C-Präprozessor & Compiler

— Proseminar: C - Grundlagen und Konzepte —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Jan Niklas Hennings
E-Mail-Adresse:	3henning@informatik.uni-hamburg.de
Matrikelnummer:	6536984
Studiengang:	Informatik B.Sc.

Ansprechpartner:	Michaela Zimmer
Betreuer:	Konstantinos Chasapis

Hamburg, den 24.04.2014

Inhaltsverzeichnis

1. Einleitung

Im Gegensatz zur Programmiersprache **Java** wird in **C** ein grundlegendes Verständnis über die Aufgaben des Präprozessors und eines Compilers erwartet. Nur so lässt sich effizienter, funktionierender Code schreiben und dieser in einen ausführbaren Zustand überführen (*"kompilieren"*).

Ein Verständnis über die Funktionsweise des Präprozessors ist besonders wichtig zum Schreiben und Verstehen von Quellcode.

Da **C** im Gegensatz zu **Java** keine *Virtual Machine* zur Ausführung von Software nutzt, sondern direkt in Maschinencode übersetzt wird, ist auch der Vorgang des Kompilierens essentiell. Denn eine einfache Portierung über verschiedene Betriebssysteme ist somit nicht ohne weiteres möglich.

Außerdem existieren viele verschiedene Compiler für **C**-Code, welche unterschiedlich arbeiten und jeweils andere Funktionen zur Beeinflussung der Übersetzung bieten. Man kann somit allein durch den Prozess des Kompilierens direkten Einfluss auf das endgültige Programm ausüben.

2. C-Präprozessor

2.1. Der Präprozessor

Der Präprozessor stellt die erste Instanz vor dem eigentlichen Kompilieren in **C** dar. Musste früher ein alleinstehendes Programm für die Aufgaben des Präprozessors vor dem Kompilieren genutzt werden, so ist der Präprozessor heute ein fester Bestandteil der meisten Compiler.

Einfach ausgedrückt durchsucht der Präprozessor den Quelltext und ersetzt Teile des Codes nach bestimmten Vorgaben zur Aufbereitung für den Kompilierungsvorgang ("Suchen und Ersetzen").

Es ist ein sehr mächtiges, jedoch fehleranfälliges Tool in der Händen unerfahrener Anwender. Denn der Präprozessor kann nicht garantieren, dass der aufbereitete Code sinnvoll oder richtig ist!

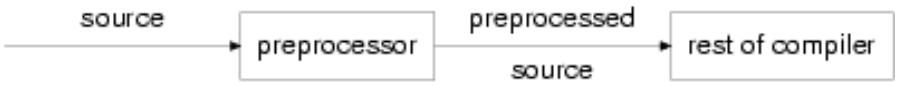

Abbildung 2.1.: "The preprocessor" [4]

2.2. Elementare Aufgaben

Eine einfache Aufgabe des Präprozessors besteht darin, Kommentare zu entfernen, da diese für den Compiler irrelevant sind und die Größe des Quellcodes verringern.

2.3. #include

Listing 2.1: Hello World

```
1 #include<stdio.h>
2
3 int main() {
4     printf("Hello World\n");
5     return 0;
6 }
```

Bereits ein einfaches *Hello World*-Beispiel benötigt das Einbinden (engl. *(to) include*) von sog. *Header Files* (erkennbar an der Dateiendung *.h*).

Mit dem Befehl

```
#include<dateiname>
```

ersetzt der Präprozessor die Codezeile durch den gesamten Inhalt des *Header Files*. Viele solcher Header Files finden sich in der Standardbibliothek jedes **C**-Compilers.

Im *Hello World*-Beispiel werden durch *stdio.h* viele Funktionen für die Standard-Ein- und Ausgabe (`printf()`) eingebunden.

