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

[1] Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.
[2] Wofür ich mich auch nochmal Entschuldigen will.

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

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

[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 🙂.
[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

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Einführung

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

Anmerkung

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

Compiler, wie der **GCC**[2] oder **Clang**[3] werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm zu **Maschinencode** (Definition 2.4) zu kompilieren. Das Programm muss hierzu in der **Sprache** geschrieben sein, die der Compiler kompiliert[4]

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

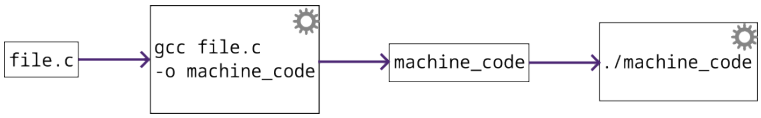


Abbildung 1.1: *Schritte zum Ausführen eines Programmes mit dem GCC.*

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

[1] Dieser Begriff ist **nicht** weiter **relevant**.
[2] *GCC, the GNU Compiler Collection - GNU Project*.
[3] *clang: C++ Compiler*.
[4] Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C.
[5] Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objektcode** (Definition 5.5) kompiliert wird. Das Kompilieren zu **Objektcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

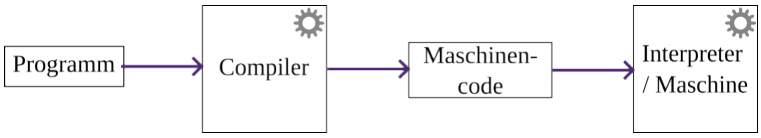


Abbildung 1.2: *Stark vereinfachte Schritte zum Ausführen eines Programmes.*

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

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

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

1.1 RETI-Architektur

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

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

Anmerkung

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

Die **Register** dieser **RETI-Architektur** werden in Tabelle 1.1 aufgezählt und erläutert. Der **Befehlssatz** und die **Datenpfade** der **RETI-Architektur** sind im Kapitel Appendix dokumentiert, da diese **nicht explizit**

zum **Verständnis** der späteren Kapitel notwendig sind. Allerdings sind diese zum **tieferen Verständnis** notwendig, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** L_{RETI} ist durch die Grammatiken 3.3.7 und 3.3.8 zusammengenommen beschrieben.

Register Kürzel	Register Ausgeschrieben	Aufgabe
`PC`	**P**rogram **C**ounter	Zeigt auf den **Maschinenbefehl**, der als nächstes ausgeführt werden soll.
`ACC`	**Acc**umulator	Für **Operanden** von Operationen oder für **temporäre** Werte.
`IN1`	**In**dexregister **1**	Hat dieselbe Aufgabe wie das `ACC`-Register.
`IN2`	**In**dexregister **2**	Hat dieselbe Aufgabe wie das `ACC`-Register.
`SP`	**S**tack**p**ointer	Zeigt immer auf die **erste freie Speicherzelle** am **Ende** des **Stacks**[a], wo als nächstes Speicher **allokiert** werden kann.
`BAF`	**B**egin **A**ktive **F**unktion	Zeigt auf den **Beginn** des **Stackframes** der aktuell **aktiven Funktion**.
`CS`	**C**odesegment	Zeigt auf den **Beginn** des **Codesegments**.
`DS`	**D**atensegment	Zeigt auf den **Beginn** des **Datensegments**. Die letzten 10 Bits werden verwendet, um 22 Bit Immediates **aufzufüllen**. Kann dadurch dazu verwendet werden, festzulegen welches der 3 **Peripheriegeräte**[b] in der **Memory Map**[c] angesprochen werden soll.

[a] Wird noch erläutert.
[b] **EPROM**, **UART** und **SRAM**.
[c] Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, wird diese **nicht** mehr als nötig im weiteren Verlauf **erläutert**.

Tabelle 1.1: *Register der RETI-Architektur.*

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

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

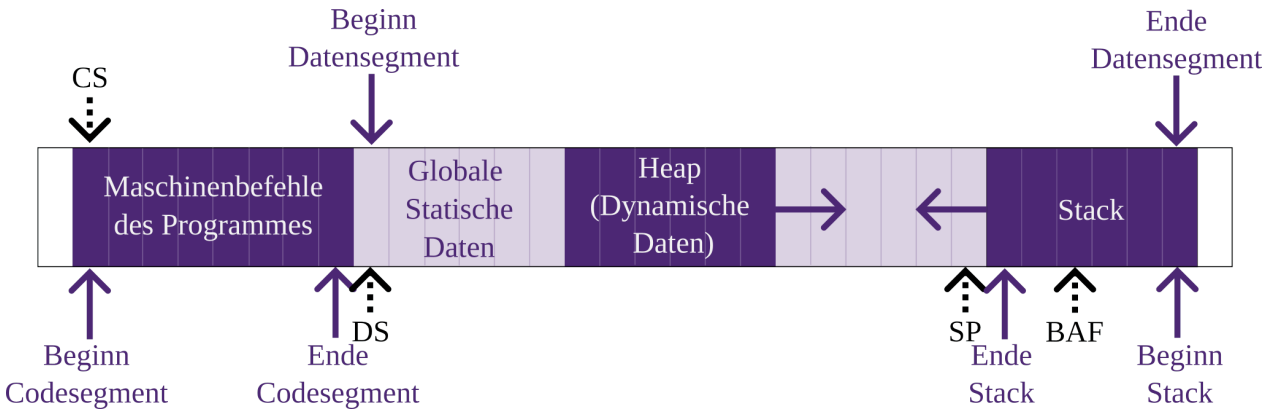


Abbildung 1.3: *Speicherorganisation.*

Die RETI-Architektur nutzt 3 verschiedene **Peripheriegeräte**, **EPROM**, **UART** und **SRAM**, die über eine **Memory Map**[6] den über die **Datenpfade** der RETI-Architektur 5.1 ansprechbaren **Adressraum** von 2^{32} Adressen[7] unter sich aufteilen.

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

Die **UART**[9] ist eine **elektronische Schaltung** mit je nach Umsetzung mehr oder weniger **Registern**. Es gibt allerdings immer einen `RX`- und einen `TX`-Register, für jeweils Empfangen[10] und Versenden[11] von Daten. Jedes der **Register** wird dabei mit einer anderen von 2^3 verschiedenen Adressen angsprochen. Jeweils 8-Bit können nach den **Datenpfaden der RETI-CPU** 5.1 auf einmal in ein **Register** der **UART** geschrieben werden, um **versandt** zu werden oder von einem Register **empfangen** werden. Die **UART** dient als **serielle Schnittstelle** und kann z.B. genutzt werden, um Daten an einen sehr einfach gehaltenen **Monitor** zu senden, der diese dann anzeigt.

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

1.2 Die Sprache PicoC

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

[6] Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, sondern nur bei der Umsetzung des **RETI-Interpreters**, wird diese **nicht näher erläutert** als notwendig.
[7] Von 0 bis $2^{32} - 1$.
[8] Kurz für **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
[9] Kurz für **U**niversal **A**synchronous **R**eceiver **T**ransmitter.
[10] Engl. **R**eceiving, daher das `R`.
[11] Engl. **T**ransmission, daher das `T`.
[12] Kurz für **S**tatic **R**andom-**A**ccess **M**emory.
[13] Ist für **Arithmetische**, **Bitweise** und **Logische Berechnungen** zuständig.

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/* comment */`.
- die **Basisdatentypen**[14] `int`, `char` und `void`.
- die **Zusammengesetzten** Datentypen[15] **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; int attr2;}`) und **Zeiger** (z.B. `int *pntr`), inklusive:
  - **Initialisierung** (z.B. `struct st st_var = {.attr1=42, .attr2={.attr={&var, &var}}}`).
  - dazugehörige **Operationen** `[i]`, `.attr`, `*` und `&`.
  - **Kombinationen** der eben genannten **Operationen** (z.B. `(*complex_var[0][1])[1].attr`) und **Datentypen** (z.B. `struct st (*complex_var[1][2])[2]`).
  - **Zeigerarithmetik** (z.B. `*(var + 2)`).
- `if(cond){ }`- und `else{ }`-**Anweisungen**[16].
- `while(cond){ }`- und `do while(cond){ };`-**Anweisungen**.
- **Arithmetische und Bitweise Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>` und **unären Operatoren** `-`, $\sim$ umgesetzt sind.[17]
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.
- **Zuweisungen**, welche mithilfe des **Zuweisungsoperators** `=` umgesetzt sind, inklusive:
  - Zuweisung an **Feldelement**, **Verbundsattribut** oder **Zeigerelement** (z.B. `(*var.attr)[2] = fun() + 42`).
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`), **Funktionsaufrufe** (z.B. `fun(ar, st_var)`) und **Sichtbarkeitsbereiche** innerhalb der Codeblöcke {} der Funktionen.

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

Der grundlegende **Aufbau** von Programmen der **Programmiersprache** L_{PicoC} ist durch Grammatik 3.1.1 und Grammatik 3.2.8 zusammengenommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

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

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

Anmerkung

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

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

Und aufgrund dessen, dass es sich um eine **Strukturierte** und **Prozedurale Programmiersprache** handelt, ist es eine gute Methode bei der Implementierung auf **Blöcke**[19] zu setzen, zwischen denen **hin und her** gesprungen werden kann. **Blöcke** stellen in den einzelnen Implementierungsschritten die **notwendige Datenstruktur** dar, um **Auswahl** zwischen Codestücken, **Wiederholung** von Codestücken und **Sprünge** zu Blöcken mit entsprechend zu bestimmten **Bezeichnern** (Definition 5.1) passenden **Labeln** (Definition 5.2) umzusetzen.

Definition 1.1: Imperative Programmierung

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

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

Definition 1.2: Strukturierte Programmierung

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

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

Definition 1.3: Prozedurale Programmierung

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

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

In L_{PicoC} ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen `[<i>]`-Klammern zur Festlegung der **Mächtigkeit**

[19] Werden später im Kapitel 3 genauer erklärt.

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

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

In Code 1.1 ist ein Beispiel zu sehen, indem die Variable `complex_var` den **Datentyp „Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Zeigern auf Felder der Mächtigkeit 2 von Verbunden vom Typ `st`“** hat. Ein Vorteil davon die eckigen `[<i>]`-Klammern **hinter** die Variable zu schreiben ist in der **markierten Zeile** in Code 1.1 zu sehen. Will man auf ein **Element dieses Datentyps** zugreifen `(*complex_var[0][1])[1].attr`, so ist der Ausdruck fast genau **gleich aufgebaut**, wie der Ausdruck für den **Datentyp** `struct st (*complex_var[1][2])[2]`. Die **Ausgabe** des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

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

Code 1.1: *Beispiel für die Spiralregel.*

```
1 42
```

Code 1.2: *Ausgabe des Beispiels für die Spiralregel.*

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

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

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

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

Code 1.3: *Beispiel für unterschiedliche Ausführung.*

```
1 42 42
```

Code 1.4: *Ausgabe des Beispiels für unterschiedliche Ausführung.*

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

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

Code 1.5: *Beispiel mit Dereferenzierungsoperator.*

```
1 42 42
```

Code 1.6: *Ausgabe des Beispiels mit Dereferenzierungsoperator.*

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

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

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

Definition 1.4: Call by Value

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

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

[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 des Beispiel dafür, dass Struct kopiert wird.*

In der Programmiersprache L_{PicoC} gibt es **kein Call by Reference** (Definition 1.5), allerdings kann der **Effekt** von Call by Reference mittels **Zeigern simuliert** werden, wie es in Code 1.11[22] bei der Funktion `fun_declared_before` und dem Parameter `int *param` zu sehen ist. Genau dieser **Trick** wird bei **Feldern** verwendet, um **nicht** das gesamte Feld bei einem Funktionsaufruf in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopieren** zu müssen.

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

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

[22] Unten im Code schauen.

Definition 1.5: Call by Reference

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

***Implizit** meint hier, dass der Benutzer einer Funktionalität mit Call by Reference **nicht mitbekommt**, dass er das **Argument selbst verändert** und **keine lokale Kopie des Arguments**.*[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 des Beispiels dafür, dass Zeiger auf Feld übergeben wird.*

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

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

Definition 1.6: Funktionsprototyp

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

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

unterschieden.
[b] *What is the difference between function prototype and function signature?*

Definition 1.7: Deklaration

*Der **Datentyp** bzw. **Prototyp** einer **Variablen** bzw. **Funktion**, sowie der **Bezeichner** dieser **Variable** bzw. **Funktion** wird dem **Compiler** mitgeteilt.*[abc]

[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 oder bereits vor der Ausführung **Speicher reserviert** werden soll und **wo**[a] dieser angelegt werden soll.*

[a] Im Fall des PicoC-Compilers im Abschnitt für die **Globalen Statischen Daten** oder auf dem **Stack**.

```
1  void fun_declared_before(int *param);
2
3  int fun_defined(int param) {
4    return param + 10;
5  }
6
7  void main() {
8    int res = fun_defined(22);
9    fun_declared_before(&res);
10   print(res);
11 }
12
13 void fun_declared_before(int *param) {
14   *param = *param + 10;
15 }
```

Code 1.11: *Beispiel für Deklaration und Definition.*

```
1 42
```

Code 1.12: *Ausgabe des Beispiels für Deklaration und Definition.*

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

Verschiedene **Sichtbarkeitsbereiche** können dabei **identische** Bezeichner besitzen. Im Beispiel in Code 1.13 kommt der markierte **Bezeichner** `local_var` in 2 verschiedenen **Sichtbarkeitsbereichen** vor und bezeichnet somit 2 **unterschiedliche Variablen**. Der **Parameter** `param` und die **Lokale Variable** `local_var` dürfen

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

Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)

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

[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 Sichtbarkeitsbereiche.*

```
1 4 2
```

Code 1.14: *Ausgabe des Beispiels für Sichtbarkeitsbereiche.*

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit war es, bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinensprache L_{RETI}, die **Syntax** und **Semantik** der Programmiersprache L_C **identisch** nachzuahmen. Der **PicoC-Compiler** sollte die Programmiersprache L_{PicoC} im Vergleich zu z.B. dem **GCC**[23] ohne merklichen Unterschied[24] kompilieren können.

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

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter Link[25] zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter **„Getting Started“** ein kleines **Einführungstutorial** verlinkt. Unter **„Usage“** ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene

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

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

[25] `https://github.com/matthejue/PicoC-Compiler`.

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

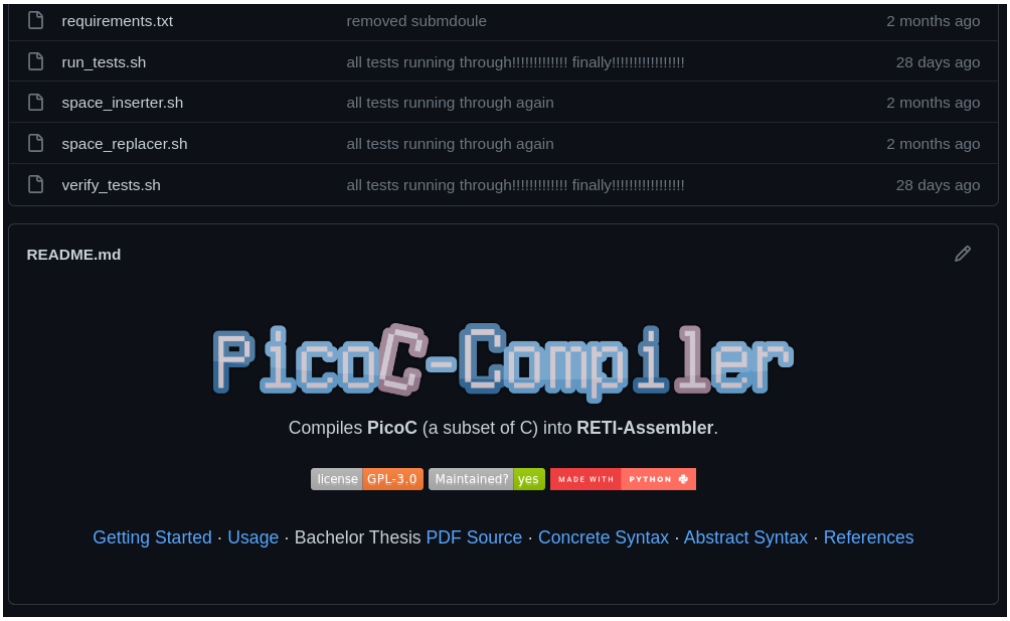


Abbildung 1.4: *README.md im Github Repository der Bachelorarbeit.*

Die **schriftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **schriftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF**-Datei unter Link[27] zu finden. Die **PDF**-Datei der schriftlichen Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode** automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert. Der **Latexquellcode** ist unter Link[28] veröffentlicht.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[29] zu finden[30]. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vektorgraphikeditors **Inkscape**[31] erstellt. Falls Interesse besteht, **Grafiken** aus dieser schriftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den `.svg`-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[32] zu finden.

Um die verschiedenen **Aspekte** der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel Einführung werden Codebeispiele zur **Anschauung** verwendet. Mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters** werden **Ausgaben** erzeugt, die in dieses Dokument **eingelesen** wurden. Im Kapitel Implementierung werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** in Form von Codebeispielen gezeigt[33].

[26] `https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971`.
[27] `https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf`.
[28] `https://github.com/matthejue/Bachelorarbeit`.
[29] `https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex`.
[30] Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.
[31] Developers, *Draw Freely — Inkscape*.
[32] `https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md`.
[33] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist. Später mehr dazu.

Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[34] finden. Mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` lassen sich die Codebeispiele genauso **kompilieren**, wie sie hier dargestellt sind[35].

1.5.1 Stil der schriftlichen Ausarbeitung

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

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas genau so gemacht wurde und nicht anders“. Wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist eine der **zentralen Fragen**, die ein Leser in erster Linie unter anderem zum initialen **wirklichen Verständnis** eines Themas beantwortet braucht[36], die Frage des **„warum“**.

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

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

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

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

[34] https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.
[35] Es wurde zu diesem Zweck die **Command-line Option** `-t`, `--thesis` erstellt, die bestimmte Kommentare **herausfiltert**, damit die generierten **Abstrakten Syntaxbäume** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu **überfüllt** mit Kommentaren sind.
[36] Vor allem am **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[37] Das ist ein **Beispiel** für eine **Fußnote**.
[38] Bei dem z.B. auch im Deutschen geläufigen Fachbegriff **„Statement“** war es eine schwierige Entscheidung, ob man nicht das deutsche Wort **„Anweisung“** verwenden soll. Da es **nicht verwirrend** klingt wurde sich dazu entschieden überall das deutsche Wort **„Anweisung“** zu verwenden.

diese Bezeichner in den **Ausgaben** des **PicoC-Compilers** vorkommen[39].

1.5.2 Aufbau der schriftlichen Arbeit

Der Inhalt dieser **schriftlichen Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: Einführung, Theoretische Grundlagen, Implementierung und Ergebnisse und Ausblick. **Zusätzlich** gibt es noch den Appendix.

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

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

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

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

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

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

Die **Sonstigen Defintionen** und das Thema **Bootstrapping** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Generell wurde immer versucht **Parallelen** zur Implementierung **echter** Compiler zu ziehen. Die **Erklärungen** und **Definitionen** hierfür wurden allerdings in den Appendix **ausgelaggert**. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition 5.12) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden. Es sollte sich an die **vorgegebenen Paradigmen** aus der Vorlesung C. Scholl, „Betriebssysteme“ gehalten

[39] Später werden unter anderem sogenannte **Abstrakte Syntaxbäume** (Definition 2.42) 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**.

werden.

2 Theoretische Grundlagen

In diesem Kapitel wird auf die **Theoretischen Grundlagen** eingegangen, die zum Verständnis der **Implementierung** in Kapitel 3 notwendig sind. Zuerst wird in Unterkapitel 2.1 genauer darauf eingegangen was ein **Compiler** und **Interpreter** eigentlich sind und damit in Verbindung stehende **Begriffe** und **T-Diagramme** erklärt. Danach wird in Unterkapitel 2.2 eine kleine Einführung zu einem der Grundpfeiler des Compilerbau, den **Formalen Sprachen** gegeben. Danach werden die einzelnen **Filter** des üblicherweise bei der Implementierung von Compilern genutzten **Pipe-Filter-Architekturpatterns** (Definition 2.1) nacheinander erklärt. Die **Filter** beinhalten die **Lexikalische Analyse** 2.3, **Syntaktische Analyse** 2.4 und **Code Generierung** 2.5. Zum Schluss wird in Unterkapitel 2.6 darauf eingegangen in welchen Situationen **Fehlermeldungen** auszugeben sind.

Definition 2.1: Pipe-Filter Architekturpattern

*Ist ein **Archikteturpattern**, welches aus **Pipes** und **Filtern** besteht, wobei der **Ausgang** eines **Filters** der **Eingang** des durch eine **Pipe** verbundenen adjazenten nächsten **Filters** ist, falls es einen gibt.*

*Ein **Filter** stellt einen Schritt dar, indem eine Eingabe **weiterverarbeitet** wird. Bei der **Weiterverarbeitung** können Teile der Eingabe **entfernt**, **hinzugefügt** oder **vollständig ersetzt** werden.*

*Eine **Pipe** stellt ein **Bindeglied** zwischen zwei **Filtern** dar.*[a][b]

[a] Das ein **Bindeglied** eine eigene Bezeichnung erhält, bedeutet allerdings nicht, dass es eine eigene wichtige **Aufgabe** erfüllt. Wie bei vielen **Pattern**, soll mit dem Namen des **Pattern**, in diesem Fall durch das **Pipe** die Anlehung an z.B. die **Pipes aus Unix**, z.B. `cat /proc/bus/input/devices | less` zum Ausdruck gebracht werden. Und so banal es klingt, sollen manche Bezeichnungen von Pattern auch einfach nur **gut klingen**.

[b] Westphal, „Softwaretechnik“.

2.1 Compiler und Interpreter

Die wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.3) und eines **Interpreters** (Definition 2.2), da das Schreiben eines Compilers von der **PicoC-Sprache** L_{PicoC} in die **RETI-Sprache** L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist.

Anmerkung

Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[a] und von **Tests** die **Beziehungen** in 2.3.1 zu belegen (siehe Unterkapitel 4.2), weshalb es auch nochmal wichtig ist die Definition eines **Interpreters** eingeführt zu haben.

[a]Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **G**NU **C**ompiler **C**ollection

Definition 2.2: Interpreter

*Programm, dass die **Anweisungen** eines Programmes mehr oder weniger **direkt** ausführt.*

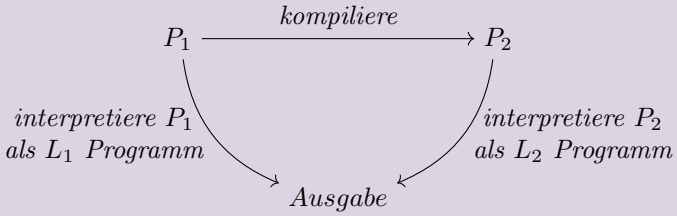
*In einer konkreten Implementierung arbeitet ein Interpreter auf einem compilerinternen **Abstrakten Syntaxbaum** (wird später eingeführt unter Definition 2.42) und führt je nach Komposition der **Knoten** des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche **Aufgaben** aus.*[a]

[a]G. Siek, *Essentials of Compilation*.

Definition 2.3: Compiler

***Übersetzt** ein **beliebiges** Program P_1, welches in einer Sprache L_1 geschrieben ist in ein Programm P_2, welches in einer Sprache L_2 geschrieben ist.*

*Dabei muss gelten, dass die beiden Programme P_1 und P_2, wenn sie von **Interpretern** ihrer jeweiligen Sprachen L_1 und L_2 **interpretiert** werden die gleiche **Ausgabe** haben. Dies ist in Diagramm 2.3.1 dargestellt. Beide Programme P_1 und P_2 sollen die gleiche **Semantik** (Definition 2.16) haben und unterscheiden sich nur **syntaktisch** (Definition 2.15).*[a]



(2.3.1)

[a]G. Siek, *Essentials of Compilation*.

Üblicherweise kompiliert ein **Compiler** ein **Programm**, das in einer **Programmiersprache** geschrieben ist zu einem **Maschinenprogramm**, das in **Maschinensprache** (Definition 2.4) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 5.7 im Appendix) oder **Cross-Compiler** (Definition 2.6)[1].

Definition 2.4: Maschinensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** von zuvor hierzu kompilierten bzw. assemblierten Programmen darstellen.*

*Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, welche eine CPU im **einfachen Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauergesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen, konstanten** Anzahl von Fetch- und Execute Phasen im **komplexeren Fall**.*

*Die Maschinenbefehle sind meist so entworfen, dass sie sich innerhalb bestimmter **Wortbreiten**, die **Zweierpotenzen** sind kodieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.*[a]

[1]Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 5.3 im Appendix) voneinander zu unterscheiden.

*Die Programme[b] einer Maschinensprache können dabei in verschiedenen **Repräsentationen** dargestellt werden, wie z.B. in **binärer** Rerpräsentation, **hexadezimaler** Repräsentation, aber auch in **menschenlesbarer**[c] Repräsentation.*[d]

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.5) haben.
[b] Bzw. **Wörter**.
[c] So wie die **Programme** des **PicoC-Compilers** dargestellt werden.
[d] C. Scholl, „Betriebssysteme“.

Die **Folge von Maschinenbefehlen**, die ein üblicher Compiler generiert, ist üblicherweise in **binärer Repräsentation**, da die Maschinenbefehle in erster Linie für die Maschine, die **binär** arbeitet verständlich sein sollen und nicht für den Programmierer.

Anmerkung

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings RETI-Code, der die RETI-Befehle in **menschenlesbarer Repräsentation** mit menschenlesbar ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.5) enthält. Für den **RETI-Interpreter** ist es ebenfalls **nicht** notwendig, dass der RETI-Code, den der PicoC-Compiler generiert, in **binärer Repräsentation** ist, denn es ist für den RETI-Interpreter ebenfalls leichter diesen einfach direkt in **menschenlesbarer Repräsentation** zu interpretieren. Der RETI-Interpreter soll nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** und **nicht** deren mögliche **interne Umsetzung**[a].

[a] Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschinencode in z.B. **UTF-8 Kodierung** ausführen kann, wäre dagegen **unnötig kompliziert** und **aufwändig**, da Hardware **binär** arbeitet und man dieser daher lieber direkt die binär kodierten Maschinenbefehle übergibt, anstatt z.B. eine unnötig **platzverbrauchenden** UTF-8 Codierung zu verwenden, die nur in sehr **vielen Schritten** einen Befehl verarbeiten kann, da die Register und Speicherzellen des Hauptspeichers üblicherweise nur **32-** bzw. **64-Bit Breite** haben.

Definition 2.5: Immediate

***Konstanter Wert**, der als **Teil** eines **Maschinenbefehls** gespeichert ist und dessen **Wertebereich** dementsprechend auch durch die die **Anzahl an Bits**, die ihm innerhalb dieses **Maschinenbefehls** zur Verfügung stehen **beschränkt** ist. Der Wertebereich ist **beschränkter** als bei sonstigen Werten innerhalb des Hauptspeichers, denen eine **ganze Speicherzelle** des Hauptspeichers zur Verfügung steht.*[a]

[a] Ljohhuh, *What is an immediate value?*

Definition 2.6: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Programm, dass in einer **Wunschsprache** L_w geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschinensprachen** L_{M_1} und L_{M_2} haben.*[a][b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C_{PicoC}^{Python}, der in der Sprache L_{Python} geschrieben ist und die Sprache L_{PicoC} kompiliert.
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Ein **Cross-Compiler** ist entweder notwendig, wenn eine Zielmaschine M_2 nicht ausreichend **Rechenleistung** besitzt, um ein Programm in der Wunschsprache L_w selbst **zeitnah** zu kompilieren oder wenn noch **keine** Compiler C_w oder C_o für die **Wunschsprache** L_w oder eine **andere** Programmiersprache L_o, in welcher der

Wunschcompiler C_w implementiert ist existieren, die unter der **Zielmaschine** M_2 laufen.[2]

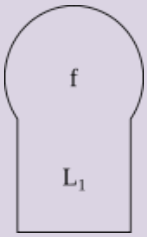
2.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus der Wissenschaftlichen Publikation J. Earley und Sturgis, „A formalism for translator interactions“ entnommen ist besonders gut, da diese optimal darauf **zugeschnitten** sind die Eigenheiten von Compilern und Interpretern in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Programm (Definition 2.7), einen Übersetzer (Definition 2.8), einen Interpreter (Definition 2.9) und eine Maschine (Definition 2.10) zusammen.

Definition 2.7: T-Diagramm Programm

*Repräsentiert ein **Programm**, dass in der **Sprache** L_1 geschrieben ist und die **Funktion** f berechnet.*[a]



[a] J. Earley und Sturgis, „A formalism for translator interactions“.

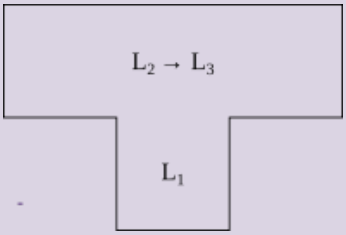
Anmerkung

Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.7 also reichen einfach eine 1 hinzuschreiben.

Definition 2.8: T-Diagramm Übersetzer (bzw. eng. Translator)

*Repräsentiert einen **Übersetzer**, der in der **Sprache** L_1 geschrieben ist und **Programme** von der **Sprache** L_2 in die **Sprache** L_3 übersetzt.*

*Für einen **Übersetzer** gelten genauso, wie für einen **Compiler** die Relationen in Abbildung 2.3.1.*[a]



[a] J. Earley und Sturgis, „A formalism for translator interactions“.

[2] Die an vielen Universitäten und Schulen eingesetzten programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren **Microcontroller** eine zu L_C **ähnliche Sprache** in die **Maschinensprache** des Microcontrollers zu kompilieren, da es **schneller** geht ein Programm direkt auf der Maschine, auf der man programmiert zu kompilieren.

Anmerkung

Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen **Unterschied**.

Übersetzung ist der **allgemeinere** Begriff und verlangt nur, dass **Eingabe** und **Ausgabe** des **Übersetzers** die **gleiche Bedeutung**[a] haben müssen, also die Relationen in Abbildung 2.3.1 erfüllt sind[b].

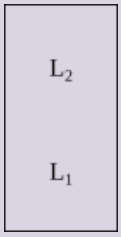
Kompilierung beinhaltet dagegen meist auch das **Lexen** und **Parsen** oder irgendeine Form von **Umwandlung** eines Programmes von der **Textrepräsentation** in eine **compilerinterne Datenstruktur** und erst dann ein oder mehrere **Übersetzungsschritte**.

Kompilieren ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

[a] Auch **Semantik** (Definition 2.16) genannt.
[b] Und ist auch zwischen **Passes** (Definition 2.43) möglich.

Definition 2.9: T-Diagramm Interpreter

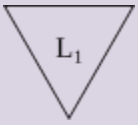
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



[a] J. Earley und Sturgis, „A formalism for translator interactions“.

Definition 2.10: T-Diagram Maschine

*Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** L_1 ausführt.*[a b]



[a] Wenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Aus den verschiedenen **Blöcken** lassen sich **Kompositionen** bilden, indem man sie **adjazent** zueinander platziert. Allgemein lässt sich grob sagen, dass **vertikale Adjazenz** für **Interpretation** und **horinzontale Adjazenz** für **Übersetzung** steht.

Die **horinzontale Adjazenz** lässt sich, wie man in Abbildung 2.1 erkennen kann zusammenfassen.

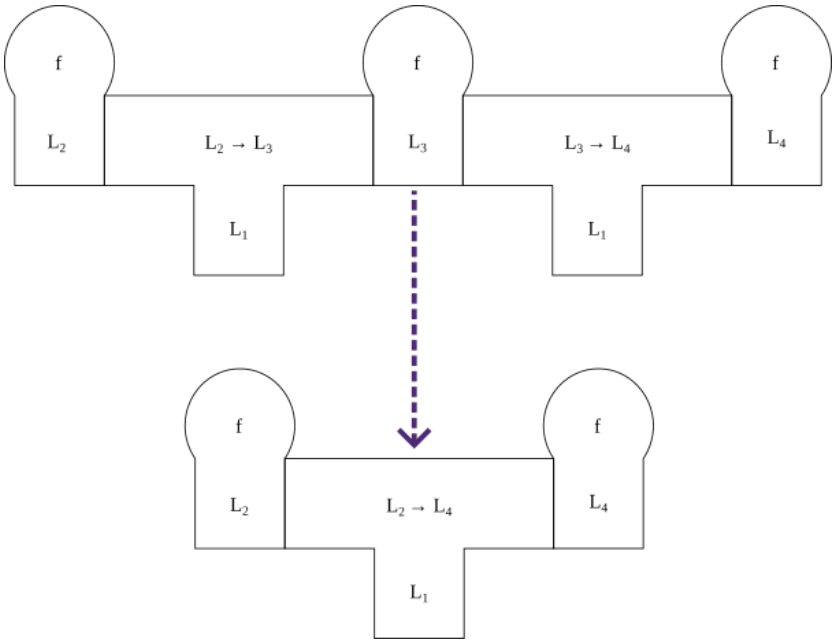


Abbildung 2.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen.*

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.14) zu tun, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der **Sprache** L_1 in eine **Sprache** L_2 ist. Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen** vorher eingeführt zu haben, was die Begriffe **Symbol** (Definiton 2.11), **Alphabet** (Definition 2.12) und **Wort** (Definition 2.13) beinhaltet.

Definition 2.11: Symbol

*„Ein **Element** eines **Alphabets** Σ. “*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.12: Alphabet

*„Ist eine **endliche, nicht-leere** Menge von **Symbolen** (Definition 2.11). “*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.13: Wort

*„Ein Wort $w = a_1...a_n \in \Sigma^*$ ist eine **endliche Folge** von **Symbolen** aus einem **Alphabet** Σ.*

*Es gibt es die **Konkatenation** $w_1w_2 = a_1 \dots a_nb_1 \dots b_n$ von **Wörtern** $w_1 = a_1 \dots a_n$ und $w_2 = b_1 \dots b_n$ und die **Länge** eines **Wortes** $|w|$.*

*Ein wichtiges Wort ist das **leere Wort** ε, für das gilt: $|\varepsilon| = 0$ und $\forall w \in \Sigma^* : \varepsilon w = w\varepsilon = w$. Es handelt sich bei ε also um das **Neutrale Element** bei der **Konkatenation** von **Wörtern**.*

*Bei Σ^* handelt es sich um **Kleenesche Hülle** eines **Alphabets** Σ, es ist die **Sprache aller Wörter**, welche durch beliebige **Konkatenation** von **Symbolen** aus dem **Alphabet** Σ gebildet werden können. Die **Kleenesche Hülle** ist die **größte Sprache** über Σ und **jede Sprache** über Σ ist eine **Teilmenge** davon. Es gilt des Weiteren: $\varepsilon \in \Sigma^*$. "*[a]

[a] Nebel, „Theoretische Informatik".

Definition 2.14: Formale Sprache

*„**Menge** von **Wörtern** (Definition 2.13) über dem **Alphabet** Σ (Definition 2.12). "*[a]

*Das Adjektiv **„formal"** kann dabei weggelassen werden, wenn der **Kontext** indem die **Sprache** verwendet wird **eindeutig** ist, da man das Adjektiv **„formal"** nur verwendet, um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Sprache** herauszustellen.*

[a] Nebel, „Theoretische Informatik".

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die **Semantik** (Definition 2.16) **gleich** bleiben. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 2.17), welche diese beschreibt und können in verschiedenen **Syntaxen** (Definition 2.15) dargestellt sein.

Definition 2.15: Syntax

*Bezeichnet alles was mit dem **Aufbau** von Wörtern einer **Formalen Sprache** zu tun hat. Eine **Formale Grammatik** oder in **Natürlicher Sprache** ausgedrückte Regeln können die Syntax einer Sprache beschreiben. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben*[a].[b]

[a] Z.B. die **Konkrete** (Definition 2.33) und **Abstrakte Syntax** (Definition 2.40), die später eingeführt werden.
[b] Thiemann, „Einführung in die Programmierung".

Definition 2.16: Semantik

*Bezeichnet alles was mit der **Bedeutung** von Wörtern einer **Formalen Sprache** zu tun hat.*[a]

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

Definition 2.17: Formale Grammatik

*„Beschreibt, wie **Wörter** einer **Formalen Sprache** abgeleitet werden können.*

*Das Adjektiv **‚formal'** kann dabei weggelassen werden, wenn der **Kontext** indem die **Grammatik** verwendet wird **eindeutig** ist, da man das Adjektiv **‚formal'** nur verwendet, um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Grammatik** herauszustellen.*

*Eine **Formale Grammatik** wird durch das Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt, wobei":*

- $N \mathrel{\hat{=}}$ ***Nicht-Terminalsymbole**.*
- $\Sigma \mathrel{\hat{=}}$ ***Terminalsymbole**, wobei $N \cap \Sigma = \emptyset$.*[a][b]
- $P \mathrel{\hat{=}}$ *Menge von **Produktionsregeln** $w \to v$, wobei $w, v \in (N \cup \Sigma)^*$ und $w \notin \Sigma^*$.*[c][d]
- $S \mathrel{\hat{=}}$ ***Startsymbol**, wobei $S \in N$.*

„Zusätzlich ist es praktisch ***Nicht-Terminalsymbole*** *N,* ***Terminalsymbole*** *Σ und das* ***leere Wort*** *ε allgemein als Menge der* ***Grammatiksymbole*** *$C = N \cup \Sigma \cup \varepsilon$ zu definieren.*

Es ist möglich ***zwei Grammatiken*** *G_1 und G_2 in einer* ***Vereinigungsgrammatik*** *$G_1 \uplus G_2 = \langle N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S ::= S_1 \mid S_2\}, S\rangle$ zu vereinigen.“*[e]

„Des Weiteren gibt es die von einer ***Grammatik erzeugte Sprache****: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ “*[f].[g]

[a] Weil mit ihnen **terminiert** wird.
[b] Kann auch als **Alphabet** bezeichnet werden.
[c] w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.
[d] Bzw. $w, v \in C^*$ und $w \notin \Sigma^*$.
[e] Die **Grammatik des PicoC-Compilers** lässt sich in eine Grammatik für die **Lexikalische Analyse** G_{Lex} und eine für die **Syntaktische Analyse** G_{Parse} unterteilen. Die **gesamte Grammatik** des **PicoC-Compilers** steht allerdings **vereinigt** in einer Datei.
[f] Die **Nicht-Terminlsymbole** in w fallen dabei weg, weil $w \in \Sigma^*$
[g] Nebel, „Theoretische Informatik“.

Formale Sprachen lassen sich in Klassen der **Chromsky Hierarchie** (Definition 2.18) einteilen.

Definition 2.18: Chromsky Hierarchie

Eine Hierarchie, in der ***Formale Sprachen*** *nach der* ***Komplexität*** *ihrer* ***Formalen Grammatiken*** *in verschiedene* ***Klassen*** *unterteilt werden. Jede dieser Klassen hat verschiedene* ***Eigenschaften****, wie* ***Entscheidungeprobleme****, die in einer Klasse* ***entscheidbar*** *und in einer anderen* ***unentscheidbar*** *sind usw.*

Eine Sprache L_i ist in der ***Chromsky Hierarchie*** *in der* ***Klasse*** *$i \in \{0, \ldots, 3\}$, falls sie von einer Grammatik dieser* ***Klasse*** *i erzeugt*[a] *werden kann.*

Zwischen den Sprachmengen ***benachbarter Klassen*** *in Abbildung 2.18.1 besteht eine* ***echte Teilmengenbeziehung****: $L_3 \subset L_2 \subset L_1 \subset L_0$.*[b][c]

(2.18.1)

[a] **Erzeugen** meint hier, dass **genau** die **Wörter** der Sprache sich mit der **Grammatik** ableiten lassen, **keines mehr** oder **weniger**.
[b] Z.B. ist jede **Reguläre Sprache** auch eine **Kontextfreie Sprache**, aber nicht jede **Kontextfreie Sprache** ist auch eine **Reguläre Sprache**.
[c] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 2.19) und **Kontextfreien Grammatiken** (Definition 2.20) bestimmt werden.

Definition 2.19: Reguläre Grammatik

*"Ist eine Grammatik für die gilt, dass **alle Produktionen** eine der Formen:*

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{2.19.1}$$

*haben, wobei A, B **Nicht-Terminalsymbole** sind und c ein **Terminalsymbol** ist[a][b]."[c]*

[a] Diese Definition einer **Regulären Grammatik** ist **rechtsregulär**, es ist auch möglich diese Definition **linksregulär** zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.
[b] Dadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.
[c] Nebel, „Theoretische Informatik".

Definition 2.20: Kontextfreie Grammatik

*"Ist eine Grammatik für die gilt, dass **alle Produktionen** die Form:*

$$A \to v \tag{2.20.1}$$

*haben, wobei A ein **Nicht-Terminalsymbol** ist und v ein beliebige Folge von **Grammatiksymbolen**[a] ist."[b]*

[a] Also eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.
[b] Nebel, „Theoretische Informatik".

Ob sich ein Programm überhaupt kompilieren lässt, entscheidet sich anhand des **Wortproblems** (Definition 2.21). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise **entscheidbar**. Wenn das Programm ein **Wort** der **Sprache** ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es **kein Wort** der **Sprache**, die der Compiler kompiliert, wird eine **Fehlermeldung** ausgegeben.

Definition 2.21: Wortproblem

*Ein Entscheidungeproblem, bei dem man zu einem **Wort** $w \in \Sigma^*$ und einer **Sprache** L als **Eingabe** 1 oder 0 **ausgibt**[a], je nachdem, ob dieses **Wort** w Teil der **Sprache** L ist ($w \in L$) oder **nicht** ($w \notin L$).*

*Das Wortproblem kann durch die folgende **Indikatorfunktion**[b] zusammengefasst werden:*[c]

$$\mathbb{1}_L : \Sigma^* \to \{0,1\}, w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \tag{2.21.1}$$

[a] Bzw. **„ja"** oder **„nein"** usw., es muss **nicht** umgedingt 1 oder 0 sein.
[b] Auch **Charakteristische Funktion** genannt.
[c] Nebel, „Theoretische Informatik".

2.2.1 Ableitungen

Jedes mit einen Compiler kompilierbare **Programm** kann mithilfe der **Grammatik** der **Sprache** des Compilers **abgeleitet** werden. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 2.22) und der normalen **Ableitungsrelation** (Definition 2.23) unterschieden.

Definition 2.22: 1-Schritt-Ableitungsrelation

*„Eine **binäre Relattion** $\Rightarrow$, welche alle Anordnungen zweier Wörter $w_1, w_2 \in (N \cup \Sigma)^*$ in Relation zueinander setzt, die sich nur durch das **einmalige** Anwenden einer Produktionsregel auf w_1 voneinander*

unterscheiden.

*Es gilt $u \Rightarrow v$ **genau dann wenn** $u = w_1\,x\,w_2$, $v = w_1\,y\,w_2$ **und** es eine Regel $x \to y \in P$ gibt, wobei $w_1, w_2, x, y \in (N \cup \Sigma)^*$ “*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.23: Ableitungsrelation

*„Eine **binäre Relation** $\Rightarrow^*$, welche der **reflexive, transitive Abschluss** der **1-Schritt-Ableitungsrelation** $\Rightarrow$ ist. Auf der **rechten** Seite der Ableitungsrelation $\Rightarrow^*$ steht also ein Wort v aus $(N \cup \Sigma)^*$, welches durch **beliebig häufiges** Anwenden von Produktionsregeln auf ein Wort u entsteht.*

*Es gilt $u \Rightarrow^* v$ **genau dann wenn** $u = w_1 \Rightarrow \ldots \Rightarrow w_n = v$, wobei $n \geq 1$ und $w_1, \ldots, w_n \in (N \cup \Sigma)^*$. “*[a]

[a] Nebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden. Dasselbe **Programm** kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.24) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 und bei den **Linksableitungen** im Unterkapitel 3.2.1 relevant.

Definition 2.24: Links- und Rechtsableitungableitung

*„In jedem **Ableitungsschritt** wird bei **Typ-3- und Typ-2-Grammatiken** auf das am **weitesten links (Linksableitung)** bzw. **rechts (Rechtsableitung)** stehende **Nicht-Terminalsymbol** eine Produktionsregel angewandt. “*[a]

*Mit diesem Vorgehen kann man jedes ableitbare Wort generieren, denn dieses Vorgehen entspricht **Tiefensuche** von **links-nach-rechts** bzw. **rechts-nach-links**.*[b]

[a] Bei **Typ-1- und Typ-0-Grammatiken** ist es statt einem **Nicht-Terminalsymbol** die **linke** Seite einer Produktion.
[b] Nebel, „Theoretische Informatik“.

Ob eine Grammatik **mehrdeutig** (Definition 2.26) ist, kann durch Betrachtung **Formaler Ableitungsbäume** (Definition 2.25) festgestellt werden. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Formale Ableitungsbäume (Definition 2.35) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 2.25: Formaler Ableitungsbaum

*Ist ein Baum, in dem die Syntax eines **Wortes**[a] nach den **Produktionen** einer zugehörigen Grammatik, die angewendet werden mussten um das Wort abzuleiten **hierarchisch** zergliedert dargestellt wird.*

*Das Adjektiv **„formal“** kann dabei weggelassen werden, wenn der **Kontext** indem der **Ableitungsbaum** verwendet wird **eindeutig** ist, da man das Adjektiv **„formal“** nur verwendet, um den Unterschied zum **compilerinternen Ableitungsbaum** herauszustellen, der den Formalen Ableitungsbaum als **compilerinterne Datentstruktur** umsetzt.*

*Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.17) zugeordnet. Den **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** N zugeordnet und den **Blättern** sind entweder **Terminalsymbole** Σ oder das **leere Wort** ε zugeordnet.*[b]

[a] Z.B. **Programmcode**.
[b] Nebel, „Theoretische Informatik“.

In Abbildung 2.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 2.25.1 nach der im **Dialekt der Erweiterten Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.4) angegebenen **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ 2.1 ergibt.

NUM	::=	"4" | "2"	*L_Lex*
ADD_OP	::=	"+"	
MUL_OP	::=	"∗"	
mul	::=	*mul MUL_OP NUM* | *NUM*	*L_Parse*
add	::=	*add ADD_OP mul* | *mul*	

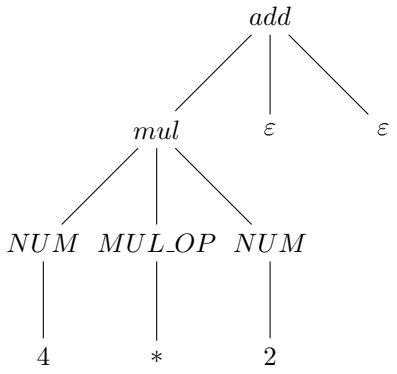
Grammatik 2.1: *Grammatik für einen Ableitungsbaum in EBNF*

Anmerkung

Werden die **Produktionen** einer Grammatik angegeben, wie in Grammatik 3.1.1, wird die Angabe dieser Produktionen auch oft als **Grammatik** bezeichnet, obwohl Grammatiken eigentlich durch ein Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt sind.

$$add \Rightarrow mul \Rightarrow mul\ \ MUL\_OP\ \ NUM \Rightarrow NUM\ \ MUL\_OP\ \ NUM \Rightarrow^* \text{"4"}\ \ \text{"}*\text{"}\ \ \text{"2"} \tag{2.25.1}$$

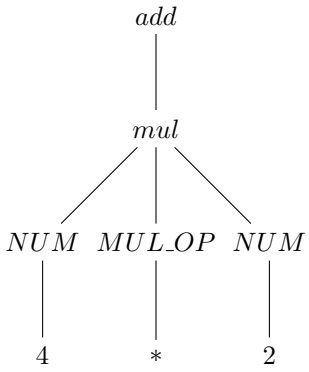
Bei Ableitungsbäumen gibt es **keine** einheitliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 2.25.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort** ε hinzuzufügen.



(2.25.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 2.25.3, nur die in den gewählten Produktionen vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[3].

[3] Diese Option wurde beim **PicoC-Compiler** gewählt.



(2.25.3)

2.2.2 Präzedenz und Assoziativität

Für einen Compiler ist es notwendig, dass die Grammatik des Compilers keine **Mehrdeutige Grammatik** (Definition 2.26) ist, denn sonst können unter anderem die **Assoziativität** (Definiton 2.27) und die **Präzedenz** (Definition 2.28) der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel 3.2.1 an einem Beispiel demonstriert wird. Ein Schema, um die Grammatiken zu definieren, die **nicht mehrdeutig** sind, wird in Unterkapitel 3.2.1 genauer erklärt.

Definition 2.26: Mehrdeutige Grammatik

*„Eine Grammatik ist **mehrdeutig**, wenn es ein Wort $w \in L(G)$ gibt, das mehrere **Ableitungsbäume** zulässt“*[a][b][c].

[a] Für die Bedeutung von $L(G)$, siehe Definition 2.17.
[b] **Alternativ** geht auch die Definition: „Wenn es für w **mehrere** unterschiedliche **Linksableitungen** gibt“.
[c] Nebel, „Theoretische Informatik“.

Definition 2.27: Assoziativität

*„Bestimmt, welcher Operator aus einer Reihe von Operatoren mit **gleicher Präzedenz** (Definition 2.28) **zuerst** ausgewertet wird.“*

*Es wird zwischen **linksassoziativen** Operatoren, bei denen der **linke Operator** vor dem **rechten Operator** ausgewertet wird und **rechtsassoziativen** Operatoren, bei denen es genau anders rum ist unterschieden.*[a][b]

[a] Nystrom, *Parsing Expressions · Crafting Interpreters*.
[b] *2.1.7 Vorrangregeln und Assoziativität*.

Bei **Assoziativität** ist z.B. der **Multitplikationsoperator** `*` ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator** `=` ein Beispiel für einen **rechtsassoziativen** Operator. In Abbildung 2.2 ist ein Beispiel hierfür dargestellt, indem die resultierenden **Auswertungsreihenfolgen** mithilfe von Klammern `()` veranschaulicht sind.

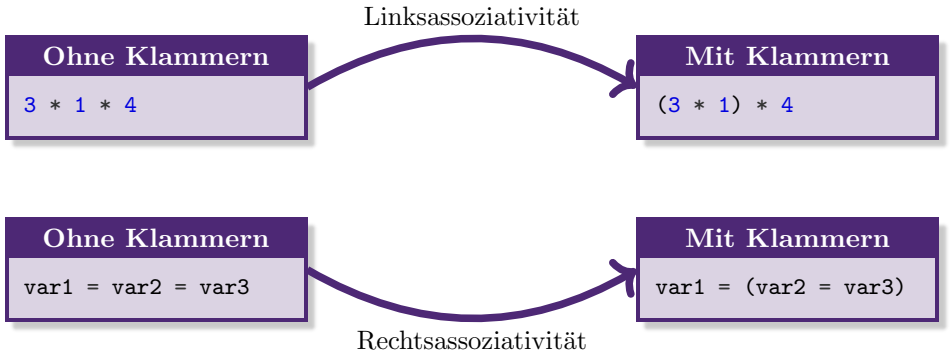


Abbildung 2.2: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität.*

Definition 2.28: Präzedenz

*„Bestimmt für einem Ausdruck, der eine Mischung **verschiedener** Operatoren enthält, welcher Operator **zuerst** ausgewertet wird. Operatoren mit einer **höheren Präzedenz** werden **vor** Operatoren mit einer **niedrigeren Präzedenz** ausgewertet.“*[a][b]

[a] Ein anderes Wort für **Präzedenz** ist **Vorrang**.
[b] Nystrom, *Parsing Expressions · Crafting Interpreters*.

Die Mischung der Operatoren für **Subtraktion** `'-'` und für **Multiplikation** `*`, welche beide eine **unterschiedliche Präzedenz** haben, ist ein Beispiel für den Einfluss von Präzedenz. In Abbildung 2.3 ist ein Beispiel hierfür dargestellt, indem mithilfe der Klammern `()` die resultierende **Auswertungsreihenfolge** veranschaulicht ist. Im Beispiel in Abbildung 2.3 ist durch die beiden **Subtraktionsoperatoren** `'-'` nacheinander und den darauffolgenden **Multitplikationsoperator** `*`, sowohl **Assoziativität** als auch **Präzedenz** im Spiel.

Abbildung 2.3: *Veranschaulichung von Präzedenz.*

2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.1) bei der Implementierung von Compilern. Die Aufgabe der Lexikalischen Analyse ist in einem ersten Schritt in einem Eingabewort[4] **Lexeme** (Definition 2.29) zu finden, die mit einer **Grammatik** für die Lexikalische Analyse G_{Lex} abgeleitet werden können.

Definition 2.29: Lexeme

*Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches mit einer **Grammatik** für die Lexikalische Analyse G_{Lex} abgeleitet werden kann.*[a]

[a] Thiemann, „Compilerbau“.

[4] Z.B. dem Inhalt einer Datei, welche in **UTF-8** kodiert ist.

Diese **Lexeme** werden vom **Lexer** (Definition 2.31) im **Eingabewort** identifziert und **Tokens** (Definition 2.30) zugeordnet. Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.30: Token

*Ist ein **Tupel** (T, V) mit einem **Tokentyp** T und einem **Tokenwert** V. Ein **Tokentyp** T kann hierbei als ein **Überbegriff** für eine möglicherweise unendliche Menge verschiedener **Tokenwerte** V verstanden werden*[a].

[a] Z.B. gibt es im **PicoC-Compiler** viele verschiedene **Tokenwerte**, wie z.B. `42`, `314` oder `12`, welche alle unter dem **Tokentyp** `NUM`, für Zahl zusammengefasst sind.

Definition 2.31: Lexer (bzw. Scanner oder auch Tokenizer)

*Ein **Lexer** ist eine **Totale Funktion**[a]: $lex : \Sigma^* \to (T \times V)^*, w \mapsto (t_1, v_1) \dots (t_n, v_n)$, welche ein **Eingabewort**[b] $w \in \Sigma^*$ auf eine **endliche Folge von Tokens** $(t_1, v_n) \dots (t_n, v_n) \in (T \times V)^*$ abbildet.*

*Die **Definitionsmenge** der **Totalen Funktion** lex erlaubt nur **Wörter**, die sich mit der **Grammatik** für die Lexikalische Analyse G_{Lex} ableiten lassen.*[c]

[a] Alternativ könnte man die Funktion `lex` auch als **Partielle Funktion** definieren, aber das würde zum Ausdruck bringen, dass der **Lexer** bei einem **Eingabewort**, das sich **nicht** mit der Grammatik für die Lexikalische Analyse G_{Lex} ableiten lässt trotzdem durchchläuft. Die Funktion *lex* als **Totale Funktion** zu definieren drückt eher aus, dass ein **Eingabewort**, das **nicht** in der **Definitionsmenge** liegt zu einer **Fehlermeldung** führt.
[b] Z.B. **Quellcode** eines **Eingabeprogramms**.
[c] Thiemann, „Compilerbau“.

Ist das Abbilden eines **Eingabeworts** w auf eine **Folge von Tokens** $(t_1, v_n) \dots (t_n, v_n)$ nicht möglich, da das Eingabewort **Teilwörter** enthält, die sich **nicht** mit der **Grammatik** für Lexikalische Analyse G_{Lex} ableiten lassen, so wird in diesem Fall eine **Fehlermeldung** (Definition 2.53) ausgegeben.

Anmerkung

Um Verwirrung vorzubeugen ist es wichtig die **kontextabhängigen** unterschiedliche Bedeutungen des Wortes **Symbol** hervorzuheben:

Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktischen Analyse** und der **Code Generierung** unterschiedliche Konzepte als Symbole bezeichnet.

In der Lexikalischen Analyse sind einzelne **Zeichen eines Zeichensatzes** die Symbole.

In der Syntaktischen Analyse sind die **Tokentypen** die Symbole.

In der Code Generierung sind die **Bezeichner** (Definition 5.1) von **Variablen, Konstanten und Funktionen** die Symbole[a].

[a] Das ist der Grund, warum in Kapitel 3 die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, **Symboltabelle** genannt wird.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die **Syntaktische Analyse** unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[5] und Tabs `\t` aus dem Eingabewort w **herauszufiltern**. Das geschieht im **Lexer**, indem dieser für alle unwichtigen Symbole bzw. Folgen von Symbolen[6] **kein Token** in der Folge von Tokens $(t_1, v_n) \dots (t_n, v_n)$ vorsieht.

[5] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt **wegabstrahiert**.
[6] Bzw. **Teilwörter** des Eingabeworts.

Der Grund, warum nicht einfach nur **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung von **Tokens** in **Tokentyp** T und **Tokenwert** V, ist, weil z.B. die **Bezeichner** von Variablen, Konstanten und Funktionen und auch **Zahlen** beliebige Folgen von Symbolen sein können. Später in der **Syntaktischen Analyse** in Unterkapitel 2.4 ist es nur relevant, ob an einer bestimmten Stelle ein bestimmter **Tokentyp** T, z.B. eine Zahl `NUM` steht und der **Tokenwert** V ist erst wieder in der **Code Generierung** in Unterkapitel 2.5 relevant.

Wie es in Tabelle 2.1 zu sehen ist, gibt es für verschiedene **Bezeichner**, wie z.B. `my_fun`, `my_var` oder `my_const` und verschiedene **Zahlen**, wie z.B. `42`, `314` oder `12` passende **Tokentypen** `NAME` und `NUM`[7] [8], die einen **Überbegriff** darstellen. Für **Lexeme**, wie `if` oder `}` sind die **Tokentypen** dagegen genau die Bezeichnungen, die man diesen beiden Folgen von Symbolen geben würde, nämlich `IF` und `RBRACE`.

