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.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Codeverzeichnis

# Tabellenverzeichnis

# Definitionsverzeichnis

# Grammatikverzeichnis

# 1 Motivation

## 1.1 RETI

... basiert auf ... der Vorlesung C. Scholl, „Betriebssysteme“.

**Definition 1.1: Caller-save Register**

*a*

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

**Definition 1.2: Callee-save Register**

*a*

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

## 1.2 PicoC

## 1.3 Aufgabenstellung

## 1.4 Eigenheiten der Sprache C

**Definition 1.3: Deklaration**

*a*

[a] P. Scholl, „Einführung in Embedded Systems“.

**Definition 1.4: Definition**

*a*

[a] P. Scholl, „Einführung in Embedded Systems“.

**Definition 1.5: Allokation**

*a*

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

**Definition 1.6: Initialisierung**

*a*

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

**Definition 1.7: Scope**

*a*

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

**Definition 1.8: Call by value**

*a*

[a] Bast, „Programmieren in C“.

**Definition 1.9: Call by reference**

*a*

[a] Bast, „Programmieren in C“.

## 1.5 Richtlinien

# 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 4.0.1 zu belegen (siehe Subkapitel 4.2).

**Definition 2.1: Interpreter**

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

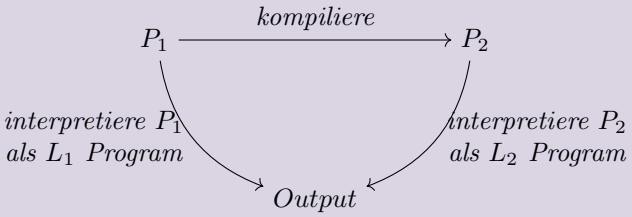
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen* ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 2.42) und führt je nach Komposition der* ***Nodes*** *des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

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

**Definition 2.2: Compiler**

***Kompiliert*** *ein 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 das Program $P_1$ in das Program $P_2$ so übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen $L_1$ und $L_2$* ***interpretiert*** *werden, der gleiche* ***Output*** *rauskommt. Also beide Programme $P_1$ und $P_2$ die gleiche* ***Semantik*** *haben und sich nur* ***syntaktisch*** *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

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 4.2) 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 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition 2.4) 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 2.8) haben.

[b] C. Scholl, „Betriebssysteme“.

**Definition 2.4: Assemblersprache (bzw. engl. Assembly Language)**

*Eine sehr **hardwarenahe** Programmiersprache, derren **Instructions** eine starke Entsprechung zu bestimmten Maschienenbefehlen bzw. Folgen von Maschienenbefehlen*[a] *haben. Viele **Instructions** 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 Instructions und drumherum*[c].[d]

[a] Instructions der Assemblersprache, die mehreren Maschienenbefehlen entsprechen werden auch als **Pseudo-Instructions** 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 Instructions 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. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschienencode** bzw. **Objectcode** (Definition 2.6) 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 2.7) zu Maschiendencode zusammengesetzt wird ausführbar

ist.

**Definition 2.5: 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. Scholl, „Einführung in Embedded Systems“.

**Definition 2.6: 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. Scholl, „Einführung in Embedded Systems“.

**Definition 2.7: Linker**

*Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschienencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt**, sodass unter anderem kein vermeidbarer **doppelter** Code darin vorkommt.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

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 2.8) 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 2.8: 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, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.*[a]

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

[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.

**Definition 2.9: 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 2.10: 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] 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]

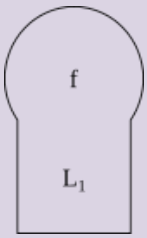
### 2.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus dem Paper 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 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschiene (Definition 2.14) zusammen.

**Definition 2.11: T-Diagram Programm**

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




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

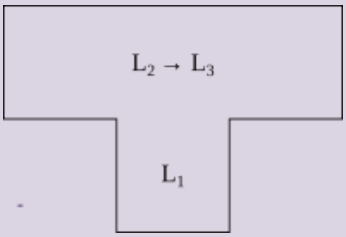
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein $L$ dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

[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 2.12: 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.*

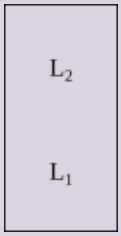
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 4.0.1.*[b]




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

**Definition 2.13: T-Diagram Interpreter**

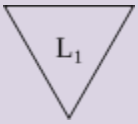
*Repräsentiert einen **Interpreter**, der in der **Sprache** $L_1$ geschrieben ist und **Programme** in der **Sprache** $L_2$ interpretiert.*[a]




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

**Definition 2.14: 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] 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.

