Compiler

Ausarbeitung zum Proseminar
„C-Programmierung - Grundlagen und Konzepte"

Torsten Zühlke
9zuehlke@informatik.uni-hamburg.de

30. September 2011

Compiler sind Programme, die Eingaben aus einer Quellsprache in eine Zielsprache überführen. Hauptsächlich kennt man Sie als nicht mehr wegzudenkendes Hilfsmittel in der Softwareentwicklung. Dort überführen Sie Programme die in einer oder mehreren Hochsprachen entwickelte Programme in Maschinencode für verschiedenene Zielsysteme.

Bei Compilern handelt es sich jedoch darüber hinaus auch um alle anderen Programme, die eine in einer Quellsprache gegebene Eingabe in eine semantisch äquivalente Ausgabe in einer Zielsprache überführen, und damit veredeln.

Diese Arbeit behandelt hauptsächlich solche Compiler, die in einer Hochsprache geschriebene Programme in Maschinencode überführen.

Im 1. Abschnitt wird eine Definition für Compiler gezeigt und ihre Nutzung motiviert.

Im 2. Abschnitt wird sich genauer mit dem Übersetzungsvorgang auseinandergesetzt und die einzelnen Phasen der Übersetzung betrachtet.

Im 3. Abschnitt werden drei Möglichkeiten Compiler zu klassifizieren betrachtet.

Inhaltsverzeichnis

1 Grundlagen **3**
- 1.1 Motivation zur Nutzung von Compilern . . . 3

2 Compilerphasen **4**
- 2.1 Analysephase . . . 4
  - 2.1.1 Lexikalische Analyse . . . 5
  - 2.1.2 Syntaktische Analyse . . . 6
  - 2.1.3 Semantische Analyse . . . 6
- 2.2 Synthesephase . . . 7
  - 2.2.1 Zwischencodeerzeugung . . . 7
  - 2.2.2 Programmoptimierung . . . 8
  - 2.2.3 Codeerzeugung . . . 10
- 2.3 Alternatives 3-Phasen-Modell . . . 11

3 Klassifikationen von Compilern **12**
- 3.1 Compiler . . . 12
- 3.2 Interpreter . . . 12
- 3.3 Compreter . . . 12
- 3.4 Just-In-Time-Compiler (JIT-Compiler) . . . 12
- 3.5 Weitere Klassifikationen . . . 13

4 Zusammenfassung **14**

5 Quellen **15**

1 Grundlagen

Compiler (auch Übersetzer oder Kompilierer) sind Programme, die eine in einer Quellsprache geschriebene Eingabe in eine semantisch äquivalente Ausgabe einer Zielsprache umwandeln.[1]
Hierbei müssen Compiler die Bedeutung der Eingabe beibehalten, und den Quellcode in einer Art veredeln.
Beispiele hierfür sind:

- `tpic` überführt Grafiken aus dem `pic`-Format in LaTeX-Dateien
  Hierbei liegt die Veredelung in der Überführung in ein weiter verbreitetes Format.
- `gcc` überführt Quellcode aus verschiedenen Hochsprachen in Maschinencode
  Hierbei liegt die Verededlung darin, das ein ausführbarer Code erzeugt wird.

Im Folgenden wird sich diese Arbeit nur mit solchen Compilern beschäftigen, die in in einer Hochsprache gegebenen Quellcode in ausführbaren Maschinencode überführen.

1.1 Motivation zur Nutzung von Compilern

Compiler ermöglichen die Programmierung in Hochsprachen (im Gegensatz zur Programmierung in Maschinencode), und damit die Nutzung von Quellcode für mehrere Plattformen. Hieraus resultieren weniger Entwicklungsaufwand, da gleiche Funktionalität nicht für jede Zielplattform einzel entwickelt werden muss und somit schnellere Softwareentwicklung.
Außerdem erkennen Compiler während der Übersetzung Fehler, die eine Ausführung unmöglich machen, was dazu führt, das manche Fehler schneller und sicherer entdeckt werden.

