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

Betreung:
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.

Heitersheim, 24. August
Ort, Datum

Jürgen Mattheis
Unterschrift

Danksagungen

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

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 Selbverständlichkeit 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 heraus, was man bereits als eine gute Betreuung bezeichnen würde.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe[1], 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. anzumerken und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbverständlichkeit, 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 meinen PicoC-Compiler implementieren 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 den PicoC-Compiler ist der Zeitplan ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegen wohl mit der Zeit äußerst kritisch geworden. Dass Prof. Dr. Scholl mir zu seinem eigenen Nachteil[2] [3] weniger Arbeit aufgebrummt hat, empfand ich als ich eine äußerst nette Geste.

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 eigensinnig 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 eigene 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

[1] Wofür ich mich auch nochmal entschuldigen will.

[2] Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

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

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 ausgeinandergesetzt 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 Mist baue. Der RETI-Emulator von Michel Giehl ist unter Link[4] zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt. Das Aufschreiben dieser schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht[5]. 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 kann, wie bestimmte Funktionalitäten einer Programmiersprache zu verwenden sind. Manch eine Funktionalität einer Programmiesprache kann in der Verwendung ziemlich wilkürlich 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.

[4] https://github.com/michel-giehl/Reti-Emulator.

[5] Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was Wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Einführung

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiel** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprachen **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 kompiliert[4]

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll. Im Fall des **GCC** funktioiert 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.

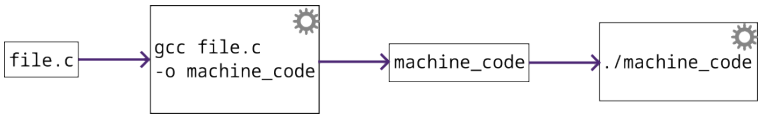


Abbildung 1.1: *Schritte zum Ausführen eines Programmes mit dem GCC.*

Der ganze Kompiliervorgang 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**.

[1] Dieser Begriff ist **nicht** weiter **relevant**.
[2] *GCC, the GNU Compiler Collection - GNU Project*.
[3] *clang: C++ Compiler*.
[4] Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C.
[5] Bei **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.

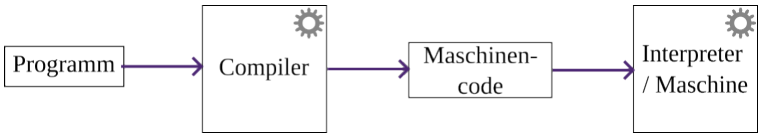


Abbildung 1.2: *Stark vereinfachte Schritte zum Ausführen eines Programmes.*

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 **Maschinensprache** 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 **Maschinensprache** 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 Kontrukte, 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-Achitektur** 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 **Maschinensprache** 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 1.1 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 **??** zusammengenommen beschrieben.

Register Kürzel	Register Ausgeschrieben	Aufgabe
`PC`	**P**rogram **C**ounter	Zeigt auf den **Maschinenbefehl**, der als nächstes ausgeführt werden soll.
`ACC`	**Acc**umulator	Für **Operanden** von Operationen oder für **temporäre** Werte.
`IN1`	**In**dexregister **1**	Hat dieselbe Aufgabe wie das `ACC`-Register.
`IN2`	**In**dexregister **2**	Hat dieselbe Aufgabe wie das `ACC`-Register.
`SP`	**S**tack**p**ointer	Zeigt immer auf die **erste freie Speicherzelle** am **Ende** des **Stacks**[a], wo als nächstes Speicher **allokiert** werden kann.
`BAF`	**B**egin **A**ktive **F**unktion	Zeigt auf den **Beginn** des **Stackframes** der aktuell **aktiven Funktion**.
`CS`	**C**odesegment	Zeigt auf den **Beginn** des **Codesegments**.
`DS`	**D**atensegment	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.

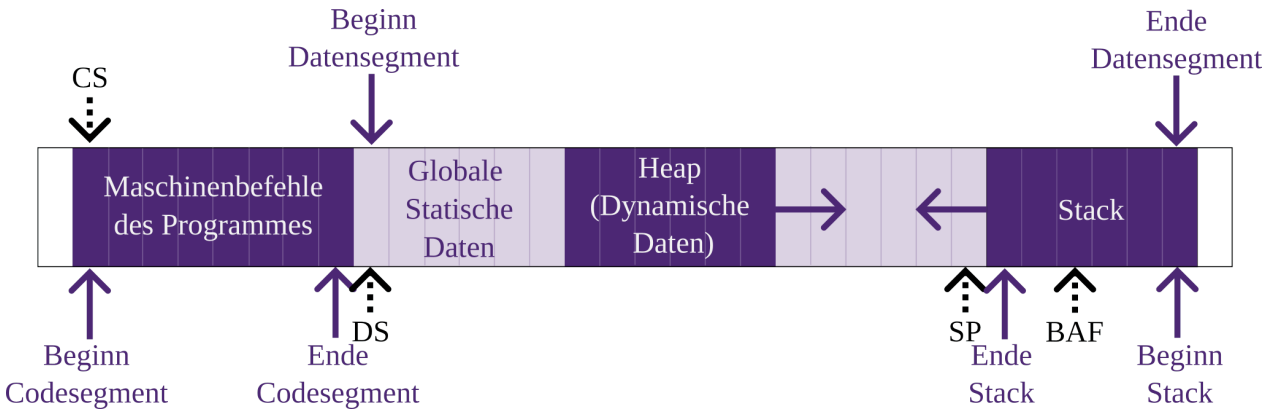


Abbildung 1.3: *Speicherorganisation.*

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} Adressen[7] 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 Empfangen[10] und Versenden[11] von Daten. Jedes der **Register** wird dabei mit einer anderen von 2^3 verschiedenen Adressen angsprochen. 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 $Bit = 4$ $Byte$. 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:

[6] Da 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.
[7] Von 0 bis $2^{32} - 1$.
[8] Kurz für **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
[9] Kurz für **U**niversal **A**synchronous **R**eceiver **T**ransmitter.
[10] Engl. **R**eceiving, daher das `R`.
[11] Engl. **T**ransmission, daher das `T`.
[12] Kurz für **S**tatic **R**andom-**A**ccess **M**emory.
[13] Ist für **Arithmetische**, **Bitweise** und **Logische Berechnungen** zuständig.

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/* comment */`.
- die **Basisdatentypen**[14] `int`, `char` und `void`.
- die **Zusammengesetzten** Datentypen[15] **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; int attr2;}`) und **Zeiger** (z.B. `int *pntr`), inklusive:
  - **Initialisierung** (z.B. `struct st st_var = {.attr1=42, .attr2={.attr={&var, &var}}}`).
  - dazugehörige **Operationen** `[i]`, `.attr`, `*` und `&`.
  - **Kombinationen** der eben genannten **Operationen** (z.B. `(*complex_var[0][1])[1].attr`) und **Datentypen** (z.B. `struct st (*complex_var[1][2])[2]`).
  - **Zeigerarithmetik** (z.B. `*(var + 2)`).
- `if(cond){ }`- und `else{ }`-**Anweisungen**[16].
- `while(cond){ }`- und `do while(cond){ };`-**Anweisungen**.
- **Arithmetische und Bitweise Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` und **unären Operatoren** `-`, $\sim$ umgesetzt sind.[17]
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.
- **Zuweisungen**, welche mithilfe des **Zuweisungsoperators** `=` umgesetzt sind, inklusive:
  - Zuweisung an **Feldelement**, **Verbundsattribut** oder **Zeigerelement** (z.B. `(*var.attr)[2] = fun() + 42`).
- **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 **??** zusammengenommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache L_C, die genauso ein Teil der Programmiersprache L_{PicoC} sind[18], werden im Folgenden genauer erläutert. Diese **Eigenheiten** werden in der Implementierung des **PicoC-Compilers** im Kapitel **??** noch eine **wichtige Rolle** spielen.

[14] Bzw. `int` und `char` werden auch als **Primitive Datentypen** bezeichnet.
[15] Bzw. engl. **compound datatypes**.
[16] Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.
[17] Theoretisch sind die Operatoren `<<`, `>>` und $\sim$ unnötig, da sie durch **Multiplikation** `*`, **Division** `/` und Anwendung des **Xor**-/\-Operators auf eine Zahl, deren **binäre Repräsentation** ein Folge von 1en **gleicher Länge** ist ersetzt werden können.
[18] Da L_{PicoC} eine **Untermenge** von L_C ist.

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 erläutertem Grund genauso für L_C.

