

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

BACHELORARBEIT

Abgabedatum: 13. September 2022

Autor:

Jürgen Mattheis

Gutachter:

Prof. Dr. Scholl

Betreuung:

M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Danksagungen

Bevor der Inhalt dieser schriftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie „mein aufrichtigster Dank“ oder „aus tiefstem Herzen“, sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbstverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das hinaus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschaft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias¹ konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe², er hat sich bei der Korrektur dieser schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbstverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

¹Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

²Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil^{3 4} weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbstverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler auseinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter [Link](#)⁵ zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein persönliches Interessengebiet fällt⁶. Das Aufschreiben dieser schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁷. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiersprache kann in der Verwendung ziemlich willkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

³Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

⁴Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes 🤔.

⁵<https://github.com/michel-giehl/Reti-Emulator>.

⁶Womit nicht alle Studenten so viel Glück haben.

⁷Dieses ständige überlegen, wo man möglicherweise eine Erklärücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
1 Einführung	1
1.1 RETI-Architektur	2
1.2 Die Sprache PicoC	4
1.3 Eigenheiten der Sprachen C und PicoC	5
1.4 Gesetzte Schwerpunkte	12
1.5 Über diese Arbeit	12
1.5.1 Still der schriftlichen Ausarbeitung	14
1.5.2 Aufbau der schriftlichen Arbeit	15
Literatur	A

Abbildungsverzeichnis

1.1	Schritte zum Ausführen eines Programmes mit dem GCC.	1
1.2	Stark vereinfachte Schritte zum Ausführen eines Programmes.	2
1.3	Speicherorganisation.	4
1.4	README.md im Github Repository der Bachelorarbeit.	13

Codeverzeichnis

1.1	Beispiel für die Spiralregel.	7
1.2	Ausgabe des Beispiels für die Spiralregel.	7
1.3	Beispiel für unterschiedliche Ausführung.	7
1.4	Ausgabe des Beispiels für unterschiedliche Ausführung.	7
1.5	Beispiel mit Dereferenzierungsoperator.	8
1.6	Ausgabe des Beispiels mit Dereferenzierungsoperator.	8
1.7	Beispiel dafür, dass Struct kopiert wird.	9
1.8	Ausgabe des Beispiels dafür, dass Struct kopiert wird.	9
1.9	Beispiel dafür, dass Zeiger auf Feld übergeben wird.	9
1.10	Ausgabe des Beispiels dafür, dass Zeiger auf Feld übergeben wird.	10
1.11	Beispiel für Deklaration und Definition.	11
1.12	Ausgabe des Beispiels für Deklaration und Definition.	11
1.13	Beispiel für Sichtbarkeitsbereiche.	11
1.14	Ausgabe des Beispiels für Sichtbarkeitsbereiche.	12

Tabellenverzeichnis

1.1 Register der RETI-Architektur.	3
--	---

Definitionsverzeichnis

1.1	Imperative Programmierung	6
1.2	Strukturierte Programmierung	6
1.3	Prozedurale Programmierung	6
1.4	Call by Value	8
1.5	Call by Reference	9
1.6	Funktionsprototyp	10
1.7	Deklaration	10
1.8	Definition	10
1.9	Sichtbarkeitsbereich (bzw. engl. Scope)	11

Grammatikverzeichnis

1 Einführung

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiell** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprache **Python**, welche als **interpretierte** Sprache bekannt ist, wird ein in der Programmiersprache **Python** geschriebenes Programm vorher zu **Bytecode**¹ kompiliert, bevor dieses von der **Python Virtual Machine (PVM)** interpretiert wird.

Anmerkung 🔍

Die Programmiersprache **Python** und jegliche **andere Sprache** wird fortan als L_{Python} bzw. als $L_{Name\ der\ Sprache}$ bezeichnet wird.

Compiler, wie der **GCC**² oder **Clang**³ werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm zu **Maschinencode** (Definition ??) zu kompilieren. Das Programm muss hierzu in der **Sprache** geschrieben sein, die der Compiler kompiliert⁴

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll. Im Fall des **GCC** funktioniert das über `> gcc program.c -o machine_code`⁵. Als Ergebnis erhält man im Fall des **GCC** die mit der Option `-o` selbst benannte Datei `machine_code`. Diese kann dann z.B. unter **Unix-Systemen** über `> ./machine_code` **ausgeführt** werden, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

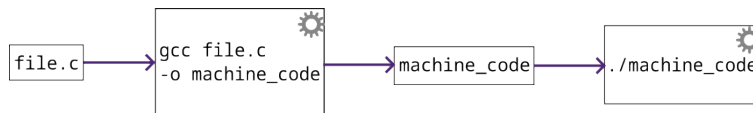


Abbildung 1.1: Schritte zum Ausführen eines Programmes mit dem GCC.

Der ganze Kompilervorgang kann, wie er in Abbildung 1.2 dargestellt ist, zu einer Box **Compiler** abstrahiert werden. Der Benutzer gibt ein **Programm** in der Sprache des Compilers rein und erhält **Maschinencode**. Diesen **Maschinencode** kann er dann im besten Fall in eine andere Box hineingeben, welche die passende **Maschine** oder den passenden **Interpreter** in Form einer **Virtuellen Maschine** repräsentiert. Die Maschine bzw. der Interpreter kann den **Maschinencode** dann **ausführen**.

¹Dieser Begriff ist **nicht** weiter **relevant**.

²*GCC, the GNU Compiler Collection - GNU Project.*

³*clang: C++ Compiler.*

⁴Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C .

⁵Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objektcode** (Definition ??) kompiliert wird. Das Kompilieren zu **Objektcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

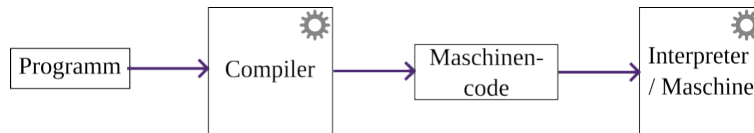


Abbildung 1.2: Stark vereinfachte Schritte zum Ausführen eines Programmes.

