1 Einführung [1]

1.1 Einordnung Compilerbau [2]

1.1.1 Teilgebiet der Informatik [3]
- verhältnismäßig alt, Anfänge in den 1960er Jahren
- gut erforscht
- beschäftigt sich mit Grundsatzproblem der Informatik => unverändert aktuell
- reicher Erfahrungsschatz, solides theoretisches Fundament

1.1.2 Grundproblem [4]
- Analyse und Transformation von formalem Text
- Analyse: Erkennen der (hierarchischen) Textstruktur
- Abbildung in interne Baumdarstellung
- Transformation des Baumes in neue, andere Repräsentation

1.1.3 Black-Box-Modell [5]
- Klassischer Programmiersprachen-Compiler als black box

\[\text{Quelltext} \rightarrow \text{Compiler} \rightarrow \text{Code}\]

- Input: Quelltext (source)
- Output: Code (native code, executable, binary, target, ...)
- Nebenprodukte: Meldungen, Hinweise, Warnungen, Fehler, Protokolldaten

1.1.4 Transformator [6]
- Übersetzen eines Textes zwischen zwei verschiedenen Sprachen: Quellsprache => Zielsprache
- „Sprachen“ hier formale Sprachen: eindeutige
  - Syntax (= Rechtschreibung) und
  - Semantik (= Bedeutung und Wechselwirkung der Sprachkonstrukte),
  kein Interpretationsspielraum
- Quellsprache: Programmiersprache (tausende verschiedene zzgl. Dialekte)
- Zielsprache: MaschinenSprache, so vielfältig wie Prozessoren (real oder virtuell)
- Meist Abstraktionsniveau Quellsprache => Zielsprache
- Compiler transformiert Sprachkonstrukte: wenige komplexe => viele einfache

1.1.5 Werkzeugkette [7]
- Isolierter Compiler wenig nützlich, nur ein Werkzeug unter vielen
- Werkzeugkette

<table>
<thead>
<tr>
<th>Werkzeug</th>
<th>Output (= Input für nächstes Werkzeug)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editor</td>
<td>Quelltext</td>
</tr>
<tr>
<td>Compiler</td>
<td>Objektc ode</td>
</tr>
<tr>
<td>Linker (Binder)</td>
<td>Executable</td>
</tr>
</tbody>
</table>
Loader: Ausführbares Abbild des Programms im Hauptspeicher
Debugger, Profiler, Profile, Traces, etc.

- Organisationsform zwischen
  - integriert (integrated development environment, IDE) und
  - Werkzeugkasten (toolbox)

- Werkzeuge der Kette abgestimmt (bspweise Datenformate)

1.1.6 Erscheinungsformen [8]
- Programmiersprachenübersetzer (GNU-Compiler gcc, Java compiler javac)
- Interpreter (HTML-Interpreter in Webbrowsern)
- Pretty Printer, Formatierer
- Static Checker (validating XML parser)
- Dialogsprachen (Unix-Shell)
- Kommandosprachen (SQL-Frontend für DBMS)
- Struktureditor (Eclipse, Netbeans)
- Dokumentationsgenerator (javadoc im Java SDK)

1.1.7 Verwandte Aufgaben [9]
- Entwicklung von Compilern betrifft auch ...
  - Programmiersprachen
  - Bibliotheken
  - Laufzeitsysteme
  - Entwicklungswerkzeuge (siehe Werkzeugkette)
  - Entwicklungsumgebungen (ergonomische Aspekte)

1.2 Komponenten eines Compilers [10]

1.2.1 Compilierschritte [11]
- Zweiteilung: Analyse (Frontend) + Synthese (Backend)
- Zwischendarstellung: AST (abstract syntax tree, abstrakter Syntaxbaum)
  
  \[
  \text{abstract}
  \]
  keine redundanten Informationen

  \[
  \text{syntax}
  \]
  repräsentiert Quelltext

  \[
  \text{tree}
  \]
  hierarchisch organisiert (Knoten, Kanten, ...)

- Frontend schreibt AST, Backend liest AST
- Schema:

![Compiler Diagram](image)

- AST enthält alle relevanten Informationen über Quelltext

1.2.2 Beispiel [13]
Quelltext: Imperative Programmiersprache (Variablen bereits definiert):

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

AST:

```
assignment

  identifier
    position

  add

  identifier
    initial

  mult
```

Code: Konkrete Form abhängig vom Zielsystem

Beispiel JVM, Java-Bytecode:

```
  iload_1
  bipush 60
  iload_2
  imul
  iadd
  istore_3
```

Abbildung der Variablen auf Speicherzellen:

```
1 initial  I
2 rate     I
3 position I
```

1.2.3 Phasen [14]

- Feinere Aufteilung von Analyse und Synthese in Phasen:
  
  **Frontend** (Analyse)
  
  - Lexikalische (= lineare) Analyse, **Scanner**
  - Syntaktische (= hierarchische) Analyse, **Parser**
  - Semantische Analyse