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

Und aufgrund dessen, dass es sich 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

*Man spricht hiervon, wenn ein Programm aus einer **Folge von Anweisungen** besteht, **deren Reihenfolge** auch bestimmt in welcher **Reihenfolge** die entsprechenden **Befehle** auf einer **Maschine** ausgeführt werden.*[a][b]

[a] Thiemann, „Einführung in die Programmierung“.
[b] *Imperative Programmierung*.

Definition 1.2: Strukturierte Programmierung

*Man spricht hiervon, 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** umbedingt immer **gleich aufgebaut**.*[a][b]

[a] Thiemann, „Einführung in die Programmierung“.
[b] *Strukturierte Programmierung*.

Definition 1.3: Prozedurale Programmierung

*Man spricht hiervon, wenn Programme z.B. mittels **Funktionen** in **überschaubare Unterprogramme**[a] aufgeteilt werden, 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][c]

[a] Bzw. auch **Prozeduren** genannt.
[b] Thiemann, „Einführung in die Programmierung“.
[c] *Prozedurale Programmierung*.

[19] Werden später im Kapitel **??** genauer erklärt.

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 `(*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, in beiden Fällen 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**.

[20] https://c-faq.com/decl/spiral.anderson.html

```
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`.

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

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 b]

[a] Bast, „Programmieren in C“.
[b] *Evaluation strategy*.

```
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}};
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 1.11[22] 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.

[22] Unten im Code schauen.

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**.*[ab]

[a] Bast, „Programmieren in C“.
[b] *Evaluation strategy*.

```
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 }
```

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 **Parametereihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei **Funktionsprototypen** mit dem **gleichen Funktionsbezeichner** zu haben.*[abc]

[a] Der **Funktionsprototyp** ist von der **Funktionssignatur** zu unterschieden, 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 **Datentpyen**

von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Sichtbarkeitsbereiche** und **Klassentypen** usw. unterschieden.
[b] *What is the difference between function prototype and function signature?*
[c] *Funktionsprototyp*.

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 d]

[a] Über das **Schlüsselwort** `extern` lassen sich in der Programiersprache L_C Veriablen **deklarieren**, **ohne** sie zu **definieren**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. Scholl, „Einführung in Embedded Systems“.
[d] *Deklaration (Programmierung)*.