Der Programmierer muss für das Vorgehen in Abbildung 1.2 **nichts** über die **Theoretischen Grundlagen des Compilerbau** wissen, noch wie der Compiler **intern** umgesetzt ist. In dieser Bachelorarbeit soll diese **Compilerbox** allerdings geöffnet werden und anhand eines eigenen im Vergleich zum **GCC** im Funktionsumfang **reduzierten Compilers** gezeigt werden, wie so ein Compiler **unter der Haube** grundlegend funktioniert.

Die konkrete **Aufgabe** besteht darin, einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC} in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinsprache** L_{RETI} kompilieren kann. Die Sprache L_{PicoC} ist hierbei eine **Untermenge** der äußerst bekannten Programmiersprache L_C , die der **GCC** kompilieren kann.

In dieser **Einführung** werden die für diese Bachelorarbeit elementaren **Thematiken** erstmals angeschnitten und grundlegende **Informationen zu dieser Arbeit** genannt. Gerade wurde das **Thema** dieser Bachelorarbeit veranschaulicht und die konkrete **Aufgabenstellung** ausformuliert. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zugrunde liegt und im Unterkapitel 1.2 wird näher auf die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll. Des Weiteren wird in Unterkapitel 1.3 insbesondere auf bestimmte **Eigenheiten der Sprachen** L_C und L_{PicoC} eingegangen, auf welche in dieser Bachelorarbeit ein besonderes Augenmerk gerichtet wird. Danach wird in Unterkapitel 1.4 auf für diese Bachelorarbeit gesetzte **Schwerpunkte** eingegangen und in Unterkapitel 1.5 etwas zum **Aufbau** und **Stil** dieser schriftlichen Ausarbeitung gesagt.

1.1 RETI-Architektur

Die **RETI-Architektur** ist eine zu Lernzwecken für die Vorlesungen C. Scholl, „Betriebssysteme“ und C. Scholl, „Technische Informatik“ eingesetzte **32-Bit** Architektur, die sich vor allem durch ihre einfache Zugänglichkeit kennzeichnet. Deren **Maschinsprache** L_{RETI} wurde als Zielsprache des **PicoC-Compilers** hergenommen. In der Vorlesung C. Scholl, „Technische Informatik“ wird die **grundlegende RETI-Architektur** erklärt und in der Vorlesung C. Scholl, „Betriebssysteme“ wird diese Architektur erweitert, sodass diese mehr darauf angepasst ist, dass auch komplexere Konstrukte, wie ein **Betriebssystem**, **Interrupts**, **Prozesse**, **Funktionen** usw. auf nicht zu komplizierte Weise implementiert werden können.

Um den **PicoC-Compiler** zu **testen** war es notwendig einen **RETI-Interpreter** zu implementieren, der genau die Variante der **RETI-Architektur** aus der Vorlesung C. Scholl, „Betriebssysteme“ **simuliert**. Für genauere **Implementierungsdetails** der RETI-Architektur ist auf die Vorlesungen C. Scholl, „Technische Informatik“ und C. Scholl, „Betriebssysteme“ zu verweisen.

Anmerkung

In dieser **Bachelorarbeit** wird **im Folgenden** bei der **Maschinsprache** L_{RETI} immer von der Variante ausgegangen, welche durch die **RETI-Architektur** aus der Vorlesung C. Scholl, „Betriebssysteme“ umgesetzt ist.

Die **Register** dieser **RETI-Architektur** werden in Tabelle ?? aufgezählt und erläutert. Der **Befehlssatz** und die **Datenpfade** der **RETI-Architektur** sind im Kapitel ?? dokumentiert, da diese **nicht explizit** zum

Verständnis der späteren Kapitel notwendig sind. Allerdings sind diese zum **tieferen Verständnis** notwendig, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** L_{RETI} ist durch die Grammatiken ?? und ?? zusammengekommen beschrieben.

Register Kürzel	Register Ausgeschrieben	Aufgabe
PC	Program Counter	Zeigt auf den Maschinenbefehl , der als nächstes ausgeführt werden soll.
ACC	Accumulator	Für Operanden von Operationen oder für temporäre Werte .
IN1	Indexregister 1	Hat dieselbe Aufgabe wie das ACC-Register.
IN2	Indexregister 2	Hat dieselbe Aufgabe wie das ACC-Register.
SP	Stackpointer	Zeigt immer auf die erste freie Speicherzelle am Ende des Stacks ^a , wo als nächstes Speicher allokiert werden kann.
BAF	Begin Aktive Funktion	Zeigt auf den Beginn des Stackframes der aktuell aktiven Funktion .
CS	Codesegment	Zeigt auf den Beginn des Codesegments .
DS	Datensegment	Zeigt auf den Beginn des Datensegments . Die letzten 10 Bits werden verwendet, um 22 Bit Immediates aufzufüllen . Kann dadurch dazu verwendet werden, festzulegen welches der 3 Peripheriegeräte ^b in der Memory Map ^c angesprochen werden soll.

^a Wird noch erläutert.

^b **EPROM**, **UART** und **SRAM**.

^c Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, wird diese **nicht** mehr als nötig im weiteren Verlauf **erläutert**.

Tabelle 1.1: Register der RETI-Architektur.

Die **RETI-Architektur** ermöglicht es, bei der Ausführung von RETI-Programmen **Prozesse** aufzubauen bzw. zu nutzen. In Abbildung 1.3 ist der **Aufbau** eines **Prozesses** im **Hauptspeicher** der **RETI-Architektur** zu sehen. Ein RETI-Programm nutzt dabei den **Stack** für **temporäre Zwischenergebnisse** von Berechnungen und zum **Anlegen der Stackframes** von **Funktionen**, welche die **Lokalen Variablen** und **Parameter** einer Funktion speichern. Das SP- und BAF-Register erfüllen dabei ihre in Tabelle 1.1 zugeteilten Aufgaben für den **Stack**.

Der Abschnitt für die **Globalen Statischen Daten** ist allgemein dazu da, Daten zu beherbergen, die für den Rest der Programmausführung **global** zugänglich sein sollen, aber auch für die **Lokalen Variablen** der **main-Funktion**. Das DS-Register markiert den **Anfang** des **Datensegments** und damit auch die **Anfangsadresse**, ab der die **Globalen Statischen Daten** abgespeichert sind und kann als **relativer Orientierungspunkt** beim **Zugriff** und **Abspeichern** Globaler Statischer Daten dienen. Das CS-Register wird als **relativer Orientierungspunkt** genutzt, an dem die **Ausführung** von RETI-Programmen **startet**. Darüberhinaus wird das CS-Register dazu genutzt, die **relative Startadresse** zu bestimmen, an welcher der **RETI-Code** einer bestimmten **Funktion** anfängt. Der **Heap** ist nicht weiter relevant, da die **Funktionalitäten** der Sprache L_C , welche diesen nutzen in L_{PicoC} **nicht enthalten** sind.