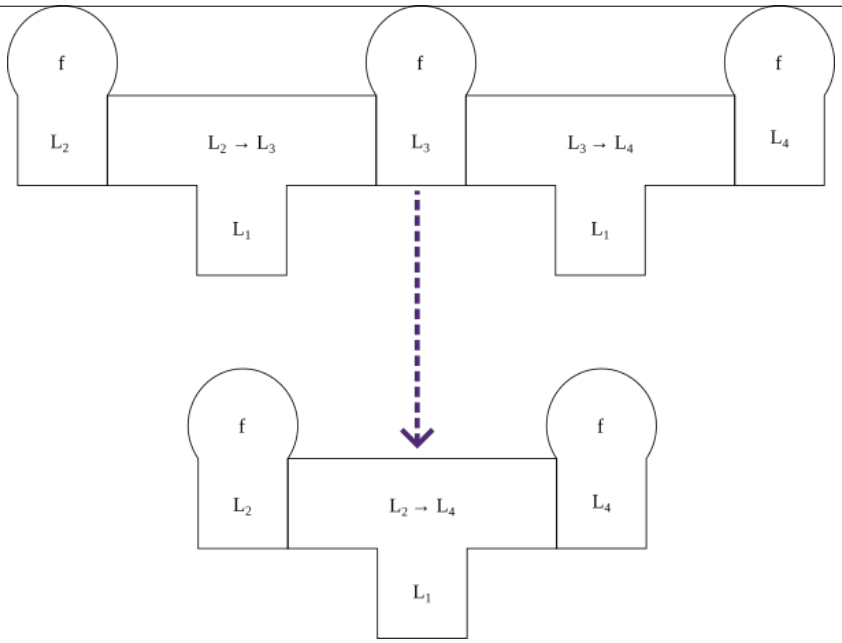



**Abbildung 2.1:** Horinzontale Übersetzungszwischenschritte zusammenfassen

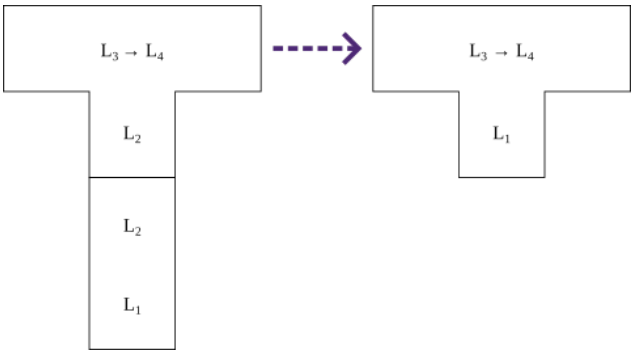



**Abbildung 2.2:** Vertikale Interpretierungszwischenschritte zusammenfassen

## 2.2 Formale Sprachen

**Definition 2.15: Sprache**

*a*

[a] Nebel, „Theoretische Informatik“.

**Definition 2.16: Chromsky Hierarchie**

*a*

[a] Nebel, „Theoretische Informatik“.

**Definition 2.17: Grammatik**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.18: Reguläre Sprachen**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.19: Kontextfreie Sprachen**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.20: Ableitung**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.21: Links- und Rechtsableitung**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.22: 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*.

### 2.2.1 Mehrdeutige Grammatiken

**Definition 2.23: Ableitungsbaum**

[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.24: Mehrdeutige Grammatik**

*a*

[a] Nebel, „Theoretische Informatik“.

### 2.2.2 Präzidenz und Assoziativität

**Definition 2.25: Assoziativität**

*a*

[a] *Parsing Expressions · Crafting Interpreters*.

**Definition 2.26: Präzidenz**

*a*

[a] *Parsing Expressions · Crafting Interpreters*.

**Definition 2.27: Wortproblem**

*a*

[a] Nebel, „Theoretische Informatik“.

**Definition 2.28: LL(k)-Grammatik**

*Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die nächsten k **Symbole** des **Eingabeworts** bzw. in Bezug zu Compilerbau **Token** des **Inputstrings** 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“.

