Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 13. September 2022

Autor:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

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

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

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

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

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 Leidenschafft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias[1] 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 habe[2], er hat sich bei der Korrektur dieser Schrifftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird. Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu seinem eigenen Nachteil[3] [4] weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

[1] Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.
[2] Wofür ich mich auch nochmal Entschuldigen will.
[3] Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.
[4] Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes 😉.

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

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

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

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt[6]. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht[7]. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiesprache kann in der Verwendung ziemlich wilkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

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

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

[5] https://github.com/michel-giehl/Reti-Emulator.
[6] Womit nicht alle Studenten so viel Glück haben.
[7] Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis **I**

Codeverzeichnis **II**

Tabellenverzeichnis **IV**

Definitionsverzeichnis **V**

Grammatikverzeichnis **VI**

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 Still der Schrifftlichen Ausarbeitung . . . 13
1.5.2 Aufbau der Schrifftlichen Arbeit . . . 14
1.5.3 Umsetzung von Verbunden . . . 16
1.5.3.1 Deklaration von Verbundstypen und Definition von Verbunden . . . 16
1.5.3.2 Initialisierung von Verbunden . . . 18
1.5.3.3 Zugriff auf Verbundsattribut . . . 21
1.5.3.4 Zuweisung an Verbundsattribut . . . 25
1.5.4 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen . . . 27
1.5.4.1 Anfangsteil . . . 30
1.5.4.2 Mittelteil . . . 33
1.5.4.3 Schlussteil . . . 36
1.5.5 Umsetzung von Funktionen . . . 41
1.5.5.1 Mehrere Funktionen . . . 41
1.5.5.1.1 Sprung zur Main Funktion . . . 44
1.5.5.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen 47
1.5.5.3 Funktionsaufruf . . . 49
1.5.5.3.1 Rückgabewert . . . 54
1.5.5.3.2 Umsetzung der Übergabe eines Feldes . . . 58
1.5.5.3.3 Umsetzung einer Übergabe eines Verbundes . . . 62

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
1.5 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen. 28

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 die Deklaration eines Verbundstyps. . . . 16
1.16 Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps. . . . 16
1.17 Symboltabelle für die Deklaration eines Verbundstyps. . . . 18
1.18 PicoC-Code für Initialisierung von Verbunden. . . . 18
1.19 Abstrakter Syntaxbaum für Initialisierung von Verbunden. . . . 19
1.20 PicoC-ANF Pass für Initialisierung von Verbunden. . . . 20
1.21 RETI-Blocks Pass für Initialisierung von Verbunden. . . . 21
1.22 PicoC-Code für Zugriff auf Verbundsattribut. . . . 21
1.23 Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut. . . . 22
1.24 PicoC-ANF Pass für Zugriff auf Verbundsattribut. . . . 24
1.25 RETI-Blocks Pass für Zugriff auf Verbundsattribut. . . . 25
1.26 PicoC-Code für Zuweisung an Verbundsattribut. . . . 25
1.27 Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut. . . . 26
1.28 PicoC-ANF Pass für Zuweisung an Verbundsattribut. . . . 26
1.29 RETI-Blocks Pass für Zuweisung an Verbndsattribut. . . . 27
1.30 PicoC-Code für den Anfangsteil. . . . 31
1.31 Abstrakter Syntaxbaum für den Anfangsteil. . . . 32
1.32 PicoC-ANF Pass für den Anfangsteil. . . . 32
1.33 RETI-Blocks Pass für den Anfangsteil. . . . 33
1.34 PicoC-Code für den Mittelteil. . . . 33
1.35 Abstrakter Syntaxbaum für den Mittelteil. . . . 34
1.36 PicoC-ANF Pass für den Mittelteil. . . . 34
1.37 RETI-Blocks Pass für den Mittelteil. . . . 36
1.38 PicoC-Code für den Schlussteil. . . . 36
1.39 Abstrakter Syntaxbaum für den Schlussteil. . . . 37
1.40 PicoC-ANF Pass für den Schlussteil. . . . 38
1.41 RETI-Blocks Pass für den Schlussteil. . . . 40
1.42 PicoC-Code für 3 Funktionen. . . . 41
1.43 Abstrakter Syntaxbaum für 3 Funktionen. . . . 42
1.44 PicoC-Blocks Pass für 3 Funktionen. . . . 43
1.45 PicoC-ANF Pass für 3 Funktionen. . . . 43
1.46 RETI-Blocks Pass für 3 Funktionen. . . . 44
1.47 PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . . 44

1.48 RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . . 45
1.49 RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . . 46
1.50 RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. 47
1.51 PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss. 47
1.52 Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss. 49
1.53 PicoC-Code für Funktionsaufruf ohne Rückgabewert. 49
1.54 Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert. 50
1.55 PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert. 51
1.56 RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert. 52
1.57 RETI-Pass für Funktionsaufruf ohne Rückgabewert. 54
1.58 PicoC-Code für Funktionsaufruf mit Rückgabewert. 54
1.59 Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert. 55
1.60 PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert. 56
1.61 RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert. 58
1.62 PicoC-Code für die Übergabe eines Feldes. 58
1.63 Symboltabelle für die Übergabe eines Feldes. 60
1.64 PicoC-ANF Pass für die Übergabe eines Feldes. 61
1.65 RETI-Block Pass für die Übergabe eines Feldes. 62
1.66 PicoC-Code für die Übergabe eines Verbundes. 63
1.67 PicoC-ANF Pass für die Übergabe eines Verbundes. 63
1.68 RETI-Block Pass für die Übergabe eines Verbundes. 65

Tabellenverzeichnis

1.1 Präzedenzregeln von PicoC. 3

Definitionsverzeichnis

1.1 Imperative Programmierung 5
1.2 Strukturierte Programmierung 5
1.3 Prozedurale Programmierung 6
1.4 Call by Value 8
1.5 Call by Reference 9
1.6 Funktionsprototyp 9
1.7 Deklaration 10
1.8 Definition 10
1.9 Sichtbarkeitsbereich (bzw. engl. Scope) 11
1.10 Entarteter Baum 24

Grammatikverzeichnis

1 Motivation

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiel** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprachen **Python**, welche als **interpretierte** Sprache bekannt ist, wird das in der Programmiersprache **Python** geschriebene Programm vorher zu **Bytecode** kompiliert, bevor dieser von der **Python Virtual Machine (PVM)** interpretiert wird. Die Programmiersprache **Python** und jegliche **andere Sprache** wird fortan als L_{Python} bzw. als $L_{Name\ der\ Sprache}$ bezeichnet 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 zu **Maschinencode** zu kompilieren. Das Programm muss hierzu in der Programmiersprache geschrieben sein, die der Compiler kompiliert[3]

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 z.B. unter **Unix-Systemen** über `> ./machine_code` **ausgeführt** werden kann, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

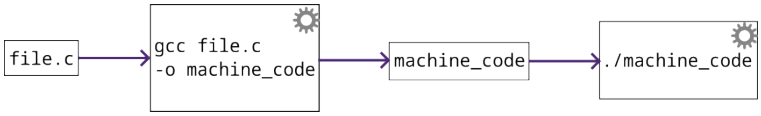


Abbildung 1.1: *Schritte zum Ausführen eines Programmes mit dem GCC.*

Der ganze Kompiliervorgang kann, wie er in Abbildung 1.2 dargestellt ist zu einer Box `Compiler` abstrahiert werden. Der Benutzer gibt ein **Programm** in der Sprache des Compilers rein und erhält **Maschinencode**, 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. Die Maschine bzw. der Interpreter kann den **Maschinencode** dann **ausführen**.

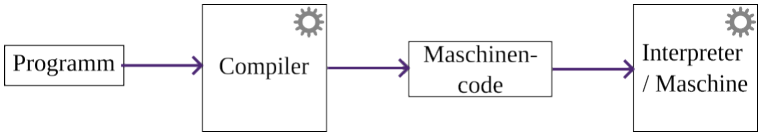


Abbildung 1.2: *Stark vereinfachte Schritte zum Ausführen eines Programmes.*

[1] *GCC, the GNU Compiler Collection - GNU Project*.
[2] *clang: C++ Compiler*.
[3] Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C.
[4] Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **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 funktioniert.

Die konkrete **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC}, welche eine Untermenge der Sprache L_C ist[5] in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann.

In dieser Motivation werden für diese Bachelorarbeit **elementare** Thematiken und Informationen erstmals angeschnitten. 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 die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll. Des Weiteren wird in Unterkapitel 1.3 insbesondere auf bestimmte **Eigenheiten der Sprachen** L_C und L_{PicoC} eingegangen, auf welche in dieser Bachelorarbeit ein besonderes Augenmerk gerichtet wird. Danach wird in Unterkapitel 1.4 auf für diese Bachelorarbeit gesetzte **Schwerpunkte** eingegangen und in Unterkapitel 1.5 etwas zum **Aufbau** und **Still** dieser Schrifftlichen Ausarbeitung der Bachelorarbeit gesagt.

1.1 RETI-Architektur

Die **RETI-Architektur** ist eine zu Lernzwecken für die Vorlesungen 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 C. Scholl, „Betriebssysteme“ wird diese Architektur erweitert, sodass diese mehr darauf angepasst ist, dass auch komplexere Kontrukte, wie ein **Betriebssystem**, **Interrupts**, **Prozesse**, **Funktionen** usw. auf nicht zu komplizierte Weise implementiert werden können.

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

Anmerkung

In dieser **Bachelorarbeit** wird **im Folgenden** bei der **Maschinensprache** L_{RETI} immer von der Variante, welche durch die **RETI-Architektur** aus der Vorlesung 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 **??** zusammengenommen beschrieben. Für genauere **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und C. Scholl, „Betriebssysteme“ zu verweisen.

[5] Die 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	Begin 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.

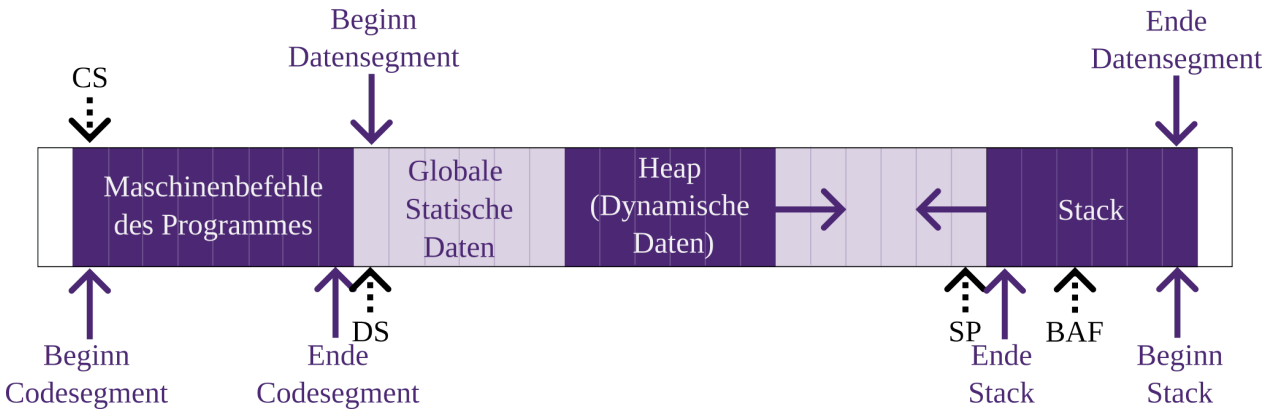


Abbildung 1.3: *Speicherorganisation.*

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} Adressen[7] unter sich aufteilen.

