

Vorlesung Übersetzerbau



Prof. G. Goos

Universität Karlsruhe

Adenauerring 20a, R. 201

Tel. 608-4760

ggoos@ipd.info.uni-karlsruhe.de

www.info.uni-karlsruhe.de



Übung Übersetzerbau



Rubino Geiß

Universität Karlsruhe

Adenauerring 20a, R. 229

Tel. 608-8352

rubino@ipd.info.uni-karlsruhe.de

www.info.uni-karlsruhe.de



Übersetzerbau Praktikum

Themen

- Übersetzerbau in der Praxis
- Bau eines Übersetzers für MiniJava
- Einsatz von Werkzeugen
- Moderne Konzepte: OO, SSA
- Erprobung von Optimierungen

Organisation

- IPD, Lehrstuhl Goos
- Michael Beck und Götz Lindenmaier
{beck | goetz}@ipd.info.uni-karlsruhe.de
- Vorbesprechung: Mi. 15.10.2003, 09:45
Geb. 50.41 (AVG), SR 207
- 2 SWS, prüfbar
- <http://www.info.uni-karlsruhe.de/>

Ausgebucht

Algorithmen Praktikum

Themen

- Algorithmen in der Praxis
- Schwerpunkt auf Implementierung
- Definierte Testumgebung
- Laufzeitmessungen
- Framework für Visualisierungen

Organisation

- IPD, Lehrstuhl Goos
- Florian Liekweg, Rubino Geiß
{liekweg | rubino}@ipd.info.uni-karlsruhe.de
- Vorbesprechung: Do. 16.10.03, 16:00
Geb. 50.41 (AVG), SR 207
- 2 SWS, prüfbar
- <http://www.info.uni-karlsruhe.de/>

Höhere Programmier-Sprachen

- Elemente und Konzepte
- Paradigmen: funktionale, logische, objekt-orientierte und Skript-Sprachen
- Exemplarisches Verständnis vieler Sprachen und deren Spezifika
- Definition von Semantik: operational, denotationell, axiomatisch

Organisation

- IPD, Lehrstuhl Goos
- Dr. Sabine Glesner
glesner@ipd.info.uni-karlsruhe.de
- Mo. 14:00 - 15:30, Info -102
- 2 SWS, prüfbar
- <http://www.info.uni-karlsruhe.de/>

Kapitel 1: Einleitung

1. Motivation und Literatur

2. Übersetzungsarten
3. Semantik und korrekte Übersetzung
4. Architektur / Phasen
 - 4.1 Analyse
 - 4.2 Abbildung
 - 4.3 Codierung
 - 4.4. Datenstrukturen

1.1 Warum ist Übersetzerbau interessant?

- 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
- 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, Programmtransformationen, automatische Anwendung von Entwurfsmustern, Metaprogrammierung, aspektorientierte Programmierung, Verarbeitung von XML, Metriken, ...

1.1 Literatur

- W.M. Waite, G. Goos: Compiler Construction, Springer 1984,
<http://www.info.uni-karlsruhe.de/teaching.php?id=20021>
- A.W. Appel: Modern Compiler Implementation in Java.
Cambridge University Press 1998, umfassendes leicht lesbares Buch
- R. Morgan: Building an Optimizing Compiler. Butterworth-Heinemann 1998,
erstklassiges Buch über Optimierung
- S.S. Muchnik: Advanced Compiler Design & Implementation. Morgan-Kaufmann 1997.
- R. Wilhelm, D. Maurer: Übersetzerbau, 2. Aufl. Springer 1997,
behandelt ausführlich auch funkt. und logische Programmiersprachen,
Einführung in Baumautomaten
- W.M. Waite, L.R. Carter: An Introduction to Compiler Construction,
Harper-Collins 1992, praktisch, leicht lesbar, Architektur
- A.V. Aho, R. Sethi, J.D. Ullman: Compilers: Principles, Techniques and Tools,
Addison-Wesley 1986, Standardverfahren der Optimierung und Codegen.
- M. Wolfe: High-Performance Compilers for Parallel Computing. Addison-Wesley 1996.