Definition 1.8: Definition

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

[a] Im Fall des PicoC-Compilers im Abschnitt für die **Globalen Statischen Daten** oder auf dem **Stack**.
[b] P. Scholl, „Einführung in Embedded Systems“.

```
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 b]

[a] Thiemann, „Einführung in die Programmierung“.
[b] *Variable (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, 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 Unterschied[24] kompilieren können.

Des Weiteren 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.

[23] Da 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}.
[24] Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinensprachen** zu welchen kompiliert wird und der unterschiedlichen **Kommandozeilenoptionen** und **Fehlermeldungen**.

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. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter Link[26] zu finden.

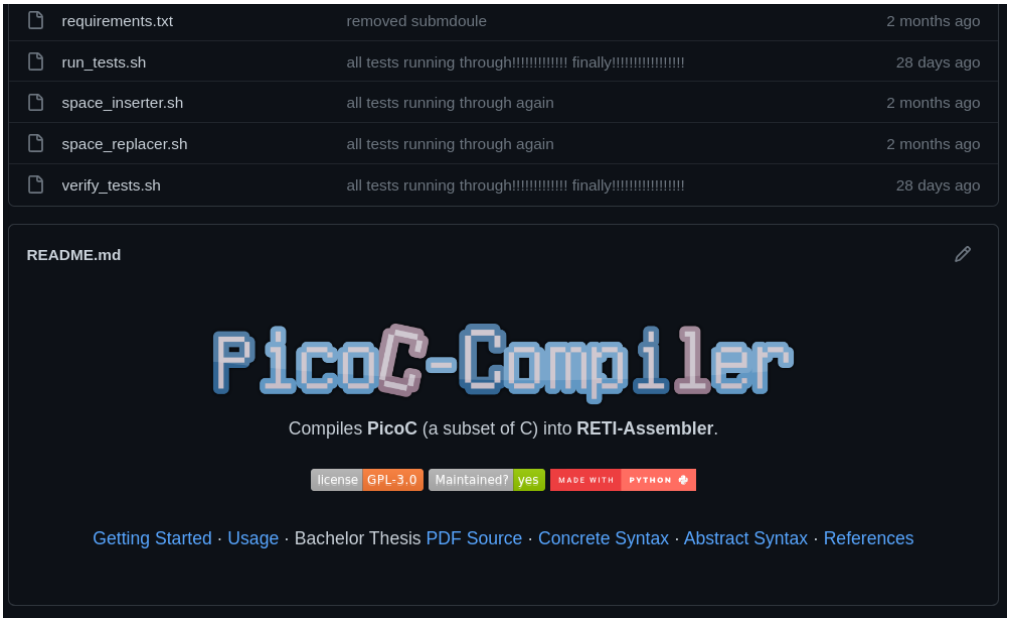


Abbildung 1.4: *README.md im Github Repository der Bachelorarbeit.*

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 Link[27] zu finden. Die **PDF**-Datei der schriftlichen Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode** automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert. Der **Latexquellcode** ist unter Link[28] veröffentlicht.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[29] zu finden[30]. Die Grafiken, die nicht mittels der Tikz Bibltiothek 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 Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[32] zu finden.

[25] https://github.com/matthejue/PicoC-Compiler.
[26] https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971.
[27] https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.
[28] https://github.com/matthejue/Bachelorarbeit.
[29] https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.
[30] Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.
[31] Developers, *Draw Freely — Inkscape*.
[32] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

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 **eingelesen** wurden. Im Kapitel **??** werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** in Form von Codebeispielen gezeigt[33].

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 eingelesen. Alle 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 sind[35].

1.5.1 Stil 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 gelesenen 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 eine der **zentralen Fragen**, die ein Leser in erster Linie unter anderem zum initialen **wirklichen Verständnis** eines Themas beantwortet braucht[36], 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**.

Viele der in dieser schriftlichen Ausarbeitung verwendeten **Definitionen** wurden mit **Wikipedia abgeglichen**. Wikipedia diente dabei allerdings immer nur dazu manche Definitionen **aufzubessern** und **nicht** als **Primärquelle**. Alle Definitionen, bis auf die in „Anführungszeichen“ sind in **eigenen Worten** formuliert, aber **gute Aspekte** mancher Wikipedia-Definitionen sind in die Definitionen in dieser schriftlichen Ausarbeitung **miteingeflossen**. Die **Wikipedia-Definitionen**, an denen sich **orientiert** wurde, sind im **Literaturverzeichnis** in der Kategorie **„Sonstige Quellen“** zu finden.

Des Weiteren gibt es `Anmerkung`'s-Kästen, welche kleine **Anmerkungen** enhalten, die über **Konventionen**

[33] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist. Später mehr dazu.
[34] https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.
[35] Es 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.
[36] Vor allem am **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[37] Das ist ein **Beispiel** für eine **Fußnote**.

aufklären sollen, vor **Fallstricken warnen**, die leicht zur Verwirrung führen können oder Informationen bei **tiefergehenderem Interesse** oder für den besseren **Überblick** enthalten. Der Inhalt dieser `Anmerkung`'s-Kästen ist allerdings zum Verständnis dieser Arbeit **nicht essentiel** 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** vorkommen[39].

1.5.2 Aufbau der schriftlichen Arbeit

Der Inhalt dieser **schriftlichen Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: Einführung, **??**, **??** und Ergebnisse und Ausblick. **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 dennen das Kapitel **Einführung** ebenfalls unterteilt ist. Dadurch, dass die Kapitel **theoretische Grundlagen** und **Implementierung** eine **ähliche Kapiteleinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln.

Im Kapitel Ergebnisse und Ausblick 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 **Qualitätsicherung** 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 Verständis 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

[38] Bei 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.

[39] Später 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**.

Appendix ausgelaggert.

Die **Sonstigen Defintionen** 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 **??** **ausgelaggert**. 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.

2 Ergebnisse und Ausblick

Zum Schluss soll ein **Überblick** über das Resultat dessen, was im Kapitel **??** implementiert wurde gegeben werden. Im Unterkapitel 2.1 wird darauf eingegangen, ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel Einführung alle implementiert werden konnten. Daraufhin wird mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten zugegriffen werden kann und es wird auch auf Funktionalitäten **anderer mitimplementierter Tools** eingegangen. Im Unterkapitel 2.2 wird aufgezeigt, was zur **Qualitätssicherung** implementiert wurde, um zu gewährleisten, dass der **PicoC-Compiler** die Kompilierung der **Programmiersprache** L_{PicoC} in **Syntax** und **Semantik identisch** zur entsprechenden **Untermenge** der Programmiersprache L_C umsetzt. Als allerletztes wird im Unterkapitel 2.3 ein Ausblick gegeben, wie der PicoC-Compiler **erweitert** werden könnte.

2.1 Funktionsumfang

Alle Funktionalitäten, die in Kapitel Einführung erläutert und versprochen wurden, konnten in dieser **Bachelorarbeit** implementiert werden. In Kapitel **??** wurde die Umsetzung **aller** dieser Funktionalitäten erklärt. Während der **Funktionsumfang** des **PicoC-Compiler** zum Stand des **Bachelorprojektes** noch sehr beschränkt war und einzig eine **Strukturierte Programmierung** mit `if(cond) { } else { }`, `while(cond) { }` usw. erlaubte und komplexere Programme nur mit **viel Aufwand** und **unübersichtlichem Spaghetticode** implementierbar waren, erlaubt es der **PicoC-Compiler** nachdem er in der **Bachelorarbeit** um **Felder**, **Zeiger**, **Verbunde** und **Funktionen** erweitert wurde mittels der **Funktionen** eine **Prozedurale Programmierung** umzusetzen. **Prozedurale Programmierung** zusammen mit der Möglichkeit **Felder**, **Zeiger** und **Verbunde** zu verwenden trägt zu einem **geordneteren**, **intuitiv verständlicheren** und **übersichtlicheren** Code bei.

Bei der Implementierung des **PicoC-Compilers** wurden verschiedene **Kommandozeilenoptionen** und **Modes** implementiert. Diese werden in den folgenden Unterkapiteln 2.1.1, 2.1.3 und 2.1.4 mithilfe kurzer **Anleitungen** erklärt.

Die kurzen **Anleitungen** in dieser **schriftlichen Ausarbeitung** der Bachelorarbeit sollen nur zu einem **schnellen, grundlegenden Verständnis** der Verwendung des **PicoC-Compilers** und seiner **Kommandozeilenoptionen** und **Befehle** beihelfen, sowie zum Verständnis der **weiteren implementierten Tools**. Alle weiteren **Kommandozeilenoptionen** und **Befehle** sind für die Verwendung des PicoC-Compilers **unwichtig** und erweisen sich nur in **speziellen Situationen** als nützlich, weshalb für diese auf die **ausführlichere Dokumentation** unter Link[1] verwiesen wird.

2.1.1 Kommandozeilenoptionen

Will man einfach nur ein **Programm** `program.picoc` kompilieren ist das mit dem **PicoC-Compiler** genauso **unkompliziert**, wie mit dem **GCC** durch einfaches **Angeben der Datei**, die kompiliert werden soll: `> picoc_compiler program.picoc`. Als Ergebnis des Kompiliervorgangs wird eine Datei `program.reti` mit dem entsprechenden **RETI-Code** erstellt, wobei für die **Benennung der Datei** einfach nur der

[1] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

Basisname der Datei `program` an eine neue **Dateiendung** `.reti` angehängt wird[2].

Daneben gibt es allerdings auch die Möglichkeit **Kommandozeilenoptionen** `<cli-options>` in der Form `> picoc_compiler <cli-options> program.picoc`[3] mitanzugeben, von denen die **wichtigsten** in Tabelle 2.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumenation** unter Link[4] nachgelesen werden. Die die letzte Spalte gibt den **Standardwert** an, der bei der normalen Nutzung des **PicoC-Compilers** gesetzt ist.[5]

Kommandozeilen-option	Beschreibung	Standard-wert
`-i`, `--intermediate_stages`	Gibt **Zwischenstufen** der Kompilierung in Form der verschiedenen **Tokens**, **Ableitungsbäume**, **Abstrakten Syntaxbäume** der verschiedenen **Passes** in Dateien mit **entsprechenden Dateiendungen** aber gleichem **Basisnamen** aus. Im **Shell-Mode** erfolgt **keine** Ausgabe in Dateien, sondern nur im **Terminal**.	`false`, most_used: `true`
`-p`, `--print`	Gibt alle **Dateiausgaben** auch im **Terminal** aus. Diese Option ist im **Shell-Mode** dauerhaft aktiviert.	`false` (`true` im **Shell-Mode** und für den most_used-Befehl)
`-v`, `--verbose`	Fügt den verschiedenen **Zwischenschritten der Kompilierung**, unter anderem auch dem finalen RETI-Code **Kommentare** hinzu. Diese Kommentare beinhalten eine **Anweisung** oder einen **Befehl** aus einem **vorherigen Pass**, der durch die darunterliegenden Anweisungen oder Befehle **ersetzt** wurde. Wenn die `--run`-Option aktivert ist, wird der **Zustand** der virtuellen RETI-CPU **vor** und **nach jedem Befehl** angezeigt.	`false`
`-vv`, `--double_verbose`	Hat **dieselben Effekte**, wie die `--verbose`-Option, aber bewirkt zusätzlich **weitere Effekte**. **PicoC-Knoten** erhalten bei der Ausgabe als zusammenhängende **Abstrakte Syntaxbäume** zustätzliche **runde Klammern**, sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In **Fehlermeldungen** werden **mehr Tokens** angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der `--intermediate_stages`-Option werden in den dadurch ausgegebenen **Abstrakten Syntaxbäumen** zusätzlich versteckte Attribute, die **Informationen** zu **Datentypen** und für **Fehlermeldungen** beinhalten angezeigt.	`false`
`-h`, `--help`	Zeigt die **Dokumentation**, welche ebenfalls unter Link gefunden werden kann **im Terminal** an. Mit der `--color`-Option kann die **Dokumentation** mit **farblicher Hervorhebung** im Terminal angezeigt werden.	`false`

Tabelle 2.1: *Kommandozeilenoptionen, Teil 1.*

[2] Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der **festen** Bezeichnung `a.out` erstellt.
[3] Die **Kommandozeilenoptionen** `<cli-options>` haben **keine** feste Position, es geht ebenfalls `picoc_compiler program.picoc <cli-options>`.
[4] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.
[5] In grau ist unter most_used des Weiteren der **Standardwert** bei der Verwendung des `most_used`-Befehls angegeben.

Kommandozeilen-option	Beschreibung	Standard-wert
`-l`, `--lines`	Es lässt sich einstellen, **wieviele Zeilen** rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	`2`
`-c`, `--color`	Aktiviert **farbige Ausgabe**.	`false`, most_used: `true`
`-t`, `--thesis`	**Filtert** für die **Codebeispiele** in dieser schriftlichen Ausarbeitung der Bachelorarbeit bestimmte **Kommentare** in den Abstrakten Syntaxbäumen heraus, damit alles **übersichtlich** bleibt.	`false`
`-R`, `--run`	Führt die **RETI-Befehle**, die das Ergebnis der Kompilierung sind mit einer **virtuellen RETI-CPU** aus. Wenn die `--intermediate_stages`-Option aktiviert ist, wird eine Datei `<basename>.reti_states` erstellt, welche den **Zustsand der RETI-CPU** nach dem **letzten** ausgeführten RETI-Befehl enthält. Wenn die `--verbose-` oder `--double_verbose`-Option aktiviert ist, wird der Zustand der RETI-CPU **vor** und **nach** jedem Befehl auch noch zusätlich in die Datei `<basename>.reti_states` ausgegeben.	`false`, most_used: `true`
`-B`, `--process_begin`	Setzt die **relative Adresse**, wo der **Prozess** bzw. das **Codesegment** für das ausgeführte Programm **beginnt**.	`3`
`-D`, `--datasegment_size`	Setzt die Größe des **Datensegments**. Diese Option muss mit **Vorsicht** gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die **Globalen Statischen Daten** und der **Stack** miteinander **kollidieren**.	`32`

Tabelle 2.2: *Kommandozeilenoptionen, Teil 2.*

Alle **kleingeschriebenen** Kommandozeilenoptionen, wie `-i`, `-p`, `-v` usw. betreffen den **PicoC-Compiler** und alle **großgeschriebenen** Kommandozeilenoptionen, wie `-R`, `-B`, `-D` usw. betreffen den **RETI-Interpreter**.

2.1.2 RETI-Interpreter

Um den nach der Kompilierung durch den **PicoC-Compiler** generierten **RETI-Code** auch **ausführen** zu können, wurde zusätzlilch ein **RETI-Interpreter** implementiert. Die Möglichkeit den **RETI-Code** auch **ausführen** zu können, ist für die **Qualitätssicherung**, die in Unterkapitel 2.2 genauer erklärt wird notwendig. Ein **Programm**, dass in der Sprache L_{PicoC} geschrieben ist, lässt sich mit der Kommandozeilenoption `-R` im Anschluss an die **Kompilierung** durch den **PicoC-Compiler**, mit dem **RETI-Interpreter** ausführen. Der **vollständige Befehl** hierfür lautet `> picoc_compiler -R program.picoc`, wie es bereits aus dem vorherigen Unterkapitel 2.1.1 bekannt ist.

Der **RETI-Interpreter** wird an der Stelle, an welcher normalerweise die **RETI-CPU** den **RETI-Code** ausführen würde eingesetzt. In Abbildung 2.1 ist in einem weiteren **T-Diagramm** (siehe Unterkapitel **??**) dargestellt, wie anstelle einer **RETI-CPU** der **RETI-Intepreter** den **RETI-Code** ausführt, indem er die **Semantik** der **RETI-Befehle** simuliert. In dem **T-Diagramm** ist der Vollständigkeit halber auch dargestellt, dass der **PicoC-Compiler** und **RETI-Interpreter** beide mithilfe des **Python-Interpreters** ausgeführt werden, der wiederum z.B. auf einer X_{86_64}-**Maschine** ausgeführt wird[6].

[6] Es gibt neben X_{86_64} z.B. noch Architekturen, wie *ARM* (wird vor allem bei **Mobiltelefonen** eingesetzt) oder $RISC{-}V$ (gezielt **simpel** gehaltene Architektur unter **BSD-Lizenz**).

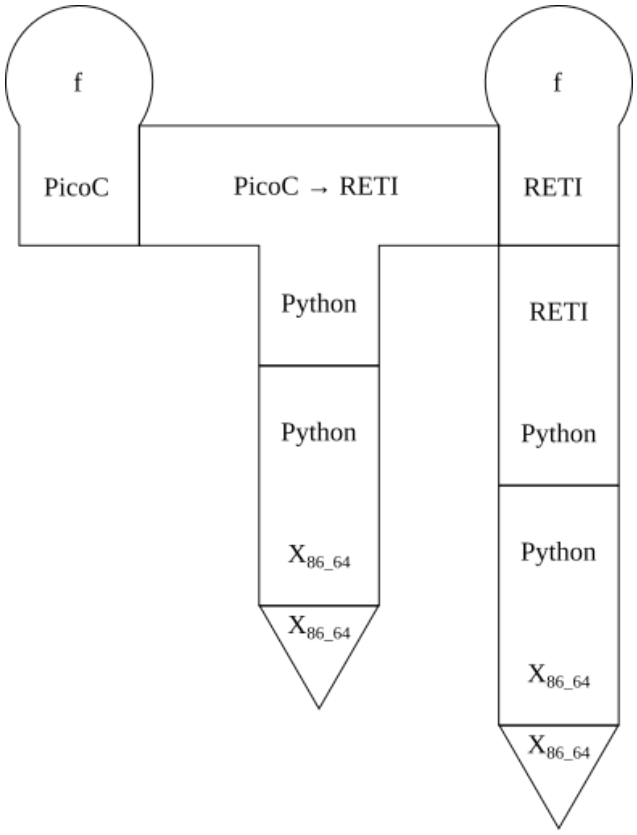


Abbildung 2.1: *Ausführung von RETI-Code mit dem RETI-Interpreter.*

Für den **RETI-Interpreter** konnte sehr viel aus der Architektur des PicoC-Compilers **wiederverwendet** werden. Der **RETI-Interpreter** operiert genauso, wie der **PicoC-Compiler** auf **Abstrakten Syntaxbäumen**, deren Kompositionen von RETI-Knoten er direkt **ausführt**. Beim **Ausführen**, der durch RETI-Knoten dargestellten Befehle, ändert sich der **Zustand** von **Registern**, **SRAM**, **UART** und **EPROM**, die im RETI-Interpreter **simuliert** werden. Die genaue **Umsetzung** des **RETI-Interpreters** wird in dieser schriftlichen Ausarbeitung der Bachelorarbeit allerdings **nicht** genauer **erläutert**, da Interpreter **nicht** das **Thema** dieser Bachelorarbeit sind.

2.1.3 Shell-Mode

Will man z.B. eine **Folge von Anweisungen** in der Programmiersprache L_{PicoC} **schnell** kompilieren ohne eine Datei erstellen zu müssen, so kann der **PicoC-Compiler** im sogenannten **Shell-Mode** aufgerufen werden. Hierzu wird der PicoC-Compiler **ohne Argumente** `> picoc_compiler` aufgerufen, wie es in Code 2.1 zu sehen ist.

Mit dem `> compile <cli-options> <seq-of-stmts>`-Befehl (oder der **Abkürzung** `cpl`) kann **PicoC-Code** zu **RETI-Code** kompiliert werden. Die Kommandozeilenoptionen `<cli-options>` sind dieselben, wie wenn der Compiler **direkt** mit Kommandozeilenoptionen aufgerufen wird. Die **wichtigsten** dieser **Kommandozeilenoptionen** sind in Tabelle 2.1 angegeben. Die angegebene **Folge von Anweisungen** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem Befehl `> quit` kann der **Shell-Mode** wieder **verlassen** werden.