  **Backend** (Synthese)
  
  - Zwischencode-Generierung
  - Optimierung
  - Native-Code-Codegenerierung

1.2.4 Phasenfolge [15]

- Übersetzen über Zwischenprodukte (**compiler pipeline**)

<table>
<thead>
<tr>
<th>Schritt</th>
<th>Input</th>
<th>Phase</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontend</td>
<td>Quelltext</td>
<td>Präprozessor</td>
<td>Quelltext</td>
</tr>
<tr>
<td></td>
<td>Quelltext</td>
<td>Scanner</td>
<td>Tokenstrom</td>
</tr>
<tr>
<td></td>
<td>Tokenstrom</td>
<td>Parser</td>
<td>Parsebaum</td>
</tr>
<tr>
<td></td>
<td>Parsebaum</td>
<td>Semantische Analyse</td>
<td>AST</td>
</tr>
<tr>
<td>Backend</td>
<td>AST</td>
<td>Zwischencode-Generierung</td>
<td>Zwischencode</td>
</tr>
<tr>
<td></td>
<td>Zwischencode</td>
<td>Optimierung</td>
<td>Zwischencode, optimiert</td>
</tr>
<tr>
<td></td>
<td>Zwischencode</td>
<td>Native-Code-Generierung</td>
<td>Native Code</td>
</tr>
</tbody>
</table>
1.3 Weitere Themen [16]

1.3.1 Fehlerbehandlung [17]

- Umgang mit fehlerhaften Programmen (Syntax, statische Semantik, nicht Laufzeitfehler)
- Aus der Sicht des Compilers: Die Mehrzahl der übersetzten Programme fehlerhaft = Fehlerbehandlung wichtig
- Ursache und Wirkung eines Fehlers im Quelltext oft weit entfernt

1.3.2 Anforderungen an Compiler [18]

- selbst fehlerfrei
- Quellsprache komplett umsetzen
- inhaltlich zutreffende und prägnante Meldungen
- Fehlerursache lokalisieren, nicht Auswirkung
- Folgefehler erkennen, zusammenfassen oder zurückhalten
- nach Fehlern fortfahren, Rest des Programms analysieren

1.3.3 Formulieren von Fehlermeldungen [19]

- Wünschenswert: angemessene Meldungen bzgl. Inhalt, Länge, Formulierung
- Möglichst aus der Wirkung auf die Ursache eines Fehlers schließen und diese melden
- Unbrauchbare Extreme:
  - Abbruch beim ersten Fehler
  - Automatische Korrektur und Fortsetzung

- Kein formales Verfahren zur Fehlerbehandlung

1.3.4 Bibliotheken [20]

- Isolierte „Übersetzten“ nicht brauchbar
- Compilerhersteller muss Bibliothek mitliefern
- Komplexität einzelner Bibliotheksfunctionen sehr unterschiedlich (Berechnung des Absolutwertes einer ganzen Zahl, Allocieren eines Objektes in einer OO-Sprache)
- Aufwand zur Entwicklung einer Bibliothek größer als Konstruktion des bloßen Compilers
- Umfang von Bibliotheken

(C) R. Schiedermeier, rs@cs.hm.edu 2017-03-12 09:15
1.3.5 Laufzeitsysteme [21]
- Zur Laufzeit viele Aufgaben im Hintergrund, beispielsweise:
  - Dynamische Speicherverwaltung (garbage collector)
  - Umgang mit Peripheriegeräten (I/O)
  - Initialisierung und Beenden des Programms (Einordnung in das Prozesssystem des Betriebssystems)
  - Dynamisches Binden von Methoden in OO-Sprachen
  - Abwicklung von Exceptions
  - Nebenläufigkeit (threads)

- Laufzeitsystem (runtime system, RTS)
- Teil einer Compilerinstallation, nicht explizit sichtbar
- Teils aufwendige Algorithmen (garbage collection, I/O, thread scheduling),
- Hohe Anforderungen (Korrektheit, Effizienz, Flexibilität)

1.3.6 Universelle Zwischensprache [22]
- Beobachtung:
  Frontend
  hängt komplett von der Sprache ab, kaum von der Maschine

  Backend
  hängt wenig von der Sprache ab, überwiegend von der Maschine

- Idee: Universeller Zwischencode für beliebige Sprachen und beliebige Maschinen
  - Frontend übersetzt Sprache → Universeller Zwischencode
  - Backend übersetzt Universellen Zwischencode → Maschinencode

- Unabhängige Front- und Backends
- Aber: Universeller Zwischencode muss sehr unterschiedlichen Sprachen und sehr unterschiedlichen Maschinen gerecht werden
- Dilemma: Mächtigkeit des universellen Zwischencodes?
  - Spezifische Eigenschaften einzelner Sprachen und Prozessoren werden nicht genutzt → Im konkreten Einzelfall langsamer, ineffizienter Code
  - Vereinigung der Features
    - Spezifische Eigenschaften aller Sprachen und Prozessoren werden berücksichtigt → voluminöse Frontends und Backends, unnötiger Ballast für die allermeisten Fälle

