Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
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

asdf

asdf

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Motivation

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

Compiler, wie der **GCC**[1] oder **Clang**[2] werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert[3] zu **Maschinencode** zu kompilieren.

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

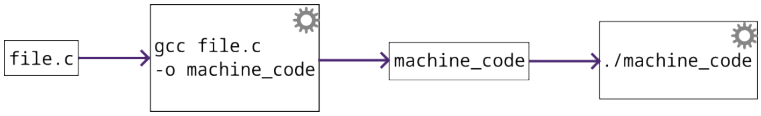


Abbildung 1.1: *Schritte zum Ausführen eines Programmes mit dem GCC*

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

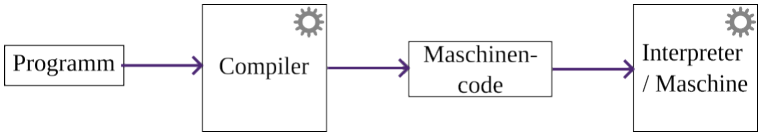


Abbildung 1.2: *Stark vereinfachte Schritte zum Ausführen eines Programmes*

[1] *GCC, the GNU Compiler Collection - GNU Project*.
[2] *clang: C++ Compiler*.
[3] Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C.
[4] Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objectcode** (Definition 5.3) kompiliert wird. Das Kompilieren zu **Objectcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

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

Die konkrette **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC}, welche eine Untermenge der Sprache L_C ist[5] in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zu Grunde liegt und im Unterkapitel 1.2 wird näher auf die die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll.

1.1 RETI-Architektur

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

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

Anmerkung

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

Die **Register** der **RETI-Architektur** werden in Tabelle 1.1 aufgezählt und erläutert. Die **Maschinenbefehle** und **Datenpfade** der **RETI-Architektur** sind im Kapitel Appendix dokumentiert, da diese **nicht explizit** zum **Verständnis** der späteren Kapitel notwendig sind, aber zum **vollständigen Verständnis** notwendig sind, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** L_{RETI} ist durch Grammatik 3.3.6 und Grammatik 3.3.7 zusammengenommen beschrieben. Für genauere **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

[5] Die der **GCC** kompilieren kann.

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

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

Tabelle 1.1: *Präzidenzregeln von PicoC*

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

Der Abschnitt für die Globalen Statischen Daten ist allgemein dazu da Daten zu beherbergen, die für den Rest der Programmausführung global zugänglich sein sollen, aber auch für die Lokalen Variablen der main-Funktion. Das DS-Register markiert den Anfang des Datensegments und damit auch den Anfang der Globalen Statischen Daten und kann als relativer Orientierungspunkt beim Zugriff und Abspeichern Globaler Statischer Daten dienen. Das CS-Register wird als relativer Orientierungspunkt genutzt, an dem die Ausführung von RETI-Programmen startet und zur Bestimmung der relativen Startadresse, an welcher der RETI-Code einer bestimmten Funktion anfängt.

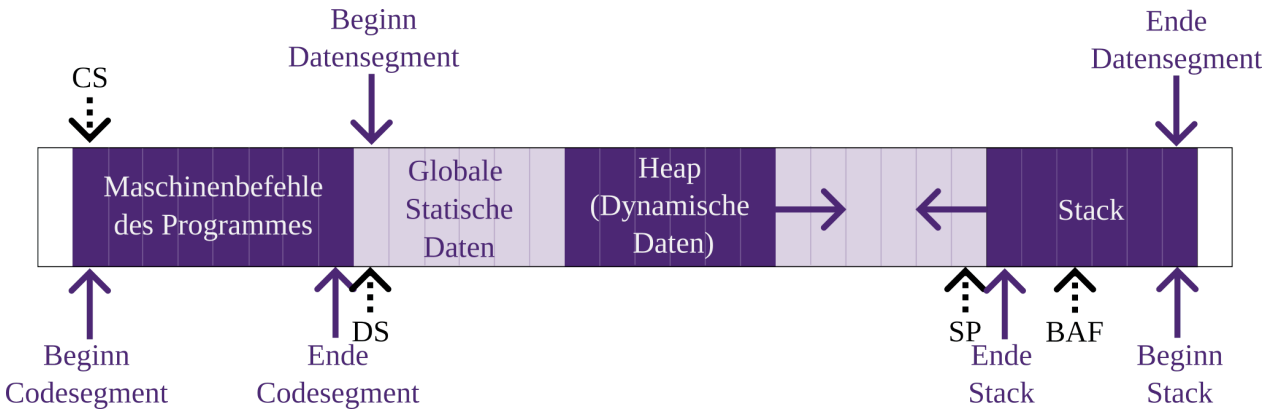


Abbildung 1.3: *Speicherorganisation*

Die RETI-Architektur nutzt 3 verschiedene **Peripheriegeräte**, **EPROM**, **UART** und **SRAM**, die über eine **Memory Map**[6] den über die **Datenpfade** der RETI-Architektur 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, da nach der **Memory Map** der erste **Adressraum** von 0 bis $2^{30} - 1$ dem **EPROM** zugeordnet ist und das `PC`-Register **initial** den Wert 0 hat, also als **erstes** das Programm ausgeführt wird, welches an **Adresse** 0 im EPROM anfängt.

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

An letzter Stelle muss der **SRAM**[12] erwähnt werden, bei dem es sich um den **Hauptspeicher** der **RETI-CPU** handelt. Der **Zugriff auf den Hauptspeicher** ist deutlich schneller als z.B. auf ein **externes Speichermedium**, aber **langsamer** als der **Zugriff auf Register**.

1.2 Die Sprache PicoC

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

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

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

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

Der **Aufbau** der **Programmiersprache** L_C ist durch Grammatik 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, da L_{PicoC} eine Untermenge von L_C ist und welche in der Implementierung des **PicoC-Compilers** in Kapitel Implementierung noch eine **wichtige Rolle** spielen werden im Folgenden genauer erläutert. Im Folgenden wird immer von der Programmiersprache L_{PicoC} gesprochen, da es in dieser Bachelorarbeit um diese geht und die folgenden Beispiele für die Ausgaben alle mithilfe des **PicoC-Compilers** und **RETI-Interpreters kompiliert** bzw **ausgeführt** wurden, aber selbiges gilt genauso für L_C aus bereits erläutertem Grund.

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

Definition 1.1: Imperative Programmierung

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

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

Definition 1.2: Strukturierte Programmierung

*Wenn ein Programm anstelle von z.B. `goto label`-Statements **Kontrollstruturen**, wie z.B. `if(cond) { } else { }`, `while(cond) { }` usw. verwendet, welche dem Programmcode **mehr Struktur** geben, weil die **Auswahl** zwischen Statements und die **Wiederholung** von Statements eine **klare und eindeutige Struktur** hat, welche bei Umsetzung mit einem `goto label`-Statement **nicht** so **eindeutig erkennbar** wäre und auch **nicht** umbedingt immer **gleich aufgebaut** wäre.*[a]

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

Definition 1.3: Prozedurale Programmierung

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

[14] Werden später im Kapitel 3 genauer erklärt.

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

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

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

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

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

Code 1.1: *Beispiel für Spiralregel*

```
1 42
```

Code 1.2: *Ausgabe von Beispiel für Spiralregel*

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

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

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

Code 1.3: *Beispiel für unterschiedliche Ausführung*

```
1 42 42
```

Code 1.4: *Ausgabe des Beispiels für unterschiedliche Ausführung*

Eine weitere interessante Eigenheit, die tätsächlich nur in der **Untermenge** von L_C, die L_{PicoC} darstellt gültig ist[15], ist dass die Operationen `<var>[0][1]` und `*(*(<var>+0)+1)` aus Code 1.3 und Code 1.5 komplett **austauschbar** sind. Die Ausgabe in Code 1.4 ist folglich **identisch** zur Ausgabe in Code 1.6.

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

Code 1.5: *Beispiel mit Dereferenzierungsoperator*

```
1 42 42
```

Code 1.6: *Ausgabe des Beispiels mit Dereferenzierungsoperator*

In der Programmiersprache L_{PicoC} werden alle **Argumente** bei einem Funktionsaufruf nach der **Call By Value**-Strategie (Definition 1.4) übergeben. Ein Beispiel hierfür ist in Code 1.7 zu sehen. Hierbei wird ein **Verbund** `struct st copyable_ar = {.ar={314, 314}};`[16] an die Funktion `fun` übergeben. Hierzu wird der **Verbund** in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopiert** und an den Parameter `fun` gebunden.

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

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

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

Definition 1.4: Call by Value

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

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

[a] Bast, „Programmieren in C“.

```
1  struct st {int ar[2];};
2
3  int fun(struct st copyable_ar) {
4    copyable_ar.ar[1] = 42;
5  }
6
7  void main() {
8    struct st copyable_ar = {.ar={314, 314}};
9    print(copyable_ar.ar[1]);
10   fun(copyable_ar);
11   print(copyable_ar.ar[1]);
12 }
```

Code 1.7: *Beispiel dafür, dass Struct kopiert wird*

```
1 314 314
```

Code 1.8: *Ausgabe von Beispiel, dass Struct kopiert wird*

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

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

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

Definition 1.5: Call by Reference

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

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

[a] Bast, „Programmieren in C“.

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

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

```
1 314 42
```

Code 1.10: *Ausgabe von Beispiel dafür, dass Zeiger auf Feld übergeben wird*

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

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

Definition 1.6: Funktionsprototyp

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

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

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

Definition 1.7: Deklaration

*Der **Datentyp** bzw. **Prototyp** einer **Variablen** bzw. **Funktion**, sowie der **Bezeichner** dieser **Variable** bzw. **Funktion** wird dem **Compiler** mitgeteilt.*[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. D. P. Scholl, „Einführung in Embedded Systems“.

Definition 1.8: Definition

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

[a] Im Fall des PicoC-Compilers in den **Globalen Statischen Daten** oder auf dem **Stack**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. D. P. Scholl, „Einführung in Embedded Systems“.

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

Code 1.11: *Beispiel für Deklaration und Definition*

```
1 42
```

Code 1.12: *Ausgabe von Beispiel für Deklaration und Definition*

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

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

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

Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)

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

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

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

Code 1.13: *Beispiel für Sichtbarkeitsbereichs*

```
1 4 2
```

Code 1.14: *Ausgabe von Beispiel für Sichtbarkeitsbereichs*

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinensprache L_{RETI} die **Syntax** und **Semantik** der Sprache L_C identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache L_{PicoC} im Vergleich zu z.B. dem **GCC**[17] ohne merklichen Unterschied[18] komplieren können.

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

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

[17] 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}.

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

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter Link[19] zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter **„Getting Started“** ein kleines **Einführungstutorial** verlinkt. Unter **„Usage“** ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter Link[20] zu finden.

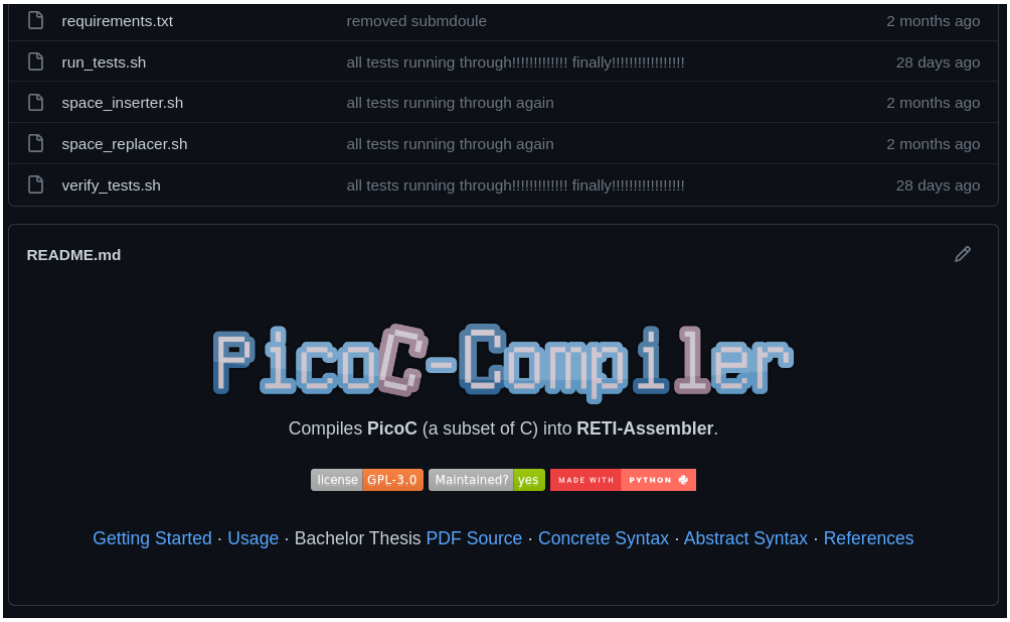


Abbildung 1.4: *README.md im Github Repository der Bachelorarbeit*

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schrifftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter Link[21] zu finden. Die **PDF** der Schrifftliche Ausarbeitung der Bachleorararbeit wird aus dem **Latexquellcode**, welcher unter Link[22] veröffentlicht ist automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[23] zu finden[24]. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**[25] erstellt. Falls Interesse besteht **Grafiken** aus der Schrifftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den `.svg`-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[26] zu finden.

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

Um die verschiedenen **Aspekte** dieser Schrifftlichen Ausarbeitung der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel Motivation werden Codebeispiele zur **Anschauung** verwendet und mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters Ausgaben** erzeugt, die in dieses Dokument **eingelesen** wurden. In Kapitel Implementierung werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** gezeigt[27].

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[28] finden und mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[29].

1.5.1 Still der Schrifftlichen Ausarbeitung

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

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas geanu so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht[30] die Frage des **„warum“**.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**[31] verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **konsistent** sind, wie auch das in Ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernomnen, so wurde die Definition oder der entsprechende Teil in **„Anführungszeichen“** gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox**, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

1.5.2 Aufbau der Schrifftlichen Arbeit

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: Motivation, Einführung, Implementierung und Ergebnisse und Ausblick.

Im momentanen Kapitel Motivation, wurde ein kurzer **Einstieg** in das Thema **Compilerbau** gegeben und die **zentrale Aufgabenstellung** der Bachelorarbeit erläutert, sowie auf **Schwerpunkte** und kleinere **Teilprobleme**, die eines **besonderen Fokus** bedürfen eingegangen.

Im Kapitel Einführung werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische**

[27] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[28] `https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples`.
[29] 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.
[30] Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[31] Das ist ein **Beispiel** für eine **Fußnote**.

Vorgehen, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen in Bezug zu Compilern und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

Des Weiteren wurden für **T-Diagramme** und **Formale Sprachen** eigene Unterkapitel erstellt. Für **T-Diagramme** wurde ein eigenes Unterkapitel erstellt, da sie häufig in der Schrifftlichen Ausarbeitung verwendet werden und die **T-Diagramm Notation** nicht allgemein bekannt ist. Für **Formale Sprachen** wurde ein eigenes Unterkapitel erstellt, da für den Gutachter Prof. Dr. Scholl das Thema **Formale Sprachen** eher **fachfremd** ist, aber dieses Thema einige zentrale und wichtige Fachbegriffe besitzt, bei denen es wichtig ist die genaue **Definition** zu haben. Generell wurde im Kapitel **Einführung** versucht an Erklärungen nicht zu sparren, damit aufgrund dessen, dass das Thema eher fachfremd für Prof. Dr. Scholl ist für das Kapitel **Implementierung** keine wichtigen Aspekte unverständlich bleiben.

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

Im Kapitel 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. Zum Schluss wird noch auf **weitere Erweiterungsideen** eingegangen, die auch interessant zu implementieren wären.

Im Kapitel Appendix werden einige **Definitionen** und **Themen** angesprochen, die bei **Interesse** zur **weiteren Vertiefung** da sind und **unabhänging** von den anderen Kapiteln sind. Diese **Themen** und **Defintionen** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Diese **Themen** und **Definitionen** passen nicht ins Kapitel Implementierung, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tuen haben und auch nichts ins **Kapitel** Einführung, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in der Schrifftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition 5.9) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

2 Einführung

2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), 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. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[1] und von **Tests** die **Beziehungen** in 2.2.1 zu belegen (siehe Subkapitel 4.2).

Definition 2.1: Interpreter

Interpretiert *die* ***Befehle*** *bzw.* ***Statements*** *eines Programmes P direkt.*

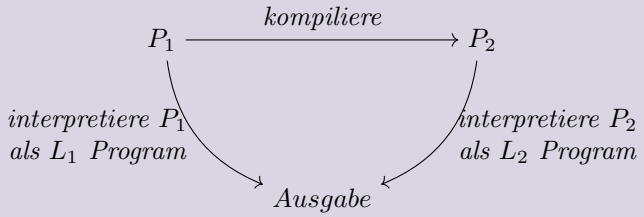
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstrakten Syntaxbaumes*** *(Definition 2.44) und führt je nach Komposition der* ***Knoten*** *des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513).*

Definition 2.2: Compiler

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

Wobei ***Kompilieren*** *meint, dass ein beliebiges Program P_1 in der Sprache L_1 so in die Sprache L_2 zu einem Programm P_2 übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen L_1 und L_2* ***interpretiert*** *werden, die gleiche* ***Ausgabe*** *rauskommt, wie es in Diagramm 2.2.1 dargestellt ist. Also beide Programme P_1 und P_2 die gleiche* ***Semantik*** *(Definition 2.15) haben und sich nur* ***syntaktisch*** *(Definition 2.14) durch die Sprachen L_1 und L_2, in denen sie geschrieben stehen unterscheiden.*[a]



(2.2.1)

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513).*

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

Anmerkung

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C^{o\_j}_{i\_w\_k\_min}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschienensprache L_{B_i} einer Maschiene M_i kompiliert. Fall die Notwendigkeit besteht die **Maschiene** M_i anzugeben, zu dessen **Maschienensprache** L_{B_i} der Compiler kompiliert, wird das als C_i geschrieben. Falls die Notwendigkeit besteht die **Sprache** L_o anzugeben, in der der Compiler selbst geschrieben ist, wird das als C^o geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert (L_{w_k}) oder in der er selbst geschrieben ist (L_{o_j}) anzugeben, wird das als $C^{o\_j}_{w\_k}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 5.18) kann man das als C_{min} schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschienenncode**, der in **Maschienensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 5.5) oder **Cross-Compiler** (Definition 2.5). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition 5.1) voneinander zu unterscheiden.

Definition 2.3: Maschienensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschienenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten 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 Maschienenbefehle sind meist so designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.*[a] [b]

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschienenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschienenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.4) haben.

[b] P. D. C. Scholl, „Betriebssysteme“.

Der **Maschienencode**, denn ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschienenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschiene, die binär arbeitet verständlich sein sollen und nicht für den Programmierer.

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschienencode, der die Maschienenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.4) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung[2].

Definition 2.4: Immediate

***Konstanter Wert**, der als **Teil** eines **Maschienenbefehls** gespeichert ist und dessen **Wertebereich** dementsprechend auch durch die die Anzahl an Bits, die ihm innerhalb dieses **Maschienenbefehls** zur Verfügung gestellt sind, **beschränkter** ist als bei sonstigen Werten innerhalb des Hauptspeichers,*

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

denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.[a]

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

Definition 2.5: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L_w geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschienensprachen** B_1 und B_2 haben.*[a b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C^{Python}_{PicoC}.
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Ein **Cross-Compiler** ist entweder notwendig, wenn eine Zielmaschine M_2 nicht ausreichend **Rechenleistung** hat, um ein Programm in der Wunschsprache L_w selbst **zeitnah** zu kompilieren oder wenn noch kein Compiler C_w für die **Wunschsprache** L_w und andere Programmiersprachen L_o, in denen man Programmieren wollen würde existiert, der unter der **Maschienensprache** B_2 einer Zielmaschine M_2 läuft.[3]

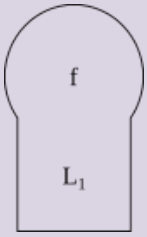
2.1.1 T-Diagramme

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 in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Program (Definition 2.6), einen Übersetzer (Definition 2.7), einen Interpreter (Definition 2.8) und eine Maschiene (Definition 2.9) zusammen.

Definition 2.6: T-Diagram 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

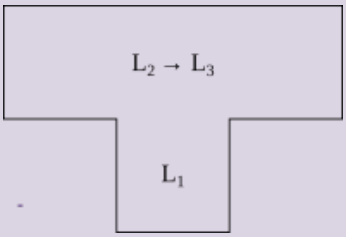
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.6 also reichen einfach eine 1 hinzuschreiben.

[3] Die an vielen Universitäten und Schulen eingesetzen programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren Microcontroller eine **C-ähnliche Sprache** in die Maschienensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst nicht genug Rechenleistung besitzt, um ein Programm selbst zeitnah zu kompilieren.

Definition 2.7: T-Diagram 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 kompiliert.*

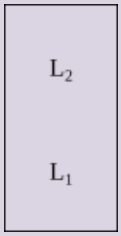
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 2.2.1.*[b]



[a] Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Definition 2.8: T-Diagram Interpreter

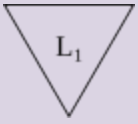
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



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

Definition 2.9: T-Diagram Maschiene

*Repräsentiert eine **Maschiene**, welche ein **Programm** in **Maschienensprache** L_1 ausführt.*[a][b]



[a] Wenn die Maschiene **Programme** in einer höheren Sprache als **Maschienensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschiene**, 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 **Kompostionen** bilden, indem man sie **adjazent** zueinander platziert. Allgemein lässt sich grob sagen, dass **vertikale Adjazents** für **Interpretation** und **horinzontale Adjazents** für **Übersetzung** steht.

Sowohl **horinzontale** als auch **vertikale Adjazents** lassen sich, wie man in den Abbildungen 2.1 und 2.2 erkennen kann zusammenfassen.

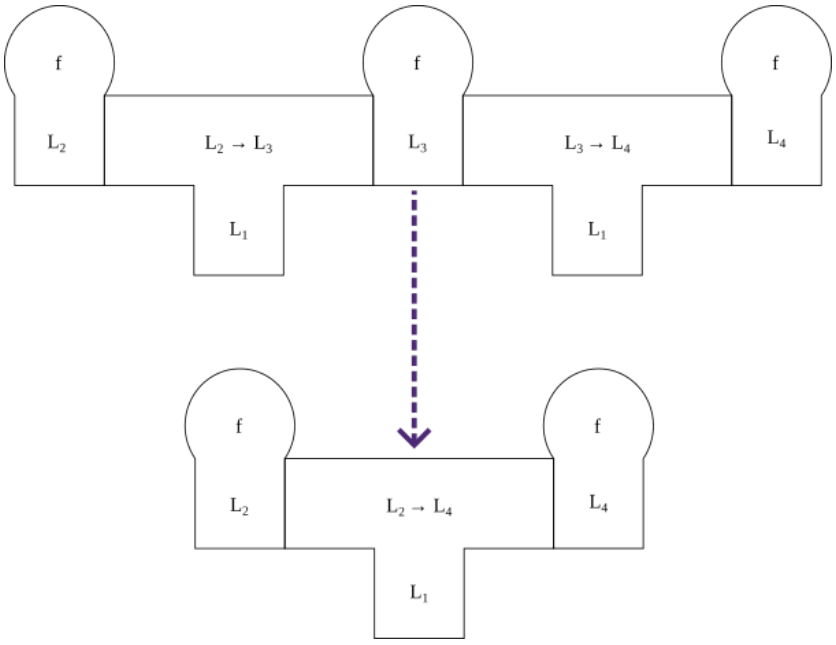


Abbildung 2.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen*

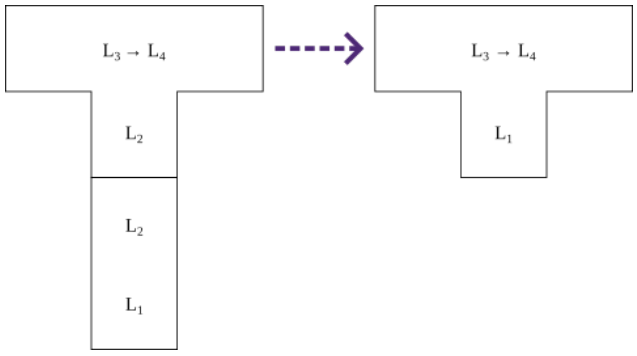


Abbildung 2.2: *Vertikale Interpretierungszwischenschritte zusammenfassen*

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.13) zu tuen, 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**, was die Begriffe **Symbol** (Definiton 2.10), **Alphabet** (Definition 2.11), **Wort** (Definition 2.12) usw. beinhaltet vorher eingeführt zu haben.

Definition 2.10: Symbol

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.11: Alphabet

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 2.10).“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.12: Wort

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.13: Formale Sprache

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.12) über dem **Alphabet** Σ (Definition 2.11).“*[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 2.15) **gleich** bleiben. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 2.16), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 2.14) haben.

Definition 2.14: Syntax

*Die Syntax bezeichnet alles was mit dem **Aufbau** von **Formalen Sprachen** zu tuen hat. Die **Grammatik** einer Sprache, aber auch die in **Natürlicher Sprache** ausgedrückten Regeln, welche den Aufbau von Wörtern einer Formalen Sprache beschreiben, beschreiben die Syntax dieser Sprache. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben*[a]*.*[b]

[a] Z.B. die **Konkrette** und **Abstrakte Syntax**, die später eingeführt werden.
[b] Thiemann, „Einführung in die Programmierung“.

Definition 2.15: Semantik

*Die Semantik bezeichnet alles was mit der **Bedeutung** von **Formalen Sprachen** zu tuen hat.*[a]

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

Definition 2.16: Formale Grammatik

*„Eine Formale Grammatik beschriebt wie **Wörter** einer **Sprache** abgeleitet werden können.“*[a]

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

- $N \mathrel{\hat{=}}$ ***Nicht-Terminalsymbole***.

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

[a] Nebel, „Theoretische Informatik“.
[b] Weil mit ihnen **terminiert** wird.
[c] Kann auch als **Alphabet** bezeichnet werden.
[d] w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.
[e] Bzw. $w, v \in V^* \wedge w \notin \Sigma^*$.

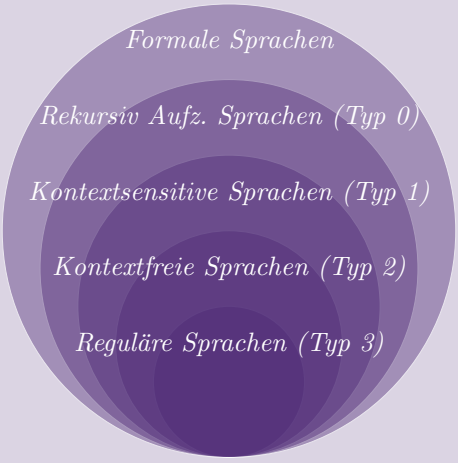
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 2.17) einteilen.

Definition 2.17: Chromsky Hierarchie

Die Chromsky Hierarchie ist 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 dieser Klasse* ***entscheidbar*** *bzw.* ***unentscheidbar*** *sind usw.*

Eine Sprache L_i *ist in der* ***Chromsky Hierarchie*** *vom Typ* $i \in \{0, \ldots, 3\}$*, falls sie von einer Grammatik dieses Typs* i *erzeugt wird.*

Zwischen den Sprachmengen ***benachbarter Klassen*** *in Abbildung 2.17.1 besteht eine* ***echte Teilmengenbeziehung****:* $L_3 \subset L_2 \subset L_1 \subset L_0$*. Jede* ***Reguläre Sprache*** *ist auch eine* ***Kontextfreie Sprache****, aber nicht jede* ***Kontextfreie Sprache*** *ist auch eine* ***Reguläre Sprache****.*[a]



(2.17.1)

[a] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 2.18) und **Kontextfreien Grammatiken** (Definition 2.19) beschrieben werden.

Definition 2.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{2.18.1}$$

*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 2.19: Kontextfreie Grammatik

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

$$A \to v \tag{2.19.1}$$

*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 2.20). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise immer **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 2.20: Wortproblem

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

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

$$\mathbb{1}_L : \Sigma^* \to \{0,1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \tag{2.20.1}$$

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

2.2.1 Ableitungen

Um sicher zu wissen, ob ein Compiler ein **Programm**[4] kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprache** des Compilers **abzuleiten**. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 2.21) und der normalen **Ableitungsrelation** (Definition 2.22) unterschieden.

[4] Bzw. **Wort**.

Definition 2.21: 1-Schritt-Ableitungsrelation

*„Eine **binäre Relattion** $\Rightarrow$ zwischen Wörtern aus $(N \cup \Sigma)^*$, die alle möglichen Wörter $(N \cup \Sigma)^*$ in Relation zueinander setzt, die sich nur durch das **einmalige** Anwenden einer Produktionsregel voneinander unterschieden.*

*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 2.22: 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 aus $(N \cup \Sigma)^*$, welches durch **beliebig häufiges** Anwenden von Produktionsregeln 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**[5] kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 relevant.

Definition 2.23: 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, bei Typ-1- und Typ-0-Grammatiken ist es statt einem **Nicht-Terminalsymbol** die **linke** Seite einer Produktion.*

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

[a] Nebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 2.24) ist[6].

Definition 2.24: 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 **Terminalsymbolen** und **Nicht-Terminalsymbolen** ist.*[a]

[a] *Parsing Expressions · Crafting Interpreters*.

[5] Bzw. **Wort**.
[6] Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (Definition 2.26) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 2.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 2.37) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 2.25: Formaler Ableitungsbaum

*Ist ein Baum, in dem die **Konkrette Syntax** eines **Wortes**[a] nach den **Produktionen** der zugehörigen Grammatik, die angewendet werden mussten um das Wort **abzuleiten** zergliedert **hierarchisch** 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 **Datentstruktur** zur einfachen Weiterverarbeitung umsetzt.*

*Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.16) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** N und die **Blätter** sind entweder **Terminalsymbole** Σ oder das **leere Wort** ε.*[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 den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.3) angegebenen **Produktionen** 2.1 einer ansonsten nicht näher spezifizierten **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ ergibt.

