

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

ERKLÄRUNG

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

Danksagungen

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

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, bin ich keine Person, die irgendwelche Dinge gerne so macht wie es üblich ist, wenn es keinen überzeugenden Grund dafür gibt. Ich schreibe meine Danksagung nicht auf eine bestimmte Weise, nur weil sich irgendwann mal etabliert hat wie eine Danksagung üblicherweise aussieht. Ich halte nicht viel von künstlichen Floskeln, wie „mein aufrichtigster Dank“ oder „aus tiefstem Herzen“, sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Ich halte es eher so, dass wenn mir wirklich etwas am Dank gegenüber Personen liegt, ich mir wirklich den Aufwand mache einen Text zu schreiben in dem ich diesen zum Ausdruck bringe, im anderen Fall kann man sich bei mir auf die typischen Standardfloskeln einstellen. Bei dieser Bachelorarbeit kann ich nur auf ersteres Zurückgreifen. 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 sehr respektvoll, was ich als keine Selbstverständlichkeit ansehe und sehr wertgeschätzt habe.

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

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

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe¹, er hat sich bei der Korrektur dieser Schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und das trotz dessen, dass meine Bachelorarbeit recht Umfangreich zu lesen ist² und war sich nicht zu Schade die Rolle des Nachrichtenübersetzers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbstverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinen Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Ich hab während meines Bachelorprojekts und meiner Bachelorarbeit wahrscheinlich einen ziemlich eigensinnigen Eindruck gemacht, bei der Weise, wie ich bestimmte Dinge umsetzen wollte. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbstverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

¹Wofür ich mich auch nochmal Entschuldigen will.

²Wobei er sich kein einziges Mal in geringster Weise entnervt darüber gezeigt hat.

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

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen, für dessen Implementierung Michel Giehl sich netterweise zur Verfügung gestellt hat. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler auseinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Mist baue. Der RETI-Emulator von Michel Giehl ist unter [Link³](https://github.com/michel-giehl/Reti-Emulator) zu finden.

Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat, was man auch selten im Studium erlebt, dass dem 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 ziemlich viel unerwartete Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl nichts geworden. Man hat daran gemerkt, dass Prof. Dr. Scholl das Wohlergehen der Studenten wichtig ist.

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

³<https://github.com/michel-giehl/Reti-Emulator>.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
1 Motivation	1
1.1 RETI-Architektur	2
1.2 Die Sprache PicoC	4
1.3 Eigenheiten der Sprachen C und PicoC	5
1.4 Gesetzte Schwerpunkte	11
1.5 Über diese Arbeit	12
1.5.1 Stil der Schriftlichen Ausarbeitung	13
1.5.2 Aufbau der Schriftlichen Arbeit	14
1.5.3 Umsetzung von Funktionen	16
1.5.3.1 Mehrere Funktionen	16
1.5.3.1.1 Sprung zur Main Funktion	19
1.5.3.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen	22
1.5.3.3 Funktionsaufruf	24
1.5.3.3.1 Rückgabewert	29
1.5.3.3.2 Umsetzung der Übergabe eines Feldes	33
1.5.3.3.3 Umsetzung einer Übergabe eines Verbundes	37
Literatur	A

Abbildungsverzeichnis

1.1	Schritte zum Ausführen eines Programmes mit dem GCC.	1
1.2	Stark vereinfachte Schritte zum Ausführen eines Programmes.	1
1.3	Speicherorganisation.	3
1.4	README.md im Github Repository der Bachelorarbeit.	12

Codeverzeichnis

1.1	Beispiel für Spiralregel.	6
1.2	Ausgabe von Beispiel für Spiralregel.	6
1.3	Beispiel für unterschiedliche Ausführung.	7
1.4	Ausgabe des Beispiels für unterschiedliche Ausführung.	7
1.5	Beispiel mit Dereferenzierungsoperator.	7
1.6	Ausgabe des Beispiels mit Dereferenzierungsoperator.	7
1.7	Beispiel dafür, dass Struct kopiert wird.	8
1.8	Ausgabe von Beispiel, dass Struct kopiert wird.	8
1.9	Beispiel dafür, dass Zeiger auf Feld übergeben wird.	9
1.10	Ausgabe von Beispiel dafür, dass Zeiger auf Feld übergeben wird.	9
1.11	Beispiel für Deklaration und Definition.	10
1.12	Ausgabe von Beispiel für Deklaration und Definition.	10
1.13	Beispiel für Sichtbarkeitsbereichs.	11
1.14	Ausgabe von Beispiel für Sichtbarkeitsbereichs.	11
1.15	PicoC-Code für 3 Funktionen.	16
1.16	Abstrakter Syntaxbaum für 3 Funktionen.	17
1.17	PicoC-Blocks Pass für 3 Funktionen.	18
1.18	PicoC-ANF Pass für 3 Funktionen.	18
1.19	RETI-Blocks Pass für 3 Funktionen.	19
1.20	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	19
1.21	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	20
1.22	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	21
1.23	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	22
1.24	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.	22
1.25	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.	24
1.26	PicoC-Code für Funktionsaufruf ohne Rückgabewert.	24
1.27	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.	25
1.28	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.	26
1.29	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.	27
1.30	RETI-Pass für Funktionsaufruf ohne Rückgabewert.	29
1.31	PicoC-Code für Funktionsaufruf mit Rückgabewert.	29
1.32	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.	30
1.33	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.	31
1.34	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.	33
1.35	PicoC-Code für die Übergabe eines Feldes.	33
1.36	Symboltabelle für die Übergabe eines Feldes.	35
1.37	PicoC-ANF Pass	35
1.38	RETI-Block Pass für die Übergabe eines Feldes.	37
1.39	PicoC-Code für die Übergabe eines Verbundes.	37
1.40	PicoC-ANF Pass für die Übergabe eines Verbundes.	38
1.41	RETI-Block Pass für die Übergabe eines Verbundes.	40

Tabellenverzeichnis

1.1 Präzedenzregeln von PicoC.	3
--	---

Definitionsverzeichnis

1.1	Imperative Programmierung	5
1.2	Strukturierte Programmierung	5
1.3	Prozedurale Programmierung	5
1.4	Call by Value	8
1.5	Call by Reference	8
1.6	Funktionsprototyp	9
1.7	Deklaration	10
1.8	Definition	10
1.9	Sichtbarkeitsbereich (bzw. engl. Scope)	11

Grammatikverzeichnis

1 Motivation

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

Compiler, wie der **GCC**¹ oder **Clang**² werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert³ zu **Maschinencode** zu kompilieren.

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll, z.B. im Fall des **GCC** über `> gcc program.c -o machine_code`⁴. Als Ergebnis erhält man im Fall des **GCC** die mit der Option `-o` selbst benannte Datei `machine_code`, welche dann zumindest unter **Unix** über `> ./machine_code` **ausgeführt** werden kann, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

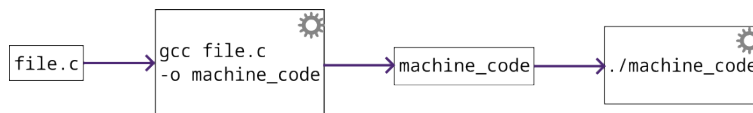


Abbildung 1.1: Schritte zum Ausführen eines Programmes mit dem GCC.

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

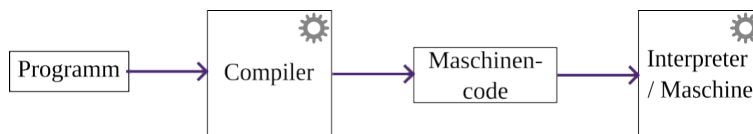


Abbildung 1.2: Stark vereinfachte Schritte zum Ausführen eines Programmes.

¹GCC, the GNU Compiler Collection - GNU Project.

²clang: C++ Compiler.

³Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C .

⁴Bei **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 **Objectcode** (Definition ??) kompiliert wird. Das Kompilieren zu **Objectcode** 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.

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** stark vereinfacht funktionieren könnte.

Die konkrete **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC} , welche eine Untermenge der Sprache L_C ist⁵ in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zu Grunde 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.

1.1 RETI-Architektur

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

Um den den **PicoC-Compiler** zu **testen** war es notwendig einen **RETI-Interpreter** zu implementieren, der genau die Variante der **RETI-Architektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ **simuliert**.

Anmerkung

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

Die **Register** der **RETI-Architektur** werden in Tabelle 1.1 aufgezählt und erläutert. Die **Maschinenbefehle** und **Datenpfade** der **RETI-Architektur** sind im Kapitel ?? dokumentiert, da diese **nicht explizit** zum **Verständnis** der späteren Kapitel notwendig sind, aber zum **vollständigen Verständnis** notwendig sind, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** L_{RETI} ist durch Grammatik ?? und Grammatik ?? zusammengefasst beschrieben. Für genauere **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

⁵Die der **GCC** kompilieren kann.