```
> picoc_compiler
PicoC Shell. Enter `help` (shortcut `?`) to see the manual.
PicoC> cpl "6 * 7;";
--------------------------- RETI ---------------------------
SUBI SP 1;
LOADI ACC 6;
STOREIN SP ACC 1;
SUBI SP 1;
LOADI ACC 7;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
LOADIN BAF PC -1;

Compilation successfull

PicoC> quit
```

Code 2.1: *Shellaufruf und die Befehle* `compile` *und* `quit`*.*

Wenn man möglichst alle nützlichen **Kommandozeilenoptionen** direkt aktiviert haben will, bei denen es **keinen** Grund gibt sie nicht mitanzugeben, kann der Befehl `> most_used <cli-options> <seq-of-stmts>` (oder seine **Abkürzung** `mu`) genutzt werden. Auf diese Weise müssen diese Kommandozeilenoptionen **nicht** wie beim `compile`-Befehl jedes mal **selbst** angeben werden. In Tabelle 2.1 sind in grau die **Standardwerte** der einzelnen **Kommandozeilenoptionen** angegeben, die bei dem Befehl `most_used` gesetzt werden. In Code 2.2 ist der `most_used`-Befehl in seiner Verwendung zu sehen.

Dadurch, dass die `--intermediate_stages`-, `print`- und die `--run`-Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie **Tokens**, **Ableitungsbaum**, **Passes** usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls in das Terminal **ausgegeben**. Aus **Platzgründen** ist das meiste allerdings mit `'...'` ausgelassen.

```
PicoC> mu "int var = 42;";
--------------------------- Code ---------------------------
// stdin.picoc:
void main() {int var = 42;}
-------------------------- Tokens --------------------------
...
--------------------- Derivation Tree ----------------------
...
----------------- Derivation Tree Simple ------------------
...
----------------- Abstract Syntax Tree -------------------
...
---------------------- PicoC Shrink ------------------------
...
---------------------- PicoC Blocks ------------------------
...
----------------------- PicoC Mon --------------------------
...
---------------------- Symbol Table ------------------------
...
----------------------- RETI Blocks ------------------------
...
----------------------- RETI Patch -------------------------
...
--------------------------- RETI ---------------------------
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
LOADIN BAF PC -1;
------------------------- RETI Run -------------------------
...