Die **Ausführung** eines Programmes **startet** auf die einfachste Weise, indem es von einem **Startprogramm** im **EPROM**[8] aufgerufen wird. Der **EPROM** wird beim Start einer **RETI-CPU** als **erstes** aufgerufen, 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**. Es gibt allerdings immer einen `RX`- und einen `TX`-Pin, für jeweils Empfangen[10] und Versenden[11] von Daten. Jeder der **Pins** wird dabei mit einer anderen **Adresse** von 2^3 verschiedenen Adressen angsprochen. Jeweils 8-Bit können nach den **Datenpfaden der RETI-CPU** **??** 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** `/* comment */`.
- die **Basisdatentypen**[13] `int`, `char` und `void`.
- die **Zusammengesetzten** Datentypen[14] **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`) und ihre zugehörigen **Operationen** `[i]`, `.attr` und `*` usw.
- `if(cond){ }`- und `else{ }`-**Anweisungen**[15].
- `while(cond){ }`- und `do while(cond){ };`-**Anweisungen**.
- **Arithmetische und Bitweise Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` und **unären Operatoren** `-`, $\sim$ umgesetzt sind.[16]
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.

[6] Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, sondern nur bei der Umsetzung des **RETI-Interpreters**, wird diese **nicht näher erläutert** als notwendig.
[7] Von 0 bis $2^{32} - 1$.
[8] Kurz für **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
[9] Kurz für **U**niversal **A**synchronous **R**eceiver **T**ransmitter.
[10] Engl. **R**eceiving, daher das `R`.
[11] Engl. **T**ransmission, daher das `T`.
[12] Kurz für **S**tatic **r**andom-**a**ccess **m**emory.
[13] Bzw. `int` und `char` werden auch als **Primitive Datentypen** bezeichnet.
[14] Bzw. engl. **compound datatypes**.
[15] Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.
[16] Theoretisch sind die Operatoren `<<`, `>>` und $\sim$ unnötig, da sie durch **Multiplikation** `*`, **Division** `/` und Anwendung des **Xor**-^-Operators auf eine Zahl, deren **binäre Repräsentation** ein Folge von 1en **gleicher Länge** ist ersetzt werden können.

- **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)`).

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 **??** zusammengenommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache L_C, die genauso ein Teil der Programmiersprache L_{PicoC} sind, 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 erläutertem Grund.

Bei der Programmiersprache L_{PicoC} handelt es sich im eine **imperative** (Definition 1.1), **strukturierte** (Definition 1.2) und **prozedurale Programmiersprache** (Definition 1.3). Aufgrund dessen, dass es sich 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**[17] 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]

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

Definition 1.2: Strukturierte Programmierung

*Wenn ein Programm anstelle von z.B. `goto label`-Anweisungen **Kontrollstruturen**, 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** umbedingt immer **gleich aufgebaut** wäre.*[a]

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

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

Definition 1.3: Prozedurale Programmierung

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

[a] Thiemann, „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>]`-Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: `<remaining-datatype><var>[<i>]`, während andere Programmiersprachen die eckigen `[<i>]`-Klammern **vor** die Variable schreiben `<remaining-datatype>[<i>]<var>`.

Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, ist es **schwieriger** den **Datentyp abzulesen**, als auch ein **Programm zu implementieren** was diesen erkennt. Damit ein Programmierer den **Datentyp ablesen** kann, kann dieser die **Spiralregel** verwenden, die unter der Webseite *Clockwise/Spiral Rule* nachgelesen werden kann. Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, wirken diese zum verwechseln ähnlich zum `<var>[<i>]`-Operator für den **Zugriff auf den Index eines Feldes**. Wenn ein Ausdruck geschrieben wird, wie `int ar[1] = {42}` wird, ist dieser vom Ausdruck `var[0] = 42` nur durch den **Kontext** um `var[1]` bzw. `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>]`-Klammern **hinter** die Variable zu schreiben ist in der **markierten Zeile** in Code 1.1 zu sehen. Will man auf ein **Element dieses Datentyps** zugreifen `(*complex_var[0][1])[1].attr`, so ist der Ausdruck fast genau **gleich aufgebaut**, wie der Ausdruck für den **Datentyp** `struct st (*complex_var[1][2])[2]`. Die **Ausgabe** des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

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

Code 1.1: *Beispiel für 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** `<var>[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 tätsächlich nur in der **Untermenge** von L_C, die L_{PicoC} darstellt gültig ist[18], 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

[18] In 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 **Zusammengesetzter Datentyp** ist.

ein **Verbund** `struct st copyable_ar = {.ar={314, 314}};`[19] 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 **aufrufenden** Funktion. Der Eintrag an Index 1 im Feld bleibt bei `314`.

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]

[a] Bast, „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 **Funktionsuaufruf** 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.

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

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]

[a] Bast, „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]

[a] Der **Funktionsprototyp** ist von der **Funktionsignatur** zu unterschieden, die in Programmiersprache wie `C++` und `Java` für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Metho-**

den oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentpyen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Sichtbarkeitsbereiche** und **Klassentpyen** usw. unterschieden.
[b] *What 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]

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

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]

[a] Im Fall des PicoC-Compilers in den **Globalen Statischen Daten** oder auf dem **Stack**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] 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 herbei 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]

[a] Thiemann, „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 Maschinensprache 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**[20] ohne merklichen Unterschied[21] komplieren können.

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

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

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

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.

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**[22], nämlich das **Lark Parsing Toolkit**[23] 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[24] zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter **„Getting Started“** ein kleines **Einführungstutorial** verlinkt. Unter **„Usage“** ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter Link[25] zu finden.

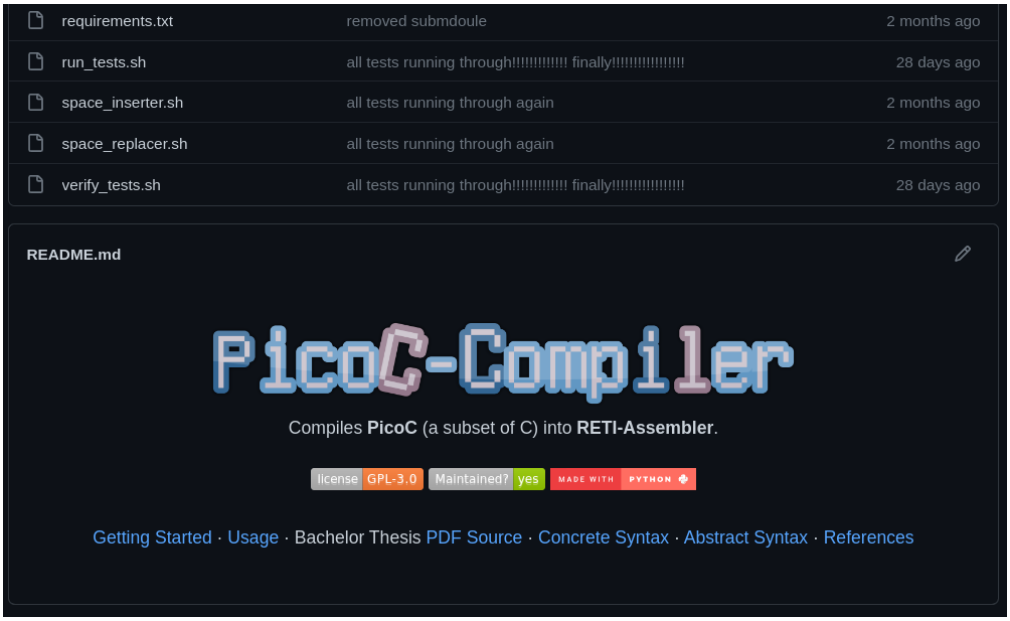


Abbildung 1.4: *README.md im Github Repository der Bachelorarbeit.*

Die **Schrifftliche 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 **Schrifftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter Link[26] zu finden. Die **PDF** der Schrifftliche Ausarbeitung der Bachleorararbeit wird aus dem **Latexquellcode**, welcher unter

[22] *Welcome to Lark's documentation! — Lark documentation*.
[23] *Lark - a parsing toolkit for Python*.
[24] `https://github.com/matthejue/PicoC-Compiler`.
[25] `https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971`.
[26] `https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf`.

Link[27] veröffentlicht ist automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[28] zu finden[29]. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**[30] erstellt. Falls Interesse besteht **Grafiken** aus der Schrifftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den `.svg`-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[31] zu finden.

Um die verschiedenen **Aspekte** dieser Schrifftlichen 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 **eingelesen** wurden. In Kapitel **??** werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** in Form von Codebeispielen gezeigt[32].

Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[33] finden. Mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` lassen sich die Codebeispiele genauso **kompilieren**, wie sie hier dargestellt sind[34].

1.5.1 Still der Schrifftlichen Ausarbeitung

In dieser **Schrifftliche 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 geanu 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 braucht[35] 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**[36] 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 übernomnen, so wurde die Definition oder der entsprechende Teil in **„Anführungszeichen“** gesetzt. Beim Verweis auf Quellen **außerhalb** einer

[27] https://github.com/matthejue/Bachelorarbeit.
[28] https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.
[29] Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.
[30] Developers, *Draw Freely — Inkscape*.
[31] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.
[32] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[33] https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.
[34] Es wurde zu diesem Zweck die **Command-line Option** `-t`, `--thesis` erstellt, die bestimmte Kommentare **herausfiltert**, damit die generierten **Abstrakten Syntaxbäume** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu **überfüllt** mit Kommentaren sind.
[35] Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[36] Das ist ein **Beispiel** für eine **Fußnote**.