DIG_NO_0	::=	”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”	*L_Lex*
DIG_WITH_0	::=	”0” | *DIG_NO_0*	
NUM	::=	”0” | *DIG_NO_0* *DIG_WITH_0*∗	
add	::=	*add* ”+” *mul* | *mul*	*L_Parse*
mul	::=	*mul* ”∗” *NUM* | *NUM*	

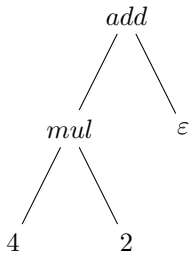
Grammatik 2.1: *Produktionen für Ableitungsbaum in EBNF*

Anmerkung

Werden die **Produktionen** einer Grammatik in z.B. **EBNF** 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.

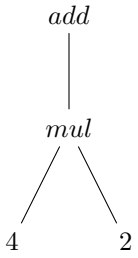
$$add \Rightarrow mul \Rightarrow mul\ ”*”\ NUM \Rightarrow NUM\ ”*”\ NUM \Rightarrow 4\ ”*”\ NUM \Rightarrow 4\ ”*”\ 2 \tag{2.25.1}$$

Bei Ableitungsbäumen gibt es **keine** einheutliche **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 vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[7].



(2.25.3)

Für einen Compiler ist es notwendig, dass die **Konkrette Grammatik** keine **Mehrdeutige Grammatik** (Definition 2.26) ist, denn sonst können unter anderem die **Präzidenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel 3.2.1 an einem Beispiel demonstriert wird.

Definition 2.26: Mehrdeutige Grammatik

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

[a] Alternativ, wenn es für w **mehrere** unterschiedliche **Linksableitungen** gibt.
[b] Nebel, „Theoretische Informatik“.

2.2.2 Präzidenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Konkretten Grammatik** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 2.27) und **Präzidenz** (Definition 2.28) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel 3.2.1 genauer erklärt.

Definition 2.27: Assoziativität

*„Bestimmt, welcher Operator aus einer Reihe **gleicher** Operatoren **zuerst** ausgewertet wird.“*

*Es wird grundsätzlich 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]

[a] *Parsing Expressions · Crafting Interpreters*.

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

Bei **Assoziativität** ist z.B. der **Multitplikationsoperator** `*` ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator** `=` ein Beispiel für einen **rechtsassoziativen** Operator. Dies ist in Abbildung 2.3 mithilfe von Klammern `()` veranschaulicht.

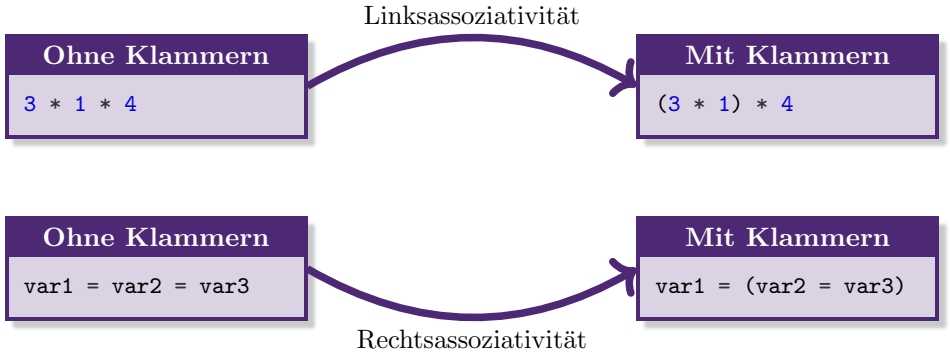


Abbildung 2.3: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität*

Definition 2.28: Präzidenz

*„Bestimmt, welcher Operator **zuerst** in einem Ausdruck, der eine Mischung **verschiedener** Operatoren enthält, ausgewertet wird. Operatoren mit einer **höheren Präzidenz**, werden **vor** Operatoren mit **niedrigerer Präzidenz** ausgewertet.“*[a]

[a] *Parsing Expressions · Crafting Interpreters.*

Bei **Präzidenz** ist die Mischung der Operatoren für **Subraktion** `'-'` und für **Multiplikation** `*` ein Beispiel für den Einfluss von Präzidenz. Dies ist in Abbildung 2.4 mithilfe der Klammern `()` veranschaulicht. Im Beispiel in Abbildung 2.4 ist bei den beiden **Subtraktionsoperatoren** `'-'` nacheinander und dem darauffolgenden **Multitplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzidenz** im Spiel.

Abbildung 2.4: *Veranschaulichung von Präzidenz*

2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.29) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Eingabewort, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.30) matchen, die durch eine **reguläre Grammatik** spezifiziert sind. Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.31) genannt.

Definition 2.29: 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 und **weitergereicht** 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“.

Definition 2.30: Pattern

***Beschreibung** aller möglichen **Lexeme**, die eine Menge $\mathbb{P}_T$ bilden und einem bestimmten **Token** T zugeordnet werden. Die Menge $\mathbb{P}_T$ ist eine möglicherweise unendliche Menge von **Wörtern**, die sich mit den Produktionen einer **regulären Grammatik** G_{Lex} einer **regulären Sprache** L_{Lex} beschreiben lassen [a], die für die Beschreibung eines **Tokens** T zuständig sind.*[b]

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] Thiemann, „Compilerbau“.

Definition 2.31: Lexeme

*Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches von einem **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} erkannt wird.*[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.32) im **Eingebewort** identifziert und **Tokens** T zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** (Definition 2.32) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.32: Lexer (bzw. Scanner oder auch Tokenizer)

*Ein **Lexer** ist eine **partielle** Funktion $lex : \Sigma^* \rightharpoonup (N \times W)^*$, welche ein **Wort** bzw. **Lexeme** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls dieses **Wort** sich unter der **regulären Grammatik** G_{Lex}, der **regulären Sprache** L_{Lex} ableiten lässt bzw. einem der **Pattern** der Sprache L_{Lex} entspricht.*[a]

[a] Thiemann, „Compilerbau“.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die kein **Pattern** eines **Tokens** der Sprache L_{Lex} matchen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** ausgegeben.

Anmerkung

Um Verwirrung verzubäugen ist es wichtig folgende Unterscheidung hervorzuheben:

Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktische Analyse** und der **Code Generierung**, auf diesen verschiedenen Ebenen unterschiedliche Konzepte als Symbole bezeichnet.

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

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

In der Code Generierung sind die **Bezeichner** (Definition 2.33) von **Variablen, Konstanten und Funktionen** die Symbole[a].

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

Definition 2.33: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres **Scopes eindeutig** benennt.*[a][b]

[a] Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.
[b] Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[8] und Tabs `\t` aus dem Eingebewort herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** *. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie `42`, `314` oder `12`. Die **Überbegriffe** bzw. **Tokennamen** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`[9], bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie `if` oder `}` sind die **Tokennamen** bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.34) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[10]. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

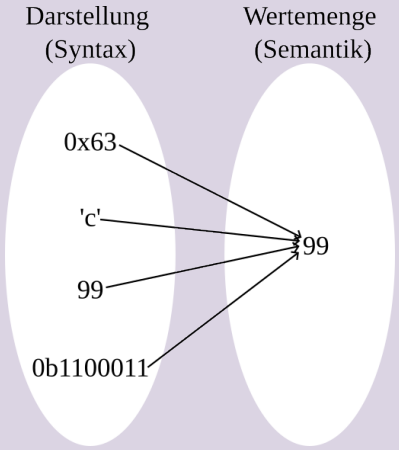
Die **Konkrette Grammatik** G_{Lex}, die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird ist

[8] 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.
[9] Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Knoten haben will, damit unter anderem **mehr Code** in eine Zeile passt.
[10] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[11], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 3.1.1 liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.18 erfüllt. Einzig die Produktion `CHAR ::= "'"ASCII_CHAR"'"` sieht problematisch aus, kann allerdings auch als `{CHAR ::= "'"CHAR2, CHAR2 ::= ASCII_CHAR"'"}` **regulär** ausgedrückt werden[12]. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

Definition 2.34: Literal

*Eine von möglicherweise vielen weiteren **Darstellungsformen** (als **Zeichenkette**) für ein und denselben **Wert** eines **Datentyps**.*[a]



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

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

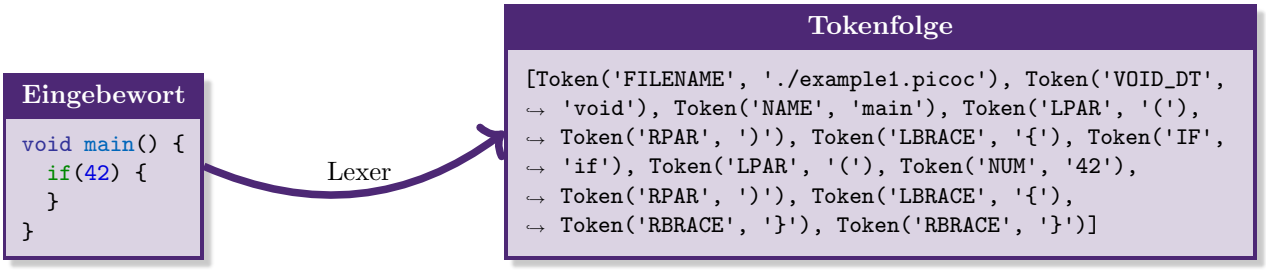


Abbildung 2.5: *Veranschaulichung der Lexikalischen Analyse*

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 öffnende runde Klammern `'('` bzw. öffnende geschweifte Klammern `'{'` es momentan gibt, die noch nicht durch

[11] Man nennt das auch einem **Lookahead** von 1
[12] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

eine entsprechende schließende runde Klammer `')'` bzw. schließende geschweifte Klammer `'}'` geschlossen wurden.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 2.38) ein **Ableitungsbaum** (Definition 2.37) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 2.44) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 2.35: Konkrette Syntax

*Steht für alles, was mit dem **Aufbau** von **Ableitungen** zu tuen hat, also z.B. was für **Ableitungen** mit den **Grammatiken** G_{Lex} und G_{Parse} zusammengenommen möglich sind.*

*Ein **Programm** in seiner **Textrepräsentation**, wie es in einer Textdatei nach den Produktionen der **Grammatiken** G_{Lex} und G_{Parse} abgeleitet steht, bevor man es kompiliert, ist in **Konkretter Syntax** aufgeschrieben.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Um einen kurzen Begriff für die **Grammatik**, welche die **Konkrette Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Konkrette Grammatik** (Definition 2.36) bezeichnet.

Definition 2.36: Konkrette Grammatik

*Grammatik, die eine **Konkrette Syntax** beschreibt.*

Definition 2.37: Ableitungsbaum (bzw. Konkretter Syntaxbaum oder auch engl. Derivation Tree)

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

*Die **Konkrette Syntax** nach der **Ableitungsbaum** konstruiert ist, wird optimalerweise immer so definiert, dass sich möglichst **einfach** aus dem **Ableitungsbaum** ein **Abstrakter Syntaxbaum** konstruieren lässt.*[a]

[a] *JSON parser - Tutorial — Lark documentation*.

Definition 2.38: Parser

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

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.38 verwendet.

[b] *JSON parser - Tutorial — Lark documentation*.

Anmerkung

An dieser Stelle könnte möglicherweise eine Verwirrung enstehen, welche Rolle dann überhaupt ein **Lexer** hier spielt.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines **Parsers**. Der **Lexer** ist ausschließlich für die **Lexikalische Analyse** verantwortlich und entspricht z.B., wenn man bei einem Wanderausflug verschiedenen Insekten entdeckt, dem Nachschlagen in einem Insekten**lexikon** und dem Aufschreiben, welchen Insekten man in welcher **Reihenfolge** 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 Beispiel zurückzukommen, dem Darstellen von **Beziehungen** zwischen den Insektenbegnungen in einer für die **Weiterverarbeitung tauglichen Form**[b].

In der Weiterverarbeitung kann der **Interpreter** das interpretieren und daraus bestimmte Schlüsse ziehen und ein **Compiler** könnte es vielleicht in eine für Menschen leichter entschüsselbare Sprache kompilieren.

[a] Das würde z.B. der Rolle eines **Semikolon** `;` in der Sprache L_{PicoC} entsprechen.
[b] Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beinflussen sich gegenseitig.

Die vom **Lexer** im Eingebewort identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht. Dabei wird in der Grammatik L_{Parse} nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und nicht, welchen konkretten Wert diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.39), denn ein Parser löst das **Wortproblem** (Definition 2.20) für die **Sprache**, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Ableitungsbaum**.

Definition 2.39: Recognizer (bzw. Erkenner)

*Entspricht einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit den Produktionen der **Konkretten Grammatik** einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkretten Grammatik** beschrieben wird oder nicht. Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** (Definition 2.20) bekannt.*[a]

[a] Thiemann, „Compilerbau“.

Anmerkung

Für das **Parsen** gibt es grundsätzlich **drei** verschiedene Ansätze:

- **Top-Down Parsing:** Der **Ableitungsbaum** wird von **oben-nach-unten** generiert, also von der **Wurzel** zu den **Blättern**. Dementsprechend fängt die Generierung des **Ableitungsbaumes** mit dem **Startsymbol** der **Konkretten Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man **Terminalsymbole** hat, die sich zum gewünschten **Eingabewort** abgeleitet haben oder sich herausstellt, dass dieses nicht abgeleitet werden kann.[a]

  Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist, weil das **Eingabewort** von **links nach rechts** eingelesen wird, was gut damit zusammenpasst, dass die **Linksableitung** die **Blätter** von **links-nach-rechts** generiert.

Welche der **Produktionen** für ein **Nicht-Terminalsymbol** angewandt wird, wenn es mehrere **Alternativen** gibt, wird entweder durch **Backtracking** oder durch **Vorausschauen** gelöst.

Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg** (Definition 5.6).

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.24) allerdings nicht möglich, ohne die Konkrette Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der Konkretten Grammatik entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Konkrette Grammatiken parsen zu können, die nicht **LL(k)** (Definition 5.7) sind. Dabei werden meist nach dem Prinzip der **Tiefensuche** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass das **Eingabewort** sich **nicht** mit der verwendeten Konkretten Grammatik ableiten lässt.[b]

Wenn man eine **LL(k)**-Grammatik hat, kann man auf **Backtracking verzichten** und es reicht einfach nur immer k **Token** im Eingabewort **vorauszuschauen**. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.[c]

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

[a] *What is Top-Down Parsing?*
[b] Diese Form von Parsing 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] 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.
[d] *What is Bottom-up Parsing?*
[e] Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstrakte Syntaxbaum** wird mithilfe von **Transformern** (Definition 2.40) und **Visitors** (Definition 2.41) 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 **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.42).

Definition 2.40: Transformer

*Ein Programm, dass von **unten-nach-oben** nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaum** besucht und beim Antreffen eines bestimmten Knoten des **Ableitungsbaumes** je nach Kontext einen entsprechenden Knoten des **Abstrakten Syntaxbaumes** erzeugt und diesen anstelle*

*des Knotens des **Ableitungsbaumes** setzt und so Stück für Stück den **Abstrakten Syntaxbaum** konstruiert.*[a]

[a] *Transformers & Visitors — Lark documentation*.

Definition 2.41: Visitor

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

[a] 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.

[b] *Transformers & Visitors — Lark documentation*.

Definition 2.42: Abstrakte Syntax

*Steht für alles, was mit dem **Aufbau** von **Abstrakten Syntaxbäumen** zu tuen hat, also z.B. was für Arten von **Kompositionen** mit den **Knoten** eines **Abstrakten Syntaxbaumes** möglich sind.*

*Ein **Abstrakter Syntaxbaum**, der zur Kompilierung eines Wortes*[a] *generiert wurde ist nach einer **Abstrakten Grammatik** konstruiert.*

*Jene Produktionen, die in der **Konkretten Grammatik** für die Umsetzung von **Präzidenz** notwendig waren sind in der **Abstrakten Grammatik** abgeflacht. Dadurch sind die **Kompositionen**, welche die Knoten im **Abstrakten Syntaxbaum** bilden können **syntaktisch** meist näher zur Syntax von **Maschinenbefehlen**.*[b]

[a] Z.B. **Programmcode**.

[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Um einen kurzen Begriff für die **Grammatik**, welche die **Abstrakte Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 2.43) bezeichnet.

Definition 2.43: Abstrakte Grammatik

*Grammatik, die eine **Abstrakte Syntax** beschreibt.*

Definition 2.44: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

*Ist ein **compilerinterne Datenstruktur**, welche eine **Abstraktion** eines dazugehörigen **Ableitungsbaumes** darstellt, in dessen Aufbau auch das Erfordernis eines **leichten Zugriffs** und einer **leichten Weiterverarbeitbarkeit** eingeflossen ist. Bei der Betrachtung eines **Knoten**, der für einen Teil des Programms 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.*

*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 Sinn 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[a] möglichst **einfach weiterverarbeitet** werden kann.*

[a] Den verschiedenen **Passes**.

In Abbildung 2.6 wird das Beispiel aus Unterkapitel 2.2.1 fortgeführt, welches den **Arithmetischen Ausdruck** `4 * 2` in Bezug auf die Konkrette Grammatik 2.1, welche die **höhere Präzidenz** der **Multipikation** $*$ berücksichtigt in einem **Ableitungsbaum** darstellt. In Abbildung 2.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**. Das geschieht bezogen auf das Beispiel aus Unterkapitel 2.2.1, indem jegliche Knoten **wewgabstrahiert** werden, die im **Ableitungsbaum** nur existieren, weil die Konkrette Grammatik so umgesetzt ist, dass es nur **einen** einzigen möglichen **Ableitungsbaum** geben kann.

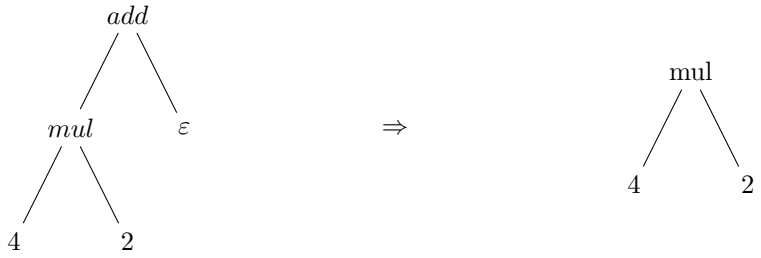


Abbildung 2.6: *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

Die **Baumdatenstruktur** des **Ableitungsbaumes** und **Abstrakten Syntaxbaumes** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Programmes ausführen muss möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.7 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

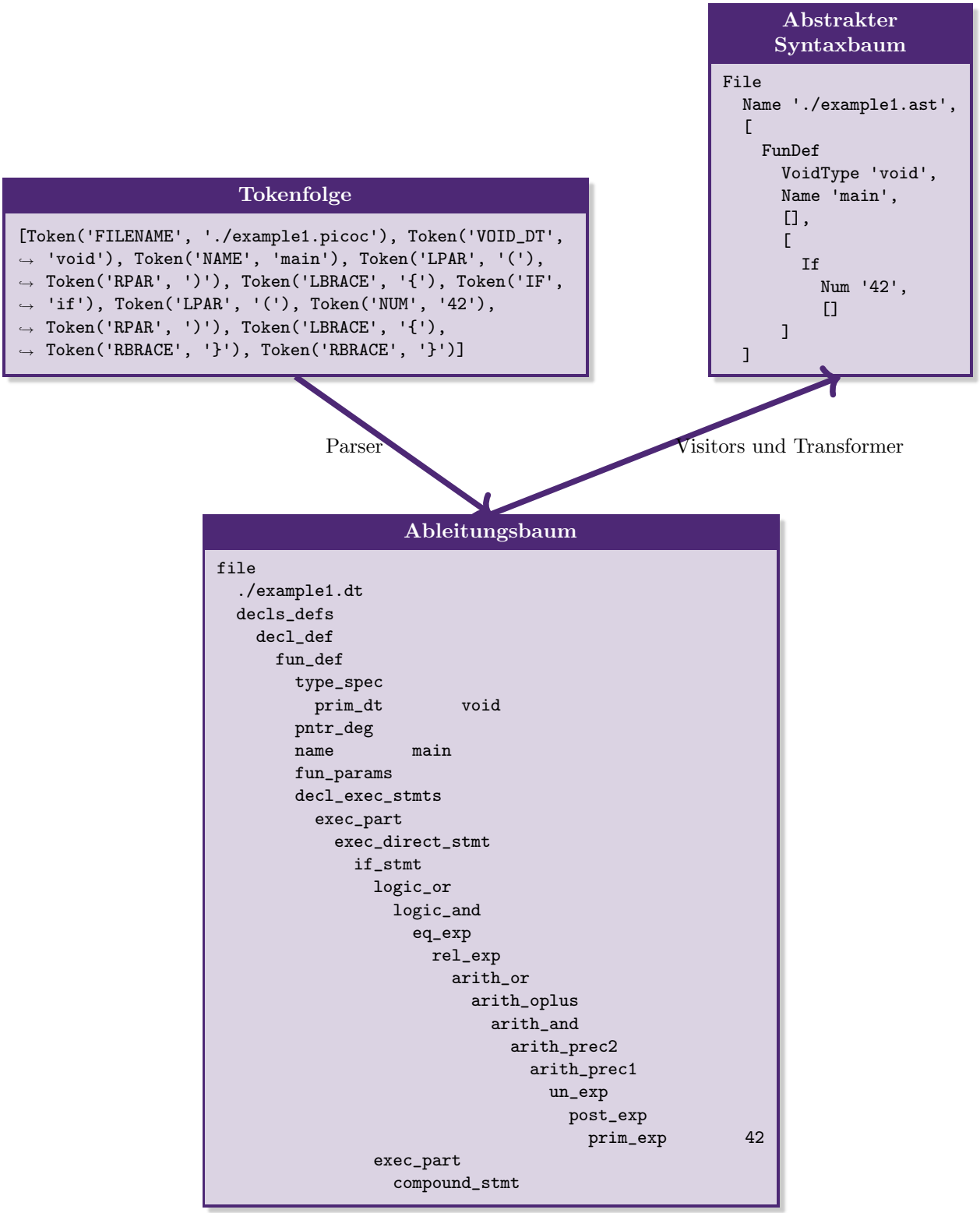


Abbildung 2.7: *Veranschaulichung der Syntaktischen Analyse*

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 2.45) nennt. So wie es auch schon mit dem **Ableitungsbaum** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakten Syntaxbaum** kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

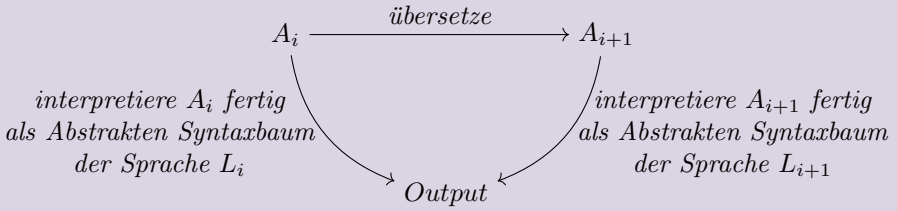
Anmerkung

Man spricht hier von dem **„Abstrakten Syntaxbaum einer Sprache L_1 (bzw. L_2)“** und meint hier mit der Sprache L_1 (bzw. L_2) **nicht** die Sprache, welche durch die **Abstrakte Grammatik**, nach welcher der **Abstrakte Syntaxbaum** abgeleitet ist beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck der **Abstrakte Syntaxbaum** überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Grammatik** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Konvention** wurde aus dem Buch G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

Definition 2.45: Pass

Einzelner Übersetzungsschritt *in einem* ***Kompiliervorgang*** *von einem beliebigen* ***Abstrakten Syntaxbaum*** A_i *einer Sprache* L_i *zu einem* ***Abstrakten Syntaxbaum*** A_{i+1} *einer Sprache* L_{i+1}*, der meist* ***eine*** *bestimmte* ***Teilaufgabe*** *übernimmt, die sich mit keiner* ***Teilaufgabe*** *eines anderen* ***Passes*** *überschneidet und möglichst wenig* ***Ähnlichkeit*** *mit den* ***Teilaufgaben*** *anderer* ***Passes*** *haben sollte.*[a][b]

Für ***jeden Pass*** *und für einen* ***beliebigen*** *Abstrakten Syntaxbaum* A_i *gilt ähnlich, wie bei einem* ***vollständigen Compiler*** *in 2.45.1, dass:*



(2.45.1)

wobei man hier so tut, 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.*[c][d]

[a] Ein **Pass** kann mit einem **Transpiler** 5.5 (Definition 5.5) verglichen werden, da sich die zwei Sprachen L_i und L_{i+1} aufgrund der **Kleinschrittigkeit** meist auf einem ähnlichen **Abstraktionslevel** befinden. Der Unterschied ist allerdings, dass ein **Transpiler** zwei Programme, die in L_i bzw. L_{i+1} geschrieben sind kompiliert. Ein **Pass** ist dagegen immer **kleinschrittig** und operiert ausschließlich auf **Abstrakten Syntaxbäumen**, ohne Parsing usw.

[b] Der Begriff kommt aus dem **Englischen** von „passing over“, da der gesamte **Abstrakte Syntaxbaum** in einem **Pass** durchlaufen wird.

[c] **Interpretieren** geht immer von einem Programm in **Konkretter Syntax** aus, wobei der **Abstrakte Syntaxbaum** ein **Zwischenschritt** bei der **Interpretierung** ist.

[d] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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

2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche **Unreine Ausdrücke** (Definition 2.47) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.46) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 2.46: Reiner Ausdruck (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 **RETI-Code** darstellen lässt.*[a][b]

[a] Sondern z.B. **intern** etwas am **Kompilierprozess** ändert.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.47: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definiton 2.48).

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

*Ein **Statement** oder **Ausdruck** ist in **Monadischer Normalform**, wenn es oder er nach einer **Konkretten Grammatik** in **Monadischer Normalform** abgeleitet wurde.*

*Eine **Konkrette Grammatik** ist in **Monadischer Normalform**, wenn sie **reine Ausdrücke** und **unreine Ausdrücke nicht** miteinander **mischt**, sondern voneinander **trennt**.*[a]

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

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[13] aufgeschrieben wurden.

In der Abbildung 2.8 ist der Ausdruck mit dem **Nebeneffekt** eine Variable zu **allokieren:** `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt, daher muss der **Unreine** Ausdruck als eigenständiges Statement **vorangestellt** werden.

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

Abbildung 2.8: *Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten*

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschienenbefehlen** anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in **RETI-Knoten** haben. Somit wird eine **Seperation** von Subbäumen, die keine Entsprechung in **RETI-Knoten** haben und denen, die eine haben bewerkstelligt wird. Ein **Reiner Ausdruck** ist **Maschienenbefehlen** ähnlicher als ein Ausdruck, indem ein **Reiner** und **Unreiner Ausdruck** gemischt sind. Somit sparrt man sich in der Implementierung **Fallunterscheidungen**, indem die **Reinen Ausdrücke** direkt in **RETI-Code** übersetzt werden können und **nicht** unterschieden werden muss, ob darin **Unreine Ausdrücke** vorkommen.

2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um **Maschienensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.51) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Statements macht, in denen die **Komplexen Ausdrücke temporären Locations** zugewiesen werden (Definiton 2.49) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statemtent**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 2.50) enthalten.

Sollte es sich bei dem **Komplexen Ausdruck** um einen **Unreinen Ausdruck** handeln, welcher nur einen **Nebeneffekt** ausführt und sich nicht in **RETI-Befehle** übersetzt, so wird aus diesem ein **vorangestelltes Statement** gemacht, welches einfach nur den **Nebeneffekt** dieses **Unreinen Ausdrucks** ausführt.

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 tuen** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.52). Wenn eine **Konkrette Grammatik** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.52), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.50).

Definition 2.50: Atomarer Ausdruck

*Ein **Atomarer Ausdruck** ist ein Ausdruck, der ein **Reiner Ausdruck** ist und der in eine **Folge von RETI-Befehlen** übersetzt werden kann, die **atomar** ist, also **nicht** mehr weiter in kleinere Folgen von RETI-Befehlen **zerkleinert** werden kann, welche die **Übersetzung** eines anderen Ausdrucks sind.*

*Also z.B. im Fall der Sprache L_{PicoC} entweder eine **Variable** `var`, eine **Zahl** `12`, ein **ASCII-Zeichen** `'c'` oder ein **Zugriff auf eine Location**, wie z.B. `stack(1)`.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.51: Komplexer Ausdruck

*Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist, wie z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`.*[a][b]

[a] `int var` ist eine **Allokation**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.52: A-Normalform (ANF)

*Ein **Statement** oder **Ausdruck** ist in **A-Normalform**, wenn es oder er nach einer **Konkretten Grammatik** in **A-Normalform** abgeleitet wurde.*

*Eine **Konkrette Grammatik** ist in **A-Normalform**, wenn sie in **Monadischer Normalform** ist und wenn alle **Komplexen Ausdrücke** nur **Atomare Ausdrücke** enthalten und einer **Location** zugewiesen sind.*

*Eine **Abstrakte Grammatik** ist in **A-Normalform**, wenn die **Konkrette 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, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.9 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[14] aufgeschrieben wurden.

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition 5.11) inklusive **Liveness Analysis** (Definition 5.9) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.[15]

Aus diesem Grund verwendet das Beispiel in Abbildung 2.9 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **Abstrakte Syntaxbaum** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdrück** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

Im Fall, dass **Register** für z.B. **temporäre Zwischenergebnisse** genutzt werden und der **Maschienen-**

[14] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[15] Die in diesem **Paragraph** 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 und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

befehlssatz es erlaubt **zwei Register** miteinander zu verechnen[16], ist es möglich **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ als **atomar** zu definieren, da sie mit einem **Maschinenbefehl** verarbeitet werden können[17]. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das `SP`-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verechnet werden können.