[1] Frei nach: Compiler. `http://de.wikipedia.org/wiki/Compiler` 2011-05-24

2 Compilerphasen

Compiler sind üblicherweise in zwei Phasen aufgeteilt, die in einzelne Schritte unterteilt sind. Die erste Phase, die Analysephase oder auch Frontend, prüft den Quellcode auf Korrektheit. Die zweite Phase ist die Synthesephase (Backend). Sie erzeugt und optimiert den Zielcode.

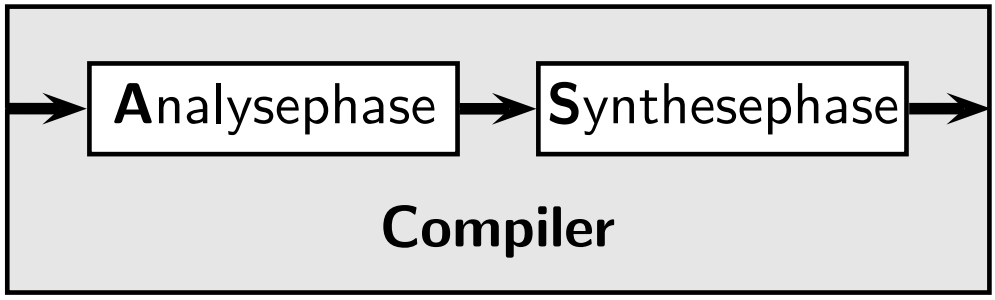

Abbildung 1: Die 2 Compilerphasen

2.1 Analysephase

Die Analysephase besteht aus 3 Schritten:

- Lexikalische Analyse
  Während der lexikalischen Analyse wird der Quellcode in *Tokens*, zusammengehörende Grundeinheiten umgewandelt. Beispiele: Schlüsselwort, Operator, etc.
- Syntaktische Analyse Während der Syntaktischen Analyse wird ein Syntaxbaum erstellt und die Eingabe mit einer formalen Definition der Eingabesprache verglichen.
- Semantische Analsye Hier werden Eigenschaften der benutzen Werte und Typen gesammelt, mit denen u.a. die zulässigkeit der verwendeten Anweisungen überprüft wird.

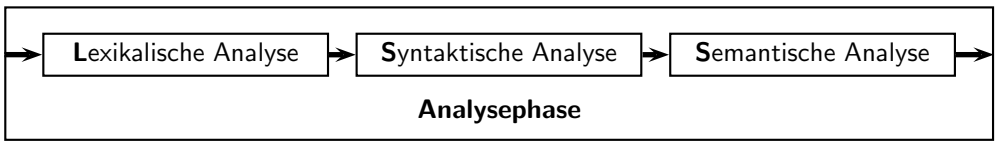

Abbildung 2: Die Analysephase

2.1.1 Lexikalische Analyse

Um die weitere Übersetzung zu ermöglichen, muss der Quellcode zunächst in zusammengehörende Grundeinheiten, sog. Tokens umgewandelt werden. Mögliche Tokens sind z.B.:

- Schlüsselwörter
  - if, for, while, switch, return, ...
- Bezeichner
  - Variablenbezeichner, Funktionsbezeichner, ...
- Typen
  - int, float, double, char, ...
- Konstanten
  - Ganzahlen, Fließkommazahlen, Zeichen, Zeichenketten, ...
- Operatoren
  - +, +=, |, &, ||, &&, ., ->, ...

  Die folgenden zwei Codefragmente zeigen die Umwandlung von Quellcode in eine Tokendarstellung:

```
1  int main(int argc, char *argv[])
2  {
3   int *pointer, value;
4   value = 0;
5   pointer = &value;
6
7   for(int i = 0; i < 5; i++){
8     value += i;
9     printf("%d\n", *pointer);
10  }
11
12  return *pointer;
13 }
```