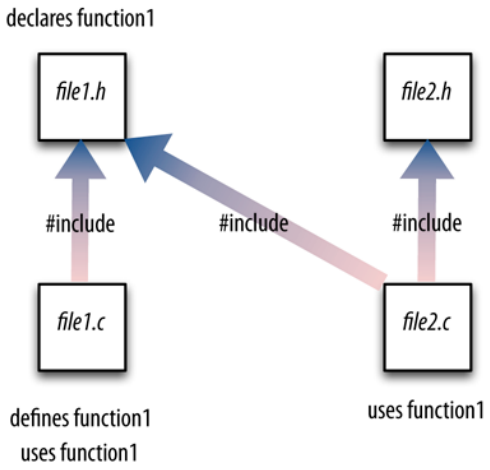

Abbildung 2.2.: "How a large C program is divided into files" [14]

2.4. Makros

Mit der Anweisung

```
#define BEZEICHNER wert
```

lässt sich ein `BEZEICHNER` für den C-Präprozessor definieren, welcher bei der Verarbeitung im Quelltext durch `wert` ersetzt wird. Hierbei ist jedoch zu beachten, dass keine Variable initialisiert wird, sondern lediglich eine *symbolische Konstante*, welche bei jedem Vorkommen von `BEZEICHNER` vom Präprozessor durch `wert` ersetzt wird. Dies ist ein häufiges Missverständnis bei unerfahrenen Programmierern!

Soll ab einer bestimmten Zeile im Code `BEZEICHNER` nicht mehr durch `wert` ersetzt werden, lässt sich die Definition rückgängig machen:

```
#undef BEZEICHNER
```

Listing 2.2: Quelltext

```
1 #define PI 3.1415926f
2
3 float umfang(float d) {
4     return PI * d;
5 }
```

Listing 2.3: nach Präprozessor-Aufruf

```
1 float umfang(float d) {
2     return 3.1415926f * d;
3 }
```

Beispiele folgen teilweise: *"C von A bis Z"* [18]

2.4.1. parameterbehaftete Makros

Durch die Definition

```
#define BEZEICHNER(Args) ausdruck
```

lassen sich auch Parameter `Args` innerhalb von `ausdruck` ersetzen und verarbeiten.

Listing 2.4: parameterbehaftes Makro

```
1 #define QUADRIEREN(x) ( ( x ) * ( x ) )
```

Grundsätzlich sollte auf defensive Klammerung geachtet werden, um zum Beispiel Fehler durch Punktrechnung vor Strichrechnung zu vermeiden:

Listing 2.5: parameterbehaftes Makro

```
1 #define TWOTIMES(x)  x+x
2
3 int calc(int d) {
4     return (TWOTIMES(d) * 2)
5 }
```

Die Methode `calc(int)` liefert für den aktuellen Parameter **5** die Rückgabe **15** und nicht etwa **20**, wie zu erwarten wäre. Nach dem preprocessing enthält die return-Anweisung lediglich `return (5+5 * 2)`.

2.5. bedingte Kompilierung

Das Konstrukt

```
#if ausdruck
    ...
#endif
```

schließt einen Quelltextblock ein, welcher je nach erfüllter `#if`-Bedingung `ausdruck` dem Compiler im verarbeiteten Quellcode übergeben wird.

Weiterhin existiert der Ausdruck `#ifdef BEZEICHNER` um zu überprüfen, ob bisher ein `BEZEICHNER` definiert wurde. Zusätzlich sind noch die Anweisungen `#elif` und `#else` vorhanden, welche sich wie die bekannten `else if` und `else` Anweisungen verhalten.

Somit lässt sich zum Beispiel die Portierbarkeit zwischen unterschiedlichen Systemen gewährleisten, ohne aufwendige Änderungen am Code vornehmen zu müssen:

Listing 2.6: Portierung durch #if-Bedingung

```
1 #ifdef WINDOWS
2     printf("Programm läuft unter Windows\n");
3 #elif UNIX || LINUX
4     printf("Programm läuft unter UNIX/LINUX\n");
5 #else
6     printf("Unbekanntes Betriebssystem!\n");
7 #endif
```

Beispiel folgt: *"C-Buch"* [17]

In diesem Beispiel wird auf der Konsole in Abhängigkeit der Definition von einer *symbolischen Konstanten* eine System-spezifische Nachricht ausgegeben. Um dieses Programm auf unterschiedliche Betriebssysteme zur portieren muss bei einem erneuten Compiler-Aufruf lediglich eine Definition geändert werden, um eine andere Ausgabe zu erzeugen.