Die Statements `4`, `x`, usw. für sich sind in diesem Fall **Statements**, bei denen ein **Komplexer Ausdruck** einer **Location**, in diesem Fall einer **Speicherzelle des Stack** zugewiesen wird, da `4`, `x` usw. in diesem Fall auch als **Komplexe Ausdrücke** zählen. Auf das Ergebnis dieser **Komplexen Ausdrücke** wird mittels `stack(2)` und `stack(1)` zugegriffen, um diese im **Komplexen Ausdruck** `stack(2) % stack(1)` miteinander zu verrechnen und wiederum einer Speicherzelle des Stack zuzuweisen.

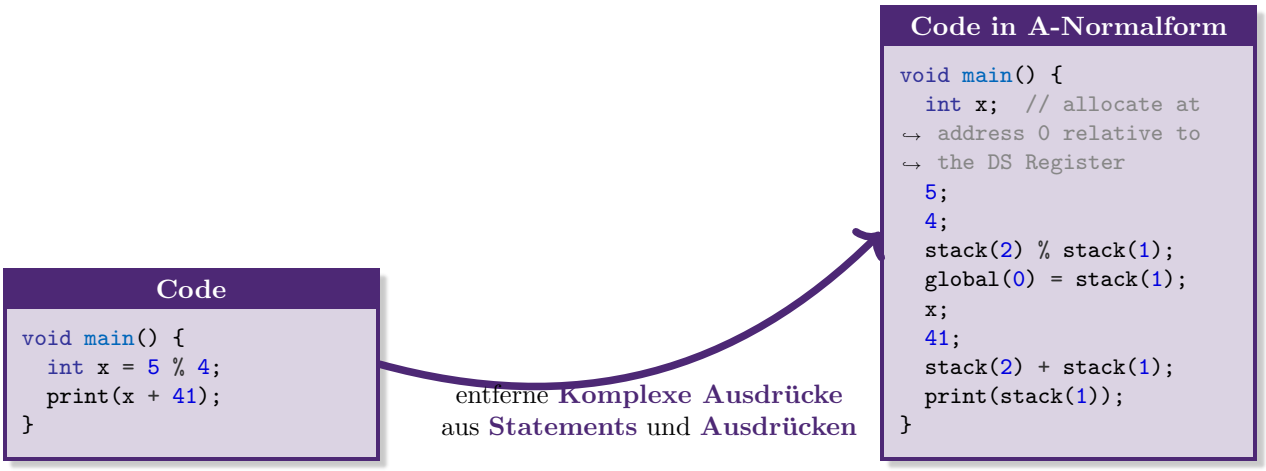


Abbildung 2.9: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen*

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

Alle weiteren denkbaren **Passes** sind zu **spezifisch** auf bestimmte **Statements** 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 definiert.

2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden ist es notwendig aus dem finalen **Abstrakten Syntaxbaum** den eigentlichen **Maschinencode** in **Konkretter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt. Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschienencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben. Der Weg von der **Abstrakten Syntax** zur **Konkretten Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkretten Syntax** zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden musste.

[16] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[17] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.

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** ; usw. ausgibt. Dabei wird nach dem Prinzip der **Tiefensuche** der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die immer 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**[18] eintritt:

1. der **Parser**[19] entscheidet das **Wortproblem** für ein **Eingabeprogramm**[20] mit `0`, also das **Eingabeprogramm** befolgt nicht die **Syntax** der **Sprache** des Compilers[21].

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

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

   - bei einem **Funktionsaufruf** werden **mehr** Argumente oder Argumente des **falschen Datentyps** übergeben, als in der **Funktionsdeklaration** oder **Funktionsdefinition** angegeben ist.

3. Während der **Laufzeit** des Compilers tritt ein Ereignis ein, das **nicht** durch die **Semantik** der Sprache des Compilers abgedeckt ist oder 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 während des des **Linkens** (Definition 5.4) etwas nicht zusammenpasst, wie z.B.:

- es gibt **keine** oder **mehr als eine** `main`-Funktion

- eine Funktion, die in einer **Objektdatei** (Definition 5.3) benötigt wird, wird von **keiner** anderen oder **mehr als einer** 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*** *einer Fehlermeldung kann dabei auf* ***verschiedene Weisen*** *erfolgen, wie z.B.*

- *über* `stdout` *oder* `stderr` *im einem* ***Terminal Emulator*** *oder* ***richtigen Terminal***[b]*.*

- *über eine* ***Dialogbox*** *in einer* ***Graphischen Benutzerfläche***[c] *oder* ***Zeichenorientierten Benutzerschnittstelle***[d]*.*

[18] *Errors in C/C++ - GeeksforGeeks*.
[19] Bzw. der **Recognizer** im Parser.
[20] Bzw. **Wort**.
[21] Bzw. das **Eingabeprogramm** lässt sich **nicht** mit der Konkretten Grammatik des Compilers **ableiten**.

- *in ein **Register** oder an eine **spezielle Adresse** des **Hauptspeichers** wird ein **Wert geschrieben**.*
- ***Logdatei**[e] auf einem **Speichermedium**.*

[a] Dieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler selbst kompiliert** hat.
[b] Nur unter **Linux**, **Windows** hat sowas nicht.
[c] In engl. **G**raphical **U**ser **I**nterface, kurz **GUI**.
[d] In engl. **T**ext-based **U**ser **I**nterface, kurz **TUI**.
[e] 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 **Konkretten Syntax** der Sprache L_{PicoC} hin zum einem Programm mit derselben **Semantik** in der **Konkretten Syntax** der Sprache L_{RETI} erklärt.

Für das Parsen[1] des Programmes in der **Konkretten 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 Konkretten Grammatik ein Programm in **Konkretter 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.*[b] [c]

*Die Erweiterte Backus-Naur-Form ist zwar **standartisiert** und die Spezifikation des Standards kann unter Link[d] aufgefunden werden, allerdings werden in der Praxis, wie z.B. in Lark oft **eigene Notationen** verwendet.*

[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] `https://standards.iso.org/ittf/PubliclyAvailableStandards/`.

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

Definition 3.4: Dialekt der EBNF aus Lark

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

*Ein wichtiger **Unterschied** ist z.B., dass dieser Dialekt anstelle von **geschweiften Klammern** {} für die Darstellung von **Wiederholung**, den aus **regulären Ausdrücken** bekannten ∗-Quantor optional zusammen mit **runden Klammern** `()` verwendet: `()`∗.[b]*

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

[a] `https://lark-parser.readthedocs.io/en/latest/grammar.html`.
[b] Bzw. kann der ∗-Quantor auch **keinmal** wiederholen bedeuteten.

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 **Konkretten Grammatik** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkretten Grammatik** direkt ein Programm in dieser Konkretten Grammatik **parsen** und einen **Ableitungsbaum** dafür generieren zu können.

Eine möglichst **geringe Laufzeit** durch Verwenden der effizientesten Algorithmen zu erreichen war keine der Hauptzielsetzungen für den **PicoC-Compiler**, da der **PicoC-Compiler** vor allem als **Lerntool** konzipiert ist, mit dem Studenten lernen können, wie der **Kompiliervorgang** von der Programmiersprache L_{PicoC} zur Maschinensprache L_{RETI} funktioniert. Eine ausführliche Diskussion zur Priorisierung Laufziet wurde in Unterkapitel 1.4 geführt. Lark besitzt des Weiteren eine **sehr gute Dokumentation** *Welcome to Lark's documentation! — Lark documentation*, sodass anderen Studenten, die den **PicoC-Compiler** vielleicht in ihr Projekt einbinden wollen, unkompliziert **Erweiterungen** für den **PicoC-Compiler** schreiben können.

Neben den Konkretten Grammatiken, die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** spezifiziert sind, werden in den folgenden Unterkapiteln die **Abstrakten Grammatiken**, welche spezifzieren, welche Kompositionen für die **Abstrakten Syntaxbäume** der verschiedenden **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben, die allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit** hat.

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, wodurch es die Abstrakten Grammatiken nur **unnötig verkomplizieren** würde, 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 Syntax Form Notation** (Definition 3.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrette** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 3.5: Abstrakte Syntax Form (ASF)

*Die Abstrakte Syntax Form ist eine eigene **Metasyntax** für **Abstrakte Grammatiken**, die für diese Bachelorarbeit definiert wurde und sich von dem **Dialekt** der **Backus-Naur-Form** des **Lark Parsing Toolkit** nur dadurch unterschiedet, dass **Terminalsymbole** nicht von `""` engeschlossen sein müssen, da die **Knoten** in der **Abstrakten Syntax**, sowieso schon **klar auszumachen** sind und von anderen Symbolen der **Metasprache** leicht zu unterschiden sind.*

[4] *ANTLR*.

Letztendlich geht es allerdings nur darum, dass aufgrund der Verwendung des **Lark Parsing Toolkit** die **Konkrette 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 Konkrette Grammatik zu definieren, die den Teil der **Konkretten Syntax** beschreibt, der die **verschiedenen Pattern** für die verschiedenen Token der Sprache L_{PicoC} beschreibt, also den Teil der für die **Lexikalische Analyse** wichtig ist. Diese Konkrette Grammatik wird dann vom **Lark Parsing Toolkit** dazu verwendet ein Programm in **Konkretter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel 2.3 erläutert ist.

3.1.1 Konkrette Grammatik für die Lexikalische Analyse

In der Konkretten Grammatik 3.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokennamen** oder einen **Teil der Beschreibung** eines **Tokennamen**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol `NUM` um einen **Tokennamen**, der durch die **Produktion** `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` beschrieben wird und beschreibt, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann. Das ist daran festzumachen, dass das Nicht-Terminalsymbol `NUM` in keiner anderen Produktion vorkommt, die auf der **linken Seite** des `::=`-Symbols ebenfalls ein **großgeschriebenen** Nicht-Terminalsymbol hat. Dagegen dient das **großgeschriebene** Nicht-Terminalsymbol `DIG_NO_0` aus der Produktion `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` nur zu Beschreibung von `NUM`.

Die in der Konkretten Grammatik 3.1.1 für die Lexikalische Analyse definierten **Nicht-Terminalsymbole** können in der Konkretten 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 Konkrette Grammatik vereinzelt **kleingeschriebenen** Nicht-Terminalsymbole, wie `name` haben nur den Zweck mehrere **Tokennamen**, wie `NAME | INT_NAME | CHAR_NAME` unter einem Überbegriff zu sammeln.

In Lark steht eine Zahl `.ZAHL`, die an ein **Nicht-Terminalsymbol** angehängt ist, dass auf der linken Seite des `::=`-Symbols einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. 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[5], damit das Wort `int` den **Tokennamen** `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 Tokennamen zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. In diesem Fall würden aus `int_var` die Token `Token('INT_DT', 'int'), Token('NAME', '_var')` generiert, anstatt `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[6] zu finden ist. Diese

[5] Es wird immer die **höchste** Priorität **zuerst** genommen.
[6] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.

Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten und ist aufgrund dessen, dass sie in der Lage ist nach einer spezifizierten Konkretten 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, allerdinds berücksichtigt der **Basic Lexer** ebenfalls **Priortiäten**, sodass für den aktuellen Index im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Wort** an dem aktuellen Index im Eingabeprogramm erkennt, bekommt es direkt den **Tokenwert** dieser Produktion zugewiesen. Weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

COMMENT	::=	"//" /[^\n]*/ | "/*" /(. | \n)*?/ "*/"	*L_Comment*
RETI_COMMENT.2	::=	"//""␣"?"#"/[^\n]*/	
DIG_NO_0	::=	"1" | "2" | "3" | "4" | "5" | "6"	*L_Arith*
	|	"7" | "8" | "9"	
DIG_WITH_0	::=	"0" | *DIG_NO_0*	
NUM	::=	"0" | *DIG_NO_0* *DIG_WITH_0*∗	
ASCII_CHAR	::=	"␣".."∼"	
CHAR	::=	"'" *ASCII_CHAR* "'"	
FILENAME	::=	*ASCII_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	::=	"%"	
*prec*1*_op*	::=	*MUL_DEREF_PNTR* | *DIV* | *MOD*	
ADD	::=	"+"	
SUB_MINUS	::=	"−"	
*prec*2*_op*	::=	*ADD* | *SUB_MINUS*	
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: *Konkrette 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** erkannten **Token** 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 **Konkretter 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 Präzidenz und Assoziativität

In diesem Unterkapitel wird eine ähnliche **Erklärweise**, wie in Quelle *Parsing Expressions · Crafting Interpreters* verwendet. Die Programmiersprache L_{PicoC} hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache L_C[7]. Die **Präzidenzregeln** der verschiedenen **Operatoren** der Programmiersprache L_{PicoC} sind in Tabelle 3.1 aufgelistet.

[7] *C Operator Precedence - cppreference.com*.

Präzidenzstufe	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 a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	`a==b a!=b`	Gleichheit und Ungleichheit	
7	`a&b`	Bitweise UND	
8	`a^b`	Bitweise XOR (exclusive or)	
9	`a|b`	Bitweise ODER (inclusive or)	
10	`a&&b`	Logiches UND	
11	`a||b`	Logisches ODER	
12	`a=b`	Zuweisung	Rechts, dann links ←

Tabelle 3.1: *Präzidenzregeln von PicoC*

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

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*exp*")"	*L_Arith*
	|	*exp*"["*exp*"]" | *exp*"."*name* | *name*"("*fun_args*")"	*+L_Logic*
fun_args	::=	[*exp*"," *exp*)∗]	*+ L_Pntr*
un_op	::=	"−" | "∼" | "!" | "∗" | "&"	*+ L_Array*
un_exp	::=	*un_op exp*	*+ L_Struct*
bin_op	::=	"∗" | "/" | "%" | "+" | "−" | "&" | "∧" | "|"	*+ L_Fun*
	|	"<" | "<=" | ">" | ">=" | "!=" | "=="	
	|	"&&" | "||" | "="	
bin_exp	::=	*exp bin_op exp*	
exp	::=	*prim_exp* | *un_exp* | *bin_exp*	

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

Die Konkrette Grammatik 3.2.1 ist allerdings **mehrdeutig**, d.h. verschiedene **Linksableitungen** in der Konkretten 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.

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

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

Beide **Wörter** sind **gleich**, allerdings sind die **Ableitungsbäume unterschiedlich**, wie in Abbildung 3.1 zu sehen ist.

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 in **Klammern** setzen, um die **Präzidenz** 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.

Aus diesem Grund ist es wichtig die **Präzidenzregeln** und die **Assoziativität** der Operatoren beim Erstellen der Konkretten Grammatik miteinzubeziehen. Hierzu wird nun Tabelle 3.1 betrachtet. Für jede **Präzidenzstufe** in der Tabelle 3.1 wird eine eigene Regel erstellt werden, wie es in Grammatik 3.2.2 dargestellt ist. Zudem braucht es eine **Produktion** `prim_exp` für die höchste **Präzidenzstufe**, welche **Literale**, wie `'c'`, `5` oder `var` und geklammerte Ausdrücke wie `(3 && 14)` abdeckt.

prim_exp	::=	...	*L_Arith + L_Array*
post_exp	::=	...	*+ L_Pntr + L_Struct*
un_exp	::=	...	*+ L_Fun*
arith_prec1	::=	...	
arith_prec2	::=	...	
arith_and	::=	...	
arith_oplus	::=	...	
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 Konkretten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet*

Einige **Bezeichnungen** von **Nicht-Terminalsymbolen** auf der **linken Seite** des `::=`-Operators der **Produktionen** 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_and	a<b a<=b a>b a>=b
arith_oplus	a==b a!=b
arith_or	a&b
rel_exp	a^b
eq_exp	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 **Präzidenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzidenzstufe **höher** ist. Z.B. soll `un_op` sowohl den Ausdruck `-(3 * 14)` als auch einfach nur `(3 * 14)`[8] erkennen können, aber nicht `3 * 14` ohne Klammern, da dieser Ausdruck eine **geringe Präzidenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

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 auch **terminieren** kann und es auch möglich ist auschließlich einen Ausdruck **höherer Präzidenz** (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*

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 Präzidenz** (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*

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 Präzidenzstufe 4 in Folge erkannt werden können[9] (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzidenzstufe 4

[8] **Geklammerte Ausdrücke** werden nämlich von `prim_exp` erkannt, welches eine höhere **Präzidenzstufe** hat.
[9] Bezogen auf Tabelle 3.1.

haben). Das **Nicht-Terminalsymbol** `arith_prec1` auf der **rechten Seite** ermöglicht es, dass zwischen den Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzidenzstufe 4 haben und `/` Präzidenzstufe 3). Mit der Alternative `arith_prec1` ist es möglich, dass auschließlich ein Ausdruck **höherer Präzidenz** 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*

Anmerkung

Manche **Parser**[a] haben allerdings ein Problem mit **Linksrekursion** (Definition 2.24), 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*

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 Präzidenzstufe 4 in Folge erkennen zu können (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzidenzstufe 4 haben).

Das **Nicht-Terminalsymbol** `arith_prec1` erlaubt es, dass **zwischen** der Folge von Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzidenzstufe 4 haben und `/` Präzidenzstufe 3). Da der in der EBNF definierte `*`-Operator 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 auschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B. `1 / 4`).

[a] Darunter zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird **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()`). Somit ergibt sich die Konkrette Grammatik 3.2.7.

$prec1\_op$	::=	$"*" \mid "/" \mid "\%"$	*L_Misc*
$prec2\_op$	::=	$"+" \mid "-"$	
rel_op	::=	$"<" \mid "<=" \mid ">" \mid ">="$	
eq_op	::=	$"==" \mid "!="$	
fun_args	::=	$[logic\_or(","\ logic\_or)*]$	
$prim\_exp$	::=	$name \mid NUM \mid CHAR \mid "("logic\_or")"$	*L_Arith*
$post\_exp$	::=	$post\_exp"["logic\_or"]" \mid post\_exp"."name \mid name"("fun\_args")"$	*+ L_Array*
	$\mid$	$prim\_exp$	*+ L_Pntr*
un_exp	::=	$un\_op\ un\_exp \mid post\_exp$	*+ L_Struct*
$arith\_prec1$	::=	$arith\_prec1\ prec1\_op\ un\_exp \mid un\_exp$	*+ L_Fun*
$arith\_prec2$	::=	$arith\_prec2\ prec2\_op\ arith\_prec1 \mid arith\_prec1$	
$arith\_and$	::=	$arith\_and\ "\&"\ arith\_prec2 \mid arith\_prec2$	
$arith\_oplus$	::=	$arith\_oplus\ "\wedge"\ arith\_and \mid arith\_and$	
$arith\_or$	::=	$arith\_or\ "|"\ arith\_oplus \mid arith\_oplus$	
rel_exp	::=	$rel\_exp\ rel\_op\ arith\_or \mid arith\_or$	*L_Logic*
eq_exp	::=	$eq\_exp\ eq\_op\ rel\_exp \mid rel\_exp$	
$logic\_and$	::=	$logic\_and\ "\&\&"\ eq\_exp \mid eq\_exp$	
$logic\_or$	::=	$logic\_or\ "||"\ logic\_and \mid logic\_and$	
$assign\_stmt$	::=	$un\_exp\ "="\ logic\_or";"$	*L_Assign*

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

3.2.2 Konkrette Grammatik für die Syntaktische Analyse

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

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrette Grammatik** an die **aktuellste kostenlos auffindbare Version** der echten Konkretten 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[10], damit es sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C. Wobei z.B. die Compiler **GCC**[11] und **Clang**[12] zu nennen wären.

In der Konkretten Grammatik 3.2.8 für die **Syntaktischen Analyse** werden einige der **Tokennamen** aus der Konkretten Grammatik 3.1.1 für die **Lexikalischen Analyse** verwendet, wie z.B. `NUM` aber auch `name`, welches eine Produktion ist, die mehrere **Tokennamen** unter einem Überbegriff zusammenfasst.

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

[10] An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts verändert, wie die Konkretten Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Yacc) old* und **Lexikalische Analyse** *ANSI C grammar (Lex) old* aus dem Jahre 1985 zeigen.
[11] *GCC, the GNU Compiler Collection - GNU Project*.
[12] *clang: C++ Compiler*.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith + 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*""("")"	
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_and	::=	*arith_and* "&" *arith_prec*2 | *arith_prec*2	
arith_oplus	::=	*arith_oplus* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_oplus* | *arith_oplus*	
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: *Konkrette 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: *Konkrette Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2*

Anmerkung

In der Konkretten 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 Konkrette 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** zu generieren.

Definition 3.6: Earley Parser

*Ist ein Algorithmus für das **Parsen** von **Wörtern** einer **Kontextfreien Sprache**, der ein **Chart Parser** ist, welcher einen mittels **Dynamischer Programmierung** und dem **Top-Down Ansatz** arbeitenden **Earley Recognizer** (Defintion 5.8 im Kapitel Appendix) nutzt, um einen **Ableitungsbaum** zu konstruieren.*

*Zur **Konstruktion des Ableitungsbaumes** muss dafür gesorgt werden, dass der **Earley Recognizer** bei der **Vervollständigungsoperation** Zeiger auf den **vorherigen Zustand** hinzugefügt, um durch **Rückwärtsverfolgen** dieser **Zeiger** die **Ableitung** 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 **Token** der **Lexikalischen Analyse** aus dem Beispiel in Code 3.1 generiert wurde, ist in Code 3.3 zu sehen. 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 Konkretten Grammatik den **linken Seiten** des `::=`-Symbols[13] entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition 2.37 eines **Ableitungsbaumes** auch schon definiert ist. Die **EBNF-Grammatik** 3.2.8 des **PicoC-Compilers** erlaubt es allerdings 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.

Die Ausgabe des **Abstrakten Syntaxbaumes** ist bewusst so gewählt, dass sie sich optisch vom **Ableitungsbaum** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase**[14] geschrieben sind, im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des `::=`-Symbols.

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 die Programmiersprache L_{PicoC} korrekt bestimmen zu können, wie z.B. ein **„Feld der Mächtigkeit** 3 **von Zeigern auf Felder der Mächtigkeit** 2 **von Integern“** `int (*ar[3])[2]` die **Spiralregel**[15] in der Implementeirung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Konkretten Grammatik 3.2.8 auf eine spezielle Weise passend spezifiziert.

Was man erhalten will, ist ein **entarteter Baum** 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])`.

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.

[13] *Grammar: The language of languages (BNF, EBNF, ABNF and more).*
[14] *Naming convention (programming).*
[15] *Clockwise/Spiral Rule.*

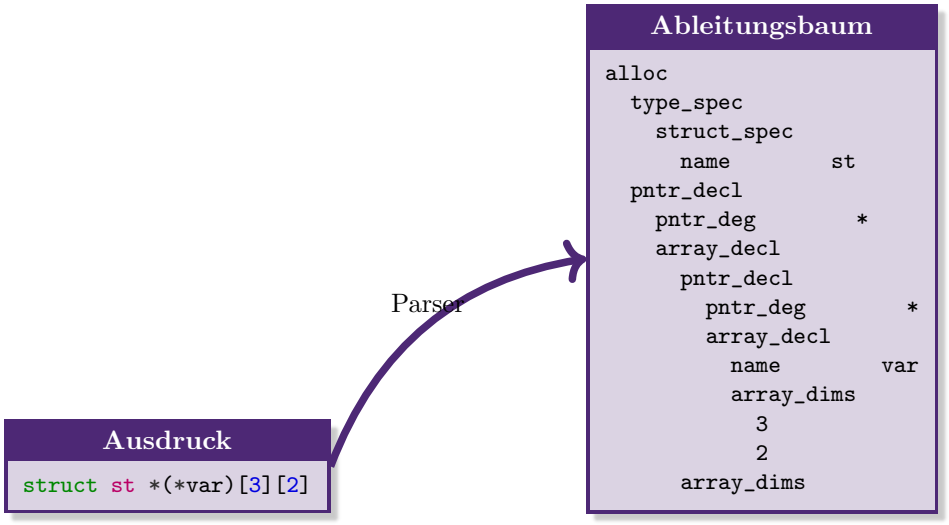


Abbildung 3.2: *Ableitungsbaum nach Parsen eines Ausdrucks*

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 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 muss beim **FeldDeclarator** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht werden.

Des Weiteren befindet sich in der **Mitte** dieser **Spirale**, die der **Ableitungsbaum** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`, da der **Ableitungsbaum** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Programms** in **Konkretter Syntax** (z.B. `struct st *(*var[3][2])`) **generiert** wird darstellt. Der **Name der Variable** `name(var)` sollte daher mit dem **innersten Datentyp** `struct st` ausgetauscht werden.

In Abbildung 3.3 ist daher zu sehen, wie der **Ableitungsbaum** aus Abbildung 3.2 mithilfe eines **Visitors** (Definition 2.41) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des **Visitors** aus dem **Lark Parsing Toolkit** ist unter Link[16] 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 2.41 erklärt wurde.

[16] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.