## 2.3 Lexikalische Analyse

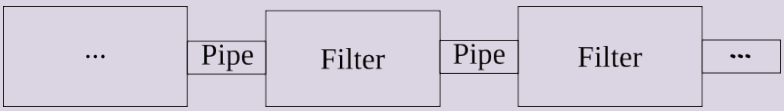
Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.29) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8**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

**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“.

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.31) genannt.

**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 **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token** $T$ einer **Sprache** $L_{Lex}$ matched.*[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.32) im **Inputstring** 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 $\Sigma^*$ 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.

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`[4] und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort $\epsilon$ 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`[5], 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`[6]. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

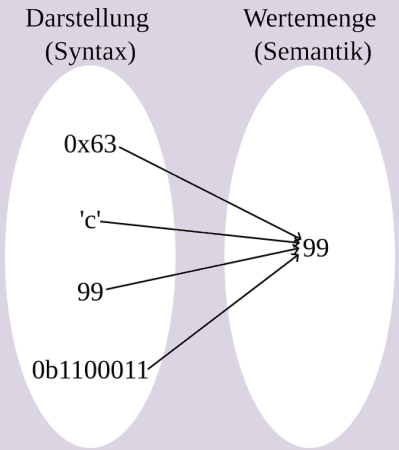
Die **Grammatik** $G_{Lex}$, die zur Beschreibung der Token $T$ der Sprache $L_{Lex}$ verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[7], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 3.1.1 liefert den Beweis,

[4] 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.
[5] Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Nodes haben will, damit unter anderem **mehr Code** in eine Zeile passt.
[6] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.
[7] Man nennt das auch einem **Lookahead** von 1

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[8]. 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 2.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

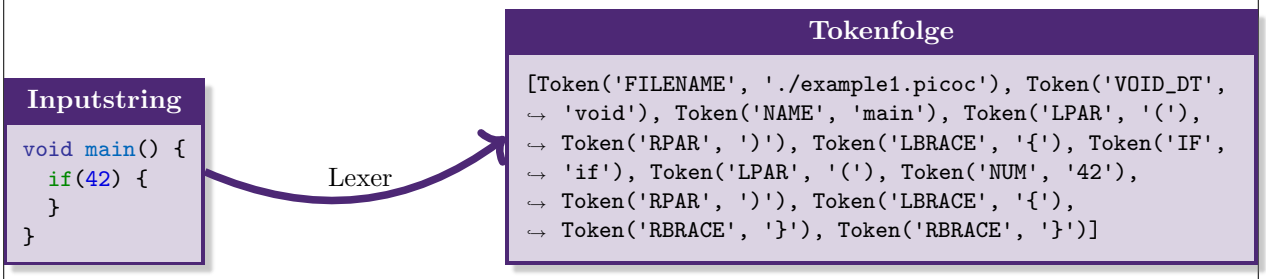



**Abbildung 2.3:** Veranschaulichung der Lexikalischen Analyse

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

[8] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.37), ein **Derivation Tree** (Definition 2.36) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 2.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

**Definition 2.35: Konkrette Syntax**

*__Syntax__ einer __Sprache__, die durch die __Grammatiken__ $G_{Lex}$ und $G_{Parse}$ zusammengenommen beschrieben wird.*

*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)*.

**Definition 2.36: Derivation Tree (bzw. Parse Tree)**

*__Compilerinterne Darstellung__ eines in __Konkretter Syntax__ geschriebenen Inputstrings als __Baumdatenstruktur__, in der __Nichtterminalsymbole__ die __Inneren Knoten__ der Baumdatenstruktur und __Terminalsymbole__ die __Blätter__ der Baumdatenstruktur bilden. Jedes zum Ableiten des Inputstrings verwendetete __Nicht-Terminalsymbol__ einer __Produktion__ der __Grammatik__ $G_{Parse}$, die ein Teil der __Konkrette Syntax__ ist, bildet einen eigenen __Inneren Knoten__.*

*Der __Derivation Tree__ wird optimalerweise immer so konstruiert bzw. die __Konkrette Syntax__ immer so definiert, dass sich möglichst einfach ein __Abstract Syntax Tree__ daraus konstruieren lässt.*[a]

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

**Definition 2.37: Parser**

*Ein __Parser__ ist ein Programm, dass aus einem Inputstring, der in __Konkretter Syntax__ geschrieben ist, eine compilerinterne Darstellung, den __Derivation Tree__ 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 einen Inputstring von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.37 verwendet.