Definitionsbox, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

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

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

Es wurde immer versucht möglichst **deutsche Fachbegriffe** zu verwenden, sofern sie einigermaßen **geläufig** sind und bei der Verwendung **nicht** eher **verwirren**[37]. Bei **Code** und anderem **Text**, dessen Zweck **nicht** dem Erklären dient, sondern der Veranschaulichung, wurde dieser konsequent in **Englisch** geschrieben bzw. belassen. Der Grund hierfür ist unter anderem, da die **Bezeichner** in der **Implementierung** des PicoC-Compilers, wie es mehr oder weniger **Konvention** beim Programmieren ist in **Englisch** benannt sind und diese Bezeichner in den **Ausgaben** des **PicoC-Compilers** vorkommen[38].

1.5.2 Aufbau der Schrifftlichen Arbeit

Der Inhalt dieser **Schrifftlichen 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** relavant 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 Schrifftlichen 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 dennen das Kapitel **Einführung** ebenfalls unterteilt ist

[37] 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.

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

erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähliche Kapiteleinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständis eines Aspekts in Kapitel **Implemenentierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteleinteilung** 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 **Unterkapitel**, die bei **Interesse** zur **weiteren Vertiefung** da sind und zum Verständis der anderen Kapitel **nicht notwendig** sind. Damit der **Rote Faden** in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit erkennbar bleibt und der **Lesefluss** nicht gestört wird, wurden alle diese Informationen in den **Appendix** ausgelaggert.

Die **Sonstigen Defintionen** und das Thema **Bootstrapping** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Diese **Themen** und **Definitionen** passen nicht ins Kapitel **??**, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tuen haben und auch nichts ins **Kapitel ??**, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in dieser Schrifftlichen Ausarbeitung immer versucht **Parallelen** zur Implementierung **echter** Compiler zu ziehen. Die **Erklärungen** und **Definitionen** hierfür wurden allerdings in den **?? ausgelaggert**. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition **??**) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung C. Scholl, „Betriebssysteme“ gehalten werden sollte.

1.5.3 Umsetzung von Verbunden

Bei Verbunden wird in diesem Unterkapitel zunächst geklärt, wie die **Deklaration von Verbundstypen** umgesetzt ist. Ist ein **Verbundstyp** deklariert, kann damit einhergehend ein **Verbund** mit diesem Verbundstyp **definiert** werden. Die Umsetzung von beidem wird in Unterkapitel 1.5.3.1 erläutert. Des Weiteren ist die Umsetzung der **Innitialisierung eines Verbundes** 1.5.3.2, des **Zugriffs auf ein Verbundsattribut** 1.5.3.3, der **Zuweisung an ein Verbundsattribut** 1.5.3.4 zu klären.

1.5.3.1 Deklaration von Verbundstypen und Definition von Verbunden

Die **Deklaration** (Definition 1.7) eines neuen **Verbundstyps** (z.B. `struct st {int len; int ar[2];}`) und die **Definition** (Definition 1.8) eines **Verbundes** mit diesem **Verbundstyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 1.15 erläutert.

```
1 struct st {int len; int ar[2];};
2
3 void main() {
4   struct st st_var;
5 }
```

Code 1.15: *PicoC-Code für die Deklaration eines Verbundstyps.*

Bevor irgendwas definiert werden kann, muss erstmal ein **Verbundstyp** deklariert werden. Im **Abstrakten Syntaxbaum** in Code 1.17 wird die **Deklaration eines Verbundstyps** `struct st {int len; int ar[2];}` durch die Komposition `StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` dargestellt.

Die **Definition** einer Variable mit diesem **Verbundstyp** `struct st st_var;` wird durch die Komposition `Alloc(Writeable(), StructSpec(Name('st')), Name('st_var'))` dargestellt.

```
1  File
2    Name './example_struct_decl_def.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), IntType('int'), Name('len'))
8          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')))
16       ]
17   ]
```

Code 1.16: *Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.*

Für den **Verbundstyp** selbst wird in der **Symboltabelle**, die in Code 1.17 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Attribute dieses Symbols `type_qualifier`, `datatype`, `name`, `position` und `size` sind

wie üblich belegt, allerdings sind in dem `value_address`-Attribut des Symbols die Attribute des **Verbundstyps** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Verbundstyp** `st` die **Attribute** des Verbundstyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Sichtbarkeitsbereich** innerhalb des **Verbundtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Verbundstyps** zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Verbundstypen** ihre Attribute gleich benennen können.

Jedes der **Attribute** `[Name('len@st'), Name('ar@st')]` erhält auch einen eigenen Eintrag in der **Symboltabelle**, wobei die Attribute `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt werden. Die Attribute `type_qualifier`, `datatype` und `name` werden z.B. bei `Name('ar@st')` mithilfe der Attribute von `Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` belegt.

Für die **Definition** einer Variable `st_var@main` mit diesem **Verbundstyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatyp`-Attribut enthält dabei den Namen des **Verbundstyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Verbundstyp**[39] und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Anmerkung

Die **Größe** einer Variable `st_var`, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist und mit dem **Verbundstyp** `struct st` {$\texttt{datatype}_1$ $\texttt{attr}_1$; ... $\texttt{datatype}_n$ $\texttt{attr}_n$;};[a] definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** $\texttt{datatype}_1$ ... $\texttt{datatpye}_n$ der **Attribute** $\texttt{attr}_1$, ... $\texttt{attr}_n$ des **Verbundstyps**: $\texttt{size}(\texttt{st}) = \sum_{i=1}^{n} \texttt{size}(\texttt{datatype}_i)$.

[a] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die Fragwürdige Designentscheidung, auch die eckigen Klammern `[]` für die Definition eines Feldes **vor** die Variable zu schreiben von Lc übernommen. Es wird so getann, als würde der komplette **Datentyp** immer **hinter** der Variable stehen: `datatype var`.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                IntType('int')
7          name:                    Name('len@st')
8          value or address:        Empty()
9          position:                Pos(Num('1'), Num('15'))
10         size:                    Num('1')
11       },
12     Symbol
13       {
14         type qualifier:          Empty()
15         datatype:                ArrayDecl([Num('2')], IntType('int'))
16         name:                    Name('ar@st')
17         value or address:        Empty()
18         position:                Pos(Num('1'), Num('24'))
19         size:                    Num('2')
20       },
21     Symbol
22       {
23         type qualifier:          Empty()
```

[39] Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Verbundstyp** einnimmt.

```
24        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
          ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
          ↪  Name('ar'))])
25        name:                   Name('st')
26        value or address:       [Name('len@st'), Name('ar@st')]
27        position:               Pos(Num('1'), Num('7'))
28        size:                   Num('3')
29      },
30    Symbol
31      {
32        type qualifier:         Empty()
33        datatype:               FunDecl(VoidType('void'), Name('main'), [])
34        name:                   Name('main')
35        value or address:       Empty()
36        position:               Pos(Num('3'), Num('5'))
37        size:                   Empty()
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               StructSpec(Name('st'))
43        name:                   Name('st_var@main')
44        value or address:       Num('0')
45        position:               Pos(Num('4'), Num('12'))
46        size:                   Num('3')
47      }
48  ]
```

Code 1.17: *Symboltabelle für die Deklaration eines Verbundstyps.*

1.5.3.2 Initialisierung von Verbunden

Die **Initialisierung eines Verbundes** wird im Folgenden mithilfe des Beispiels in Code 1.18 erklärt.

```
1 struct st1 {int *attr[2];};
2
3 struct st2 {int attr1; struct st1 attr2;};
4
5 void main() {
6   int var = 42;
7   struct st2 st = {.attr1=var, .attr2={.attr={&var, &var}}};
8 }
```

Code 1.18: *PicoC-Code für Initialisierung von Verbunden.*

Im **Abstrakten Syntaxbaum** in Code 1.19 wird die **Initialisierung eines Verbundes** `struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}}` mithilfe der **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` dargestellt.

```
1 File
2   Name './example_struct_init.ast',
```

```
3  [
4    StructDecl
5      Name 'st1',
6      [
7        Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
         ↪  Name('attr'))
8      ],
9    StructDecl
10     Name 'st2',
11     [
12       Alloc(Writeable(), IntType('int'), Name('attr1'))
13       Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
14     ],
15   FunDef
16     VoidType 'void',
17     Name 'main',
18     [],
19     [
20       Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
21       Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
         ↪  Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
         ↪  Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))]))]))]))
22     ]
23 ]
```

Code 1.19: *Abstrakter Syntaxbaum für Initialisierung von Verbunden.*

Im **PicoC-ANF Pass** in Code 1.20 wird die **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Feldes** in Subkapitel **??** daher wird um keine Wiederholung zu betreiben auf Subkapitel **??** verwiesen. Um das ganze interressanter zu gestalten wurde das Beispiel in Code 1.18 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der **Verbund-Initializer** Teilbaum `Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))]))])`, der beim **Verbund-Initializer** Knoten anfängt, wird auf dieselbe Weise nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Feldes** in Subkapitel **??** bereits erklärt wurde.

Beim **Iterieren** über den **Teilbaum**, muss beim **Verbund-Initializer** nur beachtet werden, dass bei den `Assign(lhs, exp)`-Knoten, über welche die **Attributzuweisung** dargestellt wird (z.B. `Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))]))`) der Teilbaum beim rechten `exp` Attribut weitergeht.

Im Allgemeinen gibt es beim **Initialisieren** eines **Feldes** oder **Verbundes** im Teilbaum auf der **rechten Seite**, der beim jeweiligen obersten **Initializer** anfängt immer nur 3 Fällte, man hat es auf der **rechten** Seite entweder mit einem **Verbund-Initialiser**, einem **Feld-Initialiser** oder einem **Logischen Ausdruck** zu tuen. Bei **Feld-** und **Verbund-Initialisier** wird einfach über diese nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** iteriert und die Ergebnisse der **Logischen Ausdrücken** in den **Blättern** auf den **Stack** gespeichert. Der Fall, dass ein **Logischer Ausdruck** vorliegt erübrigt sich damit.

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
        ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
        ↪  Ref(Name('var'))]))]))]))
        Exp(Global(Num('0')))
        Ref(Global(Num('0')))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('3')))
        Return(Empty())
      ]
  ]
```

Code 1.20: *PicoC-ANF Pass für Initialisierung von Verbunden.*

Im **RETI-Blocks Pass** in Code 1.21 werden die **Kompositionen** `Exp(exp)`, `Ref(exp)` und `Assign(Global(Num('1')), Stack(Num('3')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_init.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
        ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
        ↪  Ref(Name('var'))]))]))]))
        # Exp(Global(Num('0')))
        SUBI SP 1;
        LOADIN DS ACC 0;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
```