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

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

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

Tabelle 1.1: Präzedenzregeln von PicoC.

Die **RETI-Architektur** ermöglicht bei der Ausführung von RETI-Programmen **Prozesse** zu nutzen. In Abbildung 1.3 ist der **Aufbau** eines **Prozesses** im **Hauptspeicher** der **RETI-Architektur** zu sehen. Das 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 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 den **Anfang** der **Globalen Statischen Daten** 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** und zur Bestimmung der **relativen Startadresse**, an welcher der **RETI-Code** einer bestimmten **Funktion** anfängt.

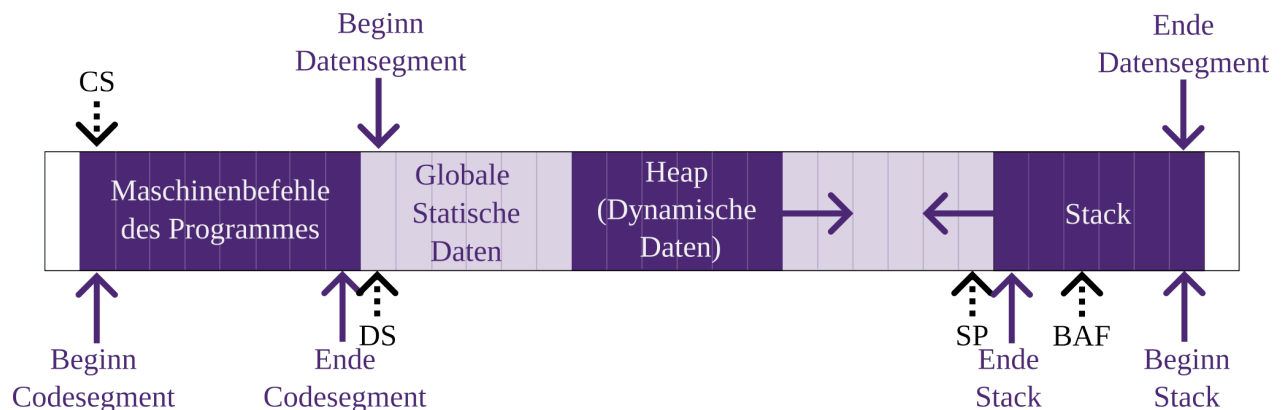


Abbildung 1.3: Speicherorganisation.

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

Die **Ausführung** eines Programmes **startet** auf die einfachste Weise, indem es von einem **Startprogramm** im **EPROM**⁸ aufgerufen wird. Der **EPROM** wird beim Start einer **RETI-CPU** als **erstes** aufgerufen, da nach 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, also als **erstes** das Programm ausgeführt wird, welches an **Adresse** 0 im EPROM anfängt.

Die **UART**⁹ ist eine **elektronische Schaltung** mit je nach Umsetzung mehr oder weniger **Pins**, wobei es allerdings immer einen **RX**- und einen **TX**-Pin gibt, für jeweils Empfangen¹⁰ und Versenden¹¹ von Daten gibt. Jeder der **Pins** wird dabei mit einer anderen **Adresse** von 2^3 verschiedenen Adressen angesprochen und jeweils 8-Bit können nach den Datenpfaden ?? auf einmal über einen Pin in ein **Register** der **UART** geschrieben werden, um **versandt** zu werden oder von einem Pin **empfangen** werden. Die **UART** 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**.

1.2 Die Sprache PicoC

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

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/*` and `*/`.
- die **Primitiven Datentypen** `int`, `char` und `void`.
- die **Abgeleiteten Datentypen Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`).
- `if(cond){ }-` und `else{ }-`**Anweisungen**¹³.
- `while(cond){ }-` und `do while(cond){ };`**Anweisungen**.
- **Arihmatische Ausdrücke**, welche mithilfe der binären Operatoren `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` und unären Operatoren `-`, `~` umgesetzt sind.
- **Logische Ausdrücke**, welche mithilfe der Relationen `==`, `!=`, `<`, `>`, `<=`, `>=` und Logischer Verknüpfungen `!`, `&&`, `||` umgesetzt sind.
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind.
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`) und **Funktionsaufrufe** (z.B. `fun(ar, st.var)`).

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

⁷Von 0 bis $2^{32} - 1$.

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

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

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

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

¹²Kurz für **S**tatic **r**andom-**a**ccess **m**emory.

¹³Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

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

Der **Aufbau** der **Programmiersprache** L_C ist durch Grammatik ?? und Grammatik ?? zusammengekommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache L_C , die genauso ein Teil der Programmiersprache L_{PicoC} sind, da L_{PicoC} eine Untermenge von L_C ist und welche in der Implementierung des **PicoC-Compilers** in Kapitel ?? noch eine **wichtige Rolle** spielen werden im Folgenden genauer erläutert. 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** bzw. **ausgeführt** wurden, aber selbiges gilt genauso für L_C aus bereits erläuterten Grund.

Bei der Programmiersprache L_{PicoC} handelt es sich um eine **imperative** (Definition 1.1), **strukturierte** (Definition 1.2) und **prozedurale Programmiersprache** (Definition 1.3). Aufgrund dessen, dass es sich bei beiden um **Imperative Programmiersprachen** handelt ist es wichtig bei der Implementierung die **Reihenfolge** zu beachten und aufgrund dessen, dass es sich bei beiden um **Strukturierte** und **Prozedurale Programmiersprachen** handelt, ist es eine gute Methode bei der Implementierung auf **Blöcke**¹⁴ zu setzen zwischen denen **hin und her** gesprungen werden kann und welche in den einzelnen Implementierungsschritten die **notwendige Datenstruktur** darstellen um **Auswahl** zwischen Codestücken, **Wiederholung** von Codestücken und **Sprünge** zu Blöcken mit entsprechend zu bestimmten **Bezeichnern** (Definition ??) passenden **Labeln** (Definition ??) umzusetzen.

Definition 1.1: Imperative Programmierung



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

^aThiemann, „Einführung in die Programmierung“.

Definition 1.2: Strukturierte Programmierung



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

^aThiemann, „Einführung in die Programmierung“.

Definition 1.3: Prozedurale Programmierung



Programme werden z.B. mittels **Funktionen** in **überschaubare Unterprogramme** bzw. **Prozeduren** aufgeteilt, die **aufgerufen** 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**, was den Code deutlich **überschaubarer** macht. da man die Aufgabe eines Codestücks nun nur noch mit seinem **Bezeichner assoziieren** muss.^a

¹⁴Werden später im Kapitel ?? genauer erklärt.

^aThiemann, „Einführung in die Programmierung“.

In L_C ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen $[<i>]</i>$ -Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: $\langle \text{remaining-datatype} \rangle \langle \text{var} \rangle [<i>]</i>$, während andere Programmiersprachen die eckigen $[<i>]</i>$ -Klammern **vor** die Variable schreiben $\langle \text{remaining-datatype} \rangle [<i>]</i> \langle \text{var} \rangle$.

Werden die eckigen $[<i>]</i>$ -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* nachgelesen werden kann. Werden die eckigen $[<i>]</i>$ -Klammern **hinter** die Variable geschrieben, wirken diese zum Verwechseln ähnlich zum $\langle \text{var} \rangle [<i>]</i>$ -Operator für den **Zugriff auf den Index eines Feldes**. Wenn ein Ausdruck geschrieben wird, wie $\text{int ar}[1] = \{42\}$ wird, ist dieser vom Ausdruck $\text{var}[0] = 42$ nur durch den **Kontext** um $\text{var}[1]$ bzw. $\text{var}[0]$ rum zu unterscheiden.

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 die eckigen $[<i>]</i>$ -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 ($\text{*complex_var}[0][1])[1].\text{attr}$, 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 Spiralregel.

```
1 42
```

Code 1.2: Ausgabe von Beispiel für Spiralregel.

In L_C ist die **Ausführbarkeit einer Operation** oder **wie** diese Operation ausgeführt wird davon abhängig, was für einen **Datentyp** die Variable in diesem **Kontext** der Operation hat. In dem Beispiel in Code 1.3 wird in Zeile 2 ein „**Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2**“ und Zeile 3 ein „**Zeiger auf Felder der Mächtigkeit 2**“ erstellt. In den **markierten Zeilen** wird zweimal in Folge die **gleiche Operation** $\langle \text{var} \rangle [0][1]$ ausgeführt, allerdings hat die Operation aufgrund der **unterschiedlichen Datentypen** der Variablen einen **unterschiedlichen Effekt**.

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


```

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

```

Code 1.3: Beispiel für unterschiedliche Ausführung.

```

1 42 42

```

Code 1.4: Ausgabe des Beispiels für unterschiedliche Ausführung.