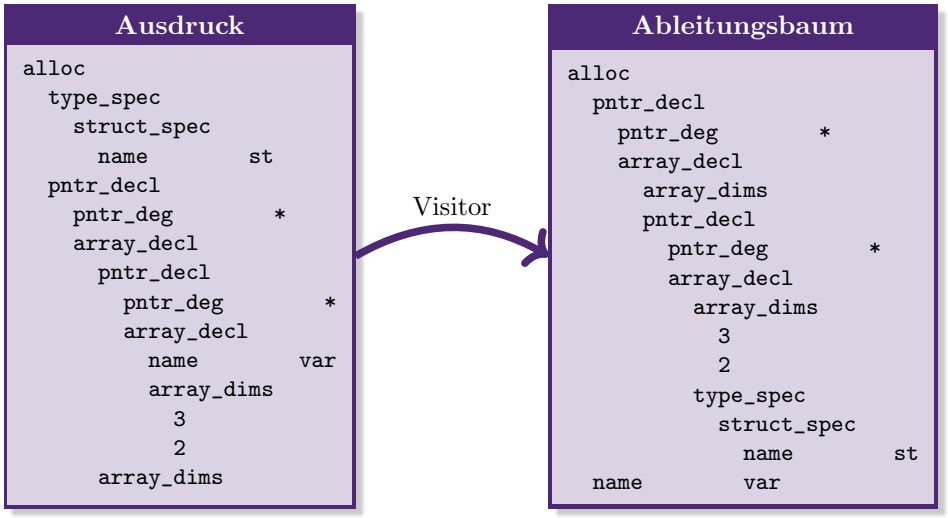


Abbildung 3.3: *Ableitungsbaum nach Vereinfachung*

3.2.4.1 Codebeispiel

In Code 3.4 ist der **Ableitungsbaum** aus Code 3.3 nach der **Vereinfachung** mithilfe eines **Visitors** zu sehen.

```
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 2.40) einen **Abstrakten Syntaxbaum** aus ihm zu generieren. Würde man den **vereinfachten Ableitungsbaum** des Ausdrucks `struct st *(*var[3][2])` auf passende Weise in einen **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[17] 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 2.40 erklärt wurde.

Den Teilbaum, der den Datentyp darstellt würde man von von **oben-nach-unten**[18] als **„Zeiger auf einen Zeiger auf ein Feld der Mächtigkeit 2 von Feldern der Mächtigkeit 3 von Verbunden des Typs `st`“** lesen, also genau anders herum, 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.

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

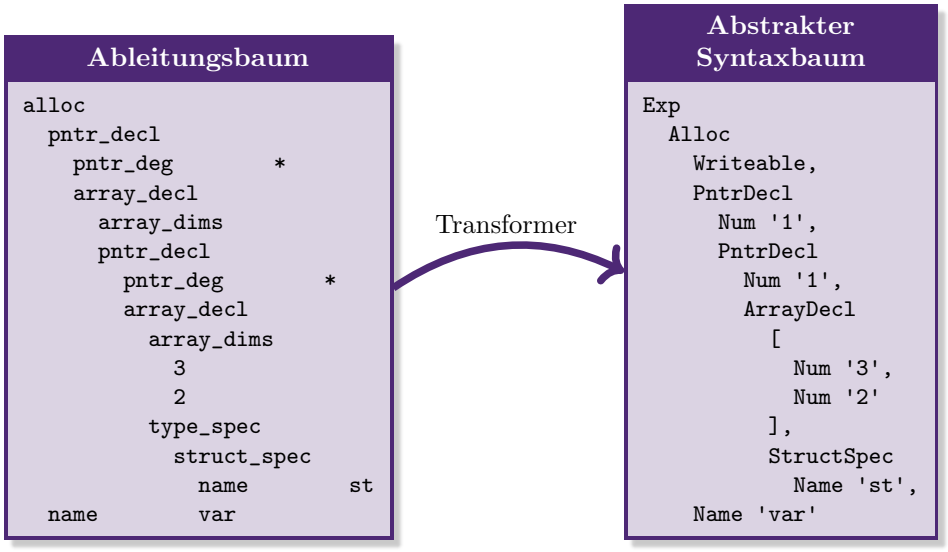


Abbildung 3.4: *Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen*

Dieser **Abstrakte Syntaxbaum** ist für die Weiterverarbeitung ungeeignet, denn für die **Adressberechnung** für eine Aneinandereihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundattribute**, welche in Unterkapitel 3.3.5.3 genauer erläutert wird, 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 **Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie in Abbildung 3.5 zustande, der für die Weiterverarbeitung geeignet ist.

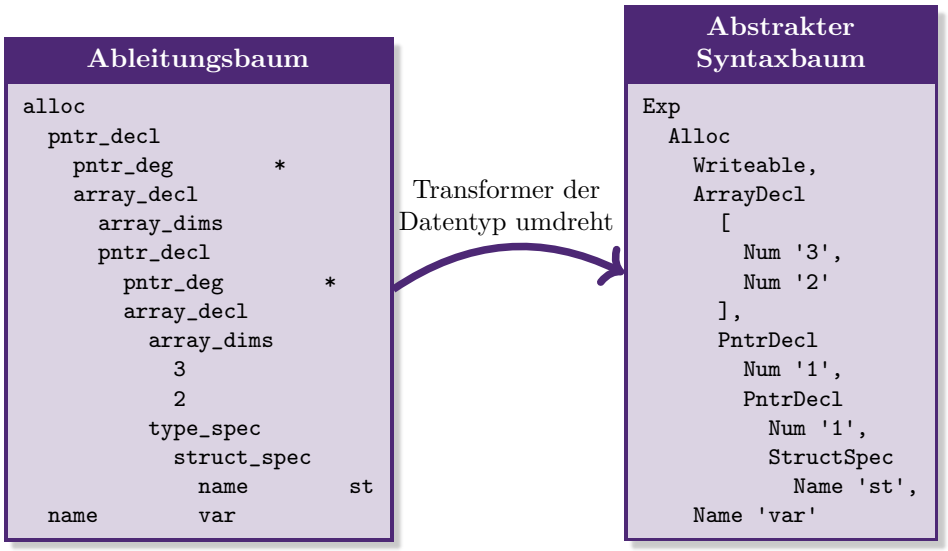


Abbildung 3.5: *Generierung eines Abstrakten Syntaxbaumes mit Umdrehen*

Die Weiterverarbeitung des **Abstrakten Syntaxbaumes** geschieht mithilfe von **Passes**, welche im Unterkapitel 2.5 genauer beschrieben werden. Da die Knoten des **Abstrakten Syntaxbaumes** anders als beim **Ableitungsbaum** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Konkretten 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**, die alle in den unterschiedlichen von den **Passes** umgeformten **Abstrakten Syntaxbäumen** vorkommen.

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 irgendeinen **Ausdruck** aus der Sprache 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[19]. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakten Syntaxbäumen** vorkommen sind in Tabelle 3.3 mit einem **Beschreibungstext** dokumentiert.

[19] 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(val)`	Ein **Bezeichner**, z.B. `my_fun`, `my_var` usw. , aber da es keine gute Kurzform für `Identifier()` (englisches Wort für Bezeichner) gibt, wurde dieser Knoten `Name()` genannt.
`Num(val)`	Eine **Zahl**, z.B. `42`, `-3` usw.
`Char(val)`	Ein **Zeichen** der **ASCII-Zeichenkodierung**, z.B. `'c'`, `'*'` usw.
`Minus()`, `Not()`, `DerefOp()`, `RefOp()`, `LogicNot()`	Die **unären Operatoren** `un_op`: `-a`, `~a`, `*a`, `&a` `!a`.
`Add()`, `Sub()`, `Mul()`, `Div()`, `Mod()`, `Oplus()`, `And()`, `Or()`, `LogicAnd()`, `LogicOr()`	Die **binären Operatoren** `bin_op`: `a + b`, `a - b`, `a * b`, `a / b`, `a % b`, `a` $\wedge$ `b`, `a & b`, `a` $\mid$ `b`, `a && b`, `a` $\mid\mid$ `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 **Primitiven Datentypen**, die in der Abstrakten Syntax, um eine intuitive Bezeichnung zu haben einfach nur als **Datentypen** `datatype` eingeordnet werden: `int`, `char`, `void`.
`Placeholder()`	**Platzhalter** für einen Knoten, der diesen später **ersetzt**.
`BinOp(exp, bin_op, exp)`	Container für eine **binäre Operation** mit 2 Expressions: `<exp1> <bin_op> <exp2>`
`UnOp(un_op, exp)`	Container für eine **unäre Operation** mit einer Expression: `<un_op> <exp>`.
`Exit(num)`	Container für einen **Exit Code**, der vor der Beendigung in das `ACC` Register geschrieben wird und steht für die **Beendigung** des laufenden Programmes.
`Atom(exp, rel, exp)`	Container für eine **binäre Relation** mit 2 Expressions: `<exp1> <rel> <exp2>`
`ToBool(exp)`	Container für einen **Arithmetischen Ausdruck**, wie z.B. `1 + 3` oder einfach nur `3`, der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis $x > 1$ auf 1 abgebildet wird.
`Alloc(type_qual, datatype, name, local_var_or_param)`	Container für eine **Allokation** `<type_qual> <datatype> <name>` mit den notwendigen Knoten `type_qual`, `datatype` und `name`, die alle für einen Eintrag in der **Symboltabelle** notwendigen Informationen enthalten. Zudem besitzt er ein verstecktes Attribut `local_var_or_param`, dass die Information trägt, ob es sich bei der **Variable** um eine **Lokale Variable** oder einen **Parameter** handelt.
`Assign(lhs, exp)`	Container für eine **Zuweisung**, wobei `lhs` ein `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` sein kann und `exp` ein beliebiger **Logischer Ausdruck** sein kann: `lhs = exp`.

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** soll. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` im **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Stack(num)`	Container, der für das **temporäre** Ergebnis einer Berechnung, das `num` Speicherzellen relativ zum **Stackpointer Register** `SP` steht.
`Stackframe(num)`	Container, der für eine Variable steht, die `num` Speicherzellen relativ zum **Begin-Aktive-Funktion Register** `BAF` steht.
`Global(num)`	Container, der für eine Variable steht, die `num` Speicherzellen relativ zum **Datensegment Register** `DS` steht.
`StackMalloc(num)`	Container, der für das **Allokieren** von `num` Speicherzellen auf dem **Stack** steht.
`PntrDecl(num, datatype)`	Container, der für den **Zeigerdatentyp** steht: `<prim_dt> *<var>`, wobei das **Attribut** `num` die **Anzahl zusammengefasster Zeiger** angibt und `datatype` der **Datentyp** ist, auf den der oder die **Zeiger** zeigen.
`Ref(exp, datatype, error_data)`	Container, der für die Anwendung des **Referenz-Operators** `&<var>` steht und die **Adresse** einer **Location** (Definition 2.49) auf den Stack schreiben soll, die über `exp` eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` im **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Deref(lhs, exp)`	Container für den **Indexzugriff** auf einen **Feld-** oder **Zeigerdatentyp**: `<var>[<i>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`ArrayDecl(nums, datatype)`	Container, der für den **Felddatentyp** steht: `<prim_dt> <var>[<i>]`, wobei das **Attribut** `nums` eine Liste von `Num('x')` ist, die die **Dimensionen** des Feld angibt und `datatype` der Datentyp ist, der über das Anwenden von `Subscript()` auf das Feld zugreifbar ist.
`Array(exps, datatype)`	Container für den **Initializer** eines **Feldes**, dessen Einträge `exps` weitere Initializer für eine **Feld-Dimension** oder ein Initializer für ein **Struct** oder ein **Logischer Ausdruck** sein können, z.B. `{{1, 2}, {3, 4}}`. Des Weiteren besitzt er 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>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`StructSpec(name)`	Container für einen selbst definierten **Structtyp**: `struct <name>`, wobei das **Attribut** `name` festlegt, welchen **selbst definierten** Structtyp dieser Knoten repräsentiert.
`Attr(exp, name)`	Container für den **Attributzugriff** auf einen **Structdatentyp**: `<var>.<attr>`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `name` das Attribut ist, auf das zugegriffen werden soll.

Tabelle 3.4: *PicoC-Knoten Teil 2*

PiocC-Knoten	Beschreibung
`Struct(assigns, datatype)`	Container für den **Initializer** eines **Structs**, z.B {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag `assigns` eine Liste von `Assign(lhs, exp)` ist mit einer Zuordnung eines **Attributezeichners**, zu einem weiteren Initializer für eine **Feld-Dimension** oder zu einem Initializer für ein **Struct** oder zu einem **Logischen Ausdruck**. Des Weiteren besitzt er ein verstecktes Attribut `datatype`, welches für den **PicoC-ANF Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind.
`StructDecl(name, allocs)`	Container für die Deklaration eines **selbstdefinierten Structtyps**, z.B. `struct <var> {<datatype> <attr1>; <datatype> <attr2>;};`, wobei `name` der **Bezeichner** des Structtyps ist und `allocs` eine Liste von Bezeichnern der **Attribute** des Structtyps mit dazugehörigem **Datentyp**, wofür sich der Knoten `Alloc(type_qual, datatype, name)` sehr gut als Container eignet.
`If(exp, stmts)`	Container für ein **If Statement** `if(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`IfElse(exp, stmts1, stmts2)`	Container für ein **If-Else Statement** `if(<exp>) { <stmts2> } else { <stmts2> }` inklusive **Codition** `exp` und 2 **Branches** `stmts1` und `stmts2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Statements** oder `GoTo(Name('block.xyz'))`'s stehen können.
`While(exp, stmts)`	Container für ein **While-Statement** `while(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`DoWhile(exp, stmts)`	Container für ein **Do-While-Statement** `do { <stmts> } while(<exp>);` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`Call(name, exps)`	Container für einen **Funktionsaufruf**: `fun_name(exps)`, wobei `name` der **Bezeichner** der Funktion ist, die aufgerufen werden soll und `exps` eine **Liste von Argumenten** ist, die an die Funktion übergeben werden soll.
`Return(exp)`	Container für ein `Return-Statement`: `return <exp>`, wobei das **Attribut** `exp` einen **Logischen Ausdruck** darstellt, dessen Ergebnis vom **Return-Statement** zurückgegeben wird.
`FunDecl(datatype, name, allocs)`	Container für eine **Funktionsdeklaration**, z.B. `<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, wobei der Knoten `Alloc(type_spec, datatype, name)` als Cotainer für die Parameter dient.

Tabelle 3.5: *PicoC-Knoten Teil 3*

PiocC-Knoten	Beschreibung
`FunDef(datatype, name, allocs, stmts_blocks)`	Container für eine **Funktionsdefinition**, z.B. `<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, wobei der Knoten `Alloc(type_spec, datatype, name)` als Container für die Parameter dient und `stmts_blocks` eine Liste von **Statemetns** bzw. **Blöcken** ist, welche diese Funktion beinhaltet.
`NewStackframe(fun_name, goto_after_call)`	Container für die **Erstellung** eines neuen **Stackframes** und Speicherung des Werts des `BAF`-Registers der **aufrufenden Funktion** und der **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 ist 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.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes** durch das **Wiederherstellen** des im noch **aktuellen Stackframe** gespeicherten Werts des `BAF`-Registes der **aufrufenden Funktion** und das Setzen des `SP`-Registers auf den Wert des `BAF`-Registesr **vor** der Wiederherstellung.
`File(name, decls_defs_blocks)`	Container für alle **Funkionen** oder **Blöcke**, welche eine Datei als Ursprung haben, wobei `name` der **Dateiname** der Datei ist, die erstellt wird 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 **Statements**, der auch als **Block** bezeichnet wird, wobei das Attribut `name` der Bezeichner des **Labels** (Definition 3.7) des Blocks ist und `stmts_instrs` eine **Liste von Statements** oder **Befehlen**. Zudem besitzt er noch 3 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.
`GoTo(name)`	Container für ein `Goto` zu einem anderen **Block**, wobei das Attribut `name` der Bezeichner des **Labels** des Blocks ist zu dem Gesprungen werden soll.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar**, den der Compiler selber während des **Kompiliervorangs** erstellt, der im **RETI-Interpreter** selbst später **nicht** sichtbar sein wird, aber in den **Immediate**-Dateien, welche die **Abstrakten Syntaxbäume** nach den verschiedenen **Passes** enthalten.
`RETIComment(value)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer **RETI-CPU nicht umsetzbar** ist und auch nicht sinnvoll wäre umzusetzen. Der **Kommentar** ist im Attribut **value**, welches jeder Knoten besitzt gespeichert.

Tabelle 3.6: *PicoC-Knoten Teil 4*

Definition 3.7: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a] Thiemann, „Compilerbau“.

Anmerkung

Die ausgegrauten Attribute der PicoC-Knoten sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Knoten** mit einem Wert **initialisiert** werden, sondern im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen bekommen, dass im weiteren Kompiliervorgang **Informationen** transportiert, die später im Kompiliervorgang nicht mehr so leicht zugänglich wären.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** `value` und `position`, wobei **value** bei einem Knoten, der einen Blatt darstellt dem **Tokenwert** des Tokens, welches es ersetzt entspricht und Inneren Knoten unbesetzt ist. Das **Attribut** `position` wird für **Fehlermeldungen** gebraucht.

3.2.5.2 RETI-Knoten

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** 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 für alle **Befehle**: `<name> <instrs>`, wobei `name` der **Dateiname** der Datei ist, die erstellt wird und `instrs` eine **Liste von Befehlen** ist.
`Instr(op, args)`	Container für einen **Befehl**: `<op> <args>`, wobei `op` eine **Operation** ist und `args` eine **Liste von Argumenten** für dieser Operation.
`Jump(rel, im_goto)`	Container für einen **Sprungbefehl**: `JUMP<rel> <im>`, wobei `rel` eine **Relation** ist und `im_goto` ein **Immediate Value** `Im(val)` für die **Anzahl an Speicherzellen**, um die relativ zum **Sprungbefehl** gesprungen werden soll oder ein `GoTo(Name('block.xyz'))`, das später im **RETI-Patch Pass** durch einen passenden **Immediate Value** ersetzt wird.
`Int(num)`	Container für einen **Interruptaufruf**: `INT <im>`, wobei `num` die **Interrruptvektornummer** (IVN) für die passende Speicherzelle in der **Interruptvektortabelle** ist, in der die Adresse der **Interrupt-Service-Routine** (ISR) steht.
`Call(name, reg)`	Container für einen **Prozeduraufruf**: `CALL <name> <reg>`, wobei `name` der **Bezeichner** der Prozedur, die aufgerufen werden soll ist und `reg` ein **Register** ist, das als **Argument** an die Prozedur dient. Diese **Operation** ist in der Betriebssysteme Vorlesung[a] nicht deklariert, sondern wurde dazuerfunden, um unkompliziert ein `CALL PRINT ACC` oder `CALL INPUT ACC` im RETI-Interpreter simulieren zu können.
`Name(val)`	Bezeichner für eine **Prozedur**, z.B. `PRINT` oder `INPUT` oder den **Programnamen**, z.B. `PROGRAMNAME`. Dieses **Argument** ist in der Betriebssysteme Vorlesung[a] nicht deklariert, sondern wurde dazuerfunden, um Bezeichner, wie `PRINT`, `INPUT` oder `PROGRAMNAME` schreiben zu können.
`Reg(reg)`	Container für ein **Register**.
`Im(val)`	Ein **Immediate Value**, z.B. `42, -3` usw.
`Add(), Sub(), Mult(), Div(), Mod(), Oplus(), Or(), And()`	**Compute-Memory** oder **Compute-Register** Operationen: `ADD, SUB, MULT, DIV, OPLUS, OR, AND`.
`Addi(), Subi(), Multi(), Divi(), Modi(), Oplusi(), 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**: `<, <=, >, >=, ==, !=, _NOP`.
`Rti()`	**R**eturn-**F**rom-**I**nterrupt Operation: `RTI`.
`Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()`	**Register**: `PC, IN1, IN2, ACC, SP, BAF, CS, DS`.

[a] P. D. C. Scholl, „Betriebssysteme“

Tabelle 3.7: *RETI-Knoten*

3.2.5.3 Kompositionen von PicoC-Knoten und RETI-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 **Datensegment Register** `DS` steht auf den **Stack**.
`Ref(Stackframe(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Begin-Aktive-Funktion Register** `BAF` steht auf den **Stack**.
`Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))))`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Stack(Num('addr1'))` steht und dem **Subscript Index**, der an Speicherzelle `Stack(Num('addr2'))` steht und speichert diese auf den **Stack**. Die Berechnung ist abhängig davon ob der **Datentyp** `ArrayDecl(datatype)` oder `PntrDecl(datatype)` ist. Der **Datentyp** ist ein versteckes Attribut von `Ref(exp)`.
`Ref(Attr(Stack(Num('addr1')), Name('attr')))`	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 des **Struct** in `StructSpec(Name('st'))` notwendig, dessen **Attribut** `Name('attr')` ist. `StructSpec(Name('st'))` ist ein verstecktes Attribut von `Ref(exp)`.
`Assign(Stack(Num('size')), Global(Num('addr')))`	Schreibt `Num('size')` viele Speicherzellen, die ab `Global(Num('addr'))` relativ zum **Datensegment Register** `DS` stehen, versetzt genauso auf den **Stack**.
`Assign(Stack(Num('size')), Stackframe(Num('addr')))`	Schreibt `Num('size')` viele Speicherzellen, die ab `Stackframe(Num('addr'))` relativ zum **Begin-Aktive-Funktion Register** `BAF` stehen, versetzt genauso auf den **Stack**.
`Exp(Global(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Datensegment Register** `DS` steht auf den **Stack**.
`Exp(Stackframe(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Begin-Aktive-Funktion Register** `BAF` steht auf den **Stack**.
`Exp(Stack(Num('addr')))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Stackpointer Register** `SP` steht auf den **Stack**.
`Assign(Stack(Num('addr1')), Stack(Num('addr2')))`	Speichert **Inhalt** der Speicherzelle `Stack(Num('addr2'))`, die `Num('addr2')` Speicherzellen **relativ** zum **Stackpointer Register** `SP` steht an der **Adresse** in der Speicherzelle, die `Num('addr1')` Speicherzellen **relativ** zum **Stackpointer Register** `SP` steht.
`Assign(Global(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt genauso auf die Speicherzellen ab `Num('addr')` **relativ** zum **Datensegment Register** `DS`.
`Assign(Stackframe(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt genauso auf die Speicherzellen ab `Num('addr')` **relativ** zum **Begin-Aktive-Funktion Register** `BAF`.
`Exp(Reg(reg))`	Schreibt den aktuellen Wert des **Registers** `reg` auf den **Stack**.
`Instr(Loadi(), [Reg(Acc()), GoTo(Name('addr@next_instr'))])`	Lädt in das Register `ACC` die **Adresse** des Befehls, der in diesem Kontext direkt nach dem Sprung zum Block einer anderen Funktion steht.

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 Kompostionen **ausgelassen**, bei denen einfach nur `exp` durch $\texttt{Stack}(\texttt{Num}('\texttt{x}')), \texttt{x} \in \mathbb{N}$ ersetzt wurde.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen einfach nur eine **Expression** an ein `Exp(exp)` bzw. `Ref(exp)` drangehängt wurde.

3.2.5.4 Abstrakte Grammatik

Die **Abstrakte Syntax** der Sprache L_{PicoC} wird durch die Abstrakte Grammatik 3.2.10 beschrieben.

$stmt$	$::=$	$SingleLineComment(str, str) \mid RETIComment()$	*L_Comment*
un_op	$::=$	$Minus() \mid Not()$	*L_Arith*
bin_op	$::=$	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	$::=$	$Name(str) \mid Num(str) \mid Char(str)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
$stmt$	$::=$	$Exp(\langle exp\rangle)$	
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	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
$stmt$	$::=$	$Assign(\langle exp\rangle, \langle exp\rangle)$	
$datatype$	$::=$	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
exp	$::=$	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
$datatype$	$::=$	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	$::=$	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	$::=$	$StructSpec(Name(str))$	*L_Struct*
exp	$::=$	$Attr(\langle exp\rangle, Name(str))$	
	$\mid$	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
$decl\_def$	$::=$	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
$stmt$	$::=$	$If(\langle exp\rangle, \langle stmt\rangle*)$	*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	$::=$	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
$stmt$	$::=$	$Return(\langle exp\rangle)$	
$decl\_def$	$::=$	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
$file$	$::=$	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

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

Anmerkung

In dieser Schrifftlichen Ausarbeitung der Bachelorarbeit wird oft von der **„Abstrakten Grammatik der Sprache L_{PicoC}“**[a] gesprochen. Gemeint ist mit der Sprache L_{PicoC} **nicht** die Sprache, welche durch die **Abstrakte Grammatik** beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck die **Abstrakt Grammatik** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakt Grammatik** beschrieben wird, interessiert man sich nicht explizit. Diese **Konvention** wurde aus dem Buch G. Siek, *Course Webpage for Compilers (P423,*

P523, E313, and E513) übernommen.

[a] Manchmal auch von der Abstrakten Syntax der Sprache L_{PicoC}.

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** generiert wurde.

```
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 **Konkretten 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__()`[20] 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 **Konkretten 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. Dabei wird nach dem Prinzip der **Tiefensuche** der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene **Textrepräsentation** passend **zusammenfügen** und selbst **zurückgeben**.

Beim **PicoC-Compiler** wurden **Abstrakte** und **Konkrette Syntax** miteinander gemischt. Für **PicoC-Knoten** wurde die **Abstrakte Syntax** verwendet, da Passes schließlich auf **Abstrakten Syntaxbäumen** operieren. Bei **RETI-Knoten** wurde die **Konkrette Syntax** verwendet, da **Maschienenbefehle** in **Konkretter Syntax** schließlich das **Endprodukt** des Kompiliervorgangs sein sollen. Da die **Abstrakte Syntax** von **RETI-Knoten** so simpel ist, macht es kaum einen Unterschied in der Erkennbarkeit, bis auf fehlende gescheifte Klammern `()` usw., ob man die **RETI-Knoten** in **Abstrakter** oder **Konkretter** Syntax schreibt. Daher kann man auch einfach gleich die **RETI-Knoten** in **Konkretter Syntax** ausgeben und muss nicht beim letzten **Pass** daran denken, am Ende die **Konkrette**, statt der **Abstrakten Syntax** für die **RETI-Knoten** auszugeben.

3.3 Code Generierung

Nach der Generierung eines **Abstrakten Syntaxbaumes** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 2.4, wird in diesem Kapitel auf Basis der verschiedenen **Kompositionen** von **PicoC-Knoten** und **RETI-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 entsprechenden **Abstrakten Syntaxbaum** der Sprache L_{RETI} umzuformen. Das ganze lässt sich, wie in Unterkapitel 2.5 bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition 2.45) herunterbricht.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton 2.5). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagram** (siehe Unterkapitel 2.1.1) in Abbildung 3.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte. Und selbst dieses **T-Diagram** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu $\texttt{X}_{86\_64}$ kompiliert wurde usw.

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

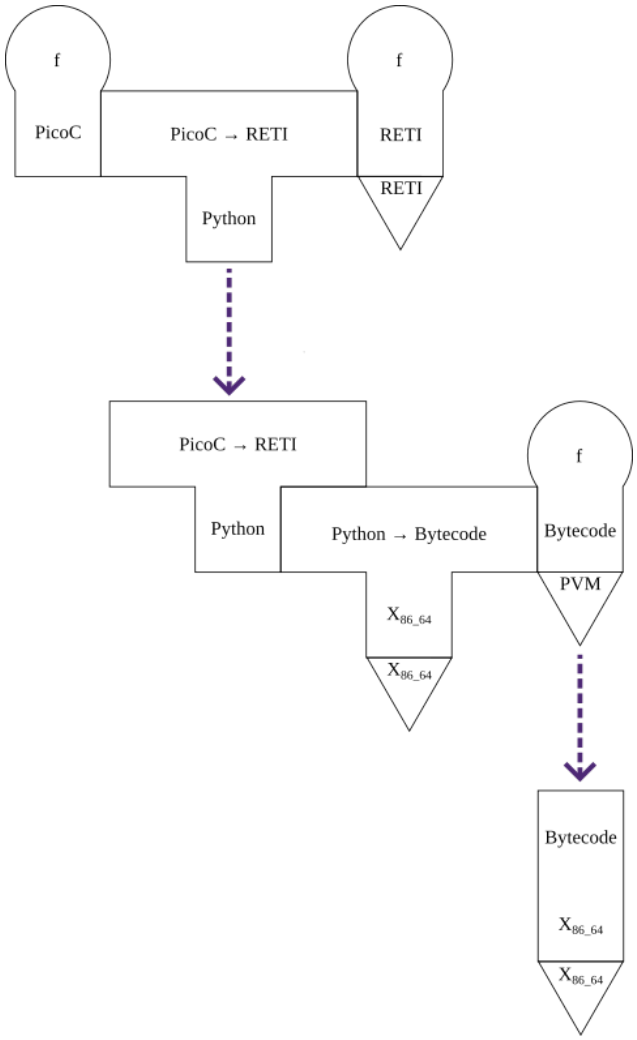


Abbildung 3.6: *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagram** in Abbildung 3.6 lässt sich zusammenfassen, sodass man das **T-Diagram** in Abbildung 3.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in $\mathtt{X}_{86\_64}$-Maschienensprache geschrieben ist.

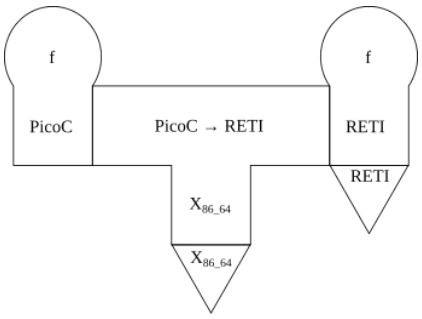


Abbildung 3.7: *Cross-Compiler Kompiliervorgang Kurzform*

Nachdem der Kompilierprozess des **PicoC-Compiler** im **vertikalen** nun genauer angesehen wurde, wird

der Kompilierprozess im Folgenden im **horinzontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 3.8 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 2.29) 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.

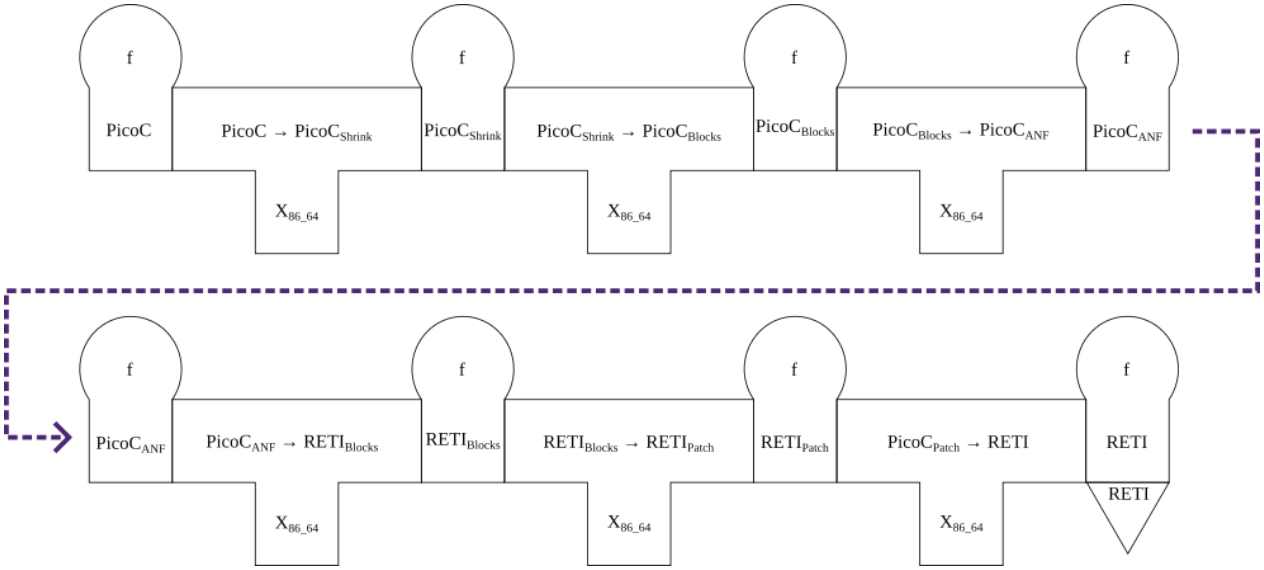


Abbildung 3.8: *Architektur mit allen Passes ausgeschrieben*

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 ausreichend vertieft wurden. Viele der verwendenten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung P. D. 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[21] 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

3.3.1.1.1 Aufgabe

Der 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)` sich in der Untermenge

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

der Sprache L_C, welche die Sprache L_{PicoC} darstellt genau gleich verhält, wie der **Operator** für den **Zugriff** auf den **Index** eines **Feldes** `ar[i]`.

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 einen **Zugriff auf einen Feldindex** `var[i]` übernehmen.

3.3.1.1.2 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(exp, exp)` in Abstrakten Grammatik 3.3.1 **nicht** mehr gibt. Das liegt daran, dass dieser Pass alle **Vorkommnisse** des Knoten `Deref(exp, exp)` durch den Knoten `Subscr(exp, exp)` auswechselt, der ebenfalls nach der **Abstrakten Grammatik** der Sprache L_{PicoC} definiert ist.

stmt	::=	$SingleLineComment(str, str) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(str) \mid Num(str) \mid Char(str)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
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	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(str))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(str))$	
	|	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
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	::=	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	|	$FunDef(\langle datatype\rangle, Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
file	::=	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

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

Anmerkung

Der rot markierte Knoten bedeutet, dass dieser im Vergleich zur voherigen **Abstrakten Grammatik** nicht mehr da ist.

3.3.1.1.3 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 P. D. C. Scholl, „Betriebssysteme“, welcher in der Vorlesung allerdings **rekursiv** implementiert ist.