```
28          LOADI IN1 0;
29          ADD IN1 DS;
30          STOREIN SP IN1 1;
31          # Assign(Global(Num('1')), Stack(Num('3')))
32          LOADIN SP ACC 1;
33          STOREIN DS ACC 3;
34          LOADIN SP ACC 2;
35          STOREIN DS ACC 2;
36          LOADIN SP ACC 3;
37          STOREIN DS ACC 1;
38          ADDI SP 3;
39          # Return(Empty())
40          LOADIN BAF PC -1;
41        ]
42    ]
```

Code 1.21: *RETI-Blocks Pass für Initialisierung von Verbunden.*

1.5.3.3 Zugriff auf Verbundsattribut

Der **Zugriff auf ein Verbundsattribut** (z.B. `st.y`) wird im Folgenden mithilfe des Beispiels in Code 1.22 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y;
6 }
```

Code 1.22: *PicoC-Code für Zugriff auf Verbundsattribut.*

Im **Abstrakten Syntaxbaum** in Code 1.23 wird der **Zugriff auf ein Verbundsattribut** `st.y` mithilfe der **Komposition** `Exp(Attr(Name('st'), Name('y')))` dargestellt.

```
1  File
2    Name './example_struct_attr_access.ast',
3    [
4      StructDecl
5        Name 'pos',
6        [
7          Alloc(Writeable(), IntType('int'), Name('x'))
8          Alloc(Writeable(), IntType('int'), Name('y'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
           ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
```

```
16          Exp(Attr(Name('st'), Name('y')))
17      ]
18  ]
```

Code 1.23: *Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 1.24 wird die Komposition `Exp(Attr(Name('st'), Name('y')))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Feldelement** `Exp(Subscr(Name('ar'), Num('0')))` in Subkapitel **??** darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel **??** verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Subkapitel **??** durch Kompositionen ersetzt, die sich in **Anfangsteil** 1.5.4.1, **Mittelteil** 1.5.4.2 und **Schlussteil** 1.5.4.3 aufteilen lassen. In diesem Fall sind es `Ref(Global(Num('0')))` (**Anfangsteil**), `Ref(Attr(Stack(Num('1')), Name('y')))` (**Mittelteil**) und `Exp(Stack(Num('1')))` (**Schlussteil**). Der **Anfangsteil** und **Schlussteil** sind genau gleich, wie in Subkapitel **??**.

Nur für den **Mittelteil** wird eine andere Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` gebraucht. Diese Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` erfüllt die Aufgabe die **Adresse**, ab der das **Attribut** auf das zugegriffen wird anfängt zu berechnen. Dabei wurde die **Anfangsadresse** des **Verbundes** indem dieses Attribut liegt bereits vorher auf den **Stack** gelegt.

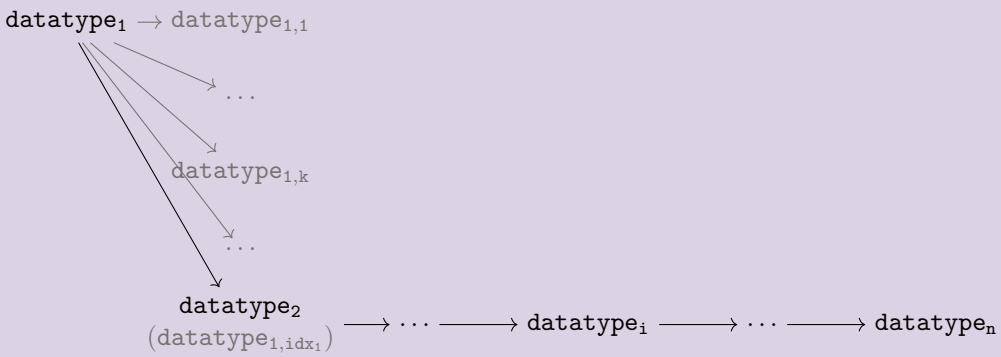
Im Gegensatz zur Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` beim **Zugriff auf einen Feldindex** in Subkapitel **??**, muss hier vorher nichts anderes als die **Anfangsadresse** des **Verbundes** auf dem **Stack** liegen. Das **Verbundsattribut** auf welches zugegriffen wird steht bereits in der Komposition `Ref(Attr(Stack(Num('1')), Name('y')))`, nämlich `Name('y')`. Den **Verbundstyp**, dem dieses Attribut gehört, kann man aus dem versteckten Attribut `datatype` herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-ANF Pass** dem Knoten `Ref(exp, datatype)` angehängt.

Anmerkung

Sei $\texttt{datatype}_\texttt{i}$ ein **Knoten** eines **entarteten Baumes** (siehe Definition 1.10 und Abbildung 1.5.1), dessen Wurzel $\texttt{datatype}_\texttt{1}$ ist. Dabei steht `i` für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\texttt{ref}(\texttt{datatype}_\texttt{i})$ von Speicherbereichen $\texttt{ref}(\texttt{datatype}_\texttt{i})$ ... $\texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$ im **Hauptspeicher** zuordnen, wobei gilt, dass $\texttt{ref}(\texttt{datatype}_\texttt{i}) \leq \texttt{ref}(\texttt{datatype}_{\texttt{i+1}}) < \texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$.[ab]

Sei $\texttt{datatype}_{\texttt{i,k}}$ ein beliebiges **Element / Attribut** des **Datentyps** $\texttt{datatype}_\texttt{i}$. Dabei gilt: $\texttt{ref}(\texttt{datatype}_{\texttt{i,k}}) < \texttt{ref}(\texttt{datatype}_{\texttt{i,k+1}})$.

Sei $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ ein beliebiges **Element / Attribut** des **Datentyps** $\texttt{datatype}_\texttt{i}$, sodass gilt: $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}} = \texttt{datatype}_{\texttt{i+1}}$.



(1.5.1)

Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen ($\texttt{datatype}_{1,\texttt{idx}_1}, \ldots, \texttt{datatype}_{n,\texttt{idx}_n}$), die das Resultat einer Aneinandereihung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente** und **Verbundsattributte** unterschiedlicher Datentypen $\texttt{datatype}_i$ ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 1.5.2:

$$\texttt{ref}(\texttt{datatype}_{1,\texttt{idx}_1}, \ldots, \texttt{datatype}_{n,\texttt{idx}_n}) = \texttt{ref}(\texttt{datatype}_1) + \sum_{i=1}^{n-1} \sum_{k=1}^{idx_i - 1} \text{size}(\texttt{datatype}_{i,k}) \quad (1.5.2)$$

berechnet werden.[c]

Dabei darf nur der letzte Knoten $\texttt{datatype}_n$ vom Datentyp **Zeiger** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Zeiger**, der nicht der **letzte Datentyp** $\texttt{datatype}_n$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp $\texttt{datatype}_1$ bis direkt zum Dantentyp Zeiger geht $\texttt{datatype}_{\texttt{pntr}}$ und die **zweite Adressberechnung** einen Dantentyp nach dem Datentyp Zeiger anfängt $\texttt{datatpye}_{\texttt{pntr}+1}$ und bis zum letzten Datenyp $\texttt{datatype}_n$ geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\texttt{ref}(\texttt{datatype}_1)$ des Summanden aus der Formel 1.5.2 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $\texttt{M}\left[\texttt{ref}(\texttt{datatype}_1, \ldots, \texttt{datatype}_{\texttt{pntr}})\right]$ gesetzt werden.

Die Formel 1.5.2 stellt dabei eine **Verallgemeinerung** der Formel **??** dar, die für alle möglichen Aneinandereihungen von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** funktioniert (z.B. `(*complex_var.attr2)[3]`). Da die Formel **allgemein** sein muss, lässt sie sich nicht so elegant mit einem Produkt $\prod$ schreiben, wie die Formel **??**, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben[d].

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden $\texttt{ref}(\texttt{datatype}_1)$ in der Formel.

Die Komposition `Exp(Attr(Stack(Num('1')), name))` repräsentiert dabei einen Summanden $\sum_{k=1}^{idx_i - 1} \text{size}(\texttt{datatype}_{i,k})$ in der Formel.

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** $\texttt{M}\left[\texttt{ref}(\texttt{datatype}_{1,\texttt{idx}_1}, \ldots, \texttt{datatype}_{n,\texttt{idx}_n})\right]$ der Speicherzelle an der finalen **Adresse** $\texttt{ref}(\texttt{datatype}_{1,\texttt{idx}_1}, \ldots, \texttt{datatype}_{n,\texttt{idx}_n})$.

[a] Es ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.
[b] `ref(datatype)` steht dabei für das Schreiben der **Startadresse**, die dem **Datentyp** `datatype` zugeordnet ist auf den **Stack**.

[c]Die **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Zeigerelmente**, **Feldelemente** oder **Verbundsattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** $\texttt{datatype}_i$, die vor dem **Element** / **Attribut** $\texttt{datatype}_{i,\texttt{idx}_i}$ liegen.
[d]Verbundsattribute haben **unterschiedliche** Größen.

Definition 1.10: Entarteter Baum

*Baum bei dem jeder Knoten **maximal** eine ausgehende Kante hat, also maximal **Außengrad** 1.*

Oder alternativ: *Baum beim dem jeder Knoten des Baumes **maximal** eine eingehende Kante hat, also maximal **Innengrad**[a] 1.*

*Der Baum entspricht also einer **verketteten Liste**.*[b]

[a]Der **Innengrad** ist die Anzahl **eingehener Kanten**.
[b]*Bäume*.

```
File
  Name './example_struct_attr_access.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Exp(Attr(Name('st'), Name('y')))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 1.24: *PicoC-ANF Pass für Zugriff auf Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 1.25 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')), Name('y')))` und `Exp(Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_attr_access.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
```

```
10          LOADI ACC 4;
11          STOREIN SP ACC 1;
12          # Exp(Num('2'))
13          SUBI SP 1;
14          LOADI ACC 2;
15          STOREIN SP ACC 1;
16          # Assign(Global(Num('0')), Stack(Num('2')))
17          LOADIN SP ACC 1;
18          STOREIN DS ACC 1;
19          LOADIN SP ACC 2;
20          STOREIN DS ACC 0;
21          ADDI SP 2;
22          # // Exp(Attr(Name('st'), Name('y')))
23          # Ref(Global(Num('0')))
24          SUBI SP 1;
25          LOADI IN1 0;
26          ADD IN1 DS;
27          STOREIN SP IN1 1;
28          # Ref(Attr(Stack(Num('1')), Name('y')))
29          LOADIN SP IN1 1;
30          ADDI IN1 1;
31          STOREIN SP IN1 1;
32          # Exp(Stack(Num('1')))
33          LOADIN SP IN1 1;
34          LOADIN IN1 ACC 0;
35          STOREIN SP ACC 1;
36          # Return(Empty())
37          LOADIN BAF PC -1;
38      ]
39  ]
```

Code 1.25: *RETI-Blocks Pass für Zugriff auf Verbundsattribut.*

1.5.3.4 Zuweisung an Verbundsattribut

Die **Zuweisung an ein Verbundsattribut** (z.B. `st.y = 42`) wird im Folgenden anhand des Beispiels in Code 1.26 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y = 42;
6 }
```

Code 1.26: *PicoC-Code für Zuweisung an Verbundsattribut.*

Im **Abstact Syntax Tree** wird eine **Zuweisung an ein Verbundsattribut** (z.B. `st.y = 42`) durch die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` dargestellt.