4 Anwendungsfelder

- Text → Information
 - Textanalyse
 - Beispiel: LaTeX, Word, Konfigurationsdateien
- Text → Text
 - Quell-Quell-Transformation
 - Beispiel: Präprozessor, UML nach C
- Programm → Kellercode
 - Beispiel: Java Byte Code, .NET MSIL
- Programm → Maschinencode
 - Beispiel: Alle C-Übersetzer



1.1 Übersetzen

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



1.1 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
- Achtung: diese Ziele widersprechen sich teilweise



1.1 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



1.1 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 Übersetzer unter Nutzung der Kompositionalität



1.1 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)



1.1 Abstrakte Maschinen

gegeben durch Typen, Objekte und Operationen, Ablaufsteuerung

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

Beispiel: virtuelle Java-Maschine, .NET CLR

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



Kapitel 1: Einleitung

1. Motivation und Literatur
2. **Übersetzungsarten**
3. Semantik und korrekte Übersetzung
4. Architektur / Phasen
 - 4.1 Analyse
 - 4.2 Abbildung
 - 4.3 Codierung
 - 4.4. Datenstrukturen



1.2 Ebenen der Interpretation bzw. Übersetzung

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



1.2 Reiner Interpretierer

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



1.2 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



1.2 Laufzeitübersetzer

- Übersetzung während Programmausführung
 - 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 sehr 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 fehlender Typinformation nicht für Interpretation eignet
- Laufzeitübersetzung erfunden 1974, CMU



1.2 Vollständige Übersetzung

Auch (maschinennahe) Zielsprache definiert eine abstrakte Maschine

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



1.2 Makrosprachen

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



Kapitel 1: Einleitung

1. Motivation und Literatur
2. Übersetzungsarten
3. Semantik und korrekte Übersetzung
4. Architektur / Phasen
 - 4.1 Analyse
 - 4.2 Abbildung
 - 4.3 Codierung
 - 4.4. Datenstrukturen



1.3 Definition formaler Semantik

- Algebraische Spezifikationen,
- denotationelle Semantik: Programm definiert Abbildungen,
- ...
- operationelle Semantik:
 - abstrakte Zustandsmaschinen (*abstract state machines, ASM*): Programm definiert Zustandsübergänge
 - natürliche Semantik

Am IPD benutzen wir ASMs bzw. natürliche Semantik, weil dies gleichmäßig die Beschreibung der Quell- und Zielsprache erlaubt



1.3 abstrakte Zustandsmaschinen

ASMs verallgemeinern endliche Zustandsmaschinen, d.h. endl. Automaten

- jeder Zustand q_t ist eine Algebra mit Wertemenge und Operationen
- durch einen Zustandsübergang $q_t \rightarrow q_{t+1}$ werden die Ergebnisse von Funktionsaufrufe verändert
 - q_{t+1} unterscheidet sich von q_t nur durch diese veränderten Funktionsergebnisse
- die Veränderungen können unter Bedingung stehen usw.

Beispiel: (ct bezeichnet den „Befehlszähler“ current-task, nullstellige Fkt.)

```
if ct = While then
  if value(ct,condition) = true then ct := truetask(ct)
  else ct := falsetask(ct)
endif
endif
```



1.3 Zustände

Sequentielles Programm durchläuft Zustände

$q_0 q_1 q_2 \dots$

- Zustand q_t besteht aus Objekten, die zum Zeitpunkt t existieren: $q_t = (q_t^0, q_t^1, \dots, q_t^n)$
- Operation im Interpreter entspricht Übergang $q_t \rightarrow q_{t+1}$
- Zustandsmenge Q die während Programmlauf existieren kann: Zustandsraum eines Programms (abhängig von Programm & Eingaben)



1.3 Beispiel: Zustände und Zustandsfolgen

- Beispiel:

```
while ( i != j )
  if (i>j) i=i-j else j=j-i
```
- Algorithmus durchläuft die Zustandsfolge
 - Anfang: $i = 36$ $j = 24$
 - $i = 12$ $j = 24$
 - Ende: $i = 12$ $j = 12$



1.3 Beispiel: Äquivalenter Algorithmus

- ```
while (i != j)
 if (i>j) i=i%j else j=j%i
```
- liefert für  $i = 36$ ,  $j = 24$  die gleiche Zustandsfolge
- Einsicht: Algorithmus kann durch besseren ersetzt werden, wenn es nur auf das Endergebnis ankommt



# 1.3 Maschinenprogramm für Beispiel

```
;; Code für alpha-Prozessor
;; $i: Registernummern
;; Annahme $10 = i, $9 = j

L0:
 cmpeq $10, $9, $0
 bne $0, L9
 cmplt $9, $10, $1
 beq $1, L7
 subl $10, $9, $10
 br L0

L7:
 subl $9, $10, $9
 br L0

L9:
 end
```

Bei der Übersetzung ändert sich der Zustandsraum; insbesondere können es mehr Zustände werden (\$0, \$1).