Compilation successfull
```

Code 2.2: *Shell-Mode und der Befehl* `most_used`.

Im **Shell-Mode** kann der **Cursor** mit den Pfeiltasten ← und → bewegt werden. In der **Befehlshistorie** kann sich mit den Pfeiltasten ↑ und ↓ **rückwarts** und **vorwärts** bewegt werden. Mit Tab kann ein Befehl **automatisch vervollständigt** werden.

Es gibt für den **Shell-Mode** noch **weitere Befehle**, wie `color_toggle`, `history` etc. und **kleinere Funktionalitäten** für die Shell, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wird allerdings auf die **Dokumentation** unter Link[7] verwiesen, welche auch über den Befehl `> help` angezeigt werden kann.

2.1.4 Show-Mode

Der **Show-Mode** ist ein Nebenprodukt der Implementierung des **PicoC-Compilers**. Dieser **Mode** wurde eigentlich nur implementiert, um beim **Testen** des PicoC-Compilers **Bugs** bei der Generierung des **RETI-**

[7] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.

Code zu **inspizieren**. Das ganze ist so umgesetzt, dass im Terminal eine **virtuelle RETI-CPU** angezeigt wird, mit dem kompletten, momentanen **Zustand** in Form aller **Register**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen**.

Die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen**, bringt auch einen großen **Lerneffekt** mit sich, weshalb der **Show-Mode** noch **weiterentwickelt** wurde, sodass auch **Studenten** ihn auf unkomplizierte Weise nutzen können.

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `> make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen** `<more-options>`, die für die **Makefile** gesetzt werden können, sind in Tabelle 2.3 aufgelistet.

Kommandozeilenoption	Beschreibung	Standardwert
`FILEPATH`	**Pfad zur Datei**, die im **Show-Mode** angezeigt werden soll.	$\emptyset$
`TESTNAME`	**Name des Tests**. Alles andere als der **Basisname**, wie die **Dateiendung** wird abgeschnitten.	$\emptyset$
`EXTENSION`	**Dateiendung**, die an `TESTNAME` angehängt werden soll, damit daraus z.B. `./tests/TESTNAME.EXTENSION` wird.	`reti_states`
`NUM_WINDOWS`	**Anzahl Fenster** auf die ein Dateiinhalt **verteilt** werden soll.	`5`
`VERBOSE`	Möglichkeit für eine **ausführlichere Ausgabe** die **Kommandozeilenoption** `-v` oder `-vv` zu aktivieren.	$\emptyset$
`DEBUG`	Möglichkeit die **Kommandozeilenoption** `-d` zu aktivieren, um bei `make test-show TESTNAME=<testname>` den **Debugger** für den entsprechenden **Test** `<testname>` zu starten.	$\emptyset$

Tabelle 2.3: *Makefileoptionen.*

Alternativ kann der **Show-Mode** mit dem Befehl `make test-show TESTNAME=<testname> <more-options>` auch für einen der geschriebenen **Tests** im Ordner `/tests` gestartet werden. Der **Test** wird bei diesem Befehl **erst ausgeführt** und dann der **Show-Mode** gestartet.

Der **Show-Mode** nutzt den Terminal Texteditor **Neovim**[8], um einen **Dateiinhalt** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 2.2 zu sehen ist. Für den **Show-Mode** wird eine eigene **Konfiguration für Neovim** verwendet, welche in der **Konfigurationsdatei** `/interpr_showcase.vim` spezifiziert ist.

Gedacht ist der **Show-Mode** vor allem dafür, etwas ähnliches wie ein **RETI-Debugger** zu sein und wird daher standardmäßig bei **Nicht-Angabe** einer `EXTENSION` auf die **Datei** oder den **Test** `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen, über mehrere Fenster **verteilt** anzuzeigen, indem `EXTENSION` auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandt, indem die verschiedenen **Zustände** der RETI-CPU **nicht** zur **Laufzeit** des **Show-Mode** berechnet werden, sondern schon **zuvor** berechnet wurden und nacheinander in die Datei `<program>.reti_states` ausgegeben wurden. Der **Show-Mode** macht nichts anderes, als immer an die Stelle zu springen, an welcher der **nächste Zustand** anfängt. Durch Drücken von `Tab` und `⇑ −Tab` können auf diese Weise die **verschiedenen Zustände** der RETI-CPU **vor** und **nach** der Ausführung eines

[8] *Home - Neovim*.

Befehls **angezeigt** werden.

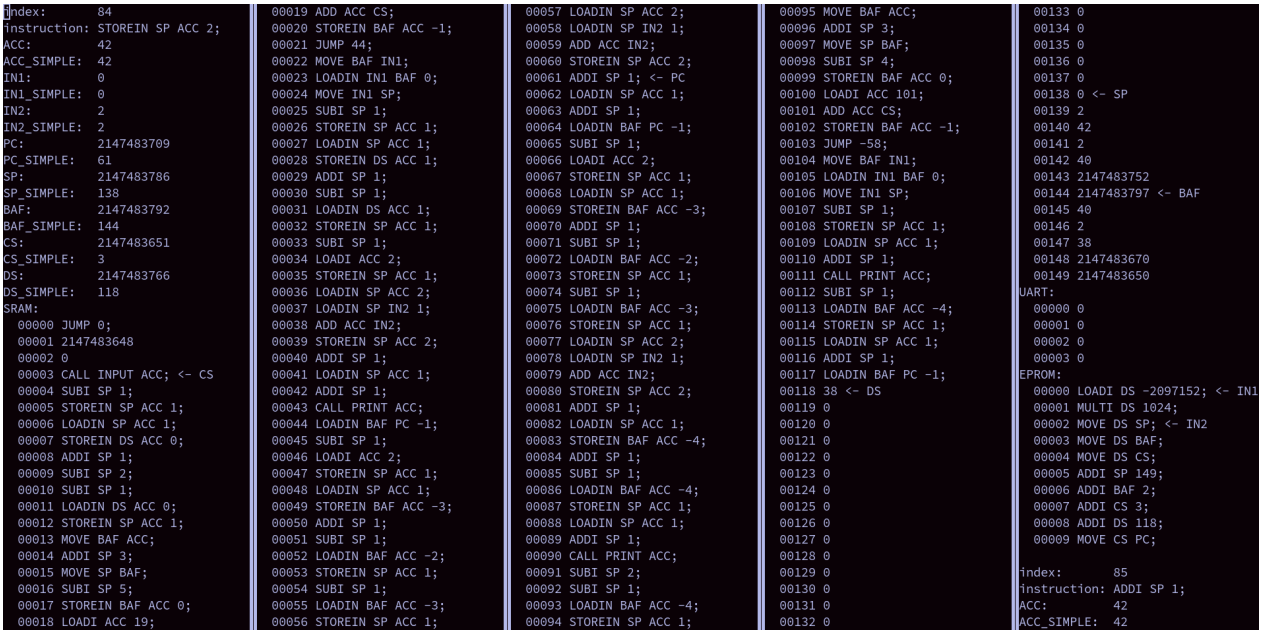


Abbildung 2.2: *Show-Mode in der Verwendung.*

Zur **besseren Orientierung** wird für **alle Register** ein mit der Registerbezeichnung beschrifteter **Zeiger** `<- REG` an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert** im entsprechenden Register nach der **Memory Map** dem **Adressbereich** von **EPROM**, **UART** oder **SRAM** entspricht.

Durch Drücken von `Esc` oder `q` kann der **Show-Mode** wieder verlassen werden. Es gibt für den **Show-Mode** noch viele **weitere Tastenkürzel**, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** aller **weiteren Tastenkürzel** wird allerdings auf die **Dokumentation** unter Link[9] verwiesen. Des Weiteren stehen durch die Nutzung des Terminal Texteditors **Neovim** auch alle **Funktionalitäten** dieses mächtigen Terminal Texteditors zur Verfügung, welche mittels der Eingabe von `:help` **nachgelesen** werden können oder mittels der Eingabe von `:Tutor` mithilfe einer kurzen **Einführungsanleitung erlernt** werden können.

2.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm **??**[10] genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 2.2.1 definieren. Ein **beliebiges** Testprogramm P_{PicoC} in der Sprache L_{PicoC} muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm P_{RETI} in der Sprache L_{RETI}, trotz der **unterschiedlichen Sprache**.

Die Beziehungen im Diagramm 2.2.1 werden mithilfe von **Tests** verifziert. Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis `/tests` bzw. unter Link[11] zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 2.4.

[9] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.
[10] In Unterkapitel **??**.
[11] https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

Testkategorie	Beschreibung
`basic`	**Einfache** Tests, welche die **grundlegenden Funktionalitäten** des PicoC-Compilers testen.
`advanced`	Tests, die **Spezialfälle** und **Kombinationen** verschiedener Funktionalitäten des PicoC-Compilers testen.
`hard`	Tests, die **lang** und **komplex** sind. Für diese Tests müssen die Funktionalitäten des PicoC-Compilers in **perfekter Harmonie** miteinander funktionieren.
`example`	Tests, die **bekannte Algorithmen** darstellen und daher als gutes, **repräsentatives Beispiel** für die Funktionsfähigkeit des PicoC-Compilers dienen.
`error`	Tests, die **Fehlermeldungen** testen. Für diese Tests wird **keine Verifikation** ausgeführt.
`exclude`	Tests, für welche aufgrund vielfältiger Gründe **keine Verifikation** ausgeführt werden soll.
`thesis`	Tests, die vorher **Codebeispiele** für diese schriftliche Ausarbeitung der Bachelorarbeit waren und etwas **umgeschrieben** wurden, damit nicht nur das Durchlaufen dieser Tests getestet wird.
`tobias`	Tests, die der Betreuer dieser Bachelorarbeit, M.Sc. **Tobias** Seufert geschrieben hat.

Tabelle 2.4: *Testkategorien.*

Dass ein Programm P_{PicoC} und das Programm P_{RETI}, welches das **kompilierte** P_{PicoC} ist nach Diagramm 2.2.1 die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn die Tests so konstruiert sind, dass es sehr **unwahrscheinlich** ist, zufällig bei der gewählten Eingabe die spezifische Ausgabe zu erhalten. Wenn **immer mehr Tests**, die alle einen unterschiedlichen **Teil der Semantik** der Sprache L_{PicoC} abdecken vorliegen, bei denen die jeweiligen Programme P_{PicoC} und P_{RETI} interpretiert die **gleiche Ausgabe** haben, dann kann mit **immer höherer Wahrscheinlichkeit** von einem **funktionierenden** Compiler ausgegangen werden.

Die Kante vom Testprogramm P_{PicoC} zur *Ausgabe* aus Diagramm 2.2.1 ist so umgesetzt, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat. Der **Schreiber des Tests** übernimmt die Rolle des entsprechenden **Interpreters** aus Diagramm **??**. Die **erwartete Ausgabe** `<space_seperated_output>` ist seine eigene Interpretation des **PicoC-Codes**.

Ein Beispiel für einen **Test** ist in Code 2.3 zu sehen. Die Tests werden mithilfe des Bashskripts `/run_tests.sh` **ausgeführt** oder mithilfe der `/Makefile` mit dem Befehl `> make test`, welcher einfach nur dieses Bashskript ausführt. Bei der **Ausführung** des Bashskripts `/run_tests.sh`, wird als erstes für **jeden** Test das Bashskript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert[12] und die entsprechenden Zeichenketten `<space_seperated_input>`, `<space_seperated_output>` und `<datasegment_size>` in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt. Das **letztere Skript** kann ebenfalls mit dem Befehl `> make extract` ausgeführt werden.

Die Datei `<program>.in` enthält **Eingaben**, welche durch `input()`-Funktionsaufrufe im Programm P_{PicoC} eingelesen werden. Die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben**, welche durch `print(<exp>)`-Funktionsaufrufe im Programm P_{PicoC} ausgegeben werden. Die Datei `<program>.out`, die später genauer erläutert wird, enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Testprogramms** P_{PicoC}. Die Datei `<program>.datasegment_size` enthält die **Größe des**

[12] Falls vorhanden.

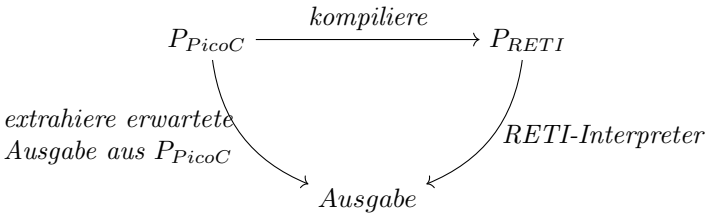
Datensegments für die Ausführung des entsprechenden Tests.