Abbildung 3: C-Codebeispiel zur Darstellung der Tokenerzeugung

```
1 <Typ, int>, <"main">,<"(">, <Typ, int>, <"argc">, <",">,
      <Typ, char>, <Op, "*">, <"argv">, <Op, []>, <")">
2 <"{">
3 <Typ, int>, <Op, "*">, <"pointer">, <"Value">, <";">
4 <"value">, <Op, "=">, <Ganzzahl, 0>, <";">
5 <"pointer">, <Op, "=">, <Op, "&">, <"value">, <";">
6
7 <Schl.wort, "for">, <"(,">, <Typ, int>, <"i">, <Op, =>,
      <Ganzzahl, 0>, <";">, <"i">, <Op, <>, <Ganzzahl, 5>,
      <";">, <"i">, <Op, ++>, <")">, <"{">
8 <"value">, <Op, +=>, <"i">, <";">
9 <"printf">, <"(">,<String, "%d\n">, <",">, <Op, *>,
      <"pointer">, <")">, <";">
10 <"}">
11
12 <Schl.wort, "return">, <Op, *>, <pointer>, <";">
13 <"}">
```

Abbildung 4: Vereinfachtes Tokenbeispiel zur Darstellung der Tokenerzeugung

2.1.2 Syntaktische Analyse

Während der Syntaktischen Analyse wird an Hand der Tokens aus der lexikalischen Analyse ein Syntaxbaum erstellt. Während der Erstellung wird dieser mittels einer formalen Definition der Eingabesprache auf syntaktische Korrektheit geprüft.
Kann hier kein Syntaxbaum erstellt werden, bricht die Übersetzung hier ab. Hier können Compiler auch versuchen, einen Punkt im Quellcode zu finden, von dem die Erstellung des Syntaxbaumes fortgesetzt werden kann, um möglichst viele Fehler in einem Analysedurchlauf aufdecken zu können.

2.1.3 Semantische Analyse

Um den Quellcode übersetzen zu können werden einige weitere Informationen über die Semantik des Programms benötigt. So muss der Compiler für die genutzten Datentypen ermitteln

- in welchem Format diese gespeichert werden,
- ob und wie das Betriebssystem Speicher zur Verfügung stellt,
- wie der Speicher initialisiert wird,
- wie lange die Werte vorgehalten werden müssen,
- in welchem Speicher die Werte gehalten werden (Stack, Heap),
- wie der Speicher freigegeben wird,

- bei überladenen und überschiebenen Funktion muss die aufzurufende Instanz gewählt werden.

Während der Semantischen Analyse wird auch geprüft, ob alle Variablen bei ihrer Nutzung ausreichend initialisiert sind und ob Anweisungen mit den ihnen übergebenen Typen ausgeführt werden können. Erlaubt die Eingabesprache Vorwärtsdeklarationen, so muss die Semantische Analyse mehrfach ausgeführt werde, bis alle Vorwärtsdeklarationen aufgelöst sind, oder feststeht, dass dies unmöglich ist.

2.2 Synthesephase

Die Synthesephase besteht aus 3 Schritten:

- Zwischencodeerzeugung
  Bei der Zwischencodeerzeugung wird die Eingabe in einen Systemnahen Zwischencode überführt, an dem Optimierungen besser durchzuführen sind.
- Programmoptimierung
  Hier wird der Quellcode in Bezug auf sein Ressourcenverhalten optimiert. Optimierungsziele könne sein:
  - Geringe Anzahl an Hauptspeicherzugriffe
  - Geringer Hauptspeicherbedarf
  - Geringe Anzahl an Anweisungen
  - Gute Nutzung von Wartezeiten auf Ressourcen
- Codeerzeugung
  Bei der Codeerzeugung wird dem optimierten Zwischencode ein äquivalenter Maschinencode zugeordnet. Bei dem erzeugten Zwischencode handelt es sim um das fertige Kompilat.