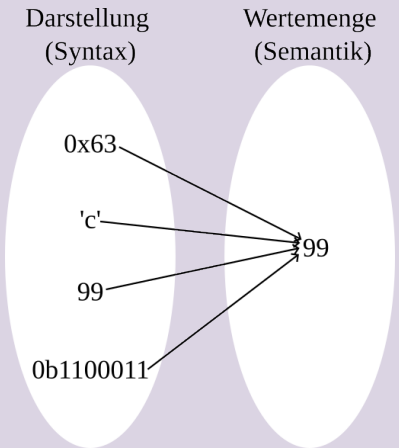
Lexeme	Token
`42`, `314`	`Token('NUM', '42')`, `Token('NUM', '314')`
`my_fun`, `my_var`, `my_const`	`Token('NAME', 'my_fun')`, `Token('NAME', 'my_var')`, `Token('NAME', 'my_const')`
`if`, `}`	`Token('IF', 'if')`, `Token('RBRACE', '}')`
`99`, `'c'`	`Token('NUM', '99')`, `Token('CHAR', '99')`

Tabelle 2.1: *Beispiele für Lexeme und ihre entsprechenden Tokens.*

Ein **Lexeme** ist nicht immer das gleiche, wie der **Tokenwert** V, denn wie in Tabelle 2.1 zu sehen ist, kann z.B. im Fall von L_{PicoC} der **Tokenwert** 99 durch zwei verschiedene **Literale** (Definition 2.32) dargestellt werden. Einmal kann der **Tokenwert** `99` als **ASCII-Zeichen** `'c'` dargestellt werden, das dann als **Tokenwert** den entsprechenden **Index** in der ASCII-Tabelle erhält, nämlich `99` und des Weiteren auch in **Dezimalschreibweise** als `99`[9]. Der **Tokenwert** ist der letztendlich verwendete **Wert** an sich, **unabhängig** von der Darstellungsform.

Definition 2.32: Literal

*Eine von möglicherweise vielen weiteren **Darstellungsformen**[a] für ein und denselben **Wert** eines **Datentyps**.*[b]



[a] Also verschiedene **Zeichenketten**.
[b] Thiemann, „Einführung in die Programmierung“.

[7] Bzw. wenn man sich **nicht Kurzformen** sucht, wären `IDENTIFIER` und `NUMBER` die **passenden Bezeichnungen**.
[8] Die **Bezeichnungen** der **Tokentypen** wurden im **PicoC-Compiler** so gewählt, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner haben will, damit unter anderem auch **mehr Code** in eine Zeile passt.
[9] Die Programmiersprache L_{Python} erlaubt es z.B. den **Wert** `99` auch mit den **Literalen** `0b1100011` und `0x63` darzustellen.

Die **Grammatik** für die Lexikalische Analyse G_{Lex} ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[10] und sich **nichts merkt**, also unabhängig davon, **was** für Symbole und **wie oft** bestimmte Symbole **davor** aufgetaucht sind funktioniert.

Der Grund, weshalb ein **Lexer** relativ **unkompliziert** zu implementieren ist, ist weil dieser nur **Lexeme** erkennen muss, welche durch eine **Reguläre Grammatik** beschrieben sind. **Parser**, welche im nächsten Unterkapitel 2.4 erklärt werden, sind dagegen deutlich **komplexer** zu implementieren, da diese bei den meisten Programmiersprachen eine **Kontextfreie Grammatik** umsetzen müssen[11].

Anmerkung

Auch für den **PicoC-Compiler** lässt sich aus der im **Dialekt der Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.4) spezifizierten Grammatik für die Lexikalische Analyse 3.1.1 G_{PicoC_Lex} schlussfolgern, dass diese **Grammatik** eine **Reguläre Grammatik** ist, da alle ihre **Produktionen** die Definition 2.19 einer **Regulären Grammatik** erfüllen.

Produktionen mit **Alternative**, wie z.B. DIG_WITH_0 ::= "0" | DIG_NO_0 in Grammatik 3.1.1 sind **unproblematisch**, denn sie können immer auch als {DIG_WITH_0 ::= "0", DIG_WITH_0 ::= DIG_NO_0} umgeschrieben werden und z.B. `DIG_WITH_0*`, `(LETTER | DIG_WITH_0 | "_")+` und "␣".."∼" in Grammatik 3.1.1 können alle zu **Alternativen** umgeschrieben werden, wie es in Definition 3.4 gezeigt wird und diese **Alternativen** können wie gerade gezeigt umgeformt werden, um ebenfalls **regulär** zu sein. Somit existiert mit der Grammatik 3.1.1 eine **Reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} nach der **Chromsky Hierarchie** (Definition 2.18) **regulär**.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.4 die Lexikalische Analyse an einem Beispiel veranschaulicht.

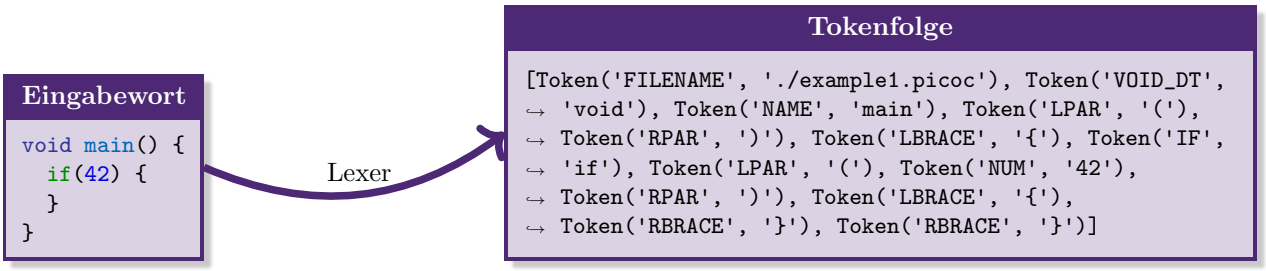


Abbildung 2.4: *Veranschaulichung der Lexikalischen Analyse.*

Anmerkung

Das Symbol ↪ zeigt in **Codebeispielen** einen **Zeilenumbruch** an, wenn eine **Zeile zu lang** ist.

2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele

[10] Man nennt das auch einem **Lookahead** von 1.

[11] Hierzu kann z.B. der **Earley Erkenner** in Definition 5.11 im Appendix als Beispiel genannt werden, der ein Bestandteil eines **Earley Parsers** (Definition 3.6) ist und bereits deutlich **komplexer** ist als ein typischer **Lexer**.

öffnende runde Klammern `(` bzw. öffnende geschweifte Klammern `{` es zu einem bestimmten Zeitpunkt gibt, die noch nicht durch eine entsprechende schließende runde Klammer `)` bzw. schließende geschweifte Klammer `}` geschlossen wurden. Diese syntaktischen Mittel lassen sich nicht mehr mit einer **Regulären Grammatik** (Definition 2.19) beschreiben, sondern es braucht eine **Kontextfreie Grammatik** (Definition 2.20) hierfür, die es erlaubt zwischen **zwei Terminalsymbolen** ein **Nicht-Terminalsymbol** abzuleiten.

Anmerkung

Für den **PicoC-Compiler** lässt sich aus der im **Dialekt der Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.4) spezifizierten Grammatik für die Syntaktische Analyse 3.2.8 G_{PicoC_Parse} schlussfolgern, dass diese eine **Kontextfreie Grammatik**, aber **keine Reguläre Grammatik** ist, da alle ihre Produktionen die Definition 2.20 einer **Kontextfreien Grammatik** erfüllen, aber **nicht** die Definition 2.19 einer **Regulären Grammatik**.

Es lässt sich sehr leicht erkennen, dass die Grammatik 3.2.8 eine **Kontextfreie Grammatik** ist, da **alle Produktionen** auf der **linken Seite** des `::=`-Symbols immer nur ein einzelnes **Nicht-Terminalsymbol** haben und sich auf der **rechten Seite** eine **beliebige Folge** von **Grammatiksymbolen**[a] befindet.

Es lässt sich wiederum sehr einfach erkennen, dass die Grammatik 3.2.8 **keine Reguläre Grammatik** ist, denn z.B. bei der Produktion *if_stmt* ::= "*if*""("*logic_or*")" *exec_part* ist das **Nicht-Terminalsymbol** *logic_or* von den **Terminalsymbolen** für eine **öffnende Klammer** { und eine **schließende Klammer** } eingeschlossen, was mit einer Regulären Grammatik **nicht** ausgedrückt werden kann.

Somit existiert mit der Grammatik 3.2.8 eine **Kontextfreie Grammatik**, die allerdings **keine Reguläre Grammatik** ist, welche die **Sprache** L_{PicoC_Parse} beschreibt. Hierdurch ist die **Sprache** L_{PicoC_Parse} nach der **Chromsky Hierarchie** (Definition 2.18) **kontextfrei**, aber **nicht regulär**.

[a] Also eine **beliebige Folge** von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

Die **Syntax**, in welcher ein **Programm** vor dem kompilieren in einer Textdatei aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.33) bezeichnet. In einem Zwischenschritt, dem **Parsen**, wird aus diesem Programm mithilfe eines **Parsers** (Definition 2.36) ein **Ableitungsbaum** (Definition 2.35) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 2.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 2.33: Konkrete Syntax

*Bezeichnet alles, was mit dem **Aufbau** von Wörtern[a] zu tun hat, die nach einer **Konkreten Grammatik** $G_{Lex} \uplus G_{Parse}$ (Definition 2.34) abgeleitet wurden.*

*Die **Konkrete Syntax** ist die Teilmenge der **gesamten Syntax** einer Sprache, welche für die **Lexikalische** und **Syntaktische Analyse** relevant ist. In der **gesamten Syntax** einer Sprache[b] kann es z.B. Wörter geben, welche die gesamte Syntax **nicht einhalten**, die allerdings **korrekt** nach einer **Konkreten Grammatik** abgeleitet sind[c].*

*Ein **Programm**, wie es in einer **Textdatei** nach der Konkreten Grammatik[d] abgeleitet steht, bevor man es kompiliert, ist in **Konkreter Syntax** aufgeschrieben.[e]*

[a] Bzw. **Programmen**.
[b] Vor allem bei **Programmiersprachen**.
[c] Wenn ein Programm z.B. **nicht deklarierte Variablen** hat und aufgrund dessen **nicht kompiliert** werden kann, hält dieses die gesamte Syntax **nicht** ein, kann allerdings so nach einer **Konkreten Grammatik** abgeleitet werden. Solche Details werden üblicherweise **nicht** in eine **Konkrete Grammatik** mitaufgenommen.
[d] **Vereinigungsgrammatik**, wie in Definition 2.17 erklärt.

[e] G. Siek, *Essentials of Compilation*.

Um einen kurzen Begriff für die **Grammatik** zu haben, welche die **Konkrete Syntax** einer Sprache beschreibt, wird diese im Folgenden als **Konkrete Grammatik** (Definition 2.34) bezeichnet.

Definition 2.34: Konkrete Grammatik

*Grammatik, welche die **Konkrete Syntax** einer Sprache beschreibt und die Grammatiken G_{Lex} und G_{Parse} miteinander vereinigt: $G_{Lex} \uplus G_{Parse}$*[a].

[a] **Vereinigungsgrammatik**, wie in Definition 2.17 erklärt.

Definition 2.35: Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)

***Compilerinterne Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 2.25) eines in **Konkreter Syntax** geschriebenen Programmes.*

*Die **Blätter**, die beim Formalen Ableitungsbaum das **leere Wort** ε oder **Terminalsymbole** Σ einer Konkreten Grammatik $G = \langle N, \Sigma, P, S \rangle$ sein können, sind in dieser Datenstruktur **Tokens** (T, W) und die **Inneren Knoten** entsprechen weiterhin **Nicht-Terminalsymbolen** N einer Konkreten Grammatik $G = \langle N, \Sigma, P, S \rangle$. In dieser Datenstruktur wird allerdings nur der Teil eines Formalen Ableitungsbaumes dargestellt, der den **Ableitungen** einer Grammatik G_{Parse} entspricht. Die **Tokens** sind in der Syntaktischen Analyse ein **atomarer Grundbaustein**[a], daher sind die **Ableitungen** der Grammatik G_{Lex} uninteressant.*[b]

[a] **Nicht** mehr weiter **teilbar**.
[b] *JSON parser - Tutorial — Lark documentation*.

Die **Konkrete Grammatik** nach der ein **Ableitungsbaum** konstruiert wird, ist optimalerweise immer so definiert, dass sich möglichst **einfach** aus dem **Ableitungsbaum** ein **Abstrakter Syntaxbaum** konstruieren lässt.

Definition 2.36: Parser

*Ein **Parser** ist ein Programm, dass aus einem **Eingabewort**[a], welches in **Konkreter Syntax** geschrieben ist eine compilerinterne Datenstruktur, den **Ableitungsbaum** generiert*[b]*. Dies wird auch als **Parsen** bezeichnet.*[c]

[a] Z.B. ein **Programm** in einer **Textdatei**.
[b] Es gibt allerdings auch **alternative Definitionen**, denen nach ein Parser ein Programm ist, dass ein Eingabewort von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die hiesige Definition 2.36 verwendet.
[c] *JSON parser - Tutorial — Lark documentation*.

Anmerkung

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines **Parsers** und ist ausschließlich für die **Lexikalische Analyse** verantwortlich. In einem alltäglicheren Szenario entspricht **lexen** z.B. bei einem Wanderausflug dem Nachschlagen in einem Insekten**lexikon** und dem Aufschreiben, welcher Insektenart man in welcher **Reihenfolge** begegnet ist, mit jeweils einem Bild des konkreten Exemplars, dem man begegnet ist. Zudem kann man bestimmte **Sehenswürdigkeiten**, an denen man während des Ausflugs vorbeikommt ebenfalls festhalten, da es eine Rolle spielen kann in welchem örtlichen **Kontext** man den Insekten begegnet ist[a].

Der **Parser** vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen**

Analyse in sich und entspricht, um auf das gerade angefangene Beispiel zurückzukommen, dem **Weiterverarbeiten** der **Beziehungen** zwischen den Insektenbegegnungen unter Berücksichtigung des **örtlichen Kontexts**[b] in eine **taugliche Form**[c].

In der Weiterverarbeitung könnte ein **Interpreter** die in eine taugliche Form gebrachten Daten interpretieren und daraus bestimmte **Schlüsse ziehen** und ein **Compiler** könnte die Daten vielleicht in eine für Menschen **leichter verständliche** Sprache kompilieren[d].

[a] Das würde z.B. der Rolle eines **Semikolon** `;` in der Sprache L_{PicoC} entsprechen.
[b] Das entspricht z.B. **Semikolons** `;`, die vorhin bereits als Beispiel genannt wurden.
[c] Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beinflussen sich gegenseitig und ihre Umwelt.
[d] Normalerweise kompiliert man in eine für die **CPU** verständliche Sprache.

Die vom **Lexer** aus dem Eingabewort generierten **Tokens** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welchem Kontext bestimmte **Tokens** (T, V) mit einem bestimmten **Tokentyp** T auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht, die zum Parsen eines Eingabeworts notwendig ist. Dabei wird in der Konkreten Grammatik G_{Parse} nach **Tokentypen** unterschieden und **nicht** nach **Tokenwerten**, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und **nicht**, welchen konkreten **Wert** diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Erkenner** (Definition 2.37), denn ein Parser löst das **Wortproblem** (Definition 2.21) für die **Sprache**, in welcher das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Erkennungsalgorithmus[12] gesammelt wurden den **Ableitungsbaum**.

Definition 2.37: Erkenner (bzw. engl. Recognizer)

*Entspricht in seiner Funktion einem **Kellerautomaten**[a], in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden.*

*Es ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit der **Konkreten Grammatik** einer Sprache ableiten lässt. Der Algorithmus überprüft also, ob das **Eingabewort** Teil der **Sprache** ist, die von der **Konkreten Grammatik** beschrieben wird oder **nicht**. Ein **Erkenner** löst folglich das **Wortproblem** (Definition 2.21).*[b]

[a] **Automat** mit dem **Kontextfreie Grammatiken** erkannt werden.
[b] Thiemann, „Compilerbau“.

Anmerkung

Für das **Parsen** gibt es grundsätzlich **drei** geläufige Ansätze, die unterschieden werden:

- **Top-Down Parsing:** Der Algorithmus arbeitet von **oben-nach-unten**, also anschaulich von der **Wurzel** zu den **Blättern** eines im Nachhinein oder parallel dazu generierten Ableitungsbaumes. Dementsprechend fängt der Algorithmus mit dem **Startsymbol** einer **Konkreten Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man die **Folge von Terminalsymbolen** hat, die sich zum gewünschten **Eingabewort** abgeleitet hat oder sich herausstellt, dass dieses **nicht** abgeleitet werden kann.[a]

  Der Grund, warum **Linksableitungen** verwendet werden und nicht z.B. **Rechtsableitungen**, ist, weil versucht wird das **Eingabewort** von **links nach rechts** mithilfe von Terminalsymbolen aus Ableitungen nachzubilden. Das passt gut damit zusammen, dass durch die **Linksableitung**

[12] Bzw. engl. recognition algorithm.

die Terminalsymbole von **links-nach-rechts** erzeugt werden.

Welche der **Produktionen** für ein **Nicht-Terminalsymbol** angewandt wird, wenn es mehrere **Alternativen** gibt, wird entweder durch **Backtracking** oder durch **Vorausschauen** entschieden. Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg** (Definition 5.8). Für diese Methode muss allerdings, wenn die Konkrete Grammatik eine **Linksrekursive Grammatik** (Definition 5.9) ist, diese umgeformt werden, um jegliche **Linksrekursion** aus dieser zu **entfernen**, da diese sonst zu **unendlicher Rekursion** führt.

Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Konkrete Grammatiken parsen zu können, die nicht **LL(k)** (Definition 5.10) sind. Dabei werden meist nach dem Prinzip der **Tiefensuche** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen, bis das gewünschte Eingabewort abgeleitet ist oder alle **Alternativen** für das momentane **Nicht-Terminalsymbol** abgesucht sind. Das wird solange durchgeführt, bis man wieder beim **Startsymbol** angekommen ist und da auch alle **Alternativen** abgesucht sind. Dies bedeutet dann, dass das **Eingabewort** sich **nicht** mit der verwendeten Konkreten Grammatik ableiten lässt.[b]

Wenn man eine **LL(k)**-Grammatik hat, ist **Backtracking** nicht notwendig und es reicht einfach nur immer k **Tokens** im Eingabewort **vorauszuschauen**[c]. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.[d]

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** einer Konkreten Grammatik **rückwärts** anzuwenden, bis man beim **Startsymbol** landet.[e]
- **Chart Parsing:** Es wird **Dynamische Programmierung** verwendet, indem **partielle Zwischenergebnisse** in einer **Tabelle**[f] gespeichert werden und **wiederverwendet** werden können. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist. **Chart Parser** können dabei **top-down** oder **bottom-up** Ansätze umsetzen[g] [h].

[a] *What is Top-Down Parsing?*
[b] Diese Methode des Parsens wurde im **PicoC-Compiler** implementiert, als dieser noch auf dem Stand des **Bachelorprojektes** war, bevor er durch den **nicht** selbst implementierten **Earley Parser** von **Lark** (siehe Webseite *Lark - a parsing toolkit for Python*) ersetzt wurde.
[c] Das wird auch als **Lookahead** von k bezeichnet.
[d] Diese Art von Parser ist im **RETI-Interpreter** implementiert, da die **RETI-Sprache** eine besonders simple **LL(1) Grammatik** besitzt. Diese Art von **Parser** wird auch als **Predictive Parser** oder **LL(k) Recursive Descent Parser** bezeichnet, wobei **Recursive Descent** das englische Wort für **Rekursiven Abstieg** ist.
[e] *What is Bottom-up Parsing?*
[f] Bzw. einem **Chart**.
[g] Da die **Implementierung** von **Chart Parsern** fundamental anders ist als bei **Top-Down** und **Bottom-Up Parsern**, wird diese **Kategorie** von Parsern nochmal **speziell unterschieden** und nicht gesagt, es sei ein **Top-Down Parser** oder **Bottom-Up Parser**, der **Dynamische Programmierung** verwendet.
[h] Der **Earley Parser** von **Lark**, welchen der **PicoC-Compiler** verwendet, fällt unter diese Kategorie.

Ein **Abstrakter Syntaxbaum** (Definition 2.42) wird durch einen **Transformer** (Definition 2.38) und **Visitors** (Definition 2.39) mithilfe eines **Ableitungsbaumes** generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese ein Programm von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 2.40).

Definition 2.38: Transformer

*Ein Programm, das von **unten-nach-oben**[a], nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaumes** besucht. Beim Antreffen eines bestimmten Knoten des **Ableitungsbaumes** erzeugt*

*es je nach Kontext einen entsprechenden Knoten des **Abstrakten Syntaxbaumes** und setzt diesen anstelle des Knotens des **Ableitungsbaumes** und konstruiert so Stück für Stück den **Abstrakten Syntaxbaum**.*[b]

[a] **Zur Erinnerung:** In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.
[b] *Transformers & Visitors — Lark documentation*.

Definition 2.39: Visitor

*Ein Programm, das von **unten-nach-oben**[a], nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaumes** besucht. Beim Antreffen eines bestimmten **Knoten** des Ableitungsbaumes **manipuliert** oder **tauscht** es diesen **in-place** mit anderen Knoten, um den Ableitungbaum für die weitere Verarbeitung durch z.B. einen Transformer zu **vereinfachen**[b].*[c]

[a] **Zur Erinnerung:** In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.
[b] Kann theoretisch auch zur Konstruktion eines **Abstrakten Syntaxbaumes** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakten Syntaxbaumes** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.
[c] *Transformers & Visitors — Lark documentation*.

Definition 2.40: Abstrakte Syntax

*Bezeichnet alles, was mit dem **Aufbau** von **Abstrakten Syntaxbäumen** zu tun hat.*

*Ein **Abstrakter Syntaxbaum**, der zur Kompilierung eines Eingabewortes*[a] *generiert wurde, befindet sich in **Abstrakter Syntax**.*[b]

[a] Z.B. ein **Programm** in einer **Textdatei**.
[b] G. Siek, *Essentials of Compilation*.

Um einen kurzen Begriff für die **Grammatik** zu haben, welche die **Abstrakte Syntax** einer Sprache beschreibt, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 2.41) bezeichnet.

Definition 2.41: Abstrakte Grammatik

*Grammatik, welche eine **Abstrakte Syntax** beschreibt, also beschreibt was für Arten von **Kompositionen** mit den **Knoten** eines **Abstrakten Syntaxbaumes** möglich sind.*

*Jene Produktionen, welche Produktionen in der Konkreten Grammatik entsprechen und in der **Konkreten Grammatik** für die Umsetzung von **Präzedenz** notwendig waren, sind in der **Abstrakten Grammatik** abgeflacht. Hierdurch sind die **Kompositionen**, welche die Knoten im **Abstrakten Syntaxbaum** bilden können **syntaktisch** meist näher an der Syntax von **Maschinenbefehlen**.*

Definition 2.42: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

*Ist eine **compilerinterne Datenstruktur**, welche eine **Abstraktion** eines **Ableitungsbaumes** darstellt. In den Aufbau des **Abstrakten Syntaxbaumes** ist das Erfordernis eines **leichten Zugriffs** und einer **leichten Weiterverarbeitbarkeit** eingeflossen ist. Bei der Betrachtung eines **Knoten**, der zusammen mit seinen Kinderknoten für einen Teil eines Eingabewortes*[a] *steht, soll man möglichst schnell die Fragen beantworten können, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.*

*Die Knoten des **Abstrakten Syntaxbaumes** enthalten dabei verschiedene **Attribute**, welche wichtige Informationen für den **Kompiliervorgang** und **Fehlermeldungen** enthalten.*[b]

[a] Z.B. ein **Programm** in einer **Textdatei**.
[b] G. Siek, *Essentials of Compilation*.

Anmerkung

In dieser Bachelorarbeit wird häufig von **„der Abstrakten Syntax“**, **„der Abstrakten Grammatik“** bzw. **„dem Abstrakten Syntaxbaum einer Sprache L“** gesprochen. Gemeint ist hier mit der Sprache L **nicht** die Sprache, welche durch die **Abstrakte Grammatik**[a] erzeugt wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll[b] und zu deren Zweck der **Abstrakte Syntaxbaum** überhaupt konstruiert bzw. die **Abstrakte Grammatik** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Grammatik** beschrieben wird, interessiert man sich nie wirklich explizit. Die Sprache L besitzt eine **Konkrete** und eine **Abstrakte Syntax** mit entsprechenden **Grammatiken** und **Bäumen**, die man zu beiden **definieren** bzw. **ableiten**[c] kann. Diese **Konvention** wurde aus dem Buch G. Siek, *Essentials of Compilation* übernommen.

[a] Nach welcher der **Abstrakte Syntaxbaum** konstruiert ist.
[b] Bzw. es ist die **Sprache**, welche durch die **Konkrete Grammatik** beschrieben wird.
[c] Bzw. **konstruieren** beim **Abstrakten Syntaxbaum**.

Im **Abstrakten Syntaxbaum** können theoretisch auch die **Tokens** aus der **Lexikalischen Analyse** weiterverwendet werden, allerdings ist dies **nicht empfehlenswert**. Es ist zum empfehlen die **Tokens** durch **eigene** entsprechende **Knoten** zu ersetzen, damit der **Zugriff** auf Knoten des Abstrakten Syntaxbaumes immer **einheitlich** erfolgen kann und auch, da manche **Tokentypen** noch **nicht optimal benannt** sind.

In z.B. der Sprache L_{PicoC} werden manche Symbole mehrfach verwendet, wie z.B. das Symbol `'-'`, welches für die **binäre Subtraktionsoperation** als auch die **unäre Minusoperation** verwendet wurde. Der verwendete **Tokentyp** dieses Symbols lautet beim **PicoC-Compiler** `SUB_MINUS`. Da in der **Syntaktischen Analyse** beide Operationen nur in **bestimmten Kontexten** vorkommen, **lassen** sie sich **unterscheiden** und dementsprechend können für beide Operationen jeweils zwei **unterschiedlich benannte Knoten** erstellt werden. Im Fall des **PicoC-Compilers** sind es die Knoten `Sub()` und `Minus()`.

Im Gegensatz zum **Formalen Ableitungsbaum**, ergibt es beim **Abstrakten Syntaxbaum** keinen Sinn zusätzlich einen **Formalen Abstrakten Syntaxbaum** zu unterschieden, da das Konzept eines **Abstrakten Syntaxbaumes** ohne eine Datenstruktur zu sein für sich allein gesehen keine Anwendung hat. Wenn von Abstrakten Syntaxbäumen die Rede ist, ist immer eine **Datenstruktur** gemeint.

Die **Abstrakte Grammatik** nach der ein **Abstrakter Syntaxbaum** konstruiert ist, wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten[13] möglichst **einfach weiterverarbeitet** werden kann.

Auf der **linken** Seite in Abbildung 2.5 wird das Beispiel 2.25.3 aus Unterkapitel 2.2.1 fortgeführt. Dieses Beispiel stellt die **Ableitung** des **Arithmetischen Ausdruck** `4 * 2` nach der Konkreten Grammatik 2.2[14] in einem **Ableitungsbaum** dar. Die Konkrete Grammatik 2.2 berücksichtigt die **höhere Präzedenz** der **Multipikation** $*$. Allerdings handelt es sich bei diesem Ableitungsbaum **nicht** um einen **Formalen Ableitungsbaum**, sondern um eine **compilerinterne Datenstruktur** für einen solchen[15]. Die **Blätter** sind dementsprechend **Tokens**, die mithilfe der Grammatik L_{Lex} generiert wurden. Der **Ableitungsbaum** beschränkt sich somit auf den Teil der **Ableitung**, der sich aus der Grammatik L_{Parse} ergibt.

Auf der **rechten** Seite in Abbildung 2.5 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**, der nach der Abstrakten Grammatik 2.3 konstruiert ist. In der **Abstrakten Grammatik** 2.3

[13] Den verschiedenen **Passes**.
[14] Die **Konkrete Grammatik** ist hierbei im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.4) angegeben.
[15] Die **Baumdarstellung** wird nur zur **Veranschaulichung** genutzt.

sind jegliche Produktionen **wegabstrahiert**, die in der **Konkreten Grammatik** 2.2 den Zweck erfüllen die **Präzidenz** umzusetzen und **mehrdeutig** zu verhindern. Aus diesem Grund gibt es nur noch einen **allgemeinen Knoten** für **binäre Operationen** $BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$.

NUM	::=	"4" | "2"	*L_Lex*
ADD_OP	::=	"+"	
MUL_OP	::=	"∗"	
mul	::=	*mul MUL_OP NUM* | *NUM*	*L_Parse*
add	::=	*add ADD_OP mul* | *mul*	

Grammatik 2.2: *Produktionen für Ableitungsbaum in EBNF*

bin_op	::=	$Add()$ | $Mul()$	
exp	::=	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$ | $Num(\langle str\rangle)$	

Grammatik 2.3: *Produktionen für Abstrakten Syntaxbaum in ASF*

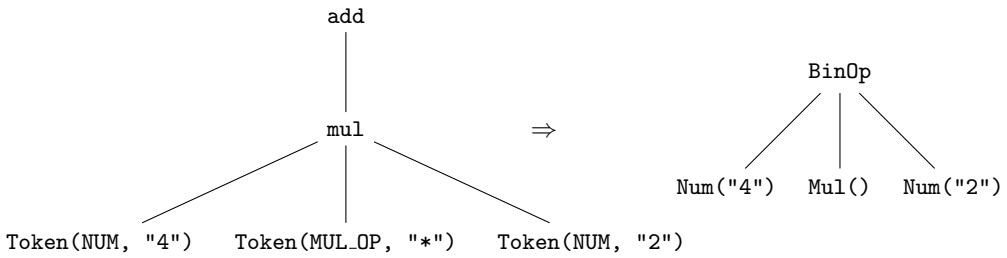


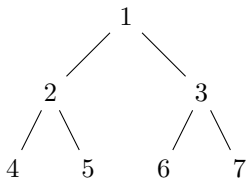
Abbildung 2.5: *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

Die **Baumdatenstruktur** des **Ableitungsbaumes** erfüllt den Zweck, dass sich aus dieser durch Iteration über diese besonders einfach ein **Abstrakter Syntaxbaum** konstruieren lässt, da auf diese Weise der Ableitungsbaum und Abstrakte Syntaxbaum, beide durch eine **Baumdatenstruktur** umgesetzt sind[16]. Beim **Abstrakten Syntaxbaum** ermöglicht die **Baumdatenstruktur** es die Operationen, die bei einem Compiler bzw. einem Interpreter bei der Weiterverarbeitung auszuführen sind, möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche auszuführen sind.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, sind in Abbildung 2.7 die einzelnen **Zwischenschritte** von den **Tokens** der Lexikalischen Analyse zum **Abstrakten Syntaxbaum** anhand des fortgeführten Beispiels aus Abbildung 2.4 und Unterkapitel 2.3 veranschaulicht. In Abbildung 2.7 werden die Darstellungen des **Ableitungsbaumes** und des **Abstrakten Syntaxbaumes** verwendet, wie sie vom **PicoC-Compiler** ausgegeben werden.

In der Darstellung von Bäumen beim **PicoC-Compiler**, stellen die verschiedenen **Einrückungen** die verschiedenen **Ebenen** dieser Bäume dar. Kanten gibt es **keine**, diese müssen sich zwischen den Knoten **dazugedacht** werden. Diese dazuzudenkenden Kanten bestehen immer **zwischen** einem **Knoten** und den **darauffolgenden Knoten**, die um eine Ebene **eingerückt** sind, bis vor den nächsten Knoten mit der selben **Einrückung**. Diese Bäume **wachsen** von **links-nach-rechts**, von der **Wurzel** zu den **Blättern**. In Abbildung 2.6 ist zur Veranschaulichung an einem **Beispielbaum** dargestellt, wie dieser in der Darstellung des **PicoC-Compilers** aussieht.

[16] Und zwischen **gleichen Datenstrukturen** ist es einfacher ineinander umzuformen.



⇒

Baum in der Darstellung des PicoC-Compilers

```
1
  2
    4
    5
  3
    6
    7
```

Abbildung 2.6: *Veranschaulichung eines Baumes in der Darstellung des PicoC-Compilers.*

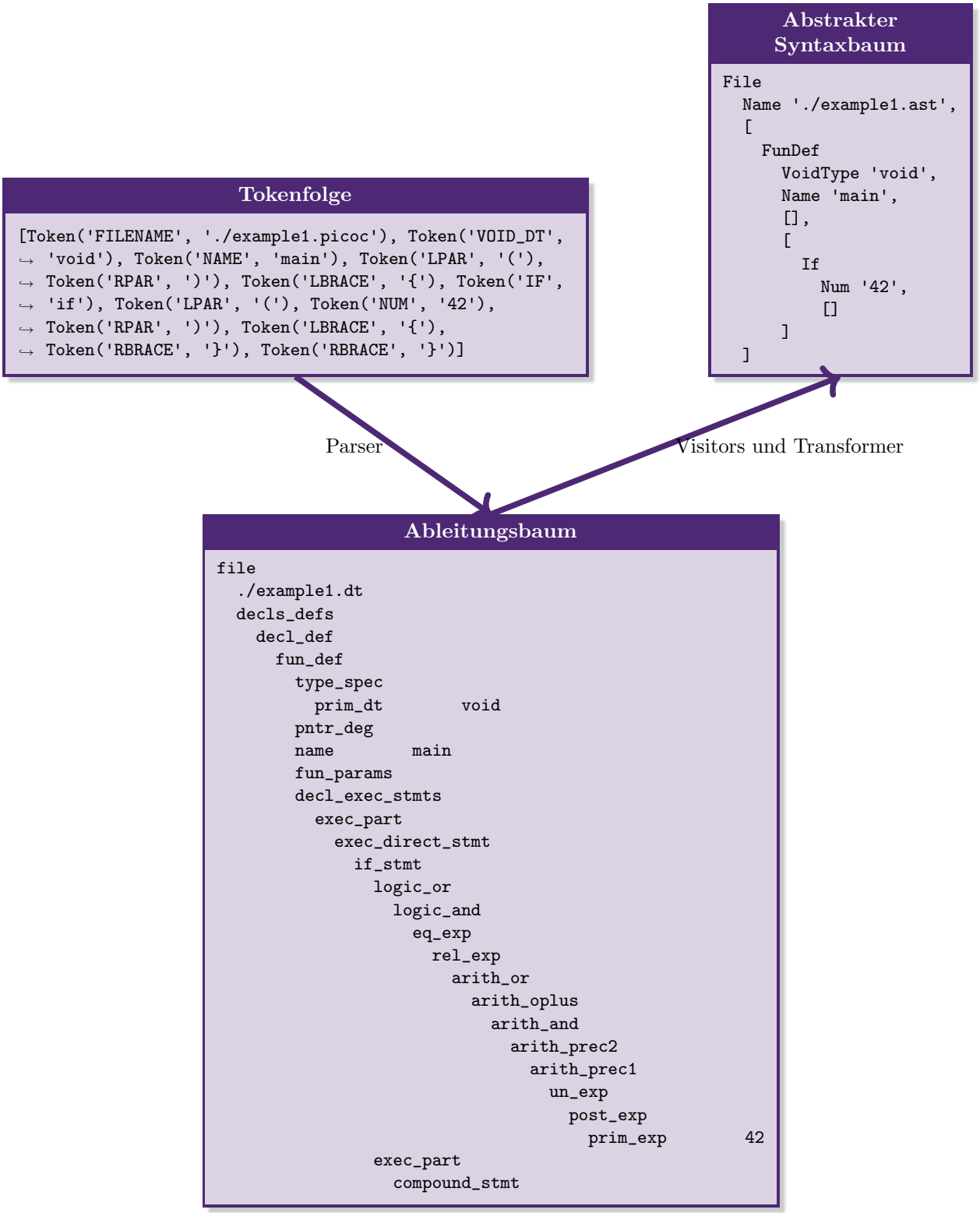


Abbildung 2.7: *Veranschaulichung der Syntaktischen Analyse.*

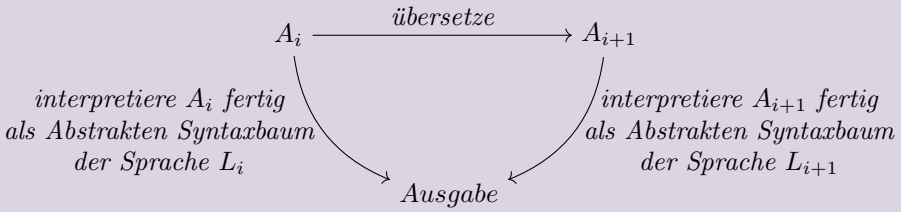
2.5 Code Generierung

In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** einer Sprache L_1 in den **Abstrakten Syntaxbaum** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.43) nennt. Dieses Vorgehen wurde auch schon beim **Ableitungsbaum** in der Syntaktischen Analyse angewandet, den man als **Zwischenstufe** zum **Abstrakten Syntaxbaum** kontstruiert hatte, um nicht direkt mithilfe der **Tokens** einen **Abstrakten Syntaxbaum** konstruieren zu müssen. Aus dem **Ableitungsbaum** kann dann unkompliziert und einfach mit **Transformern** und **Visitors** in mehreren Schritten ein **Abstrakter Syntaxbaum** generiert werden. Genauso kann mithilfe der **Passes** kleinschrittig die Syntax der Sprache L_2 erreicht werden.

Definition 2.43: Pass

Einzelner Übersetzungsschritt *in einem* ***Kompiliervorgang*** *von einem* ***Abstrakten Syntaxbaum*** A_i *einer Sprache* L_i *zu einem* ***Abstrakten Syntaxbaum*** A_{i+1} *einer Sprache* L_{i+1}*. Ein Pass übernimmt meist* ***eine*** *bestimmte* ***Teilaufgabe****, die sich mit* ***keiner Teilaufgabe*** *eines anderen Passes* ***überschneiden*** *sollte.*[a]

Für ***jeden Pass*** *gelten ähnlich, wie bei einem* ***vollständigen Compiler*** *(vergleiche mit Abbildung 2.3.1) die Beziehungen in Abbildung 2.43.1.*



(2.43.1)

Hierbei tut man so, als gäbe es zwei ***Interpreter*** *für die zwei Sprachen* L_i *und* L_{i+1}*, welche den jeweiligen* ***Abstrakten Syntaxbaum*** A_i *bzw.* A_{i+1} ***„fertig interpretieren“***[b]*.*[c]

[a] Der Begriff **„Pass“** kommt aus dem Englischen von **„passing over“**, da der gesamte **Abstrakte Syntaxbaum** nach dem Anwenden eines Pass an den nächsten Pass **„weitergegeben“** wird.
[b] Hier wird von **„fertig interpretieren“** gesprochen, da beim **Interpretieren** immer von einem Programm in **Konkreter Syntax** ausgegangen wird, dass erst zu einem **Abstrakten Syntaxbaum** umgeformt werden muss und dieser **Abstrakte Syntaxbaum** muss dann **„fertig interpretiert“** werden.
[c] G. Siek, *Essentials of Compilation*.

Die von den **Passes** umgeformten **Abstrakten Syntaxbäume** sollten dabei mit jedem **Pass** der **Syntax** von **Maschinenbefehlen** immer ähnlicher werden, bis es schließlich nur noch **Knoten von Maschinenbefehlen** sind.

2.5.1 Monadische Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 2.44) und **Anweisung** (Definition 2.45) wichtig.

Definition 2.44: Ausdruck (bzw. engl. Expression)

Code, der eine ***semantische Bedeutung*** *hat und in einem bestimmten* ***Kontext ausgewertet*** *werden kann, um einen* ***Wert*** *zu liefern oder etwas zu* ***deklarieren****.*

*Aufgebaut sind **Ausdrücke** meist aus **Kostanten**, **Variablen**, **Funktionsaufrufen**, **Operatoren** usw.*[a][b]

[a] Ein **Ausdruck** ist z.B `21 * 2;`.
[b] G. Siek, *Essentials of Compilation*.

Definition 2.45: Anweisung (bzw. engl. Statement)

*Code, der eine **Vorschrift** darstellt, die ausgeführt werden soll und **als ganzes keinen Wert** liefert und **nichts deklariert**. Eine Anweisung kann sich jedoch aus ein oder mehreren **Ausdrücken** zusammensetzen, die dies tun.*

*Anweisungen sind zentrale Elemente **Imperativer Programmiersprachen**, die sich zu einem großen Teil aus **Folgen von Anweisungen** zusammensetzen.*

*In **Maschinensprachen** werden **Anweisungen** häufig als **Befehle** bezeichnet.*[a][b]

[a] Eine **Anweisung** ist z.B `int var = 21 * 2;`.
[b] G. Siek, *Essentials of Compilation*.

Hat man es mit einer Programmiersprache zu tun, deren Programme **Unreine Anweisungen** (Definition 2.47) besitzen, so ist es sinnvoll einen **Pass** einzuführen, der **Unreine Ausdrücke** von den Anweisungen **trennt**, damit diese zu **Reinen Anweisungen** (Definition 2.46) werden. Das wird erreicht, indem man aus jedem Unreinen Ausdruck einen **vorangestellten Ausdruck** macht, den man **vor** die jeweilige Anweisung setzt, mit welcher der Unreine Ausdruck **gemischt** war. Der **Unreine Ausdruck** muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, den ein **Unreiner Ausdruck** hat die **Reine Anweisung**, mit der er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 2.46: Reiner Ausdruck / Reine Anweisung (bzw. engl. pure expression)

*Ein **Reiner Ausdruck** ist ein Ausdruck, der **rein** ist. Das bedeutet, dass dieser Ausdruck **keine Nebeneffekte** erzeugt. Ein **Nebeneffekt** ist eine **Bedeutung**, die ein Ausdruck hat, die sich **nicht** mit **Maschinencode** darstellen lässt. Sondern z.B. **intern** etwas am weiteren **Kompiliervorgang** ändert*[a].

*Eine **Reine Anweisung** ist eine Anweisung, bei der **alle Ausdrücke** aus denen sich die Anweisung unter anderem zusammensetzt **rein** sind.*[b]

[a] Z.B. ist die **Allokation von Variablen** `int var` **kein Reiner Ausdruck**. Eine Allokation bestimmt den Wert einiger **Immediates** im finalen **Maschinencode**, aber entspricht **keiner Folge** von **Maschinenbefehlen**.
[b] G. Siek, *Essentials of Compilation*.

Definition 2.47: Unreiner Ausdruck / Unreine Anweisung

*Ein **Unreiner Ausdruck** ist ein Ausdruck, der **kein Reiner Ausdruck** ist.*

*Eine **Unreine Anweisung** ist eine Anweisung, bei der **mindestens** einer der **Ausdrücke** aus denen sich die Anweisung unter anderem zusammensetzt **unrein** ist.*[a]

[a] G. Siek, *Essentials of Compilation*.

Auf diese Weise sind alle **Anweisungen** in **Monadischer Normalform** (Definiton 2.48).

Definition 2.48: Monadische Normalform (bzw. engl. monadic normal form)

*Code ist in **Monadischer Normalform**, wenn dieser nach einer Grammatik in **Monadischer Normalform** abgeleitet wurde.*

*Eine **Konkrete Grammatik** ist in **Monadischer Normalform**, wenn **alle** ableitbaren Anweisungen **rein** sind. Oder sehr **allgemein** ausgedrückt, wenn **Reines** und **Unreines** klar voneinander getrennt ist.*[a]

*Eine **Abstrakte Grammatik** ist in **Monadischer Normalform**, wenn die **Konkrete Grammatik** für welche sie definiert wurde in **Monadischer Normalform** ist.*

[a] G. Siek, *Essentials of Compilation*.

Ein **Beispiel** für das Vorgehen, Code in die **Monadische Normalform** zu bringen, ist in Abbildung 2.8 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser schriftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 2.8 sind daher in **Konkreter Syntax**[17] aufgeschrieben.

Links in der Abbildung 2.8 ist der Ausdruck mit dem **Nebeneffekt**, eine Variable zu **definieren**: `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt: `int var = 5 % 4`. Der **Unreine** Definitionsausdruck `int var` muss daher **vorangestellt** werden, wie es **rechts** in Abbildung 2.8 dargestellt ist[18].

Abbildung 2.8: *Codebeispiel dafür Code in die Monadische Normalform zu bringen.*

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in **Maschinenbefehlen** haben. Somit wird eine **Seperation** von Subbäumen, die keine Entsprechung in **Maschinenbefehlen** haben und denen, die eine haben bewerkstelligt wird. Eine **Reine Anweisung** ist **Maschinenbefehlen** ähnlicher als eine **Unreine Anweisung**. Somit sparrt man sich in der Implementierung **Fallunterscheidungen**, indem **Reine Ausdrücke** und **Reine Anweisungen** direkt in **Maschinenbefehle** übersetzt werden können und **nicht** unterschieden werden muss, ob darin **Unreine Ausdrücke** vorkommen.

2.5.2 A-Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 2.44) und **Anweisung** (Definition 2.45) wichtig.

Eine Programmiersprache L_1 soll in eine Maschinensprache L_2 kompiliert werden. Im Falle dessen, dass es sich bei einer **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um eine **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.51) in

[17] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[18] Obwohl hinter `int var` ein `;` steht, ist es immer noch ein **Ausdruck**. Allerdings gibt es **keine** einheitliche Festlegung, was eine **Anweisung** ist und was **nicht**, es wurde für diese schriftliche Ausarbeitung der Bachelorarbeit nur so definiert.

Anweisungen und **Ausdrücken** verhindert. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Ausdrücke macht, in denen die **Komplexen Ausdrücke temporären Locations** (Definiton 2.49) zugewiesen werden und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Definition 2.49: Location

*Kollektiver Begriff für **Variablen**, **Attribute** bzw. **Elemente** von Variablen bestimmter Datentypen, **Speicherbereiche auf dem Stack**, die **temporäre Zwischenergebnisse** speichern und **Register**.*

*Im Grunde genommen **alles**, was mit einem **Programm zu tun** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.*[a]

[a] G. Siek, *Essentials of Compilation*.

Sollte der **Komplexe Ausdruck**, welcher einer **temporären Location** zugewiesen wird, **Teilausdrücke** enthalten, die **komplex** sind, muss das gleiche Vorgehen erneut für die **Teilausdrücke** angewandt werden, bis alle **Komplexen Ausdrücke** nur noch **Atomare Ausdrücke** (Definiton 2.50) enthalten, falls sie sich überhaupt in **weitere Teilausdrücke** aufteilen lassen.

Sollte es sich bei dem **Komplexen Ausdruck** um einen **Unreinen Ausdruck** handeln, welcher nur einen **Nebeneffekt** ausführt und sich nicht in **Maschinenbefehle** übersetzen lässt, so wird aus diesem ein **vorangestellter Ausdruck** gemacht, welcher einfach nur den **Nebeneffekt** dieses **Unreinen Ausdrucks** ausführt und keiner **temporären Location** zugewiesen wird.

Definition 2.50: Atomarer Ausdruck

*Ein **Atomarer Ausdruck** ist ein **Reiner Ausdruck** (Definition 2.46), der **keinem kompletten Maschinenbefehl** entspricht, sondern nur ein **Argument**, wie z.B. einen **Immediate** in einer Folge von Maschinenbefehlen festlegt.*[a]

[a] G. Siek, *Essentials of Compilation*.

Bei einem **üblichen Compiler**, bei dem z.B. **Register** für **temporäre Zwischenergebnisse** genutzt werden und der **Maschinenbefehlssatz** es erlaubt **zwei Register** miteinander zu verechnen[19] sind **Atomare Ausdrücke** z.B. eine **Variable** (z.B. `var`), eine **Zahl** (z.B. `12`) oder ein **ASCII-Zeichen** (z.B. `'c'`), da diese häufig direkt über **Register** zugreifbar sind, die direkt mit einem **Maschinenbefehl** verechnet werden können[20] [21].

Im Fall des **PicoC-Compilers** ist ein **Zugriff auf eine Location** (z.B. `stack(i)`) der einzige **Atomare Ausdruck**, da der **PicoC-Compiler** so umgesetzt ist, dass er alle **Zwischenergebnisse** auf dem **Stack** speichert und dort dann auf diese zugreift, um sie in **Register** zu laden und miteinander zu verechnen[22]. Aus diesem Grund braucht es mindestens einen Maschinenbefehl[23], um z.B. eine Zahl überhaupt für einen **Maschinenbefehl** zugreifbar zu machen, was der Definition 2.50 widerspricht. Daher sind z.B. Zahlen beim **PicoC-Compiler** keine **Atomaren Ausdrücke**.

[19] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[20] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.
[21] Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse auf den **Stack** zu speichern und ein **Stackpointer Register** anzupassen.
[22] Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition 5.14) inklusive **Liveness Analysis** (Definition 5.12) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden. Beim **PicoC-Compiler** sollte sich an die **Paradigmen** aus der Vorlesung C. Scholl, „Betriebssysteme“ gehalten werden.
[23] Genauergesagt 4.

Definition 2.51: Komplexer Ausdruck

*Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist.*[a]

[a] G. Siek, *Essentials of Compilation*.

Im Fall des **PicoC-Compilers** sind **Komplexe Ausdrücke** z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`, da diese zur Berechnung auf jeden Fall mehrere **Maschinenbefehle** benötigen, was Definition 2.51 widerspricht oder **unrein** sind. Die Teilausdrücke `4`, `5`, `1` müssen erst auf den **Stack** geschrieben werden, um dann in **Register** geladen zu werden, damit dann der gesamte **Komplexe Ausdruck** berechnet werden kann. Die Ausdrücke `fun(12)` und `int var` sind **unrein** und daher **Komplexe Ausdrücke**.

In Abbildung 2.9 ist zur besseren Vorstellung die Einteilung von **Komplexen**, **Atomaren**, **Unreinen** und **Reinen** Ausdrücken veranschaulicht. Des Weiteren sind in der Abbildung alle Ausdrücke ausgegraut, welche die **Monadische Normalform** nicht erfüllen. Hierbei wird vom **PicoC-Compiler** ausgegangen, bei dem nur ein **Zugriff auf eine Location** (z.B. `stack(i)`) einen **Atomaren Ausdruck** darstellt.

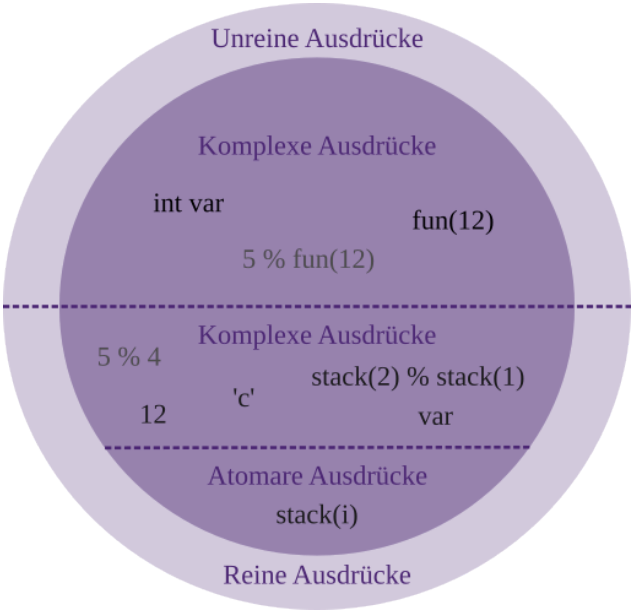


Abbildung 2.9: *Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke.*

Es wird in dem gerade beschriebenen **Pass** dafür gesorgt, dass alle **Anweisungen** und **Ausdrücke** in **A-Normalform**[24] (Definition 2.52) sind. Wenn eine **Konkrete Grammatik** in **A-Normalform** ist, ist diese per Defintition 2.52 auch automatisch in **Monadischer Normalform**, genauso, wie ein **Atomarer Ausdruck** nach Definition 2.50 auch ein **Reiner Ausdruck** ist.

Definition 2.52: A-Normalform (ANF)

*Code ist in **A-Normalform**, wenn dieser nach einer **Konkreten Grammatik** in **A-Normalform** abgeleitet ist.*

*Eine **Konkrete Grammatik** ist in **A-Normalform**, wenn sie in **Monadischer Normalform** (Definition 2.48) ist und wenn alle **Komplexen Ausdrücke** nur **Atomare Ausdrücke** enthalten,*

[24] Das **'A'** kommt vermutlich von **„atomar“** bzw. engl. „atomic“, weil alle **Komplexen Ausdrücke** nur noch **Atomare Ausdrücke** enthalten dürfen.

*falls sie sich überhaupt in **weitere Teilausdrücke** einteilen lassen.*

*Eine **Abstrakte Grammatik** ist in **A-Normalform**, wenn die **Konkrete Grammatik** für welche sie definiert wurde in **A-Normalform** ist.*[a b c]

[a] *A-Normalization: Why and How (with code)*.
[b] Bolingbroke und Peyton Jones, „Types are calling conventions“.
[c] G. Siek, *Essentials of Compilation*.

Ein **Beispiel** für dieses Vorgehen, Code in die **A-Normalform** zu bringen, ist in Abbildung 2.10 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser schriftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 2.8 sind daher in **Konkreten Syntax**[25] aufgeschrieben.

Um **konsistent** mit der Implementierung zu sein und später keine Verwirrung zu erzeugen, wird beim Beispiel in Abbildung 2.10 vom **PicoC-Compiler** ausgegangen, bei dem **Variablen** (z.B. `var`), **Zahlen** (z.B. `12`) oder **ASCII-Zeichen** (z.B. `'c'`) **Komplexe Ausdrücke** darstellen.

Die Ausdrücke `4;`, `x;`, usw. für sich sind in diesem Fall **Komplexe Ausdrücke**, deren Wert einer **Location**, in diesem Fall einer **Speicherzelle des Stack** zugewiesen werden. Auf das Ergebnis dieser **Komplexen Ausdrücke** wird mittels `stack(2)` und `stack(1)` zugegriffen, um diese in **Register** zu schreiben und dann z.B. im **Komplexen Ausdruck** `stack(2) % stack(1)` miteinander zu verrechnen und das Ergebnis wiederum einer **Location**, in diesem Fall ebenfalls einer **Speicherzelle des Stack** zuzuweisen. Dieses Ergebnis wird dann vom **Stack** `stack(1)` in die **Globalen Statischen Daten** `global(0)` gespeichert, wo die Variable `x` allokiert ist. Die Zahlen `0`, `1`, `2` sind dabei hierbei **relative Adressen** auf dem **Stack** und in den **Globalen Statischen Daten**.

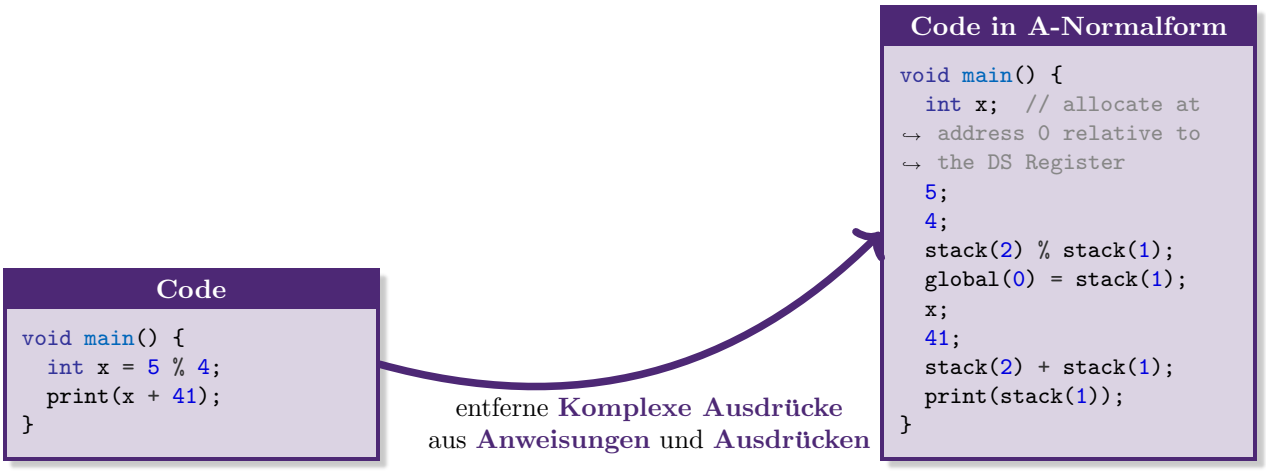


Abbildung 2.10: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.*

Ein **Pass**, wie er gerade beschrieben wurde hat vor allem in erster Linie die Aufgabe, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** besonders dadurch anzunähren, dass er die Anweisungen **weniger komplex** macht und diese dadurch den ziemlich **simplen Maschinenbefehlen** syntaktisch ähnlicher sind. Des Weiteren **vereinfacht** dieser Pass die **Implementierung** der nachfolgenden Passes enorm, indem weniger **Fallunterscheidungen** nötig sind, da Anweisungen wie z.B. **Zuweisungen** nur noch die Form `global(rel_addr) = stack(1)` haben, welche zudem viel **einfacher verarbeitet** werden kann.

