Übersetzerbau

lange Einführung

Berthold Hoffmann

Studiengang Informatik
Universität Bremen

Sommersemester 2009
(Vorlesung am 7. April 2009)

lange Einführung (7. April 2009)

1 Organisation
- Über die Veranstalter
- Regularien
- Inhalt
- Materialien

2 Einführung

Gestatten?

Berthold Hoffmann

- AG Krieg-Brückner
- Büro: Cartesium 2.48
- Telefon: 218-64222
- Email: hof@. . .

Gestatten?

Berthold Hoffmann

- AG Krieg-Brückner
- Büro: Cartesium 2.48
- Telefon: 218-64222
- Email: hof@...

Thomas Röfer

- AG Krieg-Brückner
- Büro: Cartesium 0.55
- Telefon: 218-64200
- Email: roefer@...

Interessen

- Übersetzer
- Programmiersprachen
- visuelle Sprachen
- Graphtransformation

Gestatten?

Berthold Hoffmann

- AG Krieg-Brückner
- Büro: Cartesium 2.48
- Telefon: 218-64222
- Email: hof@. . .

Interessen

- Übersetzer
- Programmiersprachen
- visuelle Sprachen
- Graphtransformation

Thomas Röfer

- AG Krieg-Brückner
- Büro: Cartesium 0.55
- Telefon: 218-64200
- Email: roefer@. . .

Interessen

- Objektorientiertes Programmieren
- Raumkognition
- Service-Robotik
- ROBOCUP

Zeit und Raum

- interaktive Vorlesung mit jeweil 1/4 Übungsanteil
- Vorlesung: Montags und Dienstags 10:15–11:45
- Übung: jeweils zu Beginn
- Raum: MZH 5210
- Sprechstunde: nach Vereinbarung (Email)

Aufgabenzettel

- Anwendung des Vorlesungsstoffs
  (ca. 30-60 min Bearbeitungsaufwand)
- werden nicht abgegeben, korrigiert oder benotet
- werden aber gemeinsam besprochen
- Musterlösungen sollen erstellt werden

Scheinkriterien

Übersetzerbau *solo*

- mündliche Prüfung
  (ohne Praktikum, 6 ETCS)

Übersetzerbau *mit* Übersetzer-Praktikum

- Fachgespräch und Abnahme des *oops*-Übersetzers
  (Kurs + Praktikum, 6+4 ETCS)

Inhaltsübersicht

Prinzipien der Implementierung von Programmiersprachen

"*Was jede(r) Informatiker(in) über Übersetzer wissen sollte*"

Inhaltsübersicht

Prinzipien der Implementierung von Programmiersprachen

"*Was jede(r) Informatiker(in) über Übersetzer wissen sollte*"

- Interpeter, Übersetzer, Struktur von Übersetzern
- lexikalische, syntaktische und kontextuelle Analyse
- Codeerzeugung

Inhaltsübersicht

Prinzipien der Implementierung von Programmiersprachen

"*Was jede(r) Informatiker(in) über Übersetzer wissen sollte*"

- Interpeter, Übersetzer, Struktur von Übersetzern
- lexikalische, syntaktische und kontextuelle Analyse
- Codeerzeugung

Lehrziele

- Grundlagen und Techniken für die Analyse von Sprachen
- Werkzeuge für die Analyse (`lex` und `yacc`)
- Grundzüge der Codeerzeugung

Plan der Veranstaltung

Lexikalische Analyse

- reguläre Definitionen
- endliche Automaten (`lex`)
- Streuspeicher (*hashing*)

Plan der Veranstaltung

Lexikalische Analyse

- reguläre Definitionen
- endliche Automaten (`lex`)
- Streuspeicher (*hashing*)

Syntaxanalyse

- kontextfreie Grammatiken
- Parsieren (SLL(k), LR(k))
- Parsergeneratoren (`yacc`)
- Baumaufbau

Plan der Veranstaltung

Lexikalische Analyse

- reguläre Definitionen
- endliche Automaten (`lex`)
- Streuspeicher (*hashing*)

Syntaxanalyse

- kontextfreie Grammatiken
- Parsieren (SLL(k), LR(k))
- Parsergeneratoren (`yacc`)
- Baumaufbau

Kontext-Analyse

- Namens- und Typanalyse
- Attributgrammatiken
- Baumtraversierung

Plan der Veranstaltung

Lexikalische Analyse

- reguläre Definitionen
- endliche Automaten (`lex`)
- Streuspeicher (*hashing*)