Die Zuordnung von Variablen zu Registern kann sich auch ändern.



# 1.3 Beobachtbare Zustände

beobachtbar:

- Anfangs- und Endzustand
- Ein- und Ausgaben
- Im beobachtbaren Zustand  $q$  nur bestimmte Objekte  $q$  beobachtbar:  $q$  Ausschnitt aus  $q_t = (q_t^0, q_t^1, \dots, q_t^n)$
- Berechnung nicht beobachtbarer Objekte unerheblich, wenn sie terminiert
- Zwischenzustände unerheblich, wenn es nicht unendlich viele sind



# 1.3 Korrekte Übersetzung

- Betrachte nur beobachtbare Zustände terminierender Programme:
  - deterministische Programme: Folge von beobachtbaren Zuständen
  - indeterministische Programme: azyklischer Graph von beobachtbaren Zuständen, Programmausführung entspricht Pfad vom Anfangs- zum Endzustand in Graph



# 1.3 Azyklischer Zustandsgraph

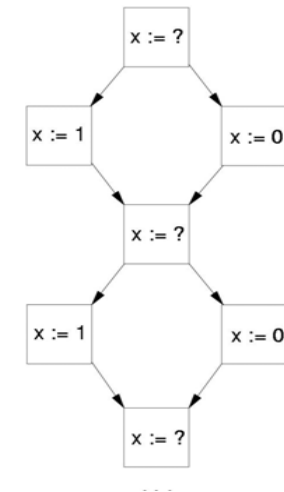
Indeterministisches Programm:

```
do
 true -> x:=1
[] true -> x:=0
od
```

Korrekte Ausführung auch:

```
do true -> x:=1 od
```

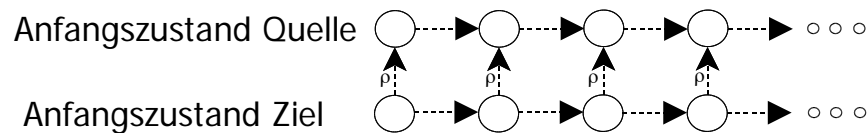
Bei nichtdeterministischer Programmausführung darf sich der Übersetzer einen gültigen Pfad aussuchen.





## 1.3 Korrekte Übersetzung I

- Quellprogramm  $\pi$
- Zielprogramm  $\pi'$
- Bisimulation:  
Bei gleicher Eingabe gleiche beobachtbare Objekte in beobachtbarer Zustandsfolge  
(Relation  $\rho$  zwischen beobachtbaren Zuständen)



## 1.3 Eigenschaften korrekter Übersetzung

- Zulässige Zustandsfolgen bzw. Pfade werden durchlaufen
- Zielprogramm divergiert nur, wenn auch Quellprogramm divergiert
- **Aber:** Endlichkeit von Betriebsmitteln:  
Zielprogramm darf mit Fehler enden, auch wenn Quellprogramm mathematisch korrekt
- Mögliche Verschärfung: Bis zur Fehlermeldung muss das Zielprogramm korrekt sein (d.h. bis zum letzten beobachtbaren Zustand vor dem Fehlerzustand korrekt)



## 1.3 Beschränkung der Zahlbereiche

- for  $i := v$  up to  $n$  do ... end
- Betrachte  $n = \text{maxint}$ :
  - Überprüfung, ob Überlauf bei Berechnung von  $(i+1)$
  - Bei Überlauf:  $\text{maxint}+1 = -\text{maxint}-1$  bei Benutzung des Zweierkomplements
  - ☞ Schleife würde nicht terminieren
  - aktuelles Beispiel: Aufzählung aller Zeichen vom Typ character



## 1.3 Korrekte Übersetzung II

- $\pi'$  ist korrekte Übersetzung von  $\pi$ , wenn für alle zulässigen Eingaben gilt:
  - wenn  $\pi'$  regulär einen nächsten beobachtbaren Zustand  $q'$  erreicht, dann kann  $\pi$  einen Zustand  $q$  mit  $q \rho q'$  erreichen;
  - wenn  $\pi$  regulär und deterministisch einen beobachtbaren Zustand  $q$  erreicht, dann erreicht  $\pi'$  regulär einen Zustand  $q'$  mit  $q \rho q'$  oder  $\pi'$  terminiert mit einer Fehlermeldung wegen Verletzung von Betriebsmittelbeschränkungen;
- regulär: endliche viele Schritte ohne Fehlermeldung