Eine weitere interessante Eigenheit, die tatsächlich nur in der **Untermenge** von L_C , die L_{PicoC} darstellt gültig ist¹⁵, 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}}`;¹⁶ 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** `copyable_ar` der **aufzufendenden** Funktion. Der Eintrag an Index 1 im Feld bleibt bei 314.

¹⁵In der Sprache L_C gibt es einen **Unterschied** bei der **Initialisierung** bei z.B. `int *var = "string"` und `int var[1] = "string"`, der allerdings nichts mit den Operatoren `*var` und `var[1]` zu tun hat, sondern mit der **Initialisierung**, bei der die Sprache L_C verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Feldindex**, hinter den Bezeichner schreibt (z.B. `var[1]`), obwohl es ein **Abgeleiteter Datentyp** ist.

¹⁶**Später** wird darauf eingegangen, warum der Verbund den **Bezeichner** `copyable_ar` erhalten hat.

Definition 1.4: Call by Value

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

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

^aBast, „Programmieren in C“.

```

1 struct st {int ar[2];};
2
3 int fun(struct st copyable_ar) {
4     copyable_ar.ar[1] = 42;
5 }
6
7 void main() {
8     struct st copyable_ar = {.ar={314, 314}};
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 von Beispiel, 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 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 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 **erste Eintrag** des Felds.

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

Definition 1.5: Call by Reference

Es wird eine **implizite Referenz** einer Variable, welche ein **Argument** eines **Funktionsaufrufes** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.

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

^aBast, „Programmieren in C“.

```

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

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

```

1 314 42
```

Code 1.10: Ausgabe von Beispiel 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 vorher durch die **Funktionsdefinition** bekannt ist, damit 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 die Funktionen **nicht** immer so anordnen, dass jede in einem Funktionsaufruf referenzierte 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.*^{a,b}

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

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

Definition 1.7: Deklaration

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

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

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

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

Definition 1.8: Definition

Dem **Compiler** wird mitgeteilt, dass zu einem **bestimmten Zeitpunkt** in der Programmausführung **Speicher** für eine **Variable** **angelegt** werden soll und **wo**^a dieser **angelegt** werden soll. Eine **Funktion** ist **definiert** ihr eine **relative Anfangsadresse** im **Hauptspeicher** zugewiesen werden kann, aber welcher die **Maschinenbefehle** für diese **Funktion** abgespeichert werden können.^{b c}

^aIm Fall des PicoC-Compilers in den **Globalen Statischen Daten** oder auf dem **Stack**.

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

^cP. D. 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 von Beispiel für Deklaration und Definition.

In L_{PicoC} lässt sich eine **definierte Variable** nur innerhalb ihres **Sichtbarkeitsbereichs** (Definition 1.9) verwenden. **Lokale Variablen** und **Parameter** lassen sich nur innerhalb der **Funktion** in welcher sie **deklariert** bzw. **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, doch bezeichnet er 2 **unterschiedliche Variablen**. Der **Parameter** `param` und die **Lokale Variable** `local_var` dürfen **nicht**

den **gleichen Bezeichner** haben, da sie sich im gleichen **Sichtbarkeitsbereich** der Funktion `fun_scope` befinden. Die **Ausgabe** des Beispiels in Code 1.13 ist in Code 1.14 zu sehen.

Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)



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

^aThiemann, „Einführung in die Programmierung“.

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

Code 1.13: Beispiel für Sichtbarkeitsbereichs.

```
1 4 2
```

Code 1.14: Ausgabe von Beispiel für Sichtbarkeitsbereichs.

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinsprache L_{RETI} die **Syntax** und **Semantik** der Sprache L_C identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache L_{PicoC} im Vergleich zu z.B. dem **GCC**¹⁷ ohne merklichen Unterschied¹⁸ kompilieren können.

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

Des Weiteren ist die **Laufzeit** bei Compilern zwar vor allem in der Industrie **nicht unwichtig**, aber bei **Compilern**, verglichen mit **Interpretern** weniger zu gewichten, da ein Compiler bei einem fertig implementierten Programm nur **einmal** Maschinencode generieren muss und dieser Maschinencode danach fortan ausgeführt wird. Beim einem **Compiler** ist daher eher zu priorisieren, dass der kompilierte **Maschinencode** möglichst **effizient** ist.

¹⁷Da die Sprache L_{PicoC} eine **Untermenge** von L_C ist, kann der **GCC** L_{PicoC} ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinsprache L_{RETI} .

¹⁸Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinsprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

Beim **PicoC-Compiler** wurde daher eher darauf Wert gelegt **sauberen** und **strukturierten Code** zu schreiben, den interessierte Studenten bei Interesse selber nachvollziehen können und eine **unkomplizierte Bibliothek** mit **guter Dokumentation**¹⁹, nämlich das **Lark Parsing Toolkit**²⁰ für das **Parsen** (Definition ??) zu verwenden. Und wie man auch beim **Ausführen der Tests** (wie in Unterkapitel ?? beschrieben) sieht, macht die **Laufzeit** des Compilers für **übliche** und auch **längere Programme**, wie ein Student sie zu Lernzwecken mit dem Compiler kompilieren würde absolut **keine Probleme**.

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter [Link](#)²¹ zu finden. In der Datei **README.md** (siehe Abbildung 1.4) ist unter „**Getting Started**“ ein kleines **Einführungstutorial** verlinkt. Unter „**Usage**“ ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Daneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter [Link](#)²² zu finden.

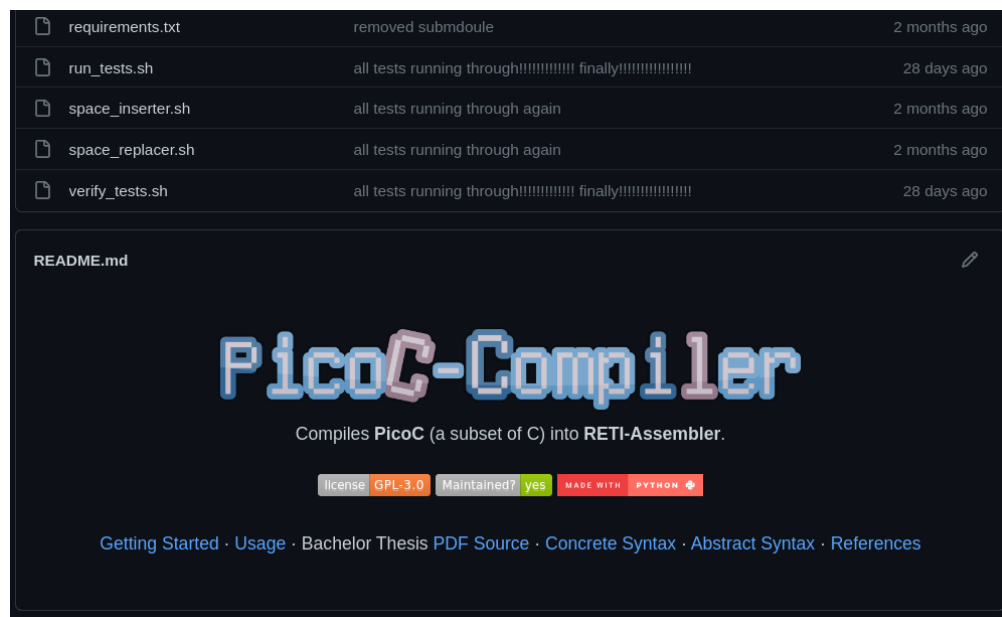


Abbildung 1.4: *README.md* im Github Repository der Bachelorarbeit.

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** unter [Link](#)²³ zu finden. Die **PDF** der Schriftliche Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode**, welcher unter [Link](#)²⁴ veröffentlicht ist automatisch mithilfe der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibliotheken** sind unter [Link](#)²⁵ zu finden²⁶. Die Grafiken, die nicht mittels der

¹⁹ Welcome to Lark's documentation! — Lark documentation.

²⁰ Lark - a parsing toolkit for Python.

²¹ <https://github.com/matthejue/PicoC-Compiler>.

²² <https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971>.

²³ https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.

²⁴ <https://github.com/matthejue/Bachelorarbeit>.

²⁵ https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.

²⁶ Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.

Tikz Bibliothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**²⁷ erstellt. Falls Interesse besteht **Grafiken** aus der Schriftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den .svg-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibliotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter „References“ bzw. direkt unter [Link](#)²⁸ zu finden.

Um die verschiedenen **Aspekte** dieser Schriftlichen Ausarbeitung der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel **Motivation** werden Codebeispiele zur **Anschauung** verwendet und mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters Ausgaben** erzeugt, die in dieses Dokument **eingelezen** wurden. In Kapitel ?? werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** gezeigt²⁹.

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 eingelezen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem [Link](#)³⁰ finden und mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind³¹.