Kontext-Analyse

- Namens- und Typanalyse
- Attributgrammatiken
- Baumtraversierung

Syntaxanalyse

- kontextfreie Grammatiken
- Parsieren (SLL(k), LR(k))
- Parsergeneratoren (`yacc`)
- Baumaufbau

Codeerzeugung

- imperative Sprachen
- objektorientierte Sprachen
- Registerzuteilung
- Instruktionsauswahl

Inhalt des Übersetzerpraktikms

Vertiefung der Kenntnisse aus der Vorlesung

- Studieren des *oops*$_0$-Systems (in JAVA geschrieben)
- schrittweise Erweiterung des Üersetzers
  - *oops*$_1$: geklammerte Anweisungen
  - *oops*$_2$: Boolesche Ausdrücke
  - *oops*$_3$: Methoden mit Parameters und Resultaten
  - *oops*$_4$: Vererbung und dynamische Bindung
  - *oops*$_5$: Speicherbereinigung (garbage collection)
  - *oops*! (für Süchtige): einige *Kür*-Aufgaben
- Implementierung in JAVA (Thomas Röfer) oder
- HASKELL (Berthold Hoffmann)

Material zur Veranstaltung

www.informatik.uni-bremen.de/agbkb/lehre/uebersetzer/

- Folienkopien
- Hinweise zum Nachlesen
- Aufgaben
- Literatur
- Links

empfohlene Lehrbücher I

- A. V. AHO, M. S. LAM, R. SETHI, J. D. ULLMAN.
  Compilers – Priciples, Techniques and Tools (2/e)
  Bonn: Addison-Wesley, 2006.
- R. WILHELM, D. MAURER.
  Übersetzerbau: Theorie – Konstruktion – Generierung.
  2. Aufl., Berlin-Heidelberg: Springer, 1996.
- D. A. WATT. *Programming Language Processors in Java*.
  Wokingham: Prentice-Hall, 2000.
- A. APPEL.
  Modern Compiler Implementation in JAVA (*. . . in* ML / C)
  Cambridge, MA: Cambridge University Press 1998.

lange Einführung (7. April 2009)

1 Organisation

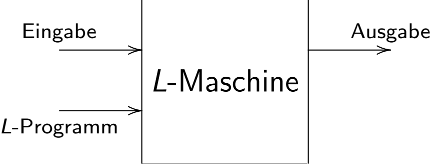
2 Einführung

- konkrete und abstrakte Maschinen
- Implementierung von Übersetzern
- Struktur von Übersetzern

universelle programmierbare Rechenmaschinen

Charakteristische Eigenschaften

- kann ein beliebiges Programm p einer Sprache L ausführen
- berechnet Ausgaben für beliebige Eingaben von p
- Uns interessieren L-Maschinen (für *höhere Sprachen L*)



konkrete Maschinen

die harte Lösung

Baue die L-Maschine in *Hardware*

- Das ist *teuer* und *kompliziert*.
- Solche Maschinen sind nicht flexibel"
- Ausnahme: *Silicon compilers*

die weiche Lösung

Realisiere die L-Maschine als *Programm*

- als *Interpreter* oder
- als *Übersetzer*

I p O
L
L

Interpreter

Simulation einer L-Maschine

Vorteil Benutzung herkömmlicher Hardware
L-Maschinen sind *abstrakt*

Nachteil Programm und Eingabe werden gleichzeitig bearbeitet
Dadurch entsteht *overhead*

Trotzdem Interpreter sind interessant

- für einfache Sprachen
- für sehr komplizierte Sprachen

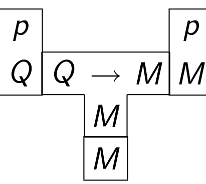
Übersetzer

Erst übersetzen, dann ausführen

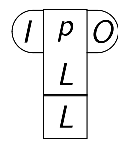
Vorteil Quell-Programm wird nur einmal bearbeitet
kein *overhead* beim Ausführen

Nachteil aufwändiger
(man muss ein vollständiges M-Programm konstruieren)

Trotzdem die bessere Art der Implementierung



Übersetzungszeit

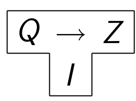


Laufzeit

Arten von Übersetzer

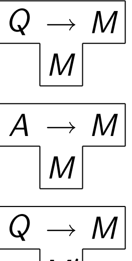
typische Eigenschaften

- Q ist eine höhere Programmiersprache
- Z ist eine maschinennahe Sprache
- I ist eine bereits implementierte Sprache (aber: siehe *bootstrapping*)



einige spezielle Übersetzer:

- "eingeboren" (native): $Z = I$ und Z ist Maschinensprache
- Assemler: eingeborene Übersetzer für eine maschinennahe Sprache
- "Cross-compiler": Z und I sind verschiedene Maschinensprachen

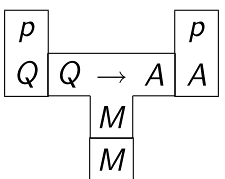


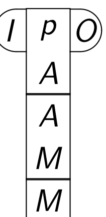
Kombination von Übersetzern und Interpretern

Erst übersetzen, dann interpretieren

- Übersetzen in *einfache* Sprache A (abstrakte Maschinensprache)
- Interretation der A-Programms auf M
- A sollte *plattformunabhängig* sein Weshalb wohl?
- reale Beispiele
  - Das PL0-System
  - Der Zürcher PASCAL-Übersetzer
  - Welche Sprache noch?

Übersetzer können auch mit Übersetzern kombiniert werden





Implementierungssprache

Anforderungen

- komplexe Daten
  - Bäume
  - Graphen
  - Tabellen
- Rekursion
- Modularität

Konsequenz

- Implementierung in einer höheren Programiersprache H
- Übersezer müssen selbst übersetzt werden

$Q \longrightarrow M$ $\quad$ $Q \longrightarrow M$

H $\quad$ $H \longrightarrow M$ $\quad$ M

M

M

Übersetzer schreiben in der eigenen Quellsprache

Frage Wie bekommt man das zum Laufen?

Antwort Durch *bootstrapping* mit *zwei* Übersetzern

Voraussetzungen für *bootstrapping*

ein *Master*-Übersetzer

$Q \xrightarrow{m} M$ / Q'

- effizient
- prodziert guten Code
- benutzt nur eine Teilsprache $Q' \subseteq Q$

ein *Einweg*-Übersetzer

$Q' \xrightarrow{e} M$ / M

- ineffizient
- produziert schlechten Code
- übersetzt die Teilsprache $Q' \subseteq Q$

bootstrapping

Erster Schritt: $Q \xrightarrow[M]{1} M$

- ineffizient
- prodziert guten Code

Zweiter Schritt: $Q \xrightarrow[M]{2} M$

- effizient
- produziert guten Code

Wiederverwendung von Übersetzern

re-hosting: $Q \xrightarrow[H]{} Z \Rightarrow Q \xrightarrow[H']{} Z$

- Übertragen auf einen anderen Gastrechner (*host*)
- *bootstrapping* mit Einweg-Übersetzer $Q' \xrightarrow[P]{} Z$ in universell verfügbarer Portierungssprache P

re-targetting: $Q \xrightarrow[H]{} Z \Rightarrow Q \xrightarrow[H]{} Z'$

- Code für eine andere Zielmaschine erzeugen
- Aufteilen in plattformunabhängige und plattformabhängige Teile

re-sourcing: $Q \xrightarrow[H]{} Z \Rightarrow Q' \xrightarrow[H]{} Z$

- Anpassen auf andere Quellsprache
- Spezialfall *Erweiterung*: $Q \subseteq Q'$
- im allgemeinen: mit Übersetzer-Werkzeugen

Der Anspruch der universellen Verfügbarkeit

- Alle höheren Sprachen sollen auf allen Plattformen laufen.
- Das erfordert hohen Aufwand:
  - Bei k Sprachen und n Plattformen braucht man dazu $k \times n$ (eingeborene) Übersetzer $L_i \underset{M_j}{\longrightarrow} M_j$
  - Für jede neue Sprache L' braucht man n Übersetzer $L' \underset{M_j}{\longrightarrow} M_j$.
  - Für jede neue Plattform M' braucht man k Übersetzer $L_i \underset{M'}{\longrightarrow} M'$.

Wie kann der Aufwand vermindert werden?

Der UNCOL-Ansatz (60er Jahre)

- Definiere eine *universelle Zwischensprache U*
- Implementiere $l + m$ Übersetzer $L_i \longrightarrow U$ und $U \underset{M_j}{\longrightarrow} M_j$
- Für jede neue Sprache L' braucht man *einen* Übersetzer $L' \longrightarrow U$.
- Für jede neue Plattform M' braucht man *einen* Übersetzer $U \underset{M'}{\longrightarrow} M'$.
- Eine *effiziente* Sprache U konnte nicht gefunden werden.
- Weshalb wohl?