3. Compiler

3.1. Allgemeines

Ein C-Compiler erhält den vom Präprozessor bereits vollständig vorbereiteten Quellcode und übersetzt diesen aus der Hochsprache in einen semantisch äquivalenten Maschinencode.

Dabei werden die zwei vom Compiler ausgeführten Phasen in jeweils drei unterschiedliche Arbeitsschritte unterteilt.

Der erzeugte Maschinencode wird in **C** als sog. *Object-Files* (erkennbar an der Dateiendung *.o*) gespeichert. Diese *Object-Files* werden im letzten Arbeitsschritt vom Linker (siehe 4.1) endgültig zusammengeführt.

3.2. Phasen-Modell

- Analyse-Phase (Front-end)
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
- Synthese-Phase (Back-end)
  - Zwischencode-Erzeugung
  - Programmoptimierung
  - Maschinencode-Erzeugung

Die Beispiele in diesem Abschnitt folgen [9] Ulrich Helmich. *"Was ist ein Compiler?"* URL: `http://www.u-helmich.de/inf/BlueJ/kurs131/Seite25/seite25-1.html` (besucht am 23. 04. 2014)

3.2.1. Analyse-Phase (Front-end)

Lexikalische Analyse

"Rechtschreibkorrektur"

Der Quellcode wird während der *lexikalische Analyse* in sog. *Tokens* zerlegt. *Tokens* sind dabei z.B. ein Bezeichner, Zahlen oder Schlüsselworte wie *if, while* etc. Trifft der Compiler während dieses Vorgangs auf ein Token, welches sich aufgrund der Sprachspezifikation nicht verarbeiten lässt, bricht der Kompilierungsvorgang mit der Ausgabe eines lexikalischen Fehlers ab.

Listing 3.1: Quellcode

```
umfang   =    radius   *   radius   *   3,14   ;
```

Listing 3.2: Tokenzerlegung

```
<umfang> <=>  <radius> <*> <radius> <*> <3,14> <;>
```

Die lexikalische Analyse würde bei diesem Beispiel eine Fehlermeldung ausgeben, da die Real-Zahl "3,14" ein Komma anstelle eines Punktes enthält, was einem lexikalischen Fehler entspricht.

Syntaktische Analyse

Prüfung auf Syntax-Fehler

In dieser Arbeitsphase wird der lexikalisch korrekte Code auf seine syntaktische Korrektheit überprüft. Dazu erfolgt ein *Parsing* der zerlegten linearen Tokens in einen *Syntax-Baum* (Abbildung 3.1).

Listing 3.3: Syntaxanalyse

```
flaeche = pi * r * ;
```

Der vorliegende Ausdruck ist lexikalisch korrekt, da keine ungültigen Tokens genutzt werden. Jedoch bricht der Compiler in dieser Phase den Übersetzungsprozess ab, da auf den zweiten Operator "*" kein Operand folgt und somit die Syntax-Bestimmungen von **C** verletzt wurden.

Semantische Analyse

"Funktioniert das Programm?"

Wenn der Quellcode bisher sowohl lexikalisch als auch syntaktisch korrekt war, muss in einer letzten Phase vor der eigentlichen Übersetzung noch eine Prüfung auf die korrekte *Funktionsweise* des Quelltextes erfolgen. Dabei kann keine Prüfung auf pragmatische Korrektheit erfolgen ("Leistet der Quellcode, wofür er geschrieben wurde?"), sondern lediglich ob der Quelltext über die lexikalischen und syntaktischen Spezifikationen hinaus sinnvoll ist. Dazu sind beispielsweise folgende Schritte notwendig:

- Datentyp-Überprüfung
- *Object binding* (Variablen mit Typen verknüpfen, Methodenaufrufen folgen und verknüpfen)