1.5.1 Still der Schriftlichen Ausarbeitung

In dieser **Schriftliche Ausarbeitung der Bachelorarbeit** sind die 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 auch damit der **Inhalt** eines vorher gelesener **Paragraphs** nochmal durch Überfliegen der Hervorgehobenen Wörter in **Erinnerung gerufen** werden kann.

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas genau so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht³² die Frage des „**warum**“.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**³³ verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **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**.

²⁷Developers, *Draw Freely — Inkscape*.

²⁸https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

²⁹Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

³⁰https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.

³¹Es 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.

³²Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.

³³Das ist ein **Beispiel** für eine **Fußnote**.

Es wurde immer versucht möglichst **deutsche Fachbegriffe** zu verwenden, sofern sie einigermaßen **geläufig** sind und bei der Verwendung **nicht** eher **verwirren**. 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.

1.5.2 Aufbau der Schriftlichen Arbeit

Der Hauptteil der **Schriftlichen Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: **Motivation**, **??**, **??** und **??**. **Zusätzlich** gibt es noch den **??**.

Im momentanen Kapitel **Motivation** wurde ein kurzer **Einstieg** in das Thema **Compilerbau** gegeben, die **zentrale Aufgabenstellung** der Bachelorarbeit erläutert und **Schwerpunkte** gesetzt, sowie auf **Eigenheiten der Sprache L_C** eingegangen, die für die **Implementierung** relevant sein werden.

Im Kapitel **??** werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische Vorgehen**, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen 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 der 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. Generell wurde im Kapitel **Einführung** versucht an Erklärungen nicht zu sparren, damit aufgrund dessen, dass das Thema eher fachfremd für Prof. Dr. Scholl ist für das Kapitel **Implementierung** keine wichtigen Aspekte unverständlich bleiben.

Im Kapitel **??** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers**, unterteilt in die verschiedenen **Phasen der Kompilierung** nach denen das Kapitel **Einführung** ebenfalls unterteilt ist erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähnliche Kapiteileinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständnis eines Aspekts in Kapitel **Implementierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteileinteilung** die entsprechende Definition analog im Kapitel **Einleitung** gefunden werden.

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

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 **Kapitel**, die bei **Interesse** zur **weiteren Vertiefung** da sind und zum Verständnis der anderen Kapitel **nicht notwendig** sind. Damit der **Rote Faden** in dieser Schriftlichen Ausarbeitung der Bachelorarbeit erkennbar bleibt und der **Lesefluss** nicht gestört wird, wurden alle diese Informationen in den **Appendix** ausgelagert.

Die **Sonstigen 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. Diese **Themen** und **Definitionen** passen nicht ins Kapitel ??, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tun haben und auch nichts ins **Kapitel ??**, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in der Schriftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. 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, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

1.5.3 Umsetzung von Funktionen

Um die **Umsetzung** von **Funktionen** zu verstehen, ist es erstmal wichtig zu verstehen, wie **Funktionen** später im **RETI-Code** aussehen (Unterkapitel 1.5.3.1), wie Funktionen **dekliert** und **definiert** werden können und hierbei **Sichtbarkeitsbereiche** umgesetzt sind (Unterkapitel 1.5.3.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines **Funktionsaufrufes** erklärt werden (Unterkapitel 1.5.3.3). Beim Thema **Funktionsaufruf** muss im speziellen nochmal darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 1.5.3.3.1) umgesetzt sind und die **Übergabe** von **Abgeleiteten Datentypen**, die mehr als eine Speicherzelle belegen, wie **Verbunden** (Unterkapitel 1.5.3.3.3) und **Feldern** (Unterkapitel 1.5.3.3.2) umgesetzt ist.

1.5.3.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 1.15 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

```

1 void main() {
2     return;
3 }
4
5 void fun1() {
6 }
7
8 int fun2() {
9     return 1;
10 }

```

Code 1.15: *PicoC-Code für 3 Funktionen.*

Im **Abstrakten Syntaxbaum** in Code 1.16 werden die 3 **Funktionen** durch entsprechende Knoten dargestellt. Am Beispiel der **Funktion** `void fun2() {return 1;}` wäre der hierzu passende **Knoten** `FunDef(VoidType(), Name('fun2'), [], [Return(Num('1'))])`. Die einzelnen **Attribute** dieses `FunDef(datatype, name, allocs, stmts.blocks)`-Knoten sind in Tabelle ?? erklärt.

```

1 File
2   Name './verbose_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return
10          Empty
11      ],
12     FunDef
13       VoidType 'void',
14       Name 'fun1',
15       [],
16       [],

```

```

17   FunDef
18     IntType 'int',
19     Name 'fun2',
20     [],
21     [
22       Return
23         Num '1'
24     ]
25 ]

```

Code 1.16: *Abstrakter Syntaxbaum für 3 Funktionen.*

Im **PicoC-Blocks Pass** in Code 1.17 werden die **Anweisungen** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Anweisungen der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`³⁴ beinhaltet den **Bezeichner** bzw. den `Name(str)`-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`³⁵.

Es werden parallel dazu neue Zuordnungen im **Assoziativen Feld** `fun_name_to_block_name` hinzugefügt. Das **Assoziative Feld** `fun_name_to_block_name` ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der die erste **Anweisung** der Funktion enthält zu. Der **Bezeichner** des Blockes `<name>.<nummer>` ist dabei bis auf die angehängte **Nummer** identisch zu dem der Funktion. Diese Zuordnung ist nötig, da **Blöcke** eben noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```

1 File
2   Name './verbose_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.2',
11          [
12            Return(Empty())
13          ]
14      ],
15     FunDef
16       VoidType 'void',
17       Name 'fun1',
18       [],
19       [
20         Block
21          Name 'fun1.1',
22          []
23      ],
24     FunDef
25       IntType 'int',
26       Name 'fun2',

```

³⁴Eine **Erklärung** dazu ist in Unterkapitel ?? zu finden.

³⁵Der **Grund** dafür kann im Unterkapitel ?? nachgelesen werden.

```

27     [],
28     [
29         Block
30         Name 'fun2.0',
31         [
32             Return(Num('1'))
33         ]
34     ]
35 ]

```

Code 1.17: *PicoC-Blocks Pass für 3 Funktionen.*

Im **PicoC-ANF Pass** in Code 1.18 werden die `FunDef(datatype, name, allocs, stmts)`-Knoten komplett aufgelöst, sodass sich im `File(name, decls.defs.blocks)`-Knoten nur noch Blöcke befinden.

```

1 File
2   Name './verbose_3_funs.picoc_mon',
3   [
4       Block
5       Name 'main.2',
6       [
7           Return(Empty())
8       ],
9       Block
10      Name 'fun1.1',
11      [
12          Return(Empty())
13      ],
14      Block
15      Name 'fun2.0',
16      [
17          // Return(Num('1'))
18          Exp(Num('1'))
19          Return(Stack(Num('1')))
20      ]
21 ]

```

Code 1.18: *PicoC-ANF Pass für 3 Funktionen.*

Nach dem **RETI Pass** in Code 1.19 gibt es nur noch **RETI-Befehle**, die Blöcke wurden entfernt und die **RETI-Befehle** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man die Funktionen nicht mehr direkt ausmachen, denn die **Kommentare** enthalten die **Labelbezeichner** `<name>.<nummer>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer, dem **Namen** der jeweiligen **Funktion** entsprechen.

Da es in der `main`-Funktion keinen **Funktionsaufruf** gab, wird der Code, der nach dem **Befehl** in der **markierten Zeile** kommt nicht mehr betreten. Funktionen sind im **RETI-Code** nur dadurch existent, dass im RETI-Code **Sprünge** (z.B. `JUMP<rel> <im>`) zu den jeweils richtigen Positionen gemacht werden. Die Sprünge werden nämlich dorthin gemacht, wo die **RETI-Befehle**, die aus den **Anweisungen** einer **Funktion** kompiliert wurden anfangen.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.2'))))
3 # // not included Exp(GoTo(Name('main.2'))))
4 # // Block(Name('main.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.1'), [])
8 # Return(Empty())
9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1'))))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 LOADIN BAF PC -1;

```

Code 1.19: RETI-Blocks Pass für 3 Funktionen.

1.5.3.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 1.15 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 1.20 ist die `main`-Funktion allerdings **nicht** die **erste** Funktion. Daher muss dafür gesorgt werden, dass die `main`-Funktion die erste Funktion ist, die ausgeführt wird.

```

1 void fun1() {
2 }
3
4 int fun2() {
5     return 1;
6 }
7
8 void main() {
9     return;
10 }

```

Code 1.20: PicoC-Code für Funktionen, wobei die `main` Funktion nicht die erste Funktion ist.