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

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 Inputstring 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 2.38), denn ein Parser löst das **Wortproblem** (Definition 2.27) für die **Sprache**, die durch die **Konkrette Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Derivation Tree**.

**Definition 2.38: Recognizer (bzw. Erkenner)**

*Entspricht dem Maschienenmodell eines **Automaten**. Im Bezug auf Compilerbau entspricht der **Recognizer** einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** erkennt, ob ein Inputstring bzw. **Wort** sich mit den Produktionen der **Konkrette Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkretten Syntax** beschrieben wird oder nicht*[ab]

[a]Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.
[b]Thiemann, „Compilerbau“.

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

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

  Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist, weil der **Eingabewert** bzw. der **Inputstring** 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**. Dabei 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-Terminal** enthält.

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.22) allerdings nicht möglich, ohne die Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der **Grammatik** entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

**Rekursiver Abstieg** kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.28) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** 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 der **Inputstring** sich **nicht** mit der verwendeten 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 Inputstring **vorauszuschauen**. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.[c]

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkretten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.[d]
- **Chart Parser:** 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]

[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 *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 **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 2.39) und **Visitors** (Definition 2.40) generiert und ist das Endprodukt der **Syntaktischen Analyse**. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inputstring von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.41).

**Definition 2.39: Transformer**

*Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** besucht und beim Antreffen eines bestimmten Knoten des **Derivation Tree** einen entsprechenden Knoten des **Abstract Syntax Tree** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstract Syntax Tree** konstruiert.*[a]

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

**Definition 2.40: Visitor**

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

[a] Kann theoretisch auch zur Konstruktion eines **Abstract Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstract Syntax Tree** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.
[b] *Transformers & Visitors — Lark documentation*.

**Definition 2.41: Abstrakte Syntax**

***Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstract Syntax Trees** möglich sind.*

*Jene Produktionen, die in der **Konkretten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.*[a]

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

**Definition 2.42: Abstract Syntax Tree**

***Compilerinterne Darstellung** eines Programs, in welcher sich anhand der Knoten auf dem **Pfad** von der **Wurzel** zu einem **Blatt** nicht mehr direkt nachvollziehen lässt, durch welche **Produktionen** dieses Blatt abgeleitet wurde.*

*Der **Abstract Syntax Tree** hat einmal den Zweck, dass die **Kompositionen**, die die Knoten bilden können **semantisch** näher an den **Instructions eines Assemblers** dran sind und, dass man mit einem **Abstract Syntax Tree** bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, möglichst schnell die Fragen beantworten kann, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.*[a]

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

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstract Syntax Tree** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Inputstrings 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 2.4 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