[25] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

Alle weiteren denkbaren **Passes** sind zu **spezifisch** auf bestimmte **Anweisungen** und **Ausdrücke** ausgelegt, als das sich zu diesen allgemein etwas mit einer **Theorie** dahinter sagen lässt. Alle **Passes**, die zur Implementierung des **PicoC-Compilers** geplant und ausgedacht wurden sind im Unterkapitel 3.3.1 erklärt.

2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakten Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt[26]. Der Weg von der **Abstrakten Syntax** zur **Konkreten Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax** zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden muss.

Jeder **Knoten** des **Abstrakten Syntaxbaumes** erhält dazu eine Methode, welche hier `to_string` genannt wird, die eine **Textrepräsentation** seiner selbst und all seiner Knoten, mit an den richtigen Stellen passend gesetzten **Semikolons** `;`, **öffnenden-** und **schließenden runden Klammern** `()`, **öffnenden-** und **schließenden geschweiften Klammern** {} usw. ausgibt. Dabei wird der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die wiederum die Methode `to_string` ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgebeben**.

2.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**[27] eintritt:

1. In der **Lexikalischen** oder **Syntaktischen Analyse** tritt ein Fall ein, der **nicht** in der **Syntax** der Sprache des Compilers abgedeckt ist, z.B.:

   - Der **Lexer** kann ein Lexeme **nicht** mit der Konkreten Grammatik für die Lexikalische Analyse G_{Lex} **ableiten**. Der **Lexer** ist genaugenommen ein **Teil des Parsers** und ist damit bereits durch den nachfolgenden Punkt „Parser“ abgedeckt. Um die unterschiedlichen Ebenen, **Lexikalische** und **Syntaktische Analyse** gesondert zu betrachten wurde der Lexer an dieser Stelle ebenfalls kurz eingebracht.

   - Der **Parser**[28] entscheidet das **Wortproblem** (Definition 2.21) für ein **Eingabeprogramm**[29] mit `0`, also das **Eingabeprogramm** lässt sich **nicht** durch die Konkrete Grammatik $G_{Lex} \uplus G_{Parse}$ des Compilers **ableiten**.

2. In den **Passes** tritt ein Fall ein, der **nicht** in der **Syntax** der Sprache des Compilers abgedeckt ist, z.B.:

   - Eine **Variable** wird **verwendet**, obwohl sie noch **nicht deklariert** ist.

   - Bei einem **Funktionsaufruf** werden **mehr** oder **weniger** Argumente, Argumente des **falschen Datentyps** oder Argumente in der **falschen Reihenfolge** übergeben, als sie im **Funktionsprototyp** angegeben sind.

3. Während der **Laufzeit** des Compilers tritt ein Ereignis ein, das **nicht** durch die **Semantik** der Sprache

[26] Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben
[27] *Errors in C/C++ - GeeksforGeeks*.
[28] Bzw. der **Erkenner** innerhalb des Parsers.
[29] Bzw. ein **Wort**.

des Compilers abgedeckt ist oder welches das **Betriebssystem** nicht erlaubt, z.B.:

- Eine **nicht erlaubte Operation**, wie **Division durch** 0 (z.B. `42 / 0`) soll ausgeführt werden.
- **Segmentation Fault:** Wenn auf **Speicher zugegriffen** wird, der vom **Betriebssystem geschützt** ist.

oder wenn während des **Linkens** (Definition 5.6) etwas nicht zusammenpasst, wie z.B.:

- Es gibt **keine** oder **mehr als eine** `main`-Funktion.
- Eine Funktion, die in einer **Objektdatei** (Definition 5.5) benötigt wird, wird von **keiner** oder **mehr als einer** anderen Objektdatei bereitsgestellt.

wird eine **Fehlermeldung** (Definition 2.53) ausgegeben.

Definition 2.53: Fehlermeldung

Benachrichtigung *beliebiger Form, die einen* ***Grund*** *angibt, weshalb ein Programm* ***nicht weiter ausgeführt*** *werden kann*[a]*. Das* ***Ausgeben*** *bzw.* ***Übermitteln*** *einer Fehlermeldung kann dabei auf* ***verschiedene Weisen*** *erfolgen, wie z.B.:*

- *über* `stdout` *oder* `stderr` *im einem* ***Terminal Emulator*** *oder* ***richtigen Terminal****.*
- *über eine* ***Dialogbox*** *in einer* ***Graphischen Benutzeroberfläche***[b] *oder* ***Zeichenorientierten Benutzerschnittstelle***[c]*.*
- *über ein* ***Register*** *oder eine* ***spezielle Adresse*** *des* ***Hauptspeichers*** *mithilfe eines* ***Wertes****.*
- *über eine* ***Logdatei***[d] *auf einem* ***Speichermedium****.*

[a] Dieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler selbst kompiliert** hat.
[b] In engl. **G**raphical **U**ser **I**nterface, kurz **GUI**.
[c] In engl. **T**ext-based **U**ser **I**nterface, kurz **TUI**.
[d] In engl. **log file**.

3 Implementierung

In diesem Kapitel wird, nachdem im Kapitel 2 die nötigen **theoretischen Grundlagen** des **Compilerbau** vermittelt wurden, nun auf die **Implementierung** des **PicoC-Compilers** eingegangen. Aufgeteilt in die selben Kategorien **Lexikalische Analyse** 3.1, **Syntaktische Analyse** 3.2 und **Code Generierung** 3.3, wie in Kapitel 2, werden in den folgenden Unterkapiteln die einzelnen **Zwischenschritte** vom einem **Programm** in der **Konkreten Syntax** der Sprache L_{PicoC} hin zum einem Programm mit derselben **Semantik** in der **Konkreten Syntax** der Sprache L_{RETI} erklärt.

Für das Parsen[1] des Programmes in der **Konkreten Syntax** der Sprache L_{PicoC} wird das **Lark Parsing Toolkit**[2] [3] verwendet. Das **Lark Parsing Toolkit** ist eine Bibliothek, die es ermöglicht mittels einer in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** (Definition 3.3 bzw. für den Dialekt von Lark Definition 3.4) spezifizierten Konkreten Grammatik ein Programm in **Konkreter Syntax** zu parsen und daraus einen **Ableitungsbaum** für die kompilerintere Weiterverarbeitung zu generieren.

Definition 3.1: Metasyntax

*Steht für den **Aufbau** einer **Metasprache** (Definition 3.2), der durch eine **Grammatik** oder **Natürliche Sprache** beschrieben werden kann.*

Definition 3.2: Metasprache

*Eine **Sprache**, die dazu genutzt wird **andere Sprachen** zu **beschreiben**[a].*

[a] Das **„Meta“** drückt allgemein aus, dass sich etwas auf einer **höheren Ebene** befindet. Um über die Ebene sprechen zu können, in der man sich **selbst befindet**, muss man von einer **höheren, außenstehenden Ebene** darüber reden.

Definition 3.3: Erweiterte Backus-Naur-Form (EBNF)

*Die Erweiterte Backus-Naur-Form[a] ist eine **Metasyntax** (Definition 3.1), die dazu verwendet wird **Kontextfreie Grammatiken** darzustellen.*

*Am grundlegensten lässt sich die **Erweiterte Backus-Naur-Form** in **Kürze** wie folgt beschreiben:*[b] [c]

- ***Terminalsymbole** werden in **Anführungszeichen** "" geschrieben (z.B. `"term"`).*
- ***Nicht-Terminalsymbole** werden normal hingeschrieben (z.B. `non-term`).*
- ***Leerzeichen** ␣ dienen zur visuellen **Abtrennung** von **Grammatiksymbolen**[d].*

*Weitere Details sind in der **Spezifikation** des **Standards** unter Link[e] zu finden. Allerdings werden in der Praxis, wie z.B. in Lark oft eigene **abgewandelte Notationen** wie in Definition 3.4 verwendet.*

[1] Wobei beim **Parsen** auch das **Lexen** inbegriffen ist.
[2] *Lark - a parsing toolkit for Python*.
[3] Shinan, *lark*.

[a] Der Name kommt daher, dass es eine **Erweiterung** der **Backus-Naur-Form** ist, die hier allerdings **nicht** weiter erläutert wird.
[b] Nebel, „Theoretische Informatik“.
[c] *Grammar Reference — Lark documentation*.
[d] Also von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[e] `https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.

Definition 3.4: Dialekt der Erweiterten Backus-Naur-Form aus Lark

*Das **Lark Parsing Toolkit** verwendet eine **eigene Notation** für die **Erweiterte Backus-Naur-Form** (Definition 3.3), die sich teilweise in einzelnen Aspekten von der Syntax aus dem **Standard** unterscheidet und unter Link[a] dokumentiert ist.*

*Wichtige **Unterschiede** dieses Dialekts sind hierbei z.B.:*

- *für die Darstellung von **Optionaler Wiederholung** wird der aus **regulären Ausdrücken** bekannte `*`-Quantor zusammen mit optionalen **runden Klammern** `()` verwendet (z.B. `()*`).[b] Die Verwendung des `*`-Quantors kann wie in Umformung 3.4.1 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= b*\} \quad \Rightarrow \quad \{a ::= b\_tmp,\ \ b\_tmp ::= b\ \ b\_tmp \ \mid\ \varepsilon\} \tag{3.4.1}$$

- *für die Darstellung von **mindestents 1-Mal Wiederholung** wird der ebenfalls aus **regulären Ausdrücken** bekannte `+`-Operator zusammen mit optionalen **runden Klammern** `()` verwendet (z.B. `()+`). Die Verwendung des `+`-Quantors kann wie in Umformung 3.4.2 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= b+\} \quad \Rightarrow \quad \{a ::= b\ \ b\_tmp,\ \ b\_tmp ::= b\ \ b\_tmp \ \mid\ \varepsilon\} \tag{3.4.2}$$

- *für alle **ASCII-Symbole** zwischen z.B. ␣ und ∼ als **Alternative** aufgeschrieben kann auch die Abkürzung "␣".."∼" verwendet werden. Die Verwendung dieser Schreibweise kann wie in Umformung 3.4.3 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= "ascii1"..."ascii2"\} \quad \Rightarrow \quad \{a ::= "ascii1" \ \mid\ \ldots\ \mid\ "ascii2"\} \tag{3.4.3}$$

*Um bei einer **Produktion** auszudrücken, wozu die **linke Seite** abgeleitet werden kann, wird das `::=`-Symbol verwendet. Dieses Symbol wird als **„kann abgeleitet werden zu“** gelesen.*

[a] `https://lark-parser.readthedocs.io/en/latest/grammar.html`.
[b] Der `*`-Quantor bedeutet im Gegensatz zum `+`-Quantor auch **keinmal** wiederholen.

Das **Lark Parsing Toolkit** wurde vor allem deswegen gewählt, weil es sehr **einfach in der Verwendung** ist. Andere derartige Tools, wie z.B. **ANTLR**[4] sind **Parser Generatoren**, die zur **Konkreten Grammatik** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkreten Grammatik** direkt ein Programm in dieser Konkreten Grammatik **parsen** und einen **Ableitungsbaum** dafür generieren zu können. Lark besitzt des Weiteren eine **sehr gute Dokumentation** *Welcome to Lark's documentation! — Lark documentation*.

Neben den Konkreten Grammatiken, die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** spezifiziert sind, werden in den folgenden Unter-

[4] *ANTLR*.

kapiteln die **Abstrakten Grammatiken**, welche spezifzieren, welche Kompositionen für die **Abstrakten Syntaxbäume** der verschiedenden **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben. Diese Notation hat allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit**.

Die Notation für die **Abstrakte Syntax** unterscheidet sich bewusst von der **Erweiterten Backus-Naur-Form**, da in der Abstrakten Syntax **Kompositionen von Knoten** beschrieben werden, die **klar auszumachen** sind. Hierdurch würde die Abstrakten Grammatiken nur **unnötig verkompliziert**, wenn man die **Erweiterte Backus-Naur-Form** verwenden würde. Es gibt leider **keine** Standardnotation für **Abstrakte Grammatiken**, die sich deutlich durchgesetzt hat, daher wird für Abstrakte Grammatiken eine eigene **Abstrakte Syntaxform Notation** (Definition 3.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrete** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 3.5: Abstrakte Syntaxform (ASF)

*Ist eine eigene **Metasyntax** für **Abstrakte Grammatiken**, die für diese Bachelorarbeit definiert wurde. Sie unterscheidet sich vom **Dialekt** der **Backus-Naur-Form** des **Lark Parsing Toolkit** (Definition 3.4) nur durch:*

- ***Terminalsymbole** müssen nicht von "" engeschlossen sein, da die **Knoten** in der **Abstrakten Syntax** sowieso schon **klar auszumachen** sind und von anderen Symbolen der **Metasprache** leicht zu unterscheiden sind (z.B. `Node(<non-term>, <non-term>)`).*
- *dafür müssen allerdings **Nicht-Terminalsymbole** von <>-Klammern eingeschlossen sein (z.B. `<non-term>`).*

Letztendlich geht es nur darum, dass aufgrund der Verwendung des **Lark Parsing Toolkit** die **Konkrete Grammatik** in einem eigenen Dialekt der **Erweiterter Backus-Naur-Form** angegeben sein muss und für das Implementieren der Passes die **Abstrakte Grammatik** für den **Programmierer** möglichst **einfach verständlich** sein sollte, weshalb sich die **Abstrake Syntax Form** gut dafür eignet.

3.1 Lexikalische Analyse

Für die **Lexikalische Analyse** ist es nur notwendig eine **Konkrete Grammatik** zu definieren, die den Teil der **Konkreten Syntax** beschreibt, der für die **Lexikalische Analyse** wichtig ist. Diese Konkrete Grammatik wird dann vom **Lark Parsing Toolkit** dazu verwendet ein Programm in **Konkreter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel 2.3 erläutert ist.

3.1.1 Konkrete Grammatik für die Lexikalische Analyse

In der Konkreten Grammatik 3.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokentyp** oder einen **Teil der Beschreibung** des Aufbaus der zum **Tokentyp** gehörenden möglichen **Tokenswerte**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol `NUM` um einen **Tokentyp**, dessen zugeordnete **mögliche Tokenwerte** durch die **Produktion** `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` beschrieben werden. Diese **Produktionen** beschreiben, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann.

Die in der Konkreten Grammatik 3.1.1 für die **Lexikalische Analyse** definierten **Nicht-Terminalsymbole** können in der Konkreten Grammatik 3.2.8 für die **Syntaktischen Analayse** verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl `NUM` stehen darf.

Die in der Konkreten Grammatik vereinzelt **kleingeschriebenen** Nicht-Terminalsymbole, wie z.B. `name` haben nur den Zweck mehrere **Tokentypen**, wie z.B. `NAME | INT_NAME | CHAR_NAME` unter einem **Überbegriff** zu sammeln.

In Lark steht eine Zahl `.<number>`, die an ein **Nicht-Terminalsymbol** angehängt ist (z.B. `NONTERM.<number>`), dass auf der linken Seite des `::=`-Symbols einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. Es wird immer die Produktion mit der **höchste** Priorität, also der höchsten Zahl `<number>` **zuerst** genommen.

Es gibt den Fall, dass ein Wort von **mehreren Produktionen** erkannt wird, z.B. wird das Wort `int` sowohl von der Produktion `NAME`, als auch von der Produktion `INT_DT` erkannt. Daher ist es notwendig für `INT_DT` eine **Priorität** `INT_DT.2` zu setzen, damit das Wort `int` den **Tokentyp** `INT_DT` zugewiesen bekommt und nicht `NAME`.

Allerdings muss für den Fall, dass `int` der **Präfix** eines Wortes ist (z.B. `int_var`) noch die Produktion `INT_NAME.3` definiert werden, da der im **Lark Parsing Toolkit** verwendete **Basic Lexer** sobald ein Wort von einer Produktion erkannt wird, diesem direkt einen **Tokentyp** zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. Ansonsten würden aus `int_var` die Tokens `Token('INT_DT', 'int'), Token('NAME', '_var')` generiert, anstatt dem Token`Token(NAME, 'int_var')`. Daher muss die Produktion `INT_NAME.3` eingeführt werden, die immer zuerst geprüft wird. Wenn es sich nur um das Wort `int` handelt, wird zuerst die Produktion `INT_NAME.3` geprüft. Es stellt sich heraus, dass `int` von der Produktion `INT_NAME.3` nicht erkannt wird, daher wird als nächstes `INT_DT.2` geprüft, welches `int` erkennt.

Die Implementierung des **Basic Lexer** aus dem **Lark Parsing Toolkit** ist unter Link[5] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten und ist aufgrund dessen, dass sie in der Lage ist nach einer spezifizierten Konkreten Grammatik zu lexen, **zu komplex**, um sie an dieser Stelle allgemein erklären zu können.

Der **Basic Lexer** verhält sich allerdings grundlegend so, wie es im Unterkapitel 2.3 erklärt wurde, nur berücksichtigt der **Basic Lexer** ebenfalls **Priortiäten**, sodass für den aktuellen Index[6] im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Lexeme** an dem aktuellen Index im Eingabeprogramm ableiten kann, wird hieraus direkt ein **Token** mit dem entsprechenden **Tokentyp** dieser Produktion und dem abgeleiteten **Tokenwert** erstellt. Weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

[5] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.

[6] Ein **Lexer** bewegt sich über das Eingabeprogramm und erstellt, wenn ein **Lexeme** sich in der **Konkreten Grammatik** ableiten lässt ein **Token** und bewegt sich danach um die **Länge des Lexemes** viele Indices weiter.

COMMENT	::=	"//" /[^\n]*/ | "/*" /(. | \n)*?/ "*/"	*L_Comment*
RETI_COMMENT.2	::=	"//""␣"?"#"/[^\n]*/	
DIG_NO_0	::=	"1" | "2" | "3" | "4" | "5" | "6"	*L_Arith_Bit*
	|	"7" | "8" | "9"	
DIG_WITH_0	::=	"0" | *DIG_NO_0*	
NUM	::=	"0" | *DIG_NO_0 DIG_WITH_0*∗	
CHAR	::=	"␣".."∼"	
FILENAME	::=	*CHAR* + ".picoc"	
LETTER	::=	"a".."z" | "A".."Z"	
NAME	::=	(*LETTER* | "_") (LETTER | DIG_WITH_0 | "_")*	
name	::=	*NAME* | *INT_NAME* | *CHAR_NAME*	
	|	*VOID_NAME*	
LOGIC_NOT	::=	"!"	
NOT	::=	" ∼ "	
REF_AND	::=	"&"	
un_op	::=	*SUB_MINUS* | *LOGIC_NOT* | *NOT*	
	|	*MUL_DEREF_PNTR* | *REF_AND*	
MUL_DEREF_PNTR	::=	"∗"	
DIV	::=	"/"	
MOD	::=	"%"	
prec1_op	::=	*MUL_DEREF_PNTR* | *DIV* | *MOD*	
ADD	::=	"+"	
SUB_MINUS	::=	"−"	
prec2_op	::=	*ADD* | *SUB_MINUS*	
L_SHIFT	::=	"<<"	
R_SHIFT	::=	">>"	
shift_op	::=	*L_SHIFT* | *R_SHIFT*	
LT	::=	"<"	*L_Logic*
LTE	::=	"<="	
GT	::=	">"	
GTE	::=	">="	
rel_op	::=	*LT* | *LTE* | *GT* | *GTE*	
EQ	::=	"=="	
NEQ	::=	"!="	
eq_op	::=	*EQ* | *NEQ*	
INT_DT.2	::=	"*int*"	*L_Assign_Alloc*
INT_NAME.3	::=	"*int*" (*LETTER* | *DIG_WITH_0* | "_")+	
CHAR_DT.2	::=	"*char*"	
CHAR_NAME.3	::=	"*char*" (*LETTER* | *DIG_WITH_0* | "_")+	
VOID_DT.2	::=	"*void*"	
VOID_NAME.3	::=	"*void*" (*LETTER* | *DIG_WITH_0* | "_")+	
prim_dt	::=	*INT_DT* | *CHAR_DT* | *VOID_DT*	

Grammatik 3.1.1: *Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF*

3.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 3.1 dazu verwendet die Konstruktion eines **Abstrakten Syntaxbaumes** in seinen einzelnen **Zwischenschritten** zu erläutern.

```c
struct st {int *(*attr)[4][5];};

void main() {
  struct st *(*var[3][2]);
}
```

Code 3.1: *PicoC-Code des Codebeispiels.*

Die vom **Basic Lexer** des **Lark Parsing Toolkit** generierten **Tokens** sind Code 3.2 zu sehen.

```
[Token('FILENAME', './verbose_dt_simple_ast_gen_array_decl_and_alloc.picoc'), Token('STRUCT',
↪  'struct'), Token('NAME', 'st'), Token('LBRACE', '{'), Token('INT_DT', 'int'),
↪  Token('MUL_DEREF_PNTR', '*'), Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'),
↪  Token('NAME', 'attr'), Token('RPAR', ')'), Token('LSQB', '['), Token('NUM', '4'),
↪  Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '5'), Token('RSQB', ']'),
↪  Token('SEMICOLON', ';'), Token('RBRACE', '}'), Token('SEMICOLON', ';'), Token('VOID_DT',
↪  'void'), Token('NAME', 'main'), Token('LPAR', '('), Token('RPAR', ')'), Token('LBRACE',
↪  '{'), Token('STRUCT', 'struct'), Token('NAME', 'st'), Token('MUL_DEREF_PNTR', '*'),
↪  Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'), Token('NAME', 'var'), Token('LSQB',
↪  '['), Token('NUM', '3'), Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '2'),
↪  Token('RSQB', ']'), Token('RPAR', ')'), Token('SEMICOLON', ';'), Token('RBRACE', '}')]
```

Code 3.2: *Tokens für das Codebeispiel.*

3.2 Syntaktische Analyse

In der **Syntaktischen Analyse** ist es die Aufgabe des **Parsers** aus einem Programm in **Konkreter Syntax** unter Verwendung der **Tokens** aus der **Lexikalischen Analyse** einen **Ableitungsbaum** zu generieren. Es ist danach die Aufgabe möglicher **Visitors** und die Aufgabe des **Transformers** aus diesem **Ableitungsbaum** einen **Abstrakten Syntaxbaum** in **Abstrakter Syntax** zu generieren.

3.2.1 Umsetzung von Präzedenz und Assoziativität

In diesem Unterkapitel wird eine ähnliche **Erklärweise**, wie in dem Buch Nystrom, *Parsing Expressions · Crafting Interpreters* verwendet. Die Programmiersprache L_{PicoC} hat dieselben **Präzedenzregeln** implementiert, wie die Programmiersprache L_C. Die **Präzedenzregeln** sind von der Webseite *C Operator Precedence - cppreference.com* übernommen. Die **Präzedenzregeln** der verschiedenen **Operatoren** der Programmiersprache L_{PicoC} sind in Tabelle 3.1 aufgelistet.

Präzedenzstufe	Operatoren	Beschreibung	Assoziativität
1	`a()`	Funktionsaufruf	Links, dann rechts →
	`a[]`	Indexzugriff	
	`a.b`	Attributzugriff	
2	`-a`	Unäres Minus	Rechts, dann links ←
	`!a ~a`	Logisches NOT und Bitweise NOT	
	`*a &a`	Dereferenz und Referenz, auch Adresse-von	
3	`a*b a/b a%b`	Multiplikation, Division und Modulo	Links, dann rechts →
4	`a+b a-b`	Addition und Subtraktion	
5	`a<<b a>>b`	Bitweise Linksshift und Rechtsshift	
6	`a<b a<=b` `a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer Gleich	
7	`a==b a!=b`	Gleichheit und Ungleichheit	
8	`a&b`	Bitweise UND	
9	`a^b`	Bitweise XOR (exclusive or)	
10	`a|b`	Bitweise ODER (inclusive or)	
11	`a&&b`	Logiches UND	
12	`a||b`	Logisches ODER	
13	`a=b`	Zuweisung	Rechts, dann links ←

Tabelle 3.1: *Präzedenzregeln von PicoC.*

Würde man diese **Operatoren** ohne Beachtung von **Präzedenzreglen** (Definiton 2.28) und **Assoziativität** (Definition 2.27) in eine Konkrete Grammatik verarbeiten wollen, so könnte eine Konkrete Grammatik $G = \langle N, \Sigma, P, exp \rangle$ 3.2.1 dabei rauskommen.

$$
\begin{array}{lcll}
prim\_exp & ::= & name \mid NUM \mid \text{"'"} CHAR \text{"'"} \mid \text{"("} exp \text{")"} & \textit{L\_Arith\_Bit} \\
 & \mid & exp \text{"["} exp \text{"]"} \mid exp \text{"."} name \mid name \text{"("} fun\_args \text{")"} & \textit{+L\_Logic} \\
fun\_args & ::= & [exp(\text{","}\ exp)*] & \textit{+ L\_Pntr} \\
un\_op & ::= & \text{"}-\text{"} \mid \text{"}\sim\text{"} \mid \text{"!"} \mid \text{"}*\text{"} \mid \text{"\&"} & \textit{+ L\_Array} \\
un\_exp & ::= & un\_op\ exp & \textit{+ L\_Struct} \\
bin\_op & ::= & \text{"}*\text{"} \mid \text{"/"} \mid \text{"\%"} \mid \text{"+"} \mid \text{"}-\text{"} \mid \text{"\&"} \mid \text{"}\wedge\text{"} \mid \text{"}|\text{"} & \textit{+ L\_Fun} \\
 & \mid & \text{"<<"} \mid \text{">>"} \mid \text{"<"} \mid \text{"<="} \mid \text{">"} \mid \text{">="} & \\
 & \mid & \text{"!="} \mid \text{"=="} \mid \text{"\&\&"} \mid \text{"||"} \mid \text{"="} & \\
bin\_exp & ::= & exp\ bin\_op\ exp & \\
exp & ::= & prim\_exp \mid un\_exp \mid bin\_exp &
\end{array}
$$

Grammatik 3.2.1: *Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet*

Die Konkrete Grammatik 3.2.1 ist allerdings **mehrdeutig** (Definition 2.26), d.h. verschiedene **Linksableitungen** in der Konkreten Grammatik können zum selben **Wort** abgeleitet werden. Z.B. kann das Wort `3 * 1 && 4` sowohl über die **Linksableitung** 3.5.1 als auch über die **Linksableitung** 3.5.2 abgeleitet werden. Ab dem Moment, wo der Trick klar ist, wird das **Ableiten** mit der $\Rightarrow^*$-Relation **beschleunigt**.

$$
\begin{aligned}
exp \quad & \Rightarrow bin\_exp \Rightarrow exp\ bin\_op\ exp \Rightarrow bin\_exp\ bin\_op\ exp \\
& \Rightarrow exp\ bin\_op\ exp\ bin\_op\ exp \Rightarrow^* \text{"3"}\ \text{"}*\text{"}\ \text{"1"}\ \text{"\&\&"}\ \text{"4"}
\end{aligned}
\tag{3.5.1}
$$

$$\begin{aligned} exp &\Rightarrow bin\_exp \Rightarrow exp\ bin\_op\ exp \Rightarrow prim\_exp\ bin\_op\ exp \\ &\Rightarrow NUM\ bin\_op\ exp \Rightarrow \text{"3"}\ bin\_op\ exp \Rightarrow \text{"3"}\ \text{"}*\text{"}\ exp \\ &\Rightarrow \text{"3"}\ \text{"}*\text{"}\ bin\_exp \Rightarrow \text{"3"}\ \text{"}*\text{"}\ exp\ bin\_op\ exp \Rightarrow^* \text{"3"}\ \text{"}*\text{"}\ \text{"1"}\ \text{"\&\&"}\ \text{"4"} \end{aligned} \tag{3.5.2}$$

Die beiden **abgeleiteten Wörter** sind **gleich**, allerdings sind die **Ableitungsbäume unterschiedlich**, wie in Abbildung 3.1 zu sehen ist. Da hier nur ein **Konzept** vermittelt werden soll, entsprechen die beiden **Ableitungsbäume** in Abbildung 3.1 nicht **1-zu-1** den Ableitungen 3.5.1 und 3.5.2, sondern sind **vereinfacht**.

Abbildung 3.1: *Ableitungsbäume zu den beiden Ableitungen.*

Der **linke Baum** entspricht Ableitung 3.5.1 und der **rechte Baum** entspricht Ableitung 3.5.2. Würde man in den Ausdrücken, die von diesen Bäumen darsgestellt sind **Klammern** setzen, um die **Präzedenz** sichtbar zu machen, so würde Ableitung 3.5.1 die Klammerung `(3 * 1) && 4` haben und die Ableitung 3.5.2 die Klammerung `3 * (1 && 4)` haben. Es ist wichtig die **Präzedenzregeln** und die **Assoziativität** von Operatoren beim Erstellen der Konkreten Grammatik miteinzubeziehen, da das Ergebnis des **gleichen Ausdrucks** sich bei **unterschiedlicher Klammerung unterscheiden** kann.

Hierzu wird nun Tabelle 3.1 betrachtet. Für jede **Präzedenzstufe** in der Tabelle 3.1 wird eine eigene Produktion erstellt, wie es in Grammatik 3.2.2 dargestellt ist. Zudem braucht es eine **Produktion** `prim_exp` für die **„höchste“ Präzedenzstufe**, welche **Literale**, wie `'c'`, `5` oder `var` und geklammerte Ausdrücke wie `(3 && 14)` abdeckt.

prim_exp	::=	...	*L_Arith_Bit + L_Array*
post_exp	::=	...	*+ L_Pntr + L_Struct*
un_exp	::=	...	*+ L_Fun*
*arith_prec*1	::=	...	
*arith_prec*2	::=	...	
arith_shift	::=	...	
arith_and	::=	...	
arith_xor	::=	...	
arith_or	::=	...	
rel_exp	::=	...	*L_Logic*
eq_exp	::=	...	
logic_and	::=	...	
logic_or	::=	...	
assign_stmt	::=	...	*L_Assign*

Grammatik 3.2.2: *Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet*

Einige **Bezeichnungen** von **Nicht-Terminalsymbolen** auf der **linken Seite** des `::=`-Symbols in Grammatik 3.2.2 sind in Tabelle 3.2 ihren jeweiligen **Operatoren** zugeordnet, für welche sie zuständig sind.

Bezeichnung der Produktionsregel	Operatoren
post_exp	a() a[] a.b
un_exp	-a !a ~a *a &a
arith_prec1	a*b a/b a%b
arith_prec2	a+b a-b
arith_shift	a<<b a>>b
arith_and	a&b
arith_xor	a^b
arith_or	a|b
rel_exp	a<b a<=b a>b a>=b
eq_exp	a==b a!=b
logic_and	a&&b
logic_or	a||b
assign	a=b

Tabelle 3.2: *Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren.*

Als nächstes müssen die einzelnen **Produktionen** entsprechend der **Ausdrücke** für die sie zuständig sind definiert werden. Jede der **Produktionen** soll nur Ausdrücke **erkennen** können, deren **Präzedenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzedenzstufe **höher** ist. Z.B. soll `un_op` sowohl den Ausdruck `-(3 * 14)` als auch einfach nur `(3 * 14)`[7] erkennen können, aber nicht `3 * 14` ohne Klammern, da dieser Ausdruck eine **geringe Präzedenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

Im Folgenden werden **Produktionen** für alle **relevanten Fälle** von verschiedenen Kombinationen von **Präzedenzen** und **Assoziativitäten** erklärt. Bei z.B. der Produktion `un_exp` in 3.2.3 für die **rechtsassoziativen unären Operatoren** `-a`, `!a ~a`, `*a` und `&a` ist die **Alternative** `un_op un_exp` dafür zuständig, dass diese **unären Operatoren rechtsassoziativ** geschachtelt werden können (z.B. `!-~42`). Die Alternative `post_exp` ist dafür zuständig, dass die Produktion beim Ableiten auch **terminieren** kann und es auch möglich ist, auschließlich einen Ausdruck **höherer Präzedenz** (z.B. `42`) zu haben.

$$un\_exp \quad ::= \quad un\_op\ un\_exp \quad | \quad post\_exp$$

Grammatik 3.2.3: *Beispiel für eine unäre rechtsassoziative Produktion in EBNF*

Bei z.B. der Produktion `post_exp` in 3.2.4 für die **linksassoziativen unären Operatoren** `a()`, `a[]` und `a.b` sind die Alternativen `post_exp"["logic_or"]"` und `post_exp"."name` dafür zuständig, dass diese unären Operatoren **linksassoziativ** geschachtelt werden können (z.B. `ar[3][1].car[4]`). Die Alternative `name"("fun_args")"` ist für einen **einzelnen Funktionsaufruf** zuständig. Die Alternative `prim_exp` ist dafür zuständig, dass die Produktion nicht nur bei `name"("fun_args")"` **terminieren** kann und es auch möglich ist, auschließlich einen Ausdruck der **höchsten Präzedenz** (z.B. `42`) zu haben.

$$post\_exp \quad ::= \quad post\_exp\text{"["}logic\_or\text{"]"} \quad | \quad post\_exp\text{"."}name \quad | \quad name\text{"("}fun\_args\text{")"} \quad | \quad prim\_exp$$

Grammatik 3.2.4: *Beispiel für eine unäre linksassoziative Produktion in EBNF*

[7] **Geklammerte Ausdrücke** werden nämlich von `prim_exp` erkannt, welches eine höhere **Präzedenzstufe** hat.

Bei z.B. der Produktion `prec2_exp` in 3.2.5 für die **binären linksassoziativen Operatoren** `a+b` und `a-b` ist die **Alternative** `arith_prec2 prec2_op arith_prec1` dafür zuständig, dass **mehrere** Operationen der Präzedenzstufe 4 in Folge erkannt werden können[8] (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzedenzstufe 4 haben). Die Alternative `arith_prec1` auf der **rechten Seite** ermöglicht es, dass zwischen den Operationen der Präzedenzstufe 4 auch Operationen der Präzedenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzedenzstufe 4 haben und `/` Präzedenzstufe 3). Mit der Alternative `arith_prec1` ist es möglich, dass ausschließlich ein Ausdruck **höherer Präzedenz** erkannt wird (z.B. `1 / 4`).

$$arith\_prec2 \quad ::= \quad arith\_prec2 \;\; prec2\_op \;\; arith\_prec1 \quad | \quad arith\_prec1$$

Grammatik 3.2.5: *Beispiel für eine binäre linksassoziative Produktion in EBNF*

Anmerkung

Manche **Parser**[a] haben allerdings ein Problem mit **Linksrekursion** (Definition 5.9), wie sie z.B. in der Produktion 3.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 3.2.5 zur Produktion 3.2.6 umschreibt.

$$arith\_prec2 \quad ::= \quad arith\_prec1 \;\; (prec2\_op \;\; arith\_prec1)*$$

Grammatik 3.2.6: *Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF*

Die von der Grammatik 3.2.6 erkannten Ausdrücke sind dieselben, wie für die Grammatik 3.2.5, allerdings ist die Grammatik 3.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 3.3) definierten `*`-Operator, um **mehrere Operationen** der Präzedenzstufe 4 in Folge erkennen zu können (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzedenzstufe 4 haben).

Das **Nicht-Terminalsymbol** `arith_prec1` erlaubt es, dass **zwischen** der Folge von Operationen der Präzedenzstufe 4 auch Operationen der Präzedenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzedenzstufe 4 haben und `/` Präzedenzstufe 3). Da der in der EBNF definierte `*`-Quantor auch bedeutet, dass das Teilpattern auf das er sich bezieht **kein einziges mal** vorkommen kann, ist es mit dem **linken Nicht-Terminalsymbol** `arith_prec1` möglich, dass ausschließlich ein Ausdruck **höherer Präzedenz** erkannt wird (z.B. `1 / 4`).

[a] Zu diesen Parsern zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird glücklicherweise **nicht**.

Alle **Operatoren** der Sprache L_{PicoC} sind also entweder **binär** und **linksassoziativ** (z.B. `a*b`, `a-b`, `a>=b` oder `a&&b`), **unär** und **rechtsassoziativ** (z.B. `&a` oder `!a`) oder **unär** und **linksassoziativ** (z.B. `a[]` oder `a()`). Mithilfe dieser Paradigmen lässt sich die **Konkrete Grammatik** 3.2.7 definieren.

[8] Bezogen auf Tabelle 3.1.

prec1_op	::=	"*" | "/" | "%"	*L_Misc*
prec2_op	::=	"+" | "−"	
shift_op	::=	"<<" | ">>"	
rel_op	::=	"<" | "<=" | ">" | ">="	
eq_op	::=	"==" | "!="	
fun_args	::=	[*logic_or*("," *logic_or*)*]	
prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith_Bit*
post_exp	::=	*post_exp*"["*logic_or*"]" | *post_exp*"."*name* | *name*"("*fun_args*")"	*+ L_Array*
	|	*prim_exp*	*+ L_Pntr*
un_exp	::=	*un_op un_exp* | *post_exp*	*+ L_Struct*
arith_prec1	::=	*arith_prec1 prec1_op un_exp* | *un_exp*	*+ L_Fun*
arith_prec2	::=	*arith_prec2 prec2_op arith_prec1* | *arith_prec1*	
arith_shift	::=	*arith_shift shift_op arith_prec2* | *arith_prec2*	
arith_and	::=	*arith_and* "&" *arith_shift* | *arith_shift*	
arith_xor	::=	*arith_xor* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_xor* | *arith_xor*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_op rel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	*L_Assign*

Grammatik 3.2.7: *Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet*

3.2.2 Konkrete Grammatik für die Syntaktische Analyse

Die gesamte Konkrete Grammatik 3.2.8 ergibt sich wenn man die Konkrete Grammatik 3.2.7 um die **restliche Syntax** der Sprache L_{PicoC} erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 3.2.1 erläutert ergibt.

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrete Grammatik** an die **aktuellste kostenlos auffindbare Version** der echten Konkreten Grammatik der Sprache L_C, zusammengesetzt aus einer Grammatik für die **Syntaktische Analyse** *ANSI C grammar (Yacc)* und **Lexikalische Analyse** *ANSI C grammar (Lex)* angepasst[9]. Auf diese Weise konnte sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C[10].

In der Konkreten Grammatik 3.2.8 für die **Syntaktische Analyse** werden einige der **Nicht-Terminalsymbole** bzw. **Tokentypen** aus der Konkreten Grammatik 3.1.1 für die **Lexikalischen Analyse** verwendet, wie z.B. `NUM`. Es werde aber auch Produktionen, wie `name` verwendet, die mehrere **Tokentypen** unter einem Überbegriff zusammenfassen.

Terminalsymbole, wie `;` oder `&&` gehören eigentlich zur **Lexikalischen Analyse**, jedoch erlaubt das **Lark Parsing Toolkit**, um die Konkrete Grammatik leichter lesbar zu machen einige **Terminalsymbole** einfach direkt in die Konkrete Grammatik 3.2.8 für die **Syntaktische Analyse** zu schreiben. Der **Tokentyp** für diese Terminalsymbole wird in diesem Fall vom **Lark Parsing Toolkit** bestimmt, welches einige sehr häufig verwendete **Terminalsymbole**, wie z.B. `;` oder `&&` bereits einen eigenen **Tokentyp** zugewiesen hat.

[9] An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts wichtiges verändert, wie die Konkreten Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Yacc) old* und **Lexikalische Analyse** *ANSI C grammar (Lex) old* aus dem Jahre 1985 zeigen.

[10] Wobei z.B. die Compiler **GCC** (*GCC, the GNU Compiler Collection - GNU Project*) und **Clang** (*clang: C++ Compiler*) zu nennen wären.

Diese **Terminalsymbole** werden aber weiterhin vom **Basic Lexer** als Teil der **Lexikalischen Analyse** generiert.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith_Bit + L_Array*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*+ L_Pntr + L_Struct*
	|	*input_exp* | *print_exp* | *prim_exp*	*+ L_Fun*
un_exp	::=	*un_op un_exp* | *post_exp*	
input_exp	::=	"*input*""("")"	*L_Arith_Bit*
print_exp	::=	"*print*""("*logic_or*")"	
*arith_prec*1	::=	*arith_prec*1 *prec*1*_op un_exp* | *un_exp*	
*arith_prec*2	::=	*arith_prec*2 *prec*2*_op arith_prec*1 | *arith_prec*1	
arith_shift	::=	*arith_shift shift_op arith_prec*2 | *arith_prec*2	
arith_and	::=	*arith_and* "&" *arith_shift* | *arith_shift*	
arith_xor	::=	*arith_xor* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_xor* | *arith_xor*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_op rel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
type_spec	::=	*prim_dt* | *struct_spec*	*L_Assign_Alloc*
alloc	::=	*type_spec pntr_decl*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	
initializer	::=	*logic_or* | *array_init* | *struct_init*	
init_stmt	::=	*alloc* "=" *initializer*";"	
const_init_stmt	::=	"*const*" *type_spec name* "=" *NUM*";"	
pntr_deg	::=	"∗"∗	*L_Pntr*
pntr_decl	::=	*pntr_deg array_decl* | *array_decl*	
array_dims	::=	("["*NUM*"]")∗	*L_Array*
array_decl	::=	*name array_dims* | "("*pntr_decl*")"*array_dims*	
array_init	::=	"{"*initializer*("," *initializer*) ∗ "}"	
array_subscr	::=	*post_exp*"["*logic_or*"]"	
struct_spec	::=	"*struct*" *name*	*L_Struct*
struct_params	::=	(*alloc*";")+	
struct_decl	::=	"*struct*" *name* "{"*struct_params*"}"	
struct_init	::=	"{""."*name*"="*initializer* ("," "."name"="initializer)*"}"	
struct_attr	::=	*post_exp*"."*name*	
if_stmt	::=	"*if*""("*logic_or*")" *exec_part*	*L_If_Else*
if_else_stmt	::=	"*if*""("*logic_or*")" *exec_part* "*else*" *exec_part*	
while_stmt	::=	"*while*""("*logic_or*")" *exec_part*	*L_Loop*
do_while_stmt	::=	"*do*" *exec_part* "*while*""("*logic_or*")"""	

Grammatik 3.2.8: *Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1*

decl_exp_stmt	::=	*alloc*";"	*L_Stmt*
decl_direct_stmt	::=	*assign_stmt* | *init_stmt* | *const_init_stmt*	
decl_part	::=	*decl_exp_stmt* | *decl_direct_stmt* | *RETI_COMMENT*	
compound_stmt	::=	"{" *exec_part* ∗ "}"	
exec_exp_stmt	::=	*logic_or*";"	
exec_direct_stmt	::=	*if_stmt* | *if_else_stmt* | *while_stmt* | *do_while_stmt*	
	|	*assign_stmt* | *fun_return_stmt*	
exec_part	::=	*compound_stmt* | *exec_exp_stmt* | *exec_direct_stmt*	
	|	*RETI_COMMENT*	
decl_exec_stmts	::=	*decl_part* ∗ *exec_part*∗	
fun_args	::=	[*logic_or*(",") *logic_or*)∗]	*L_Fun*
fun_call	::=	*name*"(" *fun_args*")"	
fun_return_stmt	::=	"*return*" [*logic_or*]";"	
fun_params	::=	[*alloc*(",") *alloc*)∗]	
fun_decl	::=	*type_spec pntr_deg name*"(" *fun_params*")"	
fun_def	::=	*type_spec pntr_deg name*"(" *fun_params*")" "{" *decl_exec_stmts*"}"	
decl_def	::=	(*struct_decl* | *fun_decl*)";" | *fun_def*	*L_File*
decls_defs	::=	*decl_def*∗	
file	::=	*FILENAME decls_defs*	

Grammatik 3.2.9: *Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2*

Anmerkung

In der Konkreten Grammatik 3.2.8 sind alle **Grammatiksymbole** ausgegraut, die das **Bachelorprojekt** betreffen. Alle `nicht ausgegrauten` **Grammatiksymbole** wurden für die Implementierung der **neuen Funktionalitäten**, welche die **Bachelorarbeit** betreffen hinzugefügt.

3.2.3 Ableitungsbaum Generierung

Die in Unterkapitel 3.2.2 definierte Konkrete Grammatik 3.2.8 lässt sich mithilfe des **Earley Parsers** (Definition 3.6) von Lark dazu verwenden, Code, der in der Sprache L_{PicoC} geschrieben ist zu **parsen**, um einen **Ableitungsbaum** daraus zu generieren.

Definition 3.6: Earley Parser

*Ist ein Algorithmus für das **Parsen** von **Wörtern** einer **Kontextfreien Sprache**. Der **Earley Parser** ist ein **Chart Parser** ist, welcher einen mittels **Dynamischer Programmierung** und **Top-Down Ansatz** arbeitenden **Earley Erkenner** (Defintion 5.11 im Appendix) nutzt, um einen **Ableitungsbaum** zu konstruieren.*

*Zur **Konstruktion des Ableitungsbaumes** muss dafür gesorgt werden, dass der **Earley Erkenner** bei der **Vervollständigungsoperation** Zeiger auf den **vorherigen Zustand** hinzugefügt, um durch **Rückwärtsverfolgen** dieser **Zeiger** die genommenen **Ableitungen** wieder **nachvollziehen** zu können und so einen **Ableitungsbaum** konstruieren zu können.*[a]

[a] Jay Earley, „An efiicient context-free parsing“.

3.2.3.1 Codebeispiel

Der **Ableitungsbaum**, der mithilfe des **Earley Parsers** und der **Tokens** der **Lexikalischen Analyse** in Code 3.2 generiert wurde, ist in Code 3.3 zu sehen. Das Beispiel aus Code 3.1 wird hier **fortgeführt**. Im Code 3.3 wurden einige Zeilen **markiert**, die später in Unterkapitel 3.2.4.1 zum Vergleich wichtig sind.

```
1  file
2    ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt
3    decls_defs
4      decl_def
5        struct_decl
6          name        st
7          struct_params
8            alloc
9              type_spec
10               prim_dt        int
11             pntr_decl
12               pntr_deg        *
13               array_decl
14                 pntr_decl
15                   pntr_deg        *
16                   array_decl
17                     name        attr
18                     array_dims
19                 array_dims
20                   4
21                   5
22     decl_def
23       fun_def
24         type_spec
25           prim_dt        void
26         pntr_deg
27         name        main
28         fun_params
29         decl_exec_stmts
30           decl_part
31             decl_exp_stmt
32               alloc
33                 type_spec
34                   struct_spec
35                     name        st
36                 pntr_decl
37                   pntr_deg        *
38                   array_decl
39                     pntr_decl
40                       pntr_deg        *
41                       array_decl
42                         name        var
43                         array_dims
44                           3
45                           2
46                     array_dims
```

Code 3.3: *Ableitungsbaum nach Ableitungsbaum Generierung.*

3.2.3.2 Ausgabe des Ableitunsgbaumes

Die Ausgabe des **Ableitungsbaumes** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Konkreten Grammatik 3.2.8 den **linken Seiten** des `::=`-Symbols[11] entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition 2.35 eines **Ableitungsbaumes** auch schon definiert ist. Die **Konkrete Grammatik** 3.2.8 des **PicoC-Compilers** erlaubt es auch, dass in einem **Blatt** garnichts ε steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("["NUM"]")*` gibt, in denen auch das **leere Wort** ε abgeleitet werden kann.

3.2.4 Ableitungsbaum Vereinfachung

Der **Ableitungsbaum** in Code 3.3, dessen Generierung in Unterkapitel 3.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Tramsformers** ein **Abstrakter Syntaxbaum** generiert werden kann. Das Problem ist, dass um den **Datentyp** einer Variable in der Programmiersprache L_C und somit auch der Programmiersprache L_{PicoC} korrekt bestimmen zu können die **Spiralregel** *Clockwise/Spiral Rule* in der Implementeirung des **PicoC-Compilers** umgesetzt werden muss. Dies ist allerdings nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Konkreten Grammatik 3.2.8 auf eine spezielle Weise passend spezifiziert. Der **PicoC-Compiler** soll in der Lage sein den Ausdruck `int (*ar[3])[2]` als **„Feld der Mächtigkeit** 3 **von Zeigern auf Felder der Mächtigkeit** 2 **von Integern“** erkennen zu können.

Was man erhalten will, ist ein **entarteter Baum** (Definition 3.7) von **PicoC-Knoten**, an dem man den **Datentyp** direkt ablesen kann, indem man sich einfach über den **entarteten Baum** bewegt, wie z.B. `PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], PntrDecl(Num('1'), StructSpec(Name('st')))))` für den Ausdruck `struct st *(*var[3][2])`.

Definition 3.7: Entarteter Baum

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

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

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

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

Es sind hierbei mehrere Probleme zu lösen. Hat man den Ausdruck `struct st *(*var[3][2])`, wird dieser zu einem **Ableitungsbaum**, wie er in Abbildung 3.2 zu sehen ist.

[11] *Grammar: The language of languages (BNF, EBNF, ABNF and more)*.

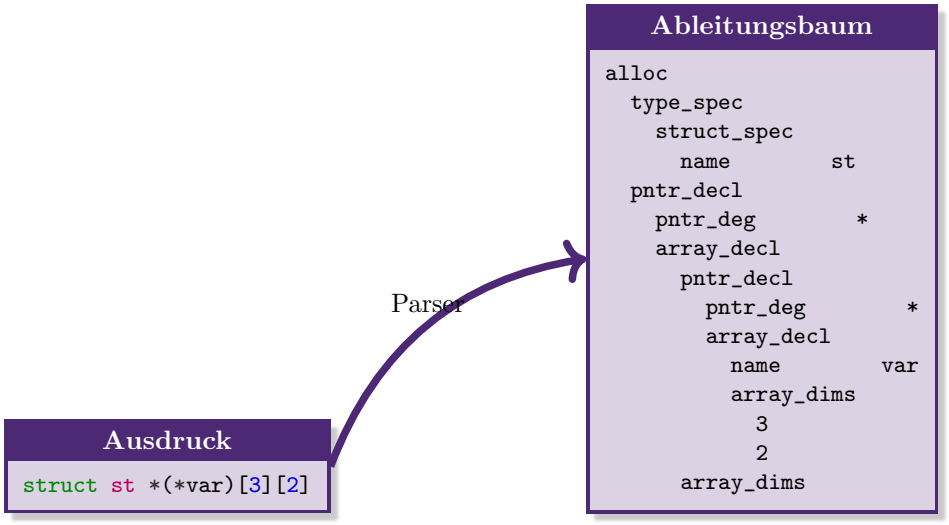


Abbildung 3.2: *Ableitungsbaum nach Parsen eines Ausdrucks.*

Dieser **Ableitungsbaum** für den Ausdruck `struct st *(*var[3][2])` hat allerdings einen Aufbau welcher durch die **Syntax** der **Zeigerdeklaratoren** `pntr_decl(num, datatype)` und **Felddeklaratoren** `array_decl(datatype, nums)` bestimmt ist, die **spiralähnlich** ist. Man würde allerdings gerne einen **entarteten Baum** erhalten, bei dem der Datentyp z.B. immer im **zweiten Attribut** weitergeht, anstatt abwechselnd im **zweiten** und **ersten**, wie beim **Zeigerdeklarator** `pntr_decl(num, datatype)` und **Felddeklarator** `array_decl(datatype, nums)`. Daher wird bei allen **Felddeklaratoren** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht.

Des Weiteren befindet sich in der **Mitte** der **Spirale**, die der **Ableitungsbaum** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`. Das liegt daran, dass der **Ableitungsbaum** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Ausdrucks** in **Konkreter Syntax** (z.B. `struct st *(*var[3][2])`) **generiert** wird darstellt. Der Knoten für den **Bezeichner der Variable** `name(var)` wird daher mithilfe eines **Visitors** (Definition 2.39) mit dem Knoten für den **innersten Datentyp** `struct st` getauscht.

Eine Änderung, die eher der **Ästhetik** dient, ist zusätzlich den Teilbaum, der den **Datentyp** darstellt mit dem Knoten `name(var)` zu **tauschen**. Das hat den Grund, dass der **Datentyp** üblicherweise **vor** dem **Bezeichner** der Variable steht: `datatype identifier`[12] und im vorherigen Schritt `name(var)` **vor** den **Datentyp** getauscht wurde. In Abbildung 3.3 ist zu sehen, wie der **Ableitungsbaum** aus Abbildung 3.2 mithilfe eines **Visitors vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des **Visitors** aus dem **Lark Parsing Toolkit** ist unter Link[13] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Visitor** verhält sich allerdings grundlegend so, wie es in Definition 2.39 erklärt wurde.

[12] Wie z.B. bei `int var`.
[13] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.

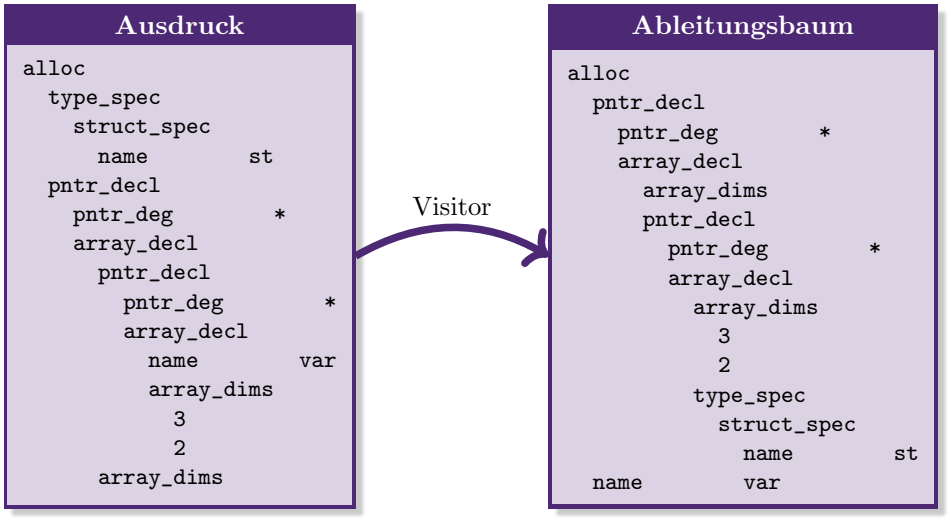


Abbildung 3.3: *Ableitungsbaum nach Vereinfachung.*

3.2.4.1 Codebeispiel

In Code 3.4 ist der **Ableitungsbaum** aus Code 3.3 nach der **Vereinfachung** mithilfe eines **Visitors** zu sehen. Das Beispiel aus Code 3.1 wird hier **fortgeführt**.

```
1  file
2    ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3    decls_defs
4      decl_def
5        struct_decl
6          name        st
7          struct_params
8            alloc
9              pntr_decl
10               pntr_deg        *
11               array_decl
12                 array_dims
13                   4
14                   5
15                 pntr_decl
16                   pntr_deg        *
17                   array_decl
18                     array_dims
19                     type_spec
20                       prim_dt        int
21             name        attr
22     decl_def
23       fun_def
24         type_spec
25           prim_dt        void
26         pntr_deg
27         name        main
28         fun_params
29         decl_exec_stmts
```

```
30          decl_part
31            decl_exp_stmt
32              alloc
33                pntr_decl
34                  pntr_deg        *
35                  array_decl
36                    array_dims
37                    pntr_decl
38                      pntr_deg        *
39                      array_decl
40                        array_dims
41                          3
42                          2
43                        type_spec
44                          struct_spec
45                            name        st
46                name        var
```

Code 3.4: *Ableitungsbaum nach Ableitungsbaum Vereinfachung.*

3.2.5 Generierung des Abstrakten Syntaxbaumes

Nachdem der **Ableitungsbaum** in Unterkapitel 3.2.4 vereinfacht wurde, ist der **vereinfachte Ableitungsbaum** in Code 3.4 nun dazu geeignet, um mit einem **Transformer** (Definition 2.38) einen **Abstrakten Syntaxbaum** aus ihm zu generieren. Würde man den **vereinfachten Ableitungsbaum** des Ausdrucks `struct st *(*var[3][2])` auf die übliche Weise in einen entsprechenden **Abstrakten Syntaxbaum** umwandeln, so würde dabei ein **Abstrakter Syntaxbaum** wie in Abbildung 3.4 rauskommen.

Die Implementierung des **Transformers** aus dem **Lark Parsing Toolkit** ist unter Link[14] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Transformer** verhält sich allerdings grundlegend so, wie es in Definition 2.38 erklärt wurde.

Den Teilbaum, der **rechts** in Abbildung 3.4 den Datentyp darstellt, würde man von **oben-nach-unten**[15] als **„Zeiger auf einen Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs `st`“** lesen. Man würde es also bis auf die **Dimensionen der Felder**, bei denen es freisteht, wie man sie liest genau **anders herum** lesen, als man den Ausdruck `struct st *(*var[3][2])` mit der **Spiralregel** lesen würde. Bei der **Spiralregel** fängt man beim Ausdruck `struct st *(*var[3][2])` bei der Variable `var` an und arbeitet sich dann auf **„Spiralbahnen“**, von **innen-nach-außen** durch den Ausdruck, um herauszufinden, dass dieser Datentyp ein **„Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Zeigern auf einen Zeiger auf einen Verbund vom Typ `st`“** ist.

[14] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.
[15] In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**, bzw. in diesem Beispiel von **links-nach-rechts**.