Im **RETI-Blocks Pass** in Code 1.21 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```

1 File
2 Name './verbose_3_funs_main.reti_blocks',
3 [

```

```

4   Block
5     Name 'fun1.2',
6     [
7       # Return(Empty())
8       LOADIN BAF PC -1;
9     ],
10  Block
11    Name 'fun2.1',
12    [
13      # // Return(Num('1'))
14      # Exp(Num('1'))
15      SUBI SP 1;
16      LOADI ACC 1;
17      STOREIN SP ACC 1;
18      # Return(Stack(Num('1')))
19      LOADIN SP ACC 1;
20      ADDI SP 1;
21      LOADIN BAF PC -1;
22    ],
23  Block
24    Name 'main.0',
25    [
26      # Return(Empty())
27      LOADIN BAF PC -1;
28    ]
29 ]

```

Code 1.21: *RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Eine simple Möglichkeit ist es, die **main-Funktion** einfach nach **vorne** zu schieben, damit diese als **erstes** ausgeführt wird. Im `File(name, decls_defs)`-Knoten muss dazu im `decls_defs`-Attribut, welches eine **Liste von Funktionen** ist, die **main-Funktion** an Index 0 geschoben werden.

Die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die **main-Funktion** nicht die erste auftauchende Funktion ist, einen `start.<nummer>`-Block als ersten Block einzufügen. Dieser `start.<nummer>`-Block enthält einen `GoTo(Name('main.<nummer>'))`-Knoten, der im **RETI Pass 1.23** in einen Sprung zur **main-Funktion** übersetzt wird.

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für Verwender³⁶ des **PiocC-Compilers** vermutlich am **intuitivsten** ist, wenn der **RETI-Code** für die Funktionen an denselben Stellen relativ zueinander verortet ist, wie die Funktionsdefinitionen im **PicoC-Code**.

Das **Einsetzen** des `start.<nummer>`-Blockes erfolgt im **RETI-Patch Pass** in Code 1.22, da der **RETI-Patch-Pass** der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die **main-Funktion** nicht die erste Funktion ist.

```

1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',

```

³⁶Also die kommenden **Studentengenerationen**.

```

6      [
7          # // Exp(GoTo(Name('main.0')))
8          Exp(GoTo(Name('main.0')))
9      ],
10     Block
11     Name 'fun1.2',
12     [
13         # Return(Empty())
14         LOADIN BAF PC -1;
15     ],
16     Block
17     Name 'fun2.1',
18     [
19         # // Return(Num('1'))
20         # Exp(Num('1'))
21         SUBI SP 1;
22         LOADI ACC 1;
23         STOREIN SP ACC 1;
24         # Return(Stack(Num('1')))
25         LOADIN SP ACC 1;
26         ADDI SP 1;
27         LOADIN BAF PC -1;
28     ],
29     Block
30     Name 'main.0',
31     [
32         # Return(Empty())
33         LOADIN BAF PC -1;
34     ]
35 ]

```

Code 1.22: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

Im **RETI Pass** in Code 1.23 wird das `GoTo(Name('main.<number>'))` durch den entsprechenden **Sprung** `JUMP <distance.to.main.function>` ersetzt und es werden die **Blöcke entfernt**.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.0')))
3 JUMP 8;
4 # // Block(Name('fun1.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun2.1'), [])
8 # // Return(Num('1'))
9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Return(Stack(Num('1')))
14 LOADIN SP ACC 1;
15 ADDI SP 1;
16 LOADIN BAF PC -1;
17 # // Block(Name('main.0'), [])
18 # Return(Empty())

```

```
19 LOADIN BAF PC -1;
```

Code 1.23: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

1.5.3.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen

In der Programmiersprache L_C und somit auch L_{PicoC} ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehlermeldungen** ausgegeben werden können, wenn der **Prototyp** (Definition 1.6) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

Die Deklaration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 1.24 **markierten Zeile 1** oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){};`), wie in den **markierten Zeilen 3-4**.

In dem Beispiel in Code 1.24 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdeklaration**, wie in der **markierten Zeile 1** notwendig. Beim **Funktionsaufruf** zur Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen 3-4** zu sehen ist.

```
1 int fun2(int var);
2
3 void fun1() {
4 }
5
6 void main() {
7     int var = fun2(42);
8     fun1();
9     return;
10 }
11
12 int fun2(int var) {
13     return var;
14 }
```

Code 1.24: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 1.25 für das Beispiel in Code 1.24 dargestellt ist. Die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` werden wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette Komposition der **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(), IntType('int'), Name('var'))])` zugewiesen.

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Sichtbarkeitsbereiche** (Definition 1.9). Die **Sichtbarkeitsbereiche** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**³⁷ Durchiterieren über den **Abstrakten Syntaxbaum** des aktuellen **Passes** ein **Funktionswechsel** eintritt und über die Anweisungen der nächsten Funktion iteriert wird, für

³⁷D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

Ein Grund, warum **Sichtbarkeitsbereiche** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel**, die aus dem **Bezeichner** einer Variable und einem angehängten **Suffix** bestehen, in der als **Assoziatives Feld** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt ihren **Sichtbarkeitsbereich** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im `Name(str)`-Knoten des `name`-Attribut eines **Symbols** der Symboltabelle angehängt. Zur besseren Vorstellung ist dies in Code 1.25 markiert.

Die Variable `var@main`, bei der es sich um eine **Lokale Variable** der `main`-Funktion handelt, ist nur innerhalb des **Codeblocks** `{}` der `main`-Funktion **sichtbar** und die Variable `var@fun2` bei der es sich um einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** `{}` der Funktion `fun2` **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symbols** in der **Symboltabelle** an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird. Und da die Zuordnungen im **Assoziativen Feld** **eindeutig** sind, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

Das Zeichen '@' wurde aus einem bestimmten Grund als **Trennzeichen** verwendet, nämlich, weil kein Bezeichner das Zeichen '@' jemals selbst enthalten kann. Damit ist ausgeschlossen, dass falls ein **Benutzer** des **PicoC-Compilers** zufällig auf die Idee kommt seine Funktion genauso zu nennen (z.B. `var@fun2` als Funktionsname), es zu Problemen kommt, weil bei einem Nachschlagen der **Variable** die **Funktion** nachgeschlagen wird.

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
7         ↪ IntType('int'), Name('var'))])
8       name:                Name('fun2')
9       value or address:    Empty()
10      position:             Pos(Num('1'), Num('4'))
11      size:                 Empty()
12    },
13    Symbol
14    {
15      type qualifier:      Empty()
16      datatype:            FunDecl(VoidType('void'), Name('fun1'), [])
17      name:                Name('fun1')
18      value or address:    Empty()
19      position:             Pos(Num('3'), Num('5'))
20      size:                 Empty()
21    },
22    Symbol
23    {
24      type qualifier:      Empty()
25      datatype:            FunDecl(VoidType('void'), Name('main'), [])
26      name:                Name('main')
27      value or address:    Empty()
28      position:             Pos(Num('6'), Num('5'))
29      size:                 Empty()
30    },
31  ]

```

```

31  {
32      type qualifier:      Writeable()
33      datatype:            IntType('int')
34      name:                 Name('var@main')
35      value or address:     Num('0')
36      position:             Pos(Num('7'), Num('6'))
37      size:                 Num('1')
38  },
39  Symbol
40  {
41      type qualifier:      Writeable()
42      datatype:            IntType('int')
43      name:                 Name('var@fun2')
44      value or address:     Num('0')
45      position:             Pos(Num('12'), Num('13'))
46      size:                 Num('1')
47  }
48 ]

```

Code 1.25: *Symbottabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

1.5.3.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 1.26 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und dieses Kapitel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 1.5.3.3.1 erklärt.

```

1 struct st {int attr1; int attr2[2];};
2
3 void stack_fun(struct st param[2][3]);
4
5 void main() {
6     struct st local_var[2][3];
7     stack_fun(local_var);
8     return;
9 }
10
11 void stack_fun(struct st param[2][3]) {
12     int local_var;
13 }

```

Code 1.26: *PicoC-Code für Funktionsaufruf ohne Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 1.27 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```

1 File
2   Name './example_fun_call_no_return_value.ast',
3   [
4     StructDecl

```

```

5      Name 'st',
6      [
7          Alloc(Writable(), IntType('int'), Name('attr1'))
8          Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9      ],
10     FunDecl
11     VoidType 'void',
12     Name 'stack_fun',
13     [
14         Alloc
15         Writable,
16         ArrayDecl
17         [
18             Num '2',
19             Num '3'
20         ],
21         StructSpec
22         Name 'st',
23         Name 'param'
24     ],
25     FunDef
26     VoidType 'void',
27     Name 'main',
28     [],
29     [
30         Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
31             ↪ Name('local_var')))
32         Exp(Call(Name('stack_fun'), [Name('local_var')]))
33         Return(Empty())
34     ],
35     FunDef
36     VoidType 'void',
37     Name 'stack_fun',
38     [
39         Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
40             ↪ Name('param'))
41     ],
42     [
43         Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
44     ]
45 ]

```

Code 1.27: Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.