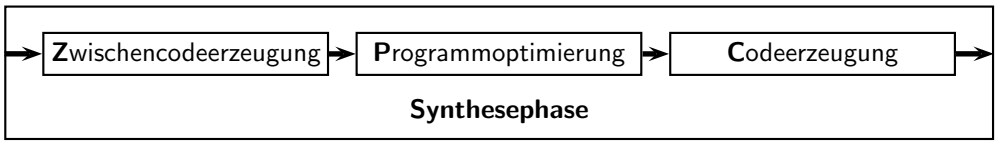

Abbildung 5: Die Synthesephase

2.2.1 Zwischencodeerzeugung

An Hand des, in der Analysephase erzeugten und mit Informationen zur Semantik angereicherten Syntaxbaumes wird die Eingabe in einen Zwischencode überführt.
Der Zwischencode verwendet einen Systemnahen Zwischencode, kennt jedoch keine Speicherplatzbegrenzungen, und kann alle Werte in Registern halten.

Durch die Systemnähe des Zwischencodes können effektive Optimierungen für das Zielsystem hieran gut durchgeführt werden.

$$w \leftarrow x \times 2 \times y \times z$$

`loadAI`	$r_{arp}, @w$	$\Rightarrow r_w$	// load w
`loadI`	2	$\Rightarrow r_2$	// constant 2 into r_2
`loadAI`	$r_{arp}, @x$	$\Rightarrow r_x$	// load x
`loadAI`	$r_{arp}, @y$	$\Rightarrow r_y$	// load y
`loadAI`	$r_{arp}, @z$	$\Rightarrow r_z$	// load z
`mult`	r_w, r_2	$\Rightarrow r_w$	// $r_w \leftarrow w \times 2$
`mult`	r_w, r_x	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2) \times x$
`mult`	r_w, r_y	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2 \times x) \times y$
`mult`	r_w, r_z	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2 \times x \times y) \times z$
`storeAI`	r_w	$\Rightarrow r_{arp}, 0$	// write r_w back to w

Abbildung 6: Beispiel für einen Zwischencode in ILOC

Quelle[2]

2.2.2 Programmoptimierung

Der Zwischencode wird nun in Bezug auf sein Ressourcenverhalten verbessert. Ein optimales Programm zu erzeugen ist sehr unwahrscheinlich, da sich einige Optimierungsziele gegenseitig ausschließen.

It [the compiler] will almost always fail to produce optimal code.[3]

So kann z.B. die Minimierung von Speicherzugriffen z.B. durch das entfernen von Sprunganweisungen dazu führen, das mehr Anweisungen benötigt werden, z.B. Anweisungsstränge mehrfach erzeugt werden.
Mögliche Optimierungen sind:

- Abwickeln von Schleifen
  Durch das Abwickeln von Schleifen werden bedingte Sprunganweisungen entfernt, und somit die Ausführungsgeschwindigkeit erhöht. Insbesondere Schleifen, die eine zur Übersetzungszeit bekannte Anzahl an Durchläufen haben, lassen sich leicht abwickeln.

- Toten Code eliminieren
  Der Compiler kann Code, der nicht ausgeführt werden kann, z.B. in nicht aufgerufenen Funktionen oder bei Abfragen die nur ein Ergebnis liefern können aus dem Zwischencode entfernen.

[2] Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003
[3] Cooper & Torczon: Engineering a Compiler. Morgan Kaufmann 2003

- Neuordnug von Befehlen
  Durch die Fähigkeit moderner Prozessoren, während der Speicherzugriffe andere Befehle auszuführen, macht es häufig Sinn, Speicherzugriffe vorzuziehen oder zu verzögern. Dadurch kann die Zeit, während des Speicherzugriffs für Berechnungen genutzt werden.

- Statische Formelauswertung
  Statische Formeln im Quelltext können durch den Compiler zur Übersetzungszeit ausgewertet, um so Anweisungen zu sparen. So kann z.B. die Berechnung des Kreisumfangs von

```
1 float flaeche = 2 * 3.14 * r;
```

  zu

```
1 float flaeche = 6.28 * r;
```

  ausgewertet werden.

- Erkennung von Konstanten
  In Variablen gehaltene Konstanten, oder solche Variablen, der Wert zur Übersetzungszeit durchgehend bestimmbar sind können vom Compiler konstant in den Maschinencode integriert werden. So werden Anweisungen und benötigte Register eingespart.