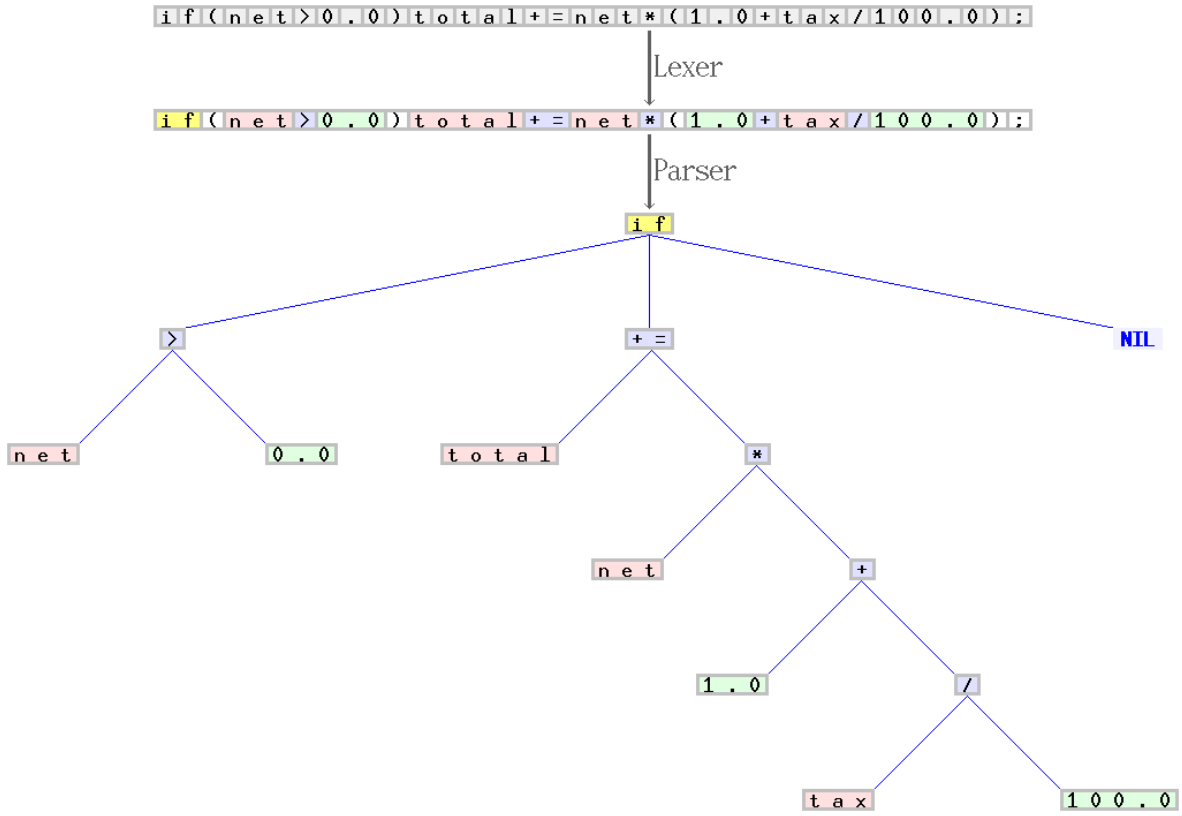

Abbildung 3.1.: ”Scanner and parser example for C” [5]

- Initialisierung aller Variablen prüfen [8]

Ein anschauliches Beispiel:

```
double a = 3 * wurzel(5);
```

Diese Variableninitialisierung ist lexikalisch und syntaktisch korrekt, *könnte* semantisch jedoch inkorrekt sein, bspw. wenn `wurzel()` keinen geeigneten Datentypen als Rückgabewert liefert (z.B. ein String-Literal).

3.2.2. Synthese-Phase (Back-end)

Zwischencode-Erzeugung

Grobe Übersetzung

In dieser Phase wird der Hochsprachen-Quelltext erstmals in eine maschinennahe, semantisch äquivalente Form übersetzt. Der entstandene Zwischencode ist noch nicht ausführbar und dient lediglich als Basis zur Weiterverarbeitung. Dieser Zwischencode zeichnet sich durch folgende Charakteristika aus:

- maschinennahe Repräsentation
- Eine fundamentale Operation pro Zeile
- unbegrenzte Speicheradressierung und Registeranzahl

Als Folge ergibt sich ein sehr langer und unoptimierter Code, welcher durch die unbegrenzte Speicheradressierung und Registeranzahl sehr viel Speicherplatz benötigt.