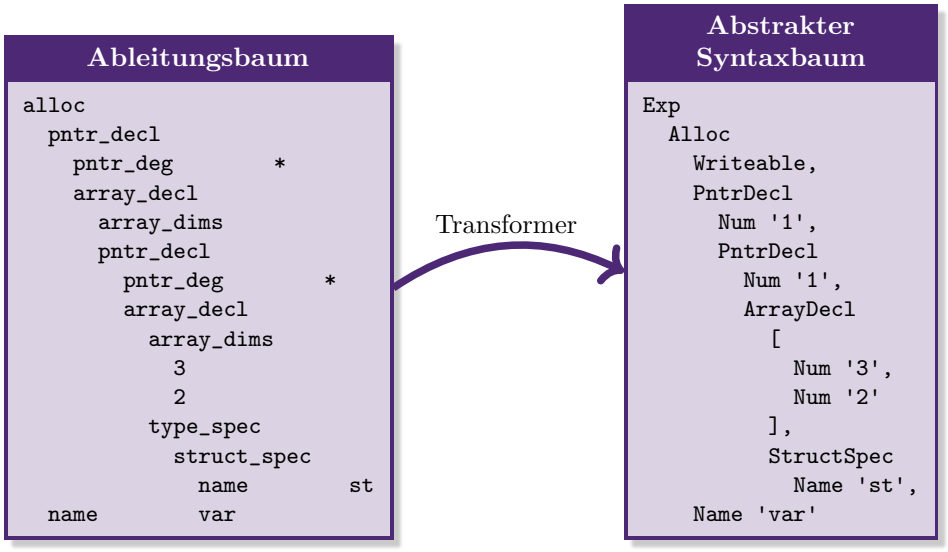


Abbildung 3.4: *Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen.*

Der **Abstrakte Syntaxbaum** rechts in Abbildung 3.4 ist für die Weiterverarbeitung also ungeeignet, denn für die **Adressberechnung** bei einer Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute**[16] will man den Datentyp in **umgekehrter Reihenfolge**. Aus diesem Grund muss der **Transformer** bei der Konstruktion des **Abstrakten Syntaxbaumes** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **vollständigen Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie **rechts** in Abbildung 3.5 zustande, der für die Weiterverarbeitung geeignet ist.

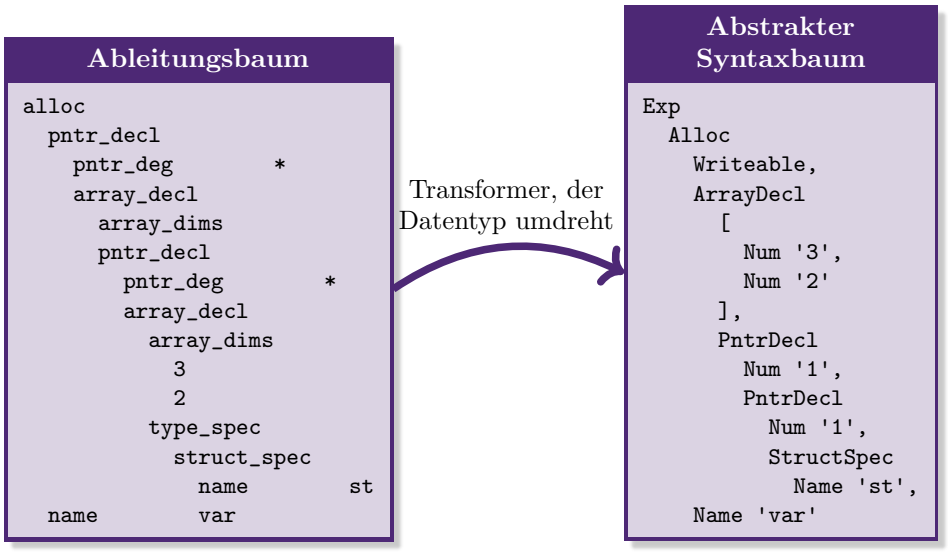


Abbildung 3.5: *Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.*

Die Weiterverarbeitung des **Abstrakten Syntaxbaumes** geschieht mithilfe von **Passes**, welche im Unterkapitel 3.3 genauer beschrieben werden. Da die Knoten des **Abstrakten Syntaxbaumes** anders als beim

[16] Welche in Unterkapitel 3.3.5.2 genauer erläutert wird

Ableitungsbaum nicht die gleichen Bezeichnungen haben, wie **Nicht-Terminalsymbole** der Konkreten Grammatik, ist es in den folgenden Unterkapiteln 3.2.5.1, 3.2.5.2 und 3.2.5.3 notwendig die **Bedeutung** der einzelnen **PicoC-Knoten**, **RETI-Knoten** und bestimmter **Kompositionen** dieser Knoten zu **dokumentieren**. Diese Knoten kommen später im Unterkapitel 3.3 in den unterschiedlichen von den **Passes** umgeformten **Abstrakten Syntaxbäumen** vor.

Des Weiteren gibt die Abstrakte Grammatik 3.2.10 in Unterkapitel 3.2.5.4 Aufschluss darüber welche **Kompositionen von PicoC-Knoten** neben den bereits in Tabelle 3.8 definierten Kompositionen mit Bedeutung insgesamt überhaupt **möglich** sind.

3.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die wenn man die Programmiersprache L_{PicoC} auf das herunterbricht, was wirklich entscheidend ist ein **abstraktes Konstrukt** darstellen, welches z.B. ein **Container** für andere Knoten sein kann oder einen der ursprünglichen **Token darstellt**. Diese **Abstrakten Konstrukte** sollen allerdings immer noch etwas aus der Programmiersprache L_{PicoC} darstellen.

Für die **PicoC-Knoten** wurden möglichst **kurze** und **leicht** verständliche Bezeichner gewählt, da auf diese Weise bei der Implementierung der einzelnen Passes möglichst **viel Code in eine Zeile** passt und dieser Code auch durch leicht verständliche Bezeichner von Knoten **intuitiv verständlich** sein sollte[17]. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakten Syntaxbäumen** vorkommen sind in Tabelle 3.3 mit einem **Beschreibungstext** dokumentiert.

[17] Z.B. steht der **PicoC-Knoten** `Name(str)` für einen **Bezeichner**. Anstatt diesen Knoten in englisch `Identifier(str)` zu nennen, wurde dieser als `Name(str)` gewählt, da `Name(str)` **kürzer** ist und **inuitiver verständlich**.

PiocC-Knoten	Beschreibung
`Name(str)`	Repräsentiert einen **Bezeichner** (z.B. `my_fun, my_var usw.`). Das Attribut `str` ist eine Zeichenkette, welche beliebige **Groß-** und **Kleinbuchstaben** (`a - z, A - Z`), **Zahlen** (`0 - 9`) und den **Unterstrich** (`_`) enthalten kann. An **erster Stelle** darf allerdings **keine Zahl** stehen.
`Num(str)`	Eine **Zahl** (z.B. `42, -3` usw.). Hierbei ist das Attribut `str` eine Zeichenkette, welche **beliebige Ziffern** (`0 - 9`) enthalten kann. Es darf nur **nicht** eine `0` am Anfang stehen und danach **weitere Ziffern** folgen. Der **Wert**, welcher durch die Zeichenkette dargestellt wird, darf **nicht größer** als $2^{32} - 1$ sein.
`Char(str)`	Ein **Zeichen** (`'c', '*'` usw.). Das Attribut `str` ist ein Zeichnen der **ASCII-Zeichenkodierung**.
`Minus(), Not(), DerefOp(), RefOp(), LogicNot()`	Die **unären Operatoren** `un_op`: `-a, ~a, *a, &a !a`.
`Add(), Sub(), Mul(), Div(), Mod(), LShift(), RShift(), Xor(), And(), Or(), LogicAnd(), LogicOr()`	Die **binären Operatoren** `bin_op`: `a + b, a - b, a * b, a / b, a % b, a << b, a >> b, a ^ b, a & b, a | b, a && b, a || b`.
`Eq(), NEq(), Lt(), LtE(), Gt(), GtE()`	Die **Relationen** `rel`: `a == b, a != b, a < b, a <= b, a > b, a >= b`.
`Const(), Writeable()`	Die **Type Qualifier** `type_qual`: `const`, was für ein **nicht beschreibbare Konstante** steht und das **nicht** Angeben von `const`, was für einen **beschreibbare** Variable steht.
`IntType(), CharType(), VoidType()`	Die **Type Specifier** für **Basisdatentypen**: `int`, `char`, `void`. Um eine **intuitive Bezeichnung** zu haben werden sie in der **Abstrakten Grammatik** einfach nur als **Datentypen** `datatype` eingeordnet werden.
`Placeholder()`	**Platzhalter** für einen Knoten, der diesen später **ersetzt**.
`BinOp(exp, bin_op, exp)`	Container für eine **binäre Operation**: `<exp1> <bin_op> <exp2>`.
`UnOp(un_op, exp)`	Container für eine **unäre Operation**: `<un_op> <exp>`.
`Atom(exp, rel, exp)`	Container für eine **binäre Relation**: `<exp1> <rel> <exp2>`
`ToBool(exp)`	Container für einen **Arithmetischen oder Bitweise Ausdruck**, wie z.B. `1 + 3` oder einfach nur `3`. Aufgrund des **Kontext** in dem sich dieser Ausdruck befindet, wird bei einem Ergebnis $x > 0$ auf 1 abgebildet und bei $x = 0$ auf 0.
`Alloc(type_qual, datatype, name, local_var_or_param)`	Container für eine **Allokation** `<type_qual> <datatype> <name>` mit den Attributen `type_qual`, `datatype` und `name`, die alle Informationen für einen Eintrag in der **Symboltabelle** enthalten. Zudem besitzt er ein versteckes Attribut `local_var_or_param`, dass die Information trägt, ob es sich bei der **Variable** um eine **Lokale Variable** oder einen **Parameter** handelt.
`Assign(exp1, exp2)`	Container für eine **Zuweisung** `exp1 = exp2`, wobei `exp1` z.B. ein `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` sein kann und `exp2` ein beliebiger **Logischer Ausdruck** sein kann.

Tabelle 3.3: *PicoC-Knoten Teil 1.*

PiocC-Knoten	Beschreibung
`Exp(exp, datatype, error_data)`	Container für einen **beliebigen Ausdruck**, dessen Ergebnis auf den **Stack** geschrieben werden soll. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` einen **Datentyp** transportiert, der für den **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Stack(num)`	Holt sich z.B. für eine Berechnung einen zuvor auf den **Stack** geschriebenen Wert vom **Stack**, der `num` Speicherzellen relativ zum `SP`-Register steht.
`Stackframe(num)`	Holt sich den **Wert** einer **Variable**, eines **Verbundsattributs**, eines **Feldelements** etc., der `num` + 2 Speicherzellen relativ zum `BAF`-Register steht.
`Global(num)`	Holt sich den **Wert** einer **Variable**, eines **Verbundsattributs**, eines **Feldelements** etc., der `num` Speicherzellen relativ zum `DS`-Register steht.
`PntrDecl(num, datatype)`	Container, der für den **Zeigerdatentyp** steht: `<prim_dt> *<var>`. Hierbei gibt das **Attribut** `num` die **Anzahl zusammengefasster Zeiger** an und `datatype` ist der **Datentyp**, auf den der letzte dieser **Zeiger** zeigt.
`Ref(exp, datatype, error_data)`	Steht für die Anwendung des **Referenz-Operators** `&<var>`, der die **Adresse** einer **Location** (Definition 2.49) auf den Stack schreiben soll, die über `exp` bestimmt ist. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` einen **Datentyp** transportiert, der für den **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Deref(exp1, exp2)`	Container für den **Indexzugriff** auf einen **Feld-** oder **Zeigerdatentyp**: `<var>[<i>]`. Hierbei ist `exp1` ein angehängtes weiteres `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` und `exp2` ist der **Index** auf den zugegriffen werden soll.
`ArrayDecl(nums, datatype)`	Container, der für den **Felddatentyp** steht: `<prim_dt> <var>[<i>]`. Hierbei ist das **Attribut** `nums` eine Liste von `Num('x')`, welche die **Dimensionen** des Felds angibt und `datatype` ist ein **Unterdatentyp** (Definition 3.9).
`Array(exps, datatype)`	Container für den **Initialisierer** eines **Feldes**, z.B. {{`1, 2`}, {`3, 4`}}, dessen Attribut `exps` weitere Initialisierer für ein **Feld**, weitere Initialisierer für einen **Verbund** oder **Logische Ausdrücke** beinhalten kann. Des Weiteren besitzt es ein verstecktes Attribut `datatype`, welches für den **PicoC-ANF Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind.
`Subscr(exp1, exp2)`	Container für den **Indexzugriff** auf einen **Feld-** oder **Zeigerdatentyp**: `<var>[<i>]`. Hierbei ist `exp1` ein angehängtes weiteres `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` und `exp2` ist der **Index** auf den zugegriffen werden soll.
`StructSpec(name)`	Container für die **Spezifikation** eines **Verbundstyps**: `struct <name>`. Hierbei legt das **Attribut** `name` fest, welchen selbst definierten **Verbundstyp** dieser Knoten spezifiziert.
`Attr(exp, name)`	Container für den **Zugriff** auf ein **Verbundsattribut**: `<var>.<attr>`. Hierbei kann `exp` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation oder ein `Name('var')` sein. Das Attribut `name` ist das **Verbundsattribut** auf das zugegriffen werden soll.

Tabelle 3.4: *PicoC-Knoten Teil 2.*

PiocC-Knoten	Beschreibung
`Struct(assigns, datatype)`	Container für den **Initialisierer** eines **Verbundes**, z.B {.<attr1>={1,2},.<attr2>={3,4}}, dessen Attribut `assigns` eine Liste von `Assign(exp1, exp2)` ist, mit der Zuordnung eines **Attributezeichners** zu einem **Initialisierer** für ein **Feld** oder einen **Initialisierer** für einen **Verbund** oder einem **Logischen Ausdruck**. Des Weiteren besitzt er ein versteckes Attribut `datatype`, welches für den **PicoC-ANF Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind.
`StructDecl(name, allocs)`	Container für die **Deklaration** eines **selbstdefinierten Verbundstyps**, z.B. `struct <var> {<datatype> <attr1>; <datatype> <attr2>;};`. Hierbei ist `name` der **Bezeichner** des **Verbundstyps** und `allocs` eine Liste von Bezeichnern der **Verbundsattribute** mit dazugehörigem **Datentyp**, wofür sich der Knoten `Alloc(type_qual, datatype, name)` sehr gut als Container eignet.
`If(exp, stmts)`	Container für eine **If-Anweisung** `if(<exp>) { <stmts> }` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`IfElse(exp, stmts1, stmts2)`	Container für eine **If-Else Anweisung** `if(<exp>) { <stmts1> } else { <stmts2> }` inklusive **Bedingung** `exp` und 2 **Branches** `stmts1` und `stmts2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Anweisungen** stehen.
`While(exp, stmts)`	Container für eine **While-Anweisung** `while(<exp>) { <stmts> }` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`DoWhile(exp, stmts)`	Container für eine **Do-While-Anweisung** `do { <stmts> } while(<exp>);` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`Call(name, exps)`	Container für einen **Funktionsaufruf** `fun_name(exps)`, wobei `name` der **Bezeichner** der Funktion `fun_name` ist, die aufgerufen werden soll und `exps` eine **Liste von Argumenten** ist, die an diese Funktion übergeben werden soll.
`Return(exp)`	Container für eine **Return-Anweisung** `return <exp>`, wobei das **Attribut** `exp` einen **Logischen Ausdruck** darstellt, dessen Ergebnis von der **Return-Anweisung** zurückgegeben wird.
`FunDecl(datatype, name, allocs)`	Container für eine **Funktionsdeklaration** `<datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>)`, wobei `datatype` der **Rückgabewert** der Funktion ist, `name` der **Bezeichner** der Funktion ist und `allocs` die **Parameter** der Funktion sind. Der Knoten `Alloc(type_spec, datatype, name)` dient dabei als Container für die **Parameter** in `allocs`.

Tabelle 3.5: *PicoC-Knoten Teil 3.*

PiocC-Knoten	Beschreibung
`FunDef(datatype, name, allocs, stmts_blocks)`	Container für eine **Funktionsdefinition** `<datatype> <fun_name>(<datatype> <param>) {<stmts>}`, wobei `datatype` der **Rückgabewert** der Funktion ist, `name` der **Bezeichner** der Funktion ist, `allocs` die **Parameter** der Funktion sind und `stmts_blocks` eine Liste von **Statemetns** bzw. **Blöcken** ist, welche diese Funktion beinhaltet. Der Knoten `Alloc(type_spec, datatype, name)` dient dabei als Container für die **Parameter** in `allocs`.
`StackMalloc(num)`	siehe Tabelle 3.20
`NewStackframe(fun_name, goto_after_call)`	siehe Tabelle 3.20
`RemoveStackframe()`	siehe Tabelle 3.20
`File(name, decls_defs_blocks)`	Container der eine **Datei** repräsentiert, wobei `name` der **Dateiname** der Datei ist und `decls_defs_blocks` eine Liste von **Funktionen** bzw. **Blöcken** ist.
`Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`	Container für **Anweisungen**, wobei das Attribut `name` der Bezeichner des **Labels** (Definition 5.2) des Blocks ist und `stmts_instrs` eine **Liste von Anweisungen** oder **Befehlen** ist. Zudem besitzt er noch 4 versteckte Attribute, wobei `instrs_before` die Zahl der **Befehle vor** diesem **Block** zählt, `num_instrs` die Zahl der **Befehle ohne Kommentare** in diesem Block zählt, `param_size` die voraussichtliche Anzahl an **Speicherzellen aufaddiert**, die für die **Parameter** der Funktion belegt werden müssen und `local_vars_size` die voraussichtliche Anzahl an **Speicherzellen aufaddiert**, die für die **lokalen Variablen** der Funktion belegt werden müssen.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar** (`//`, `/* <comment> */`), den der Compiler selbst während des **Kompiliervorgangs** erstellt, damit die **Zwischenschritte** der Kompilierung und auch der **finale RETI-Code** bei Betrachtung ausgegebener **Abstrakter Syntaxbäume** der verschiedenen Passes **leichter verständlich** sind.
`RETIComment(str)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** später sichtbar ist und zur Orientierung genutzt werden kann. So ein **Kommentar** wäre allerdings in einer tatsächlichen Implementierung einer RETI-CPU **nicht umsetzbar** und eine Umsetzung wäre auch nicht sinnvoll. Der **Kommentar** ist im Attribut `str`, welches jeder Knoten besitzt gespeichert.

Tabelle 3.6: *PicoC-Knoten Teil 4.*

Anmerkung

Die ausgegrauten Attribute der PicoC-Knoten sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Knoten** mit einem Wert **initialisiert** werden. Diese Attribute bekommen im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen, um im weiteren Kompiliervorgang **Informationen** zu transportieren. Das sind **Informationen**, die später im Kompiliervorgang **nicht** mehr so **leicht zugänglich** sind, wie zu dem Zeitpunkt, zu dem sie zugewiesen werden.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** `value` und `position`. Das **Attribut** `value`

entspricht bei einem **Blatt** dem **Tokenwert** des Tokens welches es ersetzt. Bei **Inneren Knoten** ist das Attribut `value` hingegen unbesetzt. Das **Attribut** `position` wird für **Fehlermeldungen** gebraucht.

3.2.5.2 RETI-Knoten

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen einen **Bestandteil eines Befehls** aus der Sprache L_{RETI} darstellen. Für die **RETI-Knoten** wurden aus bereits in Unterkapitel 3.2.5.1 erläutertem Grund, genauso wie für die **RETI-Knoten** möglichst **kurze** und **leicht** verständliche Bezeichner gewählt. Alle **RETI-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakten Syntaxbäumen** vorkommen sind in Tabelle 3.2.5.1 mit einem **Beschreibungstext** dokumentiert.

RETI-Knoten	Beschreibung
`Program(name, instrs)`	Container der ein Programm repräsentiert: `<name> <instrs>`. Hierbei ist `name` der **Name** des **Programms**, welches ausgeführt werden soll und `instrs` ist eine **Liste von Befehlen**.
`Instr(op, args)`	Container für einen **Befehl**: `<op> <args>`. Hierbei ist `op` eine **Operation** und `args` eine **Liste von Argumenten** für diese Operation.
`Jump(rel, im_goto)`	Container für einen **Sprungbefehl**: `JUMP<rel> <im>`. Hierbei ist `rel` eine **Relation** und `im_goto` ist ein **Immediate Value** `Im(str)` für die **Anzahl an Speicherzellen**, um die relativ zum **Sprungbefehl** gesprungen werden soll. In einigen Fällen ist `im` ein `GoTo(Name('block.xyz'))`, das später im **RETI-Patch Pass** durch einen passenden **Immediate Value** ersetzt wird.
`Int(num)`	Container für den **Aufruf einer Interrupt-Service-Routine**: `INT <im>`. Hierbei ist `num` die **Interrruptvektornummer** (IVN) für die passende Adresse in der **Interruptvektortabelle**, in der die Adresse der **Interrupt-Service-Routine** (ISR) steht, die man aufrufen will.
`Call(name, reg)`	Container für einen **Prozeduraufruf**: `CALL <name> <reg>`. Hierbei ist `name` der **Bezeichner** der Prozedur, die aufgerufen werden soll und `reg` ist ein **Register**, das als **Argument** an die Prozedur dient. Diese **Operation** ist in der Betriebssysteme Vorlesung[a] **nicht deklariert**, sondern wurde hinzugefügt, um unkompliziert ein `CALL PRINT ACC` oder `CALL INPUT ACC` im RETI-Interpreter simulieren zu können.
`Name(str)`	Bezeichner für eine **Prozedur**, z.B. `PRINT`, `INPUT` oder den **Programnamen**, z.B. `PROGRAMNAME`. Hierbei ist `str` eine **Zeichenkette** für welche das gleich gilt, wie für das `str` Attribut des **PicoC-Knoten** `Name(str)`. Dieses **Argument** ist in der Betriebssysteme Vorlesung[a] **nicht deklariert**, sondern wurde hinzugefügt, um Bezeichner, wie `PRINT`, `INPUT` oder `PROGRAMNAME` schreiben zu können.
`Reg(reg)`	Container für ein **Register**. Hierbei ist `reg` das **Register** auf welches zugegriffen werden soll.
`Im(str)`	Ein **Immediate Value** (z.B. `42`, `-3` usw.). Hierbei ist das Attribut `str` eine Zeichenkette, welche **beliebige Ziffern** (`0 - 9`) enthalten kann. Es darf nur **nicht** eine `0` am Anfang stehen und danach **weitere Ziffern** folgen. Der **Wert**, welcher durch die Zeichenkette dargestellt wird, darf **nicht kleiner** als -2^{21} oder **größer** als $2^{21} - 1$ sein.
`Add(), Sub(), Mult(), Div(), Mod(), Xor(), Or(), And()`	**Compute-Memory** oder **Compute-Register** Operationen: `ADD, SUB, MULT, DIV, OPLUS, OR, AND`.
`Addi(), Subi(), Multi(), Divi(), Modi(), Xori(), Ori(), Andi()`	**Compute-Immediate** Operationen: `ADDI, SUBI, MULTI, DIVI, MODI, OPLUSI, ORI, ANDI`.
`Load(), Loadin(), Loadi()`	**Load** Operationen: `LOAD, LOADIN, LOADI`.
`Store(), Storein(), Move()`	**Store** Operationen: `STORE, STOREIN, MOVE`.
`Lt(), LtE(), Gt(), GtE(), Eq(), NEq(), Always(), NOp()`	**Relationen** `rel`: `<, <=, >, >=, ==, !=, _NOP`.
`Rti()`	**R**eturn-**F**rom-**I**nterrupt Operation: `RTI`.
`Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()`	**Register** `reg`: `PC, IN1, IN2, ACC, SP, BAF, CS, DS`.

[a] C. Scholl, „Betriebssysteme“

Tabelle 3.7: *RETI-Knoten.*

3.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung

In Tabelle 3.8 sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben.

Komposition	Beschreibung
`Ref(Global(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum `DS`-Register steht auf den **Stack**.
`Ref(Stackframe(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die $\texttt{Num}('\texttt{addr}') + 2$ Speicherzellen **relativ** zum `BAF`-Register steht auf den **Stack**.
`Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))), datatype)`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Stack(Num('addr1'))` steht und dem **Index**, der an Speicherzelle `Stack(Num('addr2'))` steht und speichert diese auf den **Stack**. Die Berechnung ist abhängig davon, ob der **Datentyp** im versteckten Attribut `datatype` ein `ArrayDecl(datatype)` oder `PntrDecl(datatype)` ist.
`Ref(Attr(Stack(Num('addr1')), Name('attr')), datatype)`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Stack(Num('addr1'))` steht und dem **Attributnamen** `Name('attr')` und speichert diese auf den **Stack**. Zur Berechnung ist der Name `Name('st')` des **Verbundstyps** in `StructSpec(Name('st'))` notwendig mit dem diese Berechnung durchgeführt wird. Dieser **Verbundstyp** ist im versteckten Attribut `datatype` zu finden. Dabei muss dieser **Datentyp** im versteckten Attribut `datatype` ein `StructSpec(name)` sein, da diese Berechnung nur bei einem **Verbund** durchgeführt werden kann.
`Assign(Stack(Num('size'))), Global(Num('addr')))`	Schreibt `Num('size')` viele **Speicherzellen**, die `Num('addr')` viele Speicherzellen **relativ** zum `DS`-Register stehen, versetzt genauso auf den **Stack**.
`Assign(Stack(Num('size')), Stackframe(Num('addr')))`	Schreibt `Num('size')` viele **Speicherzellen**, die $\texttt{Num}('\texttt{addr}') + 2$ viele Speicherzellen **relativ** zum `BAF`-Register stehen, versetzt genauso auf den **Stack**.
`Assign(Stack(Num('addr1')), Stack(Num('addr2')))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr2')` Speicherzellen **relativ** zum `SP`-Register steht an der **Adresse** in der Speicherzelle, die `Num('addr1')` Speicherzellen **relativ** zum `SP`-Register steht.
`Assign(Global(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt zu den **Globalen Statischen Daten** ab einer `Num('addr')` viele Speicherzellen **relativ** zum `DS`-Register liegenden Adresse.
`Assign(Stackframe(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt in den **Stackframe** der momentan aktiven Funktion ab einer `Num('addr')` viele Speicherzellen **relativ** zum `BAF`-Register liegenden Adresse.
`Exp(Global(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum `DS`-Register steht auf den **Stack**.
`Exp(Stackframe(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die $\texttt{Num}('\texttt{addr}') + 2$ Speicherzellen **relativ** zum `BAF`-Register steht auf den **Stack**.
`Exp(Stack(Num('addr')))`	Speichert **Inhalt** der Speicherzelle an der **Adresse**, die in der Speicherzelle gespeichert ist, die `Num('addr')` viele Speicherzellen **relativ** zum `SP`-Register liegt auf den **Stack**.
`Exp(Reg(reg))`	Schreibt den aktuellen Wert des **Registers** `reg` auf den **Stack**.
`Exp(GoTo(name))`	siehe Tabelle 3.20
`Instr(Loadi(), [Reg(reg), GoTo(Name('addr@next_instr'))])`	siehe Tabelle 3.20

Tabelle 3.8: *Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung.*

Anmerkung

Um die obige Tabelle 3.8 **nicht** mit unnötig viel **repetetiven Inhalt** zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur ein `exp`i, i ::= ”1” | ”2” | ε durch `Stack(Num('`x`'))`, $x \in \mathbb{N}$ ersetzt wurde[a].

Zudem sind auch jegliche Kombinationen **ausgelassen**, bei denen nur ein **Ausdruck** an ein `Exp(exp)` bzw. `Ref(exp)` drangehängt wurde[b].

[a] Wie z.B. bei `BinOp(Stack(Num('2')), Add(), Stack(Num('1')))`.
[b] Wie z.B. bei `Exp(Num('42'))`.

3.2.5.4 Abstrakte Grammatik

Die **Abstrakte Grammatik** der Sprache L_{PicoC} ist in Grammatik 3.2.10 dargestellt.

$stmt$	$::=$	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	$::=$	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	$::=$	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	$::=$	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
$stmt$	$::=$	$Exp(\langle exp\rangle)$	
un_op	$::=$	$LogicNot()$	*L_Logic*
rel	$::=$	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	$::=$	$LogicAnd() \mid LogicOr()$	
exp	$::=$	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle) \mid ToBool(\langle exp\rangle)$	
$type\_qual$	$::=$	$Const() \mid Writeable()$	*L_Assign_Alloc*
$datatype$	$::=$	$IntType() \mid CharType() \mid VoidType()$	
exp	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
$stmt$	$::=$	$Assign(\langle exp\rangle, \langle exp\rangle)$	
$datatype$	$::=$	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	$::=$	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
$datatype$	$::=$	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	$::=$	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	$::=$	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	$::=$	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	$\mid$	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
$decl\_def$	$::=$	$StructDecl(Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
$stmt$	$::=$	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	$\mid$	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
$stmt$	$::=$	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	$\mid$	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	$::=$	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
$stmt$	$::=$	$Return(\langle exp\rangle)$	
$decl\_def$	$::=$	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle stmt\rangle*)$	
$file$	$::=$	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 3.2.10: *Abstrakte Grammatik der Sprache L_{PiocC} in ASF*

3.2.5.5 Codebeispiel

In Code 3.5 ist der **Abstrakte Syntaxbaum** zu sehen, der aus dem **vereinfachten Ableitungsbaum** aus Code 3.4 mithilfe eines **Transformers** (Definition 2.38) generiert wurde. Das Beispiel aus Code 3.1 wird hier **fortgeführt**.

```
File
  Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc
          Writeable,
          PntrDecl
            Num '1',
            ArrayDecl
              [
                Num '4',
                Num '5'
              ],
              PntrDecl
                Num '1',
                IntType 'int',
          Name 'attr'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '3',
                Num '2'
              ],
              PntrDecl
                Num '1',
                PntrDecl
                  Num '1',
                  StructSpec
                    Name 'st',
            Name 'var'
      ]
  ]
```

Code 3.5: *Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum.*

3.2.5.6 Ausgabe des Abstrakten Syntaxbaumes

Ein **Teilbaum** eines **Abstrakten Syntaxbaumes** kann entweder in der **Konkreten Syntax** der Sprache, für dessen Kompilierung er generiert wurde oder in der **Abstrakten Syntax**, die beschreibt, wie der Abstrakte Syntaxbaum selbst aufgebaut sein darf ausgegeben werden.

Das Ausgeben eines **Abstrakten Syntaxbaumes** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`[18] der Programmiersprache L_{Python} umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkreten**

[18] Spezielle Methode, die immer aufgerufen wird, wenn das **Objekt**, dass in Besitz dieser Methode ist als **Zeichenkette** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.

Syntax aufgebaute **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **runden öffnenden** `(` und **schließenden** `)` **Klammern**, sowie **Kommas** `','`, **Semikolons** `;` usw. zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Der gesamte **Abstrakte Syntaxbaum** wird durchlaufen und die Magischen `__repr__()`-Methoden der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methoden ihrer Kinder aufrufen und die zurückgegebenen **Textrepräsentationen** passend **zusammenfügen** und selbst **zurückgeben**.

Beim **PicoC-Compiler** sind **Abstrakte** und **Konkrete Syntax** miteinander gemischt. Für **PicoC-Knoten** wird die **Abstrakte Syntax** verwendet, da Passes schließlich auf **Abstrakten Syntaxbäumen** operieren. Bei **RETI-Knoten** wird die **Konkrete Syntax** verwendet, da **Maschinenbefehle** in **Konkreter Syntax** schließlich das **Endprodukt** des Kompiliervorgangs sein sollen.

Da die **Konkrette Syntax** von **RETI-Knoten** sehr simpel ist, macht es kaum einen Unterschied in der Erkennbarkeit, bis auf fehlende gescheifte Klammern `()` usw., ob man die **RETI-Knoten** in **Abstrakter** oder **Konkreter Syntax** schreibt. Daher werden die **RETI-Knoten** einfach immer direkt in **Konkreter Syntax** ausgegeben. Auf diese Weise muss nicht beim **letzten Pass** daran gedacht werden am Ende die **Konkrete**, statt der **Abstrakten Syntax** für die **RETI-Knoten** auszugeben.

Die Ausgabe des **Abstrakten Syntaxbaums** ist bewusst so gewählt, dass sie sich optisch von der des **Ableitungsbaums** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase**[19] geschrieben sind. Das steht im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, da sie die **Nicht-Terminalsymbole** auf den **linken Seiten** des `::=`-Symbols in der **Konkreten Grammatik** 3.2.8 darstellen, welche in **snake_case** geschrieben sind.

3.3 Code Generierung

Nach der Generierung eines **Abstrakten Syntaxbaums** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 3.2.5, wird in diesem Kapitel auf Basis der verschiedenen **Kompositionen** von Knoten im **Abstrakten Syntaxbaum** das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** der Sprache L_{PicoC}, der durch die **Abstrakte Grammatik** 3.2.10 spezifiziert ist in einen **semantisch** gleichen **Abstrakten Syntaxbaum** der Sprache L_{RETI} umzuformen, der durch die **Abstrakte Grammatik** 3.3.6 spezifiziert ist. In **T-Diagramm-Notation** (siehe Unterkapitel 2.1.1) lässt sich das darstellen, wie in Abbildung 3.6.

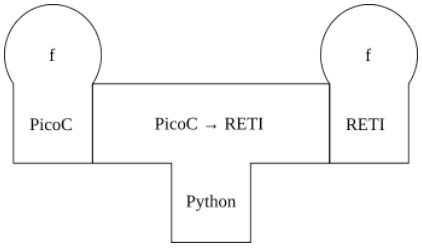


Abbildung 3.6: *Kompiliervorgang Kurzform.*

Das gerade angesprochene Problem lässt sich, wie in Unterkapitel 2.5 bereits beschrieben **vereinfachen**, indem man das Problem in mehrere **Passes** (Definition 2.43) aufteilt, die jeweils ein **überschaubares Teilproblem** lösen. Man nähert sich Schrittweise immer mehr der **Syntax** der Sprache L_{RETI} an.

[19] *Naming convention (programming).*

In Abbildung 3.7 ist das **T-Diagramm** aus Abbildung 3.6 detailierter dargestellt. Das T-Diagramm gibt einen Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 2.1) des **PicoC-Compilers** aufeinanderfolgen. In der **Pipe-Architektur** nutzt der jeweils nächste **Pass** den generierten **Abstrakten Syntaxbaum** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstrakten Syntaxbaum** in seiner eigenen **Sprache** zu generieren.

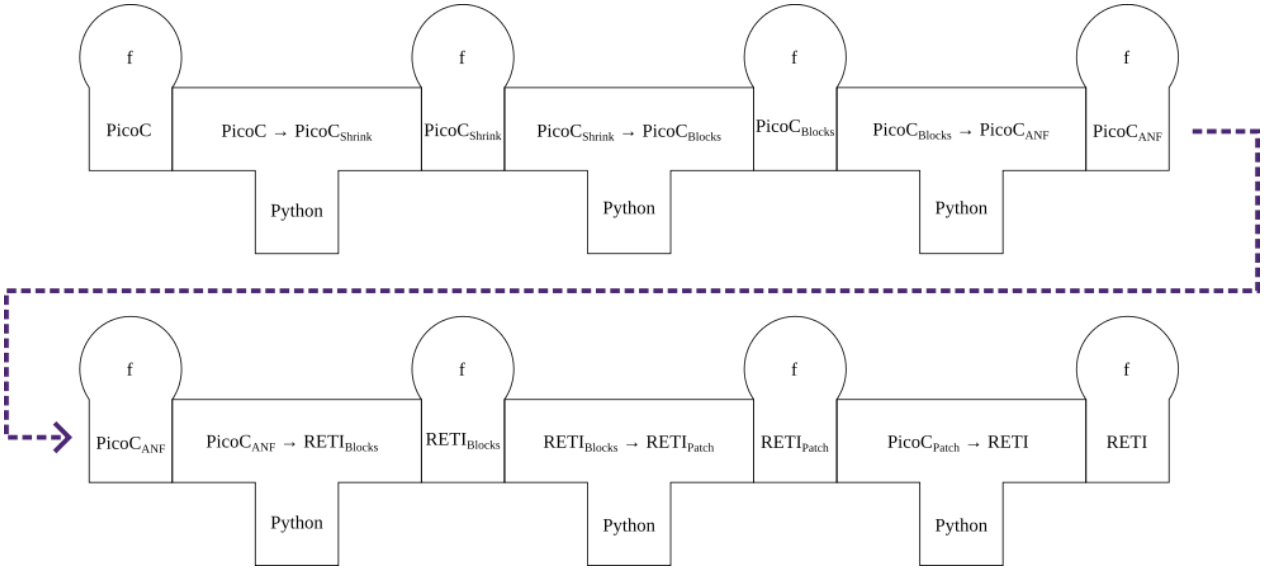


Abbildung 3.7: *Architektur mit allen Passes ausgeschrieben.*

Im Unterkapitel 3.3.1 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Zeigern**, **Feldern**, **Verbunden** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 3.3.1 nicht vertieft wurden. Viele der verwendenten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung C. Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

3.3.1 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der veschiedenen Aspekte[20] der **Bachelorarbeit** wichtig.

Auf jedes Detail der einzelnen **Passes** wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Zeigern**, **Feldern**, **Verbunden** und **Funktionen** im Detail erklärt sind und andererseits viele Aufgaben dieser **Passes** eher dem **Bachelorprojekt** zuzurechnen sind.

3.3.1.1 PicoC-Shrink Pass

Die Aufgabe des **PicoC-Shrink Pass** ist in Unterkapitel 3.3.2.2 ausführlich an einem Beispiel erklärt. Kurzgefasst hat der **PicoC-Shrink Pass** die Aufgabe, die Eigenheit auszunutzen, dass der **Dereferenzierungoperator** `*pntr` und die damit einhergehende **Zeigerarithmetik** `*(pntr + i)` in der Untermenge der Sprache L_C, welche die Sprache L_{PicoC} darstellt die **gleiche Semantik** hat, wie der **Operator** für den

[20] In kurz: **Zeiger**, **Felder**, **Verbunde** und **Funktionen**.

Zugriff auf den **Index** eines **Feldes** `ar[i]`[21].

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** `Deref(exp, i)` im jeweiligen **Abstrakten Syntaxbaum** in **Knoten** `Subscr(exp, i)` um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung vermeiden lassen. Man lässt die **Derefenzierung** `*(var + i)` einfach von den Routinen für den **Zugriff auf einen Feldindex** `var[i]` übernehmen.

3.3.1.1.1 Abstrakte Grammatik

Die **Abstrakte Grammatik** 3.3.1 der Sprache L_{PicoC_Shrink} ist fast **identisch** mit der **Abstrakten Grammatik** 3.2.10 der Sprache L_{PiocC}, nach welcher der **erste** Abstrakte Syntaxbaum in der **Syntaktischen Analyse** generiert wurde. Der einzige **Unterschied** liegt darin, dass es den Knoten `Deref(exp1, exp2)` in der Abstrakten Grammatik 3.3.1 **nicht** mehr gibt. Das liegt daran, dass dieser Pass alle **Vorkommnisse** des Knoten `Deref(exp1, exp2)` durch den Knoten `Subscr(exp1, exp2)` auswechselt.

[21] Wobei `*pntr` und `pntr[0]` einander entsprechen.

stmt	::=	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
stmt	::=	$Exp(\langle exp\rangle)$	
un_op	::=	$LogicNot()$	*L_Logic*
rel	::=	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	::=	$LogicAnd() \mid LogicOr()$	
exp	::=	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle) \mid ToBool(\langle exp\rangle)$	
type_qual	::=	$Const() \mid Writeable()$	*L_Assign_Alloc*
datatype	::=	$IntType() \mid CharType() \mid VoidType()$	
exp	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	|	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
stmt	::=	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	|	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
stmt	::=	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	|	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	::=	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	|	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle stmt\rangle*)$	
file	::=	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 3.3.1: *Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF*

Anmerkung

Alles ausgegraute bedeutet, es hat sich im Vergleich zur letzten Abstrakten Grammatik **nichts** geändert. Alles rot markierte bedeutet, es wurde **entfernt** oder **abgeändert**. Alle normal in `schwarz` geschriebenen Knoten sind **neu** hinzugefügt. Das gilt genauso für **alle** folgenden **Grammatiken**.

3.3.1.1.2 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 3.6 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 3.6 ist in der Funktion `faculty` ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen Arguments berechnet. Der Algorithmus basiert auf einem **Beispielprogramm** aus der Vorlesung C. Scholl, „Betriebssysteme“, welches diesen Algorithmus allerdings **rekursiv** implementiert.

Die **rekursive** Implementierung des Algorithmus wäre allerdings **kein** gutes **Anschauungsbeispiel**, das viele der **Aufgaben** der verschiedenen **Passes** bei der Kompilierung **veranschaulicht** hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von `if-`, `if-else-`, `while-` und `do-while`-Anweisungen, wären mit der **rekursiven** Implementierung aus der Vorlesung **nicht** veranschaulicht gewesen. Daher wurde die **rekursive Implementierung** aus der Vorlesung zu einem **iterativen** Algorithmus 3.6 umgeschrieben, um unter anderem auch `if-` und `while`-Statements zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter Link[22], unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der im jeweiligen Unterkapitel beschrieben wird und werden nicht im Detail erläutert. Viele Details der Passes werden später in den Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Zeigern**, **Feldern**, **Verbunden** und **Funktionen** mit eigenen **Codebeispielen** genauer erklärt und alle sonstigen Details sind dem **Bachelorprojekt** zuzuordnen, dessen Aspekte in dieser schriftlichen Ausarbeitung der Bachelorarbeit **nicht** im Detail erläutert werden.

```
1  // based on a example program from Christoph Scholl's Operating Systems lecture
2
3  int faculty(int n){
4    int res = 1;
5    while (1) {
6      if (n == 1) {
7        return res;
8      }
9      res = n * res;
10     n = n - 1;
11   }
12 }
13
14 void main() {
15   print(faculty(4));
16 }
```

Code 3.6: *PicoC Code für Codebespiel.*

In Code 3.7 sieht man den **Abstrakten Syntaxbaum**, der in der **Syntaktischen Analyse** generiert wurde.

```
1  File
2    Name './example_faculty_it.ast',
3    [
4      FunDef
5        IntType 'int',
```

[22] https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```
6       Name 'faculty',
7       [
8         Alloc(Writeable(), IntType('int'), Name('n'))
9       ],
10      [
11        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
12        While
13          Num '1',
14          [
15            If
16              Atom(Name('n'), Eq('=='), Num('1')),
17              [
18                Return(Name('res'))
19              ]
20            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22          ]
23      ],
24    FunDef
25      VoidType 'void',
26      Name 'main',
27      [],
28      [
29        Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30      ]
31  ]
```

Code 3.7: *Abstrakter Syntaxbaum für Codebespiel.*

Im **PicoC-Shrink Pass** ändert sich nichts im Vergleich zum **Abstrakten Syntaxbaum** in Code 3.7, da das Codebeispiel keine **Dereferenzierung** `*pntr` enthält. Es wurde auf ein weiteres Codebeispiel für diesen **Pass** verzichtet, da din diesem das gleiche zu sehen wäre, wie in Codebeispiel 3.7.

3.3.1.2 PicoC-Blocks Pass

Die Aufgabe des **PicoC-Blocks Pass** ist es die Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs-`, `GoTo(lable)-` und `IfElse(exp, stmts1, stmts2)`-Knoten umzusetzen. Der `IfElse(exp, stmts1, stmts2)`-Knoten wird zur Umsetzung der **Bedingung** verwendet und es wird, je nachdem, ob die Bedingung **wahr** oder **falsch** ist mithilfe der `GoTo(label)`-Knoten in einen von zwei **alternativen Branches** gesprungen oder ein **Branch** erneut aufgerufen usw.

3.3.1.2.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Grammatik** 3.3.1 der Sprache L_{PicoC_Shrink} um die Knoten zu erweitern, die am Anfang dieses Unterkapitels erwähnt wurden. Die Knoten `If(exp, stmts)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` gibt es nicht mehr, da sie durch `Block(name, stmts_instrs-`, `GoTo(lable)-` und `IfElse(exp, stmts1, stmts2)`-Knoten ersetzt wurden. Die **Funktionsdefinition** `FunDef(datatype, Name(str), Alloc(Writeable(), datatype, Name(str))*, ⟨block⟩*)` ist nun ein Container für **Blöcke** `Block(Name(str),stmt*)` und **keine Anweisungen** `stmt` mehr. Das resultiert in der **Abstrakten Grammatik** 3.3.2 der Sprache L_{PicoC_Blocks}.

stmt	::=	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
stmt	::=	$Exp(\langle exp\rangle)$	
un_op	::=	$LogicNot()$	*L_Logic*
rel	::=	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	::=	$LogicAnd() \mid LogicOr()$	
exp	::=	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle) \mid ToBool(\langle exp\rangle)$	
type_qual	::=	$Const() \mid Writeable()$	*L_Assign_Alloc*
datatype	::=	$IntType() \mid CharType() \mid VoidType()$	
exp	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	|	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
stmt	::=	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	|	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
stmt	::=	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	|	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	::=	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	|	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle block\rangle*)$	
block	::=	$Block(Name(\langle str\rangle), \langle stmt\rangle*)$	*L_Blocks*
stmt	::=	$GoTo(Name(\langle str\rangle))$	
file	::=	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 3.3.2: *Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF*

Anmerkung

Eine **Abstrakte Grammatik** soll im Gegensatz zu einer **Konkreten Grammatik** für den **Programmierer**, der einen darauf aufbauenden Compiler implementiert **einfach verständlich** sein und stellt daher eine **Obermenge** aller tatsächlich möglichen **Kompositionen** von **Knoten** dar[a].

[a] D.h. auch wenn dort `exp` als **Attribut** steht, kann dort **nicht** jeder Knoten, der sich aus dem **Nicht-Terminalsymbol** `exp` ergibt auch wirklich eingesetzt werden.

3.3.1.2.2 Codebeispiel

In Code 3.8 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 3.7 mithilfe des **PicoC-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel in Code 3.6 aus Unterkapitel 3.3.1.1 weitergeführt. Es wurden nun eigene **Blöcke** für die Funktion `faculty` und die `main`-Funktion erstellt, in denen die jeweils **ersten** Anweisungen der jeweiligen Funktionen bis zur **letzten** Anweisung oder bis zum ersten **Auftauchen** eines `If(exp, stmts)`-, `IfElse(exp, stmts1, stmts2)`-, `While(exp, stmts)`- oder `DoWhile(exp, stmts)`-Knoten stehen. Je nachdem, ob ein `If(exp, stmts)`-, `IfElse(exp, stmts1, stmts2)`-, `While(exp, stmts)`- oder `DoWhile(exp, stmts)`-Knoten auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      IntType 'int',
      Name 'faculty',
      [
        Alloc(Writeable(), IntType('int'), Name('n'))
      ],
      [
        Block
          Name 'faculty.6',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.5'))
          ],
        Block
          Name 'condition_check.5',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.4'))
              ],
              [
                GoTo(Name('while_after.1'))
              ]
          ],
        Block
          Name 'while_branch.4',
          [
            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
            IfElse
              Atom(Name('n'), Eq('=='), Num('1')),
              [
                GoTo(Name('if.3'))
              ],
              [
                GoTo(Name('if_else_after.2'))
```

```
41              ]
42            ],
43          Block
44            Name 'if.3',
45            [
46              Return(Name('res'))
47            ],
48          Block
49            Name 'if_else_after.2',
50            [
51              Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52              Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53              GoTo(Name('condition_check.5'))
54            ],
55          Block
56            Name 'while_after.1',
57            []
58        ],
59      FunDef
60        VoidType 'void',
61        Name 'main',
62        [],
63        [
64          Block
65            Name 'main.0',
66            [
67              Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
68            ]
69        ]
70    ]
```

Code 3.8: *PicoC-Blocks Pass für Codebespiel.*

3.3.1.3 PicoC-ANF Pass

Die Aufgabe des **PicoC-ANF Pass** ist es den **Abstrakten Syntaxbaum** der Sprache L_{PicoC_Blocks} in die **Abstrakte Grammatik** der Sprache L_{PicoC_ANF} umzuformen, welche in **A-Normalform** (Definition 2.52) und damit auch in **Monadischer Normalform** (Definition 2.48) ist. Um Redundanz zu vermeiden, wird zur Erklärung der **A-Normalform** auf Unterkapitel 2.5.2 verwiesen und zur Erklärung der **Monadischen Normalform** auf Unterkapitel 2.48 verwiesen.

Zudem wird eine **Symboltabelle** (Definition 3.8) verwendet. In der Symboltabelle werden **Symboltabelleneinträge** erstellt, deren **Attribute** Informationen transportieren, die später im Kompiliervorgang wichtig sind. Die **Verwendungszwecke** der **Attribute** eines Symboltabelleneintrages werden in Tabelle 3.9 erklärt. Beim Erstellen eines Eintrags, wird ein Suffix `@scope` an den Schlüssel des **Assoziativen Feldes**, durch welchem eine **Symboltabelle** üblicherweise umgesetzt wird angehängt. Damit wird die **Zugehörigkeit** zu einem bestimmten **Sichtbarkeitsbereich** von z.B. einer **Funktion** oder zu einem bestimmten **Verbundstyp** umgesetzt. Im Unterkapitel 3.3.6.2 wird die Umsetzung von **Sichtbarkeitsbereichen** im Zusammenhang mit **Funktionen** genauer erklärt. Im Unterkapitel 3.3.4.1 wird die Umsetzung der **Zugehörigkeit** eines **Verbundsattributes** zu einem bestimmen **Verbundstyp** genauer eklärt[23].

[23] Was auch als eine Form von **Sichtbarkeitsbereich** angesehen werden kann.

Bezeichnung des Attributs	Verwendung
`type qualifier`	**Type Qualifier**, wie `Const()` oder `Writeable()` für eine **nicht beschreibbare Konstante** (z.B. `const int var = 42`) oder **beschreibbare Variable**, bei Nicht-Angabe von `const` (z.B. `int var`).
`datatype`	**Datentyp**, **Funktionsprototyp** (Definition 1.6).
`name`	**Bezeichner** von **Konstanten**, **Variablen**, **Funktionen**, **Datentyp** usw. Bei **Variablen**, **Konstanten** und **Verbundsattributen** wird ein **Suffix** `@scope` angehängt, der die Zugehörigkeit zum **Sichtbarkeitsbereich** von z.B. einer bestimmten **Funktion** oder zu einem bestimmten **Verbundstyp** darstellt.
`value or adress`	**Wert** einer **Konstanten**. **Adresse** einer **Variablen** oder **Funktion**. Bei der **Deklaration** eines **Verbundstyps** werden seine **Verbundsattribute** als Liste in diesem Attribut abgespeichert.
`position`	**Position** des **Lexemes** für weches dieser Symboltabelleneintrag angelegt wird, mit **Zeilennummer** und **Spaltennummer** innerhalb der Textdatei eines Programms.
`size`	**Anzahl Speicherzellen**, die eine **Variable** oder ein **Verbundsattribut** eines **Verbundstyps** belegt.

Tabelle 3.9: *Attribute eines Symboltabelleneintrags.*

Definition 3.8: Symboltabelle

*Eine meist über ein **Assoziatives Feld** umgesetzte **Datenstruktur**, die notwendig ist, um das Konzept von **Variablen** und **Konstanten** in einer Sprache umzusetzen. Diese Datenstruktur ordnet jedem **Symbol**[a] einer **Variablen**, **Konstanten** oder **Funktion** aus einem **Programm**, Informationen, wie die **Adresse**, die **Position** im Programmcode oder den **Datentyp** zu, welche später nicht mehr so einfach zugänglich sind, wie zu dem Zeitpunkt, zu dem sie in einen **Symboltabelleneintrag** gespeichert werden.*

*Die **Symboltabelle** muss nur während des **Kompiliervorgangs** im **Speicher** existieren, da die **Einträge** in der Symboltabelle beeinflussen, was für **Maschinencode** generiert wird und dadurch im **Maschinencode** bereits die richtigen **Adressen** usw. angesprochen werden und es die Symboltabelle selbst **nicht** mehr braucht.*

[a] In der **Code Generierung** werden **Bezeichner** als **Symbole** bezeichnet.

In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** an das `value or address`-Attribut dieses **Eintrags** zugewiesen, die dem Wert einer von zwei **Countern** `rel_global_addr` und `rel_stack_addr` entspricht. Der **Counter** `rel_global_addr` ist für Variablen in den **Globalen Statischen Daten** und der **Counter** `rel_stack_addr` ist für Variablen auf dem **Stackframe** der momentan **aktiven Funktion**. Einer der beiden **Counter** wird nach **Anlegen eines Eintrags** entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle eine **definierte** Variable `Name('var')` vor, so wird mit der **Konkatenation** des Bezeichners der **Variable** und des Bezeichners des momentanen **Sichtbarkeitsbereichts** `var@scope` als **Schlüssel** in der **Symboltabelle** der entsprechende **Symboltabelleneintrag** der Variable `var` nachgeschlagen. Anstelle des `Name(str)`-Knotens der Variable wird ein `Global(num)`- bzw. `Stackframe(num)`-Knoten eingefügt, dessen `num`-Attribut die **Adresse** im `value or address`-Attribut des **Symboltabelleneintrags** zugewiesen bekommt.

Ob der `Global(num)`- oder der `Stackframe(num)`-Knoten für die Ersetzung verwendet wird, entscheidet sich anhand des **Sichtbarkeitsbereichs** `scope`. Für den **Sichtbarkeitsbereich** der `main`-Funktion wird der `Global(num)`-Knoten verwendet. Für den **Sichtbarkeitsbereich** jeder anderen Funktion wird der `Stackframe(num)`-Knoten verwendet.

Darüberhinaus gibt es den **Sichtbarkeitsbereich** `global!`, dessen Variablen und Konstanten **überall sichtbar** sind und der für **globale Variablen** verwendet wird. Der **Sichtbarkeitsbereich** `global!` hat die **geringste Priorität** aller Sichtbarkeitsbereiche. Das bedeutet, dass, wenn im **Sichtbarkeitsbereich einer Funktion** und im **Sichtarkeitsbereich** `global!` zweimal der **gleiche Bezeichner** vorkommt, dem Sichtbarkeitsbereich der Funktion **Vorrang** gelassen wird. Für den **Sichtbarkeitsbereich** `global!` wird der `Global(num)`-Knoten verwendet.

Das Symbol `!` im **Suffix** `global!` und das Symbol `@` als Trennzeichen in `var@scope` wurden aus einem bestimmten Grund verwendet, nämlich, weil kein **Bezeichner** die Symbole `@` und `!` jemals selbst enthalten kann. Die **Produktionen** für einen Bezeichner in der **Konkretten Grammatik** für die Lexikalische Analyse G_{Lex} (siehe 3.1.1) lassen beide Symbole `@` und `!` nicht zu. Damit ist es ausgeschlossen, dass es zu **Problemen** kommt, falls ein Benutzer des **PicoC-Compilers** zufällig auf die Idee kommt eine Funktion auf eine unpassende Weise zu benennen.

3.3.1.3.1 Abstrakte Grammatik

Zur Umsetzung dieses **Passes** ist es notwendig die **Abstrakte Grammatik** 3.3.2 der Sprache L_{PicoC_Blocks} in die **A-Normalform** zu bringen. Darunter fällt es unter anderem, dafür zu sorgen, dass **Komplexe Knoten**, wie z.B. `BinOp(exp, bin_op, exp)` nur **Atomare Knoten** enthalten können. Wie es bereits im Unterkapitel 2.5.2 erklärt wurde, ist beim **PicoC-Compiler** der Knoten `Stack(Num(str))` der einzige **Atomare Knoten**.

Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `decls_defs_blocks`-Attribut des `File(Name(str), decls_defs_blocks*)`-Knoten liegen usw.[24]. Die **Symboltabelle** ist ebenfalls als **Abstrakter Syntaxbaum** umgesetzt, wofür in der **Abstrakten Grammatik** neue Knoten eingeführt wurden. Das ganze resultiert in der **Abstrakten Grammatik** 3.3.3 der Sprache L_{PicoC_ANF}.