Dieser **rekursive** Algoirthmus ist allerdings **kein** gutes **Anschaungsbeispiel**, dass 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`-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 3.6 umgeschrieben, um `if-` und `while`-Statemtens 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 in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes 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** erklärt werden und alle sonstigen Details dem **Bachelorprojekt** zuzurechnen sind.

```
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',
6        Name 'faculty',
7        [
```

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

```
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** enthält.

3.3.1.2 PicoC-Blocks Pass

3.3.1.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** 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.

3.3.1.2.2 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 im Unterkapitel 3.3.1.2.1 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 Statements `stmt` mehr. Das resultiert in der **Abstrakten Grammatik** 3.3.2 der Sprache L_{PicoC_Blocks}.

stmt	::=	*SingleLineComment*(*str*, *str*) | *RETIComment*()	*L_Comment*
un_op	::=	*Minus*() | *Not*()	*L_Arith*
bin_op	::=	*Add*() | *Sub*() | *Mul*() | *Div*() | *Mod*()	
	|	*Oplus*() | *And*() | *Or*()	
exp	::=	*Name*(*str*) | *Num*(*str*) | *Char*(*str*)	
	|	*BinOp*(⟨*exp*⟩, ⟨*bin_op*⟩, ⟨*exp*⟩)	
	|	*UnOp*(⟨*un_op*⟩, ⟨*exp*⟩) | *Call*(*Name*('*input*'), *Empty*())	
	|	*Call*(*Name*('*print*'), ⟨*exp*⟩)	
stmt	::=	*Exp*(⟨*exp*⟩)	
un_op	::=	*LogicNot*()	*L_Logic*
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
bin_op	::=	*LogicAnd*() | *LogicOr*()	
exp	::=	*Atom*(⟨*exp*⟩, ⟨*rel*⟩, ⟨*exp*⟩) | *ToBool*(⟨*exp*⟩)	
type_qual	::=	*Const*() | *Writeable*()	*L_Assign_Alloc*
datatype	::=	*IntType*() | *CharType*() | *VoidType*()	
exp	::=	*Alloc*(⟨*type_qual*⟩, ⟨*datatype*⟩, *Name*(*str*))	
stmt	::=	*Assign*(⟨*exp*⟩, ⟨*exp*⟩)	
datatype	::=	*PntrDecl*(*Num*(*str*), ⟨*datatype*⟩)	*L_Pntr*
exp	::=	*Ref*(⟨*exp*⟩)	
datatype	::=	*ArrayDecl*(*Num*(*str*)+, ⟨*datatype*⟩)	*L_Array*
exp	::=	*Subscr*(⟨*exp*⟩, ⟨*exp*⟩) | *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	::=	*If*(⟨*exp*⟩, ⟨*stmt*⟩∗)	*L_If_Else*
	|	*IfElse*(⟨*exp*⟩, ⟨*stmt*⟩∗, ⟨*stmt*⟩∗)	
stmt	::=	*While*(⟨*exp*⟩, ⟨*stmt*⟩∗)	*L_Loop*
	|	*DoWhile*(⟨*exp*⟩, ⟨*stmt*⟩∗)	
exp	::=	*Call*(*Name*(*str*), ⟨*exp*⟩∗)	*L_Fun*
stmt	::=	*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*), ⟨*decl_def*⟩∗)	*L_File*

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

Anmerkung

Alles rot markierte bedeutet, es wurde **entfernt** oder **abgeändert**. Alles ausgegraute bedeutet, es hat sich im Vergleich zur letzten Abstrakten Grammatik **nichts** geändert. Alle normal in `schwarz` geschriebenen Knoten wurden **neu** hinzugefügt.

Die **Abstrakte Grammatik** soll im Gegensatz zur **Konkretten Grammatik** meist nur vom **Programmierer** verstanden werden, der den Compiler implementiert und sollte daher vor allem **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 der **Produktion** `exp` ergibt auch wirklich eingesetzt werden.

3.3.1.2.3 Codebeispiel

In Code 3.8 sieht man den **Abstrakten Syntaxbaum** des **PiocC-Blocks Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion `faculty` und die `main`-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement 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')),
              [
```

```
37                GoTo(Name('if.3'))
38              ],
39              [
40                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

3.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** 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 Wiederholung zu vermeiden wird zur Erklärung der **A-Normalform** auf Unterkapitel 2.5.2 verwiesen.

Zudem wird eine **Symboltabelle** (Definition 3.8) eingeführt. In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** 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**. Einer der beiden **Counter** wird entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle diese Variable `Name('symbol')` vor, so wird mit dem **Symbol**[23] als Schlüssel in der **Symboltabelle** nachgeschlagen und anstelle des `Name(str)`-Knotens die in

[23] Bzw. der **Bezeichner**

der **Symboltabelle** nachgeschlagene Adresse in einem `Global(Num('addr'))`- bzw. `Stackframe(Num('addr'))`-Knoten eingesetzt eingefügt. Ob der `Global(Num('addr'))`- oder der `Stackframe(Num('addr'))`-Knoten zum Einsatz kommt, entscheidet sich anhand des **Scopes** (z.B. `@scope`), der in der **Symboltabelle** an den **Bezeichner** drangehängt ist (z.B. `identifier@scope`).[24]

Definition 3.8: Symboltabelle

*Eine über ein **Assoziatives Feld** umgesetzte **Datenstruktur**, die notwendig ist, um das Konzept einer **Variablen** 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.*

*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 einer **Symboltabelle** werden **Bezeichner** als **Symbole** bezeichnet.

3.3.1.3.2 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**, wie z.B. `Stack(Num(str))` enthalten können. Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass u.a. die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `File(Name(str), block*)`-Knoten liegen usw., was in Unterkapitel 3.3.6 genauer erklärt wird. Die **Symboltabelle** ist ebenfalls als **Abstrakter Syntaxbaum** umgesetzt, wofür in der Abstrakten Grammatik 3.3.3 der Sprache L_{PicoC_ANF} der Sprache L_{PicoC_ANF} neue Knoten eingeführt werden.

Das ganze resultiert in der Abstrakten Grammatik 3.3.3 der Sprache L_{PicoC_ANF}.

[24] Die Umsetzung von **Scopes** wird in Unterkapitel 3.3.6.2 genauer beschrieben.

stmt	::=	*SingleLineComment*(*str*, *str*) | *RETIComment*()	*L_Comment*
un_op	::=	*Minus*() | *Not*()	*L_Arith*
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}

3.3.1.3.3 Codebeispiel

In Code 3.9 sieht man den **Abstrakten Syntaxbaum** des **PiocC-ANF Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem alls Statements und Ausdrücke in **A-Normalform** sind. Die `IfElse(exp, stmts, stmts)`-Knoten sind hier in **A-Normalform** gebracht worden, indem ihre **Komplexe Bedingung** vorgezogen wurde und das Ergebnis der **Komplexen Bedingung** einer **Location** zugewiesen ist und sie selbst das Ergebnis über den **Atomaren Ausdruck** `Stack(Num(str))` vom Stack lesen: `IfElse(Stack(Num(str)), stmts, stmts)`. **Funktionen** sind nur noch über die **Labels** von Blöcken zu erkennen, die den gleichen Bezeichner haben, wie die ursprüngliche Funktion und es lässt sich nur durch das **Nachverfolgen** der `GoTo(Name('label'))`-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'))
         ]
     ],
```

```
46    Block
47      Name 'if.3',
48      [
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'), 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 3.9: *PicoC-ANF Pass für Codebespiel*

3.3.1.4 RETI-Blocks Pass

3.3.1.4.1 Aufgabe

Die Aufgabe des **RETI-Blocks Passes** ist es die **Statements** in den **Blöcken**, die durch **PicoC-Knoten** im **Abstrakten Syntaxbaum** der Sprache L_{PicoC_ANF} dargestellt sind durch ihren entsprechenden **RETI-Knoten** zu ersetzen.

3.3.1.4.2 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 entsprechende **RETI-Knoten** ersetzt. 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 3.3.8 aufgelöst wird.

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

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

3.3.1.4.3 Codebeispiel

In Code 3.10 sieht man den **Abstrakten Syntaxbaum** des **RETI-Blocks Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem die **Statements**, die durch entsprechende **PicoC-Knoten** im **Abstrakten Syntaxbaum** der Sprache L_{PicoC_ANF}[25] repräsentiert waren nun durch ihre entsprechennden **RETI-Knoten** ersetzt werden.

```
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',
      [
```

[25] Beschrieben durch die Grammatik 3.3.3.

```
23        # // IfElse(Num('1'), [], [])
24        # Exp(Num('1'))
25        SUBI SP 1;
26        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'), 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 3.10: *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 **Konkretter Syntax** ausgeben werden.

3.3.1.5 RETI-Patch Pass

3.3.1.5.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist das **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes**, durch:

- das **Einfügen** eines `start.<nummer>`-Blockes, 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**, die vor jeder **Division** `Instr(Div(), args)` prüfen, ob, nicht durch `0` geteilt wird.[26]
- das Überprüfen darauf, ob bestimmte **Immediates** `Im(str)` in Befehlen, wie z.B. `Jump(rel, Im(str))`, `Instr(Loadin(), [reg, reg, Im(str)])`, `Instr(Loadi(), [reg, Im(str)])` usw. **kleiner** -2^{21} oder **größer** $2^{21} - 1$ sind. Im Fall dessen, dass es so ist, muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **bitweise Oder** berechnet werden. Im Fall, dessen, dass der **Immediate** allerdings **kleiner** $-(2^{31})$ oder **größer** $2^{31} - 1$ ist, wird eine Fehlermeldung `TooLargeLiteral` ausgegeben.

[26] Das fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

3.3.1.5.2 Abstrakte Grammatik

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} kaum verändert. Es muss nur ein Knoten `Exit()` hinzugefügt werden, der im Falle einer **Division durch 0** die Ausführung des Programs beendet.

reg	::=	*ACC*() | *IN*1() | *IN*2() | *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*)))	
instr	::=	*Exp*(*GoTo*(*str*)) | *Exit*(*Num*(*str*))	*L_PicoC*
block	::=	*Block*(*Name*(*str*), ⟨*instr*⟩∗)	
file	::=	*File*(*Name*(*str*), ⟨*block*⟩∗)	

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

3.3.1.5.3 Codebeispiel

In Code 3.11 sieht man den **Abstrakten Syntaxbaum** des **PiocC-Patch Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel. 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))`-**Statements** entfernt[28], die nur einen Sprung um **eine** Position entsprochen hätten.

```
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 **Block** wurde im Code 3.8 markiert.
[28] Diese **entfernten** `GoTo(Name(str))`'s' wurden ebenfalls im Code 3.8 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'), 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 3.11: *RETI-Patch Pass für Codebespiel*

3.3.1.6 RETI Pass

3.3.1.6.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist es 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 eine entsprechende **Adresse** zu ersetzen, die entsprechende **Distanz** oder einen entsprechenden **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))`. Die **Distanz-** und **Adressberechnung** wird in Unterkapitel 3.3.6.3 genauer mit **Formeln** erklärt.

3.3.1.6.2 Konkrette und Abstrakte Grammatik

Die Abstrakte Grammatik 3.3.8 der Sprache L_{RETI} hat im Vergleich zur Abstrakten Grammatik 3.3.5 der Sprache L_{RETI_Patch} nur noch auschließlich **RETI-Knoten**. Alle **RETI-Knoten** stehen nun in einem `Program(Name(str), instr)`-Knoten.

Ausgegeben wird der finale **Maschinencode** allerdings in **Konkretter Syntax**, die durch die Konkretten Grammatiken 3.3.6 und 3.3.7 für jeweils die **Lexikalische** und **Syntaktische Analyse** beschrieben wird. Der Grund, warum die **Konkrette Grammatik** der Sprache L_{RETI} auch nochmal in einen Teil für die **Lexikalische** und **Syntaktische Analyse** unterteilt ist, hat den Grund, dass für die Bachelorarbeit zum

Testen des **PicoC-Compilers** ein **RETI-Interpreter** implementiert wurde, der den RETI-Code **lexen** und **parsen** muss, um ihn später **interpretieren** zu können.

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*" | "*IN*1" | "*IN*2" | "*PC*" | "*SP*"	
		| "*BAF*" | "*CS*" | "*DS*"	
arg	::=	*reg* | *num*	
rel	::=	"==" | "!=" | "<" | "<=" | ">"	
		| ">=" | "*NOP*"	

Grammatik 3.3.6: *Konkrette 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	::=	*name* (*instr*";")$*$	

Grammatik 3.3.7: *Konkrette Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF*

reg	::=	$ACC() \mid IN1() \mid IN2() \mid PC() \mid SP() \mid BAF()$	*L_RETI*
		$\mid CS() \mid DS()$	
arg	::=	$Reg(\langle reg \rangle) \mid Num(str)$	
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	::=	$Instr(\langle op \rangle, \langle arg \rangle+) \mid Jump(\langle rel \rangle, Num(str)) \mid Int(Num(str))$	
		$\mid RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)$	
		$\mid SingleLineComment(str, str)$	
		$\mid Instr(Loadi(), [Reg(Acc()), GoTo(Name(str))]) \mid Jump(Eq(), GoTo(Name(str)))$	
program	::=	$Program(Name(str), \langle instr \rangle*)$	
instr	::=	$Exp(GoTo(str)) \mid Exit(Num(str))$	*L_PicoC*
block	::=	$Block(Name(str), \langle instr \rangle*)$	
file	::=	$File(Name(str), \langle block \rangle*)$	

Grammatik 3.3.8: *Abstrakte Grammatik der Sprache* L_{RETI}

3.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkretten Syntax** ausgegeben werden, wie in Code 3.12 zu sehen ist. Es gibt **keine Blöcke** mehr und die **RETI-Befehle** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** auschließlich aus **RETI-Ausdrücken** bestehen[29], die sich in den **Blöcken** befunden haben, wurden durch **RETI-Befehle** ersetzt.

Der `Program(Name(str), instr)`-Knoten, indem alle **RETI-Knoten** stehen gibt alleinig die **RETI-Knoten**, die er beinhaltet aus und fügt ansonsten nichts hinzu, wodurch der **Abstrakte Syntaxbaum**, wenn er in eine Datei ausgegeben wird, direkt **RETI-Code** in **menschenlesbarer Repräsentation** erzeugt.

```
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;
```

[29] Wie z.B. `LOADI ACC GoTo(Name('addr@next_instr'))`, `Exp(GoTo(Name('main.0')))` und `JUMP== GoTo(Name('if_else_after.2'))`.

```
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;
75 STOREIN BAF ACC -3;
```

```
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1')))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5')))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;
100 # StackMalloc(Num('2'))
101 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6')))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 3.12: *RETI Pass für Codebespiel*

3.3.2 Umsetzung von Zeigern

3.3.2.1 Referenzierung

Die **Referenzierung** (z.B. `&var`) wird im Folgenden anhand des Beispiels in Code 3.13 erklärt.

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

Code 3.13: *PicoC-Code für Zeigerreferenzierung*

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 3.14 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 3.14: *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 3.15.

Anmerkung

Die **Größe** eines Zeigers (z.B. eines Zeigers auf ein Feld von `int`: `pntr = int *pntr[3]`), die ihm `size`-Attribut der **Symboltabelle** eingetragen ist, ist dabei immer: `size(pntr)` = 1.

```
1 SymbolTable
2   [
3     Symbol
4       {
5         type qualifier:         Empty()
6         datatype:               FunDecl(VoidType('void'), Name('main'), [])
7         name:                   Name('main')
8         value or address:       Empty()
9         position:               Pos(Num('1'), Num('5'))
```

```
10        size:                   Empty()
11      },
12    Symbol
13      {
14        type qualifier:         Writeable()
15        datatype:               IntType('int')
16        name:                   Name('var@main')
17        value or address:       Num('0')
18        position:               Pos(Num('2'), Num('6'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         Writeable()
24        datatype:               PntrDecl(Num('1'), IntType('int'))
25        name:                   Name('pntr@main')
26        value or address:       Num('1')
27        position:               Pos(Num('3'), Num('7'))
28        size:                   Num('1')
29      }
30  ]
```

Code 3.15: *Symboltabelle für Zeigerreferenzierung*

Im **PicoC-ANF Pass** in Code 3.16 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')))` ist, sind noch weitere Anweisungen zwischen den Zeilen `11` und `12` nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 3.3.5.3 kümmern.

```
1 File
2   Name './example_pntr_ref.picoc_mon',
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 3.16: *PicoC-ANF Pass für Zeigerreferenzierung*

Im **RETI-Blocks Pass** in Code 3.17 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_pntr_ref.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('pntr'), 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;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 3.17: *RETI-Blocks Pass für Zeigerreferenzierung*

3.3.2.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

Die **Dereferenzierung** (z.B. `*var`) wird im Folgenden anhand des Beispiels in Code 3.18 erklärt.

```
void main() {
  int var = 42;
  int *pntr = &var;
  *pntr;
}
```

Code 3.18: *PicoC-Code für Zeigerdereferenzierung*

Der Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 3.19 eine **Dereferenzierung** `*var`. Es gibt herbei **zwei** Fälle. Bei der Anwendung von **Zeigerarithmetik**, wie z.B. `*(var + 2 - 1)` übersetzt sich diese zu `Deref(Name('var'), BinOp(Num('2'), Sub(), BinOp(Num('1'))))`. Bei einer normalen **Dereferenzierung**, wie z.B. `*var`, übersetzt sich diese zu `Deref(Name('var'), Num('0'))`.

```
File
  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 3.19: *Abstrakter Syntaxbaum für Zeigerdereferenzierung*

Im **PicoC-Shrink Pass** in Code 3.20 wird ein Trick angewandet, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Arrayindex** (z.B. `var[1]`) verhält. Damit sparrt man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Arrayindex** (z.B. `var[1]`) übernehmen lassen.

```
1 File
2   Name './example_pntr_deref.picoc_shrink',
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(Subscr(Name('pntr'), Num('0')))
12      ]
13  ]
```

Code 3.20: *PicoC-Shrink Pass für Zeigerdereferenzierung*

3.3.3 Umsetzung von Arrays

3.3.3.1 Initialisierung von Arrays

Die **Initialisierung** eines **Arrays** (z.B. `int ar[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 3.21 erklärt.

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

```
5 void fun() {
6   int ar[2][2] = {{3, 4}, {5, 6}};
7 }
```

Code 3.21: *PicoC-Code für Array Initialisierung*

Die **Initialisierung** eines **Feldes** `intar[2][1]={{3+1},{4}}` wird im **Abstrakten Syntaxbaum** in Code 3.22 mithilfe der Komposition `Assign(Alloc(Writeable(),ArrayDecl([Num('2'),Num('1')],IntType('int')),Name('ar')),Array([Array([BinOp(Num('3'),Add('+'),Num('1'))]),Array([Num('4')])]))` dargestellt.

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

Code 3.22: *Abstrakter Syntaxbaum für Array Initialisierung*

Bei der **Initialisierung** eines **Arrays** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[30]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 3.23 dargestellt ist.

Bei Variablen auf dem **Stackframe** wird ein Array **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Arrays genommen. Dies macht den **Zugriff auf einen Arrayindex** in Subkapitel 3.3.3.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf einen Arrayindex** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die entgegengesetzte Richtung wächst, verglichen mit den **Globalen Statischen Daten**[31].

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

[31] Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

Anmerkung

Das **Größe** des Arrays `datatype ar`$[\texttt{dim}_1]\ldots[\texttt{dim}_k]$, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Arrays multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Arrayelemente**: $\texttt{size(datatype(ar))} = \left(\prod_{j=1}^{n} \texttt{dim}_j\right) \cdot \texttt{size(datatype)}$[a].

[a] Die **Funktion** `type` ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die **Funktion** `size` nur bei einem **Datentyp** als **Funktionsargument** die **Größe dieses Datentyps** als **Zielwert** liefert

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                FunDecl(VoidType('void'), Name('main'), [])
7          name:                    Name('main')
8          value or address:        Empty()
9          position:                Pos(Num('1'), Num('5'))
10         size:                    Empty()
11       },
12     Symbol
13       {
14         type qualifier:          Writeable()
15         datatype:                ArrayDecl([Num('2'), Num('1')], IntType('int'))
16         name:                    Name('ar@main')
17         value or address:        Num('0')
18         position:                Pos(Num('2'), Num('6'))
19         size:                    Num('2')
20       },
21     Symbol
22       {
23         type qualifier:          Empty()
24         datatype:                FunDecl(VoidType('void'), Name('fun'), [])
25         name:                    Name('fun')
26         value or address:        Empty()
27         position:                Pos(Num('5'), Num('5'))
28         size:                    Empty()
29       },
30     Symbol
31       {
32         type qualifier:          Writeable()
33         datatype:                ArrayDecl([Num('2'), Num('2')], IntType('int'))
34         name:                    Name('ar@fun')
35         value or address:        Num('3')
36         position:                Pos(Num('6'), Num('6'))
37         size:                    Num('4')
38       }
39   ]
```

Code 3.23: *Symboltabelle für Array Initialisierung*

Im **PiocC-Mon Pass** in Code 3.24 werden zuerst die **Logischen Ausdrücke** in den Blättern des Teilbaumes, der beim **Array-Initializers** Knoten `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),`

`Array([Num('4')])])` anfängt nach dem Prinzip der **Tiefensuche**, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben[32].

Im finalen Schritt muss zwischen **Globalen Statischen Daten** bei der `main`-Funktion und **Stackframe** bei der Funktion `fun` unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))`, die in Tabelle 3.8 genauer beschrieben ist, versetzt in der selben Reihenfolge zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.

Der **Trick** ist hier, dass egal wieviele Dimensionen und was für einen Datentyp das **Array** hat, man letztendlich immer das gesamte Array erwischt, wenn man einfach die **Größe des Arrays** viele **Speicherzellen** mit z.B. der **Komposition** `Assign(Global(Num('0')), Stack(Num('2')))` verschiebt.

In die Knoten `Global('0')` und `Stackframe('3')` wurde hierbei die **Startadresse** des jeweiligen Arrays geschrieben, sodass man nach dem **PicoC-ANF Pass** nie mehr Variablen in der **Symboltabelle** nachsehen muss und gleich weiß, ob sie in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

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

Code 3.24: *PicoC-ANF Pass für Array Initialisierung*

Im **RETI-Blocks Pass** in Code 3.25 werden die **Kompositionen** `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

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

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        LOADIN SP ACC 2;
        LOADIN SP IN2 1;
        ADD ACC IN2;
        STOREIN SP ACC 2;
        ADDI SP 1;
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // 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'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('5'))
        SUBI SP 1;
        LOADI ACC 5;
        STOREIN SP ACC 1;
        # Exp(Num('6'))
        SUBI SP 1;
        LOADI ACC 6;
```

```
54          STOREIN SP ACC 1;
55          # Assign(Stackframe(Num('3')), Stack(Num('4')))
56          LOADIN SP ACC 1;
57          STOREIN BAF ACC -2;
58          LOADIN SP ACC 2;
59          STOREIN BAF ACC -3;
60          LOADIN SP ACC 3;
61          STOREIN BAF ACC -4;
62          LOADIN SP ACC 4;
63          STOREIN BAF ACC -5;
64          ADDI SP 4;
65          # Return(Empty())
66          LOADIN BAF PC -1;
67        ]
68    ]
```

Code 3.25: *RETI-Blocks Pass für Array Initialisierung*

3.3.3.2 Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 3.26 erklärt.

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

Code 3.26: *PicoC-Code für Zugriff auf einen Arrayindex*

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im **Abstrakten Syntaxbaum** in Code 3.27 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 'main',
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 'fun',
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 3.27: *Abstrakter Syntaxbaum für Zugriff auf einen Arrayindex*

Im **PicoC-ANF Pass** in Code 3.28 wird vom Knoten `Subscr(Name('ar'), Num('0'))` zuerst im **Anfangsteil** 3.3.5.2 die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktionm `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil** 3.3.5.3 wird die **Adresse** ab der das **Arrayelement** des Arrays auf das Zugegriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Arrays, in dem dieses **Arrayelement** liegt auf den **Stack** gelegt. Da ein **Index** auf den Zugegriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `ar[1 + var]` bestimmt sein kann, indem auch **Variablen** vorkommen können, 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 bestimmt werden, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle 3.8 dokumentiert.

Zur **Adressberechnung** ist es notwendig auf die **Dimensionen** (z.B. `[Num('3')]`) des Arrays, auf dessen **Arrayelement** zugegriffen wird, zugreifen zu können. Daher ist der **Arraydatentyp** (z.B. `ArrayDecl([Num('3')], IntType('int'))`) dem Knoten `Ref(exp, datatype)` als versteckte Attribut `datatype` angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon 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 und es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und eine **Zugriff auf einen Arryindex** `var[1]` miteinander zu kombinieren, was in Subkapitel 3.3.5.3 allgemein erklärt ist.

Im letzten Schritt, dem **Schlussteil** 3.3.5.4 wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde, nun auf den **Stack** geschrieben, wobei dieser die **Adresse** auf dem Stack ersetzt, die es zum Finden des **Index** brauchte. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. Je nachdem, welchen **Datentyp** die Variable `ar` hat und auf welchen **Unterdatentyp** folglich im **Kontext** zuletzt zugegriffen wird, abhängig davon wird der **Schlussteil** `Exp(Stack(Num('1')))` auf eine andere Weise verarbeitet (siehe Subkapitel 3.3.5.4). Der **Unterdatentyp** ist dabei ein versteckte Attribut des `Exp(Stack(Num('1')))`-Knoten.

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Arrayindex** (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, bei dem unterschiedliche **RETI-Befehle** für eine Variable, die in den **Globalen Statischen Daten** liegt und eine Variable, die auf dem **Stackframe** liegt erzeugt werden

müssen.

Anmerkung

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist, kann mittels der Formel 3.3.1:

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

aus der Betriebssysteme Vorlesung P. D. C. Scholl, „Betriebssysteme“ berechnet werden[a].

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden `ref(ar)` in der Formel.

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

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

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

[a] `ref(exp)` steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte.

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

```
29        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
30        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
31        Exp(Stack(Num('1')))
32        Return(Empty())
33      ]
34  ]
```

Code 3.28: *PicoC-ANF Pass für Zugriff auf einen Arrayindex*

Im **RETI-Blocks Pass** in Code 3.29 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')) und Stack(Num('1'))))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_array_access.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([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        # // Exp(Subscr(Name('ar'), Num('0')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Exp(Num('0'))
23        SUBI SP 1;
24        LOADI ACC 0;
25        STOREIN SP ACC 1;
26        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        LOADIN SP IN1 2;
28        LOADIN SP IN2 1;
29        MULTI IN2 1;
30        ADD IN1 IN2;
31        ADDI SP 1;
32        STOREIN SP IN1 1;
33        # Exp(Stack(Num('1')))
34        LOADIN SP IN1 1;
35        LOADIN IN1 ACC 0;
36        STOREIN SP ACC 1;
37        # Return(Empty())
38        LOADIN BAF PC -1;
39      ],
40    Block
41      Name 'fun.0',
```

```
42      [
43        # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44        # Exp(Num('1'))
45        SUBI SP 1;
46        LOADI ACC 1;
47        STOREIN SP ACC 1;
48        # Exp(Num('2'))
49        SUBI SP 1;
50        LOADI ACC 2;
51        STOREIN SP ACC 1;
52        # Exp(Num('3'))
53        SUBI SP 1;
54        LOADI ACC 3;
55        STOREIN SP ACC 1;
56        # Assign(Stackframe(Num('2')), Stack(Num('3')))
57        LOADIN SP ACC 1;
58        STOREIN BAF ACC -2;
59        LOADIN SP ACC 2;
60        STOREIN BAF ACC -3;
61        LOADIN SP ACC 3;
62        STOREIN BAF ACC -4;
63        ADDI SP 3;
64        # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65        # Ref(Stackframe(Num('2')))
66        SUBI SP 1;
67        MOVE BAF IN1;
68        SUBI IN1 4;
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 3.29: *RETI-Blocks Pass für Zugriff auf einen Arrayindex*

3.3.3.3 Zuweisung an Arrayindex

Die **Zuweisung** eines Wertes an einen **Arrayindex** (z.B. `ar[2] = 42;`) wird im Folgenden anhand des Beispiels in Code 3.30 erläutert.

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

Code 3.30: *PicoC-Code für Zuweisung an Arrayindex*

Im **Abstrakten Syntaxbaum** in Code 3.31 wird eine **Zuweisung** an einen **Arrayindex** `ar[2] = 42;` durch die Komposition `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

```
1  File
2    Name './example_array_assignment.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
10         Assign(Subscr(Name('ar'), Num('1')), Num('42'))
11       ]
12   ]
```

Code 3.31: *Abstrakter Syntaxbaum für Zuweisung an Arrayindex*

Im **PicoC-ANF Pass** in Code 3.32 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=`, bzw. des Knotens der diesen darstellt ausgewertet: `Exp(Num('42'))`.

Danach ist das Vorgehen, bzw. sind die Kompostionen, die dieses darauffolgende 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 Subkapitel 3.3.3.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird berechet, wie in Subkapitel 3.3.3.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`[33] die Zuweisung `=` des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Arrayindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

[33] Ist in Tabelle 3.8 genauer beschrieben ist

```
File
  Name './example_array_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Exp(Num('1'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 3.32: *PicoC-ANF Pass für Zuweisung an Arrayindex*

Im **RETI-Blocks Pass** in Code 3.33 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre 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;
```

```
33          # Return(Empty())
34          LOADIN BAF PC -1;
35        ]
36    ]
```

Code 3.33: *RETI-Blocks Pass für Zuweisung an Arrayindex*

3.3.4 Umsetzung von Structs

3.3.4.1 Deklaration und Definition von Structtypen

Die **Deklaration** eines neuen **Structtyps** (z.B. `struct st {int len; int ar[2];}`) und die **Definition** einer Variable mit diesem **Structtyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 3.34 erläutert.

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

Code 3.34: *PicoC-Code für die Deklaration eines Structtyps*

Bevor irgendwas definiert werden kann, muss erstmal ein **Structtyp** deklariert werden. Im **Abstrakten Syntaxbaum** in Code 3.36 wird die **Deklaration eines Structtyps** `struct st {int len; int ar[2];}` durch die Komposition `StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` dargestellt.