Listing 3.4: Quelltext

```
p = i + r * 60;
```

Listing 3.5: möglicher Zwischencode

```
1 temp1 := 60
2 temp2 := id3 * temp1
3 temp3 := id2 + temp2
4 id1 := temp3
```

Für die Registerbelegung folgt: $r \Leftrightarrow id3, i \Leftrightarrow id2, p \Leftrightarrow id1$

Zwischencode-Optimierung

Optimierung des Zwischencodes

Während der vorletzte Phase wird abermals ein semantisch äquivalenter, diesmal jedoch schnellerer bzw. kürzerer Zwischencode erzeugt. Dabei kommen viele verschiedene Optimierungsmaßnahmen zum Einsatz:

- toten Code entfernen
- Schleifen abwickeln und Kontrollfluss ändern
- Konstanten auflösen (Code: (14/2), Optimiert: 7)
- ggf. Anweisungen ersetzen (nach: [15])

Die Optimierungsmöglichkeiten von verschiedenen Compilern unterscheiden sich stark! In vielen Fällen kann während des Compiler-Aufrufs sogar der gewünschte Grad der Optimierung bestimmt werden. Dabei müssen Compiler-Entwickler jedoch vorsichtig vorgehen. Denn jeder Eingriff in die Semantik eines Quellcodes könnte die spezifizierte Pragmatik verändern und so selbst bei fehlerfreiem Quellcode am Ende ein Programm liefern, welches nicht die gewünschten Aufgaben erledigt.

Listing 3.6: möglicher Zwischencode

```
1 temp1 := 60
2 temp2 := id3 * temp1
3 temp3 := id2 + temp2
4 id1 := temp3
```

Listing 3.7: Optimierungsmöglichkeit

```
1 temp1 = id3 * 60
2 id1 = id2 + temp1
```

Der Zwischencode wurde auf die Hälfte der Anweisungen reduziert, ohne etwas an der Aussage zu verändern.

Maschinencode-Erzeugung

Maschinensprache als *Object-File*

Im letzten Übersetzungsschritt wird der erzeugte und optimierte Zwischencode endgültig in eine Maschinensprache übersetzt, welche die unterliegende System-Architektur ausführen kann. In dieser Phase können die verbliebenen Arbeitsschritte umgesetzt werden:

- Bestimmung der benötigten Register und Speichergröße
  - Abhängig von: der Architektur des unterliegenden Systems
  - den verwendeten Variablen
- Erzeugung der Maschinen-Befehle für die entsprechende System-Architektur basierend auf dem Zwischencode

Das fertige *Object-File* (.o) ist nun bestmöglich optimiert und nutzt nur so viel Speicher, wie es wirklich benötigt – natürlich lediglich im Rahmen des ursprünglichen Quellcodes.

Listing 3.8: möglicher Zwischencode

```
1 temp1 = id3 * 60
2 id1 = id2 + temp1
```

Listing 3.9: Maschinencode (repräsentiert als Assembler)

```
1 MOV id3,R2
2 MUL 60,R2
3 MOV id2,R1
4 ADD R2,R1
5 MOV R1,id1
```

4. Linker

4.1. Allgemeines

Der *Linker* bindet nach erfolgreichem Kompilieren alle entstandenen *Object-Files* zu einem zentral ausführbaren Programm zusammen. Aufgrund dessen ist auch *Binder* als Bezeichnung geläufig, sowie früher in etwas anderer Funktionsweise auch *Loader*. Ein un-

Abbildung 4.1.: "Separate compilation" [2]

mittelbarer Vorteil ergibt sich daraus, dass kein aufwendiges re-compiling des gesamten Projektes nötig ist, sollte lediglich eine Datei verändert werden (ausgenommen *Header-Files*). Ein erneutes Linken ist wesentlich schneller. [3]

Wir unterscheiden grundsätzlich zwischen zwei verschiedenen Arten des Linkens. In der Praxis haben sich Mischformen dieser beiden Verfahren als Regelfall etabliert, welche die Vorteile beider Herangehensweisen nutzen.