```
// in:21 2 6 7
// expected:42 42
// datasegment:4

void main() {
  print(input() * input());
  print(input() * input());
}
```

Code 2.3: *Typischer Test.*

Die Kante vom Programm P_{RETI} zur *Ausgabe* aus Abbildung 2.2.1 ist dadurch umgesetzt, dass das Programm P_{RETI} vom **RETI-Interpreter** interpretiert wird und jedes mal beim Antreffen des **RETI-Befehls** `CALL PRINT ACC`, der entsprechende **Inhalt** des `ACC`-Registers in die Datei `<program>.out` ausgegeben wird. Ein Test kann[13] die **Korrektheit** des **Teils der Semantik** der Sprache L_{PicoC}, die er abdeckt verifizieren, wenn der **Inhalt** von `<program>.out_expected` und `<program>.out` **identisch** ist.



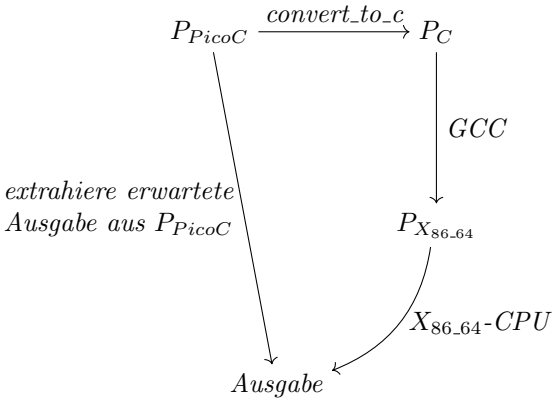
(2.2.1)

Allerdings gibt es bei dem Testverfahren, welches in Diagramm 2.2.1 dargestellt ist ein **Problem**, denn der **Schreiber der Tests** ist in diesem Fall die **gleiche Person**, die auch den **PicoC-Compiler implementiert** hat. Wenn der **Schreiber** der Tests bzw. **Implementierer** des PicoC-Compilers ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl in den **Tests** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht[14]. Die **Tests** können dann nur **bestätigen**, dass der PicoC-Compiler so implementiert wurde, wie der **Implementierer** sich die **Semenatik** der Sprache L_{PicoC} vorstellt.

Aus diesem Grund muss hier eine **weitere Maßnahme** eingeführt werden, welche in Diagramm 2.2.2 dargestellt ist. Diese Maßnahme gewährleistet, dass die *Ausgabe* sich auf jeden Fall aus der tatsächlichen **Semantik** der Sprache L_{PicoC}[15] ergeben muss. Das wird erreicht, indem wie in Diagramm 2.2.2 dargestellt ist, überprüft wird, ob die *Ausgabe* des Pfades $(P_{PicoC}, Ausgabe)$ mit der *Ausgabe* des Pfades von $(P_C, P_{X_{86\_64}}, Ausgabe)$ **identisch** ist.

Im Diagramm 2.2.2 hat die Kante *extrahiere erwartete Ausgabe aus* P_{PicoC} die **gleiche Umsetzung**, wie die entsprechende Kante in Diagramm 2.2.1, welche bereits erklärt wurde. Die Kante *GCC* ist so umgesetzt, dass der **GCC**[16] zur **Kompilierung** des Programms P_C von der Programmiersprache L_C in die Maschinensprache $L_{X_{86\_64}}$ zu $P_{X_{86\_64}}$ verwendet wird. Die Kante $X_{86\_64}{-}CPU$ ist so umgesetzt, dass sie das Programm $P_{X_{86\_64}}$ auf einer $X_{86\_64}{-}CPU$ **ausführt**, wobei hierfür zumindestens beim Computer des **Implementierers des PicoC-Compilers** eine $X_{86\_64}{-}CPU$ verwendet wird.

[13] Mit einer **bestimmten Wahrscheinlichkeit**.
[14] Welche **identisch** zu einer Teilmenge von L_C ist.
[15] Die eine **Untermenge** von L_C ist.
[16] *GCC, the GNU Compiler Collection - GNU Project*.



(2.2.2)

Das Programm P_C ergibt sich aus dem Testprogramm P_{PicoC} durch **Ausführen** des Pythonskripts `/convert_to_c.py`, welches **später näher erläutert** wird. Dieses Pythonskript lässt sich ebenfalls mithilfe der `/Makefile` und dem Befehl `> make convert` ausführen.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante $(P_C, P_{X_{86\_64}})$. Beim **GCC** handelt es sich um einen Compiler der Sprache L_C, der somit mit Ausnahme der `print()` und `input()`-Funktionen auch die Sprache L_{PicoC} kompilieren kann. Der **GCC** setzt aufgrund seiner bekanntermaßen **vielfachen Verwendung** auf der Welt und seinem **sehr langem Bestehen** seit 1987[17] [18] die **Semantik** der Sprache L_C, vor allem für die kleine Untermenge, welche L_{PicoC} darstellt mit sehr hoher Wahrscheinlichkeit **korrekt** um.

Durch das **Abgleichen** mit dem **GCC** in Diagramm 2.2.2 wird etwas wichtiges **sichergestellt**. Durch diese zweifache Überprüfung bestätigen die Tests nicht nur die Interpretation, die der **Schreiber** der Tests und **Implementierer** des PicoC-Compilers von der Semantik der Sprache L_{PicoC} hat, sondern stellen die tätsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} sicher.

Für die zweifache Überprüfung durchläuft jeder Test eine **Verifikation**, wie sie in Diagramm 2.2.2 dargestellt ist. In dieser wird **verifiziert**, ob bei der **Kompilierung** des Testprogramms P_C mit dem **GCC** und **Ausführung** des hieraus generierten X_{86_64}-Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe `// expected:<space_seperated_output>` des Testschreibers ist.

Für die **Verifikation** ist das Bashskript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt werden kann. Beim Befehl `> make test` wird dieses Bashskript **vor** dem eigentlichen Testen[19] ausgeführt. In Code 2.4 ist ein **Testdurchlauf** mit `> make test` zu sehen.

Hierbei zeigt `Verified: 80/80` an, wieviele der Tests, die überhaupt verifizierbar sind[20] sich **verifizieren** lassen, indem sie mit dem **GCC** ohne Fehlermeldung **durchlaufen** und die **erwartete Ausgabe erfüllen**. `Not verified:` gibt die **nicht** mit dem GCC **verifizierten** Tests an. `Running through: 118 / 118` zeigt an, wieviele Tests mit dem **PicoC-Compiler durchlaufen**. `Not running through:` gibt die **nicht** mit dem PicoC-Compiler **durchlaufenden** Tests an. `Passed: 118 / 118` zeigt an, bei wievielen Tests die Ausgabe beim Ausführen mit der erwarteten Ausgabe **identisch** ist. `Not passed:` zeigt die Tests an, bei denen das **nicht** der Fall ist.

[17] *History - GCC Wiki*.
[18] In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.
[19] Das **eigentliche Testen** ist hier das **Überprüfen**, ob der interpretierte RETI-Code des Tests, der vom PicoC-Compiler kompiliert wurde die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.
[20] Also **alle** Tests aus den **Kategorien** `basic`, `advanced`, `hard`, `example`, `thesis` und `tobias`.