[24] Im Unterkapitel 3.3.6 wird das **Auflösen von Funktionen** genauer erklärt.

stmt	::=	*SingleLineComment*(⟨*str*⟩, ⟨*str*⟩) | *RETIComment*()	*L_Comment*
un_op	::=	*Minus*() | *Not*()	*L_Arith_Bit*
bin_op	::=	*Add*() | *Sub*() | *Mul*() | *Div*() | *Mod*()	
		| *Oplus*() | *And*() | *Or*()	
exp	::=	*Name*(⟨*str*⟩) | *Num*(⟨*str*⟩) | *Char*(⟨*str*⟩) | *Global*(*Num*(⟨*str*⟩))	
		| *Stackframe*(*Num*(⟨*str*⟩)) | *Stack*(*Num*(⟨*str*⟩))	
		| *BinOp*(*Stack*(*Num*(⟨*str*⟩)), ⟨*bin_op*⟩, *Stack*(*Num*(⟨*str*⟩)))	
		| *UnOp*(⟨*un_op*⟩, *Stack*(*Num*(⟨*str*⟩))) | *Call*(*Name*('*input*'), *Empty*())	
		| *Call*(*Name*('*print*'), ⟨*exp*⟩)	
		| *Exp*(⟨*exp*⟩)	
un_op	::=	*LogicNot*()	*L_Logic*
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
bin_op	::=	*LogicAnd*() | *LogicOr*()	
exp	::=	*Atom*(*Stack*(*Num*(⟨*str*⟩)), ⟨*rel*⟩, *Stack*(*Num*(⟨*str*⟩)))	
		| *ToBool*(*Stack*(*Num*(⟨*str*⟩)))	
type_qual	::=	*Const*() | *Writeable*()	*L_Assign_Alloc*
datatype	::=	*IntType*() | *CharType*() | *VoidType*()	
exp	::=	*Alloc*(⟨*type_qual*⟩, ⟨*datatype*⟩, *Name*(⟨*str*⟩))	
stmt	::=	*Assign*(*Global*(*Num*(⟨*str*⟩)), *Stack*(*Num*(⟨*str*⟩)))	
		| *Assign*(*Stackframe*(*Num*(⟨*str*⟩)), *Stack*(*Num*(⟨*str*⟩)))	
		| *Assign*(*Stack*(*Num*(⟨*str*⟩)), *Global*(*Num*(⟨*str*⟩)))	
		| *Assign*(*Stack*(*Num*(⟨*str*⟩)), *Stackframe*(*Num*(⟨*str*⟩)))	
datatype	::=	*PntrDecl*(*Num*(⟨*str*⟩), ⟨*datatype*⟩)	*L_Pntr*
		| *Ref*(*Global*(⟨*str*⟩)) | *Ref*(*Stackframe*(⟨*str*⟩))	
		| *Ref*(*Subscr*(⟨*exp*⟩, ⟨*exp*⟩ | *Ref*(*Attr*(⟨*exp*⟩, *Name*(⟨*str*⟩)))))	
datatype	::=	*ArrayDecl*(*Num*(⟨*str*⟩)+, ⟨*datatype*⟩)	*L_Array*
exp	::=	*Subscr*(⟨*exp*⟩, *Stack*(*Num*(⟨*str*⟩))) | *Array*(⟨*exp*⟩+)	
datatype	::=	*StructSpec*(*Name*(⟨*str*⟩))	*L_Struct*
exp	::=	*Attr*(⟨*exp*⟩, *Name*(⟨*str*⟩))	
		| *Struct*(*Assign*(*Name*(⟨*str*⟩), ⟨*exp*⟩)+)	
decl_def	::=	*StructDecl*(*Name*(⟨*str*⟩),	
		Alloc(*Writeable*(), ⟨*datatype*⟩, *Name*(⟨*str*⟩))+)	
stmt	::=	*IfElse*(*Stack*(*Num*(⟨*str*⟩)), ⟨*stmt*⟩∗, ⟨*stmt*⟩∗)	*L_If_Else*
exp	::=	*Call*(*Name*(⟨*str*⟩), ⟨*exp*⟩∗)	*L_Fun*
stmt	::=	*StackMalloc*(*Num*(⟨*str*⟩)) | *NewStackframe*(*Name*(⟨*str*⟩), *GoTo*(⟨*str*⟩))	
		| *Exp*(*GoTo*(*Name*(⟨*str*⟩))) | *RemoveStackframe*()	
		| *Return*(*Empty*()) | *Return*(⟨*exp*⟩)	
decl_def	::=	*FunDecl*(⟨*datatype*⟩, *Name*(⟨*str*⟩)	
		Alloc(*Writeable*(), ⟨*datatype*⟩, *Name*(⟨*str*⟩))∗)	
		| *FunDef*(⟨*datatype*⟩, *Name*(⟨*str*⟩),	
		Alloc(*Writeable*(), ⟨*datatype*⟩, *Name*(⟨*str*⟩))∗, ⟨*block*⟩∗)	
block	::=	*Block*(*Name*(⟨*str*⟩), ⟨*stmt*⟩∗)	*L_Blocks*
stmt	::=	*GoTo*(*Name*(⟨*str*⟩))	
file	::=	*File*(*Name*(⟨*str*⟩), ⟨*block*⟩∗)	*L_File*
symbol_table	::=	*SymbolTable*(⟨*symbol*⟩∗)	*L_Symbol_Table*
symbol	::=	*Symbol*(⟨*type_qual*⟩, ⟨*datatype*⟩, ⟨*name*⟩, ⟨*val*⟩, ⟨*pos*⟩, ⟨*size*⟩)	
type_qual	::=	*Empty*()	
datatype	::=	*BuiltIn*() | *SelfDefined*()	
name	::=	*Name*(⟨*str*⟩)	
val	::=	*Num*(⟨*str*⟩) | *Empty*()	
pos	::=	*Pos*(*Num*(⟨*str*⟩), *Num*(⟨*str*⟩)) | *Empty*()	
size	::=	*Num*(⟨*str*⟩) | *Empty*()	

Grammatik 3.3.3: *Abstrakte Grammatik der Sprache L_{PiocC_ANF} in ASF*

3.3.1.3.2 Codebeispiel

In Code 3.9 sieht man die als **Abstrakter Syntaxbaum** umgesetzte **Symboltabelle**, in der alle **Variablen** und **Funktionen** aus dem weitergeführten Beispiel in Code 3.6 aus Unterkapitel 3.3.1.1 einen Eintrag haben.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(IntType('int'), Name('faculty'), [Alloc(Writeable(),
        ↪  IntType('int'), Name('n'))])
        name:                   Name('faculty')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('4'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('n@faculty')
        value or address:       Num('0')
        position:               Pos(Num('3'), Num('16'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('res@faculty')
        value or address:       Num('1')
        position:               Pos(Num('4'), Num('6'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('14'), Num('5'))
        size:                   Empty()
      }
  ]
```

Code 3.9: *Symboltabelle für Codebespiel.*

In Code 3.10 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 3.8 mithilfe des **PicoC-ANF Pass** resultierenden **Abstrakten Syntaxbaum**. Alle Anweisungen und Ausdrücke sind in **A-Normalform**, z.B. wird `IfElse(exp, stmts1, stmts2)` dadurch in **A-Normalform** gebracht, dass der **Komplexe Ausdruck** im `exp`-Attribut in einen `Exp(exp)`-Knoten eingesetzt und vorgezogen wird (z.B. `Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))`).

Die `Exp(exp)`-Knoten weisen die **Ergebnisse** der Komplexen Ausdrücke **Locations** zu, welche sich beim **PicoC-Compiler** auf dem **Stack** befinden. Die **Ergebnisse** werden dann über **Atomare Ausdrücke** `Stack(Num(str))` vom **Stack** gelesen (z.B. `IfElse(Stack(Num(str)), stmts1, stmts2)`).

Funktionen sind nur noch über die `Name(str)`-Knoten von **Blöcken** zu erkennen, die den gleichen **Bezeichner** haben, wie die ursprüngliche **Funktion** und es lässt sich nur durch das **Nachverfolgen** bestimmter `GoTo(Name(str))`-Knoten nachvollziehen, was ursprünglich zur Funktion gehörte.

```
File
  Name './example_faculty_it.picoc_mon',
  [
    Block
      Name 'faculty.6',
      [
        // Assign(Name('res'), Num('1'))
        Exp(Num('1'))
        Assign(Stackframe(Num('1')), Stack(Num('1')))
        // While(Num('1'), [])
        Exp(GoTo(Name('condition_check.5')))
      ],
    Block
      Name 'condition_check.5',
      [
        // IfElse(Num('1'), [], [])
        Exp(Num('1')),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('while_branch.4'))
          ],
          [
            GoTo(Name('while_after.1'))
          ]
      ],
    Block
      Name 'while_branch.4',
      [
        // If(Atom(Name('n'), Eq('==')), Num('1')), [])
        // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
        Exp(Stackframe(Num('0')))
        Exp(Num('1'))
        Exp(Atom(Stack(Num('2')), Eq('==')), Stack(Num('1')))),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('if.3'))
          ],
          [
            GoTo(Name('if_else_after.2'))
          ]
      ],
    Block
      Name 'if.3',
      [
```

```
49        // Return(Name('res'))
50        Exp(Stackframe(Num('1')))
51        Return(Stack(Num('1')))
52      ],
53    Block
54      Name 'if_else_after.2',
55      [
56        // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
57        Exp(Stackframe(Num('0')))
58        Exp(Stackframe(Num('1')))
59        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
60        Assign(Stackframe(Num('1')), Stack(Num('1')))
61        // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
62        Exp(Stackframe(Num('0')))
63        Exp(Num('1'))
64        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
65        Assign(Stackframe(Num('0')), Stack(Num('1')))
66        Exp(GoTo(Name('condition_check.5')))
67      ],
68    Block
69      Name 'while_after.1',
70      [
71        Return(Empty())
72      ],
73    Block
74      Name 'main.0',
75      [
76        StackMalloc(Num('2'))
77        Exp(Num('4'))
78        NewStackframe(Name('faculty.6'), GoTo(Name('addr@next_instr')))
79        Exp(GoTo(Name('faculty.6')))
80        RemoveStackframe()
81        Exp(ACC)
82        Exp(Call(Name('print'), [Stack(Num('1'))]))
83        Return(Empty())
84      ]
85  ]
```

Code 3.10: *PicoC-ANF Pass für Codebespiel.*

3.3.1.4 RETI-Blocks Pass

Die Aufgabe des **RETI-Blocks Pass** ist es die **PicoC-Knoten** der **Anweisungen** in den **Blöcken** des **Abstrakten Syntaxbaums** der Sprache L_{PicoC_ANF} durch **semantisch** entsprechende **RETI-Knoten** zu ersetzen.

3.3.1.4.1 Abstrakte Grammatik

Die **Abstrakte Grammatik** 3.3.4 der Sprache L_{RETI_Blocks} ist verglichen mit der **Abstrakten Grammatik** 3.3.3 der Sprache L_{PicoC_ANF} stark verändert, denn der Großteil der **PicoC-Knoten** wird in diesem Pass durch **semantisch** entsprechende **RETI-Knoten** ersetzt[25]. Die einzigen verbleibenden **PicoC-Knoten** sind `Exp(GoTo(str))`, `Block(Name(str), ⟨instr⟩*)` und `File(Name(str), ⟨block⟩*)`, da das gesamte Konzept mit den **Blöcken** erst im **RETI-Pass** in Unterkapitel 3.3.1.6 aufgelöst wird.

[25] Es wurde hiebei zur **besseren Lesbarkeit** darauf verzichtet, wie üblich alle nicht mehr vorhandenen **PicoC-Knoten** in rot nochmal hinzuschreiben, um deutlich zu machen, dass diese **nicht mehr da** sind.

reg	::=	$ACC() \mid IN1() \mid IN2() \mid PC() \mid SP() \mid BAF()$	*L_RETI*
		$\mid CS() \mid DS()$	
arg	::=	$Reg(\langle reg \rangle) \mid Num(\langle str \rangle)$	
rel	::=	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
		$\mid Always() \mid NOp()$	
op	::=	$Add() \mid Addi() \mid Sub() \mid Subi() \mid Mult() \mid Multi()$	
		$\mid Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi()$	
		$\mid Or() \mid Ori() \mid And() \mid Andi()$	
		$\mid Load() \mid Loadin() \mid Loadi() \mid Store() \mid Storein() \mid Move()$	
instr	::=	$Instr(\langle op \rangle, \langle arg \rangle+) \mid Jump(\langle rel \rangle, Num(\langle str \rangle)) \mid Int(Num(\langle str \rangle))$	
		$\mid RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)$	
		$\mid SingleLineComment(\langle str \rangle, \langle str \rangle)$	
		$\mid Instr(Loadi(), [Reg(Acc()), GoTo(Name(\langle str \rangle))])$	
		$\mid Jump(Eq(), GoTo(Name(\langle str \rangle)))$	
instr	::=	$Exp(GoTo(\langle str \rangle))$	*L_PicoC*
block	::=	$Block(Name(\langle str \rangle), \langle instr \rangle*)$	
file	::=	$File(Name(\langle str \rangle), \langle block \rangle*)$	

Grammatik 3.3.4: *Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF*

3.3.1.4.2 Codebeispiel

In Code 3.11 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 3.10 mithilfe des **RETI-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 3.6 weitergeführt. Die **Anweisungen**, die durch entsprechende **PicoC-Knoten** im **Abstrakten Syntaxbaum** der Sprache L_{PicoC_ANF} repräsentiert waren, werden nun durch **semantisch** entsprechende **RETI-Knoten** ersetzt.

```
File
  Name './example_faculty_it.reti_blocks',
  [
    Block
      Name 'faculty.6',
      [
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN BAF ACC -3;
        ADDI SP 1;
        # // While(Num('1'), [])
        # Exp(GoTo(Name('condition_check.5')))
        Exp(GoTo(Name('condition_check.5')))
      ],
    Block
      Name 'condition_check.5',
      [
        # // IfElse(Num('1'), [], [])
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
```

```
27        STOREIN SP ACC 1;
28        # IfElse(Stack(Num('1')), [], [])
29        LOADIN SP ACC 1;
30        ADDI SP 1;
31        JUMP== GoTo(Name('while_after.1'));
32        Exp(GoTo(Name('while_branch.4')))
33      ],
34    Block
35      Name 'while_branch.4',
36      [
37        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
38        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39        # Exp(Stackframe(Num('0')))
40        SUBI SP 1;
41        LOADIN BAF ACC -2;
42        STOREIN SP ACC 1;
43        # Exp(Num('1'))
44        SUBI SP 1;
45        LOADI ACC 1;
46        STOREIN SP ACC 1;
47        LOADIN SP ACC 2;
48        LOADIN SP IN2 1;
49        SUB ACC IN2;
50        JUMP== 3;
51        LOADI ACC 0;
52        JUMP 2;
53        LOADI ACC 1;
54        STOREIN SP ACC 2;
55        ADDI SP 1;
56        # IfElse(Stack(Num('1')), [], [])
57        LOADIN SP ACC 1;
58        ADDI SP 1;
59        JUMP== GoTo(Name('if_else_after.2'));
60        Exp(GoTo(Name('if.3')))
61      ],
62    Block
63      Name 'if.3',
64      [
65        # // Return(Name('res'))
66        # Exp(Stackframe(Num('1')))
67        SUBI SP 1;
68        LOADIN BAF ACC -3;
69        STOREIN SP ACC 1;
70        # Return(Stack(Num('1')))
71        LOADIN SP ACC 1;
72        ADDI SP 1;
73        LOADIN BAF PC -1;
74      ],
75    Block
76      Name 'if_else_after.2',
77      [
78        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
79        # Exp(Stackframe(Num('0')))
80        SUBI SP 1;
81        LOADIN BAF ACC -2;
82        STOREIN SP ACC 1;
83        # Exp(Stackframe(Num('1')))
```

```
84        SUBI SP 1;
85        LOADIN BAF ACC -3;
86        STOREIN SP ACC 1;
87        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88        LOADIN SP ACC 2;
89        LOADIN SP IN2 1;
90        MULT ACC IN2;
91        STOREIN SP ACC 2;
92        ADDI SP 1;
93        # Assign(Stackframe(Num('1')), Stack(Num('1')))
94        LOADIN SP ACC 1;
95        STOREIN BAF ACC -3;
96        ADDI SP 1;
97        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98        # Exp(Stackframe(Num('0')))
99        SUBI SP 1;
100       LOADIN BAF ACC -2;
101       STOREIN SP ACC 1;
102       # Exp(Num('1'))
103       SUBI SP 1;
104       LOADI ACC 1;
105       STOREIN SP ACC 1;
106       # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107       LOADIN SP ACC 2;
108       LOADIN SP IN2 1;
109       SUB ACC IN2;
110       STOREIN SP ACC 2;
111       ADDI SP 1;
112       # Assign(Stackframe(Num('0')), Stack(Num('1')))
113       LOADIN SP ACC 1;
114       STOREIN BAF ACC -2;
115       ADDI SP 1;
116       # Exp(GoTo(Name('condition_check.5')))
117       Exp(GoTo(Name('condition_check.5')))
118     ],
119   Block
120     Name 'while_after.1',
121     [
122       # Return(Empty())
123       LOADIN BAF PC -1;
124     ],
125   Block
126     Name 'main.0',
127     [
128       # StackMalloc(Num('2'))
129       SUBI SP 2;
130       # Exp(Num('4'))
131       SUBI SP 1;
132       LOADI ACC 4;
133       STOREIN SP ACC 1;
134       # NewStackframe(Name('faculty.6'), GoTo(Name('addr@next_instr')))
135       MOVE BAF ACC;
136       ADDI SP 3;
137       MOVE SP BAF;
138       SUBI SP 4;
139       STOREIN BAF ACC 0;
140       LOADI ACC GoTo(Name('addr@next_instr'));
```

```
141         ADD ACC CS;
142         STOREIN BAF ACC -1;
143         # Exp(GoTo(Name('faculty.6')))
144         Exp(GoTo(Name('faculty.6')))
145         # RemoveStackframe()
146         MOVE BAF IN1;
147         LOADIN IN1 BAF 0;
148         MOVE IN1 SP;
149         # Exp(ACC)
150         SUBI SP 1;
151         STOREIN SP ACC 1;
152         LOADIN SP ACC 1;
153         ADDI SP 1;
154         CALL PRINT ACC;
155         # Return(Empty())
156         LOADIN BAF PC -1;
157       ]
158   ]
```

Code 3.11: *RETI-Blocks Pass für Codebespiel.*

Anmerkung

Wenn der **Abstrakte Syntaxbaum** ausgegeben wird, ist die Darstellung nicht auschließlich in **Abstrakter Syntax**, da die **RETI-Knoten** aus bereits im Unterkapitel 3.2.5.6 vermitteltem Grund in **Konkreter Syntax** ausgegeben werden.

3.3.1.5 RETI-Patch Pass

Die Aufgabe des **RETI-Patch Pass** ist das **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes** der Sprache L_{RETI_Blocks} durch:

- das **Einfügen** eines `start.<number>`-Blocks, welcher ein `GoTo(Name('main'))` zur `main`-Funktion enthält, wenn in manchen Fällen die `main`-Funktion **nicht** die erste Funktion ist und daher am Anfang zur `main`-Funktion gesprungen werden muss.
- das **Entfernen** von `GoTo()`'s, deren Sprung nur **eine** Adresse weiterspringen würde.
- das **Voranstellen** von **RETI-Knoten** vor jede **Division** `Instr(Div(), args)`, die prüfen, ob durch `0` geteilt wird. Ist es der Fall, dass durch 0 geteilt wird, dann wird in das `ACC`-Register der Fehlercode 1 geschrieben, der für die Fehlerart `DivisionByZero` steht und die Ausführung des Programmes wird **beendet**. Andernfalls **läuft** das Programm **weiter**.
- das Überprüfen darauf, ob bestimmte **Immediates** `Im(str)` in Befehlen, wie z.B. `Jump(rel, Im(str))`, `Instr(Loadin(), [reg1, reg2, Im(str)])`, `Instr(Loadi(), [reg, Im(str)])` usw. **kleiner** als -2^{21} oder **größer** als $2^{21} - 1$ sind. Im diesem Fall muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **Bitweise ODER** aus Zahlenwerten berechnet werden, die sich im Zahlenbereich zwischen -2^{21} und $2^{21} - 1$ befinden. Im Fall dessen, dass der **Immediate** allerdings **kleiner** als $-(2^{31})$ oder **größer** als $2^{31} - 1$ ist, wird eine Fehlermeldung `TooLargeLiteral` ausgegeben.[26]

3.3.1.5.1 Abstrakte Grammatik

[26] Einiges in diesem **Pass** fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

Die **Abstrakte Grammatik** 3.3.5 der Sprache L_{RETI_Patch} ist im Vergleich zur **Abstrakten Grammatik** 3.3.4 der Sprache L_{RETI_Blocks} **unverändert**. Es sind **keine neuen Knoten** hinzugekommen und **keine** Knoten wurden **abgeändert** oder völlig **entfernt**.

reg	$::=$	$ACC() \mid IN1() \mid IN2() \mid PC() \mid SP() \mid BAF()$	L_RETI
	$\mid$	$CS() \mid DS()$	
arg	$::=$	$Reg(\langle reg \rangle) \mid Num(\langle str \rangle)$	
rel	$::=$	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
	$\mid$	$Always() \mid NOp()$	
op	$::=$	$Add() \mid Addi() \mid Sub() \mid Subi() \mid Mult() \mid Multi()$	
	$\mid$	$Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi()$	
	$\mid$	$Or() \mid Ori() \mid And() \mid Andi()$	
	$\mid$	$Load() \mid Loadin() \mid Loadi() \mid Store() \mid Storein() \mid Move()$	
$instr$	$::=$	$Instr(\langle op \rangle, \langle arg \rangle+) \mid Jump(\langle rel \rangle, Num(\langle str \rangle)) \mid Int(Num(\langle str \rangle))$	
	$\mid$	$RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)$	
	$\mid$	$SingleLineComment(\langle str \rangle, \langle str \rangle)$	
	$\mid$	$Instr(Loadi(), [Reg(Acc()), GoTo(Name(\langle str \rangle))])$	
	$\mid$	$Jump(Eq(), GoTo(Name(\langle str \rangle)))$	
$instr$	$::=$	$Exp(GoTo(\langle str \rangle))$	L_PicoC
$block$	$::=$	$Block(Name(\langle str \rangle), \langle instr \rangle*)$	
$file$	$::=$	$File(Name(\langle str \rangle), \langle block \rangle*)$	

Grammatik 3.3.5: *Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF*

3.3.1.5.2 Codebeispiel

In Code 3.12 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 3.11 mithilfe des **RETI-Patch Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 3.6 weitergeführt. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>`-Block[27] eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-Knoten **entfernt**, die nur einem Sprung um **eine Adresse** weiter entsprochen hätten[28].

```
File
  Name './example_faculty_it.reti_patch',
  [
    Block
      Name 'start.7',
      [
        # // Exp(GoTo(Name('main.0')))
        Exp(GoTo(Name('main.0')))
      ],
    Block
      Name 'faculty.6',
      [
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('1')), Stack(Num('1')))
```

[27] Dieser `start.<nummer>`-Block wurde im Code 3.12 markiert.
[28] Diese **entfernten** `GoTo(Name(str))`-Knoten wurden in Code 3.12 markiert.

```
19        LOADIN SP ACC 1;
20        STOREIN BAF ACC -3;
21        ADDI SP 1;
22        # // While(Num('1'), [])
23        # Exp(GoTo(Name('condition_check.5')))
24        # // not included Exp(GoTo(Name('condition_check.5')))
25      ],
26    Block
27      Name 'condition_check.5',
28      [
29        # // IfElse(Num('1'), [], [])
30        # Exp(Num('1'))
31        SUBI SP 1;
32        LOADI ACC 1;
33        STOREIN SP ACC 1;
34        # IfElse(Stack(Num('1')), [], [])
35        LOADIN SP ACC 1;
36        ADDI SP 1;
37        JUMP== GoTo(Name('while_after.1'));
38        # // not included Exp(GoTo(Name('while_branch.4')))
39      ],
40    Block
41      Name 'while_branch.4',
42      [
43        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
44        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
45        # Exp(Stackframe(Num('0')))
46        SUBI SP 1;
47        LOADIN BAF ACC -2;
48        STOREIN SP ACC 1;
49        # Exp(Num('1'))
50        SUBI SP 1;
51        LOADI ACC 1;
52        STOREIN SP ACC 1;
53        LOADIN SP ACC 2;
54        LOADIN SP IN2 1;
55        SUB ACC IN2;
56        JUMP== 3;
57        LOADI ACC 0;
58        JUMP 2;
59        LOADI ACC 1;
60        STOREIN SP ACC 2;
61        ADDI SP 1;
62        # IfElse(Stack(Num('1')), [], [])
63        LOADIN SP ACC 1;
64        ADDI SP 1;
65        JUMP== GoTo(Name('if_else_after.2'));
66        # // not included Exp(GoTo(Name('if.3')))
67      ],
68    Block
69      Name 'if.3',
70      [
71        # // Return(Name('res'))
72        # Exp(Stackframe(Num('1')))
73        SUBI SP 1;
74        LOADIN BAF ACC -3;
75        STOREIN SP ACC 1;
```

```
76        # Return(Stack(Num('1')))
77        LOADIN SP ACC 1;
78        ADDI SP 1;
79        LOADIN BAF PC -1;
80      ],
81    Block
82      Name 'if_else_after.2',
83      [
84        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85        # Exp(Stackframe(Num('0')))
86        SUBI SP 1;
87        LOADIN BAF ACC -2;
88        STOREIN SP ACC 1;
89        # Exp(Stackframe(Num('1')))
90        SUBI SP 1;
91        LOADIN BAF ACC -3;
92        STOREIN SP ACC 1;
93        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94        LOADIN SP ACC 2;
95        LOADIN SP IN2 1;
96        MULT ACC IN2;
97        STOREIN SP ACC 2;
98        ADDI SP 1;
99        # Assign(Stackframe(Num('1')), Stack(Num('1')))
100       LOADIN SP ACC 1;
101       STOREIN BAF ACC -3;
102       ADDI SP 1;
103       # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104       # Exp(Stackframe(Num('0')))
105       SUBI SP 1;
106       LOADIN BAF ACC -2;
107       STOREIN SP ACC 1;
108       # Exp(Num('1'))
109       SUBI SP 1;
110       LOADI ACC 1;
111       STOREIN SP ACC 1;
112       # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113       LOADIN SP ACC 2;
114       LOADIN SP IN2 1;
115       SUB ACC IN2;
116       STOREIN SP ACC 2;
117       ADDI SP 1;
118       # Assign(Stackframe(Num('0')), Stack(Num('1')))
119       LOADIN SP ACC 1;
120       STOREIN BAF ACC -2;
121       ADDI SP 1;
122       # Exp(GoTo(Name('condition_check.5')))
123       Exp(GoTo(Name('condition_check.5')))
124     ],
125   Block
126     Name 'while_after.1',
127     [
128       # Return(Empty())
129       LOADIN BAF PC -1;
130     ],
131   Block
132     Name 'main.0',
```

```
133       [
134         # StackMalloc(Num('2'))
135         SUBI SP 2;
136         # Exp(Num('4'))
137         SUBI SP 1;
138         LOADI ACC 4;
139         STOREIN SP ACC 1;
140         # NewStackframe(Name('faculty.6'), GoTo(Name('addr@next_instr')))
141         MOVE BAF ACC;
142         ADDI SP 3;
143         MOVE SP BAF;
144         SUBI SP 4;
145         STOREIN BAF ACC 0;
146         LOADI ACC GoTo(Name('addr@next_instr'));
147         ADD ACC CS;
148         STOREIN BAF ACC -1;
149         # Exp(GoTo(Name('faculty.6')))
150         Exp(GoTo(Name('faculty.6')))
151         # RemoveStackframe()
152         MOVE BAF IN1;
153         LOADIN IN1 BAF 0;
154         MOVE IN1 SP;
155         # Exp(ACC)
156         SUBI SP 1;
157         STOREIN SP ACC 1;
158         LOADIN SP ACC 1;
159         ADDI SP 1;
160         CALL PRINT ACC;
161         # Return(Empty())
162         LOADIN BAF PC -1;
163       ]
164   ]
```

Code 3.12: *RETI-Patch Pass für Codebespiel.*

3.3.1.6 RETI Pass

Die Aufgabe des **RETI-Patch Pass** ist es die letzten verbliebenen **PicoC-Knoten** im **Abstrakten Syntaxbaum** zu entfernen bzw. zu ersetzen. Dementsprechend werden die **Blöcke** `Block(Name(str), instr*)` entfernt und die **Knoten** in diesen Blöcken werden genauso **zusammengefügt**, wie die Blöcke **angeordnet** waren.

Des Weiteren werden die `GoTo(Name(str))`-Knoten in den den Knoten `Instr(Loadi(), [reg, GoTo(Name(str))])`, `Jump(Eq(), GoTo(Name(str)))` und `Exp(GoTo(Name(str)))` durch einen **Immediate** `Im(str(distance_or_address))` mit passender **Adresse** oder **Distanz** oder einen **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))` ersetzt. Die **Distanz-** und **Adressberechnung** wird in Unterkapitel 3.3.6.3 genauer erklärt.

3.3.1.6.1 Konkrete und Abstrakte Grammatik

Die **Abstrakte Grammatik** 3.3.6 der Sprache L_{RETI} hat im Vergleich zur **Abstrakten Grammatik** 3.3.5 der Sprache L_{RETI_Patch} nur noch ausschließlich **RETI-Knoten**, dementsprechend gibt es die Knoten `Instr(Loadi(), [Reg(Acc()), GoTo(Name(str))])`, `Jump(Eq(), GoTo(Name(str)))` nicht mehr. Des Weiteren gibt es **keine Blöcke** `Block(Name(str), instr*)` mehr, alle **RETI-Knoten** stehen nun in einem `Program(Name(str), instr*)`-Knoten.

Ausgegeben wird der finale Abstrakte Syntaxbaum in **Konkreter Syntax**, die durch die Konkreten Grammatiken 3.3.7 und 3.3.8 für jeweils die **Lexikalische** und **Syntaktische Analyse** $G_{RETI\_Lex} \uplus G_{RETI\_Parse}$ beschrieben wird.

reg	::=	*ACC*() | *IN1*() | *IN2*() | *PC*() | *SP*() | *BAF*()	*L_RETI*
		| *CS*() | *DS*()	
arg	::=	*Reg*(⟨*reg*⟩) | *Num*(⟨*str*⟩)	
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
		| *Always*() | *NOp*()	
op	::=	*Add*() | *Addi*() | *Sub*() | *Subi*() | *Mult*() | *Multi*()	
		| *Div*() | *Divi*() | *Mod*() | *Modi*() | *Oplus*() | *Oplusi*()	
		| *Or*() | *Ori*() | *And*() | *Andi*()	
		| *Load*() | *Loadin*() | *Loadi*() | *Store*() | *Storein*() | *Move*()	
instr	::=	*Instr*(⟨*op*⟩, ⟨*arg*⟩+) | *Jump*(⟨*rel*⟩, *Num*(⟨*str*⟩)) | *Int*(*Num*(⟨*str*⟩))	
		| *RTI*() | *Call*(*Name*('*print*'), ⟨*reg*⟩) | *Call*(*Name*('*input*'), ⟨*reg*⟩)	
		| *SingleLineComment*(⟨*str*⟩, ⟨*str*⟩)	
		| *Instr*(*Loadi*(), [*Reg*(*Acc*()), *GoTo*(*Name*(⟨*str*⟩))])	
		| *Jump*(*Eq*(), *GoTo*(*Name*(⟨*str*⟩)))	
program	::=	*Program*(*Name*(⟨*str*⟩), ⟨*instr*⟩∗)	
instr	::=	*Exp*(*GoTo*(⟨*str*⟩)) | *Exit*(*Num*(⟨*str*⟩))	*L_PicoC*
block	::=	*Block*(*Name*(⟨*str*⟩), ⟨*instr*⟩∗)	
file	::=	*File*(*Name*(⟨*str*⟩), ⟨*block*⟩∗)	

Grammatik 3.3.6: *Abstrakte Grammatik der Sprache L_{RETI} in ASF*

dig_no_0	::=	"1" | "2" | "3" | "4" | "5" | "6"	*L_Program*
		| "7" | "8" | "9"	
dig_with_0	::=	"0" | *dig_no_0*	
num	::=	"0" | *dig_no_0* *dig_with_0* ∗ | "−"*dig_no_0*∗	
letter	::=	"*a*"..."*Z*"	
name	::=	*letter*(*letter* | *dig_with_0* | _)∗	
reg	::=	"*ACC*" | "*IN1*" | "*IN2*" | "*PC*" | "*SP*"	
		| "*BAF*" | "*CS*" | "*DS*"	
arg	::=	*reg* | *num*	
rel	::=	"==" | "!=" | "<" | "<=" | ">"	
		| ">=" | "_*NOP*"	

Grammatik 3.3.7: *Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF*

instr	::=	*"ADD" reg arg* | *"ADDI" reg num* | *"SUB" reg arg*	*L_Program*
		| *"SUBI" reg num* | *"MULT" reg arg* | *"MULTI" reg num*	
		| *"DIV" reg arg* | *"DIVI" reg num* | *"MOD" reg arg*	
		| *"MODI" reg num* | *"OPLUS" reg arg* | *"OPLUSI" reg num*	
		| *"OR" reg arg* | *"ORI" reg num*	
		| *"AND" reg arg* | *"ANDI" reg num*	
		| *"LOAD" reg num* | *"LOADIN" arg arg num*	
		| *"LOADI" reg num*	
		| *"STORE" reg num* | *"STOREIN" arg argnum*	
		| *"MOVE" reg reg*	
		| *"JUMP"rel num* | *INT num* | *RTI*	
		| *"CALL" "INPUT" reg* | *"CALL" "PRINT" reg*	
program	::=	*(instr";")**	

Grammatik 3.3.8: *Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF*

3.3.1.6.2 Codebeispiel

In Code 3.13 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 3.12 mithilfe des **PicoC-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 3.6 weitergeführt. Es gibt **keine Blöcke** mehr und die **Knoten** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Das Programm ist komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkreten Syntax** ausgegeben werden.

Die letzten **PicoC-Knoten** und **RETI-Befehle**, die **nicht** ausschließlich aus **RETI-Knoten** bestanden `Exp(GoTo(Name('main.0')))`, `JUMP== GoTo(Name('while_after.1'))`, `JUMP== GoTo(Name('if_else_after.2'))`, `Exp(GoTo(Name('condition_check.5')))`, `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('faculty.6')))` wurden durch **RETI-Befehle** ersetzt, welche in Code 3.13 markiert wurden.

Der `Program(Name(str), instr)`-Knoten, der alle **RETI-Knoten** beinhaltet, gibt alleinig die **RETI-Knoten**, die er beinhaltet aus und fügt ansonsten **nichts** zur Ausgabe hinzu, wie z.B. den **Bezeichner des Programms** oder **Einrückung**. Hierdurch erzeugt der **Abstrakte Syntaxbaum**, wenn er in **Konkreter Syntax** in eine Datei ausgegeben wird direkt **RETI-Code** in **menschenlesbarer Repräsentation**.

```
1  # // Exp(GoTo(Name('main.0')))
2  JUMP 67;
3  # // Assign(Name('res'), Num('1'))
4  # Exp(Num('1'))
5  SUBI SP 1;
6  LOADI ACC 1;
7  STOREIN SP ACC 1;
8  # Assign(Stackframe(Num('1')), Stack(Num('1')))
9  LOADIN SP ACC 1;
10 STOREIN BAF ACC -3;
11 ADDI SP 1;
12 # // While(Num('1'), [])
13 # Exp(GoTo(Name('condition_check.5')))
14 # // not included Exp(GoTo(Name('condition_check.5')))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
```

```
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4')))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0')))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;
43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3;
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;
73 # Assign(Stackframe(Num('1')), Stack(Num('1')))
74 LOADIN SP ACC 1;
```

```
STOREIN BAF ACC -3;
ADDI SP 1;
# // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
# Exp(Stackframe(Num('0')))
SUBI SP 1;
LOADIN BAF ACC -2;
STOREIN SP ACC 1;
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
LOADIN SP ACC 2;
LOADIN SP IN2 1;
SUB ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
# Assign(Stackframe(Num('0')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN BAF ACC -2;
ADDI SP 1;
# Exp(GoTo(Name('condition_check.5')))
JUMP -58;
# Return(Empty())
LOADIN BAF PC -1;
# StackMalloc(Num('2'))
SUBI SP 2;
# Exp(Num('4'))
SUBI SP 1;
LOADI ACC 4;
STOREIN SP ACC 1;
# NewStackframe(Name('faculty.6'), GoTo(Name('addr@next_instr')))
MOVE BAF ACC;
ADDI SP 3;
MOVE SP BAF;
SUBI SP 4;
STOREIN BAF ACC 0;
LOADI ACC 80;
ADD ACC CS;
STOREIN BAF ACC -1;
# Exp(GoTo(Name('faculty.6')))
JUMP -78;
# RemoveStackframe()
MOVE BAF IN1;
LOADIN IN1 BAF 0;
MOVE IN1 SP;
# Exp(ACC)
SUBI SP 1;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
ADDI SP 1;
CALL PRINT ACC;
# Return(Empty())
LOADIN BAF PC -1;
```

Code 3.13: *RETI Pass für Codebespiel.*

3.3.2 Umsetzung von Zeigern

Die Umsetzung von **Zeigern** ist in diesem Unterkapitel schnell erklärt, auch Dank eines kleinen **Taschenspielertricks**[29]. Hierbei sind nur die **Operationen** für **Referenzierung** und **Dereferenzierung** in den Unterkapiteln 3.3.2.1 und 3.3.2.2 zu erläutern. **Referenzierung** kann dazu genutzt werden einen **Zeiger** zu **initialisieren** und **Dereferenzierung** kann dazu genutzt werden, um auf diesen später zuzugreifen.

3.3.2.1 Referenzierung

Die **Referenzierung** (z.B. `&var`) ist eine **Operation** bei der ein **Zeiger** auf eine **Location** (Definition 2.49) in Form der **Anfangsadresse** dieser Location als Ergebnis zurückgegeben wird. Die Umsetzung der **Referenzierung** wird im Folgenden anhand des Beispiels in Code 3.14 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4 }
```

Code 3.14: *PicoC-Code für Zeigerreferenzierung.*

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 3.15 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Zeiger `*pntr`.

```
1 File
2   Name './example_pntr_ref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
          ↪  Ref(Name('var')))
11      ]
12  ]
```

Code 3.15: *Abstrakter Syntaxbaum für Zeigerreferenzierung.*

Bevor man einem **Zeiger** eine **Adresse** (z.B. `&var`) zuweisen kann, muss dieser erstmal **definiert** sein. Dafür braucht es einen Eintrag in der **Symboltabelle** in Code 3.16.

Anmerkung

Die **Anzahl Speicherzellen**[a], die ein Zeiger[b] `datatype *pntr` belegt ist dabei immer[c]: $size(type(\texttt{pntr})) = 1$ *Speicherzelle.*[def]

[a] Die im `size`-Attribut der **Symboltabelle** eingetragen ist.
[b] Z.B. ein Zeiger auf ein Feld von Integern: `int (*pntr)[3]`.
[c] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas

[29] Später mehr dazu.

unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.

[d] Eine **Speicherzelle** ist in der **RETI-Architektur**, wie in Unterkapitel 1.1 erklärt 4 *Byte* breit.

[e] Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein **Datentyp** belegt.

[f] Die Funktion *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('var@main')
        value or address:       Num('0')
        position:               Pos(Num('2'), Num('6'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), IntType('int'))
        name:                   Name('pntr@main')
        value or address:       Num('1')
        position:               Pos(Num('3'), Num('7'))
        size:                   Num('1')
      }
  ]
```

Code 3.16: *Symboltabelle für Zeigerreferenzierung.*

Im **PicoC-ANF Pass** in Code 3.17 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')),Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp)` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var'),Name('attr')),Num('1'))` ist, sind noch **weitere Anweisungen** zwischen den Zeilen `11` und `12` nötig. Diese weiteren Anweisungen würden sich bei z.B. `Subscr(Attr(Name('var'),Name('attr')),Num('1'))` um das Übersetzen von `Subscr(exp)` und `Attr(exp,name)` nach dem Schema in Unterkapitel 3.3.5.2 kümmern.[30]

```
File
  Name './example_pntr_ref.picoc_mon',
```

[30] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
3  [
4    Block
5      Name 'main.0',
6      [
7        // Assign(Name('var'), Num('42'))
8        Exp(Num('42'))
9        Assign(Global(Num('0')), Stack(Num('1')))
10       // Assign(Name('pntr'), Ref(Name('var')))
11       Ref(Global(Num('0')))
12       Assign(Global(Num('1')), Stack(Num('1')))
13       Return(Empty())
14     ]
15 ]
```

Code 3.17: *PicoC-ANF Pass für Zeigerreferenzierung.*

Im **RETI-Blocks Pass** in Code 3.18 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')),Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_pntr_ref.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('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('pntr'), Ref(Name('var')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 3.18: *RETI-Blocks Pass für Zeigerreferenzierung.*

3.3.2.2 Dereferenzierung

Die **Dereferenzierung** (z.B. `*var`) ist eine **Operation** bei der einem **Zeiger** zur Location (Definition 2.49) hin **gefolgt** wird, auf welche dieser zeigt und das Ergebnis z.B. der **Inhalt** der ersten Speicherzelle der referenzierten Location ist. Die Umsetzung von **Dereferenzierung** wird im Folgenden anhand des Beispiels in Code 3.19 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4   *pntr;
5 }
```

Code 3.19: *PicoC-Code für Zeigerdereferenzierung.*

Der Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 3.20 eine **Dereferenzierung** `*var`. Es gibt hierbei 3 Fälle. Bei der Anwendung von **Zeigerarithmetik**, wie z.B. `*(var + 2 - 1)` übersetzt sich diese zu `Deref(Name('var'), BinOp(Num('2'), Sub(), Num('1')))` und bei z.B. `*(var - 2 - 1)` zu `Deref(Name('var'), UnOp(Minus(), BinOp(Num('2'), Sub(), Num('1'))))`. Bei einer normalen **Dereferenzierung**, wie z.B. `*var`, übersetzt sich diese zu `Deref(Name('var'), Num('0'))`[31].

```
1 File
2   Name './example_pntr_deref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
          ↪  Ref(Name('var')))
11        Exp(Deref(Name('pntr'), Num('0')))
12      ]
13  ]
```

Code 3.20: *Abstrakter Syntaxbaum für Zeigerdereferenzierung.*

Im **PicoC-Shrink Pass** in Code 3.21 wird ein **Trick** angewandet, bei dem jeder Knoten `Deref(exp1, exp2)` einfach durch den Knoten `Subscr(exp1, exp2)` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzierungsoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Feldindex** (z.B. `var[1]`) verhält, wie es bereits im Unterkapitel 1.3 erläutert wurde. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die Übersetzung der **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Feldindex** (z.B. `var[1]`) übernehmen lassen. Das Vorgehen bei der Umsetzung eines **Zugriffs auf einen Feldindex** (z.B. `*(var + 1)`) wird in Unterkapitel 3.3.3.2 erläutert.[32]

[31] Das `Num('0')` steht dafür, dass dem **Zeiger gefolgt** wird, aber danach **nicht** noch mit einem **Versatz** von der **Größe des Unterdatentyps** (Definition 3.9) auf eine nebenliegende Location zugegriffen wird.

[32] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
File
  Name './example_pntr_deref.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Subscr(Name('pntr'), Num('0')))
      ]
  ]
```

Code 3.21: *PicoC-Shrink Pass für Zeigerdereferenzierung.*

3.3.3 Umsetzung von Feldern

Bei Feldern ist in diesem Unterkapitel die Umsetzung der **Innitialisierung eines Feldes** 3.3.3.1, des **Zugriffs auf einen Feldindex** 3.3.3.2 und der **Zuweisung an einen Feldindex** 3.3.3.3 zu klären.

3.3.3.1 Initialisierung eines Feldes

Die Umsetzung der **Initialisierung** eines **Feldes** (z.B. `int ar[2][1] = {{3+1}, {5}}`) wird im Folgenden anhand des Beispiels in Code 3.22 erklärt.

```
void fun() {
  int ar[2][2] = {{3, 4}, {5, 6}};
}

void main() {
  int ar[2][1] = {{3+1}, {5}};
}
```

Code 3.22: *PicoC-Code für die Initialisierung eines Feldes.*

Die **Initialisierung** eines **Feldes** `intar[2][1]={{3+1},{5}}` wird im **Abstrakten Syntaxbaum** in Code 3.23 mithilfe der Knoten `Assign(Alloc(Writeable(),ArrayDecl([Num('2'),Num('1')],IntType('int')),Name('ar')),Array([Array([BinOp(Num('3'),Add('+'),Num('1'))]),Array([Num('5')])]))` dargestellt.

```
File
  Name './example_array_init.ast',
  [
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
```

```
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
          ↪  Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
10      ],
11    FunDef
12      VoidType 'void',
13      Name 'main',
14      [],
15      [
16        Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
          ↪  Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('5')])]))
17      ]
18  ]
```

Code 3.23: *Abstrakter Syntaxbaum für die Initialisierung eines Feldes.*

Bei der **Initialisierung** eines **Feldes** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[33]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 3.24 dargestellt ist.

Auf dem **Stackframe** wird ein Feld verglichen zur Wachstumrichtung des Stacks **rückwärts** in den Stackframe geschrieben und die **relative Adresse des ersten Elements** als Adresse des Feldes in der **Symboltabelle** in Code 3.24 genommen. Dies ist in Tabelle 3.10 für das Beispiel aus Code 3.22 dargstellt. Es wird hier so getann, als würde die **Funktion** `fun` ebenfalls **aufgerufen** werden. Obwohl der **Stack** zwar verglichen zu den **Globalen Statischen Daten** in die entgegengesetzte Richtung wächst, haben Felder in den **Globalen Statischen Daten** und in einem **Stackframe** auf diese Weise die gleiche Ausrichtung. Das macht den **Zugriff auf einen Feldindex** in Unterkapitel 3.3.3.2 deutlich unkomplizierter. Auf diese Weise muss beim **Zugriff auf einen Feldindex** nicht zwischen **Stackframe** und **Globalen Statischen Daten** unterschieden werden.

Relativ-adresse	Wert	Register
0	4	CS
1	5	
...	...	
3	3	
2	4	
1	5	
0	6	
...	...	
...	...	BAF

Tabelle 3.10: *Datensegment nach der Initialisierung beider Felder.*

Anmerkung

Die **Anzahl Speicherzellen**, die ein Feld[a] `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_n]$ belegt[b], berechnet sich aus der **Mächtigkeit** der einzelnen **Dimensionen** des Feldes und der **Anzahl Speicherzellen**, die der

[33] Das widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.

Datentyp, den alle **Feldelemente** haben belegt: $size(type(\texttt{ar})) = \left(\prod_{j=1}^{n} \texttt{dim}_\texttt{j}\right) \cdot size(\texttt{datatype})$.[cd]

[a]Die im `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.
[b]Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von $\texttt{L}_\texttt{C}$ übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[c]Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[d]Die **Funktion** *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun'), [])
        name:                   Name('fun')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('2')], IntType('int'))
        name:                   Name('ar@fun')
        value or address:       Num('3')
        position:               Pos(Num('2'), Num('6'))
        size:                   Num('4')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('5'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('1')], IntType('int'))
        name:                   Name('ar@main')
        value or address:       Num('0')
        position:               Pos(Num('6'), Num('6'))
        size:                   Num('2')
      }
  ]
```

Code 3.24: *Symboltabelle für die Initialisierung eines Feldes.*

Im **PiocC-ANF Pass** in Code 3.25 werden zuerst die Knoten für die **Logischen Ausdrücke** in den **Blättern** des Teilbaumes, dessen Wurzel der **Feld-Initialisierer**-Knoten `Array([Array([BinOp(Num('3'), Add('+'),`

`Num('1'))]), Array([Num('5')])])` ist ausgewertet. Die Auswertung geschieht hierbei nach dem Prinzip der **Tiefensuche**, von **links-nach-rechts**. Bei dieser Auswertung werden diese Knoten für die **Logischen Ausdrücke** durch Knoten erstetzt, welche das Ergebnis dieser Ausdrücke auf den **Stack** schreiben[34].

Im finalen Schritt muss zwischen den **Globalen Statischen Daten** der `main`-Funktion und dem **Stackframe** der Funktion `fun` unterschieden werden. Die auf dem **Stack** ausgewerteten **Logischen Ausdrücke** werden mittels der Knoten `Assign(Global(Num('0')), Stack(Num('2')))` (für **Globale Statische Daten**) bzw. der Knoten `Assign(Stackframe(Num('3')), Stack(Num('5')))` (für **Stackframe**) zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.[35]

Zur Veranschaulichung ist in Tabelle 3.11 ein **Ausschnitt des Datensegments** nach der Initialisierung des Feldes der Funktion `main`-Funktion dargestellt. Die auf den **Stack** ausgewerteten **Logischen Ausdrücke** sind in grauer Farbe markiert. Die **Kopien** dieser ausgewerteten Logischen Ausdrücke in den **Globalen Statischen Daten**, welche die einzelnen Elemente des Feldes darstellen sind in roter Farbe markiert. In Tabelle 3.12 ist das gleiche, allerdings für die Funktion `fun` und den **Stackframe** der Funktion `fun` dargestellt.

Relativ-adresse	Wert	Register
0	4	CS
1	5	
...	...	
1	5	
2	4	SP
...	...	

Tabelle 3.11: *Ausschnitt des Datensegments nach der Initialisierung des Feldes in der `main`-Funktion.*

Relativ-adresse	Wert	Register
...	...	
1	6	
2	5	
3	4	
4	3	SP
3	3	
2	4	
1	5	
0	6	
...	...	
...	...	BAF

Tabelle 3.12: *Ausschnitt des Datensegments nach der Initialisierung des Feldes in der Funktion `fun`.*

Der **Trick** ist hier, dass egal wieviele **Dimensionen** und was für einen **grundlegenden Datentyp**[36] das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man z.B. mit den **Knoten** `Assign(Global(Num('0')), Stack(Num('2')))` einfach so viele Speicherzellen rüberkopiert, wie das Feld **Speicherzellen** belegt.

[34] Da der **Zuweisungsoperator** `=` **rechtsassoziativ** ist und auch rein **logisch**, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

[35] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

[36] Z.B. ein **Verbund**, sodass es ein **„Feld von Verbunden“** ist.

In die Knoten `Global('0')` und `Stackframe('3')` wird hierbei die **Startadresse** des jeweiligen Feldes geschrieben. Daher müssen nach dem **PicoC-ANF Pass** nie mehr Variablen in der **Symboltabelle** nachgesehen werden und es ist möglich direkt abzulesen, ob diese in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

```
File
  Name './example_array_init.picoc_mon',
  [
    Block
      Name 'fun.1',
      [
        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        Exp(Num('3'))
        Exp(Num('4'))
        Exp(Num('5'))
        Exp(Num('6'))
        Assign(Stackframe(Num('3')), Stack(Num('4')))
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('5')])]))
        Exp(Num('3'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Exp(Num('5'))
        Assign(Global(Num('0')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 3.25: *PicoC-ANF Pass für die Initialisierung eines Feldes.*

Im **RETI-Blocks Pass** in Code 3.26 werden die **PicoC-Knoten** `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('5')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'fun.1',
      [
        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('4'))
```

```
13        SUBI SP 1;
14        LOADI ACC 4;
15        STOREIN SP ACC 1;
16        # Exp(Num('5'))
17        SUBI SP 1;
18        LOADI ACC 5;
19        STOREIN SP ACC 1;
20        # Exp(Num('6'))
21        SUBI SP 1;
22        LOADI ACC 6;
23        STOREIN SP ACC 1;
24        # Assign(Stackframe(Num('3')), Stack(Num('4')))
25        LOADIN SP ACC 1;
26        STOREIN BAF ACC -2;
27        LOADIN SP ACC 2;
28        STOREIN BAF ACC -3;
29        LOADIN SP ACC 3;
30        STOREIN BAF ACC -4;
31        LOADIN SP ACC 4;
32        STOREIN BAF ACC -5;
33        ADDI SP 4;
34        # Return(Empty())
35        LOADIN BAF PC -1;
36      ],
37    Block
38      Name 'main.0',
39      [
40        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('5')])]))
41        # Exp(Num('3'))
42        SUBI SP 1;
43        LOADI ACC 3;
44        STOREIN SP ACC 1;
45        # Exp(Num('1'))
46        SUBI SP 1;
47        LOADI ACC 1;
48        STOREIN SP ACC 1;
49        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
50        LOADIN SP ACC 2;
51        LOADIN SP IN2 1;
52        ADD ACC IN2;
53        STOREIN SP ACC 2;
54        ADDI SP 1;
55        # Exp(Num('5'))
56        SUBI SP 1;
57        LOADI ACC 5;
58        STOREIN SP ACC 1;
59        # Assign(Global(Num('0')), Stack(Num('2')))
60        LOADIN SP ACC 1;
61        STOREIN DS ACC 1;
62        LOADIN SP ACC 2;
63        STOREIN DS ACC 0;
64        ADDI SP 2;
65        # Return(Empty())
66        LOADIN BAF PC -1;
67      ]
68  ]
```

Code 3.26: *RETI-Blocks Pass für die Initialisierung eines Feldes.*

3.3.3.2 Zugriff auf einen Feldindex

Die Umsetzung des **Zugriffs auf einen Feldindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 3.27 erklärt.

```
1 void fun() {
2   int ar[1] = {42};
3   ar[0];
4 }
5
6 void main() {
7   int ar[3] = {1, 2, 3};
8   ar[1+1];
9 }
```

Code 3.27: *PicoC-Code für Zugriff auf einen Feldindex.*

Der **Zugriff auf einen Feldindex** `ar[0]` wird im **Abstrakten Syntaxbaum** in Code 3.28 mithilfe des Knotens `Subscr(Name('ar'), Num('0'))` dargestellt.

```
1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'fun',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
          ↪  Array([Num('42')]))
10        Exp(Subscr(Name('ar'), Num('0')))
11      ],
12    FunDef
13      VoidType 'void',
14      Name 'main',
15      [],
16      [
17        Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
          ↪  Array([Num('1'), Num('2'), Num('3')]))
18        Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19      ]
20  ]
```

Code 3.28: *Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.*

Im **PicoC-ANF Pass** in Code 3.29 wird zuerst das Schreiben der **Adresse** einer Variable `Name('ar')` des Knoten `Subscr(Name('ar'), Num('0'))` auf den **Stack** dargestellt. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Knoten `Ref(Global(Num('0')))` dargestellt und beim **Stackframe**

der Funktionm `fun` wird das durch die Knoten `Ref(Stackframe(Num('2')))` dargestellt. Diese Phase wird als **Anfangsteil** 3.3.5.1 bezeichnet.

Die nächste Phase wird als **Mittelteil** 3.3.5.2 bezeichnet. In dieser Phase wird die **Adresse**, ab der das **Feldelement** des Feldes auf das zugegriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Feldes, in dem dieses **Feldelement** liegt auf den **Stack** gelegt. Ein **Index** eines Feldelements auf das zugegriffen werden soll kann auch durch das Ergebnis eines **komplexeren Ausdrucks**, wie z.B. `ar[1 + var]` bestimmt sein, in dem auch **Variablen** vorkommen. Aus diesem Grund kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll auf den **Stack** gelegt werden, was z.B. im einfachsten Fall durch `Exp(Num('0'))` dargestellt wird. Danach kann die **Adresse des Index** berechnet werden, was durch die Knoten `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird.

In Tabelle 3.13 ist das ganze **veranschaulicht**. In dem Auschnitt liegt die **Startadresse** $2^{31} + 67$ des Felds `int ar[3] = {1, 2, 3}` der `main`-Funktion auf dem **Stack** und darüber wurde der **Wert des Index** `[1+1]` berechnet und auf dem Stack gespeichert (in rot markiert). Der Wert des Index wurde noch **nicht** auf die **Startadresse** des Felds **draufaddiert**.[37]

Absolutadresse	Wert	Register
...	...	
$2^{31} + 64$	1	SP
$2^{31} + 65$	2	
$2^{31} + 66$	$2^{31} + 67$	
$2^{31} + 67$	1	
$2^{31} + 68$	2	
$2^{31} + 69$	3	
...	...	
...	...	BAF

Tabelle 3.13: *Ausschnitt des Datensegments bei der Adressberechnung.*

Zur **Adressberechnung** ist es notwendig auf die **Dimensionen** (z.B. `[Num('3')]`) des Feldes, auf dessen **Feldelement** zugegriffen werden soll, zugreifen zu können. Daher ist der **Felddatentyp** (z.B. `ArrayDecl([Num('3')], IntType('int'))`) dem Knoten `Ref(exp, datatype)` als verstecktes Attribut `datatype` angehängt. Das versteckte Attribut wird zuvor, während des Kompiliervorgangs im **PiocC-ANF Pass** dem Knoten `Ref(exp, datatype)` angehängt.

Je nachdem, ob mehrere `Subscr(exp,exp)` eine Komposition bilden (z.B. `Subscr(Subscr(Name('var'), Num('1')), Num('1'))`) ist es notwendig mehrere **Adressberechnungsschritte für den Index** `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` einzuleiten. Es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und einen **Zugriff auf einen Arryindex** `var[1]` miteinander zu kombinieren, was in Unterkapitel 3.3.5.2 allgemein erklärt wird.

Die letzte Phase wird als **Schlussteil** 3.3.5.3 bezeichnet. In dieser Phase wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde nun auf den **Stack** geschrieben. Hierfür wird die **Adresse**, die in den vorherigen Schritten auf dem **Stack** berechnet wurde verwendet. Beim Schreiben des **Inhalts dieses Index** auf den **Stack**, wird dieser die **Adresse** auf dem Stack ersetzen, die in den vorherigen Schritten berechnet wurde. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. In Tabelle 3.14 ist das ganze veranschaulicht. In rot ist der **Inhalt des Feldindex** markiert, der auf den **Stack** geschrieben wurde.

[37] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	1	
$2^{31}+65$	2	SP
$2^{31}+66$	3	
$2^{31}+67$	1	
$2^{31}+68$	2	
$2^{31}+69$	3	
...	...	
...	...	BAF

Tabelle 3.14: *Ausschnitt des Datensegments nach Schlussteil.*

Je nachdem auf welchen **Unterdatentyp** (Definition 3.9) im **Kontext** zuletzt zugegriffen wird, wird der **PicoC-Knoten** `Exp(Stack(Num('1')))` durch verschiedene **semantisch** entsprechende **RETI-Knoten** ersetzt (siehe Unterkapitel 3.3.5.3 für genauere Erklärung). Der **Unterdatentyp** ist dabei über das versteckte Attribut `datatype` des `Exp(exp, datatype)`-Knoten zugänglich.

Definition 3.9: Unterdatentyp

***Datentyp**, der durch einen **Teilbaum** dargestellt wird. Dieser Teilbaum ist ein **Teil** eines Baumes, der einen gesamten **Datentyp** darstellt.*

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Feldindex** (z.B. `ar[1]`) in der `main`-Funktion oder der Funktion `fun` erfolgt, ist eigentlich nur beim **Anfangsteil**, beim Schreiben der **Adresse** der Variable `ar` auf den **Stack** zu finden. Hierbei wird, je nachdem, ob eine Variable in den **Globalen Statischen Daten** liegt oder sie auf dem **Stackframe** liegt, das ganze durch die Knoten `Ref(Global(Num('0')))` oder die Knoten `Ref(Stackframe(Num('1')))` dargestellt, die durch unterschiedliche **semantisch** entsprechende **RETI-Befehle** ersetzt werden.

Anmerkung

Die Berechnung der **Adresse**, ab der ein **Feldelement** am Ende einer Aneinanderreihung von Zugriffen auf **Feldelemente** eines Feldes `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist[a], kann mittels der Formel 3.3.1:

$$ref(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n]) = ref(\texttt{ar}) + \left(\sum_{i=1}^{n}\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\right)\cdot size(\texttt{datatype}) \tag{3.3.1}$$

aus der Betriebssysteme Vorlesung C. Scholl, „Betriebssysteme“ berechnet werden[bc].

Die Knoten `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentieren dabei den Summanden für die **Anfangsadresse** $ref(\texttt{ar})$ in der Formel.

Der Knoten `Exp(num)` repräsentiert dabei einen **Index** beim **Zugriff auf ein Feldelement** (z.B. `j` in `a[i][j][k]`), der als Faktor $\texttt{idx}_i$ in der Formel auftaucht.

Die Knoten `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` repräsentieren dabei einen **ausmultiplizierten Summanden** $\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\cdot size(\texttt{datatype})$ in der Formel.

Die Knoten `Exp(Stack(Num('1')))` repräsentieren dabei das Lesen des **Inhalts** $M\,[ref(\texttt{ar}[\texttt{idx}_1]\ldots[\texttt{idx}_n])]$ der Speicherzelle an der finalen **Adresse** $ref(\texttt{ar}[\texttt{idx}_1]\ldots[\texttt{idx}_n])$.[d]

[a]Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[b]ref(`exp`) steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte.
[c]Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[d]$M[addr]$ ist ein Zugriff auf den **Inhalt** der Speicherzelle an der **Adresse** *addr* im **SRAM**, in der **UART** oder im **EPROM**.