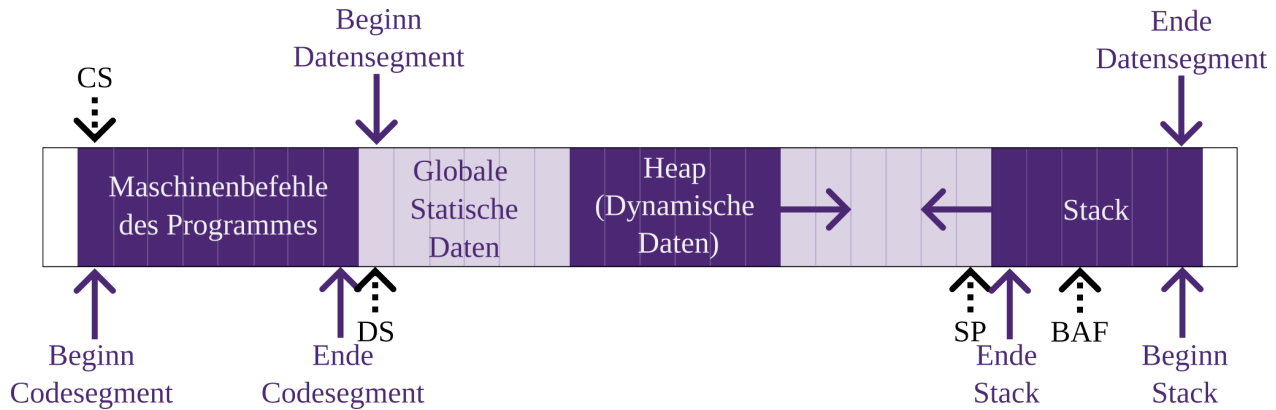


Abbildung 1.3: Speicherorganisation.

Die RETI-Architektur nutzt 3 verschiedene **Peripheriegeräte**, **EPROM**, **UART** und **SRAM**, die über eine **Memory Map**⁶ den über die **Datenpfade** der RETI-Architektur ?? ansprechbaren **Adressraum** von 2^{32} Adressen⁷ unter sich aufteilen.

Die **Ausführung** eines Programmes **startet** auf die einfachste Weise, indem es von einem **Startprogramm** im **EPROM**⁸ aufgerufen wird. Der **EPROM** wird beim Start einer **RETI-CPU** als **erstes** aufgerufen. Das liegt daran, dass bei der **Memory Map** der erste **Adressraum** von 0 bis $2^{30} - 1$ dem **EPROM** zugeordnet ist und das **PC-Register** **initial** den Wert 0 hat. Daher wird als **erstes** das Programm ausgeführt, welches an **Adresse 0** im EPROM anfängt.

Die **UART**⁹ ist eine **elektronische Schaltung** mit je nach Umsetzung mehr oder weniger **Registern**. Es gibt allerdings immer einen **RX**- und einen **TX**-Register, für jeweils Empfangen¹⁰ und Versenden¹¹ von Daten. Jedes der **Register** wird dabei mit einer anderen von 2^3 verschiedenen Adressen angesprochen. Jeweils 8-Bit können nach den **Datenpfaden der RETI-CPU** ?? auf einmal in ein **Register** der **UART** geschrieben werden, um **versandt** zu werden oder von einem Register **empfangen** werden. Die **UART** dient als **serielle Schnittstelle** und kann z.B. genutzt werden, um Daten an einen sehr einfach gehaltenen **Monitor** zu senden, der diese dann anzeigt.

An letzter Stelle muss der **SRAM**¹² erwähnt werden, bei dem es sich um den **Hauptspeicher** der **RETI-CPU** handelt. Der **Zugriff auf den Hauptspeicher** ist deutlich schneller als z.B. auf ein **externes Speichermedium**, aber **langsamer** als der **Zugriff auf Register**. Die **Datenmenge**, die in einer **Speicherzelle** des **Hauptspeichers** abgespeichert ist, beträgt hierbei $32 \text{ Bit} = 4 \text{ Byte}$. In der RETI-Architektur ist aufgrund dessen, dass es sich um eine **32-Bit Architektur** handelt ein **Datenwort** 32 Bit breit. Aus diesem Grund sind alle **Register** 32 Bit groß, die **Operanden der Arithmetisch Logischen Einheit**¹³ sind 32 Bit breit, die **Befehle** des Befehlssatzes sind innerhalb von 32 Bit codiert usw.

1.2 Die Sprache PicoC

Die Sprache L_{PicoC} ist eine **Untermenge** der Sprache L_C , welche

⁶Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, sondern nur bei der Umsetzung des **RETI-Interpreters**, wird diese **nicht näher erläutert** als notwendig.

⁷Von 0 bis $2^{32} - 1$.

⁸Kurz für **Erasable Programmable Read-Only Memory**.

⁹Kurz für **Universal Asynchronous Receiver Transmitter**.

¹⁰Engl. **Receiving**, daher das **R**.

¹¹Engl. **Transmission**, daher das **T**.

¹²Kurz für **Static Random-Access Memory**.

¹³Ist für **Arithmetische, Bitweise und Logische Berechnungen** zuständig.

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/* comment */`.
- die **Basisdatentypen**¹⁴ `int`, `char` und `void`.
- die **Zusammengesetzten** Datentypen¹⁵ **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; int attr2;}`) und **Zeiger** (z.B. `int *pntr`) und ihre zugehörigen **Operationen** `[i]`, `.attr` und `*` usw.
- `if(cond){ }-` und `else{ }-`**Anweisungen**¹⁶.
- `while(cond){ }-` und `do while(cond){ };`**Anweisungen**.
- **Arihmatische und Bitweise Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` und **unären Operatoren** `-`, `~` umgesetzt sind.¹⁷
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind.
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`), **Funktionsaufrufe** (z.B. `fun(ar, st_var)`) und **Sichtbarkeitsbereiche** innerhalb der Codeblöcke `{ }` der Funktionen.

beinhaltet. Die ausgegrauten • wurden bereits für das **Bachelorprojekt** umgesetzt und mussten für die **Bachelorarbeit** nur an die **neue Architektur** angepasst werden.

