial + rate * 60

Lexikalischer Analysator

<id, 1> <=> <id,2> <+> <id,3> <*> <60>

Syntaktischer Analysator

$$\begin{array}{c} \text{=} \\ / \quad \backslash \\ \langle \text{id, 1} \rangle \quad + \\ \quad \quad \quad / \quad \backslash \\ \quad \quad \quad \langle \text{id, 2} \rangle \quad * \\ \quad \quad \quad \quad \quad \quad \backslash \\ \quad \quad \quad \quad \quad \quad \langle \text{id, 3} \rangle \quad 60 \end{array}$$

Semantischer Analysator

$$\begin{array}{c} \text{=} \\ / \quad \backslash \\ \langle \text{id, 1} \rangle \quad + \\ \quad \quad \quad / \quad \backslash \\ \quad \quad \quad \langle \text{id, 2} \rangle \quad * \\ \quad \quad \quad \quad \quad \quad \backslash \\ \quad \quad \quad \quad \quad \quad \langle \text{id, 3} \rangle \quad \text{inttfloat} \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad | \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad 60 \end{array}$$

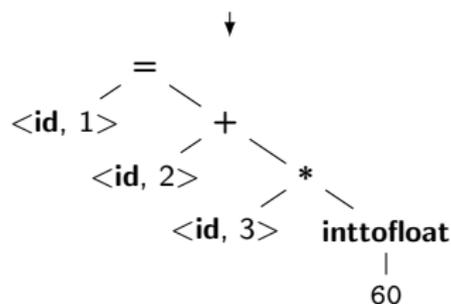
Einleitung

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

Symboltabelle



Übersetzung einer Zuweisungsanweisung



Zwischencodegenerator

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

Codeoptimierer

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

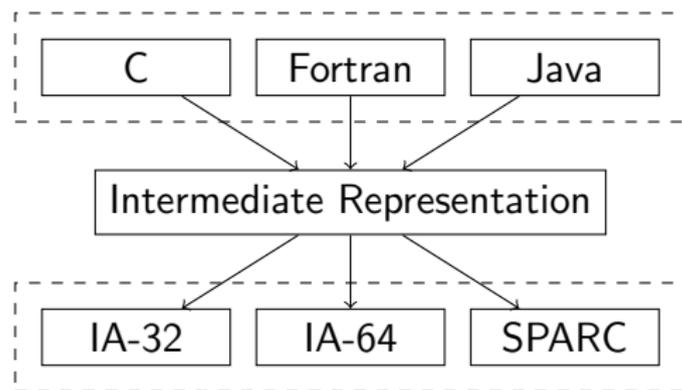
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Codegenerator

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



UNCOL



- Idee von T. B. Steel, Jr. 1961
 - UNCOL: UNiversal Computer Oriented Language.
 - Niemals direkt implementiert
-
- Beste heutige Annäherung an UNCOL: .NET CLI/CLR
 - Argumente
 - Ablaufsteuerung in allen Sprachen fast gleich
 - Anzahl der Basistypen gering und ähnlich
 - OO-Eigenschaften durch Verbunde simulierbar



Analysephase

- Aufgaben:
 - Feststellung der bedeutungstragenden Elemente
 - Zuordnung statischer Bedeutung
 - Konsistenzprüfung
- Linguistik: Syntax umfasst gesamte Analysephase
 - **Problem:** Nicht mit kontextfreier Grammatik (also effizient) zerteilbar; **deshalb Aufteilung**
- Schritte:
 - Symbolentschlüsselung
 - Syntaktische Analyse (Parsen)
 - Semantische Analyse
 - noch keine Übersetzung! nur Analyse



Lexikalische Analyse

- zerlegt Quellprogramm (Text) in Sequenz bedeutungstragender Einheiten (Tokenstrom)
- beseitigt überflüssige Zeichen(folgen) wie
 - Kommentare,
 - Leerzeichen, Tabulatoren usw.
- Abgetrennt von Parser, dadurch
 - reguläre Automaten möglich
 - Geschwindigkeit höher