## 1.3 Korrekte Übersetzung III: präzise Definition

$\pi'$  ist korrekte Übersetzung von  $\pi$ , wenn für alle zulässigen Eingaben eine der folgenden Aussagen gilt:

- Zu jeder Folge beobachtbarer Zustände  $q_0', q_1', \dots, q_k'$  von  $\pi'$  die regulär terminiert, gibt es eine terminierende Zustandsfolge  $q_0, q_1, \dots, q_k$  von  $\pi$  mit  $q_i \rho q_i'$  für  $0 \leq i \leq k$ ;
- Zu jeder nicht-terminierende Folge beobachtbarer Zustände  $q_0', q_1', \dots$  von  $\pi'$  gibt es eine Zustandsfolge  $q_0, q_1, \dots$  von  $\pi$  mit  $q_i \rho q_i'$  für alle  $i$ ;
- Zu jeder Folge beobachtbarer Zustände  $q_0', q_1', \dots, q_{k+1}'$  von  $\pi'$ , die mit einer Fehlermeldung wegen Verletzung von Betriebsmittelbeschränkungen terminiert, gibt es eine Zustandsfolge  $q_0, q_1, \dots, q_k$  von  $\pi$  mit  $q_i \rho q_i'$  für  $0 \leq i \leq k$ ;

regulär: endliche viele Schritte ohne Fehlermeldung

$\rho$ : korrespondierende beobachtbare Zustandsvariable haben gleichen Wert

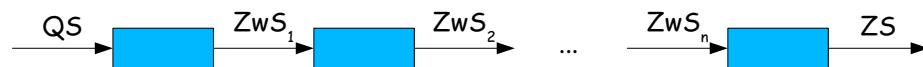


## Kapitel 1: Einleitung

1. Motivation und Literatur
2. Übersetzungsarten
3. Semantik und korrekte Übersetzung
4. **Architektur / Phasen**
  - 4.1 Analyse
  - 4.2 Abbildung
  - 4.3 Codierung
  - 4.4. Datenstrukturen



## 1.4 Fließbandarchitektur eines Übersetzer



QS: Quellsprache

ZwS<sub>i</sub>: Zwischensprache i

ZS: Zielsprache

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

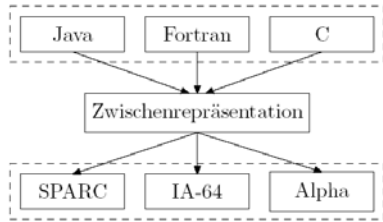


## 1.4 Komponenten eines Übersetzers

|           |                                   |                                                                                                                         |
|-----------|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| Analyse   | Symbolentschlüsselung             | Lesen, Symbole erkennen                                                                                                 |
|           | syntaktische Analyse (Zerteilung) | 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, Registerzuteilung, Nachoptimierung                                    |
|           | Assemblieren und Binden           | interne und externe Adressen auflösen, Befehle, Adressen, Daten codieren                                                |



## 1.4 UNCOL



- Erfunden 1961 von T. B. Steel, Jr.
- UNCOL: UNiversal Computer Oriented Language.
- Niemals direkt implementiert

- Beste Annäherung an UNCOL: .NET CLI/CLR
- Argumente
  - Ablaufsteuerung in allen Sprachen fast gleich
  - Anzahl der Basistypen gering und ähnlich
  - OO-Eigenschaften durch Verbunde simulierbar



## 1.4.1 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 (Zerteilung)
  - Semantische Analyse
  - noch keine Übersetzung! nur Analyse



### 1.4.1 Symbolentschlüsselung

- zerlegt Quellprogramm (Text) in Sequenz bedeutungstragender Einheiten (Symbolfolge)
- beseitigt überflüssige Zeichen(folgen) wie
  - Kommentar,
  - Leerzeichen, Tabulatoren usw
- Abtrennt von Zerteilung, dadurch
  - reguläre Automaten möglich
  - Geschwindigkeit höher