```
> make test
==================================
= ./tests/basic_array_init.picoc =
==================================
...
==================================
=          Verification          =
==================================
./tests/basic_array_init.c
...
==================================
=             Results            =
==================================
Verified: 80 / 80
Not verified:
Running through: 118 / 118
Not running through:
Passed: 118 / 118
Not passed:
```

Code 2.4: *Testdurchlauf.*

Der Befehl `> make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` `TESTNAME`, `VERBOSE` und `DEBUG` aus Tabelle 2.3 kombinieren.

Das Pythonskript `/convert_to_c.py` ist notwendig, da L_{PicoC} sich bei den Funktionen `print(<exp>)` und `input()` von der **Syntax** der Sprache L_C **unterscheidet**. Es muss z.B. `printf("%d", 12)` anstelle von `print(12)` geschrieben werden. Für die Sprache L_{PicoC} erfüllen die Funktionen `print(<exp>)` und `input()` nur den **Zweck**, dass sie zum **Testen des PicoC-Compilers** gebraucht werden. Über die Funktion `input()` soll es möglich sein, für eine bestimmte **Eingabe** die **Ausgabe** über die Funktion `print(<exp>)` testen können. Aus diesem Grund ist es notwendig die **Syntax** dieser Funktionen in L_C zu übersetzen.

Die Funktion `print(<exp>)` wird vom Pythonskript `/convert_to_c.py` zu `printf("%d", <exp>)` **übersetzt**. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibliothek** `<stdio.h>` eingebunden werden. Bei der Funktion `input()` wurde **nicht** der aufwändige **Umweg** genommen, die Funktion `input()` durch ihre entsprechende Funktion in der Sprache L_C zu ersetzen. Es geht viel direkter, indem **nacheinander** die `input()`-Funktionen durch entsprechende Eingaben aus der Datei `<program>.in` ersetzt werden. Man schreibt einfach **direkt** die Werte hin, welche die `input()`-Funktionen normalerweise einlesen sollten.

2.3 Erweiterungsideen

Nachdem der **Funktionsumfang** des **PicoC-Compilers** in Unterkapitel 2.1 erläutert wurde, wird in diesem Unterkapitel die Diskussion geführt, wie sich dieser in Zukunft vielleicht noch **erweitern** liese[21]. Weitere Ideen, die im **PicoC-Compiler** implementiert werden könnten, wären:

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern**. Erst wenn alle Register **voll** sind, werden Variablen an Adressen im **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller** ist, als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations** (Definition **??**)

[21] Möglicherweise ja im Rahmen eines **Masterprojektes** ☺.

zuzuweisen, wird mithilfe einer **Liveness Analyse** (Defintion **??**) ein **Interferenzgraph** (Definition **??**) mit **Variablen** als Knoten aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition **??**) angewandt, der den **Variablen** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Sobald eine Programmiersprache es erfordert für die Kompilierung **Blöcke** in den Passes einzuführen[22], muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussnalayse** (Definition **??**) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition **??**) seperat auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.[23]

- **Tail Call:** Wenn ein Funktionsaufruf der **letzte ausgeführte Ausdruck** in einem Funktionsblock ist, wird der **Stackframe** dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion **zurückgekehrt** werden muss[24]. Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden, **bevor** der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die **nur Tail Calls** ausführt, mit **Stackframes** nur eine **konstante Menge** an **Speicherplatz** auf dem **Stack** verbraucht. In Code 2.5 sind **zwei Tail Calls** markiert.

- **Partielle Evaluation:** Bei Ausdrücken, wie z.B. `4 + input() - 2`, `input() * 1` oder `0 + input() * 2` können **Teilausdrücke** bereits **während** des **Kompilierens partiell** zu `2 + input()`, `input()` und `input() * 2` berechnet werden. Dies kann durch einen neuen **PicoC-Eval Pass** umgesetzt werden, der **vor** oder **nach** dem **PicoC-Shrink Pass** den jeweiligen Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache L_{Picoc_Eval} umformt. In der **Abstrakten Grammatik** der Sprache L_{PicoC_Eval} sind z.B. **binäre Operationen** zwischen zwei `Num(str)`-PicoC-Knoten **nicht möglich**. Diese **partielle Vorberechnung** kann auch auf **Konstanten** und **Variablen** ausgeweitet werden. Der **Vorteil** ist, dass hierdurch weniger **RETI-Code** generiert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.

- **Lazy Evaluation:** Bei Ausdrücken, wie z.B. `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei z.B. `var1 = 0` und `var2 = 1` müssen diese Ausdrücke nur **soweit** berechnet werden, wie es **benötigt** wird. Sobald bei einer Aneinanderreihung von `&&`-Operationen einmal eine 0 auftaucht, muss der Rest des Ausdrucks **nicht** mehr berechnet werden, da mit dem Auftauchen der 0 bereits klar ist, dass dieser Ausdruck sich zu 0 auswertet. Genauso für eine Aneinanderreihung von `||`-Operationen und dem Auftauchen einer 1. Daher kommt es in beiden gerade gebrachten Beispielen aufgrund der Division durch 0 **nicht** zu einer `DivisionByZero`-Fehlermeldung, da die Ausdrücke garnicht so weit ausgewertet werden. Im Unterschied zur **Partiellen Evaluation** läuft **Lazy Evaluation**[25] zur **Laufzeit** ab.

- **Objektorientierung:** Wie in der Programmiersprache L_{C++} müssen **Klassen** und `new`-, `new[]`-, `delete`-, `delete[]`- und `::`-Operatoren eingeführt werden. Die Speicherung eines **Objekts** ist ähnlich wie bei **Verbunden**.

- **Mehrere Dateien: Funktionen** und **Attribute** werden in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden `.h`-**Headerdateien** verwendet und für deren **Definitionen** sind `.c`-**Quellcodedateien** da. Hierbei ist der **Basisname** einer `.h`-**Headerdatei identisch** zu dem der entsprechenden `.c`-**Quellcodedatei**. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei an genau dieser Stelle in die einbindende Datei entspricht. Über einen **Linker** (Definition **??**) können **kompilierte** `.o`-**Objektdateien** (Definition **??**) zusammengefügt werden. Der **Linker** achtet darauf **keinen doppelten Code** zuzulassen.

[22] Das ist **notwendig**, sobald es sich um eine **Strukturierte Programmiersprache** (Definition 1.2) handelt.
[23] Die in diesem **Unterpunkt** erwähnten **Begriffe** werden nur **grob** erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet.
[24] Was der Grund ist, warum ein **Stackframe** überhaupt angelegt wird, damit später beim **Rücksprung** aus der **aufgerufenen Funktion** die Ausführung mit allen Variablen, wie **vor dem Funktionsaufruf** fortgesetzt werden kann.
[25] Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

- **`malloc` und `free`:** Es wird eine **Bibltiothek**, wie die Bibltiothek `stdlib`[26] mit den Funktionen `malloc` und `free` implementiert, deren `.h`-Headerdatei mittels `#include "malloc_and_free.h"` eingebunden wird. Es braucht eine neue **Kommandozeilenoption** `-l`, um dem **Linker** verwendete Bibliotheken mitzuteilen. Aufgrund der Einführung von `malloc` und `free`, wird im **Datensegment** der Abschnitt nach den **Globalen Statischen Daten** als **Heap** bezeichnet, der mit dem **Stack** kollidieren kann. Im **Heap** wird von der `malloc`-Funktion **Speicherplatz allokiert** und ein **Zeiger** auf den allokierten Speicherplatz **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz an diesen Stellen im **Heap** zur **Allokation** frei ist, muss dies in einer **Datenstruktur** abgespeichert werden.

- **Garbage Collector:** Anstelle der `free`-Funktion kann auch einfach die `malloc`-Funktion direkt so implementiert werden, dass sobald der Speicherplatz auf dem **Heap** knapp wird, Speicherplatz **freigegeben** wird. Es soll Speicherplatz freigegeben werden, der sowieso **unmöglich** in der **Zukunft** mehr **gebraucht** werden würde. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition **??**) implementieren.

- **Bibliothek für `print` und `input`:** Bisher sind die Funktionen `print` und `input` über den **Trick**, einen eigenen **RETI-Befehl** `CALL (PRINT | INPUT) ACC` für den **RETI-Interpreter** zu definieren gelöst. Dieser Befehl übernimmt einfach **direkt** das **Ausgeben** und **Eingaben entgegennehmen**. Ohne Trick geht es über eine eigene `stdio`-**Bibliothek**[27] mit `print`- und `input`-Funktionen, welche die **UART** verwenden, um z.B. an einen simpel gehaltenen, simulierten **Monitor** Daten zu übertragen, die dieser anzeigt.

- **Feld mit Länge:** Man könnte in einer **Bibliothek** über eine **Klasse** einen eigenen **Felddatentyp**, wie in der Programmiersprache L_{C++} mit dem Datentyp `std::vector` implementieren, der seine **Anzahl Elemente** an den **Anfang** des Felds speichert. Auf diese Weise kann über z.B. eine **Methode** `size` die **Anzahl Elemente** direkt über die **Variable des Felds** selbst ausgelesen werden (z.B. `vec_var.size`) und muss **nicht** in einer **seperaten Variable** gespeichert werden.

- **Maschinencode in binärer Repräsentation:** Maschinencode wird **nicht**, wie momentan beim **PicoC-Compiler** in **menschenlesbarer Repräsentation** ausgegeben, sondern in **binärer Repräsentation** nach dem **Intruktionsformat**, welches in der Vorlesung C. Scholl, „Betriebssysteme“ festgelegt wurde.

- **PicoPython:** Da das **Lark Parsing Toolkit** verwendet wird, welches das **Parsen** über eine selbst angegebene **Konkrete Grammatik** umsetzt, könnte mit **relativ geringem Aufwand** ein **Konkrete Grammatik** definiert werden, die eine zur Programmiersprache L_{Python} **ähnliche Konkrete Syntax** beschreibt. Die **Konkrete Syntax** einer Programmiersprache lässt sich durch **Austauschen** der **Konkreten Grammatik** sehr einfach **ändern**. Die **Semantik** zu ändern ist dagegen **deutlich aufwändiger**. Viele der **PicoC-Knoten** könnten für die Programmiersprache $L_{PicocPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.

- **Call by Reference:** Könnte über das **Wiederverwenden** des `&`-Symbols für **Parameter** bei **Funktionsdeklarationen** und **Funktionsdefinitionen** umgesetzt werden, so wie es in der Programmiersprache L_{C++} umgesetzt ist.

- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt, durch welche spezielle **Informationen** in **Objektcode** (Definition **??**) geschrieben werden. Diese Informationen teilen einem **Debugger** unter anderem mit, wo die **RETI-Befehle** für Anweisungen, die zu diesen übersetzt wurden **beginnen** und wo sie **aufhören**. Auf diese Weise weiß der **Debugger**, bis wohin er die **RETI-Befehle** ausführen soll, damit er eine Anweisung abgearbeitet hat.

- **Bootstrapping:** Mittels **Bootstrapping** lässt sich der **PicoC-Compiler** von der Sprache L_{Python}

[26] Auch engl. **General Purpose Standard Library** genannt.
[27] `stdio` ist die **Standard-Input-Output**-Bibliothek von L_C.

unahbängig machen, in welcher dieser implementiert ist, als auch von der **Maschine**, die das **cross-compilen** (Definition **??**) übernimmt. Im Unterkapitel **??** wird **genauer** hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zu einem **Compiler** für die **RETI-CPU** gemacht, der auf der RETI-CPU **selbst** läuft.