4.1.1. Statisches Linken

Bei *statischem Linken* wird das gesamte Projekt mit all seinen Modulen einmalig zu einer einzelnen ausführbaren Datei gelinked. Sollten einmal Änderungen an einem Modul vorgenommen werden, muss dieser Prozess natürlich wiederholt werden. Dies hat den Vorteil, dass sämtliche Programmmodule fest in der Datei enthalten sind, diese dadurch entsprechend sehr portabel ist und nicht installiert werden muss.

Jedoch benötigt diese Art des Linkens viel Speicherplatz und macht das Projekt unflexibel z.B. gegenüber Änderungen an Bibliotheken, welche fest in die einzelnen Module eingebunden werden und somit bei jeder Änderung erneut in jedes benutzende Modul eingebunden werden müssen. [12]

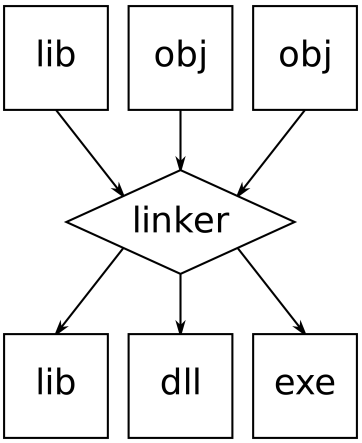

Abbildung 4.2.: "Input and output file types of the linking process." [16]

4.1.2. Dynamisches Linken

Das *dynamische Linken* verbindet einzelne Programmmodule auf eine wesentlich flexiblere Art und Weise, als es beim *Statischen Linken* der Fall ist.

Sog. *dynamically linked libraries* (Abk.: DLL) werden bei jedem Programmstart neu in das Projekt eingebunden. Dies bietet den Vorteil, dass der Speicherplatz für solch eine DLL nur ein einziges Mal benötigt wird und von vielen verschiedenen Programmen und Programmteilen auf diese zugegriffen werden kann. Da eine DLL jedes mal neu gelinked wird, kann diese bei einer Veränderung auch sehr leicht ausgetauscht werden und erhöht so die Flexibilität und Wartbarkeit des Programms.

Problematisch bei diesem Ansatz ist jedoch, dass jedes System über die entsprechenden DLLs verfügen muss. Da benutzte DLLs somit fest installiert sein müssen, sind dynamisch gelinkte Programme nicht mehr sehr portabel.

Literaturverzeichnis

[1] The C Book. *"Effect of the Standard"*. URL: http://publications.gbdirect.co.uk/c_book/chapter7/effect_of_the_standard.html (besucht am 23. 04. 2014).

[2] The C Book. *"Separate compilation"*. URL: http://publications.gbdirect.co.uk/c_book/figures/1.1.png (besucht am 23. 04. 2014).

[3] The C Book. *"The form of a C program"*. URL: http://publications.gbdirect.co.uk/c_book/chapter1/form_of_a_c_program.html (besucht am 23. 04. 2014).

[4] The C Book. *"The preprocessor"*. URL: http://publications.gbdirect.co.uk/c_book/figures/7.1.png (besucht am 23. 04. 2014).

[5] Jochen Burghardt. *"Scanner and parser example for C"*. URL: http://en.wikipedia.org/wiki/File:Xxx_Scanner_and_parser_example_for_C.gif (besucht am 23. 04. 2014).

[6] *"C Programming/Preprocessor"*. URL: http://en.wikibooks.org/wiki/C_Programming/Preprocessor#macros (besucht am 08. 05. 2014).

[7] C-HowTo. *"Dateien einbinden"*. URL: http://www.c-howto.de/tutorial-praeprozessor-dateien-einbinden.html (besucht am 23. 04. 2014).

[8] *"Compiler"*. URL: http://en.wikipedia.org/wiki/Compiler#Front_end (besucht am 23. 04. 2014).