- Einfügen von Unterprogrammen
  Unterprogramme, z.B. Funktionen, die nur an wenigen Stellen aufgerufen werden können fest an der aufrufenden Stelle eingefügt werden. So wird die Verwaltung eines eigenen Stackframe für das Unterprogrammes vermieden.

- Verwendung schnellerer Anweisungen
  Für einige Anweisungen bestehen schnellere Äquivalente. So benötigt die Multiplikation auf manchen Plattformen mehrere Prozessortakte, eine Multiplikation mit einer Zweierpotenz kann aber durch Left-Shift-Operationen ausgedrückt werden, die üblicherweise nur einen Prozessortakt benötigen.

- Paging verhindern
  Vor Schleifen kann es Sinn machen Leeranweisungen einzufügen, damit der gesamte Schleifenrumpf in einer Speicherseite liegen. So muss während der Ausführung des Schleifenrumpfs keine Speicherseite nachgeladen werden.
  Außerdem macht es Sinn Variablen nicht willkürlich im Speicher zu platzieren, solche die zusammen benötigt werden in der selben Speicherseite zu halten. So kann es auch hier Sinn machen Teile von Speicherseiten leer zu lassen um seltener Speicherseiten nachladen zu müssen.

2.2.3 Codeerzeugung

Bei der Codeerzeugung müssen hauptsächlich Entscheidungen über die Verwendung der (begrenzt vorhandenen) Register und der Speicherstellen getroffen werden. Dies hängt von der Anzahl und der Art der vorhandenen Register ab. So kann es sein, das der Prozessor spezielle Register für String-Operationen bietet. Solche sollten dann bei String-Operationen auch benutzt werden, um die Fähigkeiten des Prozessors auszunutzen. Der Compiler hält alle Variablen bevorzugt in Registern, da diese schneller zugreifbar sind. Da hierfür meißtens nicht ausreichend Register vorhanden sind, werden Variablen, die seltener genutzt werden, in den Hauptspeicher ausgelagert. Außerdem können einige Variablen von anderen Prozessen oder Peripheriekomponenten abhängig sein. Solche Variablen können nicht in Registern gehalten werden, da sie ansonsten nicht aktualisiert würden, und es zu Inkonsistenzen käme.

So ergibt sich für den Compiler ein Aufbau aus 2 Phasen mit jeweils 3 Teilschritten:

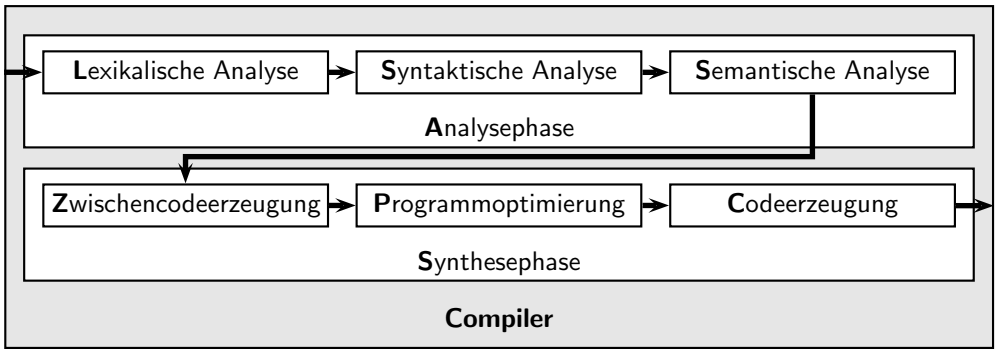

Abbildung 7: Die 2 Compilerphasen mit ihren Teilschritten

2.3 Alternatives 3-Phasen-Modell

Bei einem Alternativen Modell mit 3 Phasen wird die Optimierung aufgeteilt. Systemunabhängige Optimierungen werden am Syntaxbaum durchgeführt. So kann z.B. Toter Code bereits an dieser Stelle eliminiert werden.