Im **PicoC-ANF Pass** in Code 1.28 wird die Komposition `Exp(Call(Name('stack_fun'), [Name('local_var')]))` durch die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'))`, `GoTo(Name('addr@next_instr'))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` ersetzt, welche in den Tabellen ?? und ?? genauer erklärt sind.

Der Knoten `StackMalloc(Num('2'))` ist notwendig, weil auf dem **Stackframe** für den Wert des **BAF-Registers** der **aufzufendenden Funktion** und die **Rücksprungadresse** 2 Speicherzellen Platz am **Anfang** des **Stackframes** gelassen werden muss. Das wird durch den Knoten `StackMalloc(Num('2'))` umgesetzt, indem das **SP-Register** einfach um zwei Speicherzellen **dekrementiert** wird und somit Speicher auf dem **Stack** belegt wird³⁸.

³⁸Wobei hier "reserviert" besser passen würde.

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc(Num('2'))
8         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
10        Exp(GoTo(Name('stack_fun.0'))))
11        RemoveStackframe()
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]

```

Code 1.28: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 1.29 werden die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch ihre entsprechenden **RETI-Knoten** ersetzt.

Unter den **RETI-Knoten** entsprechen die **Kompositionen** `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` noch keine fertigen **RETI-Befehlen** und werden später in dem für sie vorgesehenen **RETI-Pass** passend ergänzt bzw. ersetzt.

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Assoziativen Feld** `fun_name_to_block_name`³⁹ mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```

1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # StackMalloc(Num('2'))
8         SUBI SP 2;
9         # Ref(Global(Num('0')))
10        SUBI SP 1;
11        LOADI IN1 0;
12        ADD IN1 DS;
13        STOREIN SP IN1 1;
14        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
15        MOVE BAF ACC;
16        ADDI SP 3;
17        MOVE SP BAF;
18        SUBI SP 4;

```

³⁹Dieses Assoziative Feld wurde in Unterkapitel 1.5.3.1 eingeführt.

```

19     STOREIN BAF ACC 0;
20     LOADI ACC GoTo(Name('addr@next_instr'));
21     ADD ACC CS;
22     STOREIN BAF ACC -1;
23     # Exp(GoTo(Name('stack_fun.0')))
24     Exp(GoTo(Name('stack_fun.0')))
25     # RemoveStackframe()
26     MOVE BAF IN1;
27     LOADIN IN1 BAF 0;
28     MOVE IN1 SP;
29     # Return(Empty())
30     LOADIN BAF PC -1;
31 ],
32 Block
33   Name 'stack_fun.0',
34   [
35     # Return(Empty())
36     LOADIN BAF PC -1;
37   ]
38 ]

```

Code 1.29: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 1.29 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist⁴⁰ ersetzt: `LOADI ACC 14`. Und auch der Knoten, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den Knoten `JUMP 5` ersetzt.

Die **Distanz 5** im **RETI-Knoten** `JUMP 5` wird mithilfe des `instrs.before`-Attribute des **Zielblocks**, der den ersten Befehl der gewünschten Funktion enthält und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` enthalten ist berechnet.

Die **relative Adresse 14** direkt nach dem Befehl `JUMP 5` wird ebenfalls mithilfe des `instrs.before`-Attributs des **aktuellen Blocks** berechnet. Es handelt sich bei bei 14 um eine **relative Adresse**, die **relativ** zum `CS`-Register berechnet wird, welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

Anmerkung

Die Berechnung der **Adresse** '`<addr@next_instr>`' (bzw. in der Formel adr_{danach}) des Befehls nach dem **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt dabei mithilfe der folgenden Formel:

$$adr_{danach} = \#Bef_{vor\ akt.\ Bl.} + idx + 4 \quad (1.5.1)$$

wobei:

- es sich bei bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum `CS`-Register berechnet wird.
- $\#Bef_{vor\ akt.\ Bl.} \hat{=}$ **Anzahl** Befehle vor dem momentanen Block. Es handelt sich hierbei um ein verstecktes Attribut `instrs.before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs.before,`

⁴⁰Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

`num_instrs, param_size, local_vars_size`), welches im **RETI-Patch**-Pass gesetzt wird. Der Grund dafür, dass das Zuweisen dieses versteckten Attributes `instrs_before` im **RETI-Patch** Pass erfolgt ist, weil erst im **RETI-Patch** Pass die **finale Anzahl** an Befehlen in einem Block feststeht, da im **RETI-Patch** Pass `GoTo()`'s entfernt werden, deren Sprung nur **eine** Adresse weiterspringen würde. Die **finale Anzahl** an Befehlen kann sich in diesem **Pass** also noch ändern und steht erst nach diesem **Pass** fest.

- $idx \hat{=}$ relativer Index des Befehls `LOADI ACC <addr@next_instr>` selbst im Block.
- 4 $\hat{=}$ **Distanz**, die zwischen den in Code 1.30 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

Die Berechnung der **Distanz** `<distanz>` für den Sprung `JUMP <distanz>` zum **ersten** Befehl eines im **Pass** zuvor **existenten Blockes** erfolgt dabei nach der folgenden Formel:

$$distanz = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} \neq \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases} \quad (1.5.2)$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und zu dem gesprungen werden soll. Es handelt sich hierbei um ein verstecktes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung** wie in der Formel 1.5.1.

```

1 # // Exp(GoTo(Name('main.1')))
2 # // not included Exp(GoTo(Name('main.1')))
3 # StackMalloc(Num('2'))
4 SUBI SP 2;
5 # Ref(Global(Num('0')))
6 SUBI SP 1;
7 LOADI IN1 0;
8 ADD IN1 DS;
9 STOREIN SP IN1 1;
10 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
11 MOVE BAF ACC;
12 ADDI SP 3;
13 MOVE SP BAF;
14 SUBI SP 4;
15 STOREIN BAF ACC 0;
16 LOADI ACC 14;
17 ADD ACC CS;
18 STOREIN BAF ACC -1;
19 # Exp(GoTo(Name('stack_fun.0')))
20 JUMP 5;
21 # RemoveStackframe()
22 MOVE BAF IN1;
23 LOADIN IN1 BAF 0;
24 MOVE IN1 SP;
25 # Return(Empty())
26 LOADIN BAF PC -1;
27 # Return(Empty())

```

```
28 LOADIN BAF PC -1;
```

Code 1.30: *RETI-Pass für Funktionsaufruf ohne Rückgabewert.*

1.5.3.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 1.31 erklärt.

Um den Unterschied zwischen einem `return` ohne **Rückgabewert** und einem `return 21 * 2` mit **Rückgabewert** hervorzuheben, wurde ist auch eine Funktion `fun_no_return_value`, die **keinen** Rückgabewert hat in das Beispiel integriert.

```
1 int fun_with_return_value() {
2     return 21 * 2;
3 }
4
5 void fun_no_return_value() {
6     return;
7 }
8
9 void main() {
10     int var = fun_with_return_value();
11     fun_no_return_value();
12 }
```

Code 1.31: *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 1.32 wird eine **Return-Anweisung mit Rückgabewert** `return 21 * 2` mit der Komposition `Return(BinOp(Num('21'), Mul('*', Num('2'))))` dargestellt, eine **Return-Anweisung ohne Rückgabewert** `return` mit der Komposition `Return(Empty())` und ein **Funktionsaufruf inklusive Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` durch die Komposition `Assign(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))`.

```
1 File
2   Name './example_fun_call_with_return_value.ast',
3   [
4       FunDef
5         IntType 'int',
6         Name 'fun_with_return_value',
7         [],
8         [
9             Return(BinOp(Num('21'), Mul('*', Num('2'))))
10        ],
11        FunDef
12          VoidType 'void',
13          Name 'fun_no_return_value',
14          [],
15          [
16              Return(Empty())
```

```

17     ],
18     FunDef
19       VoidType 'void',
20       Name 'main',
21       [],
22       [
23         Assign(Alloc(Writeable(), IntType('int'), Name('var')),
24           ↪ Call(Name('fun_with_return_value'), []))
25         Exp(Call(Name('fun_no_return_value'), []))
26       ]

```

Code 1.32: Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.

Im **PicoC-ANF Pass** in Code 1.33 wird bei der **Komposition** `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die **Expression** `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierfür erstellten Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis wird dann von der **Komposition** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register ACC** geschrieben und als letztes wird die **Rücksprungsadresse** in das PC-Register geladen, die durch den `NewStackframe()`-Token-Knoten eine Speicherzelle nach dem Wert des BAF-Registers der aufrufenden Funktion im **Stackframe** gespeichert ist.

Ein wichtiges Detail bei der **Funktion** `fun_with_return_value` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird, da die **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Um den **Rückgabewert**, der durch die Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das ACC-Register geschrieben wurde für die aufrufende Funktion, deren Stackframe nun wieder das aktuelle ist auf den **Stack** zu schreiben, muss ein neue **Komposition** `Exp(ACC)` definiert werden. In Tabelle ?? ist die **Komposition** `Exp(ACC)` genauer erklärt.