```
File
  Name './example_struct_attr_assignment.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc(Writeable(), IntType('int'), Name('x'))
        Alloc(Writeable(), IntType('int'), Name('y'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
        ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
        Assign(Attr(Name('st'), Name('y')), Num('42'))
      ]
  ]
```

Code 1.27: *Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 1.28 wird die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Feldelement** `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` in Subkapitel **??** darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel **??** verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel **??** muss hier für das Auswerten des **linken** Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Subkapitel 1.5.3.3 vorgegangen werden.

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Assign(Attr(Name('st'), Name('y')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 1.28: *PicoC-ANF Pass für Zuweisung an Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 1.29 werden die **Kompositionen** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')), Name('y')))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_attr_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # // Assign(Attr(Name('st'), Name('y')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Attr(Stack(Num('1')), Name('y')))
        LOADIN SP IN1 1;
        ADDI IN1 1;
        STOREIN SP IN1 1;
        # Assign(Stack(Num('1')), Stack(Num('2')))
        LOADIN SP IN1 1;
        LOADIN SP ACC 2;
        ADDI SP 2;
        STOREIN IN1 ACC 0;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.29: *RETI-Blocks Pass für Zuweisung an Verbndsattribut.*

1.5.4 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

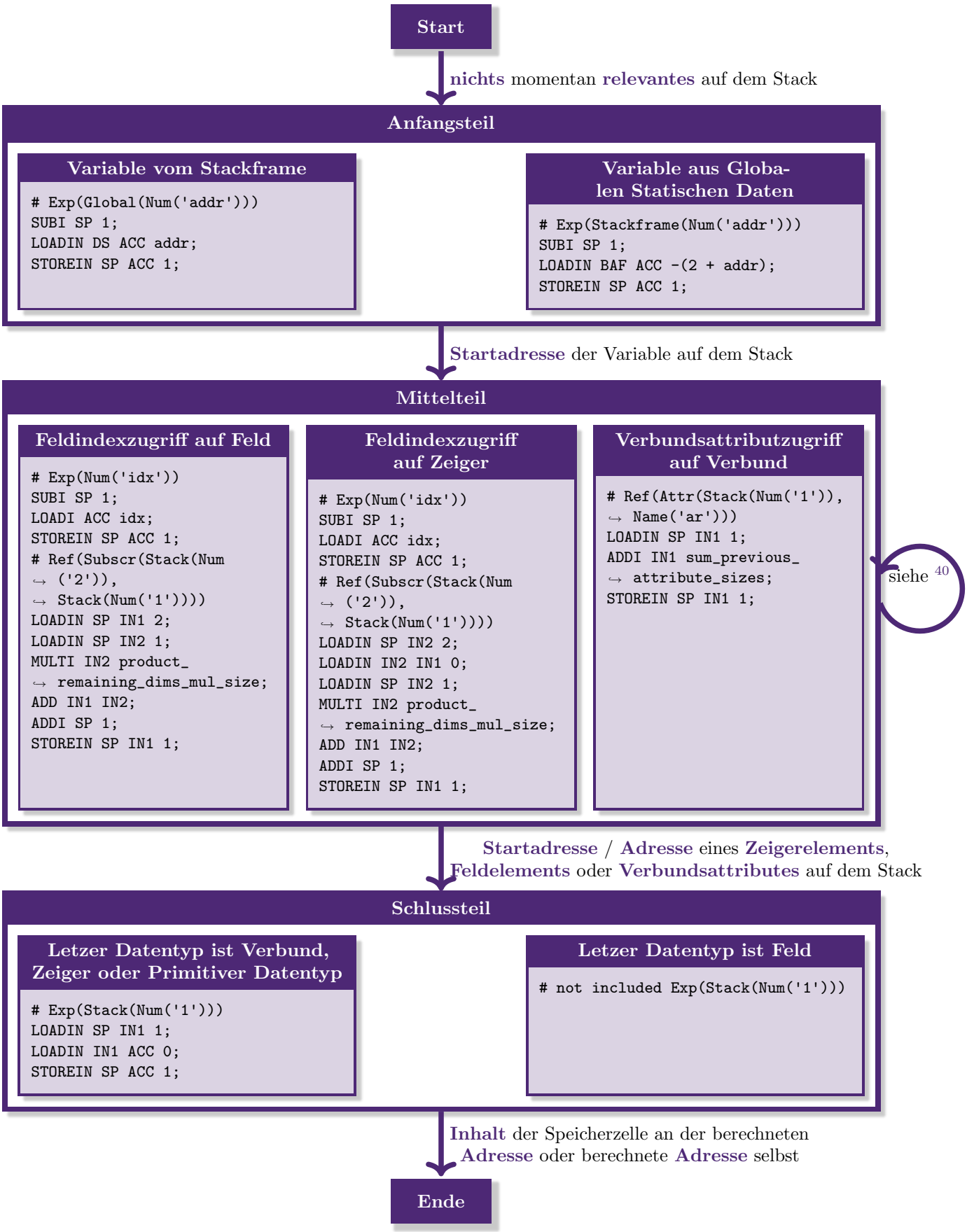


Abbildung 1.5: *Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.*

In den Unterkapiteln **??**, **??** und 1.5.3 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln beschriebenen Datentypen (**Zeiger**, **Feld** und **Verbund**) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem **Anfangsteil**, **Mittelteil** und **Schlussteil** darin erkennen.

Dieses Vorgehen ist in Abbildung 1.5 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der Datenypen **Zeiger**, **Feld** und **Verbund** gemischt sind (z.B. `(*st_var.ar)[0]`).

Dies ist möglich, indem im **Mittelteil**, je nachdem, ob das versteckte Attribut `datatype` des `Ref(exp, datatype)`-Knotens ein `ArrayDecl(nums, datatype)`, ein `PntrDecl(num, datatype)` oder `StructSpec(name)` beinhaltet und die dazu passende **Zugriffsoperation** `Subscr(exp1, exp2)` oder `Attr(exp, name)` vorliegt, einen anderen **RETI-Code** generiert wird. Dieser **RETI-Code** berechet die **Startadresse** eines gewünschten **Zeigerelements**, **Feldelements** oder **Verbundsattributs**.

Würde man bei einem `Subscr(Name('var'), exp2)` den Datentyp der Variable `Name('var')` von `ArrayDecl(nums, IntType())` zu `PointerDecl(num, IntType())` ändern, müsste nur der **Mittelteil** ausgetauscht werden. **Anfangsteil** und **Schlussteil** bleiben unverändert.

Die **Zugriffsoperation** muss dabei zum **Datentyp** im versteckten Attribut `datatype` passen, ansonsten gibt es eine `DatatypeMismatch`-**Fehlermeldung**. Ein **Zugriff auf ein Feldindex** `Subscr(exp1, epp2)` kann dabei mit den Datentypen **Feld** `ArrayDecl(nums, datatype)` und **Zeiger** `PntrDecl(num, datatype)` kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Zeiger** `PntrDecl(num, datatype)` die **Adresse**, die auf dem **Stack** liegt auf eine Speicherzelle mit einer weiteren **Adresse** zeigt und das gewünschte Element erst zu finden ist, wenn man der letzteren **Adresse** folgt. Ein **Zugriff auf ein Verbundsattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.

Anmerkung

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine **Dereferenzierung** in der Form `Deref(exp1, exp2)` nicht mehr existiert, denn wie in Unterkapitel **??** bereits erklärt wurde, wurde der Knoten `Deref(exp1, exp2)` im **PicoC-Shrink Pass** durch `Subscr(exp1, exp2)` ersetzt. Das hatte den Zweck, **doppelten Code** zu vermeiden, da die **Dereferenzierung** und der **Zugriff auf ein Feldelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Feldindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

Das versteckte Attribut `datatype` beinhaltet den **Unterdatentyp**, in welchem der Zugriff auf ein **Zeigerelment**, **Feldelement** oder **Verbundsattribut** erfolgt. Der **Unterdatentyp** ist dabei ein **Teilbaum** des Baumes, der vom gesamten **Datentyp** der **Variable** gebildet wird. Wobei man sich allerdings nur für den obersten **Knoten** in diesem **Unterdatentyp** interessiert und die möglicherweise unter diesem momentan betrachteten **Knoten** liegenden Knoten in einem anderen `Ref(exp, versteckte Attribut)`-Knoten, dem jeweiligen versteckten Attribut `datatype` zugeordnet sind. Das versteckte Attribut `datatype` enthält also die Information auf welchen **Unterdatentyp** im dem momentanen **Kontext** gerade zugegriffen wird.

Der **Anfangsteil**, der durch die Komposition `Ref(Name('var'))` repräsentiert wird, ist dafür zuständig die **Startadresse** der Variablen `Name('var')` auf den Stack zu schreiben und je nachdem, ob diese Variable in den **Globalen Statischen Daten** oder auf dem **Stackframe** liegt einen anderen **RETI-Code** zu generieren.

Der **Schlussteil** wird durch die Komposition `Exp(Stack(Num('1')), datatype)` dargestellt. Je nachdem, ob das versteckte Attribut `datatype` ein `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructType(name)` ist, wird ein entsprechender **RETI-Code** generiert, der die **Adresse**, die auf dem **Stack** liegt dazu nutzt, um den **Inhalt** der Speicherzelle an dieser **Adresse** auf den **Stack** zu schreiben. Dabei wird die Speicherzelle der **Adresse** mit dem **Inhalt** auf den sie selbst zeigt überschreiben. Bei einem `ArrayDecl(nums, datatype)` hingegen wird kein weiterer **RETI-Code** generiert, die **Adresse**, die auf dem **Stack** liegt, stellt bereits das

gewünschte Ergebnis dar.

Felder haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes **Feld** zugegriffen wird[41], die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache L_{PicoC} implementierten Datentypen wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

Anmerkung

Implementieren lässt sich dieses Vorgehen, indem beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)` Ausdrucks ein `Exp(Stack(Num('1')))` an die Spitze einer **Liste der generierten Ausdrücke** gesetzt wird und der Ausdruck selbst als `exp`-Attribut des `Ref(exp)`-Knotens gesetzt wird und hinter dem `Exp(Stack(Num('1')))`-Knoten in der Liste eingefügt wird. Beim Antreffen eines `Ref(exp)` wird fast gleich vorgegangen, wie beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)`, nur, dass kein `Exp(Stack(Num('1')))` vorne an die Spitze der **Liste der generierten Ausdrücke** gesetzt wird. Und ein `Ref(exp)` bei dem `exp` direkt ein `Name(str)` ist, wird dieser einfach direkt durch `Ref(Global(num))` bzw. `Ref(Stackframe(num))` ersetzt.

Es wird solange dem jeweiligen `exp1` des `Subscr(exp1, exp2)`-Knoten, dem `exp` des `Attr(exp, name)`-Knoten oder dem `exp` des `Ref(exp)`-Knoten gefolgt und der jeweilige Knoten selbst als `exp` des `Ref(exp)`-Knoten eingesetzt und hinten in die **Liste der generierten Ausdrücke** eingefügt, bis man bei einem `Name(name)` ankommt. Der `Name(name)`-Knoten wird zu einem `Ref(Global(num))` oder `Ref(Stackframe(num))` umgewandelt und ebenfalls ganz hinten in die **Liste der generierten Ausdrücke** eingefügt. Wenn man dem `exp` Attribut eines `Ref(exp)`-Knoten folgt, wird allerdings kein `Ref(exp)` in die **Liste der generierten Ausdrücke** eingefügt, sondern das `datatype`-Attribut des zuletzt eingefügten `Ref(exp, datatype)` manipuliert, sodass dessen `datatype` in ein `ArrayDecl([Num('1')], datatype)` eingebettet ist und so ein auf das `Ref(exp)` folgendes `Deref(exp1, exp2)` oder `Subscr(exp1, exp2)` direkt behandelt wird.

Parallel wird eine Liste der `Ref(exp)`-Knoten geführt, deren versteckte Attribute `datatype` und `error_data` die entsprechenden Informationen zugewiesen bekommen müssen. Sobald man beim `Name(name)`-Knoten angekommen ist und mithilfe dieses in der **Symboltabelle** den **Dantentyp** der Variable nachsehen kann, wird der **Datentyp** der Variable nun ebenfalls, wie die Ausdrücke `Subscr(exp1, exp2)` und `Attr(exp, name)` schrittweise durchiteriert und dem jeweils nächsten `datatype`-Attribut gefolgt werden. Das **Iterieren** über den **Datentyp** wird solange durchgeführt, bis alle `Ref(exp)`-Knoten ihren im jeweiligen **Kontext** vorliegenden **Datentyp** in ihrem `datatype`-Attribut zugewiesen bekommen haben. Alles andere führt zu einer **Fehlermeldung**, für die das versteckte Attribut `error_data` genutzt wird.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil** 1.5.4.1, **Mittelteil** 1.5.4.2 und **Schlussteil** 1.5.4.3 bei der Kompilierung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente**, **Verbundsattribute** bei gemischten Ausdrücken, wie `(*st_first.ar)[0];` einzeln isoliert betrachtet und erläutert.

1.5.4.1 Anfangsteil

Der **Anfangsteil**, bei dem die **Adresse** einer Variable auf den **Stack** geschrieben wird (z.B. `&st`), wird im Folgenden mithilfe des Beispiels in Code 1.30 erklärt.