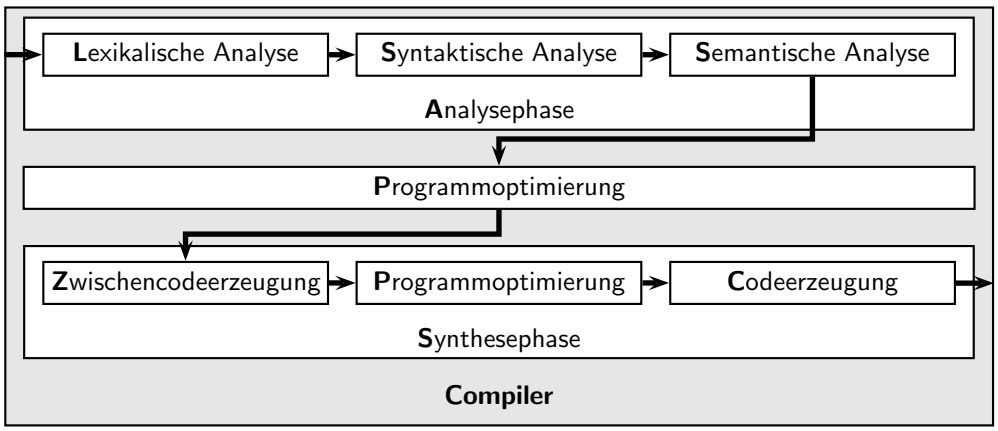

Abbildung 8: Alternative Implementation mit 3 Phasen

3 Klassifikationen von Compilern

Es gibt verschiedene Arten von Compilern, die danach eingeteilt sind, wie die Erzeugung des Programmcodes zur Ausführung in Bezug steht.

3.1 Compiler

Als Compiler bezeichnet man solche Programme, bei denen das Programm unabhängig von seiner Übersetzung und ohne Kenntnis des Compilers später ausgeführt werden.
Sie haben den Vorteil, dass zur Ausführung keine zusätzlichen Programme benötigt werden. Die erzeugten Programme funktionieren dafür meißtens nur auf einer Plattform.

3.2 Interpreter

Interpreter könnte man als Gegenteil von Compilern bezeichnen. Hier wird der Quellcode zur Ausführungszeit Anweisung für Anweisung ausgewertet um jeweils entsprechende Prozessorbefehle auszuführen.
Beim Einsatz von Interpretern kann der Programmcode auf jeder Plattform ausgeführt werden, für die entsprechende Interpreter zur Verfügung stehen. Nachteilig wirkt sich die Nutzung von Interpretern auf die Ausführungsgeschwindigkeit aus, da der Quellcode zur Ausführungszeit geparst werden muss und nicht optimiert werden kann.

3.3 Compreter

Compreter sind zwischen Compilern und Interpretern einzuordnen und versuchen die Vorteile beider Formen in sich zu vereinen. Sie übersetzen den Quellcode in einer schneller Interpretierbaren Zwischencode, dieser wird zur Ausführungszeit interpretiert.
Durch die Nutzung eines schneller interpretierbaren Zwischencode kann die Ausführungsgeschwindigkeit im Gegensatz zum Interpreter deutlich erhöht werden, während die Plattformunabhängigkeit erhalten bleibt.

3.4 Just-In-Time-Compiler (JIT-Compiler)

JIT-Compiler Compilieren Auszüge des Quellcodes (z.B. einzelne Klassen) ad hoc und führen den kompilierten Code dann aus.
Teilweise wird auch erst ein Zwischencode erzeugt, der zur Ausführungszeit ad hoc kompiliert wird. Hierbei wird die Plattformunabhängigkeit ebenfalls erhalten, während häufig ausgeführte Codestücke schneller ausgeführt werden kann als bei einem Compreter.

Die tatsächlichen Ausführungsgeschwindigkeiten hängen jedoch u.a. von der Art des Quellprogramms und der Implementation des Compilers ab.

3.5 Weitere Klassifikationen

Des Weiteren kann man Compiler danach klassifizieren, für welche Plattform der Zielcode erzeugt wird und wie der Übersetzungsvorgang abläuft.