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

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

Code 3.35: *Abstrakter Syntaxbaum für die Deklaration eines Structtyps*

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 3.36 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Attribute dieses Symbols `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Attribut des Symbols die Attribute des **Structtyps** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Structtyp** `st` die **Attribute** des Structtyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich,

dass **innerhalb** eines **Structtyps** zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Structtypen** ihre Attribute gleich benennen können.

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

Für die **Definition** einer Variable `st_var@main` mit diesem **Structtyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatyp`-Attribut enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structyp**[34] und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Anmerkung

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

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

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               IntType('int')
        name:                   Name('len@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('ar@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('24'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
        ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
        ↪  Name('ar'))])
        name:                   Name('st')
        value or address:       [Name('len@st'), Name('ar@st')]
        position:               Pos(Num('1'), Num('7'))
```

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

```
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 3.36: *Symboltabelle für die Deklaration eines Structtyps*

3.3.4.2 Initialisierung von Structs

Die **Initialisierung eines Structs** wird im Folgenden mithilfe des Beispiels in Code 3.37 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 3.37: *PicoC-Code für Initialisierung von Structs*

Im **Abstrakten Syntaxbaum** in Code 3.38 wird die **Initialisierung eines Structs** `struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}}` mithilfe der **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` dargestellt.

```
1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
          ↪  Name('attr'))
```

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

Code 3.38: *Abstrakter Syntaxbaum für Initialisierung von Structs*

Im **PicoC-ANF Pass** in Code 3.39 wird die **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Arrays** in Subkapitel 3.3.3.1 daher wird um keine Wiederholung zu betreiben auf Subkapitel 3.3.3.1 verwiesen. Um das ganze interressanter zu gestalten wurde das Beispiel in Code 3.37 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der **Struct-Initializer** Teilbaum `Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))])])`, der beim **Struct-Initializer** Knoten anfängt, wird auf dieselbe Weise nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Arrays** in Subkapitel 3.3.3.1 bereits erklärt wurde.

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

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

```
1 File
2   Name './example_struct_init.picoc_mon',
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('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
          ↪  Ref(Name('var'))]))]))]))
11        Exp(Global(Num('0')))
12        Ref(Global(Num('0')))
13        Ref(Global(Num('0')))
14        Assign(Global(Num('1')), Stack(Num('3')))
15        Return(Empty())
16      ]
17  ]
```

Code 3.39: *PicoC-ANF Pass für Initialisierung von Structs*

Im **RETI-Blocks Pass** in Code 3.40 werden die **Kompositionen** `Exp(exp)`, `Ref(exp)` und `Assign(Global(Num('1')), Stack(Num('3')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_struct_init.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('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
          ↪  Ref(Name('var'))]))]))]))
17        # Exp(Global(Num('0')))
18        SUBI SP 1;
19        LOADIN DS ACC 0;
20        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 3.40: *RETI-Blocks Pass für Initialisierung von Structs*

3.3.4.3 Zugriff auf Structattribut

Der **Zugriff auf ein Structattribut** (z.B. `st.y`) wird im Folgenden mithilfe des Beispiels in Code 3.41 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 3.41: *PicoC-Code für Zugriff auf Structattribut*

Im **Abstrakten Syntaxbaum** in Code 3.42 wird der **Zugriff auf ein Structattribut** `st.y` mithilfe der **Komposition** `Exp(Attr(Name('st'), Name('y')))` dargestellt.

```
1  File
2    Name './example_struct_attr_access.ast',
3    [
4      StructDecl
5        Name 'pos',
6        [
7          Alloc(Writeable(), IntType('int'), Name('x'))
8          Alloc(Writeable(), IntType('int'), Name('y'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
           ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
16         Exp(Attr(Name('st'), Name('y')))
17       ]
18   ]
```

Code 3.42: *Abstrakter Syntaxbaum für Zugriff auf Structattribut*

Im **PicoC-ANF Pass** in Code 3.43 wird die Komposition `Exp(Attr(Name('st'), Name('y')))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Exp(Subscr(Name('ar'), Num('0')))` in Subkapitel 3.3.3.2 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel 3.3.3.2 verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Subkapitel 3.3.3.2 durch Kompositionen ersetzt, die sich in **Anfangsteil** 3.3.5.2, **Mittelteil** 3.3.5.3 und **Schlussteil** 3.3.5.4 aufteilen lassen. In diesem Fall sind es `Ref(Global(Num('0')))` (**Anfangsteil**), `Ref(Attr(Stack(Num('1')), Name('y')))` (**Mittelteil**) und `Exp(Stack(Num('1')))` (**Schlussteil**). Der **Anfangsteil** und **Schlussteil** sind genau gleich, wie in Subkapitel 3.3.3.2.

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

Im Gegensatz zur Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` beim **Zugriff auf einen Arrayindex** in Subkapitel 3.3.3.2, muss hier vorher nichts anderes als die **Anfangsadresse** des **Structs** auf dem **Stack** liegen. Das **Structattribut** auf welches zugegriffen wird steht bereits in der Komposition `Ref(Attr(Stack(Num('1')), Name('y')))`, nämlich `Name('y')`. Den **Structtyp**, dem dieses Attribut gehört, kann man aus dem versteckten Attribut `datatype` herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon Pass** dem Knoten `Ref(exp, datatype)` angehängt.

Anmerkung

Sei $\texttt{datatype}_\texttt{i}$ ein **Knoten** eines **entarteten Baumes** (siehe Definition 3.9 und Abbildung 3.3.2), dessen Wurzel $\texttt{datatype}_1$ ist. Dabei steht $\texttt{i}$ für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\texttt{ref}(\texttt{datatype}_\texttt{i})$ von Speicherbereichen $\texttt{ref}(\texttt{datatype}_\texttt{i})$ ... $\texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$ im **Hauptspeicher** zuordnen, wobei gilt, dass $\texttt{ref}(\texttt{datatype}_\texttt{i}) \leq \texttt{ref}(\texttt{datatype}_{\texttt{i+1}}) < \texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$.[ab]

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

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

(3.3.2)

Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen $(\texttt{datatype}_{\texttt{1,idx}_1}, \ldots, \texttt{datatype}_{\texttt{n,idx}_\texttt{n}})$, die das Resultat einer Aneinandereihung von **Zugriffen**

auf **Zeigerelemente**, **Arrayelemente** und **Verbundattributte** unterschiedlicher Datentypen $\texttt{datatype}_\texttt{i}$ ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 3.3.3:

$$\texttt{ref}(\texttt{datatype}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{datatype}_{n,\texttt{idx}_n}) = \texttt{ref}(\texttt{datatype}_1) + \sum_{i=1}^{n-1}\sum_{k=1}^{idx_i-1} \text{size}(\texttt{datatype}_{i,k}) \quad (3.3.3)$$

berechnet werden.[c]

Dabei darf nur der letzte Knoten $\texttt{datatype}_\texttt{n}$ vom Datentyp **Zeiger** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Zeiger**, der nicht der **letzte Datentyp** $\texttt{datatype}_\texttt{n}$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp $\texttt{datatype}_1$ bis direkt zum Dantentyp Zeiger geht $\texttt{datatype}_\texttt{pntr}$ und die **zweite Adressberechnung** einen Dantentyp nach dem Datentyp Zeiger anfängt $\texttt{datatpye}_{\texttt{pntr}+1}$ und bis zum letzten Datenyp $\texttt{datatype}_\texttt{n}$ geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\texttt{ref}(\texttt{datatype}_1)$ des Summanden aus der Formel 3.3.3 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $\texttt{M}\left[\texttt{ref}(\texttt{datatype}_1,\ \ldots,\ \texttt{datatype}_\texttt{pntr})\right]$ gesetzt werden.

Die Formel 3.3.3 stellt dabei eine **Verallgemeinerung** der Formel 3.3.1 dar, die für alle möglichen Aneinandereihungen von Zugriffen auf **Zeigerelemente**, **Arrayelementen** und **Structattribute** 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 den gleichen Datentyp haben[d].

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

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

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

[a] Es ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.
[b] `ref(datatype)` steht dabei für das Schreiben der **Startadresse**, die dem **Datentyp** `datatype` zugeordnet ist auf den **Stack**.
[c] Die **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Zeigerelmente**, **Feldelemente** oder **Structattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** $\texttt{datatype}_\texttt{i}$, die vor dem **Element** / **Attribut** $\texttt{datatype}_{\texttt{i},\texttt{idx}_\texttt{i}}$ liegen.
[d] Structattribute haben **unterschiedliche** Größen.

Definition 3.9: Entarteter Baum

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

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

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

[a] *Bäume.*

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

Code 3.43: *PicoC-ANF Pass für Zugriff auf Structattribut*

Im **RETI-Blocks Pass** in Code 3.44 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')), Name('y')))` und `Exp(Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_attr_access.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # // Exp(Attr(Name('st'), Name('y')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # 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 3.44: *RETI-Blocks Pass für Zugriff auf Structattribut*

3.3.4.4 Zuweisung an Structattribut

Die **Zuweisung an ein Structattribut** (z.B. `st.y = 42`) wird im Folgenden anhand des Beispiels in Code 3.45 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 3.45: *PicoC-Code für Zuweisung an Structattribut*

Im **Abstact Syntax Tree** wird eine **Zuweisung an ein Structattribut** (z.B. `st.y = 42`) durch die Komposition `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 3.46: *Abstrakter Syntaxbaum für Zuweisung an Structattribut*

Im **PicoC-ANF Pass** in Code 3.47 wird die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` in Subkapitel 3.3.3.3 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel 3.3.3.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 3.3.3.3 muss hier für das Auswerten des **linken** Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Subkapitel 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 3.47: *PicoC-ANF Pass für Zuweisung an Structattribut*

Im **RETI-Blocks Pass** in Code 3.48 werden die **Kompositionen** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')), Name('y')))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

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

```
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          # // 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 3.48: *RETI-Blocks Pass für Zuweisung an Structattribut*

3.3.5 Umsetzung des Zugriffs auf Derived datatypes im Allgemeinen

3.3.5.1 Übersicht

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 beschriebenen Datentypen (**Zeiger**, **Feld** und **Verbund**) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem **Anfangsteil**, **Mittelteil** und **Schlussteil** darin erkennen.

Dieses Vorgehen ist in Abbildung 3.9 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der Datenypen **Zeiger**, **Feld** und **Verbund** gemischt sind (z.B. `(*st_var.ar)[0]`).

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

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

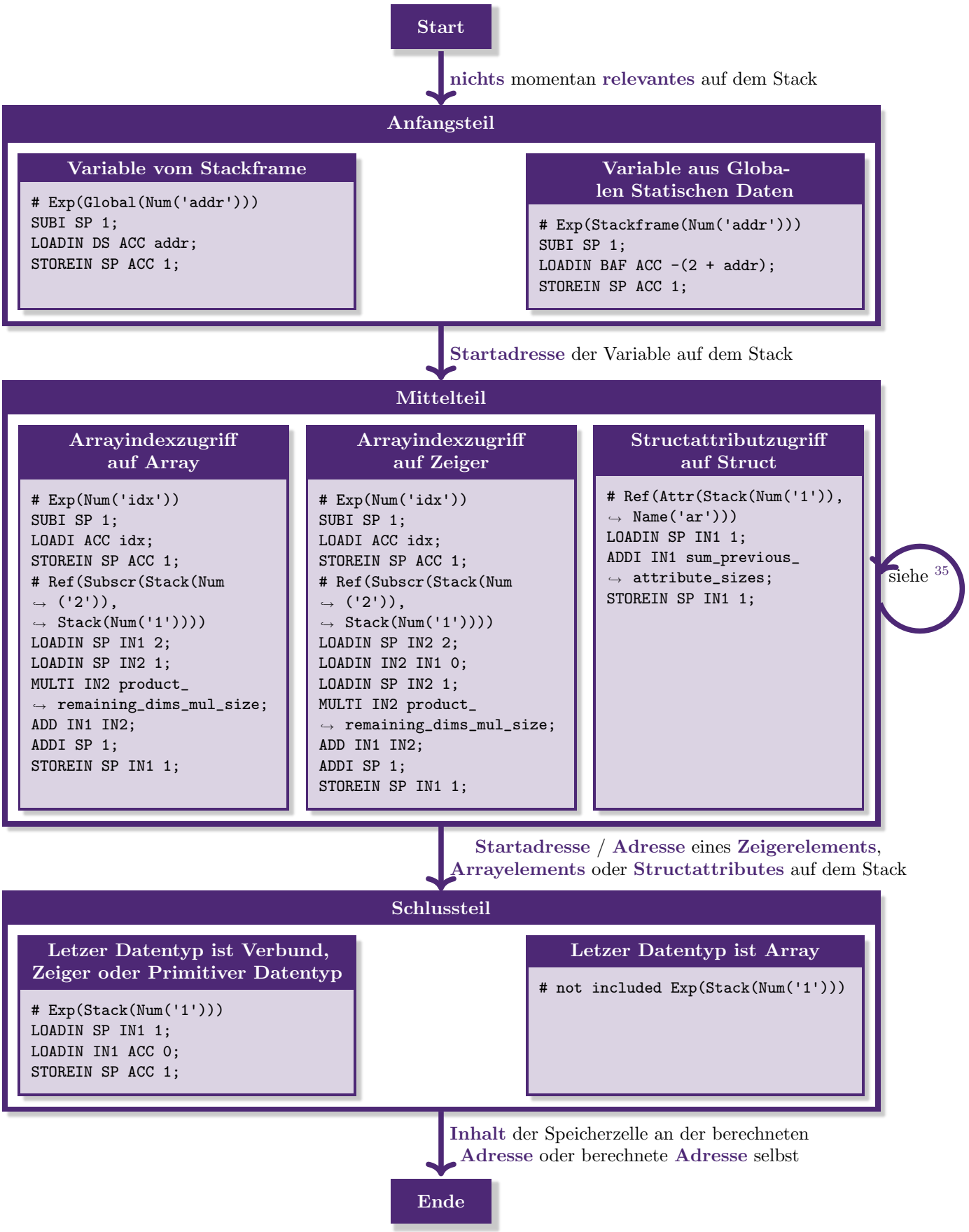


Abbildung 3.9: *Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes*

fangsteil und **Schlussteil** bleiben unverändert.

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

Anmerkung

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine **Dereferenzierung** in der Form `Deref(exp1, exp2)` nicht mehr existiert, denn wie in Unterkapitel 3.3.2 bereits erklärt wurde, wurde der Knoten `Deref(exp1, exp2)` im **PicoC-Shrink Pass** durch `Subscr(exp1, exp2)` ersetzt. Das hatte den Zweck, **doppelten Code** zu vermeiden, da die **Dereferenzierung** und der **Zugriff auf ein Arrayelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Arrayindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

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

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

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

Arrays haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes **Array** zugegriffen wird[36], 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 wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

Anmerkung

Implementieren lässt sich dieses Vorgehen, indem beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)` Ausdrucks ein `Exp(Stack(Num('1')))` an die Spitze einer **Liste der generierten Ausdrücke** gesetzt wird und der Ausdruck selbst als `exp`-Attribut des `Ref(exp)`-Knotens gesetzt wird und hinter dem `Exp(Stack(Num('1')))`-Knoten in der Liste eingefügt wird. Beim Antreffen eines

[36] Und nicht auf ein **Element** des Arrays.
[36] **Startadresse** / **Adresse** eines **Zeigerelements**, **Feldelements** oder **Verbundattributes** auf dem Stack.

`Ref(exp)` wird fast gleich vorgegangen, wie beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)`, nur, dass kein `Exp(Stack(Num('1')))` vorne an die Spitze der **Liste der generierten Ausdrücke** gesetzt wird. Und ein `Ref(exp)` bei dem `exp` direkt ein `Name(str)` ist, wird dieser einfach direkt durch `Ref(Global(num))` bzw. `Ref(Stackframe(num))` ersetzt.

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

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

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil** 3.3.5.2, **Mittelteil** 3.3.5.3 und **Schlussteil** 3.3.5.4 bei der Kompilierung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente**, **Verbundattribute** bei gemischten Ausdrücken, wie `(*st_first.ar)[0];` einzeln isoliert betrachtet und erläutert.

3.3.5.2 Anfangsteil

Der **Anfangsteil**, bei dem die **Adresse** einer Variable auf den **Stack** geschrieben wird (z.B. `&st`), wird im Folgenden mithilfe des Beispiels in Code 3.49 erklärt.

```
1  struct ar_with_len {int len; int ar[2];};
2
3  void main() {
4    struct ar_with_len st_ar[3];
5    int *(*complex_var)[3];
6    &complex_var;
7  }
8
9  void fun() {
10   struct ar_with_len st_ar[3];
11   int (*complex_var)[3];
12   &complex_var;
13 }
```

Code 3.49: *PicoC-Code für den Anfangsteil*

Im **Abstrakten Syntaxbaum** in Code 3.50 wird die **Refererenzierung** `&complex_var` mit der Komposition `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird aber einfach nur `Ref(Name('complex_var')`

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

```
File
  Name './example_derived_dts_introduction_part.ast',
  [
    StructDecl
      Name 'ar_with_len',
      [
        Alloc(Writeable(), IntType('int'), Name('len'))
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
        ↪  IntType('int')))), Name('complex_var')))
        Exp(Ref(Name('complex_var')))
      ],
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('complex_var')))
        Exp(Ref(Name('complex_var')))
      ]
  ]
```

Code 3.50: *Abstrakter Syntaxbaum für den Anfangsteil*

Im **PicoC-ANF Pass** in Code 3.51 wird die Komposition `Exp(Ref(Name('complex_var')))` durch die Komposition `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` ersetzt, je nachdem, ob die Variable `Name('complex_var')` in den **Globalen Statischen Daten** oder auf dem **Stack** liegt.

```
File
  Name './example_derived_dts_introduction_part.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Exp(Ref(Name('complex_var')))
        Ref(Global(Num('9')))
```

```
9         Return(Empty())
10      ],
11    Block
12      Name 'fun.0',
13      [
14        // Exp(Ref(Name('complex_var')))
15        Ref(Stackframe(Num('9')))
16        Return(Empty())
17      ]
18  ]
```

Code 3.51: *PicoC-ANF Pass für den Anfangsteil*

Im **RETI-Blocks Pass** in Code 3.52 werden die Komposition `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_derived_dts_introduction_part.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Exp(Ref(Name('complex_var')))
8         # Ref(Global(Num('9')))
9         SUBI SP 1;
10        LOADI IN1 9;
11        ADD IN1 DS;
12        STOREIN SP IN1 1;
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun.0',
18      [
19        # // Exp(Ref(Name('complex_var')))
20        # Ref(Stackframe(Num('9')))
21        SUBI SP 1;
22        MOVE BAF IN1;
23        SUBI IN1 11;
24        STOREIN SP IN1 1;
25        # Return(Empty())
26        LOADIN BAF PC -1;
27      ]
28  ]
```

Code 3.52: *RETI-Blocks Pass für den Anfangsteil*

3.3.5.3 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinandereihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundattribute** berechnet wird (z.B. `(*complex_var.ar)[2-2]`), wird im Folgenden mithilfe des Beispiels in Code 3.53 erklärt.

```c
struct st {int (*ar)[1];};

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

Code 3.53: *PicoC-Code für den Mittelteil*

Im **Abstrakten Syntaxbaum** in Code 3.54 wird die Aneinandererihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundattribute** `(*complex_var.ar)[2-2]` durch die Komposition `Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2'))))` dargestellt.

```
File
  Name './example_derived_dts_main_part.ast',
  [
    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(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
        ↪ Sub('-'), Num('2'))))
      ]
  ]
```

Code 3.54: *Abstrakter Syntaxbaum für den Mittelteil*

Im **PicoC-ANF Pass** in Code 3.55 wird die Komposition `Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2'))))` durch die Kompositionen `Ref(Attr(Stack(Num('1')), Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ersetzt. Bei `Subscr(exp1, exp2)` wird dieser Knoten einfach dem `exp` Attribut des `Ref(exp)`-Knoten zugewiesen und die **Indexberechnung** für `exp2` davorgezogen (in diesem Fall dargestellt durch `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`) und über `Stack(Num('1'))` auf das Ergebnis der **Indexberechnung** auf dem **Stack** zugegriffen: `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`.

```
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 3.55: *PicoC-ANF Pass für den Mittelteil*

Im **RETI-Blocks Pass** in Code 3.56 werden die Kompositionen `Ref(Attr(Stack(Num('1')), Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` durch ihre entsprechenden **RETI-Knoten** ersetzt. Bei der Generierung des **RETI-Code** muss auch das versteckte Attribut `datatype` im `Ref(exp, datatpye)`-Knoten berücksichtigt werden, was in Unterkapitel 3.3.5.1 zusammen mit der Abbildung 3.9 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')))
```

```
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          # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
            ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
27          # Ref(Global(Num('1')))
28          SUBI SP 1;
29          LOADI IN1 1;
30          ADD IN1 DS;
31          STOREIN SP IN1 1;
32          # Ref(Attr(Stack(Num('1')), Name('ar')))
33          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 3.56: *RETI-Blocks Pass für den Mittelteil*

3.3.5.4 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 3.3.5.2 und **Mittelteil** 3.3.5.3 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird[37], wird im Folgenden mithilfe des Beispiels in Code 3.57 erklärt.

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

Code 3.57: *PicoC-Code für den Schlussteil*

Das Generieren des **Abstrakten Syntaxbaumes** in Code 3.58 verläuft wie üblich.

```
1  File
2    Name './example_derived_dts_final_part.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8        ],
9      FunDef
10       VoidType 'void',
11       Name 'main',
12       [],
13       [
14         Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
           ↪  Name('complex_var1')))
15         Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
           ↪  Name('complex_var2')))
16         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
```

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

```
17        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),
          ↪  Ref(Name('var')))
18        Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),
          ↪  Ref(Name('pntr1')))
19        Exp(Subscr(Name('complex_var1'), Num('0')))
20        Exp(Subscr(Name('complex_var2'), Num('0')))
21        Exp(Deref(Name('complex_var3'), Num('0')))
22      ]
23  ]
```

Code 3.58: *Abstrakter Syntaxbaum für den Schlussteil*

Im **PicoC-ANF Pass** in Code 3.59 wird das eben angesprochene auf den **Stack** speichern des **Inhalts** der berechneten **Adresse** mit der Komposition `Exp(Stack(Num('1')))` dargestellt.

```
1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('4')), Stack(Num('1')))
10        // Assign(Name('pntr1'), Ref(Name('var')))
11        Ref(Global(Num('4')))
12        Assign(Global(Num('5')), Stack(Num('1')))
13        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14        Ref(Global(Num('5')))
15        Assign(Global(Num('6')), Stack(Num('1')))
16        // Exp(Subscr(Name('complex_var1'), Num('0')))
17        Ref(Global(Num('0')))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        // Exp(Subscr(Name('complex_var2'), Num('0')))
22        Ref(Global(Num('2')))
23        Exp(Num('0'))
24        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25        Exp(Stack(Num('1')))
26        // Exp(Subscr(Name('complex_var3'), Num('0')))
27        Ref(Global(Num('6')))
28        Exp(Num('0'))
29        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30        Exp(Stack(Num('1')))
31        Return(Empty())
32      ]
33  ]
```

Code 3.59: *PicoC-ANF Pass für den Schlussteil*

Im **RETI-Blocks Pass** in Code 3.60 wird die Komposition `Exp(Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt, wenn das versteckte Attribut `datatype` im `Exp(exp,datatpye)`-Knoten kein **Array**

`ArrayDecl(nums, datatype)` enthält, ansonsten ist bei einem **Array** die **Adresse** auf dem **Stack** bereits das gewünschte Ergebnis. Genaueres wurde in Unterkapitel 3.3.5.1 zusammen mit der Abbildung 3.9 bereits erklärt.

```
File
  Name './example_derived_dts_final_part.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('4')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 4;
        ADDI SP 1;
        # // Assign(Name('pntr1'), Ref(Name('var')))
        # Ref(Global(Num('4')))
        SUBI SP 1;
        LOADI IN1 4;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('5')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 5;
        ADDI SP 1;
        # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
        # Ref(Global(Num('5')))
        SUBI SP 1;
        LOADI IN1 5;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('6')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 6;
        ADDI SP 1;
        # // Exp(Subscr(Name('complex_var1'), Num('0')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        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 2;
        ADD IN1 IN2;
        ADDI SP 1;
        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 3.60: *RETI-Blocks Pass für den Schlussteil*

3.3.6 Umsetzung von Funktionen

3.3.6.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 3.61 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

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

Code 3.61: *PicoC-Code für 3 Funktionen*

Im **Abstrakten Syntaxbaum** in Code 3.62 wird eine **Funktion**, wie z.B. `voidfun(intparam;){ returnparam; }` mit der Komposition `FunDef(IntType(), Name('fun'), [Alloc(Writeable(), IntType(), Name('fun'))], [Return(Exp(Name('param')))])` dargestellt. Die einzelnen **Attribute** dieses Knoten sind in Tabelle 3.6 erklärt.

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

Code 3.62: *Abstrakter Syntaxbaum für 3 Funktionen*

Im **PicoC-Blocks Pass** in Code 3.63 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`[38] beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`[39].

Es werden parallel dazu neue Zuordnungen im **Assoziativen Feld** `fun_name_to_block_name` hinzugefügt. Das **Dicionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu[40]. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```
File
  Name './verbose_3_funs.picoc_blocks',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Block
          Name 'main.2',
          [
            Return(Empty())
          ]
      ],
    FunDef
      VoidType 'void',
      Name 'fun1',
      [],
      [
        Block
          Name 'fun1.1',
          []
      ],
    FunDef
      IntType 'int',
      Name 'fun2',
      [],
      [
        Block
          Name 'fun2.0',
          [
            Return(Num('1'))
          ]
      ]
  ]
```

[38] Eine Erklärung dazu ist in Unterkapitel 3.3.1.2.1 zu finden.
[39] Der **Grund** dafür kann im Unterkapitel 3.3.1.2.1 nachgelesen werden.
[40] Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 3.63: *PicoC-Blocks Pass für 3 Funktionen*

Im **PicoC-ANF Pass** in Code 3.64 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.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 3.64: *PicoC-ANF Pass für 3 Funktionen*

Nach dem **RETI Pass** in Code 3.65 gibt es nur noch **RETI-Befehle**, die Blöcke wurden entfernt und die **RETI-Befehle** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man die Funktionen nicht mehr direkt ausmachen, denn die **Kommentare** enthalten die **Labelbezeichner** `<name>.<nummer>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer, dem **Namen** der jeweiligen **Funktion** entsprechen.

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

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.2')))
# // not included Exp(GoTo(Name('main.2')))
# // Block(Name('main.2'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun1.1'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun2.0'), [])
```

```
# // Return(Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Return(Stack(Num('1')))
LOADIN SP ACC 1;
ADDI SP 1;
LOADIN BAF PC -1;
```

Code 3.65: *RETI-Blocks Pass für 3 Funktionen*

3.3.6.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 3.61 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 3.66 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 3.66: *PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist*

Im **RETI-Blocks Pass** in Code 3.67 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```
File
  Name './verbose_3_funs_main.reti_blocks',
  [
    Block
      Name 'fun1.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.1',
      [
        # // 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 3.67: *RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist*

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

Eine andere Möglichkeit und die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die `main`-Funktion nicht die erste auftauchende Funktion ist, einen `start.<nummer>`-Block als ersten Block einzufügen, der einen `GoTo(Name('main.<nummer>'))`-Knoten enthält, der im **RETI Pass** 3.69 in einen Sprung zur `main`-Funktion übersetzt wird.

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

Das **Einsetzen** des `start.<nummer>`-Blockes erfolgt im **RETI-Patch Pass** in Code 3.68, da der **RETI-Patch**-Pass der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die `main`-Funktion nicht die erste Funktion ist.

```
1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
```

```
17      Name 'fun2.1',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ],
29    Block
30      Name 'main.0',
31      [
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ]
35  ]
```

Code 3.68: *RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist*

Im **RETI Pass** in Code 3.69 wird das `GoTo(Name('main.<nummer>'))` durch den entsprechenden `Sprung JUMP <distanz_zur_main_funktion>` ersetzt und 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 3.69: *RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist*

3.3.6.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

In der Programmiersprache L_C und somit auch L_{PicoC} ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehlermeldungen** ausgegeben werden können, wenn der **Prototyp** (Definition 1.6) der Funktion nicht mit den

Datentypen der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

Die Dekleration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 3.70 **markierten Zeile** `1` oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** `3-4`.

In dem Beispiel in Code 3.70 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdekleration**, wie in der **markierten Zeile** `1` notwendig. Beim **Funktionsaufruf** zur Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen** `3-4` zu sehen ist.

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

Code 3.70: *PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss*

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 3.71 für das Beispiel in Code 3.70 dargestellt ist. Die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` werden wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette Komposition der **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(), IntType('int'), Name('var'))])` zugewiesen.

Die Varaiblen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 1.9). Die **Scopes** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**[41] Durchiterieren über den **Abstrakten Syntaxbaum** des aktuellen **Passes** ein **Funktionswechsel** eintritt und über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

Ein Grund, warum **Scopes** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel**, die aus dem **Bezeichner** einer Variable und einem angehängten **Suffix** bestehen, in der als **Assoziatives Feld** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt den **Scope** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im `Name(str)`-Token-Knoten des `name`-Attribubtes eines **Symbols** der Symboltabelle angehängt. Zur beseren Vorstellung ist dies ist in Code 3.71 **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**

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

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

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

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
           ↪  IntType('int'), Name('var'))])
7          name:                   Name('fun2')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('4'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Empty()
15         datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
16         name:                   Name('fun1')
17         value or address:       Empty()
18         position:               Pos(Num('3'), Num('5'))
19         size:                   Empty()
20       },
21     Symbol
22       {
23         type qualifier:         Empty()
24         datatype:               FunDecl(VoidType('void'), Name('main'), [])
25         name:                   Name('main')
26         value or address:       Empty()
27         position:               Pos(Num('6'), Num('5'))
28         size:                   Empty()
29       },
30     Symbol
31       {
32         type qualifier:         Writeable()
33         datatype:               IntType('int')
34         name:                   Name('var@main')
35         value or address:       Num('0')
36         position:               Pos(Num('7'), Num('6'))
37         size:                   Num('1')
38       },
39     Symbol
40       {
41         type qualifier:         Writeable()
42         datatype:               IntType('int')
43         name:                   Name('var@fun2')
```

```
44          value or address:       Num('0')
45          position:               Pos(Num('12'), Num('13'))
46          size:                   Num('1')
47        }
48    ]
```

Code 3.71: *Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss*

3.3.6.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 3.72 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und dieses Kapitel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 3.3.6.3.1 erklärt.