```
1 struct ar_with_len {int len; int ar[2];};
2
3 void main() {
```

[41] Und nicht auf ein **Element** des Feldes.
[41] **Startadresse** / **Adresse** eines **Zeigerelements**, **Feldelements** oder **Verbundsattributes** auf dem Stack.

```
4   struct ar_with_len st_ar[3];
5   int *(*complex_var)[3];
6   &complex_var;
7 }
8
9 void fun() {
10  struct ar_with_len st_ar[3];
11  int (*complex_var)[3];
12  &complex_var;
13 }
```

Code 1.30: *PicoC-Code für den Anfangsteil.*

Im **Abstrakten Syntaxbaum** in Code 1.31 wird die **Refererenzierung** `&complex_var` mit der Komposition `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird aber einfach nur `Ref(Name('complex_var')` geschrieben, aber da beim Erstellen des **Abstract Syntx Tree** jeder **Logischer Ausdruck** in ein `Exp(exp)` eingebettet wird, ist das `Ref(Name('complex_var')` in ein `Exp()` eingebettet. Man müsste an vielen Stellen eine gesonderte **Fallunterschiedung** aufstellen, um von `Exp(Ref(Name('complex_var')))` das `Exp()` zu entfernen, obwohl das `Exp()` in den darauffolgenden **Passes** so oder so herausgefiltet wird. Daher wurde darauf verzichtet den Code ohne triftigen Grund **komplexer** zu machen.

```
1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     StructDecl
5       Name 'ar_with_len',
6       [
7         Alloc(Writeable(), IntType('int'), Name('len'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪ Name('st_ar')))
16        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
          ↪ IntType('int')))), Name('complex_var')))
17        Exp(Ref(Name('complex_var')))
18      ],
19    FunDef
20      VoidType 'void',
21      Name 'fun',
22      [],
23      [
24        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪ Name('st_ar')))
25        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪ Name('complex_var')))
26        Exp(Ref(Name('complex_var')))
27      ]
28  ]
```

Code 1.31: *Abstrakter Syntaxbaum für den Anfangsteil.*

Im **PicoC-ANF Pass** in Code 1.32 wird die Komposition `Exp(Ref(Name('complex_var')))` durch die Komposition `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` ersetzt, je nachdem, ob die Variable `Name('complex_var')` in den **Globalen Statischen Daten** oder auf dem **Stack** liegt.

```
File
  Name './example_derived_dts_introduction_part.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Exp(Ref(Name('complex_var')))
        Ref(Global(Num('9')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Exp(Ref(Name('complex_var')))
        Ref(Stackframe(Num('9')))
        Return(Empty())
      ]
  ]
```

Code 1.32: *PicoC-ANF Pass für den Anfangsteil.*

Im **RETI-Blocks Pass** in Code 1.33 werden die Komposition `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_derived_dts_introduction_part.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Exp(Ref(Name('complex_var')))
        # Ref(Global(Num('9')))
        SUBI SP 1;
        LOADI IN1 9;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // Exp(Ref(Name('complex_var')))
        # Ref(Stackframe(Num('9')))
        SUBI SP 1;
```

```
22          MOVE BAF IN1;
23          SUBI IN1 11;
24          STOREIN SP IN1 1;
25          # Return(Empty())
26          LOADIN BAF PC -1;
27        ]
28    ]
```

Code 1.33: *RETI-Blocks Pass für den Anfangsteil.*

1.5.4.2 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinandereihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute** berechnet wird (z.B. `(*complex_var.ar)[2-2]`), wird im Folgenden mithilfe des Beispiels in Code 1.34 erklärt.

```
1 struct st {int (*ar)[1];};
2
3 void main() {
4   int var[1] = {42};
5   struct st complex_var = {.ar=&var};
6   (*complex_var.ar)[2-2];
7 }
```

Code 1.34: *PicoC-Code für den Mittelteil.*

Im **Abstrakten Syntaxbaum** in Code 1.35 wird die Aneinandererihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** `(*complex_var.ar)[2-2]` durch die Komposition `Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2'))))` dargestellt.

```
1  File
2    Name './example_derived_dts_main_part.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
           ↪  Name('ar'))
8        ],
9      FunDef
10       VoidType 'void',
11       Name 'main',
12       [],
13       [
14         Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
           ↪  Array([Num('42')]))
15         Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),
           ↪  Struct([Assign(Name('ar'), Ref(Name('var')))]))
16         Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
           ↪  Sub('-'), Num('2'))))
```

```
17      ]
18  ]
```

Code 1.35: *Abstrakter Syntaxbaum für den Mittelteil.*

Im **PicoC-ANF Pass** in Code 1.36 wird die Komposition `Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2'))))` durch die Kompositionen `Ref(Attr(Stack(Num('1')), Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ersetzt. Bei `Subscr(exp1, exp2)` wird dieser Knoten einfach dem `exp` Attribut des `Ref(exp)`-Knoten zugewiesen und die **Indexberechnung** für `exp2` davorgezogen (in diesem Fall dargestellt durch `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`) und über `Stack(Num('1'))` auf das Ergebnis der **Indexberechnung** auf dem **Stack** zugegriffen: `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`.

```
1  File
2    Name './example_derived_dts_main_part.picoc_mon',
3    [
4      Block
5        Name 'main.0',
6        [
7          // Assign(Name('var'), Array([Num('42')]))
8          Exp(Num('42'))
9          Assign(Global(Num('0')), Stack(Num('1')))
10         // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11         Ref(Global(Num('0')))
12         Assign(Global(Num('1')), Stack(Num('1')))
13         // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
           ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
14         Ref(Global(Num('1')))
15         Ref(Attr(Stack(Num('1')), Name('ar')))
16         Exp(Num('0'))
17         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18         Exp(Num('2'))
19         Exp(Num('2'))
20         Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
21         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22         Exp(Stack(Num('1')))
23         Return(Empty())
24       ]
25   ]
```

Code 1.36: *PicoC-ANF Pass für den Mittelteil.*

Im **RETI-Blocks Pass** in Code 1.37 werden die Kompositionen `Ref(Attr(Stack(Num('1')), Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` durch ihre entsprechenden **RETI-Knoten** ersetzt. Bei der Generierung des **RETI-Code** muss auch das versteckte Attribut `datatype` im `Ref(exp, datatpye)`-Knoten berücksichtigt werden, was in Unterkapitel 1.5.4 zusammen mit der Abbildung 1.5 bereits erklärt wurde.

```
File
  Name './example_derived_dts_main_part.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
        ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
        # Ref(Global(Num('1')))
        SUBI SP 1;
        LOADI IN1 1;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Attr(Stack(Num('1')), Name('ar')))
        LOADIN SP IN1 1;
        ADDI IN1 0;
        STOREIN SP IN1 1;
        # Exp(Num('0'))
        SUBI SP 1;
        LOADI ACC 0;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        LOADIN SP IN2 2;
        LOADIN IN2 IN1 0;
        LOADIN SP IN2 1;
        MULTI IN2 1;
        ADD IN1 IN2;
        ADDI SP 1;
        STOREIN SP IN1 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
```

```
57          LOADIN SP ACC 2;
58          LOADIN SP IN2 1;
59          SUB ACC IN2;
60          STOREIN SP ACC 2;
61          ADDI SP 1;
62          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63          LOADIN SP IN1 2;
64          LOADIN SP IN2 1;
65          MULTI IN2 1;
66          ADD IN1 IN2;
67          ADDI SP 1;
68          STOREIN SP IN1 1;
69          # Exp(Stack(Num('1')))
70          LOADIN SP IN1 1;
71          LOADIN IN1 ACC 0;
72          STOREIN SP ACC 1;
73          # Return(Empty())
74          LOADIN BAF PC -1;
75        ]
76    ]
```