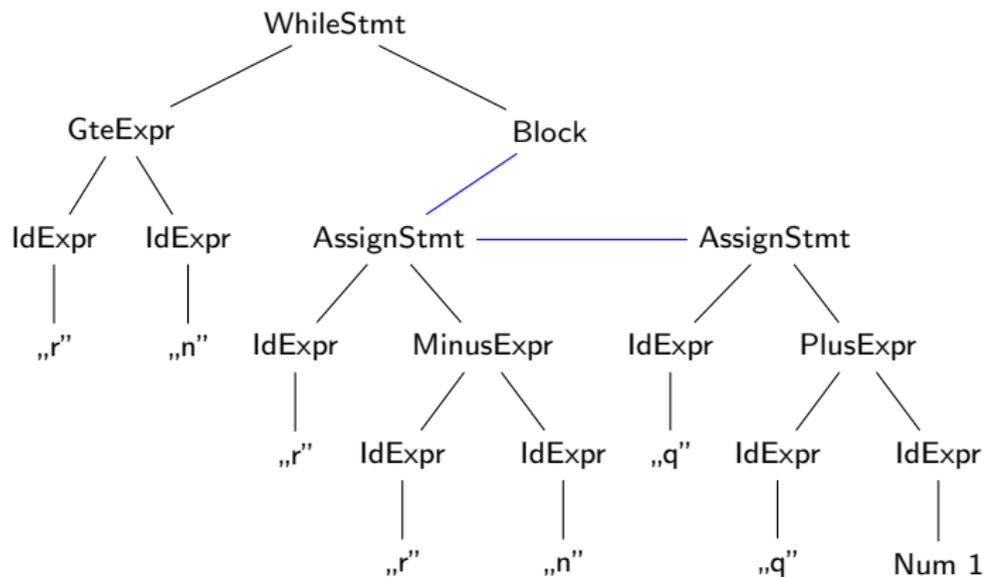


Syntaktische Analyse

- Syntaktische Analyse
- liefert abstrakten Syntax(-baum) des Programms
- aus softwaretechnischen Gründen deterministische kontextfreie Grammatik als Spezifikation



Abstrakter Syntaxbaum



Beispiel:

```
while (r>=n) {  
r=r-n;  
q=q+1  
}
```



Semantische Analyse

- Analyse des Strukturbaums
 - notwendig, da Programmiersprachen kontextsensitiv
 - Bedeutung von Bezeichnern
- Namens- und Typanalyse
 - ist eng verschränkt
- Konsistenzprüfungen
 - sind Einschränkungen der Programmiersprache
 - Beispiel: Statische negative Array-Grenzen
- Bedeutungsbindung (soweit möglich)
 - Operatoridentifikation
 - Zuordnung von Namensverwendungen zu ihren Definitionen



Transformationsphase

- Transformation
 - eigentliche Übersetzung
 - Speicherlayout der Objekte auf der Zielmaschine
 - Übersetzung der Operationen
 - Übersetzung der Ablaufsteuerung
 - Noch kein Zielcode generiert
- Optimierung
 - genauer: optimierende Transformationen (mit Korrektheitsnachweis)
 - „interessantester Teil“ heutiger Übersetzerforschung



40 „State of the Art“ Optimierungen

- Address Optimization
- Alias Analysis (by address)
- Alias Analysis (by type)
- Alias Analysis (const qualified)
- Array Bounds Optimization
- Bitfield Optimization
- Block Merging
- Branch Elimination
- Constant Folding
- Constant Propagation
- Cross Jumping
- CSE Elimination
- Dead Code Elimination
- Expression Simplification
- Forward Store
- Function Inlining
- Garbage Collection Optimisation
- Hoisting
- If Optimisation
- Induction Variable Elimination
- Instruction Combining
- Integer Divide Optimization
- Integer Modulus Optimization
- Integer Multiply Optimization
- Loop Collapsing
- Loop Fusion
- Loop Unrolling
- Narrowing
- New Expression Optimization
- Pointer Optimization
- printf Optimization
- Quick Optimization
- Register Allocation
- SPEC-Specific Optimization
- Static Declarations
- Strength Reduction
- String Optimization
- Synchronized Function Optimization
- Tail Recursion
- Try/Catch Block Optimization
- Unswitching
- Value Range Optimization
- Virtual Function Optimization
- Volatile Conformance