Der grundlegende **Aufbau** von Programmen der **Programmiersprache** L_{PicoC} ist durch Grammatik ?? und Grammatik ?? zusammengefasst beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache L_C , die genauso ein Teil der Programmiersprache L_{PicoC} sind¹⁸, werden im Folgenden genauer erläutert. Diese **Eigenheiten** werden in der Implementierung des **PicoC-Compilers** in Kapitel ?? noch eine **wichtige Rolle** spielen.

Anmerkung 🔍

Im Folgenden wird immer von der Programmiersprache L_{PicoC} gesprochen, da es in dieser Bachelorarbeit um diese geht und die folgenden Beispiele für die Ausgaben alle mithilfe des **PicoC-Compilers** und **RETI-Interpreters kompiliert** und daraufhin **ausgeführt** wurden. Aber selbiges gilt aus bereits erläuterten Grund genauso für L_C .

Bei der Programmiersprache L_{PicoC} handelt es sich um eine **Imperative** (Definition 1.1), **Strukturierte** (Definition 1.2) und **Prozedurale Programmiersprache** (Definition 1.3). Aufgrund dessen, dass es um eine **Imperative Programmiersprache** handelt ist es wichtig bei der Implementierung die **Reihenfolge** zu beachten.

¹⁴Bzw. `int` und `char` werden auch als **Primitive Datentypen** bezeichnet.

¹⁵Bzw. engl. **compound datatypes**.

¹⁶Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

¹⁷Theoretisch sind die Operatoren `<<`, `>>` und `~` unnötig, da sie durch **Multiplikation** `*`, **Division** `/` und Anwendung des **Xor-^**-Operators auf eine Zahl, deren **binäre Repräsentation** ein Folge von `len` **gleicher Länge** ist ersetzt werden können.

¹⁸Da L_{PicoC} eine **Untermenge** von L_C ist.

Und aufgrund dessen, dass es um eine **Strukturierte** und **Prozedurale Programmiersprache** handelt, ist es eine gute Methode bei der Implementierung auf **Blöcke**¹⁹ zu setzen, zwischen denen **hin und her** gesprungen werden kann. **Blöcke** stellen in den einzelnen Implementierungsschritten die **notwendige Datenstruktur** dar, um **Auswahl** zwischen Codestücken, **Wiederholung** von Codestücken und **Sprünge** zu Blöcken mit entsprechend zu bestimmten **Bezeichnern** (Definition ??) passenden **Labeln** (Definition ??) umzusetzen.

Definition 1.1: Imperative Programmierung

Wenn ein Programm aus einer **Folge von Anweisungen** besteht, deren **Reihenfolge** auch bestimmt in welcher **Reihenfolge** diese **Befehle** auf einer **Maschine** ausgeführt werden.^a

^aThieman, „Einführung in die Programmierung“.

Definition 1.2: Strukturierte Programmierung

Wenn ein Programm anstelle von z.B. `goto label`-Anweisungen **Kontrollstrukturen**, wie z.B. `if(cond) { } else { }, while(cond) { }` usw. verwendet, welche dem Programmcode **mehr Struktur** geben, weil die **Auswahl** zwischen Anweisungen und die **Wiederholung** von Anweisungen eine **klare und eindeutige Struktur** hat. Diese Struktur wäre bei der Umsetzung mit einer `goto label`-Anweisung **nicht so eindeutig erkennbar** und auch **nicht** unbedingt immer **gleich aufgebaut** wäre.^a

^aThieman, „Einführung in die Programmierung“.

Definition 1.3: Prozedurale Programmierung

Programme werden z.B. mittels **Funktionen** in **überschaubare Unterprogramme**^a aufgeteilt, die **aufrufbar** sind. Dies **vermeidet** einerseits **redundanten Code**, indem Code **wiederverwendbar** gemacht wird und andererseits erlaubt es z.B. Codestücke nach ihren Aufgaben zu **abstrahieren**. Den Codestücken wird eine **Aufgabe zugeteilt**, sie werden zu **Unterprogrammen** gemacht und fortan über einen **Bezeichner aufgerufen**. Das macht den Code deutlich **überschaubarer**, da man die Aufgabe eines Codestücks nun nur noch mit seinem **Bezeichner assoziieren** muss.^b

^aBzw. auch **Prozeduren** genannt.

^bThieman, „Einführung in die Programmierung“.

In *L_{PicoC}* ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen `<i>`-Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: `<remaining-datatype><var><i>`, während andere Programmiersprachen die eckigen `<i>`-Klammern **vor** die Variable schreiben `<remaining-datatype><i><var>`.

Werden die eckigen `<i>`-Klammern **hinter** die Variable geschrieben, ist es **schwieriger** den **Datentyp abzulesen**, als auch ein **Programm zu implementieren** was diesen erkennt. Damit ein Programmierer den **Datentyp ablesen** kann, kann dieser die **Spiralregel** verwenden, die unter der Webseite *Clockwise/Spiral Rule*²⁰ nachgelesen werden kann. Werden die eckigen `<i>`-Klammern **hinter** die Variable geschrieben, wirken diese zum verwechseln ähnlich zum `<var><i>`-Operator für den **Zugriff auf den Index eines Feldes**. Wenn Ausdrücke, wie `int ar[1] = {42}` und `var[0] = 42` vorliegen, sind `var[1]` und `var[0]` nur durch den **Kontext** um sie herum unterscheidbar.

In Code 1.1 ist ein Beispiel zu sehen, indem die Variable `complex_var` den **Datentyp „Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Zeigern auf Felder der Mächtigkeit 2 von Verbunden vom Typ st“** hat. Ein Vorteil davon die eckigen `<i>`-Klammern **hinter** die Variable zu schreiben ist in der **markierten Zeile** in Code 1.1 zu sehen. Will man auf ein **Element dieses Datentyps** zugreifen

¹⁹Werden später im Kapitel ?? genauer erklärt.