Code 1.37: *RETI-Blocks Pass für den Mittelteil.*

1.5.4.3 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 1.5.4.1 und **Mittelteil** 1.5.4.2 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird[42], wird im Folgenden mithilfe des Beispiels in Code 1.38 erklärt.

```
1  struct st {int attr[2];};
2
3  void main() {
4    int complex_var1[1][2];
5    struct st complex_var2[1];
6    int var = 42;
7    int *pntr1 = &var;
8    int **complex_var3 = &pntr1;
9
10   complex_var1[0];
11   complex_var2[0];
12   *complex_var3;
13 }
```

Code 1.38: *PicoC-Code für den Schlussteil.*

Das Generieren des **Abstrakten Syntaxbaumes** in Code 1.39 verläuft wie üblich.

```
1  File
2    Name './example_derived_dts_final_part.ast',
```

[42] Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

```
3  [
4    StructDecl
5      Name 'st',
6      [
7        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8      ],
9    FunDef
10     VoidType 'void',
11     Name 'main',
12     [],
13     [
14       Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
         ↪  Name('complex_var1')))
15       Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
         ↪  Name('complex_var2')))
16       Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
17       Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),
         ↪  Ref(Name('var')))
18       Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),
         ↪  Ref(Name('pntr1')))
19       Exp(Subscr(Name('complex_var1'), Num('0')))
20       Exp(Subscr(Name('complex_var2'), Num('0')))
21       Exp(Deref(Name('complex_var3'), Num('0')))
22     ]
23 ]
```

Code 1.39: *Abstrakter Syntaxbaum für den Schlussteil.*

Im **PicoC-ANF Pass** in Code 1.40 wird das eben angesprochene auf den **Stack** speichern des **Inhalts** der berechneten **Adresse** mit der Komposition `Exp(Stack(Num('1')))` dargestellt.

```
1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('4')), Stack(Num('1')))
10        // Assign(Name('pntr1'), Ref(Name('var')))
11        Ref(Global(Num('4')))
12        Assign(Global(Num('5')), Stack(Num('1')))
13        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14        Ref(Global(Num('5')))
15        Assign(Global(Num('6')), Stack(Num('1')))
16        // Exp(Subscr(Name('complex_var1'), Num('0')))
17        Ref(Global(Num('0')))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        // Exp(Subscr(Name('complex_var2'), Num('0')))
22        Ref(Global(Num('2')))
23        Exp(Num('0'))
```

```
24          Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25          Exp(Stack(Num('1')))
26          // Exp(Subscr(Name('complex_var3'), Num('0')))
27          Ref(Global(Num('6')))
28          Exp(Num('0'))
29          Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30          Exp(Stack(Num('1')))
31          Return(Empty())
32        ]
33    ]
```

Code 1.40: *PicoC-ANF Pass für den Schlussteil.*

Im **RETI-Blocks Pass** in Code 1.41 wird die Komposition `Exp(Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt, wenn das versteckte Attribut `datatype` im `Exp(exp,datatpye)`-Knoten kein **Feld** `ArrayDecl(nums, datatype)` enthält, ansonsten ist bei einem **Feld** die **Adresse** auf dem **Stack** bereits das gewünschte Ergebnis. Genaueres wurde in Unterkapitel 1.5.4 zusammen mit der Abbildung 1.5 bereits erklärt.

```
1  File
2    Name './example_derived_dts_final_part.reti_blocks',
3    [
4      Block
5        Name 'main.0',
6        [
7          # // Assign(Name('var'), Num('42'))
8          # Exp(Num('42'))
9          SUBI SP 1;
10         LOADI ACC 42;
11         STOREIN SP ACC 1;
12         # Assign(Global(Num('4')), Stack(Num('1')))
13         LOADIN SP ACC 1;
14         STOREIN DS ACC 4;
15         ADDI SP 1;
16         # // Assign(Name('pntr1'), Ref(Name('var')))
17         # Ref(Global(Num('4')))
18         SUBI SP 1;
19         LOADI IN1 4;
20         ADD IN1 DS;
21         STOREIN SP IN1 1;
22         # Assign(Global(Num('5')), Stack(Num('1')))
23         LOADIN SP ACC 1;
24         STOREIN DS ACC 5;
25         ADDI SP 1;
26         # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
27         # Ref(Global(Num('5')))
28         SUBI SP 1;
29         LOADI IN1 5;
30         ADD IN1 DS;
31         STOREIN SP IN1 1;
32         # Assign(Global(Num('6')), Stack(Num('1')))
33         LOADIN SP ACC 1;
34         STOREIN DS ACC 6;
35         ADDI SP 1;
```

```
36        # // Exp(Subscr(Name('complex_var1'), Num('0')))
37        # Ref(Global(Num('0')))
38        SUBI SP 1;
39        LOADI IN1 0;
40        ADD IN1 DS;
41        STOREIN SP IN1 1;
42        # Exp(Num('0'))
43        SUBI SP 1;
44        LOADI ACC 0;
45        STOREIN SP ACC 1;
46        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47        LOADIN SP IN1 2;
48        LOADIN SP IN2 1;
49        MULTI IN2 2;
50        ADD IN1 IN2;
51        ADDI SP 1;
52        STOREIN SP IN1 1;
53        # // not included Exp(Stack(Num('1')))
54        # // Exp(Subscr(Name('complex_var2'), Num('0')))
55        # Ref(Global(Num('2')))
56        SUBI SP 1;
57        LOADI IN1 2;
58        ADD IN1 DS;
59        STOREIN SP IN1 1;
60        # Exp(Num('0'))
61        SUBI SP 1;
62        LOADI ACC 0;
63        STOREIN SP ACC 1;
64        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65        LOADIN SP IN1 2;
66        LOADIN SP IN2 1;
67        MULTI IN2 2;
68        ADD IN1 IN2;
69        ADDI SP 1;
70        STOREIN SP IN1 1;
71        # Exp(Stack(Num('1')))
72        LOADIN SP IN1 1;
73        LOADIN IN1 ACC 0;
74        STOREIN SP ACC 1;
75        # // Exp(Subscr(Name('complex_var3'), Num('0')))
76        # Ref(Global(Num('6')))
77        SUBI SP 1;
78        LOADI IN1 6;
79        ADD IN1 DS;
80        STOREIN SP IN1 1;
81        # Exp(Num('0'))
82        SUBI SP 1;
83        LOADI ACC 0;
84        STOREIN SP ACC 1;
85        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86        LOADIN SP IN2 2;
87        LOADIN IN2 IN1 0;
88        LOADIN SP IN2 1;
89        MULTI IN2 1;
90        ADD IN1 IN2;
91        ADDI SP 1;
92        STOREIN SP IN1 1;
```

```
93          # Exp(Stack(Num('1')))
94          LOADIN SP IN1 1;
95          LOADIN IN1 ACC 0;
96          STOREIN SP ACC 1;
97          # Return(Empty())
98          LOADIN BAF PC -1;
99      ]
100 ]
```

Code 1.41: *RETI-Blocks Pass für den Schlussteil.*

1.5.5 Umsetzung von Funktionen

Um die **Umsetzung** von **Funktionen** zu verstehen, ist es erstmal wichtig zu verstehen, wie **Funktionen** später im **RETI-Code** aussehen (Unterkapitel 1.5.5.1), wie Funktionen **deklariert** (Definition 1.7) und **definiert** (Definition 1.8) werden können und hierbei **Sichtbarkeitsbereiche** (Definition 1.9) umgesetzt sind (Unterkapitel 1.5.5.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines **Funktionsaufrufes** erklärt werden (Unterkapitel 1.5.5.3). Beim Thema **Funktionsaufruf** muss im speziellen nochmal darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 1.5.5.3.1) umgesetzt sind und die **Übergabe** von **Zusammengesetzten Datentypen**, die mehr als eine Speicherzelle belegen, wie **Verbunden** (Unterkapitel 1.5.5.3.3) und **Feldern** (Unterkapitel 1.5.5.3.2) umgesetzt ist.

1.5.5.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 1.42 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 1.42: *PicoC-Code für 3 Funktionen.*

Im **Abstrakten Syntaxbaum** in Code 1.43 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 1.43: *Abstrakter Syntaxbaum für 3 Funktionen.*

Im **PicoC-Blocks Pass** in Code 1.44 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)`[43] 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>`[44].

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',
```

[43] Eine **Erklärung** dazu ist in Unterkapitel **??** zu finden.
[44] Der **Grund** dafür kann im Unterkapitel **??** nachgelesen werden.

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

Code 1.44: *PicoC-Blocks Pass für 3 Funktionen.*

Im **PicoC-ANF Pass** in Code 1.45 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 1.45: *PicoC-ANF Pass für 3 Funktionen.*

Nach dem **RETI Pass** in Code 1.46 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.

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.2')))
# // not included Exp(GoTo(Name('main.2')))
# // Block(Name('main.2'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun1.1'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun2.0'), [])
# // Return(Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Return(Stack(Num('1')))
LOADIN SP ACC 1;
ADDI SP 1;
LOADIN BAF PC -1;
```

Code 1.46: *RETI-Blocks Pass für 3 Funktionen.*

1.5.5.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 1.42 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 1.47 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.

```
void fun1() {
}

int fun2() {
  return 1;
}

void main() {
  return;
}
```

Code 1.47: *PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI-Blocks Pass** in Code 1.48 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```
File
  Name './verbose_3_funs_main.reti_blocks',
  [
```

```
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 1.48: *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** 1.50 in einen Sprung zur `main`-Funktion übersetzt wird.