```
1  // in:42
2  // expected:1
3
4  int ret1() {
5    return 1;
6  }
7
8  int ret0() {
9    return 0;
10 }
11
12 int tail_call_fun(int bool_val) {
13   if (bool_val) {
14     return ret1();
15   }
16   return ret0();
17 }
18
19 void main() {
20   print(tail_call_fun(input()));
21 }
```

Code 2.5: *Beispiel für Tail Call.*

Anmerkung

Partielle Evaluation und **Lazy Evaluation** wurden im PicoC-Compiler **nicht implementiert**, da dieser als **Lerntool** gedacht ist und diese Funktionalitäten den **RETI-Code** für Studenten **schwerer verständlich** machen würden. Die **Codeschnipsel** und die damit verbundenen **Paradigmen** aus der Vorlesung könnten **nicht** mehr so einfach **nachvollzogen** werden, da es durch das **schwerere Ausmachen** können von **Orientierungspunkten** und das **Fehlen erwarteter Codeschnipsel** leichter zur **Verwirrung** bei den Studenten kommen könnte.

Literatur

Bücher

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

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).
- *History - GCC Wiki*. URL: https://gcc.gnu.org/wiki/History (besucht am 06. 08. 2022).
- *Home - Neovim*. URL: http://neovim.io/ (besucht am 04. 08. 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).

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

- *Deklaration (Programmierung)*. In: *Wikipedia*. Page Version ID: 224245160. 5. Juli 2022. URL: https://de.wikipedia.org/w/index.php?title=Deklaration_(Programmierung)&oldid=224245160 (besucht am 13. 08. 2022).
- *Evaluation strategy*. In: *Wikipedia*. Page Version ID: 1109984435. 12. Sep. 2022. URL: https://en.wikipedia.org/w/index.php?title=Evaluation_strategy&oldid=1109984435 (besucht am 13. 08. 2022).
- *Funktionsprototyp*. In: *Wikipedia*. Page Version ID: 196075270. 22. Jan. 2020. URL: https://de.wikipedia.org/w/index.php?title=Funktionsprototyp&oldid=196075270 (besucht am 13. 08. 2022).
- *Imperative Programmierung*. In: *Wikipedia*. Page Version ID: 219505017. 24. Jan. 2022. URL: https://de.wikipedia.org/w/index.php?title=Imperative_Programmierung&oldid=219505017 (besucht am 13. 08. 2022).
- *Prozedurale Programmierung*. In: *Wikipedia*. Page Version ID: 214023278. 19. Juli 2021. URL: https://de.wikipedia.org/w/index.php?title=Prozedurale_Programmierung&oldid=214023278 (besucht am 13. 08. 2022).
- *Strukturierte Programmierung*. In: *Wikipedia*. Page Version ID: 212983400. 15. Juni 2021. URL: https://de.wikipedia.org/w/index.php?title=Strukturierte_Programmierung&oldid=212983400 (besucht am 13. 08. 2022).
- *Variable (Programmierung)*. In: *Wikipedia*. Page Version ID: 225592766. 24. Aug. 2022. URL: https://de.wikipedia.org/w/index.php?title=Variable_(Programmierung)&oldid=225592766 (besucht am 13. 08. 2022).