```
File
  Name './example_array_access.picoc_mon',
  [
    Block
      Name 'fun.1',
      [
        // Assign(Name('ar'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        // Exp(Subscr(Name('ar'), Num('0')))
        Ref(Stackframe(Num('0')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
        Exp(Num('1'))
        Exp(Num('2'))
        Exp(Num('3'))
        Assign(Global(Num('0')), Stack(Num('3')))
        // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
        Ref(Global(Num('0')))
        Exp(Num('1'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 3.29: *PicoC-ANF Pass für Zugriff auf einen Feldindex.*

Im **RETI-Blocks Pass** in Code 3.30 werden die **PicoC-Knoten** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2'))undStack(Num('1'))))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
      Name 'fun.1',
      [
        # // Assign(Name('ar'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN BAF ACC -2;
        ADDI SP 1;
        # // Exp(Subscr(Name('ar'), Num('0')))
        # Ref(Stackframe(Num('0')))
        SUBI SP 1;
        MOVE BAF IN1;
        SUBI IN1 2;
        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 IN1 2;
        LOADIN SP IN2 1;
        MULTI IN2 1;
        ADD IN1 IN2;
        ADDI SP 1;
        STOREIN SP IN1 1;
        # Exp(Stack(Num('1')))
        LOADIN SP IN1 1;
        LOADIN IN1 ACC 0;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('3')))
        LOADIN SP ACC 1;
```

```
58          STOREIN DS ACC 2;
59          LOADIN SP ACC 2;
60          STOREIN DS ACC 1;
61          LOADIN SP ACC 3;
62          STOREIN DS ACC 0;
63          ADDI SP 3;
64          # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65          # Ref(Global(Num('0')))
66          SUBI SP 1;
67          LOADI IN1 0;
68          ADD IN1 DS;
69          STOREIN SP IN1 1;
70          # Exp(Num('1'))
71          SUBI SP 1;
72          LOADI ACC 1;
73          STOREIN SP ACC 1;
74          # Exp(Num('1'))
75          SUBI SP 1;
76          LOADI ACC 1;
77          STOREIN SP ACC 1;
78          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79          LOADIN SP ACC 2;
80          LOADIN SP IN2 1;
81          ADD ACC IN2;
82          STOREIN SP ACC 2;
83          ADDI SP 1;
84          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85          LOADIN SP IN1 2;
86          LOADIN SP IN2 1;
87          MULTI IN2 1;
88          ADD IN1 IN2;
89          ADDI SP 1;
90          STOREIN SP IN1 1;
91          # Exp(Stack(Num('1')))
92          LOADIN SP IN1 1;
93          LOADIN IN1 ACC 0;
94          STOREIN SP ACC 1;
95          # Return(Empty())
96          LOADIN BAF PC -1;
97      ]
98  ]
```

Code 3.30: *RETI-Blocks Pass für Zugriff auf einen Feldindex.*

3.3.3.3 Zuweisung an Feldindex

Die Umsetzung einer **Zuweisung** eines Wertes an einen **Feldindex** (z.B. `ar[2] = 42;`) wird im Folgenden anhand des Beispiels in Code 3.31 erläutert.

```
1 void main() {
2   int ar[2];
3   ar[1] = 42;
4 }
```

Code 3.31: *PicoC-Code für Zuweisung an Feldindex.*

Im **Abstrakten Syntaxbaum** in Code 3.32 wird eine **Zuweisung** an einen **Feldindex** `ar[2] = 42;` durch die Knoten `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

```
File
  Name './example_array_assignment.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        Assign(Subscr(Name('ar'), Num('1')), Num('42'))
      ]
  ]
```

Code 3.32: *Abstrakter Syntaxbaum für Zuweisung an Feldindex.*

Im **PicoC-ANF Pass** in Code 3.33 wird zuerst die **rechte** Seite von `Assign(Subscr(Name('ar'), Num('1')), Num('42'))` ausgewertet durch: `Exp(Num('42'))`. Dies ist in Tabelle 3.15 für das Beispiel in Code 3.31 veranschaulicht. Der **Wert** 42 (in rot markiert) wird auf den **Stack** geschrieben.

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	...	
$2^{31}+65$	...	SP
$2^{31}+66$	42	
$2^{31}+67$	...	
$2^{31}+68$	...	
$2^{31}+69$	...	
...	...	
...	...	BAF

Tabelle 3.15: *Ausschnitt des Datensegments nach Auswerten der rechten Seite.*

Danach ist das Vorgehen und die damit verbundenen Knoten, die dieses Vorgehen darstellen: `Ref(Global(Num('0')))`, `Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum **Anfangsteil** und **Mittelteil** aus dem vorherigen Unterkapitel 3.3.3.2. Die eben genannten Knoten stellen die Berechnung der **Adresse** des **Index**, dem das Ergebnis des Logischen Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird dar. Dies ist in Tabelle 3.16 für das Beispiel in Code 3.31 veranschaulicht. Die **Adresse** $2^{31}+68$ (in rot markiert) des **Index** wurde auf dem **Stack** berechnet.

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	1	SP
$2^{31}+65$	$2^{31}+68$	
$2^{31}+66$	42	
$2^{31}+67$	...	
$2^{31}+68$	...	
$2^{31}+69$	...	
...	...	
...	...	BAF

Tabelle 3.16: *Ausschnitt des Datensegments vor Zuweisung.*

Zum Schluss stellen die Knoten `Assign(Stack(Num('1')), Stack(Num('2')))` die **Zuweisung** `stack(1) = stack(2)` des Ergebnisses des Ausdrucks auf der **rechten Seite der Zuweisung** zum **Feldindex** dar. Die **Adresse** des Feldindex wurde im Schritt davor berechnet. Die **Zuweisung** des Wertes 42 an den **Feldindex** `[1]` ist in Tabelle 3.17 veranschaulicht (in rot markiert).[38]

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	1	
$2^{31}+65$	$2^{31}+68$	
$2^{31}+66$	42	SP
$2^{31}+67$	...	
$2^{31}+68$	42	
$2^{31}+69$	...	
...	...	
...	...	BAF

Tabelle 3.17: *Ausschnitt des Datensegments nach Zuweisung.*

```
1  File
2    Name './example_array_assignment.picoc_mon',
3    [
4      Block
5        Name 'main.0',
6        [
7          // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
8          Exp(Num('42'))
9          Ref(Global(Num('0')))
10         Exp(Num('1'))
11         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
12         Assign(Stack(Num('1')), Stack(Num('2')))
13         Return(Empty())
14       ]
15   ]
```

Code 3.33: *PicoC-ANF Pass für Zuweisung an Feldindex.*

[38] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Im **RETI-Blocks Pass** in Code 3.34 werden die **PicoC-Knoten** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Exp(Num('1'))`, `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` und `Assign(Stack(Num('1')),Stack(Num('2')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Subscr(Name('ar'), Num('1')), 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;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        LOADIN SP IN1 2;
        LOADIN SP IN2 1;
        MULTI IN2 1;
        ADD IN1 IN2;
        ADDI SP 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 3.34: *RETI-Blocks Pass für Zuweisung an Feldindex.*

3.3.4 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 3.3.4.1 erläutert. Des Weiteren ist die Umsetzung der **Innitialisierung eines Verbundes** 3.3.4.2, des **Zugriffs auf ein Verbundsattribut** 3.3.4.3 und der **Zuweisung an ein Verbundsattribut** 3.3.4.4 zu klären.

3.3.4.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Umsetzung der **Deklaration** (Definition 1.7) eines neuen **Verbundstyps** (z.B. `struct st {int len; int ar[2];}`) und der **Definition** (Definition 1.8) eines **Verbundes** mit diesem **Verbundstyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 3.35 erläutert.

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

Code 3.35: *PicoC-Code für die Deklaration eines Verbundstyps.*

Bevor ein Verbund definiert werden kann, muss erstmal ein **Verbundstyp** deklariert werden. Im **Abstrakten Syntaxbaum** in Code 3.37 wird die **Deklaration eines Verbundstyps** `struct st {int len; int ar[2];}` durch die Knoten `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 Knoten `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 3.36: *Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.*

Für den **Verbundstyp** selbst und seine **Verbundsattribute** werden in der **Symboltabelle**, die in Code 3.37 dargestellt ist Symboltabelleneintrage mit den **Schlüsseln** `st`, `len@st` und `ar@st` erstellt. Die Schlüssel der

Verbundsattribute haben einen **Suffix** `@st` angehängt, welcher für die Verbundsattribute einen Verbundstyps **indirekt** einen **Sichtbarkeitsbereich** (Definition 1.9) über den **Verbundstyp** selbst erzeugt. Im Unterkapitel 3.3.6.2 wird die **Funktionsweise** von **Sichtbarkeitsbereichen** genauer erläutert. Es gilt folglich, dass **innerhalb** eines **Verbundstyps** zwei Verbundsattribute **nicht** gleich benannt werden können, aber dafür zwei **unterschiedliche Verbundstypen** ihre Verbundsattribute gleich benennen können.

Das Symbol `'@'` wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel 3.3.1.3 erläutert wurde.

Die Attribute[39] der Symboltabelleneinträge für die **Verbundsattribute** sind genauso belegt wie bei **üblichen Variablen**. Die Attribute des Symboltabelleneintrags für den **Verbundstyp** `type qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt. In dem `value or address`-Attribut des Symboltabelleneintrags für den **Verbundstyp** sind die **Verbundsattribute** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Verbundstyp** `st` als Schlüssel die **Verbundsattribute** des Verbundstyps in der **Symboltabelle** nachschlagen kann.

Für die **Definition** einer Variable `st_var@main` mit diesem **Verbundstyp** `st` wird ein **Symboltabelleneintrag** in der **Symboltabelle** angelegt. Das `datatype`-Attribut dieses **Symboltabelleneintrags** enthält dabei den Namen des **Verbundstyps** als `StructSpec(Name('st'))`. Dadurch können jederzeit alle wichtigen Informationen zu diesem **Verbundstyp**[40] und seinen **Verbundsattributen** in der **Symboltabelle** nachgeschlagen werden.

Anmerkung

Die **Anzahl Speicherzellen**, die ein **Verbund** `struct st st_var` belegt[a], der mit dem **Verbundstyp** `struct st` $\{$`datatype`$_1$ `attr`$_1$; ...; `datatype`$_n$ `attr`$_n$;$\}$ definiert ist[b], berechnet sich aus der Summe der **Anzahl Speicherzellen**, welche die einzelnen **Datentypen** `datatype`$_1$ ... `datatype`$_n$ der **Verbundsattribute** `attr`$_1$, ... `attr`$_n$ des **Verbundstyps** belegen: $size(type(\texttt{st\_var})) = \sum_{i=1}^{n} size(\texttt{datatype}_i)$.[c][d]

[a] Die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.
[b] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[c] Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein **Datentyp** belegt.
[d] Die Funktion *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

```
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       {
```

[39] Die über einen **Bezeichner** selektierbaren Elemente eines **Symboltabelleneintrags** und eines **Verbunds** heißen bei beiden **Attribute**.
[40] Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Verbundstyp** einnimmt.

```
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()
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 3.37: *Symboltabelle für die Deklaration eines Verbundstyps.*

3.3.4.2 Initialisierung von Verbunden

Die Umsetzung der **Initialisierung eines Verbundes** wird im Folgenden mithilfe des Beispiels in Code 3.38 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 3.38: *PicoC-Code für Initialisierung von Verbunden.*

Im **Abstrakten Syntaxbaum** in Code 3.39 wird die **Initialisierung eines Verbundes** `struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}}` mithilfe der **Knoten** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` dargestellt.

```
File
  Name './example_struct_init.ast',
  [
    StructDecl
      Name 'st1',
      [
        Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
        ↪  Name('attr'))
      ],
    StructDecl
      Name 'st2',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        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'))]))]))]))])
      ]
  ]
```

Code 3.39: *Abstrakter Syntaxbaum für Initialisierung von Verbunden.*

Im **PicoC-ANF Pass** in Code 3.40 wird `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Feldes** in Unterkapitel 3.3.3.1. Für **genauere Details** wird an dieser Stelle daher auf Unterkapitel 3.3.3.1 verwiesen. Um das Ganze interessanter zu gestalten, wurde das Beispiel in Code 3.38 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der Teilbaum `Struct([Assign(Name('attr1'),Name('var')),Assign(Name('attr2'),Struct([Assign(Name('attr'),Array([Array([Ref(Name('var')),Ref(Name('var'))])]))]))])`, der beim äußersten **Verbund-Initialisierer**-Knoten `Struct(...)` anfängt, wird auf dieselbe Weise nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Feldes** in Unterkapitel 3.3.3.1 bereits erklärt wurde. Beim **Iterieren** über den **Teilbaum**, muss bei einem **Verbund-Initialisierer**-Knoten `Struct(...)` nur beachtet werden, dass bei den `Assign(exp1, exp2)`-Knoten[41] der Teilbaum beim rechten `exp2` Attribut weitergeht.

Im Allgemeinen gibt es im Teilbaum beim **Initialisieren** eines **Feldes** oder **Verbundes** auf der **rechten Seite** immer nur 3 Fälle. Auf der **rechten** Seite hat man es entweder mit einem **Verbund-Initialiser**, einem

[41] Über welche die **Attributzuweisung** (z.B. `.attr2={.attr={&var,&var}}`) als z.B. `Assign(Name('attr2'),Struct([Assign(Name('attr'),Array([Array([Ref(Name('var')),Ref(Name('var'))])]))])` dargestellt wird.

Feld-Initialiser oder einem **Logischen Ausdruck** zu tun. Bei einem **Feld-** oder **Verbund-Initialiser** wird über diesen nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** iteriert und mithilfe von `Exp(exp)`-Knoten die Auswertung der **Logischen Ausdrücke** in den **Blättern** auf den **Stack** dargestellt. Der Fall, dass ein **Logischer Ausdruck** vorliegt erübrigt sich somit.

```
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 3.40: *PicoC-ANF Pass für Initialisierung von Verbunden.*

Im **RETI-Blocks Pass** in Code 3.41 werden die **PicoC-Knoten** `Exp(Global(Num('0')))`, `Ref(Global(Num('0')))` und `Assign(Global(Num('1')),Stack(Num('3')))` durch ihre **semantisch** 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;
```

```
21          # Ref(Global(Num('0')))
22          SUBI SP 1;
23          LOADI IN1 0;
24          ADD IN1 DS;
25          STOREIN SP IN1 1;
26          # Ref(Global(Num('0')))
27          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 3.41: *RETI-Blocks Pass für Initialisierung von Verbunden.*

3.3.4.3 Zugriff auf Verbundsattribut

Die Umsetzung des **Zugriffs auf ein Verbundsattribut** (z.B. `st.y`) wird im Folgenden mithilfe des Beispiels in Code 3.42 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 3.42: *PicoC-Code für Zugriff auf Verbundsattribut.*

Im **Abstrakten Syntaxbaum** in Code 3.43 wird der **Zugriff auf ein Verbundsattribut** `st.y` mithilfe der Knoten `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 3.43: *Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 3.44 werden die Knoten `Exp(Attr(Name('st'), Name('y')))` auf eine ähnliche Weise ausgewertet, wie die Knoten `Exp(Subscr(Name('ar'), Num('0')))`, die in Unterkapitel 3.3.3.2 einen **Zugriff auf ein Feldelement** darstellen. Daher wird hier, um Redundanz zu vermeiden, nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 3.3.3.2 verwiesen.

Die Knoten `Exp(Attr(Name('st'), Name('y')))` werden genauso, wie in Unterkapitel 3.3.3.2 durch Knoten ersetzt, die sich in **Anfangsteil** 3.3.5.1, **Mittelteil** 3.3.5.2 und **Schlussteil** 3.3.5.3 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 umgesetzt, wie in Unterkapitel 3.3.3.2.

Nur für den **Mittelteil** werden andere Knoten `Ref(Attr(Stack(Num('1')), Name('y')))` gebraucht. Diese Knoten `Ref(Attr(Stack(Num('1')), Name('y')))` stellen die Aufgabe dar, die **Anfangsadresse** des **Attributs** auf welches zugegriffen wird zu berechnen und auf den Stack zu legen. Hierfür wird die **Anfangsadresse** des **Verbundes**, in dem dieses Attribut liegt verwendet. Das auf den **Stack**-Speichern dieser **Anfangsadresse** wird durch Knoten des **Anfangsteils** dargestellt: `Ref(Global(Num('0')))`.

Beim **Zugriff auf einen Feldindex** musste vorher durch z.B. `Exp(Num('3'))` die **Berechnung des Indexwerts** und das auf den **Stack** legen des Ergebnisses dargestellt werden. Beim **Zugriff auf ein Verbundsattribut** steht der Bezeichner des **Verbundsattributs** `Name('y')` dagegen bereits während des Kompilierens in `Ref(Attr(Stack(Num('1')), Name('y')))` zur Verfügung. Der **Verbundstyp**, dem dieses Attribut gehört, wird im **Mittelteil** aus dem versteckten Attribut `datatype` des Knoten `Ref(exp, datatype)` herausgelesen. Der **Verbundstyp** wird während des Kompiliervorgangs im **PiocC-ANF Pass** dem Knoten `Ref(exp, datatype)` über das versteckten Attribut `datatype` angehängt.

Anmerkung

Im Unterkapitel 3.3.5.2 wird mit der **allgemeinen Formel** 3.3.3 ein allgemeines Vorgehen zur **Adressberechnung** für alle möglichen Aneinanderreihungen von **Zugriffen** auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** erklärt. Um die **Adresse**, ab der ein **Verbundsattribut** am Ende einer Aneinanderreihung von **Zugriffen** auf **Verbundsattribute** abgespeichert ist, zu berechnen, kann diese **allgemeine Formel** 3.3.3 ebenfalls genutzt werden. Im Gegensatz zu **Feldern**, lässt sich bei Verbunden **keine** vereinfachte Formel aus der allgemeinen Formel bilden, da die **Verbundsattribute** eines Verbunds **unterschiedlich viele Speicherzellen** belegen können.

```
1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
```

```
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         Exp(Num('4'))
9         Exp(Num('2'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Exp(Attr(Name('st'), Name('y')))
12        Ref(Global(Num('0')))
13        Ref(Attr(Stack(Num('1')), Name('y')))
14        Exp(Stack(Num('1')))
15        Return(Empty())
16      ]
17  ]
```

Code 3.44: *PicoC-ANF Pass für Zugriff auf Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 3.45 werden die **PicoC-Knoten** `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')),Name('y')))` und `Exp(Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # Exp(Num('4'))
9         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 3.45: *RETI-Blocks Pass für Zugriff auf Verbundsattribut.*

3.3.4.4 Zuweisung an Verbundsattribut

Die Umsetzung der **Zuweisung an ein Verbundsattribut** (z.B. `st.y = 42`) wird im Folgenden anhand des Beispiels in Code 3.46 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 3.46: *PicoC-Code für Zuweisung an Verbundsattribut.*

Im **Abstrakten Syntaxbaum** wird eine **Zuweisung an ein Verbundsattribut** `st.y = 42` durch die Knoten `Assign(Attr(Name('st'), Name('y')), Num('42'))` dargestellt.

```
1  File
2    Name './example_struct_attr_assignment.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         Assign(Attr(Name('st'), Name('y')), Num('42'))
17       ]
18   ]
```

Code 3.47: *Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 3.48 werden die Knoten `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf eine ähnliche Weise ausgewertet, wie die Knoten `Assign(Subscr(Name('ar'), Num('2')), Num('42'))`, die in Unterkapitel 3.3.3.3 einen **Zugriff auf ein Feldelement** darstellen. Daher wird hier, um Redundanz zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 3.3.3.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 3.3.3.3 muss hier zum Auswerten des **linken** Knoten `Attr(Name('st'),Name('y'))` von `Assign(Attr(Name('st'),Name('y')),Num('42'))` wie in Unterkapitel 3.3.4.3 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 3.48: *PicoC-ANF Pass für Zuweisung an Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 3.49 werden die **PicoC-Knoten** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')),Name('y')))` und `Assign(Stack(Num('1')),Stack(Num('2')))` durch ihre **semantisch** 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;
```

```
18          STOREIN DS ACC 1;
19          LOADIN SP ACC 2;
20          STOREIN DS ACC 0;
21          ADDI SP 2;
22          # // Assign(Attr(Name('st'), Name('y')), Num('42'))
23          # Exp(Num('42'))
24          SUBI SP 1;
25          LOADI ACC 42;
26          STOREIN SP ACC 1;
27          # Ref(Global(Num('0')))
28          SUBI SP 1;
29          LOADI IN1 0;
30          ADD IN1 DS;
31          STOREIN SP IN1 1;
32          # Ref(Attr(Stack(Num('1')), Name('y')))
33          LOADIN SP IN1 1;
34          ADDI IN1 1;
35          STOREIN SP IN1 1;
36          # Assign(Stack(Num('1')), Stack(Num('2')))
37          LOADIN SP IN1 1;
38          LOADIN SP ACC 2;
39          ADDI SP 2;
40          STOREIN IN1 ACC 0;
41          # Return(Empty())
42          LOADIN BAF PC -1;
43      ]
44  ]
```

Code 3.49: *RETI-Blocks Pass für Zuweisung an Verbndsattribut.*

3.3.5 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

In den Unterkapiteln 3.3.2, 3.3.3 und 3.3.4 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln vorkommenden Datentypen (**Zeiger**, **Feld** und **Verbund**) sehr **ähnlich** abläuft. Es lässt sich ein **allgemeines Vorgehen**, bestehend aus einem **Anfangsteil** 3.3.5.1 , **Mittelteil** 3.3.5.2 und **Schlussteil** 3.3.5.3 darin erkennen. In diesem **allgemeinen Vorgehen** lassen sich die verschiedenen Zugriffsarten für **Elemente** / **Attribute** von **Zeigern** (z.B. `*(pntr + i)`), **Feldern** (z.B. `ar[i]`) und **Verbunden** (z.B. `st.attr`) miteinander kombinieren und so **gemischte Ausdrücke**, wie z.B. `(*st_first.ar)[0]` bilden. Dieses **allgemeine Vorgehen** ist in Abbildung 3.8 veranschaulicht. Die **Formelzeichen** der **Formeln** in Abbildung 3.8 werden zusammen mit Formel 3.3.1 und Formel 3.3.3 erklärt.

Gemischte Ausdrücke sind 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 ein anderer **RETI-Code** generiert wird. Hierzu muss im `exp`-Attribut des `Ref(exp, datatype)`-Knoten die passende **Zugriffsoperation** `Subscr(exp1, exp2)` oder `Attr(exp, name)` vorliegen.

Das gerade erwähnte Vorgehen berechnet die **Startadresse** eines gewünschten **Zeigerelementes**, **Feldelementes** oder **Verbundsattributes**. Zur Berechnung wird die **Startadresse** des **Zeigers**, **Feldes** oder **Verbundes**, dessen **Attribut** oder **Element** berechnet werden soll gebraucht. Die **Startadresse** wird in einem **vorherigen Berechnungschritt** oder im **Anfangsteil** auf den **Stack** geschrieben. Bei einem **Zugriff** auf einen **Feldindex** wird zudem mithilfe von entsprechendem **RETI-Code** dafür gesorgt, dass beim Ausführen zur **Laufzeit** der **Wert** des **Index** berechnet wird und nach der **Startadresse** auf den **Stack**

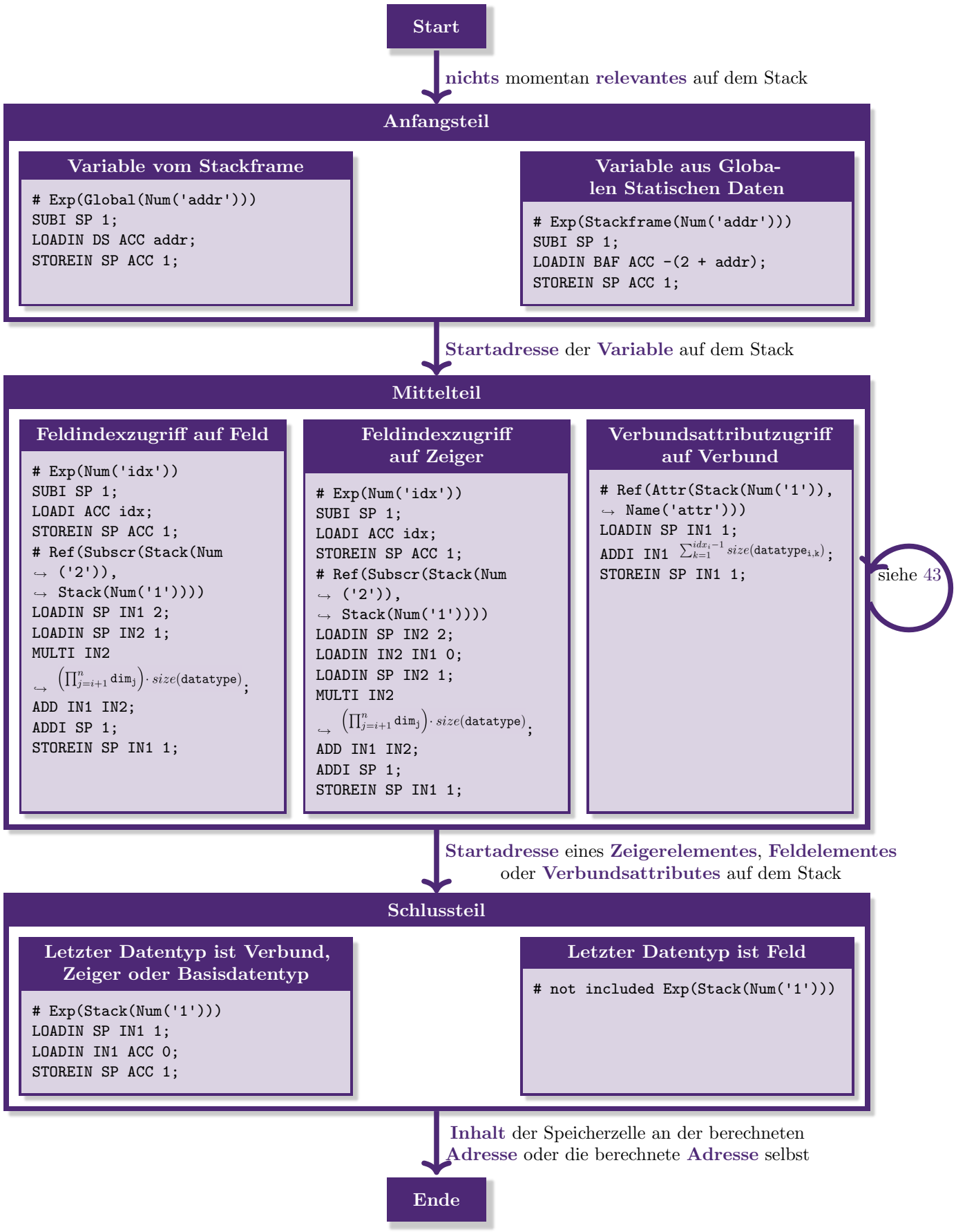


Abbildung 3.8: *Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.*

geschrieben wird. Dies wurde in Unterkapitel 3.3.3.2 bereits veranschaulicht.

Würde man bei einer Operation `Subsc(Name('var'), Num('0'))` den in der **Symboltabelle** gespeicherten **Datentyp** der Variable `Name('var')` von `ArrayDecl([Num('3')], IntType())` zu `PointerDecl(Num('1'), IntType())` ändern, müssten beim generierten **RETI-Code** nur die **RETI-Befehle** des **Mittelteils** ausgetauscht werden. Die **RETI-Befehle** des **Anfangsteils** würden unverändert bleiben, da die Variable immer noch entweder in den **Globalen Statischen Daten** oder in einem **Stackframe** abgespeichert ist. Die **RETI-Befehle** des **Schlussteils** würden unverändert bleiben, da der **letzte Datentyp** auf den Zugegriffen wird immer noch `IntType()` ist.

Im `Ref(exp, datatype)`-Knoten muss die **Zugriffsoperation** im `exp`-Attribut zum **Datentyp** im versteckten Attribut `datatype` passen. Im Fall, dass Operation und Datentyp **nicht zusammenpassen**, gibt es eine `DatatypeMismatch`-**Fehlermeldung**. Ein **Zugriff** auf einen **Feldindex** `Subscr(exp1, exp2)` kann dabei mit den Datentypen **Feld** `ArrayDecl(nums, datatype)` und **Zeiger** `PntrDecl(num, datatype)` kombiniert werden. Allerdings wird für beide Kombinationen unterschiedlicher **RETI-Code** generiert. Das liegt daran, dass in der Speicherzelle des **Zeigers** `PntrDecl(num, datatype)` eine **Adresse** steht und das **gewünschte Element** erst zu finden ist, wenn man dieser **Adresse** folgt. Hierfür muss ein anderer **RETI-Code** erzeugt werden, wie für ein **Feld** `ArrayDecl(nums, datatype)`, bei dem direkt auf dessen **Elemente** zugegriffen werden kann. Ein **Zugriff auf ein Verbundsattribut** `Attr(exp, name)` kann nur mit dem **Verbundsdatentyp** `StructSpec(name)` kombiniert werden.[42]

Anmerkung

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine **Dereferenzierung** in der Form `Deref(exp1, exp2)` nicht mehr existiert. In Unterkapitel 3.3.2.2 wurde bereits erklärt, dass alle Knoten `Deref(exp1, exp2)` im **PicoC-Shrink Pass** durch `Subscr(exp1, exp2)` ersetzt wurden. 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**.

Der **Anfangsteil**, der durch die Knoten `Ref(Name('var'))` repräsentiert wird, ist dafür zuständig die **Startadresse** der Variablen `Name('var')` auf den **Stack** zu schreiben. Je nachdem, ob diese Variable in den **Globalen Statischen Daten** oder auf einem **Stackframe** liegt, wird ein anderer **RETI-Code** generiert.

Der **Schlussteil** wird durch die Knoten `Exp(Stack(Num('1')), datatype)` dargestellt. Wenn das versteckte Attribut `datatype` ein `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructSpec(name)` ist, wird ein entsprechender **RETI-Code** generiert. Dieser **RETI-Code** nutzt die **Adresse**, die in den vorherigen Phasen auf dem **Stack** berechnet wurde dazu, um den **Inhalt** der Speicherzelle an dieser **Adresse** auf den **Stack** zu schreiben. Hierbei wird die Speicherzelle, in welcher die **Adresse** steht mit dem **Inhalt** auf den sie selbst zeigt überschrieben. 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[44], die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache L_{PicoC} implementieren Datentypen[45] wird immer der **Inhalt** der Speicherzelle der ersten Elements bzw. Elements ausgegeben.

[42] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.
[43] **Startadresse** eines **Zeigerelementes**, **Feldelementes** oder **Verbundsattributes** auf dem Stack.
[44] Und nicht auf ein **Element** des Feldes, welches den Datentyp `CharType()` oder `IntType()`, `PntrDecl(num, datatype)` oder `StructSpec(name)` hat.
[45] Also `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructSpec(name)`.

3.3.5.1 Anfangsteil

Die Umsetzung des **Anfangsteils**, bei dem die **Startadresse** einer Variable auf den **Stack** geschrieben wird, wird im Folgenden mithilfe des Beispiels in Code 3.50 erklärt. Der **Anfangsteil** entspricht dem Anwenden des **Referenzierungsoperators** auf eine Variable: `&var`.

```
1 void main() {
2   int *(*complex_var)[3];
3   &complex_var;
4 }
5
6 void fun() {
7   int (*complex_var)[3];
8   &complex_var;
9 }
```

Code 3.50: *PicoC-Code für den Anfangsteil.*

Im **Abstrakten Syntaxbaum** in Code 3.51 wird die **Refererenzierung** `&complex_var` mit den Knoten `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird für eine **Referenzierung** einfach nur `Ref(Name('complex_var')` geschrieben, aber da beim Erstellen des **Abstrakten Syntaxbaums** jeder **Logische Ausdruck** in ein `Exp(exp)` eingebettet wird, ist das `Ref(Name('complex_var')` in ein `Exp(exp)` eingebettet. **Semantisch** macht es in diesem Pass **keinen Unterschied**, ob an einer Stelle `Ref(Name('complex_var')` oder `Exp(Ref(Name('complex_var')))` steht. Man müsste an vielen Stellen eine gesonderte **Fallunterschiedung** aufstellen, um bei `Exp(Ref(Name('complex_var')))` das `Exp(exp)` zu entfernen. Das `Exp(exp)` wird allerdings in den darauffolgenden **Passes** sowieso herausgefiltert. Daher wurde darauf verzichtet den Code der Implementierung ohne triftigen Grund **komplexer** zu machen.

```
1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
          ↪  IntType('int')))), Name('complex_var')))
10        Exp(Ref(Name('complex_var')))
11      ],
12    FunDef
13      VoidType 'void',
14      Name 'fun',
15      [],
16      [
17        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('complex_var')))
18        Exp(Ref(Name('complex_var')))
19      ]
20  ]
```

Code 3.51: *Abstrakter Syntaxbaum für den Anfangsteil.*

Im **PicoC-ANF Pass** in Code 3.52 werden die Knoten `Exp(Ref(Name('complex_var')))`, je nachdem, ob die Variable `Name('complex_var')` in den **Globalen Statischen Daten** oder in einem **Stackframe** liegt durch die Knoten `Ref(Global(Num('0')))` oder `Ref(Stackframe(Num('0')))` ersetzt.[46]

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

Code 3.52: *PicoC-ANF Pass für den Anfangsteil.*

Im **RETI-Blocks Pass** in Code 3.53 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` bzw. `Ref(Stackframe(Num('0')))` durch ihre **semantisch** 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('0')))
        SUBI SP 1;
        LOADI IN1 0;
        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('0')))
        SUBI SP 1;
        MOVE BAF IN1;
        SUBI IN1 2;
```

[46] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

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

Code 3.53: *RETI-Blocks Pass für den Anfangsteil.*

3.3.5.2 Mittelteil

Der Umsetzung des **Mittelteils**, bei dem die **Startadresse** des letzten Attributes / Elementes einer Aneinanderreihung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute** berechnet wird (z.B. `(*complex_var.ar)[2-2]`), wird im Folgenden mithilfe des Beispiels in Code 3.54 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 3.54: *PicoC-Code für den Mittelteil.*

Im **Abstrakten Syntaxbaum** in Code 3.55 wird die Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** `(*complex_var.ar)[2-2]` durch die Knoten `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 3.55: *Abstrakter Syntaxbaum für den Mittelteil.*

Im **PicoC-Shrink Pass** in Code 3.56 wird der `Deref(exp1,Num('0'))`-Knoten in `Exp(Subscr(Deref(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2'))))` durch den `Subscr(exp1,Num('0'))`-Knoten in `Exp(Subscr(Subscr(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2'))))` ersetzt.

```
File
  Name './example_derived_dts_main_part.picoc_shrink',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
        ↪ Name('ar'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
        ↪ Array([Num('42')]))
        Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),
        ↪ Struct([Assign(Name('ar'), Ref(Name('var')))]))
        Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
        ↪ Sub('-'), Num('2'))))
      ]
  ]
```

Code 3.56: *PicoC-Shrink Pass für den Mittelteil.*

Im **PicoC-ANF Pass** in Code 3.57 werden die Knoten `Exp(Subscr(Deref(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2'))))` durch die Knoten `Ref(Attr(Stack(Num('1')),Name(str)))`, `Exp(num)`, `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` ersetzt. Bei z.B. dem `Subscr(exp1,exp2)`-Knoten wird dieser einfach dem `exp`-Attribut des `Ref(exp)`-Knoten zugewiesen und die **Indexberechnung** für `exp2` davor gezogen.

Bei `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` wird über `Stack(Num('1'))` auf das Ergebnis der **Indexberechnung** auf dem **Stack** zugegriffen und über `Stack(Num('2'))` auf die **Startadresse** des momentanen **Feldes**, in dem das **Feldelement** liegt, auf das zugegriffen werden soll. Diese **Startadresse** wurde **vorher** in einer **vorherigen Adressberechnung** oder durch den **Anfangsteil** auf den **Stack** geschrieben. Die vorhin erwähnte **Indexberechnung** wird bei `Exp(Subscr(exp1, BinOp(Num('2'), Sub('-'), Num('2'))))` durch die Knoten `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))` dargestellt.[47]

Anmerkung

Sei $\texttt{datatype}_\texttt{i}$ ein **Folgeglied** einer **Folge** $(\texttt{datatype}_\texttt{i})_{i=1,\dots,n+1}$, dessen erstes Folgeglied $\texttt{datatype}_\texttt{1}$ ist. Dabei steht `i` für eine **Ebene** eines Baumes. Die Folgeglieder der Folge lassen sich **Startadressen**

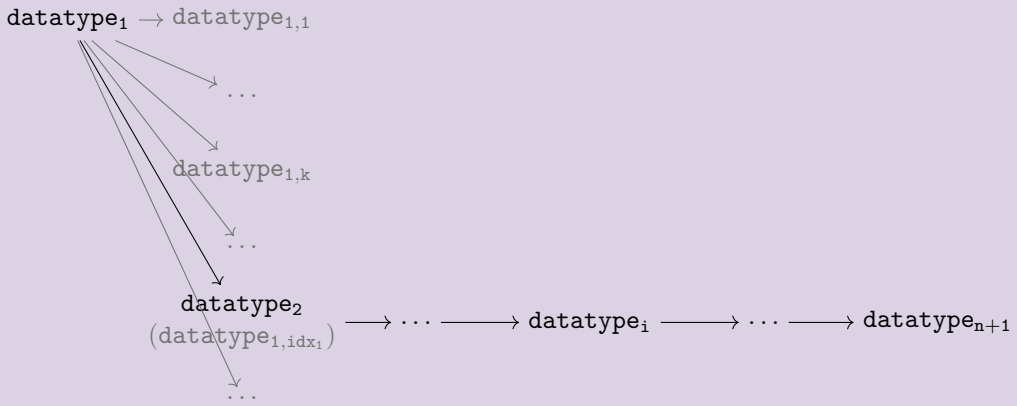
[47] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

$ref(\texttt{datatype}_\texttt{i})$ von Speicherbereichen $ref(\texttt{datatype}_\texttt{i})$ ... $ref(\texttt{datatype}_\texttt{i}) + size(\texttt{datatype}_\texttt{i})$ im **Hauptspeicher** zuordnen. Hierbei gilt, dass $ref(\texttt{datatype}_\texttt{i}) \leq ref(\texttt{datatype}_{\texttt{i+1}}) < ref(\texttt{datatype}_\texttt{i}) + size(\texttt{datatype}_\texttt{i})$.[ab]

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

Sei $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ das **Element / Attribut** des **Datentyps** $\texttt{datatype}_\texttt{i}$ für das gilt: $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}} = \texttt{datatype}_{\texttt{i+1}}$. Hierbei ist $\texttt{idx}_\texttt{i}$ der **Index**[c] des **Elements / Attributs** auf welches zugegriffen wird innerhalb des Datentyps $\texttt{datatype}_\texttt{i}$.

In Abbildung 3.3.2 ist das ganze veranschaulicht. Die ausgegrauten Knoten stellen die verschiedenen **Elemente** / **Attribute** $\texttt{datatype}_{\texttt{i,k}}$ des **Datentyps** $\texttt{datatype}_\texttt{i}$ dar. Allerdings können nur die Knoten $\texttt{datatype}_\texttt{i}$ **Folgeglieder** der **Folge** $(\texttt{datatype}_\texttt{i})_{i=1,\ldots,n+1}$ darstellen.



(3.3.2)

Die **Adresse**, ab der ein **Element / Attribut** am Ende einer Folge $(\texttt{datatype}_{\texttt{i,idx}_\texttt{i}})_{i=1,\ldots,n}$ verschiedener **Elemente / Attribute** abgespeichert ist, kann mittels der Formel 3.3.3 berechnet werden. Diese Folge ist das Resultat einer Aneinanderreihung von **Zugriffen** auf **Feldelemente** und **Verbundsattributte** unterschiedlicher Datentypen $\texttt{datatype}_\texttt{i}$ (z.B. `*complex_var.attr3[2]`).

$$ref(\texttt{datatype}_{\texttt{1,idx}_\texttt{1}},\ \ldots,\ \texttt{datatype}_{\texttt{n,idx}_\texttt{n}}) = ref(\texttt{datatype}_\texttt{1}) + \sum_{i=1}^{n} \sum_{k=1}^{idx_i - 1} size(\texttt{datatype}_{\texttt{i,k}}) \quad (3.3.3)$$

Die **äußere Schleife** iteriert nacheinander über die Folge von **Attributen / Elementen** $(\texttt{datatype}_{\texttt{i,idx}_\texttt{i}})_{i=1,\ldots,n}$, die aus den **Zugriffen** auf **Feldelemente** oder **Verbundsattribute** resultiert (z.B. `*complex_var.attr3[2]`). Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** $\texttt{datatype}_{\texttt{i,k}}$ des momentan betrachteten **Datentyps** $\texttt{datatype}_\texttt{i}$, die vor dem **Element / Attribut** $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ liegen.

Dabei darf nur das letzte Folgenglied $\texttt{datatype}_{\texttt{n+1}}$ vom Datentyp **Zeiger** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Zeiger**, der nicht der **letzte Datentyp** $\texttt{datatype}_{\texttt{n+1}}$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden. Dabei geht die **erste Adressberechnung** vom ersten Datentyp $\texttt{datatype}_\texttt{1}$ bis zum Zeiger-

Datentyp $\texttt{datatype}_{\texttt{pntr}}$ und die **zweite Adressberechnung** fängt einen Datentyp nach dem Zeiger-Datentyp an $\texttt{datatype}_{\texttt{pntr+1}}$ und geht bis zum letzten Datentyp $\texttt{datatype}_{\texttt{n+1}}$. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $ref(\texttt{datatype}_1)$ des Summanden aus der Formel 3.3.3 auf den **Inhalt**[d] der Speicherzelle an der **Adresse**, welche in der **ersten Adressberechnung**[e] $ref(\texttt{datatype}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{datatype}_{\texttt{pntr}-1,\texttt{idx}_{\texttt{pntr}-1}})$ berechnet wurde gesetzt werden: $M\left[ref(\texttt{datatype}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{datatype}_{\texttt{pntr}-1,\texttt{idx}_{\texttt{pntr}-1}})\right]$.[fg]

Die Formel 3.3.3 stellt dabei eine **Verallgemeinerung** der Formel 3.3.1 dar, die für alle möglichen Aneinanderreihungen von Zugriffen auf **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 3.3.1, da man **nicht** davon ausgehen kann, dass alle Elemente / Attribute den **gleichen Datentyp** haben[h].

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

Die Knoten `Exp(Num(num))` bzw. `Name(str)` aus `Ref(Attr(Stack(Num(num)), Name(str)))` repräsentieren dabei das $\texttt{idx}_\texttt{i}$ in der Formel.

Die Knoten `Ref(Attr(Stack(Num('1')), name))` bzw. `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` repräsentieren dabei einen Summanden $\sum_{k=1}^{idx_i-1} size(\texttt{datatype}_{\texttt{i,k}})$ in der Formel.

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

[a] $ref(\texttt{datatype})$ ordent dabei dem **Datentyp** `datatype` eine **Startadresse** zu.
[b] Die Funktion `size` berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[c] Man fängt hier bei den **Indices** von 1 zu zählen an.
[d] Der **Inhalt** dieser Speicherzelle ist eine **Adresse**, da im momentanen Kontext ein **Zeiger** betrachtet wird.
[e] Hierbei kommt die **Adresse** des **Zeigers** selbst raus.
[f] $M[addr]$ ist ein Zugriff auf den **Inhalt** der Speicherzelle an der **Adresse** *addr* im **SRAM**, in der **UART** oder im **EPROM**.
[g] Zur Erinnerung: $\texttt{datatype}_{\texttt{pntr}-1,\texttt{idx}_{\texttt{pntr}-1}} = \texttt{datatype}_{\texttt{pntr}}$, es wird also die **Adresse** des **Zeigers** berechnet und der **Inhalt** der **Speicherzelle** an dieser **Adresse**, der wiederum eine **Adresse** ist, wird als **Startadresse** der **zweiten Adressberechnung** verwendet.
[h] Verbundsattribute haben z.B. **unterschiedliche** Größen.

```
File
  Name './example_derived_dts_main_part.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
        ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
        Ref(Global(Num('1')))
        Ref(Attr(Stack(Num('1')), Name('ar')))
        Exp(Num('0'))
```

```
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Num('2'))
        Exp(Num('2'))
        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 3.57: *PicoC-ANF Pass für den Mittelteil.*

Im **RETI-Blocks Pass** in Code 3.58 werden die **PicoC-Knoten** `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 **semantisch** entsprechenden **RETI-Knoten** ersetzt. Bei der Generierung des **RETI-Code** muss auch das versteckte Attribut `datatype` des `Ref(exp, datatype)`-Knoten berücksichtigt werden, wie es am Anfang dieses Unterkapitels 3.3.5 zusammen mit der Abbildung 3.8 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;
```

```
34          ADDI IN1 0;
35          STOREIN SP IN1 1;
36          # Exp(Num('0'))
37          SUBI SP 1;
38          LOADI ACC 0;
39          STOREIN SP ACC 1;
40          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41          LOADIN SP IN2 2;
42          LOADIN IN2 IN1 0;
43          LOADIN SP IN2 1;
44          MULTI IN2 1;
45          ADD IN1 IN2;
46          ADDI SP 1;
47          STOREIN SP IN1 1;
48          # Exp(Num('2'))
49          SUBI SP 1;
50          LOADI ACC 2;
51          STOREIN SP ACC 1;
52          # Exp(Num('2'))
53          SUBI SP 1;
54          LOADI ACC 2;
55          STOREIN SP ACC 1;
56          # 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 3.58: *RETI-Blocks Pass für den Mittelteil.*

3.3.5.3 Schlussteil

Die Umsetzung des **Schlussteils**, bei dem entweder der **Inhalt** der Speicherzelle an der im **Anfangsteil** 3.3.5.1 und **Mittelteil** 3.3.5.2 berechneten **Adresse** auf den **Stack** geschrieben wird oder diese **Adresse** selbst auf den **Stack** geschrieben wird[48], wird im Folgenden mithilfe des Beispiels in Code 3.59 erklärt.

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

```c
struct st {int attr[2];};

void main() {
  int complex_var1[1][2];
  struct st complex_var2[1];
  int var = 42;
  int *pntr1 = &var;
  int **complex_var3 = &pntr1;

  complex_var1[0];
  complex_var2[0];
  *complex_var3;
}
```

Code 3.59: *PicoC-Code für den Schlussteil.*

Die Generierung des **Abstrakten Syntaxbaumes** in Code 3.60 verläuft wie üblich.

```
File
  Name './example_derived_dts_final_part.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
        ↪  Name('complex_var1')))
        Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
        ↪  Name('complex_var2')))
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),
        ↪  Ref(Name('var')))
        Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),
        ↪  Ref(Name('pntr1')))
        Exp(Subscr(Name('complex_var1'), Num('0')))
        Exp(Subscr(Name('complex_var2'), Num('0')))
        Exp(Deref(Name('complex_var3'), Num('0')))
      ]
  ]
```

Code 3.60: *Abstrakter Syntaxbaum für den Schlussteil.*

Im **PicoC-ANF Pass** in Code 3.61 wird das am Anfang dieses **Unterkapitels** angesprochene auf den **Stack** speichern des **Inhalts** der Speicherzelle an der im **Anfangsteil** 3.3.5.1 und **Mittelteil** 3.3.5.2 berechneten **Adresse** oder dieser **Adresse** selbst, mit den Knoten `Exp(Stack(Num('1')))` dargestellt.

```
File
  Name './example_derived_dts_final_part.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Global(Num('4')), Stack(Num('1')))
        // Assign(Name('pntr1'), Ref(Name('var')))
        Ref(Global(Num('4')))
        Assign(Global(Num('5')), Stack(Num('1')))
        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
        Ref(Global(Num('5')))
        Assign(Global(Num('6')), Stack(Num('1')))
        // Exp(Subscr(Name('complex_var1'), Num('0')))
        Ref(Global(Num('0')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        // Exp(Subscr(Name('complex_var2'), Num('0')))
        Ref(Global(Num('2')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        // Exp(Subscr(Name('complex_var3'), Num('0')))
        Ref(Global(Num('6')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 3.61: *PicoC-ANF Pass für den Schlussteil.*

Im **RETI-Blocks Pass** in Code 3.62 werden die **PicoC-Knoten** `Exp(Stack(Num('1')))` durch **semantisch** entsprechende **RETI-Knoten** ersetzt. Wenn das versteckte Attribut `datatype` im `Exp(exp, datatype)`-Knoten kein **Feld** `ArrayDecl(nums, datatype)` enthält, dann schreiben diese **semantisch** entsprechenden **RETI-Knoten** den Inhalt der Speicherzelle an der im **Anfangsteil** 3.3.5.1 und **Mittelteil** 3.3.5.2 berechneten **Adresse** auf den **Stack**.

Bei einem **Feld** `ArrayDecl(nums, datatype)` im versteckten Attribut `datatype` ist die **Adresse**, die in vorherigen Schritten auf dem **Stack** berechnet wurde bereits das gewünschte Ergebnis. Genaueres wurde am **Anfang** dieses Unterkapitels 3.3.5 zusammen mit der Abbildung 3.8 bereits erklärt.

```
File
  Name './example_derived_dts_final_part.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // 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 3.62: *RETI-Blocks Pass für den Schlussteil.*

3.3.6 Umsetzung von Funktionen

Um die **Umsetzung** von **Funktionen** zu verstehen, ist es erstmal wichtig zu verstehen, wie **Funktionen** später im **RETI-Code** aussehen (Unterkapitel 3.3.6.1), wie Funktionen **deklariert** (Definition 1.7) und **definiert** (Definition 1.8) werden können und hierbei **Sichtbarkeitsbereiche** (Definition 1.9) umgesetzt sind (Unterkapitel 3.3.6.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines **Funktionsaufrufes** erklärt werden (Unterkapitel 3.3.6.3). Beim Thema **Funktionsaufruf** wird im speziellen darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 3.3.6.3.1) umgesetzt sind und die **Übergabe** von **Zusammengesetzten Datentypen**, die mehr als eine Speicherzelle belegen, wie **Verbunden** (Unterkapitel 3.3.6.3.3) und **Feldern** (Unterkapitel 3.3.6.3.2) umgesetzt ist.

3.3.6.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 3.63 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** übersetzt werden.

```
1  void main() {
2    return;
3  }
4
5  void fun1() {
6    int var = 41;
7    if(1) {
8      var = 42;
9    }
10 }
11
12 int fun2() {
13   return 1;
14 }
```

Code 3.63: *PicoC-Code für 3 Funktionen.*

Im **Abstrakten Syntaxbaum** in Code 3.64 werden die 3 **Funktionen** durch entsprechende Knoten dargestellt. Am Beispiel der **Funktion** `void fun2() {return 1;}` wären die hierzu passenden **Knoten** `FunDef(VoidType(), Name('fun2'), [], [Return(Num('1'))])`.[49]

```
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
```

[49] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
13      VoidType 'void',
14      Name 'fun1',
15      [],
16      [
17        Assign
18          Alloc
19            Writeable,
20            IntType 'int',
21            Name 'var',
22          Num '41',
23        If
24          Num '1',
25          [
26            Assign
27              Name 'var',
28              Num '42'
29          ]
30      ],
31    FunDef
32      IntType 'int',
33      Name 'fun2',
34      [],
35      [
36        Return
37          Num '1'
38      ]
39  ]
```

Code 3.64: *Abstrakter Syntaxbaum für 3 Funktionen.*

Im **PicoC-Blocks Pass** in Code 3.65 werden die **Anweisungen** von Funktionen in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Hierbei bekommt ein Block `Block(name, stmts_instrs)`, der die Anweisungen einer 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)`[50] beinhaltet den **Bezeichner** bzw. den `Name(str)`-Knoten der Funktion an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** `<number>` angehängt `<name>.<number>`[51] [52].

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>.<number>` ist dabei bis auf die angehängte **Nummer** `<number>` identisch zu dem der Funktion. Diese Zuordnung ist nötig, da **Blöcke** eine **Nummer** `<number>` an ihren Bezeichner angehängt haben `<name>.<number>`, die auf anderem Wege **nicht** ohne großen Aufwand herausgefunden werden kann.

```
1 File
2   Name './verbose_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
```

[50] Eine **Erklärung** dazu ist in Unterkapitel 3.3.1.2 zu finden.
[51] Der **Grund** dafür kann im Unterkapitel 3.3.1.3 nachgelesen werden.
[52] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.4',
11          [
12            Return(Empty())
13          ]
14      ],
15    FunDef
16      VoidType 'void',
17      Name 'fun1',
18      [],
19      [
20        Block
21          Name 'fun1.3',
22          [
23            Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
24            // If(Num('1'), []),
25            IfElse
26              Num '1',
27              [
28                GoTo
29                  Name 'if.2'
30              ],
31              [
32                GoTo
33                  Name 'if_else_after.1'
34              ]
35          ],
36        Block
37          Name 'if.2',
38          [
39            Assign(Name('var'), Num('42'))
40            GoTo(Name('if_else_after.1'))
41          ],
42        Block
43          Name 'if_else_after.1',
44          []
45      ],
46    FunDef
47      IntType 'int',
48      Name 'fun2',
49      [],
50      [
51        Block
52          Name 'fun2.0',
53          [
54            Return(Num('1'))
55          ]
56      ]
57  ]
```

Code 3.65: *PicoC-Blocks Pass für 3 Funktionen.*

Im **PicoC-ANF Pass** in Code 3.66 werden die `FunDef(datatype, name, allocs, stmts)`-Knoten komplett

aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Knoten nur noch **Blöcke** befinden.

```
File
  Name './verbose_3_funs.picoc_mon',
  [
    Block
      Name 'main.4',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.3',
      [
        // Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
        // Assign(Name('var'), Num('41'))
        Exp(Num('41'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        // If(Num('1'), [])
        // IfElse(Num('1'), [], [])
        Exp(Num('1')),
        IfElse
          Stack
            Num '1',
          [
            GoTo
              Name 'if.2'
          ],
          [
            GoTo
              Name 'if_else_after.1'
          ]
      ],
    Block
      Name 'if.2',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        Exp(GoTo(Name('if_else_after.1')))
      ],
    Block
      Name 'if_else_after.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 3.66: *PicoC-ANF Pass für 3 Funktionen.*

Nach dem **RETI Pass** in Code 3.67 gibt es nur noch **RETI-Befehle**, die Blöcke wurden entfernt. Die **RETI-Befehle** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man Folgen von RETI-Befehlen **nicht** mehr direkt Funktionen zuordnen. Die **Kommentare** enthalten die **Bezeichner** `<name>.<number>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer `<number>`, dem **Bezeichner** einer jeweiligen **Funktion** entsprechen und somit die Anfänge von **Funktionen erkennbar** machen.

Da es in der `main`-Funktion keinen **Funktionsaufruf** gab, wird der Code, der nach dem **Befehl** in der **markierten Zeile** in Code 3.67 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 **Adressen** gemacht werden. Die Sprünge werden zu den Adressen gemacht, wo die Folge von **RETI-Befehlen** anfängt, die aus den **Anweisungen** einer **Funktion** kompiliert wurden.