PASCAL und *P-Code*

Der portable Zürcher PASCAL-Übersetzer (um 1970)

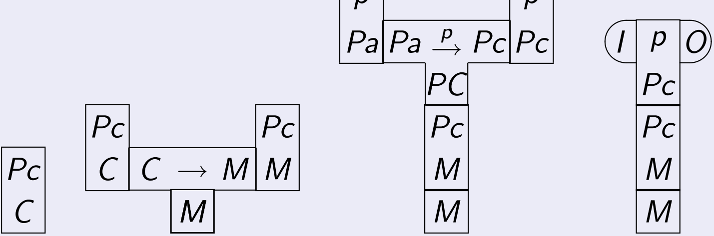
- Definiere eine Assemblerspache P-CODE, die auf allen Plattform leicht zu implementieren ist.
- Gib folgende portable Übersetzer und Interpreter vor

$Pa \xrightarrow{m} Pc$ / Pa $\qquad$ $Pa \xrightarrow{p} Pc$ / Pc $\qquad$ Pc / Pa

Portieren auf Maschine M

Annahme: Es gibt einen eingeborene C-Übersetzer

- Schreibe einen P-CODE-Interpreter in C
- Übersetze ihn in Maschinencode
- Heraus kommt ein interpretierender Übersetzer

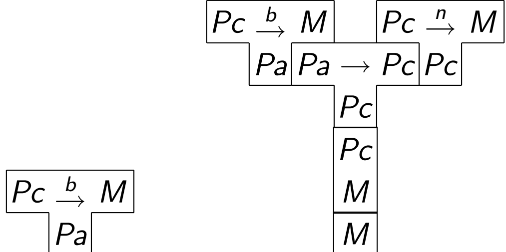


- Dieser Übersetzer ist nicht sehr schnell.
- Der Übersetzer in PASCAL wird hier nicht gebraucht.

Erzeuge eines eingeborenen P-CODE-Übersetzers

Annahme: Es gibt einen eingeborene C-Übersetzer

- Schreibe einen P-CODE-Übersetzer nach M (in PASCAL)
- Übersetze ihn in Maschinencode.



Bootstrapping des portierten Übersetzers

Der eigentliche *bootstrap*

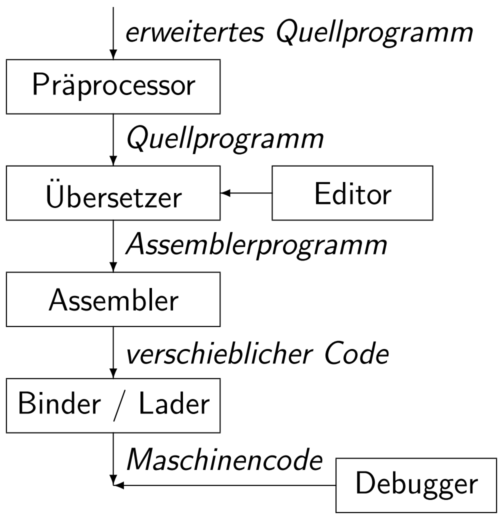
- *Bootstrappe* den P-CODE-Übersetzer mit sich selbst.
- Resultat: eingeborener Zweischrittübersetzer

$Pc \xrightarrow{n} M$ | $Pc \xrightarrow{1} M$ | Pc $Pc \xrightarrow{n} M$ M | Pc | Pc | M | M

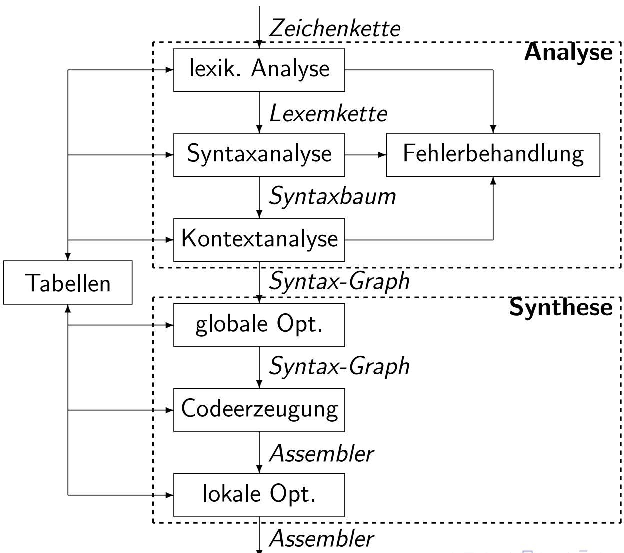
$Pa \xrightarrow{p} Pc$ | $Pa \xrightarrow{2} Pc$ | Pc $Pc \xrightarrow{1} M$ M | M | M