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

Code 3.72: *PicoC-Code für Funktionsaufruf ohne Rückgabewert*

Im **Abstrakten Syntaxbaum** in Code 3.73 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```
1  File
2    Name './example_fun_call_no_return_value.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), IntType('int'), Name('attr1'))
8          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9        ],
10     FunDecl
11       VoidType 'void',
12       Name 'stack_fun',
13       [
14         Alloc
15           Writeable,
16           ArrayDecl
17             [
```

```
18              Num '2',
19              Num '3'
20            ],
21            StructSpec
22              Name 'st',
23          Name 'param'
24      ],
25    FunDef
26      VoidType 'void',
27      Name 'main',
28      [],
29      [
30        Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
          ↪  Name('local_var')))
31        Exp(Call(Name('stack_fun'), [Name('local_var')]))
32        Return(Empty())
33      ],
34    FunDef
35      VoidType 'void',
36      Name 'stack_fun',
37      [
38        Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
          ↪  Name('param'))
39      ],
40      [
41        Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
42      ]
43  ]
```

Code 3.73: *Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert*

Im **PicoC-ANF Pass** in Code 3.74 wird die Komposition `Exp(Call(Name('stack_fun'),[Name('local_var')]))` durch die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'),GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` ersetzt, welche in den Tabellen 3.8 und 3.3 genauer erklärt sind.

Der Knoten `StackMalloc(Num('2'))` ist notwendig, weil auf dem **Stackframe** für den Wert des `BAF`-Registers der **aufrufenden Funktion** und die **Rücksprungadresse** 2 Speicherzellen Platz am **Anfang** des **Stackframes** gelassen werden muss. Das wird durch den Knoten `StackMalloc(Num('2'))` umgesetzt, indem das `SP`-Register einfach um zwei Speicherzellen **dekrementiert** wird und somit Speicher auf dem **Stack** belegt wird[42].

```
1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc(Num('2'))
8         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
10        Exp(GoTo(Name('stack_fun.0')))
11        RemoveStackframe()
```

[42] Wobei hier "**reserviert**" besser passen würde.

```
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]
```

Code 3.74: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert*

Im **RETI-Blocks Pass** in Code 3.75 werden die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch ihre entsprechenden **RETI-Knoten** ersetzt.

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

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Assoziativen Feld** `fun_name_to_block_name`[43] mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```
1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # StackMalloc(Num('2'))
8         SUBI SP 2;
9         # Ref(Global(Num('0')))
10        SUBI SP 1;
11        LOADI IN1 0;
12        ADD IN1 DS;
13        STOREIN SP IN1 1;
14        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        MOVE BAF ACC;
16        ADDI SP 3;
17        MOVE SP BAF;
18        SUBI SP 4;
19        STOREIN BAF ACC 0;
20        LOADI ACC GoTo(Name('addr@next_instr'));
21        ADD ACC CS;
22        STOREIN BAF ACC -1;
23        # Exp(GoTo(Name('stack_fun.0')))
24        Exp(GoTo(Name('stack_fun.0')))
25        # RemoveStackframe()
26        MOVE BAF IN1;
27        LOADIN IN1 BAF 0;
28        MOVE IN1 SP;
29        # Return(Empty())
```

[43] Dieses Assoziative Feld wurde in Unterkapitel 3.3.6.1 eingeführt.

```
30        LOADIN BAF PC -1;
31      ],
32    Block
33      Name 'stack_fun.0',
34      [
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ]
38  ]
```

Code 3.75: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert*

Im **RETI Pass** in Code 3.75 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist[44] ersetzt: `LOADI ACC 14`. Und auch der Knoten, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den Knoten `JUMP 5` ersetzt.

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

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

Anmerkung

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

$$adr_{danach} = \#Bef_{vor\,akt.\,Bl.} + idx + 4 \tag{3.3.1}$$

wobei:

- es sich bei bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum `CS`-Register berechnet wird.
- $\#Bef_{vor\,akt.\,Bl.} \hat{=}$ **Anzahl** Befehle vor dem momentanen Block. Es handelt sich hierbei um ein versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`, welches im **RETI-Patch**-Pass gesetzt wird. Der Grund dafür, dass das Zuweisen dieses versteckten Attributes `instrs_before` im **RETI-Patch** Pass erfolgt ist, weil erst im **RETI-Patch** Pass die **finale Anzahl** an Befehlen in einem Block feststeht, da im **RETI-Patch** Pass `GoTo()`'s entfernt werden, deren Sprung nur **eine** Adresse weiterspringen würde. Die **finale Anzahl** an Befehlen kann sich in diesem **Pass** also noch ändern und steht erst nach diesem **Pass** fest.
- $idx \hat{=}$ relativer Index des Befehls `LOADI ACC <addr@next_instr>` selbst im Block.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 3.76 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>`

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

liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

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

$$distanz = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases} \tag{3.3.2}$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und zu dem gesprungen werden soll. Es handelt sich hierbei um ein 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.

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

Code 3.76: *RETI-Pass für Funktionsaufruf ohne Rückgabewert*

3.3.6.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 3.77 erklärt.

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

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

Code 3.77: *PicoC-Code für Funktionsaufruf mit Rückgabewert*

Im **Abstrakten Syntaxbaum** in Code 3.78 wird ein **Return-Statement mit Rückgabewert** `return 21 * 2` mit der Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, ein **Return-Statement ohne Rückgabewert** `return` mit der Komposition `Return(Empty())` und ein **Funktionsaufruf** inklusive **Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` durch die Komposition `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))`.

```
1  File
2    Name './example_fun_call_with_return_value.ast',
3    [
4      FunDef
5        IntType 'int',
6        Name 'fun_with_return_value',
7        [],
8        [
9          Return(BinOp(Num('21'), Mul('*'), Num('2')))
10       ],
11     FunDef
12       VoidType 'void',
13       Name 'fun_no_return_value',
14       [],
15       [
16         Return(Empty())
17       ],
18     FunDef
19       VoidType 'void',
20       Name 'main',
21       [],
22       [
23         Assign(Alloc(Writeable(), IntType('int'), Name('var')),
           ↪  Call(Name('fun_with_return_value'), []))
24         Exp(Call(Name('fun_no_return_value'), []))
25       ]
26   ]
```

Code 3.78: *Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert*

Im **PicoC-ANF Pass** in Code 3.79 wird bei der **Komposition** `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die **Expression** `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierführ erstellten Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis wird dann von der **Komposition** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register** `ACC` geschrieben und als letztes wird die **Rücksprungadresse** in das `PC`-Register geladen, die durch den `NewStackframe()`-Token-Knoten eine Speicherzelle nach dem Wert des `BAF`-Registers der aufrufenden Funktion im **Stackframe** gespeichert ist.

Ein wichtiges Detail bei der **Funktion** `fun_with_return_value` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird, da die **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Um den **Rückgabewert**, der durch die Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das `ACC`-Register geschrieben wurde für die aufrufende Funktion, deren Stackframe nun wieder das aktuelle ist auf den **Stack** zu schreiben, muss ein neue **Komposition** `Exp(ACC)` definiert werden. In Tabelle 3.8 ist die **Komposition** `Exp(ACC)` genauer erklärt.

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

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

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

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

```
1  File
2    Name './example_fun_call_with_return_value.picoc_mon',
3    [
4      Block
5        Name 'fun_with_return_value.2',
6        [
7          // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8          Exp(Num('21'))
9          Exp(Num('2'))
10         Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
11         Return(Stack(Num('1')))
12       ],
13     Block
```

```
14      Name 'fun_no_return_value.1',
15      [
16        Return(Empty())
17      ],
18    Block
19      Name 'main.0',
20      [
21        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
22        StackMalloc(Num('2'))
23        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
24        Exp(GoTo(Name('fun_with_return_value.2')))
25        RemoveStackframe()
26        Exp(ACC)
27        Assign(Global(Num('0')), Stack(Num('1')))
28        StackMalloc(Num('2'))
29        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
30        Exp(GoTo(Name('fun_no_return_value.1')))
31        RemoveStackframe()
32        Return(Empty())
33      ]
34  ]
```

Code 3.79: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert*

Im **RETI-Blocks Pass** in Code 3.80 werden die Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Mul('*'),Stack(Num('1'))))`, `Return(Stack(Num('1')))` und `Assign(Global(Num('0')),Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         # Exp(Num('21'))
9         SUBI SP 1;
10        LOADI ACC 21;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
19        MULT ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Return(Stack(Num('1')))
23        LOADIN SP ACC 1;
24        ADDI SP 1;
25        LOADIN BAF PC -1;
26      ],
```

```
    Block
      Name 'fun_no_return_value.1',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 2;
        MOVE SP BAF;
        SUBI SP 2;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_with_return_value.2')))
        Exp(GoTo(Name('fun_with_return_value.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Exp(ACC)
        SUBI SP 1;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 2;
        MOVE SP BAF;
        SUBI SP 2;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_no_return_value.1')))
        Exp(GoTo(Name('fun_no_return_value.1')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 3.80: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert*

3.3.6.3.2 Umsetzung von Call by Sharing für Arrays

Die **Call by Reference** (Definition 1.5) Übergabe eines Arrays an eine andere Funktion, wird im Folgenden mithilfe des Beispiels in Code 3.81 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 3.81: *PicoC-Code für Call by Sharing für Arrays*

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

Diese **Umwandlung** des **Datentyps** kann in der **Symboltabelle** in Code 3.82 beobachtet werden. Die **lokalen Variablen** `local_var@main` und `local_var@fun_array_from_global_data` sind beide vom Datentyp `ArrayDecl([Num('2'), Num('3')], IntType('int'))` und bei der Übergabe ändert sich der Datentyp beider Variablen zu `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))`. Die **Größe** dieser Variablen ändert sich damit zu `Num('1')`, da ein **Zeiger** nur eine **Speicherzelle** braucht.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
           ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
           ↪  Name('param'))])
7          name:                   Name('fun_array_from_stackframe')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('5'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Writeable()
15         datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16         name:                   Name('param@fun_array_from_stackframe')
17         value or address:       Num('0')
18         position:               Pos(Num('1'), Num('37'))
19         size:                   Num('1')
20       },
```

```
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
          ↪  [Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
25        name:                   Name('fun_array_from_global_data')
26        value or address:       Empty()
27        position:               Pos(Num('4'), Num('5'))
28        size:                   Empty()
29      },
30    Symbol
31      {
32        type qualifier:         Writeable()
33        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
34        name:                   Name('param@fun_array_from_global_data')
35        value or address:       Num('0')
36        position:               Pos(Num('4'), Num('36'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
43        name:                   Name('local_var@fun_array_from_global_data')
44        value or address:       Num('6')
45        position:               Pos(Num('5'), Num('6'))
46        size:                   Num('6')
47      },
48    Symbol
49      {
50        type qualifier:         Empty()
51        datatype:               FunDecl(VoidType('void'), Name('main'), [])
52        name:                   Name('main')
53        value or address:       Empty()
54        position:               Pos(Num('9'), Num('5'))
55        size:                   Empty()
56      },
57    Symbol
58      {
59        type qualifier:         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 3.82: *Symboltabelle für Call by Sharing für Arrays*

Im **PicoC-ANF Pass** in Code 3.83 ist zu sehen, dass zur Übergabe der beiden Arrays die **Adresse** der Arrays auf den **Stack** geschrieben wird. Die **Adresse** der beiden Arrays auf den **Stack** zu schreiben wird durch die Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` repräsentiert.

Die Komposition `Ref(Global(Num('0')))` ist für Variablen in den **Globalen Statischen Daten** und die Komposition `Ref(Stackframe(Num('6')))` ist für Variablen aus dem **Stackframe**. Dabei stellen die Zahlen

in den Knoten `Global(num)` bzw. `Stackframe(num)` die **relative Adressen** relativ zum `DS`-Register bzw. `SP`-Register dar, die aus der **Symboltabelle** entnommen sind.

```
File
  Name './example_fun_call_by_sharing_array.picoc_mon',
  [
    Block
      Name 'fun_array_from_stackframe.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun_array_from_global_data.1',
      [
        StackMalloc(Num('2'))
        Ref(Stackframe(Num('6')))
        NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_array_from_stackframe.2')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_array_from_global_data.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 3.83: *PicoC-ANF Pass für Call by Sharing für Arrays*

Im **RETI-Blocks Pass** in Code 3.84 werden Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      Name 'fun_array_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_array_from_global_data.1',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Stackframe(Num('6')))
```

```
16        SUBI SP 1;
17        MOVE BAF IN1;
18        SUBI IN1 8;
19        STOREIN SP IN1 1;
20        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
21        MOVE BAF ACC;
22        ADDI SP 3;
23        MOVE SP BAF;
24        SUBI SP 3;
25        STOREIN BAF ACC 0;
26        LOADI ACC GoTo(Name('addr@next_instr'));
27        ADD ACC CS;
28        STOREIN BAF ACC -1;
29        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
30        Exp(GoTo(Name('fun_array_from_stackframe.2')))
31        # RemoveStackframe()
32        MOVE BAF IN1;
33        LOADIN IN1 BAF 0;
34        MOVE IN1 SP;
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ],
38    Block
39      Name 'main.0',
40      [
41        # StackMalloc(Num('2'))
42        SUBI SP 2;
43        # Ref(Global(Num('0')))
44        SUBI SP 1;
45        LOADI IN1 0;
46        ADD IN1 DS;
47        STOREIN SP IN1 1;
48        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
49        MOVE BAF ACC;
50        ADDI SP 3;
51        MOVE SP BAF;
52        SUBI SP 9;
53        STOREIN BAF ACC 0;
54        LOADI ACC GoTo(Name('addr@next_instr'));
55        ADD ACC CS;
56        STOREIN BAF ACC -1;
57        # Exp(GoTo(Name('fun_array_from_global_data.1')))
58        Exp(GoTo(Name('fun_array_from_global_data.1')))
59        # RemoveStackframe()
60        MOVE BAF IN1;
61        LOADIN IN1 BAF 0;
62        MOVE IN1 SP;
63        # Return(Empty())
64        LOADIN BAF PC -1;
65      ]
66  ]
```

Code 3.84: *RETI-Block Pass für Call by Sharing für Arrays*

3.3.6.3.3 Umsetzung von Call by Value für Structs

Die **Call by Value** (Definition 1.4) Übergabe eines **Structs** wird im Folgenden mithilfe des Beispiels in Code 3.85 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 3.85: *PicoC-Code für Call by Value für Structs*

Im **PicoC-ANF Pass** in Code 3.86 wird zur **Übergabe eines Struct**, das komplette Struct auf den **Stack** kopiert. Das wird mittels der Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` dargestellt.

Bei der **Übergabe** an eine **Funktion** wird der Zugriff auf ein gesamtes **Struct** anders gehandhabt als sonst. Normalerweise wird beim Zugriff auf ein Struct die **Adresse** des **ersten Attributs** dieses Strcuts auf den **Stack** geschrieben. Bei der **Übergabe an eine Funktion** wird dagegen das gesamte **Strcut** auf den **Stack** kopiert.

Das wird durch eine Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange der **Funktionsaufruf** im **Picoc-ANF Pass** verarbeitet wird und wieder auf `false` gesetzt, wenn die Verarbeitung des **Funktionaufrufs** abgeschlossen ist. Solange die Variable `argmode_on` auf `true` gesetzt ist, wird immer die Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` für die Ersetzung verwendet. Ist die Varaible `argmode_on` auf `false` wird die Komposition `Ref(Globalnum())` bzw. `Ref(Stackframe(num))` für die Ersetzung verwendet.

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

```
1  File
2    Name './example_fun_call_by_value_struct.picoc_mon',
3    [
4      Block
5        Name 'fun_struct_from_stackframe.2',
6        [
7          Return(Empty())
8        ],
9      Block
10       Name 'fun_struct_from_global_data.1',
11       [
```

```
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Stackframe(Num('2')))
14        NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
16        RemoveStackframe()
17        Return(Empty())
18      ],
19    Block
20      Name 'main.0',
21      [
22        StackMalloc(Num('2'))
23        Assign(Stack(Num('3')), Global(Num('0')))
24        NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
25        Exp(GoTo(Name('fun_struct_from_global_data.1')))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]
```

Code 3.86: *PicoC-ANF Pass für Call by Value für Structs*

Im **RETI-Blocks Pass** in Code 3.87 werden die Kompositionen `Assign(Stack(Num('3')), Stackframe(Num('2')))` und `Assign(Stack(Num('3')), Global(Num('0')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun_struct_from_global_data.1',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Assign(Stack(Num('3')), Stackframe(Num('2')))
16        SUBI SP 3;
17        LOADIN BAF ACC -4;
18        STOREIN SP ACC 1;
19        LOADIN BAF ACC -3;
20        STOREIN SP ACC 2;
21        LOADIN BAF ACC -2;
22        STOREIN SP ACC 3;
23        # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
24        MOVE BAF ACC;
25        ADDI SP 5;
26        MOVE SP BAF;
27        SUBI SP 5;
28        STOREIN BAF ACC 0;
29        LOADI ACC GoTo(Name('addr@next_instr'));
```

```
30        ADD ACC CS;
31        STOREIN BAF ACC -1;
32        # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
33        Exp(GoTo(Name('fun_struct_from_stackframe.2')))
34        # RemoveStackframe()
35        MOVE BAF IN1;
36        LOADIN IN1 BAF 0;
37        MOVE IN1 SP;
38        # Return(Empty())
39        LOADIN BAF PC -1;
40      ],
41    Block
42      Name 'main.0',
43      [
44        # StackMalloc(Num('2'))
45        SUBI SP 2;
46        # Assign(Stack(Num('3')), Global(Num('0')))
47        SUBI SP 3;
48        LOADIN DS ACC 0;
49        STOREIN SP ACC 1;
50        LOADIN DS ACC 1;
51        STOREIN SP ACC 2;
52        LOADIN DS ACC 2;
53        STOREIN SP ACC 3;
54        # NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
55        MOVE BAF ACC;
56        ADDI SP 5;
57        MOVE SP BAF;
58        SUBI SP 5;
59        STOREIN BAF ACC 0;
60        LOADI ACC GoTo(Name('addr@next_instr'));
61        ADD ACC CS;
62        STOREIN BAF ACC -1;
63        # Exp(GoTo(Name('fun_struct_from_global_data.1')))
64        Exp(GoTo(Name('fun_struct_from_global_data.1')))
65        # RemoveStackframe()
66        MOVE BAF IN1;
67        LOADIN IN1 BAF 0;
68        MOVE IN1 SP;
69        # Return(Empty())
70        LOADIN BAF PC -1;
71      ]
72  ]
```

Code 3.87: *RETI-Block Pass für Call by Value für Structs*

3.4 Fehlermeldungen

Die **Fehlerarten**, die der **PicoC-Compiler** ausgeben kann sind in den Tabellen 3.9, 3.10 und 3.11 und eingeteilt nach den Kategorien **Syntax**, **Semantik** und **Laufzeit** aus Unterkapitel 2.6.

Fehlerarten, wie z.B. `UninitialisedVariable` beim Verwenden einer **uninitialisierten Variable** oder `IndexOutOfBound` bei **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 3.88 läuft z.B. **ohne Fehlermeldungen** durch.

```c
#include <stdio.h>

void main() {
  int var;
  printf("\n%d", var);
}
```

Code 3.88: *Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet*

Fehlerart	Beschreibung
`UnexpectedCharacter`	Der **Lexer** ist auf eine **unerwartete Zeichenfolge** gestossen, die von keinem **Pattern** erkannt wird.
`UnexpectedToken`	Der **Parser** hat ein **unerwartetes Token** erhalten, dass in dem **Kontext** in dem er sich befand **nicht vorkommen** konnte.
`UnexpectedEOF`	Der **Parser** hat in dem **Kontext** in dem er sich befand bestimmte **Token erwartet**, aber die **Eingabe endete** abrupt.

Tabelle 3.9: *Syntaktische Fehlerarten*

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 **Structtyp** (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**.
`ReDeclarationDefinition`	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 **letztes Statement** kein `return`-Statement 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 3.10: *Semantische Fehlerarten*

Fehlerart	Beschreibung
`DivisionByZero`	Wenn bei einer **Division** durch 0 geteilt wird (z.B. `var / 0`).

Tabelle 3.11: *Laufzeit Fehlerarten*

In Code 3.90 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 3.90 ist in Code 3.89 zu sehen. Nacheinander stehen in Code 3.90 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 3.89: *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 **Token**, welches an der Stelle des Fehlers **vorgefunden** wurde und das Symbol ˆ soll einen **Pfeil** symbolisieren, der auf eine **Position zeigt** an der ein **anderes Token**, ein **anderer Datentyp** usw. **erwartet** worden wäre und **in der Zeile darunter** eine **Beschriftung** an sich hängen hat, die konkrett angibt, was dort eingentlich **erwartet** worden wäre.

```
1 ./tests/error_wrong_written_keyword.picoc:7:4: UnexpectedToken: Expected e.g. 'while', found
  ↪  'wile'.
2     do {
3
4     } wile (True);
5       ^~~~
6       'while'
7   }
```

Code 3.90: *Beispiel für typische Fehlermeldung mit 'found' und 'expected'*

Bei **Fehlermeldungen**, wie in Code 3.91, 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.

```
1  ./tests/error_redefinition.picoc:6:6: Redefinition: Redefinition of 'var'.
2   void main() {
3     int var = 42;
4     int var = 41;
5         ~~~
6   }
7   ./tests/error_redefinition.picoc:5:6: Note: Already defined here:
8
9   void main() {
10    int var = 42;
11        ~~~
12    int var = 41;
13  }
```

Code 3.91: *Beispiel Fehlermeldung langgestrechte fehlermeldung*

Bei manchen Fehlermeldungen, wie in Code 3.92 ist es **garnicht möglich** mit ˜ ein Token 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 Token** gibt, welches man markieren könnte. Des Weiteren ist in Code 3.92 interessant, dass in markierten Zeile in Code 3.92 **mehrere Tokens** angegeben werden, die

nach der Konkretten 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 3.92: *Beispiel für Fehlermeldung mit mehreren erwarteten Tokens*

Bei wiederum anderen Fehlermeldungen, wie in Code 3.93 ist es **nicht möglich** ein **erwartetes Token** anzugeben, da es sich um einen **Semantische Fehler** handelt und der Fehler **nicht** durch **brechen** mit der **Syntax** zustande gekommen ist, sondern auf das konkrette Beispiel in Code 3.93 bezogen daran liegt, dass die Variable `unknown_identifier` nicht definiert ist, weshalb sich hier keine **erwarteten Token** angeben lassen, da auf der **semantischen** Seite verlangt ist, dass diese Variable `unknown_identifier` eine **Bedeutung** hat, welche sie aufgrund dessen, dass sie **nicht definiert** ist nicht hat.

```
./tests/error_unknown_identifier_ref.picoc:5:18: UnknownIdentifier: Identifier
↪  'unknown_identifier' wasn't declared yet.

 void main() {
   int var = 42 + unknown_identifier;
                  ~~~~~~~~~~~~~~~~~~
 }
```

Code 3.93: *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 gegeben werden, was im Kapitel Implementierung implementiert wurde. Im Unterkapitel 4.1 wird darauf eingegangen ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel Motivation alle implementiert werden konnten und daraufhin mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten Zugegriffen werden kann, aber auch auf Funktionalitäten **anderer mitimplementierter Tools**. 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

In Kapitel Implementierung konnten **alle** Funktionalitäten, die in Kapitel Motivation erläutert wurden implementiert werden. 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 **unübersichtlichen 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** implemenetiert. Diese werden in den folgenden Kapiteln 4.1.1, 4.1.2 und 4.1.3 mithilfe kurzer **Anleitungen** erklärt.

Die kurzen **Anleitungen** in dieser **Schrifftlichen 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` mitanzugeben, von denen die **wichtigsten** in Tabelle 4.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumenation** unter Link nachgelesen werden.

[2]Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der festen Namen `a.out` erstellt.

Kommandozeilenoptio	Beschreibung	Standardwe
`-i`, `--intermediate_stages`	Gibt **Zwischenschritte** der Kompilierung in Form der verschiedenen **Tokens**, **Ableitungsbäume**, **Abstrakten Syntaxbäume** der verschiedenen **Passes** in Dateien mit **entsprechenden Dateiendungen** aber gleichem **Basinamen** 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, welche ein **Statement** oder **Befehl** aus einem **vorherigen Pass** beinhalten, der durch die darunterliegenden Statements 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 in den Abstrakten Syntaxbäumen 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** ebenfalls 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`
`-l`	Es lässt sich einstellen, **wieviele Zeilen** rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	
`-c`	Aktiviert **farbige Ausgabe**.	
`-t`, `--thesis`	**Filtert** für die **Codebeispiele** in dieser Schrifftlichen 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 4.1: *Kommandozeilenoptionen*

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

4.1.2 Shell-Mode

Will man z.B. eine **Folge von Statements** 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. Die angegebene **Folge von Statements** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem `> compile <cli-options> <filename>`-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.

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> <filename>` (oder seine **Abkürzung** `mu`) genutzt werden, um diese Kommandozeilenoptionen mit dem `compile`-Befehl **nicht** jedes mal **selbst** Angeben zu müssen. In der Tabelle 4.1 sind in grau die Werte 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`- und die `--run`-Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie `Tokens`, `Ableitungsbaum` usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls angezeigt. 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 ← und → Pfeiltasten bewegt werden. In der **Befehlshistorie** kann sich mit den ↑ und ↓ Pfeiltasten **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 verwiesen, welche auch über den Befehl `help` angezeigt werden kann.

4.1.3 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-Code** zu finden, indem im Terminal eine **virtuelle RETI-CPU** angezeigt wird, welches den **kompletten**

Zustand einer virtuell ausgeführten RETI mit allen **Registern**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen** anzeigt.

Allerdings bringt die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen** 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**, die z.B. unter `<more-options>` noch für die **Makefile** gesetzt werden können sind in Tabelle 4.2 aufgelistet.

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

Tabelle 4.2: *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**[3] um einen **Dateiinhalt** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 4.1 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 `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen anzuzeigen, indem `EXTENSION` auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandet, indem die verschiedenen **Zustände der RETI-CPU nicht** zur **Laufzeit** des **Show-Mode** berechnet werden, sondern schon 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 Zuständen** der **RETI-CPU vor** und **nach** der Ausführung eines Befehls **angezeigt** werden.

[3] *Home - Neovim*.

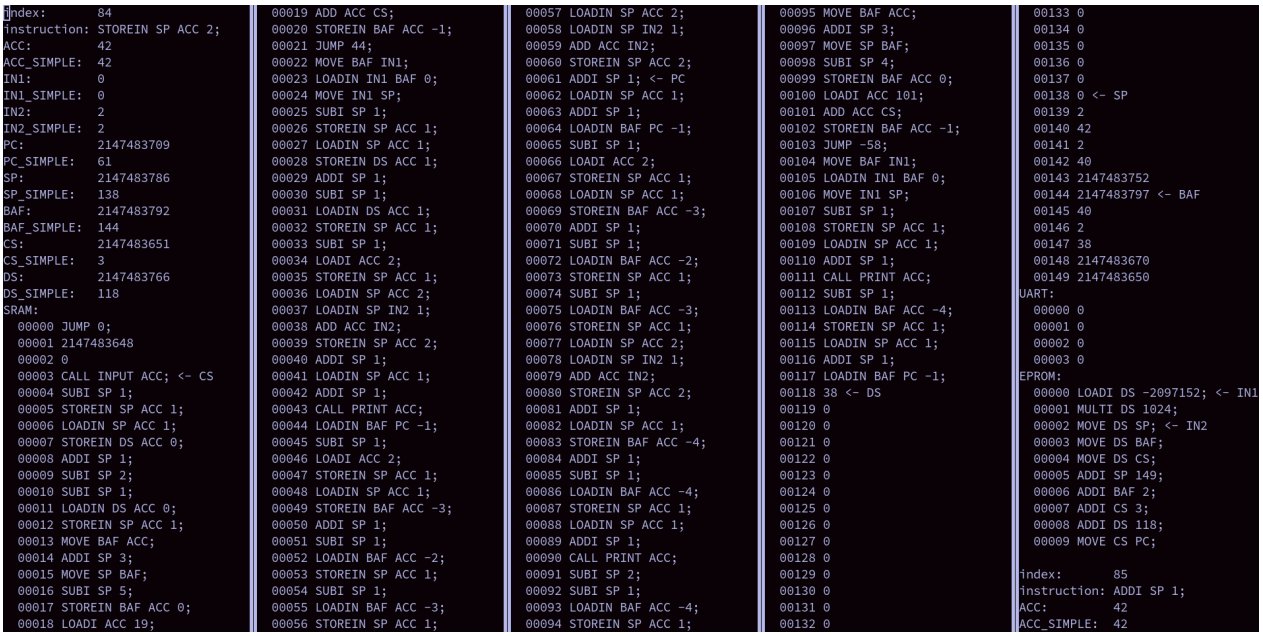


Abbildung 4.1: *Show-Mode in der Verwendung*

Zur **besseren Orientierung** wird für alle Register ebenfalls ein mit der Registerbezeichnung beschriffteter **Zeiger** `<- REG` an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert im 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** dieser wieder allerdings auf die **Dokumentation** unter Link 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 2.2.1 in Unterkapitel 2.1 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 Syntax**.

Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis `/tests` bzw. unter Link[4] zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 4.3.

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

Testkategorie	Beschreibung
`basic`	**Einfache** Tests, welche die **grundlegenden Funktionalitäten** des Compilers testen.
`advanced`	Tests, die **Spezialfälle** und **Kombinationen** verschiedener Funktionalitäten des Compilers testen.
`hard`	Tests, die **längere, komplexe Programme** testen, für welche die Funktionaliäten des Compilers in **perfekter Harmonie** miteinander funktionieren müssen.
`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 Verfikation** ausgeführt.
`exclude`	Tests, für welche aufgrund vielfältiger Gründe **keine Verifikation** ausgeführt werden soll.
`thesis`	Tests, die eigentlich vorher **Codebeispiele** für diese Schrifftliche Ausarbeitung der Bachelorarbeit waren.
`tobias`	Tests, die der Betreuer dieser Bachelorarbeit, **Tobias** geschrieben hat.

Tabelle 4.3: *Testkategorien*

Dass die Programme in beiden Sprachen die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn beide die **gleiche Ausgabe** haben und 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 drückt aus, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat, in welcher der **Schreiber des Tests** die Rolle des entsprechenden **Interpreters**[5] aus Diagramm 2.2.1 übernimmt und die **erwartete Ausgabe** seiner eigenen Interpretation des **PicoC-Codes** anstelle von `<space_seperated_output>` hineinschreibt.

Ein Beispiel für einen **Test** ist in Code 4.3 zu sehen. Sobald die Tests mithilfe des Bashcripts `/run_tests.sh` ausgeführt werden oder dieses mithilfe der `/Makefile` mit dem Befehl `> make test` ausgeführt wird, wird als erstes für **jeden** Test das Bashscript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert[6] und die entsprechenden Werte 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 eingelesen werden, die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben** der `print(<exp>)`-Funktionaufrufe, die später eingeführte Datei `<program>.out` enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Tests** und die Datei `<program>.datasegment_size` enthält die **Größe des Datensegments** für die Ausführung des entsprechenden Tests.

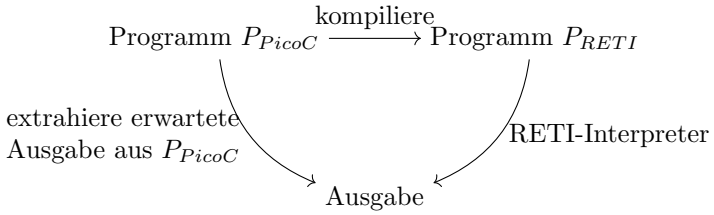
[5] Der die **Semantik** des Tests umsetzt.
[6] Falls vorhanden.