Dieser Trick mit dem Speichern des **Rückgabewertes** im ACC-Register ist notwendig, weil durch das **Entfernen** des **Stackframes** der **aufgerufenen Funktion** das SP-Register nicht mehr an der gleichen Stelle steht. Daher sind alle **temporären** Werte, die in der **aufgerufenen Funktion** auf den **Stack** geschrieben wurden unzugänglich, weil man nicht wissen kann, um wieviel die Adresse im SP-Register verglichen zu vorher verschoben ist, weil der **Stackframe** von unterschiedlichen **aufgerufenen Funktionen** unterschiedlich groß sein kann.

Die **Komposition** `Assign(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))` wird nach dem **allokieren** der Variable `Name('var')` durch die Komposition `Assign(Global(Num('0')), Stack(Num('1')))` ersetzt, welche den **Rückgabewert** der Funktion `Name('fun_with_return_value')`, welcher durch die **Komposition** `Exp(ACC)` aus dem ACC-Register auf den **Stack** geschrieben wurde nun vom **Stack** in die Speicherzelle der Variable `Name('var')` speichert. Hierzu muss die **Adresse** der Variable `Name('var')` in der **Symboltabelle** nachgeschlagen werden.

Die **Komposition** `Return(Empty())` für ein **return ohne Rückgabewert** bleibt unverändert und stellt nur das Laden der **Rücksprungsadresse** in das PC-Register dar.

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` keine **return**-Anweisung explizit ans Ende geschrieben wird, im **PicoC-ANF Pass** eines hinzugefügt wird in Form der Komposition `Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine **MissingReturn-Fehlermeldung** ausgelöst.


```

1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         Exp(Num('21'))
9         Exp(Num('2'))
10        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
11        Return(Stack(Num('1')))
12      ],
13    Block
14      Name 'fun_no_return_value.1',
15      [
16        Return(Empty())
17      ],
18    Block
19      Name 'main.0',
20      [
21        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
22        StackMalloc(Num('2'))
23        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
24        Exp(GoTo(Name('fun_with_return_value.2')))
25        RemoveStackframe()
26        Exp(ACC)
27        Assign(Global(Num('0')), Stack(Num('1')))
28        StackMalloc(Num('2'))
29        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
30        Exp(GoTo(Name('fun_no_return_value.1')))
31        RemoveStackframe()
32        Return(Empty())
33      ]
34    ]

```

Code 1.33: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 1.34 werden die Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))`, `Return(Stack(Num('1')))` und `Assign(Global(Num('0')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         # Exp(Num('21'))
9         SUBI SP 1;
10        LOADI ACC 21;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;

```

```

14     LOADI ACC 2;
15     STOREIN SP ACC 1;
16     # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17     LOADIN SP ACC 2;
18     LOADIN SP IN2 1;
19     MULT ACC IN2;
20     STOREIN SP ACC 2;
21     ADDI SP 1;
22     # Return(Stack(Num('1')))
23     LOADIN SP ACC 1;
24     ADDI SP 1;
25     LOADIN BAF PC -1;
26 ],
27 Block
28   Name 'fun_no_return_value.1',
29   [
30     # Return(Empty())
31     LOADIN BAF PC -1;
32   ],
33 Block
34   Name 'main.0',
35   [
36     # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
37     # StackMalloc(Num('2'))
38     SUBI SP 2;
39     # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
40     MOVE BAF ACC;
41     ADDI SP 2;
42     MOVE SP BAF;
43     SUBI SP 2;
44     STOREIN BAF ACC 0;
45     LOADI ACC GoTo(Name('addr@next_instr'));
46     ADD ACC CS;
47     STOREIN BAF ACC -1;
48     # Exp(GoTo(Name('fun_with_return_value.2')))
49     Exp(GoTo(Name('fun_with_return_value.2')))
50     # RemoveStackframe()
51     MOVE BAF IN1;
52     LOADIN IN1 BAF 0;
53     MOVE IN1 SP;
54     # Exp(ACC)
55     SUBI SP 1;
56     STOREIN SP ACC 1;
57     # Assign(Global(Num('0')), Stack(Num('1')))
58     LOADIN SP ACC 1;
59     STOREIN DS ACC 0;
60     ADDI SP 1;
61     # StackMalloc(Num('2'))
62     SUBI SP 2;
63     # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
64     MOVE BAF ACC;
65     ADDI SP 2;
66     MOVE SP BAF;
67     SUBI SP 2;
68     STOREIN BAF ACC 0;
69     LOADI ACC GoTo(Name('addr@next_instr'));
70     ADD ACC CS;

```

```

71     STOREIN BAF ACC -1;
72     # Exp(GoTo(Name('fun_no_return_value.1')))
73     Exp(GoTo(Name('fun_no_return_value.1')))
74     # RemoveStackframe()
75     MOVE BAF IN1;
76     LOADIN IN1 BAF 0;
77     MOVE IN1 SP;
78     # Return(Empty())
79     LOADIN BAF PC -1;
80 ]
81 ]

```

Code 1.34: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.*

1.5.3.3.2 Umsetzung der Übergabe eines Feldes

Die Umsetzung der **Übergabe** eines **Feldes** an eine andere Funktion wird im Folgenden mithilfe des Beispiels in Code 1.35 erklärt.

```

1 void fun_array_from_stackframe(int (*param)[3]) {
2 }
3
4 void fun_array_from_global_data(int param[2][3]) {
5     int local_var[2][3];
6     fun_array_from_stackframe(local_var);
7 }
8
9 void main() {
10     int local_var[2][3];
11     fun_array_from_global_data(local_var);
12 }

```

Code 1.35: *PicoC-Code für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** wird im Fall dessen, dass der **oberste Knoten** im Teilbaum, der den Datentyp darstellt und an die Funktion übergeben wird ein **Feld** `ArrayDecl(nums, datatype)` ist, dieser zu einem **Zeiger** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, an das `datatype`-Attribut des **Zeigers** `PntrDecl(num, datatype)` drangehängt.

Diese **Umwandlung** des **Datentyps** kann in der **Symboltabelle** in Code 1.36 beobachtet werden. Die **lokalen Variablen** `local_var@main` und `local_var@fun_array_from_global_data` sind beide vom Datentyp `ArrayDecl([Num('2'), Num('3')], IntType('int'))` und bei der Übergabe ändert sich der Datentyp beider Variablen zu `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))`. Die **Größe** dieser Variablen ändert sich damit zu `Num('1')`, da ein **Zeiger** nur eine **Speicherzelle** braucht.

```

1 SymbolTable
2 [
3     Symbol
4     {
5         type qualifier:      Empty()

```

```

6      datatype:      FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
    ↪ [Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
    ↪ Name('param'))])
7      name:          Name('fun_array_from_stackframe')
8      value or address: Empty()
9      position:      Pos(Num('1'), Num('5'))
10     size:          Empty()
11 },
12 Symbol
13 {
14     type qualifier:  Writable()
15     datatype:      PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16     name:          Name('param@fun_array_from_stackframe')
17     value or address: Num('0')
18     position:      Pos(Num('1'), Num('37'))
19     size:          Num('1')
20 },
21 Symbol
22 {
23     type qualifier:  Empty()
24     datatype:      FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
    ↪ [Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
25     name:          Name('fun_array_from_global_data')
26     value or address: Empty()
27     position:      Pos(Num('4'), Num('5'))
28     size:          Empty()
29 },
30 Symbol
31 {
32     type qualifier:  Writable()
33     datatype:      PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
34     name:          Name('param@fun_array_from_global_data')
35     value or address: Num('0')
36     position:      Pos(Num('4'), Num('36'))
37     size:          Num('1')
38 },
39 Symbol
40 {
41     type qualifier:  Writable()
42     datatype:      ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:          Name('local_var@fun_array_from_global_data')
44     value or address: Num('6')
45     position:      Pos(Num('5'), Num('6'))
46     size:          Num('6')
47 },
48 Symbol
49 {
50     type qualifier:  Empty()
51     datatype:      FunDecl(VoidType('void'), Name('main'), [])
52     name:          Name('main')
53     value or address: Empty()
54     position:      Pos(Num('9'), Num('5'))
55     size:          Empty()
56 },
57 Symbol
58 {
59     type qualifier:  Writable()

```