```
1  # // Block(Name('start.5'), [])
2  # // Exp(GoTo(Name('main.4')))
3  # // not included Exp(GoTo(Name('main.4')))
4  # // Block(Name('main.4'), [])
5  # Return(Empty())
6  LOADIN BAF PC -1;
7  # // Block(Name('fun1.3'), [])
8  # // Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
9  # // Assign(Name('var'), Num('41'))
10 # Exp(Num('41'))
11 SUBI SP 1;
12 LOADI ACC 41;
13 STOREIN SP ACC 1;
14 # Assign(Stackframe(Num('0')), Stack(Num('1')))
15 LOADIN SP ACC 1;
16 STOREIN BAF ACC -2;
17 ADDI SP 1;
18 # // If(Num('1'), [])
19 # // IfElse(Num('1'), [], [])
20 # Exp(Num('1'))
21 SUBI SP 1;
22 LOADI ACC 1;
23 STOREIN SP ACC 1;
24 # IfElse(Stack(Num('1')), [], [])
25 LOADIN SP ACC 1;
26 ADDI SP 1;
27 # JUMP== GoTo(Name('if_else_after.1'));
28 JUMP== 7;
29 # GoTo(Name('if.2'))
30 # // not included Exp(GoTo(Name('if.2')))
31 # // Block(Name('if.2'), [])
32 # // Assign(Name('var'), Num('42'))
33 # Exp(Num('42'))
34 SUBI SP 1;
35 LOADI ACC 42;
36 STOREIN SP ACC 1;
37 # Assign(Stackframe(Num('0')), Stack(Num('1')))
38 LOADIN SP ACC 1;
39 STOREIN BAF ACC -2;
40 ADDI SP 1;
41 # Exp(GoTo(Name('if_else_after.1')))
42 # // not included Exp(GoTo(Name('if_else_after.1')))
43 # // Block(Name('if_else_after.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 3.67: *RETI-Blocks Pass für 3 Funktionen.*

3.3.6.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 3.63 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** eines 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 3.68 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 3.68: *PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI-Blocks Pass** in Code 3.69 sind die **Funktionen** durch **Blöcke** umgesetzt.

```
File
  Name './verbose_3_funs_main.reti_blocks',
  [
    Block
      Name 'fun1.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    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 3.69: *RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Eine simple Möglichkeit die Ausführung durch die `main`-Funktion zu starten, 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 den ersten Index 0 geschoben werden.

Die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** allerdings entschieden wurde, war es, wenn die `main`-Funktion **nicht** die **erste** Funktion im `decls_defs`-Attribut des `File(name, decls_defs)`-Knoten ist, einen `start.<number>`-Block als ersten Block einzufügen. Dieser `start.<number>`-Block enthält einen `GoTo(Name('main.<number>'))`-Knoten, der im **RETI Pass** in Code 3.71 in einen Sprung zum **Block** der `main`-Funktion übersetzt wird.[53]

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für Verwender[54] des **PicoC-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 **Einfügen** des `start.<number>`-Blockes erfolgt im **RETI-Patch Pass** in Code 3.70. Der **RETI-Patch** Pass ist der Pass, der für das **Ausbessern**[55] des **Abstrakten Syntaxbaumes** zuständig ist, wenn z.B. wie hier 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',
6       [
7         # // Exp(GoTo(Name('main.0')))
```

[53] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.
[54] Also die kommenden **Studentengenerationen**.
[55] In engl. to patch.

```
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 3.70: *RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI Pass** in Code 3.71 werden die Knoten `Exp(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 3.71: *RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

3.3.6.2 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 bei der in Code 3.72 **markierten Zeile** `1` oder **indirekt** zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** `3-4`.

In dem Beispiel in Code 3.72 erfolgt in Zeile `7` ein **Funktionsaufruf** der Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdekleration**, wie in der **markierten Zeile** `1` notwendig. Beim **Funktionsaufruf** der 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 3.72: *PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

Die **Deklaration** einer **Funktion** erfolgt mithilfe einer **Symboltabelle** (Definition 3.8), die in Code 3.73 für das Beispiel in Code 3.72 dargestellt ist. Die Symboltabelle ist als **Assoziatives Feld** umgesetzt, indem über einen **Schlüssel** ein **Symboltabelleneintrag** mit verschiedenen **Attributen**, welche bereits in Tabelle 3.9 erklärt wurden für eine Variable, eine Konstante, eine Funktion, einen Datentyp usw. nachgeschlagen werden kann. Für z.B. die Funktion `int fun2(int var)` werden die **Attribute** des **Symboltabelleneintrags** `Symbol(type_qual, datatype, name, val_addr, pos, size)` wie üblich gesetzt. Dem `datatype`-Attribut werden dabei einfach die Koten des kompletten **Funktionsprototypes** `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` in Code 3.73 haben unterschiedliche **Sichtbarkeitsbereiche** (Definition 1.9), nämlich den der jeweiligen Funktion, also `main` und `fun2`. Die **Sichtbarkeitsbereiche** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` angehängt wird: `var@<fun_name>`. Dieser **Bezeichner** mit **angehängten Suffix** `<var_name>@<scope>` wird als **Schlüssel** zum Nachschlagen von Symboltabelleneinträgen in der Symboltabelle verwendet. Im Fall von Funktionen handelt es sich bei diesem **Suffix** `@<scope>` um den **Bezeichner** der jeweiligen Funktion, der aus dem `name`-Attribut des `FunDef(datatype, name, params, stmts_blocks)`-Knotens entnommen wird.

Dieser **Suffix** wird geändert, sobald beim **Top-Down**[56]-Iterieren über den **Abstrakten Syntaxbaum** ein neuer `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten betreten wird und über dessen Anweisungsknoten im `stmts_blocks`-Attribut iteriert wird. Die **Schlüssel** der Symboltabelle sind **identisch** zur Zeichenkette, die im `Name(str)`-Knoten des `name`-Attributes im Symboltabelleneintrag `Symbol(type_qual, datatype, name, val_addr, pos, size)` gespeichert ist. Dadurch lässt sich aus dem **Symboltabelleneintrag** einer **Variable** direkt ihr **Sichtbarkeitsbereich**, in dem sie definiert wurde ablesen. Dies ist in Code 3.73 **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 **Symboltabelleneintrags** für eine Variable, an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird und nicht nur beim Definieren der Variable.

Ein Grund, warum **Sichtbarkeitsbereiche** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel** der **Symboltabelle** eindeutig sind, denn z.B. bei Funktionen ist es **nicht** möglich zwei Funktionen mit dem **gleichen Bezeichner** zu deklarieren / definieren[57]. Folglich kann eine Variable nur in genau der Funktion **nachgeschlagen**, in der sie **definiert** wurde. Das Symbol `'@'` wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel 3.3.1.3 erläutert wurde.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
        ↪  IntType('int'), Name('var'))])
        name:                   Name('fun2')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('4'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
        name:                   Name('fun1')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('6'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
```

[56] D.h. von der **Wurzel** zu den **Blättern** eines Baumes.
[57] Sonst gibt es eine **Fehlermeldung**, wie `ReDeclarationOrDefinition`.

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

3.3.6.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(1+1)`) wird im Folgenden mithilfe des Beispiels in Code 3.74 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und das Beispiel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 3.3.6.3.1 erklärt. Zudem wurde, um die **Adressberechnung anschaulicher** zu machen als **Datentyp** für die beiden **lokalen Variablen** `local_var` in beiden Funktion `main` und `stack_fun` ein **Verbund** gewählt, der **mehrere Speicherzellen** im Hauptspeicher einnimmt.

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

Code 3.74: *PicoC-Code für Funktionsaufruf ohne Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 3.75 wird ein **Funktionsaufruf** `stack_fun(1+1)` durch die **Knoten** `Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))` dargestellt.

Der **Parameter** `param` der Funktion `stack_fun` wird, wie in Code 3.75 markiert ist mithilfe der Knoten `Alloc(Writeable(), IntType('int'), Name('param'))` dargestellt, die normalerweise eine **Allokation** darstellen, wie z.B. bei der Lokalen Variable `local_var`: `Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))), Name('local_var')))`.

Das hat den Grund, dass **Parameter** einer Funktion auf diese Weise als `Exp(Alloc(type_qual, datatype, name))` in der **Symboltabelle** definiert werden können. Hierzu werden die **Parameter**, die als `Alloc(type_qual, datatype, name)`-Knoten dargestellt sind `exp`-Attributen der `Exp(exp)`-Knoten zugewiesen. Die durch die `Exp(Alloc(type_qual, datatype, name))`-Knoten dargestellten Allokationsanweisungen werden im **PicoC-ANF Pass** dann als **erstes** im `stmts_blocks`-Attribut des `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten verarbeitet.

```
File
  Name './example_fun_call_no_return_value.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
      ],
    FunDecl
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc
          Writeable,
          IntType 'int',
          Name 'param'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc(Writeable(), IntType('int'), Name('param'))
      ],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
      ]
  ]
```

Code 3.75: *Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.*

Alle Funktionen **außer** der `main`-Funktion besitzen einen **Stackframe** (Definition 3.10). Bei der `main`-Funktion werden **Lokale Variablen** einfach zu den **Globalen Statischen Daten** geschrieben.

In Tabelle 3.18 ist für das Beispiel in Code 3.74 das **Datensegment** inklusive **Stackframe** der Funktion `stack_fun` mit allen **allokierten Variablen** dargestellt. Mithilfe der Spalte **Relativadresse** in der Tabelle 3.18 erklären sich auch die **Relativadressen** der Variablen `local_var@main`, `local_var@stack_fun`, `param@stack_fun` in den `value or address`-Attributen der markierten **Symboltabelleneinträge** in der **Symboltabelle** in

Code 3.76. Bei **Stackframes** fangen die **Relativadressen** erst 2 Speicherzellen relativ zum `BAF`-Register an, da die **Rücksprungadresse** und die **Startadresse des Vorgängerframes** Platz brauchen.

Dadurch, dass die Allokationsanweisungen `Exp(Alloc(type_qual, datatype, name))` der **Parameter** wie vorhin erwähnt als erstes **vor** allen anderen Anweisungen einer Funktion ausgeführt werden, wird z.B. der **Parameter** `param` der Funktion `stack_fun` als **erstes** auf dem dem **Stackframe** in Tabelle 3.18 allokiert und erst danach folgt die **Lokale Variable** `local_var`[58].

Relativ-adresse	Inhalt	Register
0	⟨*local_var@main*⟩	CS
1		
2		
3		
...	...	
...	...	SP
4	⟨*local_var@stack_fun*⟩	
3		
2		
1		
0	⟨*param@stack_fun*⟩	
...	Rücksprungadresse	
...	Startadresse Vorgängerframe	BAF

Tabelle 3.18: *Datensegment mit Stackframe.*

Definition 3.10: Stackframe

*Eine **Datenstruktur**, die dazu dient während der **Laufzeit** eines Programmes den **Zustand** einer Funktion **„konservieren“** zu können, um die **Ausführung** dieser Funktion später im **selben Zustand fortsetzen** zu können. **Stackframes** werden dabei in einem Stack **übereinander gestappelt** und in die **entgegengesetzte Richtung** wieder abgebaut, wenn sie nicht mehr benötigt werden. Der **Aufbau** eines **Stackframes** ist in Tabelle 3.19 dargestellt.*[a]

...	← SP
Temporäre Berechnungen	
Lokale Variablen	
Parameter	
Rücksprungadresse	
Startadresse Vorgängerframe	← BAF

Tabelle 3.19: *Aufbau Stackframe*

*Üblicherweise steht als **erstes**[b] in einem Stackframe die **Startadresse** des **Vorgängerframes**. Diese ist notwendig, damit beim **Rücksprung** aus einer **aufgerufenen Funktion**, zurück zur **aufrufenden Funktion** das `BAF`-Register wieder so gesetzt werden kann, dass es auf den **Stackframe** der **aufrufenden Funktion** zeigt.*

*Als **zweites** steht in einem Stackframe üblicherweise die **Rücksprungadresse**. Die*

[58] Der **Stack** wächst in Tabelle 3.18 von **unten-nach-oben**, daher wird ein **Stackframe** ebenfalls von **unten-nach-oben** gelesen.

Rücksprungadresse *ist die Adresse* ***relativ*** *zum Anfang des* ***Codesegments****, an welcher die* ***Ausführung*** *der* ***aufrufenden Funktion*** *nach einem Funktionsaufruf* ***fortgesetzt*** *wird.*

Die ***Startadresse*** *des* ***Vorgängerframes*** *und die* ***Rücksprungadresse*** *stehen beide im Stackframe der* ***aufgerufenen*** *Funktion, weil eine Funktion von allen möglichen Funktionen aufgerufen werden kann und diese beiden Informationen der* ***aufrufenden*** *Funktion für den* ***Rücksprung*** *im Stackframe der* ***aufgerufenen*** *Funktion benötigt werden, da hier auf die* ***einfachste Weise*** *darauf zugegriffen werden kann. Alles weitere in Tabelle 3.19 ist selbsterklärend.*[c]

[a] Wenn von **„auf den Stack schreiben“** gesprochen wird, dann wird damit gemeint, dass der Bereich für **Temporäre Berechnungen** (nach Tabelle 3.19) **vergrößert** wird und ein **Wert** hinein **geschrieben** wird.
[b] Die Tabelle 3.19 ist von **unten-nach-oben** zu lesen, da im **PicoC-Compiler** Stackframes in einem **Stack** untergebracht werden, der von **unten-nach-oben** wächst. Alles soll **konsistent** dazu gehalten werden, wie es im **PicoC-Compiler** umgesetzt ist.
[c] C. Scholl, „Betriebssysteme“.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('attr@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(),
        ↪  ArrayDecl([Num('2')], IntType('int')), Name('attr'))])
        name:                   Name('st')
        value or address:       [Name('attr@st')]
        position:               Pos(Num('1'), Num('7'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('stack_fun'),
        ↪  [Alloc(Writeable(), IntType('int'), Name('param'))])
        name:                   Name('stack_fun')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('5'), Num('5'))
        size:                   Empty()
      },
```

```
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               ArrayDecl([Num('2')], StructSpec(Name('st')))
43        name:                   Name('local_var@main')
44        value or address:       Num('0')
45        position:               Pos(Num('6'), Num('12'))
46        size:                   Num('4')
47      },
48    Symbol
49      {
50        type qualifier:         Writeable()
51        datatype:               IntType('int')
52        name:                   Name('param@stack_fun')
53        value or address:       Num('0')
54        position:               Pos(Num('11'), Num('19'))
55        size:                   Num('1')
56      },
57    Symbol
58      {
59        type qualifier:         Writeable()
60        datatype:               ArrayDecl([Num('2')], StructSpec(Name('st')))
61        name:                   Name('local_var@stack_fun')
62        value or address:       Num('4')
63        position:               Pos(Num('12'), Num('12'))
64        size:                   Num('4')
65      }
66  ]
```

Code 3.76: *Symboltabelle für Funktionsaufruf ohne Rückgabewert.*

Im **PicoC-ANF Pass** in Code 3.77 werden die Knoten `Exp(Call(Name('stack_fun'),[Name('local_var')]))` durch die Knoten `StackMalloc(Num('2'))`, `NewStackframe(Name('stack_fun'),GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` usw. ersetzt. Die **Bedeutung** all dieser **Knoten** und **Kompositionen** von Knoten wird in Tabelle 3.20 erläutert.

Knoten	Beschreibung
`StackMalloc(num)`	Steht für das **Allokieren** von `num` Speicherzellen auf dem **Stack**.
`NewStackframe(fun_name, goto_after_call)`	**Erstellt** einen neuen **Stackframe** und speichert den Wert des `BAF`-Registers der **aufrufenden Funktion** und die **Rücksprungadresse** nacheinander an den **Anfang** des neuen **Stackframes**. Das Attribut `fun_name` stehte dabei für den Bezeichner der Funktion, für die ein neuer **Stackframe** erstellt werden soll. Das Attribut `fun_name` dient später dazu den **Block** dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attributen gespeichert hat. Des Weiteren enthält das Attribut `goto_after_call` ein `GoTo(Name('addr@next_instr'))`, welches später durch die **Adresse** des Befehls, der direkt auf den **Sprungbefehl** folgt, ersetzt wird.
`Exp(GoTo(name))`	Sprung zu einem anderen **Block** durch Angabe des **Bezeichners** des Blocks, wobei das Attribut `name` der **Bezeichner** des Blocks ist, zu dem Gesprungen werden soll.
`Instr(Loadi(), [Reg(reg), GoTo(Name('addr@next_instr'))])`	Lädt in das `reg`-Register die **Adresse** des Befehls, der direkt **auf** dem nächsten **Sprung** zum Block einer anderen Funktion folgt. Die Adresse ist dabei **relativ** zum **Anfang** des **Codesegments**.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes**, durch das **Wiederherstellen** des im **Stackframe** der **aufgerufenen Funktion** gespeicherten Werts des `BAF`-Registes der **aufrufenden Funktion** und das Setzen des `SP`-Registers auf den Wert des `BAF`-Registers der **aufgerufenen Funktion**.

Tabelle 3.20: *Knoten für Funktionsaufruf.*

Die Knoten `StackMalloc(Num('2'))` sind notwendig, weil auf dem **Stackframe** für den Wert des `BAF`-Registers der **aufrufenden Funktion** und die **Rücksprungadresse** am **Anfang** des **Stackframes** 2 Speicherzellen Platz gelassen werden müssen. 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** allokiert wird.

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

```
17      Name 'stack_fun.0',
18      [
19        Return(Empty())
20      ]
21  ]
```

Code 3.77: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 3.78 werden die **PicoC-Knoten** `StackMalloc(Num('2'))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch **semantisch** entsprechende **Knoten** ersetzt.

Die Knoten `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` in Zeile `29` und `33` in Code 3.78 bestehen noch nicht ausschließlich aus **RETI-Knoten** und werden erst später in dem für sie vorgesehenen **RETI-Pass** in Code 3.79 passend ersetzt. Der **Bezeichner** `stack_fun.0` für `NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))` und `Exp(GoTo(Name('stack_fun.0')))`, wird mithilfe des **Assoziativen Feldes** `fun_name_to_block_name`[59] und dem **Schlüssel** `stack_fun`[60] aus `Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))` nachgeschlagen.[61]

```
1  File
2    Name './example_fun_call_no_return_value.reti_blocks',
3    [
4      Block
5        Name 'main.1',
6        [
7          # StackMalloc(Num('2'))
8          SUBI SP 2;
9          # Exp(Num('1'))
10         SUBI SP 1;
11         LOADI ACC 1;
12         STOREIN SP ACC 1;
13         # Exp(Num('1'))
14         SUBI SP 1;
15         LOADI ACC 1;
16         STOREIN SP ACC 1;
17         # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
18         LOADIN SP ACC 2;
19         LOADIN SP IN2 1;
20         ADD ACC IN2;
21         STOREIN SP ACC 2;
22         ADDI SP 1;
23         # NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))
24         MOVE BAF ACC;
25         ADDI SP 3;
26         MOVE SP BAF;
27         SUBI SP 7;
28         STOREIN BAF ACC 0;
29         LOADI ACC GoTo(Name('addr@next_instr'));
30         ADD ACC CS;
```

[59] Welches in Unterkapitel 3.3.6.1 eingeführt wurde.
[60] Dem **Bezeichner** der **Funktion** zu der gesprungen werden soll.
[61] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
31          STOREIN BAF ACC -1;
32          # Exp(GoTo(Name('stack_fun.0')))
33          Exp(GoTo(Name('stack_fun.0')))
34          # RemoveStackframe()
35          MOVE BAF IN1;
36          LOADIN IN1 BAF 0;
37          MOVE IN1 SP;
38          # Return(Empty())
39          LOADIN BAF PC -1;
40        ],
41      Block
42        Name 'stack_fun.0',
43        [
44          # Return(Empty())
45          LOADIN BAF PC -1;
46        ]
47    ]
```

Code 3.78: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 3.79 wird nun der finale **RETI-Code** generiert. Die **RETI-Befehle** aus den **Blöcken** sind nun zusammengefügt und es gibt keine **Blöcke** mehr. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in `LOADI ACC GoTo(Name('addr@next_instr'))` durch die **Adresse** des nächsten Befehls direkt nach dem Befehl `JUMP 5`[62] [63] ersetzt: `LOADI ACC 21`. Die Knoten, die den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellen, werden durch den Befehl `JUMP 5` ersetzt, der die **Distanz** als die **Anzahl Speicherzellen** `5` bis zum **ersten** Befehl der Funktion `stack_fun` enthält.

Die **Distanz** `5` im **RETI-Knoten** `JUMP 5` wird mithilfe des versteckten `instrs_before`-Attributs des **Zielblocks** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`[64] und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` selbst liegt berechnet.

Die **relative Adresse** `21` des Befehls `LOADI ACC 21` wird ebenfalls mithilfe des versteckten `instrs_before`-Attributes des **aktuellen Blocks** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)` berechnet. Es handelt sich bei `21` um eine **relative Adresse**, die **relativ** zum `CS`-Register[65] [66] [67] berechnet wird.

Anmerkung

Die Berechnung der **Adresse** adr_{danach} bzw. `<addr@next_instr>` des Befehls nach dem nächsten **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt mithilfe der folgenden Formel 3.3.1:

$$adr_{danach} = \#Bef_{vor\ akt.\ Bl.} + idx + 4 \tag{3.3.1}$$

wobei:

- es sich bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum **Anfang** des **Codesegments** berechnet wird.

[62] Der für den **Sprung zur gewünschten Funktion** verantwortlich ist.
[63] Also der Befehl, der bisher durch die Knoten `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.
[64] Welcher den **ersten Befehl** der gewünschten Funktion enthält.
[65] Also relativ zum **Anfang** des **Codesegments**.
[66] Welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.
[67] Wobei man bei den **Adressen** bei 0 anfängt zu zählen.

- $\#Bef_{vor\ akt.\ Bl.} \hat{=}$ **Anzahl** Befehle vor dem aktuellen 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. Das liegt darin begründet, dass 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 muss daher im letzten Schritt dieses **Pass** berechnet werden.
- $idx \hat{=}$ **relativer Index** des Befehls `LOADI ACC <addr@next_instr>`, für den die **Adresse** adr_{danach} bzw. `<addr@next_instr>` gerade berechnet werden soll im **aktuellen Block**.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 3.79 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` gesprungen werden muss und noch **eins** mehr, weil man ja zum Befehl nach dem Sprung `JUMP <im>` springen will.

Die Berechnug der **Sprungdistanz**[a] $Dist_{Zielbl.}$ bzw. `<distance>` zum **ersten** Befehl eines im vorhergehenden **Pass existenten Blockes**[b] für den Sprungbefehl `JUMP <distance>` erfolgt nach der folgenden Formel 3.3.2:

$$Dist_{Zielbl.} = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases} \tag{3.3.2}$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock** zu dem gesprungen werden soll. 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)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung**, wie in der Formel 3.3.1.
- $idx \hat{=}$ **relativer Index** des Befehls `JUMP <distance>`, für den gerade die **Sprungdistanz** $Dist_{Zielbl.}$ bzw. `<distance>` gerade berechnet werden soll im **aktuellen Block**.

In Abbildung 3.9 sind alle 3 möglichen **Konstellationen** $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$, $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$ und $\#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.}$ veranschaulicht, welche durch die Formel 3.3.2 abgedeckt sind[c].

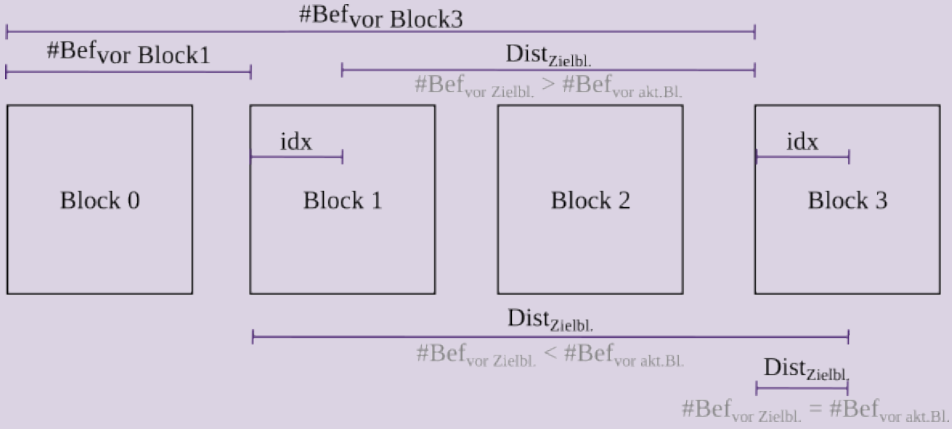


Abbildung 3.9: *Veranschaulichung der Dinstanzberechnung*

[a] Diese **Distanz** kann auch **negativ** werden.

[b] Im **RETI-Pass** gibt es **keine** Blöcke mehr.
[c] Die **Konstellationen** $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$, $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$ sind im Fall $\#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.}$ zusammengefasst, da sie auf die **gleiche Weise** berechnet werden.

```
1  # // Exp(GoTo(Name('main.1')))
2  # // not included Exp(GoTo(Name('main.1')))
3  # StackMalloc(Num('2'))
4  SUBI SP 2;
5  # Exp(Num('1'))
6  SUBI SP 1;
7  LOADI ACC 1;
8  STOREIN SP ACC 1;
9  # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
14 LOADIN SP ACC 2;
15 LOADIN SP IN2 1;
16 ADD ACC IN2;
17 STOREIN SP ACC 2;
18 ADDI SP 1;
19 # NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))
20 MOVE BAF ACC;
21 ADDI SP 3;
22 MOVE SP BAF;
23 SUBI SP 7;
24 STOREIN BAF ACC 0;
25 LOADI ACC 21;
26 ADD ACC CS;
27 STOREIN BAF ACC -1;
28 # Exp(GoTo(Name('stack_fun.0')))
29 JUMP 5;
30 # RemoveStackframe()
31 MOVE BAF IN1;
32 LOADIN IN1 BAF 0;
33 MOVE IN1 SP;
34 # Return(Empty())
35 LOADIN BAF PC -1;
36 # Return(Empty())
37 LOADIN BAF PC -1;
```

Code 3.79: *RETI-Pass für Funktionsaufruf ohne Rückgabewert.*

3.3.6.3.1 Rückgabewert

Die Umsetzung eines **Funktionsaufrufs inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 3.80 erklärt.

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

```
int fun_with_return_value() {
  return 21 * 2;
}

void fun_no_return_value() {
  return;
}

void main() {
  int var = fun_with_return_value();
  fun_no_return_value();
}
```

Code 3.80: *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 3.81 wird eine **Return-Anweisung mit Rückgabewert** `return 21 * 2` mit den Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, eine **Return-Anweisung ohne Rückgabewert** `return` mit den Knoten `Return(Empty())` und ein **Funktionsaufruf** inklusive **Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` mit den Knoten `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))`.

```
File
  Name './example_fun_call_with_return_value.ast',
  [
    FunDef
      IntType 'int',
      Name 'fun_with_return_value',
      [],
      [
        Return(BinOp(Num('21'), Mul('*'), Num('2')))
      ],
    FunDef
      VoidType 'void',
      Name 'fun_no_return_value',
      [],
      [
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')),
        ↪  Call(Name('fun_with_return_value'), []))
        Exp(Call(Name('fun_no_return_value'), []))
      ]
  ]
```

Code 3.81: *Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.*

Im **PicoC-ANF Pass** in Code 3.82 werden bei den Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die Knoten `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierfür erstellten Knoten

`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, welches den **Rückgabewert** darstellt, wird dann von den **Knoten** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register** `ACC` geschrieben. Des Weiteren wird vom `Return(Stack(Num('1')))`-Knoten die **Rücksprungadresse**, die im **Stackframe** der **aufgerufenen Funktion** gespeichert ist in das `PC`-Register geladen[68], um wieder zur **aufrufenden Funktion** zurückzuspringen.

Ein wichtiges Detail bei der **Funktion** `int fun_with_return_value() { return 21*2; }` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird[69], da diese **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Bei dieser Übersetzung wird durch die Knoten `Exp(ACC)` der **Rückgabewert** der **aufgerufenen Funktion** für die **aufrufende Funktion**, deren **Stackframe** nun wieder der aktuelle ist vom `ACC`-Register auf den **Stack** geschrieben. Der **Rückgabewert** wurde zuvor in der **aufgerufenen Funktion** durch die Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das `ACC`-Register geschrieben.

Dieser Trick mit dem Speichern des **Rückgabewerts** im `ACC`-Register ist notwendidg, da der Rückgabewert **nicht** einfach auf den **Stack** gespeichert werden kann. Nach dem **Entfernen** des **Stackframes** der **aufgerufenen Funktion** zeigt das `SP`-Register nicht mehr an die gleiche Stelle. Daher sind alle **temporären** Werte, die in der **aufgerufenen Funktion** auf den **Stack** geschrieben wurden unzugänglich. Man kann nicht direkt ohne weiteres berechnen, um wieviel die Adresse im `SP`-Register verglichen zu vorher verschoben ist, weil der Speicherplatz, den Parameter und Lokale Variablen im **Stackframe** einnehmen bei unterschiedlichen **aufgerufenen** Funktionen **unterschiedlich groß** sein kann.

Die **Knoten** `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))` vereinen **mehrere Aufgaben**. Mittels `Alloc(Writeable(), IntType('int'), Name('var'))` wird die Variable `Name('var')` **allokiert**. Die Knoten `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))` werden durch die Knoten `Assign(Global(Num('0')),Stack(Num('1')))` ersetzt, welche den **Rückgabewert** der Funktion `'fun_with_return_value'` nun vom **Stack** in die Speicherzelle der Variable `Name('var')` in den **Globalen Statischen Daten** speichern. Hierzu muss die **Adresse** der gerade allokierten Variable `Name('var')` in der **Symboltabelle** nachgeschlagen werden. Der **Rückgabewert** der Funktion `'fun_with_return_value'` wurde zuvor durch die **Knoten** `Exp(Acc)` aus dem `ACC`-Register auf den **Stack** geschrieben.[70]

Der Umgang mit einer **Funktion ohne Rückgabewert** wurde am Anfang dieses Unterkapitels 3.3.6.3 bereits besprochen. Für ein `return` **ohne Rückgabewert** bleiben die **Knoten** `Return(Empty())` in diesem Pass unverändert, sie stellen nur das Laden der **Rücksprungsadresse** in das `PC`-Register dar.

Des Weiteren kann anhand der `main`-Funktion beobachtet werden, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` keine `return`-Anweisung explizit ans Ende geschrieben wird, im **PicoC-ANF Pass** eine in Form der Knoten `Return(Empty())` hinzufügt wird. Bei **Nicht-Angeben** wird im Falle eines Rückgabedatentyps, der **nicht** `void` ist allerdings eine `MissingReturn`-**Fehlermeldung** ausgelöst.

```
1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
```

[68] Die **Rücksprungadresse** wurde zuvor durch den `NewStackframe()`-Knoten eine Speicherzelle **vor** der Speicherzelle auf die das `BAF`-Register zeigt im **Stackframe** gespeichert (siehe Unterkapitel 3.3.6.3 für Zusammenhang).
[69] Als in Unterkapitel 3.3.6.3 **bisher** erklärt wurde.
[70] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

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

Code 3.82: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 3.83 werden die **PicoC-Knoten** `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Mul('*'),Stack(Num('1'))))`, `Return(Stack(Num('1')))`, `Return(Empty())`, `Exp(ACC)` und `Assign(Global(Num('0')),Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         # Exp(Num('21'))
9         SUBI SP 1;
10        LOADI ACC 21;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
```

```
19        MULT ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Return(Stack(Num('1')))
23        LOADIN SP ACC 1;
24        ADDI SP 1;
25        LOADIN BAF PC -1;
26      ],
27    Block
28      Name 'fun_no_return_value.1',
29      [
30        # Return(Empty())
31        LOADIN BAF PC -1;
32      ],
33    Block
34      Name 'main.0',
35      [
36        # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
37        # StackMalloc(Num('2'))
38        SUBI SP 2;
39        # NewStackframe(Name('fun_with_return_value.2'), GoTo(Name('addr@next_instr')))
40        MOVE BAF ACC;
41        ADDI SP 2;
42        MOVE SP BAF;
43        SUBI SP 2;
44        STOREIN BAF ACC 0;
45        LOADI ACC GoTo(Name('addr@next_instr'));
46        ADD ACC CS;
47        STOREIN BAF ACC -1;
48        # Exp(GoTo(Name('fun_with_return_value.2')))
49        Exp(GoTo(Name('fun_with_return_value.2')))
50        # RemoveStackframe()
51        MOVE BAF IN1;
52        LOADIN IN1 BAF 0;
53        MOVE IN1 SP;
54        # Exp(ACC)
55        SUBI SP 1;
56        STOREIN SP ACC 1;
57        # Assign(Global(Num('0')), Stack(Num('1')))
58        LOADIN SP ACC 1;
59        STOREIN DS ACC 0;
60        ADDI SP 1;
61        # StackMalloc(Num('2'))
62        SUBI SP 2;
63        # NewStackframe(Name('fun_no_return_value.1'), 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 3.83: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.*

3.3.6.3.2 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 3.84 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 3.84: *PicoC-Code für die Übergabe eines Feldes.*

In der **Symboltabelle** in Code 3.85 kann die **Umwandlung** eines **Felds** zu einem **Zeiger** 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 werden sie an **Parameter** `'param@fun_array_from_global_data'` und `'param@fun_array_from_stackframe'` mit dem Datentyp `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))` gebunden. Die **Größe** dieser **Parameter** beträgt dabei, wie in der Symboltabelle in Code 3.85 gesehen werden kann `Num('1')`, da ein **Zeiger** nur **eine** Speicherzelle einnimmt.

Wie in Unterkapitel 3.3.6.3 erklärt wurde, werden **Parameter** zu Allokationsanweisungen `Exp(Alloc(type_qual, datatype, name))` umgewandelt und als **erstes** im `stmts_blocks`-Attribut des `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten verarbeitet. Bei der gerade erwähnte **Änderung** des **Datentyps** von einem **Feld-Datentyp** zu einem **Zeiger-Datentyp**, soll nur bei Allokationsanweisungen von **Parametern** der Feld-Datentyp zu einem Zeiger-Datentyp umgewandelt werden. Bei Allokationsanweisungen von **Lokalen Variablen** soll **keine Umwandlung** des Datentyps stattfinden. Um **unterscheiden** zu können, ob es sich bei der Allokationsanweisung um die Allokation eines **Parameters** oder eine **Lokalen Variablen** handelt, wird das versteckte Attribut `local_var_or_param`-Attribut des `Alloc(type qual, datatype, name, local_var_or_param)`-Knoten zu Beginn des **PicoC-ANF Pass** mit z.B. einem entsprechenden `Name('param')`-Knoten belegt, sodass in diesem Fall klar ist, dass es sich um einen **Parameter** handelt.

Um die Änderung des Datentyps von einem **Feld-Datentyp** zu einem **Zeiger-Datentyp** umzusetzen, wird bei der **Erstellung** eines **Symboltabelleneintrags** für einen **Feld-Datentyp**, vor dem Zuweisen an das

`datatype`-Attribut eines Symboltabelleneintrags, der **oberste Knoten** des Teilbaums, der den **Feld-Datentyp** `ArrayDecl(nums, datatype)` darstellt zu einem **Zeiger-Datentyp** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut des `ArrayDecl(nums, datatype)`-Knoten hängt, wird an das `datatype`-Attribut des **Zeigers-Datentypes** `PntrDecl(num, datatype)` gehängt. Bei einem **Mehrdimensionalen Feld**, fällt eine **Dimension** an den Zeiger **weg** und der Rest des **Feld-Datentypes** `ArrayDecl(nums, datatype)` wird an das `datatype`-Attribut des **Zeiger-Datentyps** `PntrDecl(num, datatype)` gehängt.

```
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')
        position:               Pos(Num('5'), Num('6'))
        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 3.85: *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 3.86 ist zu sehen, dass zur Übergabe der beiden Felder `local_var@main` und `local_var@fun_array_from_global_data` die **Adressen** der Felder mithilfe der Knoten `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` auf den **Stack** geschrieben werden. Die Knoten `Ref(Global(Num('0')))` sind für die **Variable** `local_var` aus der `main`-Funktion, da diese in den **Globalen Statischen Daten** liegt und die Knoten `Ref(Stackframe(Num('6')))` sind für die Variable `local_var` aus der Funktion `fun_array_from_global_data`, da diese auf dem **Stackframe** dieser Funktion liegt.

Die Knoten `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` werden später im **RETI-Blocks Pass** in Code 3.87 durch unterschiedliche **RETI-Befehle** ersetzt. Hierbei stellen die Zahlen `'0'` bzw. `'6'` in den Knoten `Global(num)` bzw. `Stackframe(num)`, die aus der **Symboltabelle** in Code 3.85 entnommen sind **relative Adressen** relativ zum `DS`-Register bzw. `SP`-Register dar. Die Zahl `'6'` ergibt sich dadurch, dass das Feld `local_var` der Funktion `fun_array_from_global_data` die **Dimensionen** 2×3 hat und ein Feld von **Integern** ist, also $size(type(\texttt{local\_var})) = \left(\prod_{j=1}^{n} \texttt{dim}_\texttt{j}\right) \cdot size(int) = 2 \cdot 3 \cdot 1 = 6$ *Speicherzellen*.

```
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.2'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_array_from_stackframe.2')))
```

```
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Ref(Global(Num('0')))
24        NewStackframe(Name('fun_array_from_global_data.1'), GoTo(Name('addr@next_instr')))
25        Exp(GoTo(Name('fun_array_from_global_data.1')))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]
```

Code 3.86: *PicoC-ANF Pass für die Übergabe eines Feldes.*

Im **RETI-Blocks Pass** in Code 3.87 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'fun_array_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun_array_from_global_data.1',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Ref(Stackframe(Num('6')))
16        SUBI SP 1;
17        MOVE BAF IN1;
18        SUBI IN1 8;
19        STOREIN SP IN1 1;
20        # NewStackframe(Name('fun_array_from_stackframe.2'), GoTo(Name('addr@next_instr')))
21        MOVE BAF ACC;
22        ADDI SP 3;
23        MOVE SP BAF;
24        SUBI SP 3;
25        STOREIN BAF ACC 0;
26        LOADI ACC GoTo(Name('addr@next_instr'));
27        ADD ACC CS;
28        STOREIN BAF ACC -1;
29        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
30        Exp(GoTo(Name('fun_array_from_stackframe.2')))
31        # RemoveStackframe()
32        MOVE BAF IN1;
33        LOADIN IN1 BAF 0;
34        MOVE IN1 SP;
```

```
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ],
38    Block
39      Name 'main.0',
40      [
41        # StackMalloc(Num('2'))
42        SUBI SP 2;
43        # Ref(Global(Num('0')))
44        SUBI SP 1;
45        LOADI IN1 0;
46        ADD IN1 DS;
47        STOREIN SP IN1 1;
48        # NewStackframe(Name('fun_array_from_global_data.1'), GoTo(Name('addr@next_instr')))
49        MOVE BAF ACC;
50        ADDI SP 3;
51        MOVE SP BAF;
52        SUBI SP 9;
53        STOREIN BAF ACC 0;
54        LOADI ACC GoTo(Name('addr@next_instr'));
55        ADD ACC CS;
56        STOREIN BAF ACC -1;
57        # Exp(GoTo(Name('fun_array_from_global_data.1')))
58        Exp(GoTo(Name('fun_array_from_global_data.1')))
59        # RemoveStackframe()
60        MOVE BAF IN1;
61        LOADIN IN1 BAF 0;
62        MOVE IN1 SP;
63        # Return(Empty())
64        LOADIN BAF PC -1;
65      ]
66  ]
```

Code 3.87: *RETI-Block Pass für die Übergabe eines Feldes.*

3.3.6.3.3 Umsetzung der 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 3.88 erklärt.

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

```
13   struct st local_var;
14   fun_struct_from_global_data(local_var);
15 }
```

Code 3.88: *PicoC-Code für die Übergabe eines Verbundes.*

Im **PicoC-ANF Pass** in Code 3.89 werden zur **Übergabe der beiden Verbunde** `local_var@main` und `param@fun_array_from_global_data`, diese mittels der Knoten `Assign(Stack(Num('3')), Global(Num('0')))` bzw. `Assign(Stack(Num('3')), Stackframe(Num('2')))` jeweils auf den **Stack** kopiert.

Bei der **Übergabe** an eine **Funktion** wird der Zugriff auf einen gesamten **Verbund** anders gehandhabt als bei einem **Feld**[71]. Beim einem **Feld** wurde bei der **Übergabe an eine Funktion** die **Adresse** des **ersten Feldelements** auf den **Stack** geschrieben. Bei einem **Verbund** wird bei der **Übergabe an eine Funktion** dagegen der gesamte **Verbund** auf den **Stack** kopiert.

Des Weiteren muss unterschieden werden, ob ein **Verbund** an eine **Funktion übergeben** wird oder einfach nur **normal** auf einen Verbund **zugegriffen** wird. Wenn **normal** auf einen Verbund **zugegriffen** wird, dann wird das **erste Attribut** auf den Stack geschrieben. Wenn ein Verbund dagegen an eine **Funktion übergeben** wird, wird dieser **komplett** auf den **Stack** kopiert, um später Teil des **Stackframes** der **aufgerufenen Funktion** zu werden.

Das wird durch eine globale Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange ein **Funktionsaufruf** im **Picoc-ANF Pass** übersetzt wird und wieder auf `false` gesetzt, wenn die Übersetzung des **Funktionsaufrufs** abgeschlossen ist. Solange die Variable `argmode_on` auf `true` gesetzt ist, werden die **Knoten** `Assign(Stack(Num('3')), Global(Num('0')))` bzw. `Assign(Stack(Num('3')), Stackframe(Num('2')))` für die Ersetzung verwendet. Ist die Variable `argmode_on` auf `false`, werden die Knoten `Exp(Global(num))` bzw. `Exp(Stackframe(num))` für die Ersetzung verwendet.[72]

In Code 3.89 werden die Knoten `Assign(Stack(Num('3')), Global(Num('0')))` verwendet, da die Verbundsvariable `local_var` der `main`-Funktion in den **Globalen Statischen Daten** liegt und die Knoten `Assign(Stack(Num('3')), Stackframe(Num('2')))` werden verwendet, da die Verbundsvariable `local_var` der Funktion `fun_struct_from_global_data` im **Stackframe** dieser Funktion liegt.

```
1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_struct_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Stackframe(Num('2')))
14        NewStackframe(Name('fun_struct_from_stackframe.2'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
```

[71] Wie es in Unterkapitel 3.3.6.3.2 erklärt wurde

[72] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Assign(Stack(Num('3')), Global(Num('0')))
24        NewStackframe(Name('fun_struct_from_global_data.1'), GoTo(Name('addr@next_instr')))
25        Exp(GoTo(Name('fun_struct_from_global_data.1')))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]
```

Code 3.89: *PicoC-ANF Pass für die Übergabe eines Verbundes.*

Im **RETI-Blocks Pass** in Code 3.90 werden die **PicoC-Knoten** `Assign(Stack(Num('3')),Stackframe(Num('2'))` und `Assign(Stack(Num('3')),Global(Num('0')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun_struct_from_global_data.1',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Assign(Stack(Num('3')), Stackframe(Num('2')))
16        SUBI SP 3;
17        LOADIN BAF ACC -4;
18        STOREIN SP ACC 1;
19        LOADIN BAF ACC -3;
20        STOREIN SP ACC 2;
21        LOADIN BAF ACC -2;
22        STOREIN SP ACC 3;
23        # NewStackframe(Name('fun_struct_from_stackframe.2'), GoTo(Name('addr@next_instr')))
24        MOVE BAF ACC;
25        ADDI SP 5;
26        MOVE SP BAF;
27        SUBI SP 5;
28        STOREIN BAF ACC 0;
29        LOADI ACC GoTo(Name('addr@next_instr'));
30        ADD ACC CS;
31        STOREIN BAF ACC -1;
32        # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
33        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
```

```
34        # RemoveStackframe()
35        MOVE BAF IN1;
36        LOADIN IN1 BAF 0;
37        MOVE IN1 SP;
38        # Return(Empty())
39        LOADIN BAF PC -1;
40      ],
41    Block
42      Name 'main.0',
43      [
44        # StackMalloc(Num('2'))
45        SUBI SP 2;
46        # Assign(Stack(Num('3')), Global(Num('0')))
47        SUBI SP 3;
48        LOADIN DS ACC 0;
49        STOREIN SP ACC 1;
50        LOADIN DS ACC 1;
51        STOREIN SP ACC 2;
52        LOADIN DS ACC 2;
53        STOREIN SP ACC 3;
54        # NewStackframe(Name('fun_struct_from_global_data.1'), 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 3.90: *RETI-Block Pass für die Übergabe eines Verbundes.*

3.4 Fehlermeldungen

Die **Fehlerarten**, die der **PicoC-Compiler** ausgeben kann sind in den Tabellen 3.21, 3.22 und 3.23 und eingeteilt nach den Kategorien **„Fehlerarten in der Lexikalischen und Syntaktischen Analyse“**, **„Fehlerarten in den Passes“**, **„Fehlerarten, die zur Laufzeit auftreten“** aus Unterkapitel 2.6. Da der **PicoC-Compiler** nicht in der Lage ist **mehrere Dateien** zu kompilieren und somit keinen **Linker** nötig hat, mussten Fehler, die normalerweise beim **Linken** aufgefunden werden würden in den **Passes** umgesetzt werden und sind somit der Kategorie **„Fehlerarten in den Passes“** zuzuordnen.

Fehlerarten, wie z.B. `UninitialisedVariable` beim Verwenden einer **uninitialisierten Variable** oder `IndexOutOfBound` bei einem **Feldzugriff** auf einen **Index**, der **außerhalb** des **Feldes** liegt gibt es für die Sprache L_{PicoC} **nicht**, da da bei der Programmiersprache L_C, die eine Obermenge der Programmiersprache L_{PicoC} ist diese Fehlermeldungen auch **nicht** gibt. Das Programm in Code 3.91 läuft z.B. **ohne Fehlermeldungen** durch.

```c
#include <stdio.h>

void main() {
  int var;
  printf("\n%d", var);
}
```

Code 3.91: *Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet.*

Fehlerart	Beschreibung
`UnexpectedCharacter`	Der **Lexer** ist auf eine **unerwartete Zeichenfolge** gestossen, die in der **Konkretten Grammatik** für die **Lexikalische Analyse** 3.1.1 **nicht abgeleitet** werden kann.
`UnexpectedToken`	Der **Parser** hat ein **unerwartetes Token** erhalten, das in dem **Kontext** in dem es sich befand in der **Konkretten Grammatik** für die **Syntaktische Analyse** 3.2.10 **nicht abgeleitet** werden kann.
`UnexpectedEOF`	Der **Parser** hat in dem **Kontext** in dem er sich befand bestimmte **Tokens erwartet**, aber die **Eingabe endete** abrupt.

Tabelle 3.21: *Fehlerarten in der Lexikalischen und Syntaktischen Analyse.*

Fehlerart	Beschreibung
`UnknownIdentifier`	Es wird ein Zugriff auf einen **Bezeichner** gemacht (z.B. `unknown_var + 1`), der noch **nicht deklariert** und ist daher **nicht** in der **Symboltabelle** aufgefunden werden kann.
`UnknownAttribute`	Der **Verbundstyp** (z.B. `struct st {int attr1; int attr2;}`) auf dessen Attribut im momentanen Kontext zugegriffen wird (z.B. `var[3].unknown_attr`) besitzt das **Attribut** (z.B. `unknown_attr`) auf das zugegriffen werden soll **nicht**.
`ReDeclarationOrDefinition`	Ein Bezeichner[a] der bereits **deklariert** oder **definiert** ist (z.B. `int var`) wird **erneut** deklariert oder definiert (z.B. `int var[2]`). Dieser Fehler ist leicht festzustellen, indem geprüft wird ob das **Assoziative Feld** durch welches die **Symboltabelle** umgesetzt ist diesen **Bezeichner bereits als Schlüssel** besitzt.
`ConstAssign`	Wenn einer intialisierten **Konstante** (z.B. `const int const_var = 42`) ein **Wert zugewiesen** wird (z.B. `const_var = 41`). Der **einzige Weg**, wie eine Konstante einen Wert erhält ist bei ihrere **Initialisierung**.
`TooLargeLiteral`	Der **Wert** eines Literals ist **größer** als $2^{31} - 1$ oder **kleiner** als -2^{31}.
`NotExactlyOneMainFunction`	Das Programm besitzt **keine** oder **mehr als eine** `main`-Funktion.
`PrototypeMismatch`	Der **Prototyp** einer **deklarierten** Funktion (z.B. `int fun(int arg1, int arg2[3])`) stimmt nicht mit dem **Prototyp** der späteren **Definition** dieser Funktion (z.B. `void fun(int arg1[2], int arg2) { }`) überein.
`ArgumentMismatch`	Wenn die **Argumente** eines **Funktionsaufrufs** (z.B. `fun(42, 314)`) nicht mit dem **Prototyp** der Funktion die aufgerufen werden soll (z.B. `void fun(int arg[2]) { }`) nach **Datentypen** oder **Anzahl Argumente bzw. Parameter** übereinstimmt.
`MissingReturn`	Wenn eine Funktion, die ihrem **Prototyp** zufolge einen **Rückgabewert** hat, der nicht vom **Datentyp** `void` ist (z.B. `int fun() {...}`) als **letzte Anweisung** keine `return`-Anweisung hat, dass einen Wert des **entsprechenden Datentyps** zurückgibt[b].

[a] Z.B. von einer **Funktion** oder **Variable**.
[b] Der **entsprechende Datentyp** müsste auf das Beispiel von davor `void fun(int arg[2]) {...}` bezogen z.B. `return 42` sein.

Tabelle 3.22: *Fehlerarten in den Passes.*

Fehlerart	Beschreibung
`DivisionByZero`	Wenn bei einer **Division** durch 0 geteilt wird (z.B. `var / 0`).

Tabelle 3.23: *Fehlerarten, die zur Laufzeit auftreten.*

In Code 3.93 ist eine **typische Fehlermeldung** zu sehen. Eine **Fehlermeldung** fängt immer mit einem **Header** an, bei dem sich an den Fehlermeldungen des **GCC** orientiert wurde. Ein analoges Beispiel für eine **GCC-Fehlermeldung** für Code 3.93 ist in Code 3.92 zu sehen. Nacheinander stehen in Code 3.93 im **Header** der **Dateiname**, die **Position** des Fehlers in der Datei in der das fehlerhafte Programm steht, die **Fehlerart** und ein **Beschreibungstext**.

```
1 ./tests/error_wrong_written_keyword.c:8:5: error: expected ‘while’ before ‘wile’
2  } wile (True);
3    ^~~~
```

Code 3.92: *Fehlermeldung des GCC.*

Unter dem **Header** wird beim **PicoC-Compiler** ein kleiner **Ausschnitt des Programmes** um die Stelle herum an welcher der **Fehler aufgetreten** ist angzeigt. Die Kommandozeilenoptionen `-l` und `-c`, welche in Tabelle 4.1 erläutert werden könnten in diesem Zusammenhang interessant sein.

Das Symbol ~ bzw. eine Folge von ~ kennzeichnet beim **PicoC-Compiler** das **Lexeme**, welches an der Stelle des Fehlers **vorgefunden** wurde und das Symbol ^ soll einen **Pfeil** symbolisieren, der auf eine **Position zeigt** an der ein **anderer Tokentyp**, ein **anderer Datentyp** usw. **erwartet** worden wäre und **in der Zeile darunter** eine **Beschriftung** an sich hängen hat, die konkret angibt, was dort eingentlich **erwartet** worden wäre.

```
./tests/error_wrong_written_keyword.picoc:7:4: UnexpectedToken: Expected e.g. 'while', found
↪  'wile'.
   do {

   } wile (True);
     ^~~~
     'while'
 }
```

Code 3.93: *Beispiel für typische Fehlermeldung mit 'found' und 'expected'.*

Bei **Fehlermeldungen**, wie in Code 3.94, die ihre **Ursache** an einer **anderen Stelle im Code** haben, wird einmal ein **Header** mit **Programmauschnitt** für die **Stelle** an welcher der **Fehler aufgetreten** ist erstellt und ein weiterer **Header** mit **Programmauschnitt** für die **Stelle** welche die **Ursache** für das Auftreten dieses Fehlers ist.

```
./tests/error_redefinition.picoc:6:6: Redefinition: Redefinition of 'var'.
 void main() {
   int var = 42;
   int var = 41;
       ~~~
 }
 ./tests/error_redefinition.picoc:5:6: Note: Already defined here:

 void main() {
   int var = 42;
       ~~~
   int var = 41;
 }
```

Code 3.94: *Beispiel für eine langgestreckte Fehlermeldung.*

Bei manchen Fehlermeldungen, wie in Code 3.95 ist es **garnicht möglich** mit ~ ein Lexeme an der **Stelle** zu markieren, an welcher der **Fehler vorgefunden** wurde, da z.B. beim `UnexpectedEOF`-Fehler das **Ende der Programmes** erreicht wurde, wo es **kein sichtbares Lexeme** gibt, welches man markieren könnte. Des Weiteren ist in Code 3.95 interessant, dass in markierten Zeile in Code 3.95 **mehrere Tokens** angegeben werden,

die nach der Konkreten Grammatik 3.2.8 an dieser Stelle erwartet werden können. Es werden **standardmäßig** nur die **ersten** 5 **erwarteten Tokens** angegeben, aber mittels der Kommandozeilenoptionen `-vv` kann auch aktiviert werden, dass **alle möglichen Tokens** in einer solchen `or`-Kette angegeben werden.

```
./tests/error_unexpected_eof.picoc:4:13: UnexpectedEOF: Unexpected end-of-file, expected e.g.
↪  reti comment or '}' or '(' or 'print' or '*' or 'if'.

void main() {
            ^
            reti comment or '}' or '(' or 'print' or '*' or 'if'
```

Code 3.95: *Beispiel für Fehlermeldung mit mehreren erwarteten Tokens.*

Bei wiederum anderen Fehlermeldungen, wie in Code 3.96 ist es **nicht möglich** ein **erwartetes Token** anzugeben, da das Programm in Code 3.96 eigenltich korrekt nach der **Konkretten Grammatik** 3.2.8 abgeleitet ist, weshalb sich hier keine **erwarteten Tokens** angeben lassen. Es liegt auf das konkrete Beispiel in Code 3.96 bezogen nämlich daran, dass die Variable `unknown_identifier` **nicht definiert** ist, weshalb dieses Programm nicht in der **gesamten Syntax** der Sprache L_{PicoC} sein kann.