²⁰<https://c-faq.com/decl/spiral.anderson.html>

`(*complex_var[0][1])[1].attr`, so ist der Ausdruck fast genau **gleich aufgebaut**, wie der Ausdruck für den **Datentyp** `struct st (*complex_var[1][2])[2]`. Die **Ausgabe** des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

```

1 struct st {int attr;};
2
3 void main() {
4     struct st st_var[2] = {{.attr=314}, {.attr=42}};
5     struct st (*complex_var[1][2])[2] = {{&st_var, &st_var}};
6     print((*complex_var[0][1])[1].attr);
7 }
```

Code 1.1: Beispiel für die Spiralregel.

```

1 42
```

Code 1.2: Ausgabe des Beispiels für die Spiralregel.

In L_{PicoC} ist die **Ausführbarkeit einer Operation** oder **wie** diese Operation ausgeführt wird davon abhängig, was für einen **Datentyp** die Variable im **Kontext** der auszuführenden Operation hat. In dem Beispiel in Code 1.3 wird in Zeile 2 ein „**Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Integern**“ und in Zeile 3 ein „**Zeiger auf Felder der Mächtigkeit 2 von Integern**“ erstellt. In den **markierten Zeilen** wird zweimal in Folge die **gleiche Operation** `<var>[0][1]` ausgeführt, allerdings hat die Operation aufgrund der **unterschiedlichen Datentypen** der beiden Variablen einen **unterschiedlichen Effekt**.

In der markierten Zeile 4 wird ein **normaler Zugriff** auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt. In der nachfolgend markierten Zeile 5 wird allerdings erst dem **Zeiger** `int (*pntr)[2] = &ar[0]`; **gefolgt** und dann ein Zugriff auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt. **Beide Operationen** haben, wie in Code 1.4 zu sehen ist die **gleiche Ausgabe**.

```

1 void main() {
2     int ar[1][2] = {{314, 42}};
3     int (*pntr)[2] = &ar[0];
4     print(ar[0][1]);
5     print(pntr[0][1]);
6 }
```

Code 1.3: Beispiel für unterschiedliche Ausführung.

```

1 42 42
```

Code 1.4: Ausgabe des Beispiels für unterschiedliche Ausführung.

Eine weitere interessante Eigenheit, die in L_{PicoC} gültig ist, ist, dass die Operationen `<var>[0][1]` und `*(<var>+0)+1` aus Code 1.3 und Code 1.5 komplett **austauschbar** sind. Die Ausgabe in Code 1.4 ist folglich **identisch** zur Ausgabe in Code 1.6.

```
1 void main() {
2     int ar[1][2] = {{314, 42}};
3     int (*pntr)[2] = &ar[0];
4     print(*(&ar+0)+1);
5     print(*(&pntr+0)+1);
6 }
```

Code 1.5: Beispiel mit Dereferenzierungsoperator.

```
1 42 42
```

Code 1.6: Ausgabe des Beispiels mit Dereferenzierungsoperator.

In der Programmiersprache L_{PicoC} werden alle **Argumente** bei einem Funktionsaufruf nach der **Call by Value**-Strategie (Definition 1.4) übergeben. Ein Beispiel hierfür ist in Code 1.7 zu sehen. Hierbei wird ein **Verbund** `struct st copyable_ar = {.ar={314, 314}}`;²¹ an die Funktion `fun` übergeben. Hierzu wird der **Verbund** in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopiert** und an den Parameter `fun` gebunden.

Wie an der Ausgabe in Code 1.7 zu sehen ist hat die **Zuweisung** `copyable_ar.ar[1] = 42` an den **Parameter** `struct st copyable_ar` in der **aufgerufenen** Funktion `fun` keinen Einfluss auf die übergebene **lokale Variable** `struct st copyable_ar = {.ar={314, 314}}` der **aufrufenden** Funktion. Der Eintrag an Index 1 im Feld bleibt bei 314.

Definition 1.4: Call by Value



Bei einem **Funktionsaufruf** wird eine **Kopie** des **Ergebnisses eines Ausdrucks**, welcher ein **Argument** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.

Das hat zur Folge, dass bei **Übergabe** einer Variable als **Argument** an eine Funktion, diese Variable bei **Änderungen** am entsprechenden **Parameter** der **aufgerufenen** Funktion in der **aufrufenden** Funktion **unverändert** bleibt.^a

^aBast, „Programmieren in C“.

```
1 struct st {int ar[2]};
2
3 int fun(struct st copyable_ar) {
4     copyable_ar.ar[1] = 42;
5 }
6
7 void main() {
8     struct st copyable_ar = {.ar={314, 314}};
```

²¹Später wird darauf eingegangen, warum der Verbund den **Bezeichner** `copyable_ar` erhalten hat.

```

9  print(copyable_ar.ar[1]);
10 fun(copyable_ar);
11  print(copyable_ar.ar[1]);
12 }

```

Code 1.7: Beispiel dafür, dass Struct kopiert wird.

```

1 314 314

```

Code 1.8: Ausgabe des Beispiel dafür, dass Struct kopiert wird.

In der Programmiersprache L_{PicoC} gibt es **kein Call by Reference** (Definition 1.5), allerdings kann der **Effekt** von Call by Reference mittels **Zeigern simuliert** werden, wie es in Code 1.11²² bei der Funktion `fun_declared_before` und dem Parameter `int *param` zu sehen ist. Genau dieser **Trick** wird bei **Feldern** verwendet, um **nicht** das gesamte Feld bei einem Funktionsaufruf in den **Stackframe** der **aufgerufenen** Funktion **fun kopieren** zu müssen.

Wie im Beispiel in Code 1.9 zu sehen ist, wird in der markierten Zeile ein Feld `int ar[2] = {314, 314}` an die Funktion `fun` übergeben. Wie in der **Ausgabe** in Code 1.10 zu sehen ist, hat sich der Eintrag an Index 1 im Feld durch die **Zuweisung** `ar[1] = 42` nach dem **Funktionsaufruf** zu 42 geändert. Wird ein Feld direkt als Ausdruck `ar` ohne z.B. die eckigen `[]`-Klammern für einen **Indexzugriff** hingeschrieben, wird die **Adresse** des Felds verwendet und **nicht** z.B. der Wert des **ersten Elements** des Felds.