```
// 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 erfüllt, 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 mit einer bestimmten Wahrscheinlichkeit 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.

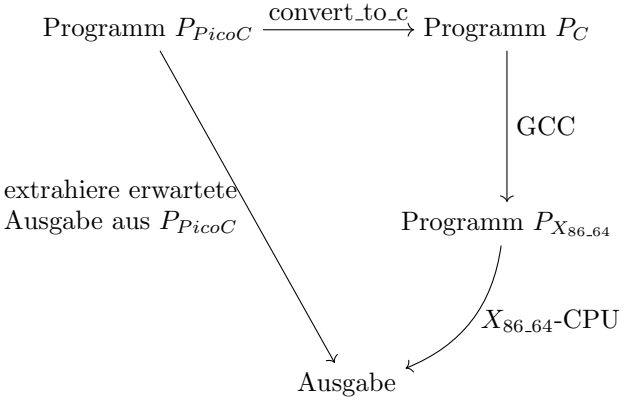


(4.2.1)

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 **Compiler implementiert**. Wenn der **Schreiber** der Tests ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl im **Test** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht[7].

Aus diesem Grund muss hier eine **weitere Maßnahme**, welche in Diagramm 4.2.2 dargestellt ist eingeführt werden, die gewährleistet, dass die **Ausgabe** in Diagramm 4.2.1 sich auf jeden Fall aus der **Semantik** der Sprache L_{PicoC}[8] ergibt. Das wird erreicht, indem wie in Diagramm 4.2.2 dargestellt ist, überprüft wird, ob die **Ausgabe** des Pfades von P_{PicoC} zur Ausgabe mit der **Ausgabe** des Pfades von P_C über $P_{X_{86\_64}}$ **identisch** ist.

[7] Welche ja identisch zu der von L_C sein sollte.
[8] Die eine **Untermenge** von L_C ist.



(4.2.2)

Das Programm P_C ergibt sich dabei aus dem Testprogramm P_{PicoC} durch **Ausführen** des Pythonscripts `/convert_to_c.py`, welches **später näher erläutert** wird. Mithilfe der `/Makefile` und dem Befehl `> make convert` lässt sich dieses Pythonscript auf **alle** Tests anwenden.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante von P_C zu $P_{X_{86\_64}}$. Beim **GCC** handelt es sich um einen Compiler der Sprache L_C, der somit auch 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[9] [10] 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 kann nun **sichergestellt** werden, dass die Tests **nicht** nur die Interpretation, die der Schreiber der Tests und Implementierer des **PicoC-Compilers** von der Semantik der Sprache L_{PicoC} hat **bestätigen**, sondern die tätsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} testen.

Dazu durchläuft jeder Test, wie in Diagramm 4.2.2 dargestellt ist eine **Verifikation**, in der **verifiziert** wird, 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. Erst dann ist ein Test **verifiziert**, d.h. man kann, wenn der Test **vernünftig definiert** ist mit **hoher Wahrscheinlichkeit** sagen[11], dass wenn dieser Test für den **PicoC-Compiler** durchläuft, der **Teil der Semantik** der Sprache L_{PicoC}, den dieser Test testet vom PicoC-Compiler **korrekt umgesetzt** ist.

Für diese **Verifikation** ist das Bashscript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt wird. Beim Befehl `> make test` wird dieses Bashscript **vor** dem eigentlichen Testen[12] durchgeführt. In Code 4.4 ist ein Testdurchlauf mit `> make test` zu sehen. Wobei `Verified: 50/50` anzeigt, wieviele der Tests **verifizierbar** sind[13], also beim **GCC** ohne Fehlermeldung durchlaufen, `Not verified:` die **nicht verifizierbaren** Tests angibt, `Running through: 88 / 88` anzeigt wieviele Tests mit dem **PicoC-Compiler** durchlaufen, `Not running through:` die **nicht** durchlaufenden Tests angibt, `Passed: 88 / 88` zeigt bei wievielen Tests die Ausgabe mit der erwarteten Ausgabe **identisch** ist, `Not passed:` die Tests anzeigt, bei denen das **nicht** der Fall ist.

[9] *History - GCC Wiki*.
[10] In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.
[11] Es besteht allerdings immer eine **Chance**, dass die Ausgabe für den Test nur **zufällig** übereinstimmt. Diese Chance kann allerdings durch **vernünftige Definition** des Tests sehr **gering** gehalten werden.
[12] Prüfen, ob der interpretierte RETI-Code des PicoC-Compilers die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.
[13] Also **alle** Tests aus den **Kategorien** `basic`, `advanced`, `hard` und `example`.

```
> 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 4.2 kombinieren.

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

Die Funktion `print(<exp>)` wird vom Pythonscript `convert_to_c.py` zu `printf("%d", <exp>)` übersetzt. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibltiothek** `<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** den Wert hin, den die `input()`-Funktionen normalerweise einlesen sollten.

4.3 Erweiterungsideen

Mit dem **Funktionsumfang** des **PicoC-Compilers**, der in Unterkapitel 4.2 erläutert wurde muss allerdings das Ende der Fahnenstange noch **nicht** erreicht sein. Weitere Ideen, die im **PicoC-Compiler**[14] implementiert werden könnten, wären:

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

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

(Definition 2.49) zuzuweisen wird mithilfe einer **Liveness Analyse** (Defintion 5.9) ein **Interferenzgraph** (Definition 5.12) aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition 5.11) angewandt, der den **Locations** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Des Weiteren muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussnalayse** (Definition 5.15) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition 5.13) auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.[15]

- **Tail Call:** Wenn ein Funktionsaufruf das **letzte** Statement in einem Funktionsblock ist, wird der Stackframe dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion zurückgekehrt werden muss[16]. 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 nur eine **konstante Menge** an **Speicherplatz** auf dem Stack verbraucht. In Code 4.5 sind **zwei Tail Calls** markiert.

- **Partielle Evaluation:** Bei Ausdrücken wie `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 Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache L_{Picoc_Eval} umformt. In der Abstrakten Syntax der Sprache L_{PicoC_Eval} sind **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** produziert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.

- **Lazy Evaluation:** Bei Ausdrücken wie `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei `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 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**[17] 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** werden zusammen mit **Attributen** in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden `.h`-**Headerdateien** verwendet, für die Definition sind `.c`-**Quellcodedateien** da. Hierbei ist der **Basisname** einer `.h`-**Headerdatei identisch** zur entsprechenden `.c`-**Quellcodedatei** mit den entsprechenden Definitionen. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei entspricht. Über einen **Linker** (Definition 5.4) können die **kompilierten** `.o`-**Objektdateien** (Definition 5.3) zusammengefügt werden, wobei der **Linker** darauf achtet **keinen doppelten Code** zuzulassen.

- `malloc` **und** `free`**:** Es wird eine **Bibltiothek** mit den Funktionen `malloc` und `free`, wie in der Bibltiothek

[15] 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 und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

[16] 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 der Ausführung** fortgesetzt werden kann.

[17] Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

`stdlib`[18] implementiert, deren `.h`-Headerdatei mittels `#include "malloc_and_free.h"` eingebunden werden muss. 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 diesen **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz 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, der sonst unmöglich in der Zukunft mehr genutzt werden würde freigegeben wird. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition 5.16) implementieren.

- `stdio.h`**:** Die Funktionen `print` und `input` werden nicht über den **Trick** einen eigenen **RETI-Befehl** `CALL (PRINT | INPUT) ACC` für den **RETI-Interpreter** zu definieren, der einfach **direkt** das **Ausgeben** und **Eingaben entgegennehmen** übernimmt gelöst, sondern über eine eigene `stdio`-**Bibliothek** mit `print`- und `input`-Funktionen, welche die **UART** verwenden, um z.B. an einem simpel gehaltenen simulierten **Monitor** Daten zu übertragen, die dieser anzeigt.

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

- **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 P. D. C. Scholl, „Betriebssysteme“ festgelegt wurde.

- **PicoPython:** Da das **Lark Parsing Toolkit** verwendet wurde, welches das **Parsen** über eine selbst angegebene **Konkrette Grammatik** übernimmt, könnte mit **relativ geringem Aufwand** ein Konkrette Grammatik defininiert werden, die eine zur Programmiersprache L_{Python} **ähnliche Konkrette Syntax** beschreibt. Die **Konkrette Syntax** einer Programmiersprache lässt sich durch Austauschen der Konkretten Grammatik **sehr einfach** ändern, nur die **Semanatik** zu ändern kann **deutlich aufwändiger** sein. Viele der **PicoC-Knoten** könnten für die Programmiersprache $L_{PicocPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.

- **Call by Reference:** Über das wiederverwenden des `&`-Symbols für **Parameter** bei **Funktiondeklaration** und **Funktionsdefinition**, wie es in der Vorlesung P. D. P. Scholl, „Einführung in Embedded Systems“ erklärt wurde.

- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt durch welche spezielle **Informationen** in den RETI-Code geschrieben werden, die einem **Debugger** unter anderem mitteilen, wo die **RETI-Befehle** für ein Statement **beginnen** und wo sie **aufhören** usw., damit der **Debugger** weiß, bis wohin er die **RETI-Befehle** ausführen soll, damit er ein Statement abgearbeitet hat.

- **Bootstrapping:** Mittels **Bootstrapping** lässt sich der **PicoC-Compiler** unabängig von der Sprache L_{Python} und der **Maschine**, die das **cross-compilen** (Definition 2.5) übernimmt machen. Im Unterkapitel 4.3 wird genauer hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zum einem **Compiler** für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

[18] Auch engl. **General Purpose Standard Library** genannt.

```
// in:42
// expected:0

int ret0() {
  return 0;
}

int ret1() {
  return 1;
}

int tail_call_fun(int bool_val) {
  if (bool_val) {
    return ret0();
  }
  return ret1();
}

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

Code 4.5: *Beispiel für Tail Call*

Anmerkung

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

Appendix

RETI Architektur Details

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*

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*

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*

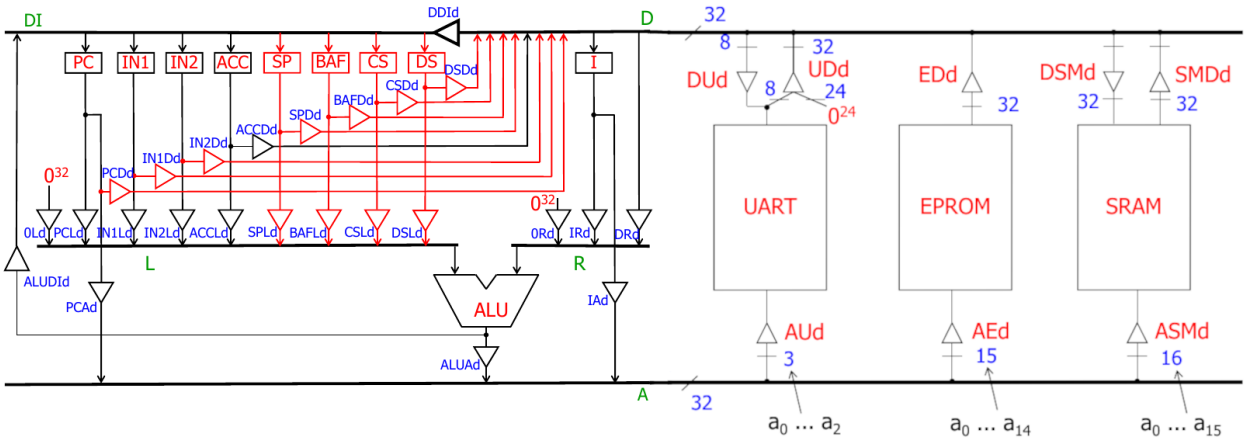


Abbildung 5.1: *Datenpfade der RETI-Architektur*

Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die zur **Erklärung** der Vorgehensweise zur Implementierung eines **üblichen Compilers** referenziert werden, aber **nichts** mit dem Vorgehen zur Implementierung des **PicoC-Compilers** zu tuen haben.

Definition 5.1: Assemblersprache (bzw. engl. Assembly Language)

*Eine sehr **hardwarenahe** Programmiersprache, derren **Befehle** eine starke Entsprechung zu bestimmten Maschienenbefehlen bzw. Folgen von Maschienenbefehlen[a] haben. Viele **Befehle** haben eine ähnliche übliche Struktur* `Operation <Operanden>`*, mit einer **Operation**, die einem **Opcode** eines Maschienenbefehls bezeichnet und keinen oder mehreren **Operanden**, wie die späteren Maschienenbefehle, 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 Maschienenbefehlen 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 mitcodiert 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 und Hauptspeicher.
[d] P. D. P. Scholl, „Einführung in Embedded Systems".

Ein **Assembler** (Definition 5.2) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschienencode** bzw. **Objectcode** (Definition 5.3) 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 den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschienencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 5.4) zu Maschiendencode zusammengesetzt wird ausführbar ist.

Definition 5.2: Assembler

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschienencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschienensprache** geschrieben ist.*[a]

[a] P. D. P. Scholl, „Einführung in Embedded Systems".

Definition 5.3: Objectcode

*Bei Komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen wird häufig **Objectcode** erzeugt, der neben der Folge von Maschienenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschienencode zusammengesetzt hat.*[a]

[a] P. D. P. Scholl, „Einführung in Embedded Systems".

Definition 5.4: Linker

*Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschienencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt** bzw. zusammenfügt, sodass unter anderem kein vermeidbarer **doppelter** Code darin vorkommt.*[a]

[a] P. D. P. Scholl, „Einführung in Embedded Systems".

Definition 5.5: Transpiler (bzw. Source-to-source Compiler)

*Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten*[a][b]

[a] Die Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.
[b] Thiemann, „Compilerbau".

Definition 5.6: Rekursiver Abstieg

*Es wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminalsymbol** enthält.*

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 5.7) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 5.6) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschafft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist[1].

Definition 5.7: LL(k)-Grammatik

*Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die nächsten k **Token** des **Eingabeworts** zu bestimmen ist[a]. Dabei steht **LL** für **l**eft-to-right und **l**eftmost-derivation, da das **Eingabewort** von **links nach rechts** geparsed 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 5.8: Earley Recognizer

*Ist ein **Recognizer**, der für **alle Kontextfreien Sprachen** das **Wortproblem** entscheiden kann und dies mittels **Dynamischer Programmierung** mit dem **Top-Down Ansatz** umsetzt.[a] [b] [c]*

***Eingabe** und **Ausgabe** des Algorithmus sind:*

- ***Eingabe:** **Eingabewort** w und **Konkrette Grammatik** $G_{Parse} = \langle N, \Sigma, P, S \rangle$*
- ***Ausgabe:** 0 wenn $w \notin L(G_{Parse})$[d] und 1 wenn $w \in L(G_{Parse})$*

*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 **Nicht-Terminalsymbole** dar*
- *a stellt ein **Terminalsymbol** dar*
- ***Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine **Produktion**, in der α **bereits geparst** wurde und β **noch geparst** werden muss*
- *Die **Indexierung** ist informell ausgedrückt so umgesetzt, dass die Indices **zwischen Tokennamen** liegen, also **Index** 0 **vor** dem ersten **Tokennamen** verortet ist, **Index** 1 **nach** dem ersten **Tokennamen** verortet ist und **Index** n **nach** dem **letzten Tokennamen** verortet ist*

*und davor müssen noch einige **Begriffe definiert** werden:*

- ***Zustandsmenge:** Für jeden der $n+1$ **Indices** j wird eine **Zustandsmenge** $Z(j)$ generiert*
- ***Zustand einer Zustandsmenge:** Ist ein **Tupel** $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die **aktuelle***

[1] Mehr **Erklärung** hierzu findet sich im Unterkapitel 2.4.

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****, welches das* ***Startsymbol*** S *auf der* ***linken Seite*** *des* ::=*-Symbols hat*

2. *es werden in der* ***aktuellen Zustandsmenge*** $Z(j)$ *die folgenden* ***Operationen ausgeführt****:*

   - ***Voraussage:*** *Für jeden* ***Zustand*** *in der* ***Zustandsmenge*** $Z(j)$*, der die Form* $(A ::= \alpha \bullet B\gamma, i)$ *hat, wird für jede* ***Produktion*** $(B ::= \beta)$ *in der Konkretten Grammatik, die ein* B *auf der* ***linken Seite*** *des* ::=*-Symbols hat ein* ***Zustand*** $(B ::= \bullet\beta, j)$ *zur* ***Zustandsmenge*** $Z(j)$ *hinzugefügt*

   - ***Überprüfung:*** *Für jeden* ***Zustand*** *in der* ***Zustandsmenge*** $Z(j)$*, der 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:*** *Für jeden* ***Zustand*** *in der* ***Zustandsmenge*** $Z(j)$*, der 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*** $(A ::= \beta\bullet, 0)$ *in der* ***Zustandsmenge*** $Z(n)$ *auftaucht, wobei* A *das* ***Startsymbol*** S *ist* $\Rightarrow w \in L(G_{Parse})$

   - ***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 **Konkrette Grammatik** G_{Parse} beschrieben wird.
[e] Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

Definition 5.9: Liveness Analyse

Findet heraus, welche ***Variablen*** *in welchen* ***Regionen*** *eines Programmes* ***verwendet*** *werden.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.10: Live Variable

Eine Location, deren momentaner Wert ***später*** *im Programmablauf noch* ***verwendet*** *wird. Man sagt auch die Location ist* ***live.***[a][b]

[a] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Live Variable**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.11: 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, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.12: Interference Graph

*Ein **ungerichteter Graph** mit **Locations** als **Knoten**, der eine **Kante** zwischen zwei Locations hat, wenn es sich bei beiden Locations **zu dem Zeitpunkt** um **Live Locations** handelt. In Bezug auf **Graph Coloring** bedeutet eine **Kante**, dass diese zwei Locations **nicht** die **gleiche Zahl**[a] zugewiesen bekommen dürfen.*[b]

[a]Bzw. **Farbe**.
[b]G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.13: Kontrollflussgraph

***Gerichteter Graph**, der den Kontrollfluss eines Programmes beschreibt.*[a]

[a]G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.14: Kontrollfluss

*Die **Reihenfolge** in der z.B. **Statements**, **Funktionsaufrufe** usw. eines Programmes ausgewertet werden*[a].

[a]Man geht hier von einem **imperativen** Programm aus.

Definition 5.15: Kontrollflussanalyse

***Analyse** des **Kontrollflusses** (Defintion 5.14) 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 **Linveness Analayse**.
[b]Bis diese sich **stabilisiert** haben
[c]G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 5.16: Two-Space Copying Collector

*Ein **Garbabe Collector** bei dem der **Heap** in `FromSpace` und `ToSpace` unterteilt wird und bei **nicht ausreichendem** Speicherplatz auf dem **Heap** alle Variablen, die in Zukunft noch verwendet werden vom `FromSpace` zum `ToSpace` kopiert werden. Der aktuelle `ToSpace` wird danach zum neuen `FromSpace` und der aktuelle `FromSpace` wird danach zum neuen `ToSpace`.*[a]

[a]G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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** $C^{PicoC}_{RETI\_PicoC}$ (Defintion 5.17) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache L_Python, in der der momentane Compiler C_{PicoC} für L_{PicoC} implementiert ist und die Unabhängigkeit von einer **anderen Maschiene**, die bisher immer für das Cross-Compiling notwendig war erreicht werden.

Definition 5.17: Self-compiling Compiler

*Compiler C^w_w, der in der Sprache L_w **geschrieben** ist, die er **selbst** kompiliert. Also ein Compiler, der sich **selbst** kompilieren kann.*[a]

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

Will man nun für eine Maschiene M_{RETI}, auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 5.20) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**[2] reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auf der **Maschiene** M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der mit der Maschienensprache B_{RETI} läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ nicht selbst komplett in der Maschienensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C^{Python}_{PicoC}, den man bereits in der Programmiersprache L_{Python} implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 5.19) darstellt, auf einer anderen Maschiene M_{other} dafür nutzt, damit dieser den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ für die Maschiene M_{RETI} kompiliert bzw. **bootstraped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschiene M_{other} auf die Maschiene M_{RETI} kopiert.[3]

[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 Maschiene M_{RETI} die Programmiersprache L_{Python} bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auch mithife des **Cross-Compilers** C^{Python}_{PicoC} als **externe Entität** und der Programmiersprache L_{Python} auf der Maschiene M_{RETI} selbst kompiliert werden.

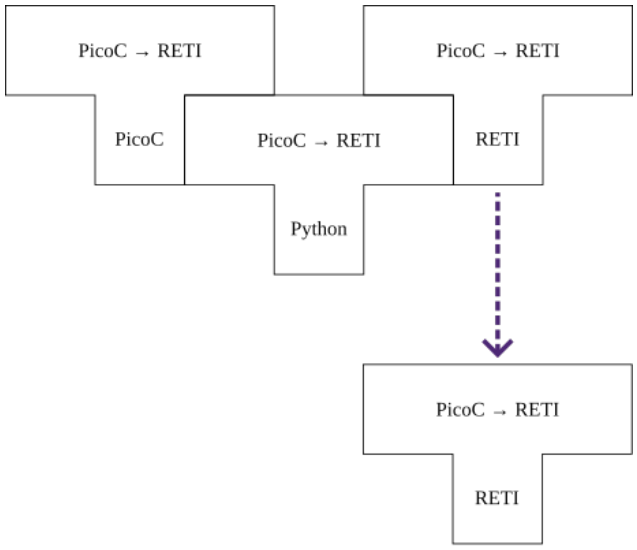


Abbildung 5.2: *Cross-Compiler als Bootstrap Compiler*

Anmerkung

Einen ersten **minimalen Compiler** $C_{2\_w\_min}$ für eine Maschiene M_2 und Wunschsprache L_w kann man entweder mittels eines **externen Bootstrap Compilers** C_w^o kompilieren[a] oder man schreibt ihn direkt in der **Maschienensprache** B_2 bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** A_2.

Die letzte Option wäre allerdings nur beim allerersten Compiler C_{first} für eine allererste **abstraktere Programmiersprache** L_{first} mit Schleifen, Verzweigungen usw. notwendig gewesen. Ansonsten hätte man immer eine Kette, die beim allersten Compiler C_{first} anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

[a] In diesem Fall, dem **Cross-Compiler** C_{PicoC}^{Python}.

Definition 5.18: Minimaler Compiler

*Compiler C_{w_min}, der nur die **notwendigsten Funktionalitäten** einer Wunschsprache L_w, wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** C_w^w oder einer **ersten Version** $C_{w_i}^{w_i}$ des Self-compiling Compilers C_w^w wichtig sind.*[a b]

[a] Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.
[b] Thiemann, „Compilerbau“.

Definition 5.19: Boostrap Compiler

*Compiler C_w^o, der es ermöglicht einen **Self-compiling Compiler** C_w^w zu **boostrapen**, indem der Self-compiling Compiler C_w^w mit dem **Bootstrap Compiler** C_w^o **kompiliert** wird*[a]*. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** C_w^w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.*[b]

[a] Dabei kann es sich um einen **lokal** auf der Maschiene selbst laufenden Compiler oder auch um einen **Cross-Compiler**

handeln.
[b] Thiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der einen **minimalen Compiler** (Definition 5.18) für eine Teilmenge der **Programmiersprache** C bzw. L_C darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw. L_C für die Maschiene M_{RETI} mittels **Bootstrapping** implementieren.[4]

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem **minimalen Compiler** $C^{PicoC}_{RETI\_PicoC}$, wie in Subdefinition 5.20.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Definition 5.20: Bootstrapping

*Wenn man einen **Self-compiling Compiler** C^w_w einer Wunschsprache L_w auf einer **Zielmaschine** M zum laufen bringt[abcd]. Dabei ist die Art von **Bootstrapping** in 5.20.1 nochmal gesondert hervorzuheben:*

5.20.1: *Wenn man die **aktuelle Version** eines **Self-compiling Compilers** $C^{w_i}_{w_i}$ der Wunschsprache L_{w_i} mithilfe von **früheren Versionen** seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ kompiliert wird und schafft es so **iterativ** immer umfangreichere Compiler zu bauen.[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^w_w auf der Maschiene M zum laufen gebracht, so kann man den Compiler auf der Maschiene M weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache L_o, in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.
[d] Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute **Probe aufs Exempel** darstellen, dass der Compiler auch wirklich funktioniert.
[e] Es ist hierbei theoretisch nicht notwendig den **letzten** Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ für das Kompilieren des **neuen** Self-compiling Compilers $C^{w_i}_{w_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C^{w_{i-3}}_{w_{i-3}}$ auch bereits alle Funktionalitäten, die beim Schreiben des **Self-compiling Compilers** C^w_w verwendet werden kompilieren kann.
[f] Der Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, 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“.

[4] Natürlich könnte man aber auch einfach den **Cross-Compiler** C^{Python}_{PicoC} um weitere Funktionalitäten von L_C erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache L_{Python}.

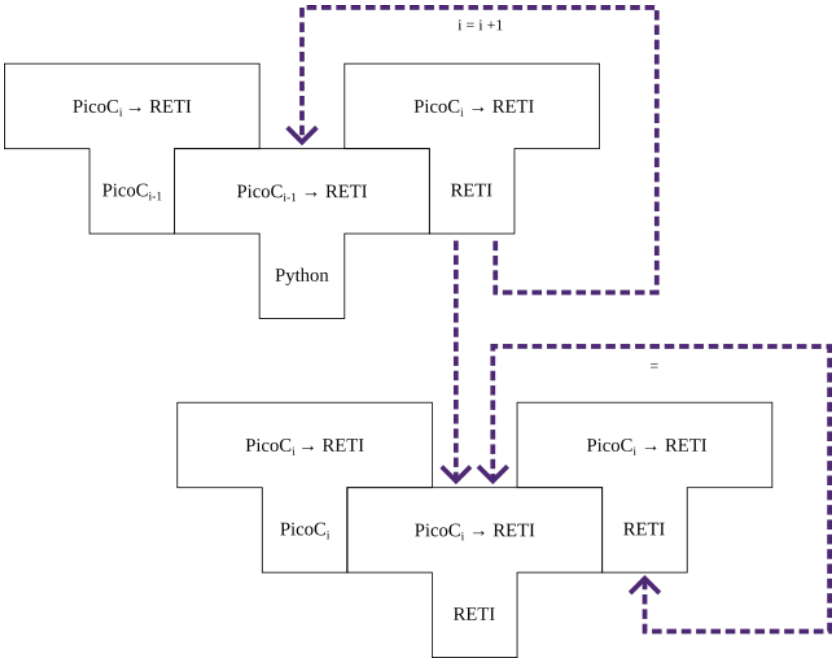


Abbildung 5.3: *Iteratives Bootstrapping*

Anmerkung

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 5.20.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 5.20.1 eine frühere Version $C_{w_{i-1}}^{w_{i-1}}$ nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

Literatur

Online

- *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).
- *Parsing Expressions · Crafting Interpreters*. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09. 07. 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. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 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.

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, Prof. Dr. Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Nebel, Prof. Dr. 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, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03. 08. 2022).
- Scholl, Prof. Dr. Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Prof. Dr. Peter. „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).