```
./tests/error_unknown_identifier_ref.picoc:5:18: UnknownIdentifier: Identifier
↪  'unknown_identifier' wasn't declared yet.

 void main() {
   int var = 42 + unknown_identifier;
                  ~~~~~~~~~~~~~~~~~~
 }
```

Code 3.96: *Beispiel für Fehlermeldung ohne expected.*

Bei z.B. dem **Laufzeit-Fehler** `DivisionByZero` wird beim Auftreten einer **Division durch** 0 mit entsprechendem **RETI-Code** gecheckt, ob der **rechte Operand** einer **Divisionsoperation** eine 0 ist und wenn dies der Fall ist in das `ACC`-Register der Wert 1 geschrieben und die Programmausführung beendet. Der Wert 1 im `ACC`-Register stellt eine `DivisionByZero`-Fehlermeldung dar. Wenn es noch weitere **Laufzeit**-Fehlerarten gebe, dann würde eine 2 im `ACC`-Register für einen anderen **Laufzeit**-Fehler stehen usw.

4 Ergebnisse und Ausblick

Zum Schluss soll ein **Überblick** über das Resultat dessen, was im Kapitel Implementierung implementiert wurde gegeben werden. Im Unterkapitel 4.1 wird darauf eingegangen, ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel Einführung alle implementiert werden konnten. Daraufhin wird mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten zugegriffen werden kann und es wird auch auf Funktionalitäten **anderer mitimplementierter Tools** eingegangen. Im Unterkapitel 4.2 wird aufgezeigt, was zur **Qualitätssicherung** implementiert wurde, um zu gewährleisten, dass der **PicoC-Compiler** die Kompilierung der **Programmiersprache** L_{PicoC} in **Syntax** und **Semantik identisch** zur entsprechenden **Untermenge** der Programmiersprache L_C umsetzt. Als allerletztes wird im Unterkapitel 4.3 ein Ausblick gegeben, wie der PicoC-Compiler **erweitert** werden könnte.

4.1 Funktionsumfang

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

Bei der Implementierung des **PicoC-Compilers** wurden verschiedene **Kommandozeilenoptionen** und **Modes** implementiert. Diese werden in den folgenden Unterkapiteln 4.1.1, 4.1.3 und 4.1.4 mithilfe kurzer **Anleitungen** erklärt.

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

4.1.1 Kommandozeilenoptionen

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

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

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

Daneben gibt es allerdings auch die Möglichkeit **Kommandozeilenoptionen** `<cli-options>` in der Form `> picoc_compiler <cli-options> program.picoc`[3] mitanzugeben, von denen die **wichtigsten** in Tabelle 4.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumenation** unter Link[4] nachgelesen werden. Die die letzte Spalte gibt den **Standardwert** an, der bei der normalen Nutzung des **PicoC-Compilers** gesetzt ist.[5]

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

Tabelle 4.1: *Kommandozeilenoptionen, Teil 1.*

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

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

Tabelle 4.2: *Kommandozeilenoptionen, Teil 2.*

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

4.1.2 RETI-Interpreter

Um den nach der Kompilierung durch den **PicoC-Compiler** generierten **RETI-Code** auch **ausführen** zu können, wurde zusätzlilch ein **RETI-Interpreter** implementiert. Die Möglichkeit den **RETI-Code** auch **ausführen** zu können, ist für die **Qualitätssicherung**, die in Unterkapitel 4.2 genauer erklärt wird notwendig. Ein **Programm**, dass in der Sprache L_{PicoC} geschrieben ist, lässt sich mit der Kommandozeilenoption `-R` im Anschluss an die **Kompilierung** durch den **PicoC-Compiler**, mit dem **RETI-Interpreter** ausführen. Der **vollständige Befehl** hierfür lautet `> picoc_compiler -R program.picoc`, wie es bereits aus dem vorherigen Unterkapitel 4.1.1 bekannt ist.

Der **RETI-Interpreter** wird an der Stelle, an welcher normalerweise die **RETI-CPU** den **RETI-Code** ausführen würde eingesetzt. In Abbildung 4.1 ist in einem weiteren **T-Diagramm** (siehe Unterkapitel 2.1.1) dargestellt, wie anstelle einer **RETI-CPU** der **RETI-Intepreter** den **RETI-Code** ausführt, indem er die **Semantik** der **RETI-Befehle** simuliert. In dem **T-Diagramm** ist der Vollständigkeit halber auch dargestellt, dass der **PicoC-Compiler** und **RETI-Interpreter** beide mithilfe des **Python-Interpreters** ausgeführt werden, der wiederum z.B. auf einer X_{86_64}-**Maschine** ausgeführt wird[6].

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

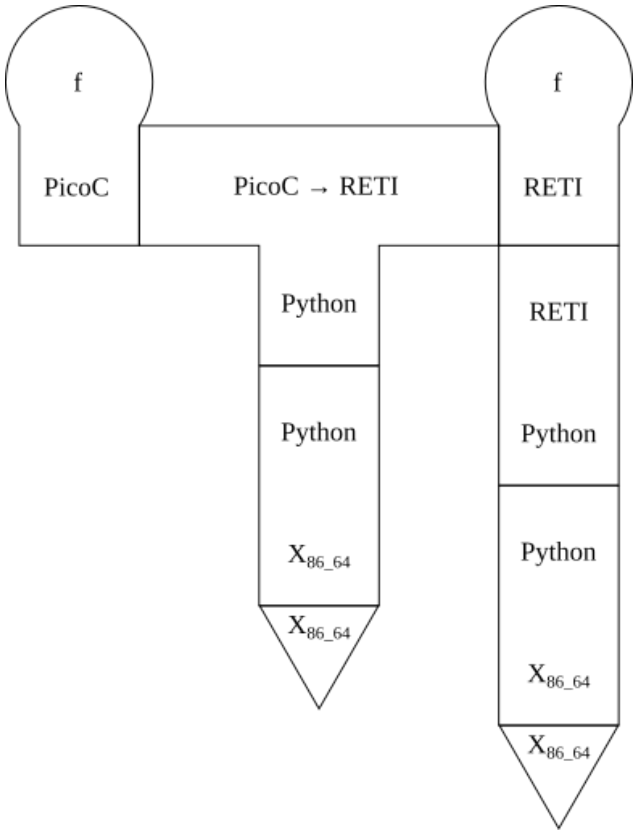


Abbildung 4.1: *Ausführung von RETI-Code mit dem RETI-Interpreter.*

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

4.1.3 Shell-Mode

Will man z.B. eine **Folge von Anweisungen** in der Programmiersprache L_{PicoC} **schnell** kompilieren ohne eine Datei erstellen zu müssen, so kann der **PicoC-Compiler** im sogenannten **Shell-Mode** aufgerufen werden. Hierzu wird der PicoC-Compiler **ohne Argumente** `> picoc_compiler` aufgerufen, wie es in Code 4.1 zu sehen ist.

Mit dem `> compile <cli-options> <seq-of-stmts>`-Befehl (oder der **Abkürzung** `cpl`) kann **PicoC-Code** zu **RETI-Code** kompiliert werden. Die Kommandozeilenoptionen `<cli-options>` sind dieselben, wie wenn der Compiler **direkt** mit Kommandozeilenoptionen aufgerufen wird. Die **wichtigsten** dieser **Kommandozeilenoptionen** sind in Tabelle 4.1 angegeben. Die angegebene **Folge von Anweisungen** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

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

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

Compilation successfull

PicoC> quit
```

Code 4.1: *Shellaufruf und die Befehle* `compile` *und* `quit`*.*

Wenn man möglichst alle nützlichen **Kommandozeilenoptionen** direkt aktiviert haben will, bei denen es **keinen** Grund gibt sie nicht mitanzugeben, kann der Befehl `> most_used <cli-options> <seq-of-stmts>` (oder seine **Abkürzung** `mu`) genutzt werden. Auf diese Weise müssen diese Kommandozeilenoptionen **nicht** wie beim `compile`-Befehl jedes mal **selbst** angeben werden. In Tabelle 4.1 sind in grau die **Standardwerte** der einzelnen **Kommandozeilenoptionen** angegeben, die bei dem Befehl `most_used` gesetzt werden. In Code 4.2 ist der `most_used`-Befehl in seiner Verwendung zu sehen.

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

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

Compilation successfull
```

Code 4.2: *Shell-Mode und der Befehl* `most_used`.

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

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

4.1.4 Show-Mode

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

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

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

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

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `> make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen** `<more-options>`, die für die **Makefile** gesetzt werden können, sind in Tabelle 4.3 aufgelistet.

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

Tabelle 4.3: *Makefileoptionen.*

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

Der **Show-Mode** nutzt den Terminal Texteditor **Neovim**[8], um einen **Dateiinhalt** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 4.2 zu sehen ist. Für den **Show-Mode** wird eine eigene **Konfiguration für Neovim** verwendet, welche in der **Konfigurationsdatei** `/interpr_showcase.vim` spezifiziert ist.

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

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

[8] *Home - Neovim*.

Befehls **angezeigt** werden.

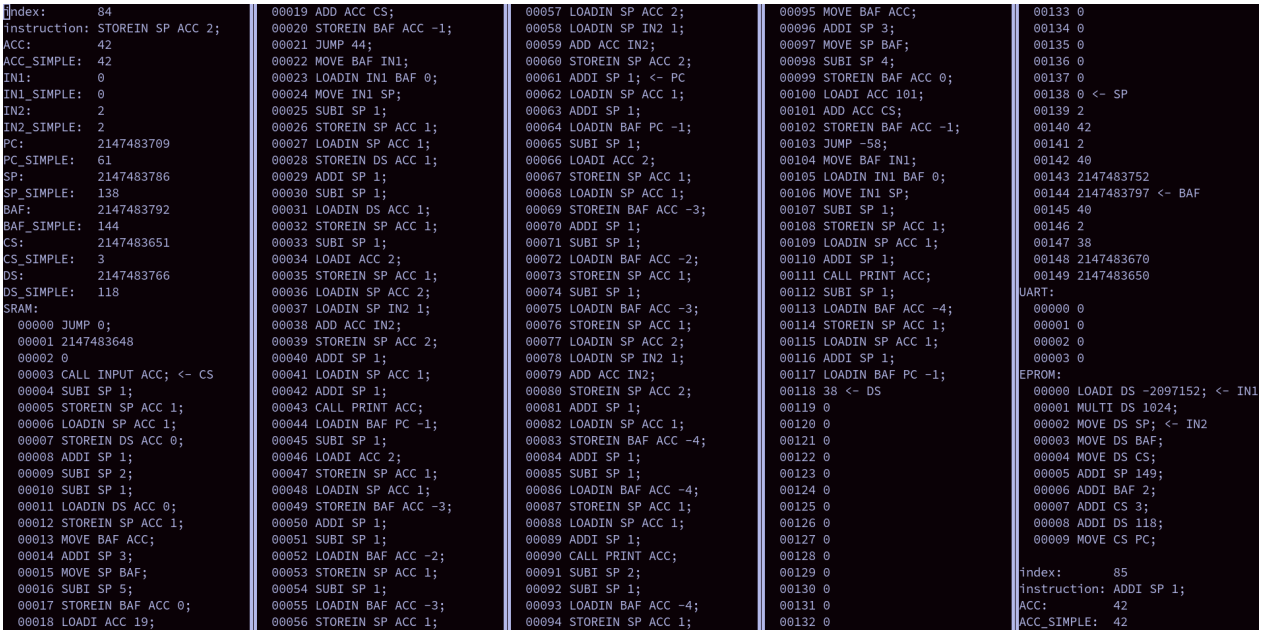


Abbildung 4.2: *Show-Mode in der Verwendung.*

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

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

4.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm 2.3.1[10] genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 4.2.1 definieren. Ein **beliebiges** Testprogramm P_{PicoC} in der Sprache L_{PicoC} muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm P_{RETI} in der Sprache L_{RETI}, trotz der **unterschiedlichen Sprache**.

Die Beziehungen im Diagramm 4.2.1 werden mithilfe von **Tests** verifziert. Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis `/tests` bzw. unter Link[11] zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 4.4.

[9] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt.
[10] In Unterkapitel 2.1.
[11] https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

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

Tabelle 4.4: *Testkategorien.*

Dass ein Programm P_{PicoC} und das Programm P_{RETI}, welches das **kompilierte** P_{PicoC} ist nach Diagramm 4.2.1 die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn die Tests so konstruiert sind, dass es sehr **unwahrscheinlich** ist, zufällig bei der gewählten Eingabe die spezifische Ausgabe zu erhalten. Wenn **immer mehr Tests**, die alle einen unterschiedlichen **Teil der Semantik** der Sprache L_{PicoC} abdecken vorliegen, bei denen die jeweiligen Programme P_{PicoC} und P_{RETI} interpretiert die **gleiche Ausgabe** haben, dann kann mit **immer höherer Wahrscheinlichkeit** von einem **funktionierenden** Compiler ausgegangen werden.

Die Kante vom Testprogramm P_{PicoC} zur *Ausgabe* aus Diagramm 4.2.1 ist so umgesetzt, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat. Der **Schreiber des Tests** übernimmt die Rolle des entsprechenden **Interpreters** aus Diagramm 2.3.1. Die **erwartete Ausgabe** `<space_seperated_output>` ist seine eigene Interpretation des **PicoC-Codes**.

Ein Beispiel für einen **Test** ist in Code 4.3 zu sehen. Die Tests werden mithilfe des Bashskripts `/run_tests.sh` **ausgeführt** oder mithilfe der `/Makefile` mit dem Befehl `> make test`, welcher einfach nur dieses Bashskript ausführt. Bei der **Ausführung** des Bashskripts `/run_tests.sh`, wird als erstes für **jeden** Test das Bashskript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert[12] und die entsprechenden Zeichenketten `<space_seperated_input>`, `<space_seperated_output>` und `<datasegment_size>` in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt. Das **letztere Skript** kann ebenfalls mit dem Befehl `> make extract` ausgeführt werden.

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

[12] Falls vorhanden.

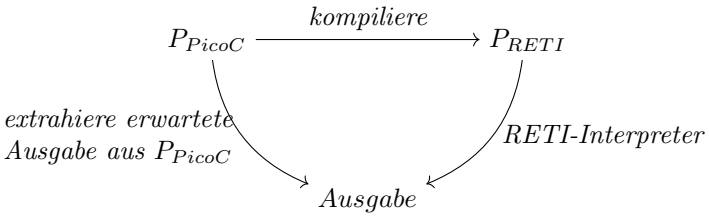
Datensegments für die Ausführung des entsprechenden Tests.

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

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

Code 4.3: *Typischer Test.*

Die Kante vom Programm P_{RETI} zur *Ausgabe* aus Abbildung 4.2.1 ist dadurch umgesetzt, dass das Programm P_{RETI} vom **RETI-Interpreter** interpretiert wird und jedes mal beim Antreffen des **RETI-Befehls** `CALL PRINT ACC`, der entsprechende **Inhalt** des `ACC`-Registers in die Datei `<program>.out` ausgegeben wird. Ein Test kann[13] die **Korrektheit** des **Teils der Semantik** der Sprache L_{PicoC}, die er abdeckt verifizieren, wenn der **Inhalt** von `<program>.out_expected` und `<program>.out` **identisch** ist.



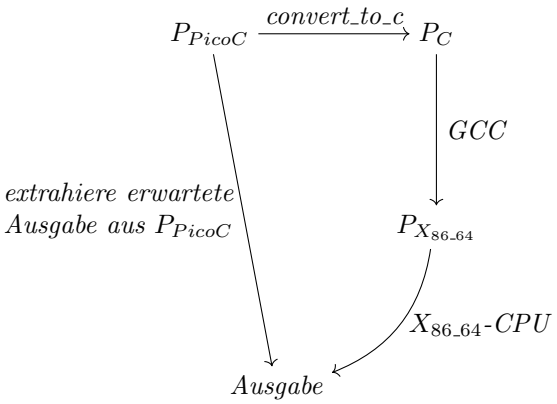
(4.2.1)

Allerdings gibt es bei dem Testverfahren, welches in Diagramm 4.2.1 dargestellt ist ein **Problem**, denn der **Schreiber der Tests** ist in diesem Fall die **gleiche Person**, die auch den **PicoC-Compiler implementiert** hat. Wenn der **Schreiber** der Tests bzw. **Implementierer** des PicoC-Compilers ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl in den **Tests** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht[14]. Die **Tests** können dann nur **bestätigen**, dass der PicoC-Compiler so implementiert wurde, wie der **Implementierer** sich die **Semenatik** der Sprache L_{PicoC} vorstellt.

Aus diesem Grund muss hier eine **weitere Maßnahme** eingeführt werden, welche in Diagramm 4.2.2 dargestellt ist. Diese Maßnahme gewährleistet, dass die *Ausgabe* sich auf jeden Fall aus der tatsächlichen **Semantik** der Sprache L_{PicoC}[15] ergeben muss. Das wird erreicht, indem wie in Diagramm 4.2.2 dargestellt ist, überprüft wird, ob die *Ausgabe* des Pfades $(P_{PicoC}, Ausgabe)$ mit der *Ausgabe* des Pfades von $(P_C, P_{X_{86\_64}}, Ausgabe)$ **identisch** ist.

Im Diagramm 4.2.2 hat die Kante *extrahiere erwartete Ausgabe aus* P_{PicoC} die **gleiche Umsetzung**, wie die entsprechende Kante in Diagramm 4.2.1, welche bereits erklärt wurde. Die Kante *GCC* ist so umgesetzt, dass der **GCC**[16] zur **Kompilierung** des Programms P_C von der Programmiersprache L_C in die Maschinensprache $L_{X_{86\_64}}$ zu $P_{X_{86\_64}}$ verwendet wird. Die Kante $X_{86\_64}{-}CPU$ ist so umgesetzt, dass sie das Programm $P_{X_{86\_64}}$ auf einer $X_{86\_64}{-}CPU$ **ausführt**, wobei hierfür zumindestens beim Computer des **Implementierers des PicoC-Compilers** eine $X_{86\_64}{-}CPU$ verwendet wird.

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



(4.2.2)

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

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

Durch das **Abgleichen** mit dem **GCC** in Diagramm 4.2.2 wird etwas wichtiges **sichergestellt**. Durch diese zweifache Überprüfung bestätigen die Tests nicht nur die Interpretation, die der **Schreiber** der Tests und **Implementierer** des PicoC-Compilers von der Semantik der Sprache L_{PicoC} hat, sondern stellen die tätsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} sicher.

Für die zweifache Überprüfung durchläuft jeder Test eine **Verifikation**, wie sie in Diagramm 4.2.2 dargestellt ist. In dieser wird **verifiziert**, ob bei der **Kompilierung** des Testprogramms P_C mit dem **GCC** und **Ausführung** des hieraus generierten X_{86_64}-Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe `// expected:<space_seperated_output>` des Testschreibers ist.

Für die **Verifikation** ist das Bashskript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt werden kann. Beim Befehl `> make test` wird dieses Bashskript **vor** dem eigentlichen Testen[19] ausgeführt. In Code 4.4 ist ein **Testdurchlauf** mit `> make test` zu sehen.

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

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

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

Code 4.4: *Testdurchlauf.*

Der Befehl `> make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` `TESTNAME`, `VERBOSE` und `DEBUG` aus Tabelle 4.3 kombinieren.

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

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

4.3 Erweiterungsideen

Nachdem der **Funktionsumfang** des **PicoC-Compilers** in Unterkapitel 4.1 erläutert wurde, wird in diesem Unterkapitel die Diskussion geführt, wie sich dieser in Zukunft vielleicht noch **erweitern** liese[21]. Weitere Ideen, die im **PicoC-Compiler** implementiert werden könnten, wären:

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern**. Erst wenn alle Register **voll** sind, werden Variablen an Adressen im **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller** ist,

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

als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations** (Definition 2.49) zuzuweisen, wird mithilfe einer **Liveness Analyse** (Defintion 5.12) ein **Interferenzgraph** (Definition 5.15) mit **Variablen** als Knoten aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition 5.14) angewandt, der den **Variablen** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Sobald eine Programmiersprache es erfordert für die Kompilierung **Blöcke** in den Passes einzuführen[22], muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussnalayse** (Definition 5.18) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition 5.16) seperat auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.[23]

- **Tail Call:** Wenn ein Funktionsaufruf der **letzte ausgeführte Ausdruck** in einem Funktionsblock ist, wird der **Stackframe** dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion **zurückgekehrt** werden muss[24]. Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden, **bevor** der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die **nur Tail Calls** ausführt, mit **Stackframes** nur eine **konstante Menge** an **Speicherplatz** auf dem **Stack** verbraucht. In Code 4.5 sind **zwei Tail Calls** markiert.

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

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

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

- **Mehrere Dateien: Funktionen** und **Attribute** werden in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden `.h`-**Headerdateien** verwendet und für deren **Definitionen** sind `.c`-**Quellcodedateien** da. Hierbei ist der **Basisname** einer `.h`-**Headerdatei identisch** zu dem der entsprechenden `.c`-**Quellcodedatei**. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei an genau dieser Stelle in die einbindende Datei entspricht. Über einen **Linker** (Definition 5.6) können **kompilierte** `.o`-**Objektdateien** (Definition 5.5) zusammengefügt werden. Der **Linker** achtet darauf **keinen doppelten Code** zuzulassen.

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

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

- **Garbage Collector:** Anstelle der `free`-Funktion kann auch einfach die `malloc`-Funktion direkt so implementiert werden, dass sobald der Speicherplatz auf dem **Heap** knapp wird, Speicherplatz **freigegeben** wird. Es soll Speicherplatz freigegeben werden, der sowieso **unmöglich** in der **Zukunft** mehr **gebraucht** werden würde. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition 5.19) implementieren.

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

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

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

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

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

- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt, durch welche spezielle **Informationen** in **Objektcode** (Definition 5.5) geschrieben werden. Diese Informationen teilen einem **Debugger** unter anderem mit, wo die **RETI-Befehle** für Anweisungen, die zu diesen übersetzt wurden **beginnen** und wo sie **aufhören**. Auf diese Weise weiß der **Debugger**, bis wohin er die **RETI-Befehle** ausführen soll, damit er eine Anweisung abgearbeitet hat.

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

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

unahbängig machen, in welcher dieser implementiert ist, als auch von der **Maschine**, die das **cross-compilen** (Definition 2.6) übernimmt. Im Unterkapitel 4.3 wird **genauer** hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zu einem **Compiler** für die **RETI-CPU** gemacht, der auf der RETI-CPU **selbst** läuft.

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

Code 4.5: *Beispiel für Tail Call.*

Anmerkung

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

Appendix

Dieses Kapitel dient als Lagerstätte für **Definitionen**, **Tabellen**, **Abbildungen** und ganze **Unterkapitel**, die zum Erhalt des **roten Fadens** und des **Leseflusses** in den vorangegangenen Kapiteln hierher ausgelaggert wurden. Im Unterkapitel RETI Architektur Details können einige Details der **RETI-Architektur** nachgeschaut werden, die im Kapitel Einführung den Lesefluss **stören** würden und zum Verständnis nur **bedingt wichtig** sind. Im Unterkapitel Sonstige Definitionen sind einige **Definitionen** ausgelaggert, die zum Verständnis der **Implementierung** des PicoC-Compilers **nicht wichtig** sind, aber z.B. an einer bestimmten Stelle in den vorangegangenen Kapiteln **kurz Erwähnung** fanden. Im Unterkapitel Bootstrapping wird ein Vorgehen, das **Bootstrapping** erklärt, welches beim PicoC-Compiler **nicht umgesetzt** wurde, es aber erlauben würde aus dem **PicoC-Compiler** einen Compiler für die **RETI-CPU** zu machen, der auf der RETI-CPU selbst läuft.

RETI Architektur Details

Hier wird die **Semantik** der verschiedenen Befehle des **Befehlssatzes** der **RETI-Architektur** mithilfe von Tabelle 5.1, Tabelle 5.2 und Tabelle 5.3 dokumentiert. Des Weiteren sind in Abbildung 5.1 die **Datenpfade** der **RETI-Architektur** dargestellt.

Typ	Modus	Befehl	Wirkung
01	00	LOAD D i	$D := M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
01	01	LOADIN S D i	$D := M(\langle S \rangle + i), \langle PC \rangle := \langle PC \rangle + 1$
01	11	LOADI D i	$D := 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$, bei $D = PC$ wird der PC nicht inkrementiert
10	00	STORE S i	$M(\langle i \rangle) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	01	STOREIN D S i	$M(\langle D \rangle + i) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	11	MOVE S D	$D := S, \langle PC \rangle := \langle PC \rangle + 1$, Move: Bei $D = PC$ wird der PC nicht inkrementiert

Tabelle 5.1: *Load und Store Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.*

Typ	M	RO	F	Befehl	Wirkung
00	0	0	000	ADDI D i	$[D] := [D] + [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	001	SUBI D i	$[D] := [D] - [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	010	MULI D i	$[D] := [D] * [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	011	DIVI D i	$[D] := [D] / [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	100	MODI D i	$[D] := [D] \% [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	OPLUSI D i	$[D] := [D] \oplus 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	110	ORI D i	$[D] := [D] \vee 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	ANDI D i	$[D] := [D] \wedge 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	000	ADD D i	$[D] := [D] + [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	001	SUB D i	$[D] := [D] - [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	010	MUL D i	$[D] := [D] * [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	011	DIV D i	$[D] := [D] / [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	100	MOD D i	$[D] := [D] \% [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	OPLUS D i	$D := D \oplus M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	110	OR D i	$D := D \vee M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	AND D i	$D := D \wedge M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	000	ADD D S	$[D] := [D] + [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	001	SUB D S	$[D] := [D] - [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	010	MUL D S	$[D] := [D] * [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	011	DIV D S	$[D] := [D] / [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	100	MOD D S	$[D] := [D] \% [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	OPLUS D S	$D := D \oplus S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	110	OR D S	$D := D \vee S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	AND D S	$D := D \wedge S, \langle PC \rangle := \langle PC \rangle + 1$

Tabelle 5.2: *Compute Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.*

Type	Condition	J	Befehl	Wirkung
11	000	00	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11	001	00	JUMP$_>$i	Falls $[ACC] > 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	010	00	JUMP$_=$i	Falls $[ACC] = 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	011	00	JUMP$_\geq$i	Falls $[ACC] \geq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	100	00	JUMP$_<$i	Falls $[ACC] < 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	101	00	JUMP$_\neq$i	Falls $[ACC] \neq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	110	00	JUMP$_\leq$i	Falls $[ACC] \leq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$ $\langle PC \rangle := \langle PC \rangle + [i]$
11	111	00	JUMPi	$\langle PC \rangle := \langle PC \rangle + [i]$
11	*	01	INT i	$\langle PC \rangle := IVT[i]$ Interrupt Nr.i wird Ausgeführt
11	*	10	RTI	Rücksprungadresse vom Stack entfernt, in PC geladen, Wechsel in Usermodus

Tabelle 5.3: *Jump Befehle aus C. Scholl, „Betriebssysteme“, nicht selbst zusammengestellt, leicht abgewandelt.*

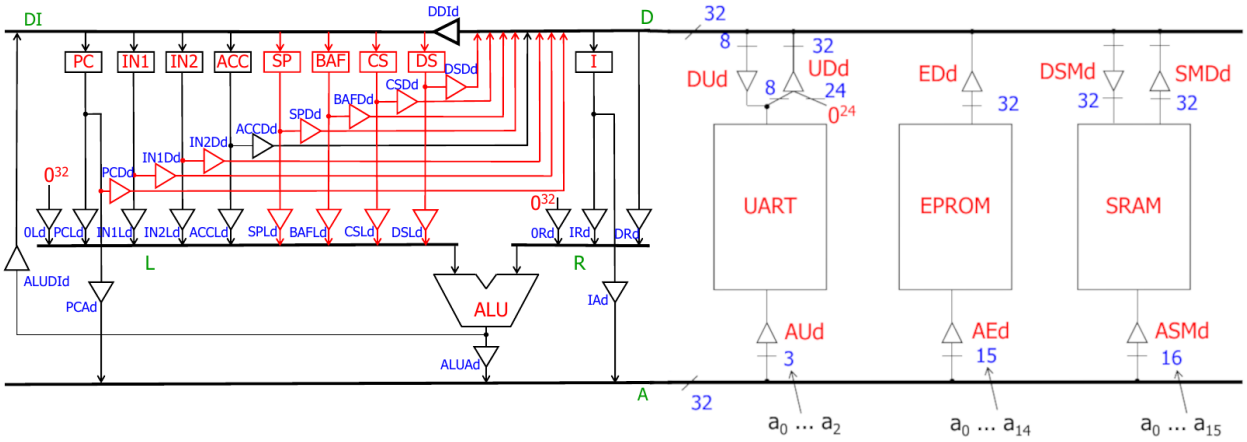


Abbildung 5.1: *Datenpfade der RETI-Architektur aus C. Scholl, „Betriebssysteme“, nicht selbst erstellt.*

Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die in den vorangegangenen Kapiteln **Erwähnung** fanden und zur Beibehaltung des **roten Fadens** und des **Leseflusses** in dieses Unterkapitel **ausgelaggert** wurden. Die Definitionen in diesem Unterkapitel vermitteln **Theorie über Compilerbau**, die über das **hinausgeht**, was zum Verständnis der Implementierung des **PicoC-Compilers** notwendig ist.

Definition 5.1: Bezeichner (bzw. Identifier)

Zeichenfolge[a]*, die eine Konstante, Variable, Funktion usw. innerhalb ihres **Sichtbarkeitsbereichs** (Definition 1.9) **eindeutig** benennt.*[b c]

[a] Bzw. **Tokenwert**.
[b] Außer wenn z.B. bei Methoden die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als **weiteres Unterscheidungsmerkmal** hinzugenommen, damit es **eindeutig** ist.
[c] Thiemann, „Einführung in die Programmierung“.

Definition 5.2: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a] Thiemann, „Compilerbau“.

Definition 5.3: Assemblersprache (bzw. engl. Assembly Language)

*Eine sehr **hardwarenahe** Programmiersprache, deren **Befehle** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen*[a] *haben. Viele **Befehle** haben eine ähnliche übliche Struktur* `Operation <Operanden>`*, mit einer **Operation**, die einen **Opcode** eines Maschinenbefehls bezeichnet und keinen oder mehreren **Operanden**, so wie die **Maschinenbefehle**, denen sie entsprechen. Allerdings gibt es oftmals noch viel „syntaktischen Zucker“ **innerhalb**[b] der Befehle und **drumherum**[c].*[d]

[a] Befehle der Assemblersprache, die **mehreren Maschinenbefehlen** entsprechen werden auch als **Pseudo-Befehle** bezeichnet und entsprechen dem, was man im allgemeinen als **Macro** bezeichnet.
[b] Z.B. erlaubt die Assemblersprache des **GCC** für die X_{86_64}**-Architektur** für manche Operanden die Syntax `n(%r)`, die einen **Speicherzugriff** mit **Offset** *n* zur Adresse, die im **Register** `%r` steht durchführt, wobei z.B. die Klammern `()` usw. nur „syntaktischer Zucker“ sind und natürlich **nicht mitkodiert** werden.

[c] Z.B. sind im X_{86_64}-Assembler die Befehle in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels `jmp <label>` gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet, hat **keine** direkte **Entsprechung** in einem handelsüblichen Prozessor oder Hauptspeicher.
[d] P. Scholl, „Einführung in Embedded Systems“.

Anmerkung

Ein **Assembler** (Definition 5.4) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt **Maschinencode** bzw. **Objektcode** (Definition 5.5) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer **abstrahieren** und dem Benutzer daher standardmäßig einfach nur die Ausgabe liefern, welche er in den allermeisten Fällen haben will, nämlich den **Maschinencode**, der direkt ausführbar ist bzw. den **Objektcode** der ausführbar ist, wenn er später mit dem **Linker** (Definition 5.6) zu Maschienencode zusammengesetzt wird.

Definition 5.4: Assembler

*Übersetzt im allgemeinen **Assemblercode** in **Assemblersprache** zu **Maschinencode** bzw. **Objektcode** in **Maschinensprache**. Der **Maschinencode** und **Objektcode** werden üblicherweise beide in **binärer Repräsentation** erzeugt.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 5.5: Objektcode

*Bei Komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen, wird häufig **Objektcode** erzeugt, der neben der Folge von Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschinencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 5.6: Linker

*Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt** bzw. zusammenfügt und Adresseresolution macht, sodass unter anderem kein vermeidbarer **doppelter** Code darin vorkommt.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 5.7: Transpiler (bzw. Source-to-source Compiler)

***Kompiliert** zwischen **Sprachen**, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten*[a].[b]

[a] Ein gutes Beispiel hierfür ist die Programmiersprache **TypeScript**, die als **Obermenge** von **JavaScript** die Sprache Javascript **erweitern** will und gleichzeitig die **Syntax** von JavaScript unterstützen will. Daher bietet es sich an Typescript zu Javascript zu **transpilieren**.
[b] Thiemann, „Compilerbau“.

Definition 5.8: Rekursiver Abstieg

*Es wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig entsprechend der Produktionen, welche sie jeweils umsetzen gegenseitig auf.*

Anmerkung

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 5.10) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 5.8) verwenden, lässt sich eine **bessere minimale Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschafft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist[a].

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die beim Algorithmus zur Entscheidung des **Wortproblems** verwendet wird, eine **Linksrekursive Grammatik** (Definition 5.9) ist[b].

[a] Mehr **Erklärung** hierzu findet sich im Unterkapitel 2.4.
[b] Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Definition 5.9: Linksrekursive Grammatiken

*Eine **Grammatik** ist **linksrekursiv**, wenn sie ein **Nicht-Terminalsymbol** enthält, dass **linksrekursiv** ist.*

*Ein **Nicht-Terminalsymbol** ist **linksrekursiv**, wenn das **linkeste Symbol** in einer seiner **Produktionen** es selbst ist oder zu sich selbst gemacht werden kann durch eine Folge von Ableitungen:*

$$A \Rightarrow^* Aa,$$

*wobei a eine beliebige Folge von **Grammatiksymbolen**[a] ist.*[b]

[a] Also eine beliebige Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[b] *Parsers — Lark documentation*.

Definition 5.10: LL(k)-Grammatik

*Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt **eindeutig** durch die **nächsten** k **Tokentypen** der **Tokens**, welche aus dem **Eingabewort** generiert wurden bestimmt werden kann*[a]*. Dabei steht **LL** für **l**eft-to-right und **l**eftmost-derivation, da das **Eingabewort** von **links nach rechts** geparst und immer **Linksableitungen** genommen werden müssen*[b]*, damit die obige Bedingung mit den **nächsten** k Symbolen gilt.*[c]

[a] Das wird auch als **Lookahead** von k bezeichnet.
[b] Wobei sich das mit den **Linksableitungen** automatisch ergibt, wenn man das Eingabewort von **links-nach-rechts** parsed und jeder der nächsten k **Ableitungsschritte** eindeutig sein soll.
[c] Nebel, „Theoretische Informatik“.

Definition 5.11: Earley Erkenner

*Ist ein **Erkenner** (Definition 2.37), der für **alle Kontextfreien Sprachen** das **Wortproblem** entscheiden kann und das mittels **Dynamischer Programmierung** mit dem **Top-Down Ansatz** (siehe Unterkapitel 2.4) umsetzt.*[a b c]

***Eingabe** und **Ausgabe** des Algorithmus sind:*

- ***Eingabe:** **Eingabewort** w und **Konkrete Grammatik** $G_{Parse} = \langle N, \Sigma, P, S \rangle$.*
- ***Ausgabe:** 0 wenn $w \notin L(G_{Parse})$ und 1 wenn $w \in L(G_{Parse})$.*[d]

*Bevor dieser **Algorithmus** erklärt wird, müssen noch einige **Symbole** und **Notationen** erklärt werden:*

- *α, β, γ stellen eine **beliebige Folge** von **Grammatiksymbolen**[e] dar.*
- *A und B stellen jeweils ein **Nicht-Terminalsymbole** dar.*
- *a stellt ein **Terminalsymbol** dar.*
- ***Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine **Produktion** dar, in der α **bereits geparst** wurde und β **noch geparst** werden muss.*
- *Es wird eine spezielle **Indexierung** innerhalb des **Eingabeworts** verwendet, die informell ausgedrückt so umgesetzt ist, dass die Indices **zwischen Lexemen** liegen. **Index** 0 ist **vor** dem ersten **Lexeme** verortet. **Index** 1 ist **nach** dem ersten **Lexeme** verortet. **Index** n ist **nach** dem **letzten Lexeme** verortet.*

*und davor müssen noch einige **Begriffe definiert** werden:*

- ***Zustandsmenge:** Für jeden der $n+1$ **Indices**[f] j des **Eingabeworts** w wird eine **Zustandsmenge** $Z(j)$ generiert.*
- ***Zustand einer Zustandsmenge:** Ist ein **Tupel** $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die **aktuelle Produktion** ist, die bis **Punkt** $\bullet$ geparst wurde und i der **Index** ist, ab welchem der Versuch der Erkennung eines **Teilworts** des **Eingabeworts** mithilfe dieser **Produktion begann**.*

*Der **Ablauf** des Algorithmus ist wie folgt:*

1. ***Initialisiere** $Z(0)$ mit der **Produktion**, welche das **Startsymbol** S auf der **linken Seite** des ::=-Symbols hat.*
2. *Es werden für jeden **Zustand** in der **aktuellen Zustandsmenge** $Z(j)$ die folgenden **Operationen ausgeführt**:*
   - ***Voraussage:** Wenn der **Zustand** die Form $(A ::= \alpha \bullet B\gamma, i)$ hat, wird für jede **Produktion** $(B ::= \beta)$ in der Konkreten Grammatik, die ein B auf der **linken Seite** des ::=-Symbols hat ein **Zustand** $(B ::= \bullet\beta, j)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt.*
   - ***Überprüfung:** Wenn der **Zustand** die Form $(A ::= \alpha \bullet a\gamma, i)$ hat, wird der **Zustand** $(A ::= \alpha a \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j+1)$ hinzugefügt.*
   - ***Vervollständigung:** Wenn der **Zustand** die Form $(B ::= \beta\bullet, i)$ hat, werden alle **Zustände** in $Z(i)$ gesucht, welche die Form $(A ::= \alpha \bullet B\gamma, i)$ haben und es wird der **Zustand** $(A ::= \alpha B \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt.*

   bis:

   - *der **Zustand** $(S ::= \beta\bullet, 0)$ in der **Zustandsmenge** $Z(n)$ auftaucht[g] $\Rightarrow w \in L(G_{Parse})$.*
   - ***keine Zustände** mehr hinzugefügt werden können $\Rightarrow w \notin L(G_{Parse})$.*

[a] Jay Earley, „An efiicient context-free parsing“.
[b] **Erklärweise** wurde von der Webseite *Earley parser* übernommen.
[c] *Earley Parser*.
[d] $L(G_{Parse})$ ist die **Sprache**, welche durch die **Konkrete Grammatik** G_{Parse} beschrieben wird.
[e] Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[f] Da man bei 0 **anfängt** zu zählen.
[g] Wobei S das **Startsymbol** ist.

Definition 5.12: Liveness Analyse

*Findet heraus, welche **Variablen** in welchen **Regionen** eines Programmes **verwendet** werden.*[a]

[a] G. Siek, *Essentials of Compilation*.

Definition 5.13: Live Variable

*Eine **Variable**, deren momentaner Wert **später** im Programmablauf noch **verwendet** wird. Man sagt auch die Variable ist **live**.*[a b]

[a] Es gibt leider **kein** allgemein gebräuchliches **deutsches** Wort für **Live Variable**.
[b] G. Siek, *Essentials of Compilation*.

Definition 5.14: Graph Coloring

*Problem, bei dem den **Knoten** eines Graphen*[a] *__Zahlen__*[b] *zugewiesen werden sollen, sodass **keine** zwei **adjazente Knoten** die **gleiche Zahl** haben und **möglichst wenige** unterschiedliche Zahlen gebraucht werden.*[c d]

[a] In Bezug zu Compilerbau ein **Ungerichteter Graph**.
[b] Bzw. **Farben**.
[c] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Graph Coloring**.
[d] G. Siek, *Essentials of Compilation*.

Definition 5.15: Interference Graph

*Ein **ungerichteter Graph** mit **Variablen** als **Knoten**, der eine **Kante** zwischen zwei Variablen hat, wenn es sich bei beiden Variablen **zu dem Zeitpunkt** um **Live Variablen** (Definition 5.13) handelt. In Bezug auf **Graph Coloring** (Definition 5.14) bedeutet eine **Kante**, dass diese zwei Variablen **nicht** die **gleiche Zahl**[a] zugewiesen bekommen dürfen, weil sie zum selben Zeitpunkt **live** sind und daher **nicht** der **gleichen Location** zugewiesen werden dürfen.*[b]

[a] Bzw. **Farbe**.
[b] G. Siek, *Essentials of Compilation*.

Definition 5.16: Kontrollflussgraph

***Gerichteter Graph**, der den **Kontrollfluss** (Definition 5.16) eines Programmes beschreibt.*[a]

[a] G. Siek, *Essentials of Compilation*.

Definition 5.17: Kontrollfluss

*Die **Reihenfolge** in der z.B. **Anweisungen**, **Funktionsaufrufe** usw. eines Programmes ausgewertet werden*[a].

[a]Man geht hier von einem Programm in einer **Imperativen Programmiersprache** aus.

Definition 5.18: Kontrollflussanalyse

***Analyse** des **Kontrollflusses** (Defintion 5.17) eines **Programmes**, um herauszufinden zwischen welchen Teilen des Programms **Daten ausgetauscht** werden und welche **Abhängigkeiten** sich daraus ergeben.*

*Der **simpelste Ansatz** ist es, in einen Kontrollflussgraph **iterativ** einen Algorithmus[a] anzuwenden, bis sich an den Werten der Knoten **nichts** mehr **ändert**[b].*[c]

[a]Im Bezug zu Compilerbau die **Liveness Analyse**.
[b]Bis diese sich **stabilisiert** haben
[c]G. Siek, *Essentials of Compilation*.

Definition 5.19: Two-Space Copying Collector

*Ein **Garbabe Collector**, bei dem der **Heap** in `FromSpace` und `ToSpace` unterteilt wird. Bei **nicht ausreichendem** Speicherplatz auf dem **Heap**, werden alle Variablen, die in Zukunft **noch verwendet** werden vom `FromSpace` zum `ToSpace` kopiert. Der **aktuelle** `ToSpace` wird danach zum **neuen** `FromSpace` und der **aktuelle** `FromSpace` wird danach zum **neuen** `ToSpace`.*[a]

[a]G. Siek, *Essentials of Compilation*.

Bootstrapping

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ (Defintion 5.20) zu schreiben. Durch einen **Self-Compiling Compiler** kann die **Unabhängigkeit** von der Programmiersprache L_{Python}, in welcher der **PicoC-Compiler** $C^{Python}_{PicoC\_RETI}$ bisher implementiert ist erreicht werden. Des Weiteren kann die **Unabhängigkeit** von einer **anderen Maschine**, die bisher immer für das **Cross-Compiling** (Definition 2.6) notwendig war erreicht werden. Mittels **Bootrapping** wird aus dem **PicoC-Compiler** ein „richtiger Compiler“[1] für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

Anmerkung

Im Folgenden wird ein **voll ausgeschriebenes Compilerkürzel** als $C^{I\_j}_{E\_A\_k\_min}$ geschrieben, wobei:

- $C^{I\_j}_{\mathbf{E}\_A\_k\_min} \hat{=}$ **E**ingabesprache.
- $C^{I\_j}_{E\_\mathbf{A}\_k\_min} \hat{=}$ **A**usgabesprache.
- $C^{I\_j}_{E\_A\_\mathbf{k}\_min} \hat{=}$ Version k der **E**ingabesprache.
- $C^{\mathbf{I}\_j}_{E\_A\_k\_min} \hat{=}$ **I**mplementierungssprache bzw. Maschinensprache mit welcher der Compiler läuft.
- $C^{I\_\mathbf{j}}_{E\_A\_k\_min} \hat{=}$ Version j der **I**mplementierungssprache.

[1]Ein **üblicher Compiler**, wie ihn ein Programmierer verwendet, wie der **GCC** oder **Clang** läuft üblicherweise selbst auf der Maschine für welche er kompiliert.

- $C^{I\_j}_{E\_A\_k\_min}$ $\hat{=}$ **Min**imaler Compiler.

bedeuten.

Definition 5.20: Self-compiling Compiler

*Compiler $C^{E}_{E\_M}$, der in der Eingabesprache L_E **implementiert** ist, die er **kompiliert**. Also ein Compiler, der sich **selbst** kompilieren könnte.*[a]

[a] J. Earley und Sturgis, „A formalism for translator interactions“.

Will man für eine Maschine M_{RETI}, auf der bisher **keine anderen** Programmiersprachen mittels **Bootstrapping** (Definition 5.21) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ in der Sprache L_{PicoC} **implementiert**, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**[2] reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ auf der **Maschine** M_{PicoC} zu **kompilieren** bereits einen **kompilierten Self-compiling Compiler** $C^{RETI}_{PicoC\_RETI}$. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Eine Möglichkeit diese **zirkulare Abhängigkeit** zu brechen, wäre, dass man den **Cross-Compiler** $C^{Python}_{PicoC\_RETI}$, den man bereits in der Programmiersprache L_{Python} implementiert hat auf einer anderen Maschine M_{other} dazu nutzt, um den **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ für die Maschine M_{RETI} zu kompilieren bzw. zu **bootstrappen**. Der **Cross-Compiler** $C^{Python}_{PicoC\_RETI}$ stellt in diesem Fall einen **Bootstrap Compiler** (Definition 5.22) dar. Den **kompilierten Compiler** $C^{RETI}_{PicoC\_RETI}$ kann man dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} **kopieren**[3]. In Abbildung 5.2 ist das ganze in einem **T-Diagramm** (siehe Unterkapitel 2.1.1) dargestellt.

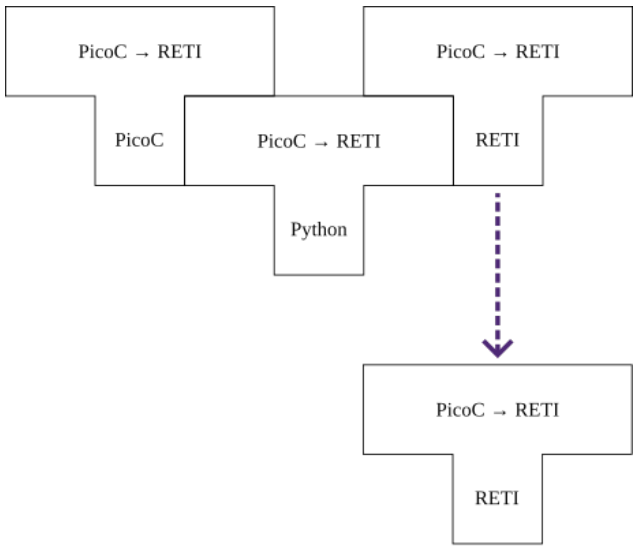


Abbildung 5.2: *Cross-Compiler als Bootstrap Compiler.*

[2] Beschreibt die Situation, wenn ein System sich **selbst** als **Abhängigkeit** hat, damit es überhaupt einen **Anfang** für dieses System geben kann. Dafür steht das Problem mit der **Henne** und dem **Ei** sinnbildlich, da hier die Frage ist, wie das ganze seinen Anfang genommen hat, da beides **zirkular** voneinander abhängt.

[3] Im Fall, dass auf der Maschine M_{RETI} die Programmiersprache L_{Python} bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$ auch mithife des **Cross-Compilers** $C^{Python}_{PicoC\_RETI}$ als **externe Entität** auf der Maschine M_{RETI} **selbst** kompiliert werden.

Definition 5.21: Bootstrapping

Wenn man einen ***Self-compiling Compiler*** $C^{E}_{E\_M}$ *einer Eingabesprache* L_E *auf einer* ***Maschine*** *mit der* ***Maschinensprache*** L_M *zum laufen bringt*[abcd]*. Dabei ist die Art von* ***Bootstrapping*** *in 5.21.1 nochmal gesondert* ***hervorzuheben****.*

5.21.1: *Wenn man die* ***aktuelle Version*** *eines* ***Self-compiling Compilers*** $C^{E\_i}_{E\_M\_i}$ *der Eingabesprache* L_{E_i} *mithilfe von* ***früheren Versionen*** *seiner selbst für eine Maschine mit der* ***Maschinensprache*** L_M ***kompiliert****. Man schreibt also z.B. die* ***aktuelle Version*** *des* ***Self-compiling Compilers*** $C^{E\_i-1}_{E\_M\_i}$ *in der Sprache* L_{E_i-1}*, welche von der* ***früheren Version*** *des* ***Self-compiling Compilers*** $C^{E\_i-1}_{E\_M\_i-1}$*, genuer dem* ***kompilierten Self-compiling Compiler*** $C^{M}_{E\_M\_i-1}$ *kompiliert wird.*

Man erhält so einen ***kompilierten Self-compiling Compiler*** $C^{M}_{E\_M\_i}$*, der dazu in der Lage wäre den* ***Self-compiling Compiler*** $C^{E\_i}_{E\_M\_i}$ *zu kompilieren, der in* ***selben Version*** *der Eingabesprache* L_E *implementiert ist, die er kompiliert. Man schafft es so* ***iterativ*** *immer umfangreichere Compiler zu erstellen.*[efg]

[a] Z.B. mithilfe eines **Bootstrap Compilers**.

[b] Der Begriff hat seinen **Ursprung** in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügengeschichten des Freiherrn von Münchhausen** bekannten **Redewendung** „sich am eigenen Schopf aus dem Sumpf ziehen“entspricht.

[c] Hat man einmal einen solchen **Self-compiling Compiler** $C^{E}_{E\_M}$ auf der Maschine mit der **Maschinensprache** L_M zum laufen gebracht, so kann man den Compiler auf dieser Maschine **weiterentwickeln**, **ohne** von **externen Entitäten**, wie einer bestimmten Programmiersprache L_O, in welcher der Compiler oder eine **frühere Version** des Compilers ursprünglich implementiert war **abhängig** zu sein.

[d] Einen Compiler in der Sprache zu implementieren, die er selbst kompiliert und diesen Compiler dann sich **selbst kompilieren** zu lassen, kann eine gute **Probe aufs Exempel** darstellen, um zu **prüfen**, ob der Compiler auch wirklich **funktioniert**.

[e] Es ist hierbei theoretisch **nicht notwendig** den **letzten Self-compiling Compiler** $C^{E\_i-1}_{E\_M\_i-1}$ für das Kompilieren des **neuen Self-compiling Compilers** $C^{E\_i}_{E\_M\_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C^{E\_i-3}_{E\_M\_i-3}$ auch bereits alle Funktionalitäten, die beim Implementieren des **Self-compiling Compilers** $C^{E\_i}_{E\_M\_i}$ verwendet wurden kompilieren kann.

[f] Der Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den **Hauptspeicher** geladen wird und darauf aufbauend von diesem dann das **Betriebssystem**, welches bei Bedarf dann **Systemsoftware** (Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber) und **Anwendungssoftware** (Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt) lädt.

[g] J. Earley und Sturgis, „A formalism for translator interactions“.

Definition 5.22: Boostrap Compiler

Compiler $C^{O}_{E\_M}$*, der es ermöglicht einen* ***Self-compiling Compiler*** $C^{E}_{E\_M}$ ***zu boostrapen*** *(Definition 5.21). Hierzu wird der* ***Self-compiling Compiler*** $C^{E}_{E\_M}$ *mit dem* ***Bootstrap Compiler*** $C^{O}_{E\_M}$ ***kompiliert***[a]*. Der* ***Bootstrap Compiler*** $C^{O}_{E\_M}$ *stellt eine* ***externe Entität*** *dar, die es ermöglicht die* ***zirkulare Abhängigkeit*** *zu brechen, dass initial ein* ***kompilierter Self-compiling Compiler*** $C^{M}_{E\_M}$ *bereits vorliegen müsste, damit der* ***Self-compiling Compiler*** $C^{E}_{E\_M}$ *sich selbst kompilieren könnte.*[b]

[a] Dabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.

[b] Thiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{PicoC\_RETI}$, der einen **Minimalen Compiler** (Definition 5.23) für eine **Teilmenge** der Programmiersprache L_C darstellt, könnte man auch noch weitere Funktionalitäten der Programmiersprache L_C mittels **Bootstrapping** implementieren[4].

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem

[4] Natürlich könnte man aber auch einfach den **Cross-Compiler** $C^{Python}_{PicoC\_RETI}$ um weitere Funktionalitäten von L_C erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache L_{Python}.

Minimalen Compiler $C^{PicoC}_{PicoC\_RETI}$, wie in Subdefinition 5.21.1 den Minimalen Compiler **schrittweise** zu einem immer umfangreicheren Compiler **weiterentwickelt**. In Abbildung 5.3 ist das ganze in einem **T-Diagramm** (siehe Unterkapitel 2.1.1) dargestellt.

Anmerkung

Einen ersten **Minimalen Compiler** $C^{O}_{E\_M\_min}$ der Sprache L_E für eine Maschine mit der **Maschinensprache** L_M kann man entweder mittels eines **externen Bootstrap Compilers** $C^{M}_{O\_M}$ kompilieren zu $C^{M}_{E\_M\_min}$ oder man implementiert ihn direkt in der **Maschinensprache** L_M oder wenn ein **Assembler** $\mathbb{A}^{M}_{A\_M}$ vorhanden ist, in der **Assemblersprache** L_A.

Dies ist nur beim **allerersten Minimalen Compiler** $C^{O}_{E\_M\_1\_min}$ für eine allererste **abstrakte Programmiersprache** L_{E_1} mit z.B. Schleifen, Verzweigungen usw. notwendig. Ansonsten kann man immer eine **Kette**, die beim **allerersten Minimalen Compiler** $C^{O}_{E\_M\_1\_min}$ anfängt fortführen, in der immer ein Compiler einen **anderen** Compiler **kompiliert** bzw. einen **ersten** Minimalen Compiler **kompiliert** und dieser **Minimale Compiler** dann eine **umfangreichere** Version von sich kompiliert usw.

Definition 5.23: Minimaler Compiler

Compiler $C^{O}_{E\_M\_min}$*, der nur die* ***notwendigsten Funktionalitäten*** *einer Wunschsprache* L_E*, wie* ***Schleifen, Verzweigungen*** *kompiliert, die für die Implementierung eines* ***Self-compiling Compilers*** $C^{E}_{E\_M}$ *oder einer* ***bestimmten Version*** $C^{E\_i}_{E\_M\_i}$ *dieses Compilers wichtig sind.*[a][b]

[a] Den **PicoC-Compiler** $C^{Python}_{PicoC\_RETI}$ könnte man auch als einen **Minimalen Compiler** ansehen.
[b] Thiemann, „Compilerbau“.

Anmerkung

Auch wenn ein **Self-compiling Compiler** $C^{E\_i}_{E\_M\_i}$ in der Subdefinition 5.21.1 selbst in einer früheren Version L_{E_i-1} der Programmiersprache L_{E_i} geschrieben wird, wird dieser nicht mit $C^{E\_i-1}_{E\_M\_i}$ bezeichnet, sondern mit $C^{E\_i}_{E\_M\_i}$. Es geht bei **Self-compiling Compilern** darum, dass diese zwar in der Subdefinition 5.21.1 eine **frühere Version** $C^{E_{i-1}}_{E_{i-1}}$ nutzen, um sich selbst kompilieren, aber auch in der Lage sind sich **selber** zu **kompilieren**.

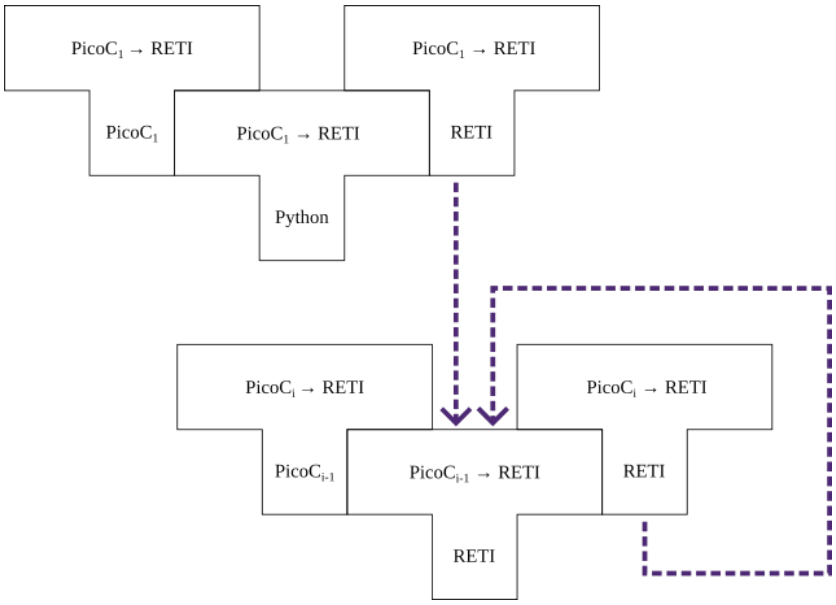


Abbildung 5.3: *Iteratives Bootstrapping.*

Literatur

Online

- *2.1.7 Vorrangregeln und Assoziativität*. URL: https://www.tu-chemnitz.de/urz/archiv/kursunterlagen/C/kap2/vorrang.htm (besucht am 05. 09. 2022).
- *A-Normalization: Why and How (with code)*. URL: https://matt.might.net/articles/a-normalization/ (besucht am 23. 07. 2022).
- *ANSI C grammar (Lex)*. URL: https://www.quut.com/c/ANSI-C-grammar-l-2011.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Lex) old*. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Yacc)*. URL: https://www.quut.com/c/ANSI-C-grammar-y.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Yacc) old*. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html (besucht am 15. 08. 2022).
- *ANTLR*. URL: https://www.antlr.org/ (besucht am 31. 07. 2022).
- *Bäume*. URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17. 07. 2022).
- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 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).
- *Earley Parser*. URL: https://rahul.gopinath.org/post/2021/02/06/earley-parsing/ (besucht am 20. 06. 2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10. 05. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 07. 2022).
- *Grammar Reference — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/grammar.html (besucht am 31. 07. 2022).

- *Grammar: The language of languages (BNF, EBNF, ABNF and more).* URL: https://matt.might.net/articles/grammars-bnf-ebnf/ (besucht am 30. 07. 2022).
- *History - GCC Wiki.* URL: https://gcc.gnu.org/wiki/History (besucht am 06. 08. 2022).
- *Home - Neovim.* URL: http://neovim.io/ (besucht am 04. 08. 2022).
- *JSON parser - Tutorial — Lark documentation.* URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09. 07. 2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13. 04. 2022).
- *Parsers — Lark documentation.* URL: https://lark-parser.readthedocs.io/en/latest/parsers.html (besucht am 20. 06. 2022).
- *Transformers & Visitors — Lark documentation.* URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09. 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 Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 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).
- *What is Top-Down Parsing?* URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22. 06. 2022).

Bücher

- G. Siek, Jeremy. *Essentials of Compilation.* 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).
- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand.* 1. Aufl. Wiley, 20. Nov. 2012.
- Nystrom, Robert. *Parsing Expressions · Crafting Interpreters.* Genever Benning, 2021. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09. 07. 2022).

Artikel

- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: 10.1145/355598.362740.

- Earley, Jay. „An efiicient context-free parsing“. In: 13 (1968). URL: https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf (besucht am 10. 08. 2022).

Vorlesungen

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Nebel, Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09. 07. 2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03. 08. 2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19. 07. 2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23. 07. 2022).
- *Earley parser*. In: *Wikipedia*. Page Version ID: 1090848932. 31. Mai 2022. URL: https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=1090848932 (besucht am 15. 08. 2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).
- *Naming convention (programming)*. In: *Wikipedia*. Page Version ID: 1100066005. 24. Juli 2022. URL: https://en.wikipedia.org/w/index.php?title=Naming_convention_(programming)&oldid=1100066005 (besucht am 30. 07. 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).

- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: https://github.com/lark-parser/lark (besucht am 31. 07. 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).