Eine Möglichkeit ein **Feld** als **Kopie** und **nicht** als **Referenz** zu übergeben ist es, wie in Code 1.7 bei der Variable `copyable_ar` das **Feld** als **Attribut** eines **Verbundes** zu übergeben.

Definition 1.5: Call by Reference



Bei einem **Funktionsaufruf** wird eine **implizite Referenz** eines **Arguments** an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.

Implizit meint hier, dass der Benutzer einer Funktionalität mit Call by Reference **nicht mitbekommt**, dass er das **Argument selbst verändert** und **keine lokale Kopie des Arguments**.^a

^aBast, „Programmieren in C“.

```

1 int fun(int ar[2]) {
2   ar[1] = 42;
3 }
4
5 void main() {
6   int ar[2] = {314, 314};
7   print(ar[1]);
8   fun(ar);
9   print(ar[1]);
10 }

```

²²Unten im Code schauen.

Code 1.9: Beispiel dafür, dass Zeiger auf Feld übergeben wird.

```
1 314 42
```

Code 1.10: Ausgabe des Beispiels dafür, dass Zeiger auf Feld übergeben wird.

Ein Programm in der Programmiersprache L_{PicoC} wird von **oben-nach-unten** ausgewertet. Ein Problem tritt auf, wenn z.B. eine Funktion verwendet werden soll, die aber erst **unter** dem entsprechenden Funktionsaufruf **definiert** (Definition 1.8) wird. Es ist wichtig, dass der **Prototyp** (Definition 1.6) einer Funktion vor dem **Funktionsaufruf** dieser Funktion bekannt ist. Das hat den Sinn, dass bereits während des Kompilierens überprüft werden kann, ob die beim Funktionsaufruf **übergebenen Argumente** den gleichen **Datentyp** haben, wie die **Parameter** des **Prototyps** und ob die **Anzahl Argumente** mit der **Anzahl Parameter** des **Prototyps** übereinstimmt.

Allerdings lassen sich Funktionen **nicht** immer so anordnen, dass jede in einem Funktionsaufruf aufzurufende Funktion vorher **definiert** sein kann. Aus diesem Grund ist es möglich den **Prototyp** einer Funktion vorher zu **deklarieren** (Definition 1.7), wie es in den **markierten Zeile** im Beispiel in Code 1.11 zu sehen ist. Die **Ausgabe** des Beispiels ist in Code 1.12 zu sehen.

Definition 1.6: Funktionsprototyp

*Deklaration einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parameterreihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei **Funktionsprototypen** mit dem **gleichen Funktionsbezeichner** zu haben.*^{a,b}

^aDer **Funktionsprototyp** ist von der **Funktionssignatur** zu unterscheiden, die in Programmiersprache wie **C++** und **Java** für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentypen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Sichtbarkeitsbereiche** und **Klassentypen** usw. unterscheiden.

^bWhat is the difference between function prototype and function signature?

Definition 1.7: Deklaration

*Der **Datentyp** bzw. **Prototyp** einer **Variablen** bzw. **Funktion**, sowie der **Bezeichner** dieser **Variable** bzw. **Funktion** wird dem **Compiler** mitgeteilt.*^{a,b,c}

^aÜber das **Schlüsselwort** **extern** lassen sich in der Programmiersprache L_C Variablen **deklarieren**, **ohne** sie zu **definieren**.

^bVariablen in C und C++, Deklaration und Definition — Coder-Welten.de.

^cP. Scholl, „Einführung in Embedded Systems“.

Definition 1.8: Definition

*Dem **Compiler** wird mitgeteilt, dass zu einem **bestimmten Zeitpunkt** in der Programmausführung oder bereits vor der Ausführung **Speicher angelegt** werden soll und **wo**^a dieser angelegt werden soll.*

^aIm Fall des PicoC-Compilers im Abschnitt für die **Globalen Statischen Daten** oder auf dem **Stack**.

```
1 void fun_declared_before(int *param);
2
3 int fun_defined(int param) {
```

```

4  return param + 10;
5  }
6
7  void main() {
8      int res = fun_defined(22);
9      fun_declared_before(&res);
10     print(res);
11 }
12
13 void fun_declared_before(int *param) {
14     *param = *param + 10;
15 }

```

Code 1.11: Beispiel für Deklaration und Definition.

```

1 42

```

Code 1.12: Ausgabe des Beispiels für Deklaration und Definition.

In L_{PicoC} lässt sich eine **Variable** nur innerhalb ihres **Sichtbarkeitsbereichs** (Definition 1.9) verwenden. **Lokale Variablen** und **Parameter** lassen sich nur innerhalb der **Funktion** in welcher sie definiert wurden verwenden. Der **Sichtbarkeitsbereich** von **Lokalen Variablen** und **Parametern** erstreckt sich hierbei von der **öffnenden** `{`-Klammer bis zur **schließenden** `}`-Klammer der **Funktionsdefinition**, in welcher sie definiert wurden.

Verschiedene **Sichtbarkeitsbereiche** können dabei **identische** Bezeichner besitzen. Im Beispiel in Code 1.13 kommt der markierte **Bezeichner** `local_var` in 2 verschiedenen **Sichtbarkeitsbereichen** vor und bezeichnet somit 2 **unterschiedliche Variablen**. Der **Parameter** `param` und die **Lokale Variable** `local_var` dürfen **nicht** den **gleichen Bezeichner** haben, da sie sich im gleichen **Sichtbarkeitsbereich** der Funktion `fun_scope` befinden. Die **Ausgabe** des Beispiels in Code 1.13 ist in Code 1.14 zu sehen.

Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)

*Bereich in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.^a*

^aThiemann, „Einführung in die Programmierung“.

```

1  int fun_scope(int param) {
2      int local_var = 2;
3      print(param);
4      print(local_var);
5  }
6
7  void main() {
8      int local_var = 4;
9      fun_scope(local_var);
10 }

```

Code 1.13: Beispiel für Sichtbarkeitsbereiche.

1 4 2

Code 1.14: *Ausgabe des Beispiels für Sichtbarkeitsbereiche.*

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit war es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinensprache L_{RETI} die **Syntax** und **Semantik** der Programmiersprache L_C **identisch** nachzuahmen. Der **PicoC-Compiler** sollte die Programmiersprache L_{PicoC} im Vergleich zu z.B. dem **GCC**²³ ohne merklichen Unterschied²⁴ kompilieren können.