p Pa | $Pc \xrightarrow{2} M$ | p Pc | $Pc \xrightarrow{n} M$ | p Pc | M M | M M

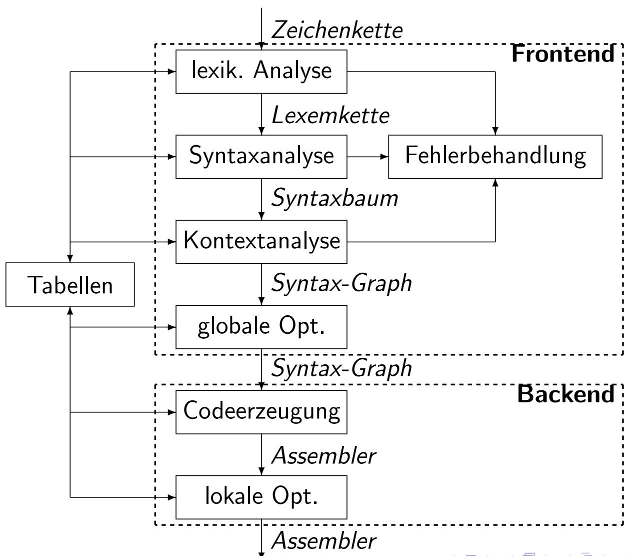
Umgebung eines Übersetzers



Phasenmodell der Übersetzung



Plattformabhängigkeit



Phasen und Pässe

Phasen

- Phasen sind Module, die nacheinander ablaufen
  - Ihre Ausführung kann verzahnt sein
  - Datenfluss von "vorn" nach "hinten"
- **Beispiel**: lexikalische und syntaktische Analyse kommunizieren über einen Puffer.
- *Also*: Phasen sind wie UNIX-*pipes*

  über = lex <*source* | syn | con | opt | code | peep >*target*

Pässe

- Pässe laufen strikt nacheinander ab
  - Zwischensprachen werden in Dateien geschrieben
  - Aus den Zeiten der Speicherknappheit

Zwischensprachen

- Zwischensprachen müssen nicht unbedingt Texte sein
- Programm -Darstellung in der für die Phase günstigsten Form
  - Ketten: Zeichen, Lexeme, Insruktionen
  - Bäume: Syntaxbaum
  - Graphen: Syntaxgraphen, Flussgraphen
  - Felder: Code
- Zwischensprachen müssen nicht vollständig sequenziell konstruiert werden
  (siehe Beispiel Lexikalische und syntaktische Analyse)

Tabellen

- Informationen über *Bezeichner* (*identifier*) werden oft und in mehreren Phasen benutzt.
- Sie werden in Tabellen abgespeichert
- Das soll helfen, diese Information schnell wiederzufinden
- Für jede Phase der Übersetzung sind andere Eigenschaften der Bezeichner relevant:

  lexikalische Analyse Repräsentation (*String Table*)
  kontextuelle Analyse Vereinbarungen(*Declaration Table*)
  Codeerzeugung Adressen (*Adresstabelle*)

lange Einführung (7. April 2009)

1. Organisation
2. Einführung

Zusammenfassung

- Programmiersprachen werden mit *Interpretern* und *Übersetzern* implementiert
- Portierung ist teuer; dafür gibt es ausgefeilte Techniken wie *backtracking*
- Übersetzer werden in Phasen aufgeteilt
- TETRIS ist von Übersetzerbauern erfunden worden

Nachlese(n)

A. V. Aho, R. Sethi, J. D. Ullman.
Compilerbau.
Bonn: Addison-Wesley, 1988.
Band 1, Kapitel 1 beschreibt die *Phasenstruktur*.

D. A. Watt.
Programming Language Processors.
Wokingham: Prentice-Hall, 1993.
Kapitel 2 beschreibt *T-Diagramme* und *bootstrapping*.

Nächstes Mal

- Lexikalische Analyse (Anfang)
  - Lexeme in Programmiersprachen
  - reguläre Definitionen (reguläre Grammatiken)
  - endliche Automaten