Native Compiler

Native Compiler erzeugen Zielcode für die Plattform, auf der der Compiler ausgeführt wird.

Crosscompiler

Crosscompiler erzeugen Zielcode für eine andere Plattform als die, auf der der Compiler ausgeführt wird. Solche Compiler findet man u.A. bei der Entwicklung von Software für eingebettete Systeme, da es häufig nicht effizient ist Compiler auf den verwendeten Plattformen auszuführen bzw. sie für diese zu entwickeln.

Transcompiler (Transpiler)

Transcompiler übersetzen Code aus einer Hochsprache in eine andere Hochsprache.

Single-Pass-Compiler

Single-Pass-Compiler erzeugen den Zielcode in einem Compilerdurchlauf.

Multi-Pass-Compiler

Multi-Pass-Compiler erzeugen den Zielcode in mehreren Compilerdurchläufen. Diese Technik wird eingesetzt, damit nicht der gesamte Compiler im Speicher gehalten werden muss. Außerdem ist so die Modularisierung des Compilers einfacher.

4 Zusammenfassung

Compiler sind Programme, die eine Eingabe in eine semantisch äquivalente Ausgabe überführen. Es gibt sie nicht nur die Übersetzung von Software in Maschinencode, sondern auch für andere Einsatzzwecke. Bei der Nutzung in der Softwareentwicklung sind sie ein hilfreiches Werkzeug, das Fehler findet und die Softwareentwicklung vereinfacht.
Die Übersetzung von Software aus einer Hochsprache in Maschinencode erfolgt in 2 Phase mit jeweils 3 Schritten.
In der ersten Phase, der Analysephase wird das Programm auf syntaktische Korrektheitgeprüft. Außerdem wird überprüft ob es einer gültigen Semantik folgt. Die Semantische Korrektheit kann mit Compilern jedoch nicht überprüft werden. Hierfür wird der Code in mehrere Zwischenformen überführt, die in späteren Schritten weiterverwendet werden.
In der Synthesephase wird ein Zwischencode erzeugt, dessen Ressourcenverhalten verbessert wird um dann in Maschinencode überführt zu werden. Insbesondere die Optimierungsprobleme sind meißtens nicht optimal lösbar, da sich einige Optimierungen gegenseitig beeinflussen.
Alternativ zum 2-Phasen-Modell gibt es auch ein Modell mit 3 Phasen, bei dem zwischen Analysephase und Synthesephase eine Optimierungsphase sitzt, in der einige der Optimierungen, die unabhängig vom Zielsystem sind, durchgeführt werden.
Compiler können nach verschiedenen Eigenschaften klassifiziert werden. Unter anderem sortiert man danach, ob der Compiler Code für die eigene Plattform oder für andere Plattformen erzeugt. Eine andere Klassifikation setzt den Übersetzungszeitpunkt mit dem Ausführungszeitpunkt in Vezug.

5 Quellen

Literatur

[1] ANSI C Grammar(Yacc). http://www.lysator.liu.se/c/ANSI-C-grammar-y.html. 2011-05-26.

[2] Compiler. http://de.wikipedia.org/wiki/Compiler. 2011-05-24.

[3] Erweiterte Backus-Naur-Form. http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form. 2011-05-26.

[4] Formal grammar. http://en.wikipedia.org/wiki/Formal_grammar. 2011-05-26.

[5] Formale Grammatik. http://de.wikipedia.org/wiki/Formale_Grammatik. 2011-05-24.

[6] Formale Sprache. http://de.wikipedia.org/wiki/Formale_Sprache. 2011-05-24.

[7] GNU compiler collection. http://en.wikipedia.org/wiki/GNU_Compiler_Collection. 2011-05-24.

[8] Keith Cooper and Linda Torczon. *Engineering a Compiler: International Student Edition*. Morgan Kaufmann, I.S.ed edition, 12 2003.

[9] Christoph Habel and Matthias Jantzen. FGI-1: Formale Grundlagen der Informatik. Universität Hamburg, SoSe 2010.