In **zweiter Linie** sollte dabei möglichst immer so Vorgegangen werden, wie es die **RETI-Codeschnipsel** aus der Vorlesung C. Scholl, „Betriebssysteme“ vorgeben. Allerdings sollten diese bei **Inkonsistenzen** bezüglich der durch sie selbst vorgegebenen **Paradigmen** und anderen **Umstimmigkeiten** angepasst werden, da der **erstere Schwerpunkt** überwiegt.

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter [Link](#)²⁵ zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter „**Getting Started**“ ein kleines **Einführungstutorial** verlinkt. Unter „**Usage**“ ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Daneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter [Link](#)²⁶ zu finden.

²³Da die Sprache L_{PicoC} eine **Untermenge** von L_C ist, kann der **GCC** L_{PicoC} ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinensprache L_{RETI} .

²⁴Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinensprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

²⁵<https://github.com/matthejue/PicoC-Compiler>.

²⁶<https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971>.

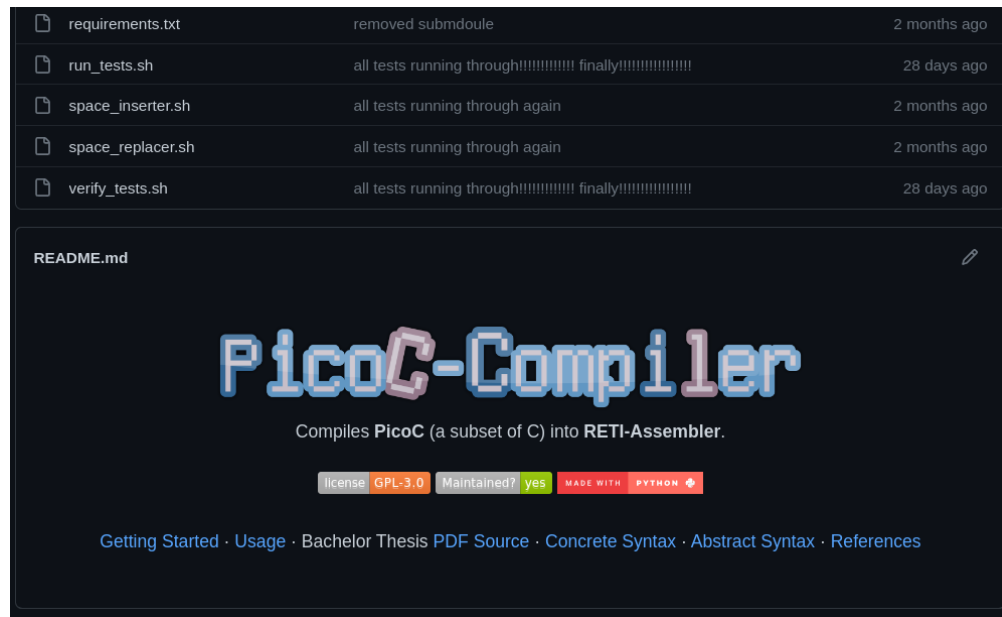


Abbildung 1.4: *README.md* im Github Repository der Bachelorarbeit.

Die **Schriftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schriftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF**-Datei unter [Link²⁷](#) zu finden. Die **PDF**-Datei der Schriftliche Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode** automatisch mithilfe der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert. Der **Latexquellcode** ist unter [Link²⁸](#) veröffentlicht.

Alle verwendeten **Latex Bibliotheken** sind unter [Link²⁹](#) zu finden³⁰. Die Grafiken, die nicht mittels der Tikz Bibliothek in Latex erstellt wurden, wurden mithilfe des Vektorgraphikeditors **Inkscape**³¹ erstellt. Falls Interesse besteht **Grafiken** aus dieser schriftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den **.svg**-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibliotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter „References“ bzw. direkt unter [Link³²](#) zu finden.

Um die verschiedenen **Aspekte** der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel **Einführung** werden Codebeispiele zur **Anschauung** verwendet. Mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters** werden **Ausgaben** erzeugt, die in dieses Dokument **eingelassen** wurden. Im Kapitel ?? werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** in Form von Codebeispielen gezeigt³³.

Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelassen. Alle

²⁷https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.

²⁸<https://github.com/matthejue/Bachelorarbeit>.

²⁹https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.

³⁰Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.

³¹Developers, *Draw Freely — Inkscape*.

³²https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

³³Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist. Später mehr dazu.

hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem [Link](#)³⁴ finden. Mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` lassen sich die Codebeispiele genauso **kompilieren**, wie sie hier dargestellt sind³⁵.

1.5.1 Still der schriftlichen Ausarbeitung

In dieser schriftlichen Ausarbeitung der Bachelorarbeit sind manche **Wörter** für einen besseren Lesefluss **hervorgehoben**. Es ist so gedacht, dass die **hervorgehobenen Wörter** beim Lesen sichtbare **Ankerpunkte** darstellen an denen sich **orientiert** werden kann. Aber es hat auch den Zweck, dass der **Inhalt** eines vorher gelesener Paragraphs nochmal durch **Überfliegen** der Hervorgehobenen Wörter in **Erinnerung gerufen** werden kann.

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas genau so gemacht wurde und nicht anders“. Wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie unter anderem zum initialen **wirklichen Verständnis** eines Themas beantwortet braucht³⁶, die Frage des „**warum**“.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** in `Definition`-s-Kästen orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**³⁷ verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen untereinander **konsistent** sind, wie auch das in ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernommen, so wurde die Definition oder der entsprechende Teil in „**Anführungszeichen**“ gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox** wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

In den **sonstigen Fußnoten** befinden sich **Informationen**, die vielleicht beim **Verständnis** helfen oder **kleinere Details** enthalten, die bei **tiefgreifenderem Interesse** interessant sein könnten. Im Allgemeinen werden die **Informationen in den Fußnoten** allerdings **nicht** zum **Verständnis** der Bachelorarbeit **benötigt**.