- Heute in Ansätzen verwirklicht für ...
  - ähnliche Sprachen (GNU-Compiler: C, C++, Objective-C, Java → GCC-IL),
  - populaire Architektur (JVM: Java, Scala, Clojure, Groovy ... → Java-Bytecode, Programming languages for the JVM),
  - einzelnes Betriebssystem (Windows: C#, Basic, ... → CIL)

1.3.7 Compiler-Compiler [23]
- Idee: Quellsprache und Maschine formal beschreiben = Compiler automatisch generieren
- Programm zum Generieren aus formalen Beschreibungen: „Compiler-Compiler“
- Gelingt gut für Frontends, weniger für Backends
• Beispiele
  
  **lex, yacc**
  Scanner und Parser (Standard auf Unixsystemen), fixiert auf C/C++

  **flex, bison**
  Weiterentwickelte Nachfolger von lex und yacc, fixiert auf C/C++

  **JavaCC, Jay**
  Scanner und Parser für Java

  • [Catalog of Compiler Construction Products](#)
Funktionstabelle (mit Funktionsergebnis)
Funktionstabelle (mit Funktionsparametern)
Funktionstabelle
Gültigkeitsbereich
Goto (Instruktion)
Grammatik (Mini-lang)
Instruktionen (Zwischencode)
JumpXX, bedingte Sprünge (Instruktion)
Kettenproduktion
Kommentare (Mini-lang)
Kontext-freie Grammatik
Kontext-sensitive Grammatik
Kontrollstrukturen (Mini-lang)
L-value
LL-Eigenschaft
LL-Parser, Algorithmus
LL-Parser
LR-Parser
Label (Instruktion)
Labelstack
Layout (Mini-lang)
Linksableitung
Linksfaaktorisierung
Linksrekursion (Grammatik)
Linksrekursion, indirekt (Grammatik)
Linksrekursive Grammatik
Load (Instruktion)
Loads (Instruktion)
Logische Operatoren (Mini-lang)
Logische Operatoren
Lookahead-Symbol
Maximum-Munch
Mehrdeutige Grammatik
Mehrdeutigkeiten (Scanner)
Mini-lang, AST
Mini-lang, Grammatik
Modulus-Operator
NFA, Konstruktion
NFA, Merkmale
NFA
Namensbindung
Native-Code
Parameter (Mini-lang)
Parameter, Begriffsdefinition
Parameter, Initialisierung
Parameterübergabe
Parsebaum
Parsermatrix, LL-Parser
Popjump (Instruktion)
Print (Instruktion)
Push (Instruktion)
Pushl (Instruktion)
R-value
Rückgabewert (Mini-lang)
Rücksprung (aus Funktion)
Rechtsableitung
Rechtsrekursive Grammatik
Recursive-Descent-Parser
Reduzierte Grammatik
Reguläre Ausdrücke
Reguläre Grammatik
Rekursion
Relationale Operatoren
Return-Anweisung
S-attribuierte Grammatik
Satz (formale Sprache)
Satzform (formale Sprache)
Scope
Semantische Regeln
Stackframe (mit Funktionsergebnis)
Stackframe (mit Funktionsparametern)
Stackframe (mit lokalen Variablen)
Stackmaschine
Startzustand (Automat)
Statische Namensbindung
Statische Semantik
Store (Instruktion)
Stores (Instruktion)
Symbol (formale Sprache)
Symbolstack, LL-Parse
Symboltable, lokal
Symboltable
Syntaxbaum (vs. Parsebaum)
Tabellengesteuerter LL-Parse
Teilweise Auswertung
Terminal-Funktion (Recursive-Descent-Parse)
Thompson's Konstruktion
Token
Tokenattribut
Tokenentyp
Top-Down-Parse
Transition (Automat)
Transitionsliste
Transitionsmatrix
Treibprogramm, LL-Parse
Übergang (Automat)
Übergangsmatrix
Überscharen
Umfassungen, Grammatik
Unäre Minus
Universalparser
Variablen, global (Minilang)
Variablen, lokal (Minilang)
Vergleichsoperatoren
Vollständige Grammatik
Wahrheitswerte
Wort (formale Sprache)
Wortproblem
Zustand (Automat)
Zwischencode
Zyklusfreie Grammatik
1.2.1 Compilierschritte
1.2.2 Beispiel
1.2.3 Phasen
1.2.4 Phasenfolge

1.3 Weitere Themen
1.3.1 Fehlerbehandlung
1.3.2 Anforderungen an Compiler
1.3.3 Formulieren von Fehlermeldungen
1.3.4 Bibliotheken
1.3.5 Laufzeitsysteme
1.3.6 Universelle Zwischensprache
1.3.7 Compiler-Compiler

Stichwortverzeichnis