```

60     datatype:      ArrayDecl([Num('2'), Num('3')], IntType('int'))
61     name:          Name('local_var@main')
62     value or address: Num('0')
63     position:      Pos(Num('10'), Num('6'))
64     size:          Num('6')
65 }
66 ]

```

Code 1.36: *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 1.37 ist zu sehen, dass zur Übergabe der beiden Felder die **Adresse** des jeweiligen Feldes auf den **Stack** geschrieben wird. Die **Adressen** der beiden Felder auf den **Stack** zu schreiben wird durch die Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` repräsentiert.

Die Komposition `Ref(Global(Num('0')))` ist für Variablen in den **Globalen Statischen Daten** und die Komposition `Ref(Stackframe(Num('6')))` ist für Variablen aus dem **Stackframe**. Dabei stellen die Zahlen in den Knoten `Global(num)` bzw. `Stackframe(num)` die **relative Adressen** relativ zum DS-Register bzw. SP-Register dar, die aus der **Symboltabelle** entnommen sind.

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'fun_array_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_array_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Ref(Stackframe(Num('6')))
14        NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_array_from_stackframe.2')))
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Ref(Global(Num('0')))
24        NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
25        Exp(GoTo(Name('fun_array_from_global_data.1')))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]

```

Code 1.37: *PicoC-ANF Pass .*

Im **RETI-Blocks Pass** in Code 1.38 werden Kompositionen `Ref(Global(Num('0')))` und

`Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'fun_array_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun_array_from_global_data.1',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Ref(Stackframe(Num('6')))
16        SUBI SP 1;
17        MOVE BAF IN1;
18        SUBI IN1 8;
19        STOREIN SP IN1 1;
20        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
21        MOVE BAF ACC;
22        ADDI SP 3;
23        MOVE SP BAF;
24        SUBI SP 3;
25        STOREIN BAF ACC 0;
26        LOADI ACC GoTo(Name('addr@next_instr'));
27        ADD ACC CS;
28        STOREIN BAF ACC -1;
29        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
30        Exp(GoTo(Name('fun_array_from_stackframe.2')))
31        # RemoveStackframe()
32        MOVE BAF IN1;
33        LOADIN IN1 BAF 0;
34        MOVE IN1 SP;
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ],
38    Block
39      Name 'main.0',
40      [
41        # StackMalloc(Num('2'))
42        SUBI SP 2;
43        # Ref(Global(Num('0')))
44        SUBI SP 1;
45        LOADI IN1 0;
46        ADD IN1 DS;
47        STOREIN SP IN1 1;
48        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
49        MOVE BAF ACC;
50        ADDI SP 3;
51        MOVE SP BAF;
52        SUBI SP 9;
53        STOREIN BAF ACC 0;
54        LOADI ACC GoTo(Name('addr@next_instr'));
55        ADD ACC CS;

```

```

56     STOREIN BAF ACC -1;
57     # Exp(GoTo(Name('fun_array_from_global_data.1')))
58     Exp(GoTo(Name('fun_array_from_global_data.1')))
59     # RemoveStackframe()
60     MOVE BAF IN1;
61     LOADIN IN1 BAF 0;
62     MOVE IN1 SP;
63     # Return(Empty())
64     LOADIN BAF PC -1;
65 ]
66 ]

```

Code 1.38: RETI-Block Pass für die Übergabe eines Feldes.

1.5.3.3.3 Umsetzung einer Übergabe eines Verbundes

Die Umsetzung der **Übergabe** eines **Verbundes** wird im Folgenden mithilfe des Beispiels in Code 1.39 erklärt.

```

1 struct st {int attr1; int attr2[2];};
2
3
4 void fun_struct_from_stackframe(struct st param) {
5 }
6
7 void fun_struct_from_global_data(struct st param) {
8     fun_struct_from_stackframe(param);
9 }
10
11
12 void main() {
13     struct st local_var;
14     fun_struct_from_global_data(local_var);
15 }

```

Code 1.39: PicoC-Code für die Übergabe eines Verbundes.

Im **PicoC-ANF Pass** in Code 1.40 wird zur **Übergabe eines Verbundes**, der komplette Verbund auf den **Stack** kopiert. Das wird mittels der Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` dargestellt.

Bei der **Übergabe** an eine **Funktion** wird der Zugriff auf einen gesamten **Verbund** anders gehandhabt als sonst. Normalerweise wird beim Zugriff auf einen Verbund die **Adresse** des **ersten Attributs** dieses Verbundes auf den **Stack** geschrieben. Bei der **Übergabe an eine Funktion** wird dagegen der gesamte **Verbund** auf den **Stack** kopiert.

Das wird durch eine Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange der **Funktionsaufruf** im **PicoC-ANF Pass** verarbeitet wird und wieder auf `false` gesetzt, wenn die Verarbeitung des **Funktionsaufrufs** abgeschlossen ist. Solange die Variable `argmode_on` auf `true` gesetzt ist, wird immer die Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')),`

`Stackframe(Num('2'))` für die Ersetzung verwendet. Ist die Variable `argmode_on` auf `false` wird die Komposition `Ref(Globalnum())` bzw. `Ref(Stackframe(num))` für die Ersetzung verwendet.

Die Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` wird im Falle dessen, dass die Verbundsvariable in den **Globalen Statischen Daten** liegt verwendet und die Komposition `Assign(Stack(Num('3')), Global(Num('0')))` wird im Falle, dessen, dass die Verbundsvariable im **Stackframe** liegt verwendet.

```

1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_struct_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Stackframe(Num('2'))))
14        NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr'))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2'))))
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Assign(Stack(Num('3')), Global(Num('0'))))
24        NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr'))
25        Exp(GoTo(Name('fun_struct_from_global_data.1'))))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]

```

Code 1.40: *PicoC-ANF Pass für die Übergabe eines Verbundes.*

Im **RETI-Blocks Pass** in Code 1.41 werden die Kompositionen `Assign(Stack(Num('3')), Stackframe(Num('2')))` und `Assign(Stack(Num('3')), Global(Num('0')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block

```



```

11     Name 'fun_struct_from_global_data.1',
12     [
13         # StackMalloc(Num('2'))
14         SUBI SP 2;
15         # Assign(Stack(Num('3')), Stackframe(Num('2')))
16         SUBI SP 3;
17         LOADIN BAF ACC -4;
18         STOREIN SP ACC 1;
19         LOADIN BAF ACC -3;
20         STOREIN SP ACC 2;
21         LOADIN BAF ACC -2;
22         STOREIN SP ACC 3;
23         # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
24         MOVE BAF ACC;
25         ADDI SP 5;
26         MOVE SP BAF;
27         SUBI SP 5;
28         STOREIN BAF ACC 0;
29         LOADI ACC GoTo(Name('addr@next_instr'));
30         ADD ACC CS;
31         STOREIN BAF ACC -1;
32         # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
33         Exp(GoTo(Name('fun_struct_from_stackframe.2')))
34         # RemoveStackframe()
35         MOVE BAF IN1;
36         LOADIN IN1 BAF 0;
37         MOVE IN1 SP;
38         # Return(Empty())
39         LOADIN BAF PC -1;
40     ],
41     Block
42     Name 'main.0',
43     [
44         # StackMalloc(Num('2'))
45         SUBI SP 2;
46         # Assign(Stack(Num('3')), Global(Num('0')))
47         SUBI SP 3;
48         LOADIN DS ACC 0;
49         STOREIN SP ACC 1;
50         LOADIN DS ACC 1;
51         STOREIN SP ACC 2;
52         LOADIN DS ACC 2;
53         STOREIN SP ACC 3;
54         # NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
55         MOVE BAF ACC;
56         ADDI SP 5;
57         MOVE SP BAF;
58         SUBI SP 5;
59         STOREIN BAF ACC 0;
60         LOADI ACC GoTo(Name('addr@next_instr'));
61         ADD ACC CS;
62         STOREIN BAF ACC -1;
63         # Exp(GoTo(Name('fun_struct_from_global_data.1')))
64         Exp(GoTo(Name('fun_struct_from_global_data.1')))
65         # RemoveStackframe()
66         MOVE BAF IN1;
67         LOADIN IN1 BAF 0;

```

```
68      MOVE IN1 SP;  
69      # Return(Empty())  
70      LOADIN BAF PC -1;  
71  ]  
72 ]
```

Code 1.41: *RETI-Block Pass für die Übergabe eines Verbundes.*

Literatur

Online

- *clang: C++ Compiler*. URL: <http://clang.org/> (besucht am 29.07.2022).
- *Clockwise/Spiral Rule*. URL: <https://c-faq.com/decl/spiral.anderson.html> (besucht am 29.07.2022).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: <https://inkscape.org/> (besucht am 03.08.2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*. URL: <https://www.coder-welten.de/einstieg/variablen-in-c-3.html> (besucht am 11.08.2022).
- *Welcome to Lark's documentation! — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/> (besucht am 31.07.2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: <https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/> (besucht am 18.07.2022).

Bücher

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

Vorlesungen

- Bast, Prof. Dr. 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, Prof. Dr. 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, Prof. Dr. 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, Prof. Dr. 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

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).
- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03.08.2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: <https://github.com/tueda/makefile4latex> (besucht am 03.08.2022).