Des Weiteren gibt es `Anmerkung`-s-Kästen, welche kleine **Anmerkungen** enthalten, die über **Konventionen** aufklären sollen, vor **Fallstricken warnen**, die leicht zur Verwirrung führen können oder Informationen bei **tiefergehendem Interesse** oder für den besseren **Überblick** enthalten. Der Inhalt dieser `Anmerkung`-s-Kästen ist allerdings zum Verständnis dieser Arbeit **nicht essentiell** wichtig.

Es wurde immer versucht möglichst **deutsche Fachbegriffe** zu verwenden, sofern sie einigermaßen **geläufig** sind und bei der Verwendung **nicht** eher **verwirren**³⁸. Bei **Code** und anderem **Text**, dessen Zweck **nicht** dem Erklären dient, sondern der Veranschaulichung, wurde dieser konsequent in **Englisch** geschrieben bzw. belassen. Der Grund hierfür ist unter anderem, da die **Bezeichner** in der **Implementierung** des PicoC-Compilers, wie es mehr oder weniger **Konvention** beim Programmieren ist in **Englisch** benannt sind und diese Bezeichner in den **Ausgaben** des **PicoC-Compilers** vorkommen³⁹.

³⁴https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.

³⁵Es wurde zu diesem Zweck die **Command-line Option** `-t, --thesis` erstellt, die bestimmte Kommentare **herausfiltert**, damit die generierten **Abstrakten Syntaxbäume** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu **überfüllt** mit Kommentaren sind.

³⁶Vor allem am **Anfang**, wo der Leser **wenig** über das Thema **weiß**.

³⁷Das ist ein **Beispiel** für eine **Fußnote**.

³⁸Bei dem z.B. auch im Deutschen geläufigen Fachbegriff „**Statement**“ war es eine schwierige Entscheidung, ob man nicht das deutsche Wort „**Anweisung**“ verwenden soll. Da es **nicht verwirrend** klingt wurde sich dazu entschieden überall das deutsche Wort „**Anweisung**“ zu verwenden.

³⁹Später werden unter anderem sogenannte **Abstrakte Syntaxbäume** (Definition ??) zur Veranschaulichung gezeigt, die vom PicoC-Compiler als **Zwischenstufen** der Kompilierung generiert werden. Diese **Abstrakten Syntaxbäume** sind in der Implementierung des PicoC-Compilers in **Englisch** benannt, daher wurden ihre **Bezeichner** in **Englisch** belassen.

1.5.2 Aufbau der schriftlichen Arbeit

Der Inhalt dieser **schriftlichen Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: **Einführung**, **??**, **??** und **??**. **Zusätzlich** gibt es noch den **??**.

Das momentane Kapitel **Einführung** hatte den Zweck einen **Einstieg** in das Thema dieser Bachelorarbeit zu geben. Der **Aufbau** dieses Kapitels wurde zu Beginn **bereits erläutert**.

Im Kapitel **??** werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen und Zusammenhänge in Bezug zu **Compilern** und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

Des Weiteren wurden für **T-Diagramme** und **Formale Sprachen** eigene Unterkapitel erstellt. Für **T-Diagramme** wurde ein eigenes Unterkapitel erstellt, da sie häufig in dieser schriftlichen Ausarbeitung verwendet werden und die **T-Diagramm Notation** nicht allgemein bekannt ist. Für **Formale Sprachen** wurde ein eigenes Unterkapitel erstellt, da für den Gutachter Prof. Dr. Scholl das Thema **Formale Sprachen** eher **fachfremd** ist, aber dieses Thema einige zentrale und wichtige Fachbegriffe besitzt, bei denen es wichtig ist die genaue **Definition** zu haben.

Im Kapitel **??** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers** erklärt. Das Kapitel ist unterteilt in die verschiedenen **Phasen der Kompilierung**, nach denen das Kapitel **Einführung** ebenfalls unterteilt ist. Dadurch, dass die Kapitel **Theoretische Grundlagen** und **Implementierung** eine **ähnliche Kapiteileinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln.

Im Kapitel **??** wird ein **Überblick** über die **wichtigsten Funktionalitäten** des PicoC-Compilers gegeben, indem anhand **kleiner Anleitungen** gezeigt wird, wie man diese verwendet. Des Weiteren wird darauf eingegangen, wie die **Qualitätssicherung** für den **PicoC-Compiler** umgesetzt wurde, also wie gewährleistet wird, dass der **PicoC-Compiler** funktioniert wie erwartet. Zum Schluss wird auf **Erweiterungsideen** eingegangen, bei denen es interessant wäre diese noch im **PicoC-Compiler** zu implementieren.

Im **??** werden einige **Details der RETI-Architektur**, **Sonstigen Definitionen** und das Thema **Bootstrapping** angesprochen. Der **Appendix** dient als eine Lagerstätte für **Definitionen**, **Tabellen**, **Abbildungen** und ganze **Unterkapitel**, die bei **Interesse zur weiteren Vertiefung** da sind und zum Verständnis der anderen Kapitel **nicht notwendig** sind. Damit der **Rote Faden** in dieser schriftlichen Ausarbeitung der Bachelorarbeit erkennbar bleibt und der **Lesefluss** nicht gestört wird, wurden alle diese Informationen in den **Appendix** ausgelagert.

Die **Sonstigen Definitionen** und das Thema **Bootstrapping** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Generell wurde immer versucht **Parallelen** zur Implementierung **echter** Compiler zu ziehen. Die **Erklärungen** und **Definitionen** hierfür wurden allerdings in den **?? ausgelagert**. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition **??**) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden. Es sollte sich an die **vorgegebenen Paradigmen** aus der Vorlesung C. Scholl, „Betriebssysteme“ gehalten werden.

Literatur

Online

- *clang: C++ Compiler*. URL: <http://clang.org/> (besucht am 29.07.2022).
- *Clockwise/Spiral Rule*. URL: <https://c-faq.com/decl/spiral.anderson.html> (besucht am 29.07.2022).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: <https://inkscape.org/> (besucht am 03.08.2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*. URL: <https://www.coder-welten.de/einstieg/variablen-in-c-3.html> (besucht am 11.08.2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: <https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/> (besucht am 18.07.2022).

Bücher

- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand*. 1. Aufl. Wiley, 20. Nov. 2012.

Vorlesungen

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03.08.2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Peter. „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).

Sonstige Quellen

- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03.08.2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: <https://github.com/tueda/makefile4latex> (besucht am 03.08.2022).