### 1.4.1 Zerteilung

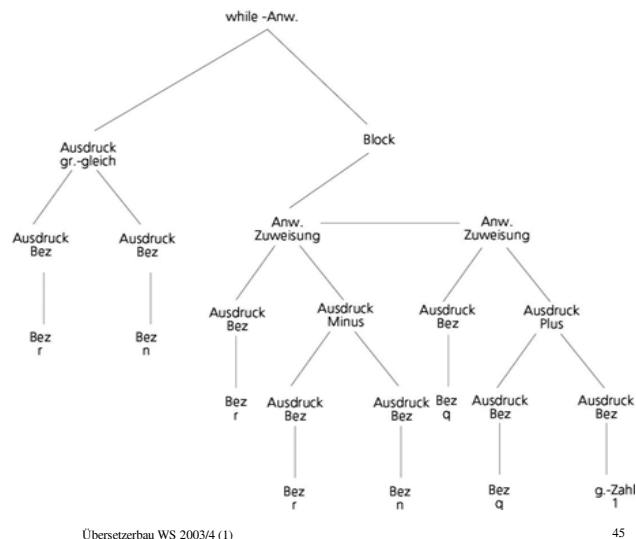
- liest Symbolfolge
- liefert Struktur(-baum) des Programms
- aus softwaretechnischen Gründen deterministische kontextfreie Grammatik als Spezifikation



## 1.4.1 Strukturbaum

Beispiel:

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



## 1.4.1 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



## 1.4.2 Abbildungsphase

- 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



## 1.4.2 40 „State of the Art“ Optimierungen

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

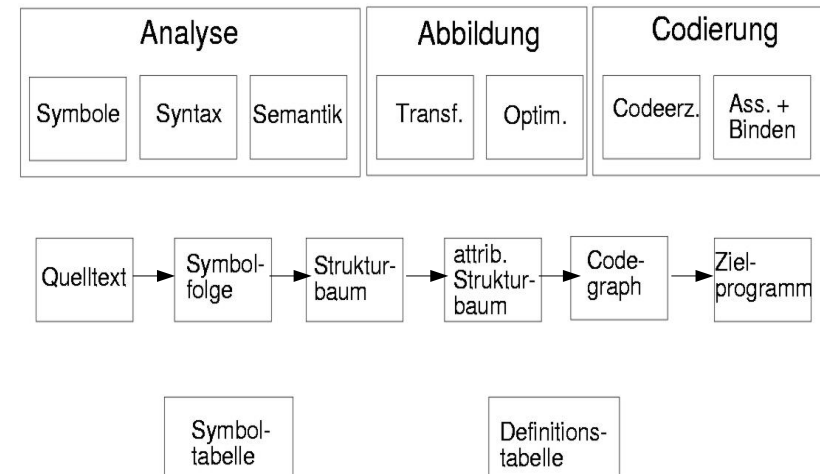


## 1.4.3 Codierung

- Codeerzeugung
  - wählt Befehle aus (Codeselektion)
  - bestimmt die Ausführungsreihenfolge
  - bestimmt konkrete Repräsentation
    - im Speicher
    - in Registern
- Nachoptimierung (lokale „Gucklochoptimierung“)
- dann Assemblieren und Binden



## 1.4.4 Modulare Struktur von Übersetzern



## 1.4.4 Datenstrukturen

- Symbolfolge
- (Attributierter) Strukturbaum
- Codegraph
- Zielprogramm
- Symboltabelle
- Definitionstabelle



## 1.4.4 Symbolfolge

- Symbol: bedeutungstragende Einheit im Programm
  - Paar aus syntaktischem Schlüssel (Token) und Merkmal (Value)
- Symbolfolge: Darstellung des Quellprogramms als Folge von Symbolen
- Schnittstelle: Symbolentschlüsseler und Zerteiler
- Vorsicht: Symbolfolge begrifflich notwendig, aber Implementierung noch offen!  
(gilt auch für die weiteren Datenstrukturen)



## 1.4.4 Attributierter Strukturbaum

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



## 1.4.4 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



## 1.4.4 Zielprogramm

- Ausgabe der Codeselektion: 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)



## 1.4.4 Java Assembler (symbolisch und sedezimal codiert)

```
; Purpose: Prints out "Hello World!"

.class public examples/HelloWorld
.super java/lang/Object

.method public <init>()V
 aload_0
 invokespecial java/lang/Object/<init>()
 return
.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;
 ; push a string onto the stack
 ldc "Hello World!"
 ; call the PrintStream.println() method.
 invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
 return
.end method
```



## 1.4.4 Symboltabelle

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



## 1.4.4 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