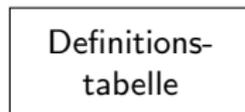
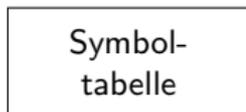
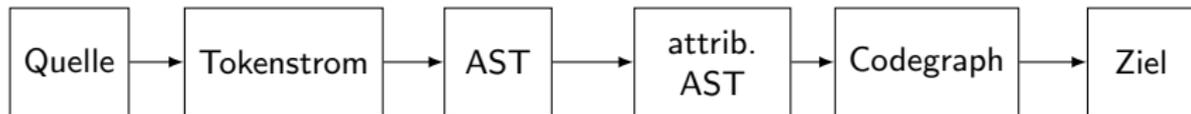
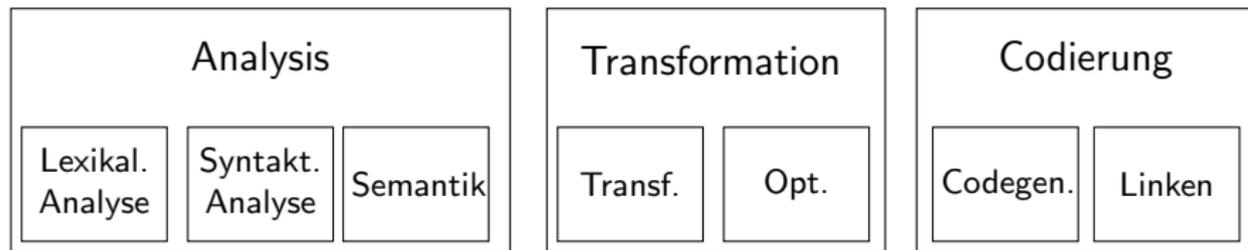


Codegenerierung

- Codegenerierung
 - wählt Befehle aus (Codeselektion)
 - bestimmt die Ausführungsreihenfolge
 - bestimmt konkrete Repräsentation
 - im Speicher
 - in Registern
- Nachoptimierung
- dann Assemblieren und Binden



Modulare Struktur von Compilern



Datenstrukturen

- Tokenstrom
- (Attributierter) Syntaxbaum (AST)
- Codegraph
- Zielprogramm
- Symboltabelle
- Definitionstabelle



Tokenstrom

Token:

bedeutungstragende Einheit im Programm Paar aus syntaktischem Schlüssel und Merkmal (Value)

Tokenstrom:

Darstellung des Quellprogramms als Folge von Tokens

- Schnittstelle: Lexer und Parser
- Vorsicht: Tokenstrom begrifflich notwendig, aber Implementierung noch offen! (gilt auch für die weiteren Datenstrukturen)



Attributierter Syntaxbaum

- Syntaxbaum:
 - Schachtelung der bedeutungstragenden Einheiten
 - kann kompositional Semantik zugeordnet werden
- Abstrakte Syntax:
 - Syntax stark vereinfacht und durch Struktur wiedergegeben
 - z.B. fallen Schlüsselworte wie while und Klammerungen weg
- Attribute:
 - Ergebnisse von semantischer Analyse
 - z.B. Typen, Namen, (konstante) Werte, Definitionsstellen, usw.
- Schnittstelle zwischen Syntaxanalyse, semantische Analyse, Transformationsphase



Codegraph

- Darstellung des Programms mit Datenobjekten und Operationen der Zielmaschine
- Befehle für Operationen noch nicht ausgewählt (Art des Registerzugriffs, Adressierungsmodi, usw. noch offen)
- Keine Register- und Speichereinschränkungen
- Schnittstelle:
 - Transformationsphase und Codegenerierung
 - Eingabe und Ergebnis vieler Optimierungen



Zielprogramm

- Ausgabe der Codesektion: symbolisch codiert
- Assembler: verschlüsselt binär, löst symbolische Adressen auf, soweit intern bekannt
- Binder: löst externe Symbole auf
- Ergebnis: Objektprogramm ohne symbolische Adressen (aber eventuell relativ adressiert)



Java Byte Code (Assembler für die JVM)

; Purpose: Print out "Hello World!"

```
.class public examples/HelloWorld
.super java/lang/Object
.method public <init>()V
aload_0                               ; 0x2a
invokespecial java/lang/Object/<init>() ; 0xb7 0x00 0x16
return                                  ; 0xb1
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 2
; push System.out onto the stack
getstatic java/lang/System/out Ljava/io/PrintStream; ; 0xb2 0x00 0x02
; push a string onto the stack
ldc "Hello World!"                       ; 0x12 0x15
; call the PrintStream.println() method.
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V ; 0x0d 0x00 0x0c
return                                    ; 0xb1
.end method
```



Symboltabelle

- Zuordnung Bezeichner, Literalkonstanten (evtl. andere Symbole) zur übersetzerinternen Codierung
- Aufbau während Symbolentschlüsselung, dann unverändert im Übersetzungslauf



Definitionstabelle

- Datenbank des Übersetzers
- Tabelle aller Definitionen (Vereinbarungen):
 - Zuordnung von Symbolen zu Bedeutungen
 - Wann bezeichnen die gleichen Symbole dieselben Objekte, Prozeduren
 - speichert analysierten semantischen Kontext von Symbolen