[9] Ulrich Helmich. *"Was ist ein Compiler?"* URL: http://www.u-helmich.de/inf/BlueJ/kurs131/Seite25/seite25-1.html (besucht am 23. 04. 2014).

[10] *"Intel C++ Compiler"*. URL: http://en.wikipedia.org/wiki/Intel_C++_Compiler (besucht am 23. 04. 2014).

[11] *"Intel Developer Zone"*. URL: https://software.intel.com/en-us/non-commercial-software-development (besucht am 23. 04. 2014).

[12] *"Linker (Computerprogramm)"*. URL: http://de.wikipedia.org/wiki/Linker_(Computerprogramm) (besucht am 23. 04. 2014).

[13] *"MinGW"*. URL: http://www.mingw.org/ (besucht am 23. 04. 2014).

[14] Matt Neuburg. *"How a large C program is divided into files"*. URL: http://www.apeth.com/iOSBook/figs/pios_0102.png (besucht am 23. 04. 2014).

[15] *"Optimizing compiler"*. URL: http://en.wikipedia.org/wiki/Compiler_optimization (besucht am 23. 04. 2014).

[16] Qef. *"Input and output file types of the linking process"*. URL: http://commons.wikimedia.org/wiki/File:Linker.svg (besucht am 23. 04. 2014).

[17] Sommergut.de. *"C-Buch"*. URL: http://c-buch.sommergut.de/Kapitel12/Bedingte-Kompilierung.shtml (besucht am 23. 04. 2014).

[18] Jürgen Wolf. *"C von A bis Z"*. URL: http://openbook.galileocomputing.de/c_von_a_bis_z/010_c_praeprozessor_001.htm (besucht am 23. 04. 2014).

Abbildungsverzeichnis

Anhänge

A. Aufgaben

Aufgabe 1)

Listing A.1: HelloWorld.c

```
1 #include<stdio.h>
2 #define HI "Hello World\n"
3
4 int main() {
5     printf(HI);     // Gibt Hello World aus!
6     return 0;
7 }
```

Kompiliere diesen "Hello World"-Quellcode mit folgenden Parametern, um einen ungefähren Eindruck von der Arbeitsweise des Präprozessors zu bekommen.

```
gcc -E HelloWorld.c
```

Aufgabe 2)

Listing A.2: Parameterbehaftetes Makro

```
1  #include <stdio.h>
2
3  #define SLICES 8
4  #define ADD(x) ( (x) / SLICES )
5
6  int main(void)
7  {
8      int a = 0, b = 10, c = 6;
9
10     a = ADD(b + c);
11     printf("%d\n", a);
12     return 0;
13 }
```

Welche Ausgabe erzeugt die main-Methode auf der Konsole? Überprüfen Sie ihre Antwort.[1]

Aufgabe 3)

Beschreiben Sie kurz in eigenen Worten die Unterschiede zwischen dem Front-end und dem Back-end eines Compilers. Geben Sie danach stichwortartig die Bedeutung der einzelnen Arbeitsschritte des Compilers an.

Aufgabe 4)

Listing A.3: Analyse-Phase)

```
1 #define PI 3.1415926f
2
3 float kreisflaeche(flot d) {
4     return PI x d * a;
5 }
```

In dieser kurzen Methode haben sich mehrere Fehler eingeschlichen. Finde diese Fehler und gib an, in welcher Phase des Front-Ends sie vom Compiler entdeckt würden.

[1] Nach: *"C Programming/Preprocessor"* [6]

Aufgabe 5)

Listing A.4: Optimierung

```
1  #include <stdio.h>
2
3  int condense() {
4      int Ary[5] = { 1, 2 };
5      int eins = Ary[0];
6      int zwei = Ary[1];
7      int drei = eins + zwei;
8      return drei;
9  }
10
11 int main() {
12     int i;
13
14     i = condense();
15
16     printf("%d", i);
17     return 0;
18 }
```

Fassen Sie den Code möglichst optimal zusammen, ohne die Ausgabe zu verfälschen. Gehen Sie dafür davon aus, dass die Initialisierung des Arrays variabel und nicht vorgegeben wäre.