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für Verwender[45] 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 1.49, 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',
```

[45] Also 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 1.49: *RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI Pass** in Code 1.50 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 1.50: *RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

1.5.5.2 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 Dekleration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 1.51 **markierten Zeile** `1` oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** `3-4`.

In dem Beispiel in Code 1.51 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdekleration**, 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 1.51: *PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 1.52 für das Beispiel in Code 1.51 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(Writeable(), 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` angehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert, sobald beim **top-down**[46] iterieren über den **Abstrakten Syntaxbaum** des aktuellen **Passes** eine neuer `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten betreten wird und über die Anweisungen im `stmts`-Attribut iteriert wird. Für das **Iterieren** über die Anweisungen dieses neuen Funktionsknotens

[46] D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

wird der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts_blocks)` angehängt, 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 einer Variable direkt ihr **Sichtbarkeitsbereich** in dem sie definiert wurde abgelesen werden kann, ist der **Suffix** ebenfalls im `Name(str)`-Knoten des `name`-Attributes eines **Symbols** der Symboltabelle angehängt. Dies ist in Code 1.52 **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 im 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(),
           ↪  IntType('int'), Name('var'))])
7          name:                   Name('fun2')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('4'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Empty()
15         datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
16         name:                   Name('fun1')
17         value or address:       Empty()
18         position:               Pos(Num('3'), Num('5'))
19         size:                   Empty()
20       },
21     Symbol
22       {
23         type qualifier:         Empty()
24         datatype:               FunDecl(VoidType('void'), Name('main'), [])
25         name:                   Name('main')
26         value or address:       Empty()
27         position:               Pos(Num('6'), Num('5'))
28         size:                   Empty()
29       },
30     Symbol
```

```
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 1.52: *Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

1.5.5.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 1.53 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. Zudem wurde, um die **Adressberechnung anschaulicher** zu machen als **Datentyp** für den **Parameter** `param` der Funktion `stack_fun` ein **Verbund** gewählt, der **mehrere Speicherzellen** im Hauptspeicher einnimmt. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 1.5.5.3.1 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 1.53: *PicoC-Code für Funktionsaufruf ohne Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 1.54 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```
File
  Name './example_fun_call_no_return_value.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
      ],
    FunDecl
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '2',
              Num '3'
            ],
            StructSpec
              Name 'st',
          Name 'param'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        Exp(Call(Name('stack_fun'), [Name('local_var')]))
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('param'))
      ],
      [
        Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
      ]
  ]
```

Code 1.54: *Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.*

Im **PicoC-ANF Pass** in Code 1.55 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 **aufrufenden 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 wird[47].

```
File
  Name './example_fun_call_no_return_value.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.0')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'stack_fun.0',
      [
        Return(Empty())
      ]
  ]
```

Code 1.55: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 1.56 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 **Kompostionen** `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`[48] mit dem Schlüssel `stack_fun`[49], der im Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist, nachgeschlagen.

```
File
  Name './example_fun_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
```

[47] Wobei hier „**reserviert**“ besser passen würde.
[48] Dieses **Assoziative Feld** wurde in Unterkapitel 1.5.5.1 eingeführt.
[49] Dem **Bezeichner der Funktion**.

```
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;
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 1.56: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 1.56 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 ist[50] 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`-Attributs des **Zielblocks** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`[51] 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** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)` berechnet. Es handelt sich bei bei `14` um eine **relative Adresse**, die **relativ** zum `CS`-Register[52] berechnet wird.

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

[50] Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.
[51] Welcher den **ersten Befehl** der gewünschten Funktion enthält.
[52] Welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

folgenden Formel:

$$adr_{danach} = \#Bef_{vor\,akt.\,Bl.} + idx + 4 \tag{1.5.3}$$

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 versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, 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 1.57 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

Die Berechnug der **Distanz** `<distanz>` für den Sprung `JUMP <distanz>` zum **ersten** Befehl eines im vorhergehenden **Pass existenten Blockes**[a] erfolgt dabei nach der folgenden Formel:

$$distanz = \begin{cases} \#Bef_{vor\,Zielbl.} - \#Bef_{vor\,akt.\,Bl.} - idx & \#Bef_{vor\,Zielbl.} != \#Bef_{vor\,akt.\,Bl.} \\ -idx & \#Bef_{vor\,Zielbl.} = \#Bef_{vor\,akt.\,Bl.} \end{cases} \tag{1.5.4}$$

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 1.5.3.

[a] Im **RETI-Pass** gibt es **keine** Blöcke mehr.

```
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 1.57: *RETI-Pass für Funktionsaufruf ohne Rückgabewert.*

1.5.5.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 1.58 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 1.58: *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 1.59 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')),
          ↪  Call(Name('fun_with_return_value'), []))
24        Exp(Call(Name('fun_no_return_value'), []))
25      ]
26  ]
```

Code 1.59: *Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.*

Im **PicoC-ANF Pass** in Code 1.60 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. Als letztes wird die **Rücksprungadresse** in das `PC`-Register geladen, die durch den `NewStackframe()`-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)`[53] definiert werden.

Dieser Trick mit dem Speichern des **Rückgabewertes** im `ACC`-Register ist notwendidg, 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

[53] Diese Komposition schreibt den **aktuellen Wert** des Registers `ACC` auf den **Stack**

Stack in die Speicherzelle der Variable `Name('var')` in den **Globalen Statischen Daten** 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, sie 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 hinzufügt wird in Form der `Komposition Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine `MissingReturn`-**Fehlermeldung** ausgelöst.

```
File
  Name './example_fun_call_with_return_value.picoc_mon',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        Exp(Num('21'))
        Exp(Num('2'))
        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
        Return(Stack(Num('1')))
      ],
    Block
      Name 'fun_no_return_value.1',
      [
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_with_return_value.2')))
        RemoveStackframe()
        Exp(ACC)
        Assign(Global(Num('0')), Stack(Num('1')))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_no_return_value.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.60: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 1.61 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.

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        # Exp(Num('21'))
        SUBI SP 1;
        LOADI ACC 21;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
        LOADIN SP ACC 2;
        LOADIN SP IN2 1;
        MULT ACC IN2;
        STOREIN SP ACC 2;
        ADDI SP 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_no_return_value.1',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 2;
        MOVE SP BAF;
        SUBI SP 2;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_with_return_value.2')))
        Exp(GoTo(Name('fun_with_return_value.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Exp(ACC)
        SUBI SP 1;
        STOREIN SP ACC 1;
        # 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 1.61: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.*

1.5.5.3.2 Umsetzung der Übergabe eines Feldes

Die Eigenheit, dass bei der **Übergabe** eines **Feldes** an eine andere Funktion, dieses als Zeiger übergeben wird, wurde bereits im Unterkapitel 1.3 erläutert. Die Umsetzung der **Übergabe** eines **Feldes** an eine andere Funktion wird im Folgenden mithilfe des Beispiels in Code 1.62 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 1.62: *PicoC-Code für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** muss 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 auf spezielle Weise vorgegangen werden. Dieser **oberste Knoten** des Teilbaums, der den Datentyp darstellt muss zu einem

Zeiger `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, muss an das `datatype`-Attribut des **Zeigers** `PntrDecl(num, datatype)` gehängt werden.

Diese **Umwandlung** des **Datentyps** kann in der **Symboltabelle** in Code 1.63 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.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
        ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('param'))])
        name:                   Name('fun_array_from_stackframe')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param@fun_array_from_stackframe')
        value or address:       Num('0')
        position:               Pos(Num('1'), Num('37'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
        ↪  [Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
        name:                   Name('fun_array_from_global_data')
        value or address:       Empty()
        position:               Pos(Num('4'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param@fun_array_from_global_data')
        value or address:       Num('0')
        position:               Pos(Num('4'), Num('36'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
        name:                   Name('local_var@fun_array_from_global_data')
        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:         Writeable()
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 1.63: *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 1.64 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 die **Variable** `local_var` aus der `main`-Funktion, da diese in den **Globalen Statischen Daten** liegt und die Komposition `Ref(Stackframe(Num('6')))` ist für die Variable `local_var` aus der Funktion `fun_array_from_global_data`, da diese auf dem **Stackframe** dieser Funktion liegt. 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      ],
```

```
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_array_from_global_data.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.64: *PicoC-ANF Pass für die Übergabe eines Feldes.*

Im **RETI-Blocks Pass** in Code 1.65 werden Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      Name 'fun_array_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_array_from_global_data.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Stackframe(Num('6')))
        SUBI SP 1;
        MOVE BAF IN1;
        SUBI IN1 8;
        STOREIN SP IN1 1;
        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 3;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
        Exp(GoTo(Name('fun_array_from_stackframe.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
```

```
    Block
      Name 'main.0',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 9;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_array_from_global_data.1')))
        Exp(GoTo(Name('fun_array_from_global_data.1')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.65: *RETI-Block Pass für die Übergabe eines Feldes.*

1.5.5.3.3 Umsetzung einer Übergabe eines Verbundes

Die Eigenheit, dass ein **Verbund** als Argument beim **Funktionsaufruf** einer anderen Funktion in den **Stackframe** der **aufgerufenen** Funktion **kopiert** wird, wurde bereits im Unterkapitel 1.3 erläutert. Die Umsetzung der **Übergabe** eines **Verbundes** wird im Folgenden mithilfe des Beispiels in Code 1.66 erklärt.

```
struct st {int attr1; int attr2[2];};


void fun_struct_from_stackframe(struct st param) {
}

void fun_struct_from_global_data(struct st param) {
  fun_struct_from_stackframe(param);
}


void main() {
  struct st local_var;
  fun_struct_from_global_data(local_var);
}
```

Code 1.66: *PicoC-Code für die Übergabe eines Verbundes.*

Im **PicoC-ANF Pass** in Code 1.67 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. die **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')), Global(Num('0')))` wird verwendet, da die Verbundsvariable `local_var` der `main`-Funktion in den **Globalen Statischen Daten** liegt und die Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` wird verwendet, da die Verbundsvariable `local_var` der Funktion `fun_struct_from_global_data` im **Stackframe** liegt.

```
File
  Name './example_fun_call_by_value_struct.picoc_mon',
  [
    Block
      Name 'fun_struct_from_stackframe.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun_struct_from_global_data.1',
      [
        StackMalloc(Num('2'))
        Assign(Stack(Num('3')), Stackframe(Num('2')))
        NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Assign(Stack(Num('3')), Global(Num('0')))
        NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_struct_from_global_data.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.67: *PicoC-ANF Pass für die Übergabe eines Verbundes.*

Im **RETI-Blocks Pass** in Code 1.68 werden die Kompositionen `Assign(Stack(Num('3')), Stackframe(Num('2')))` und `Assign(Stack(Num('3')), Global(Num('0')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_by_value_struct.reti_blocks',
  [
    Block
      Name 'fun_struct_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_struct_from_global_data.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Assign(Stack(Num('3')), Stackframe(Num('2')))
        SUBI SP 3;
        LOADIN BAF ACC -4;
        STOREIN SP ACC 1;
        LOADIN BAF ACC -3;
        STOREIN SP ACC 2;
        LOADIN BAF ACC -2;
        STOREIN SP ACC 3;
        # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 5;
        MOVE SP BAF;
        SUBI SP 5;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Assign(Stack(Num('3')), Global(Num('0')))
        SUBI SP 3;
        LOADIN DS ACC 0;
        STOREIN SP ACC 1;
        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 1.68: *RETI-Block Pass für die Übergabe eines Verbundes.*

Literatur

Online

- *Bäume*. URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17. 07. 2022).
- *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, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).

- Scholl, Prof. Dr. Christoph. „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03. 08. 2022).
- Thiemann, Peter. „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).

Sonstige Quellen

- *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).