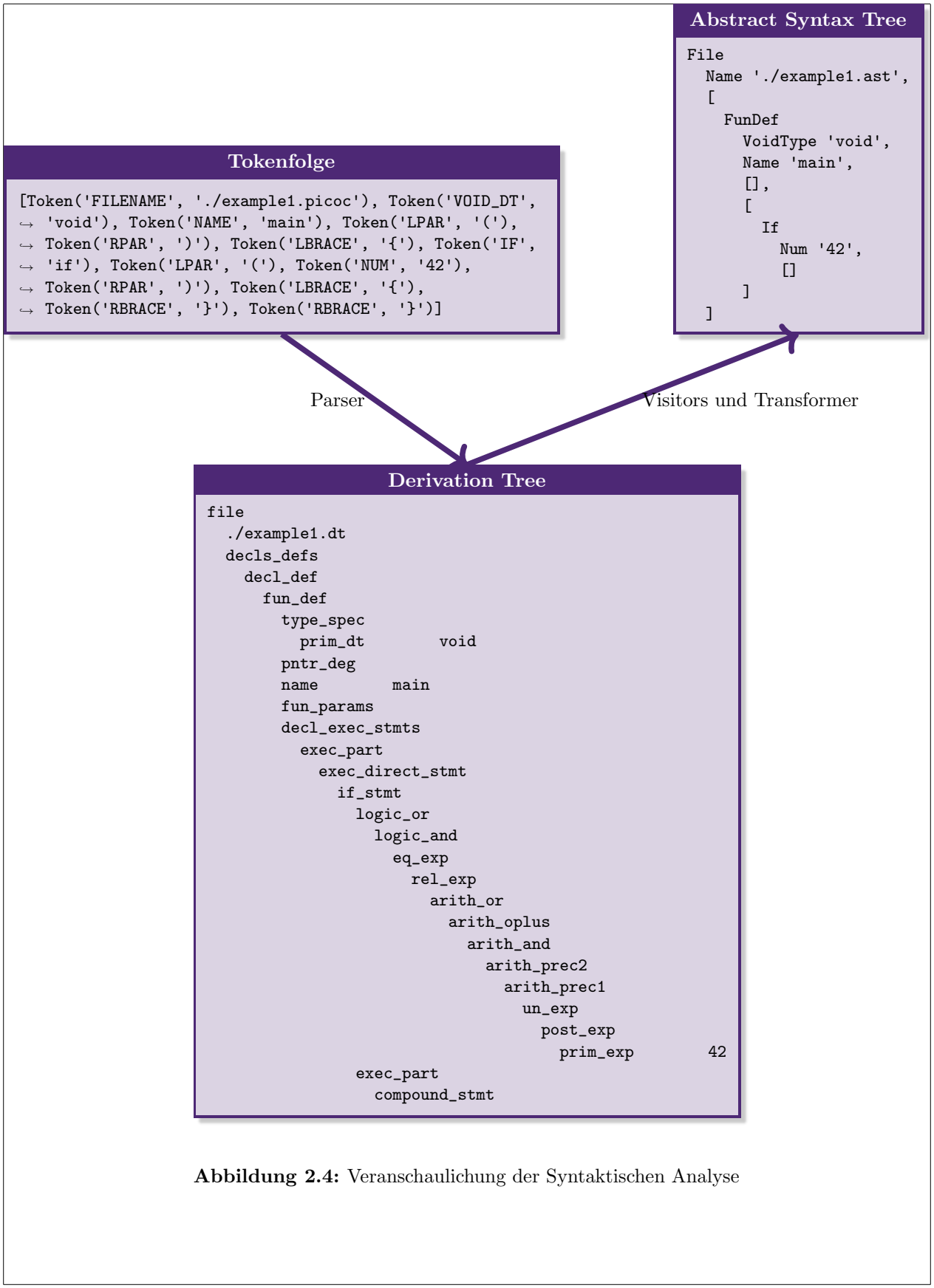



**Abbildung 2.4:** Veranschaulichung der Syntaktischen Analyse

## 2.5 Code Generierung

**Definition 2.43: Pass**

*a*

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

**Definition 2.44: Monadische Normalform**

*a*

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

Ein echter Compiler verwendet Graph Coloring ... Register ...

## 2.6 Fehlermeldungen

**Definition 2.45: Fehlermeldung**

***Benachrichtigung*** *beliebiger Form, die darüber informiert, dass:*

1. *Ein Program beim* ***Kompilieren*** *von der* ***Konkretten Syntax*** *abweicht, also der* ***Inpustring*** *sich nicht mit der Konrektten Syntax* ***ableiten*** *lässt oder auf etwas* ***zugegriffen*** *werden soll, was noch* ***nicht*** *deklariert oder definiert wurde.*

2. *Beim Ausführen eine* ***verbotene*** *Operation ausgeführt wurde.*[a]

[a] *Errors in C/C++ - GeeksforGeeks*.

### 2.6.1 Kategorien von Fehlermeldungen

# 3 Implementierung

## 3.1 Lexikalische Analyse

### 3.1.1 Konkrette Syntax für die Lexikalische Analyse

| | | | |
|---|---|---|---|
| *COMMENT* | ::= | "//" /[^\n]*/ \| "/*" /(. \| \n)*?/ "*/" | *L_Comment* |
| *RETI_COMMENT.2* | ::= | "//""␣"?"#"/[^\n]*/ | |
| *DIG_NO_0* | ::= | "1" \| "2" \| "3" \| "4" \| "5" | *L_Arith* |
| | \| | "6" \| "7" \| "8" \| "9" | |
| *DIG_WITH_0* | ::= | "0" \| *DIG_NO_0* | |
| *NUM* | ::= | "0" \| *DIG_NO_0DIG_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* | |
| *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* | |
| *LOGIC_NOT* | ::= | "!" | |

**Grammar 3.1.1:** Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 1

| | | | |
|---|---|---|---|
| $INT\_DT.2$ | ::= | "*int*" | *L_Assign_Alloc* |
| $INT\_NAME.3$ | ::= | "*int*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+ | |
| $CHAR\_DT.2$ | ::= | "*char*" | |
| $CHAR\_NAME.3$ | ::= | "*char*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+ | |
| $VOID\_DT.2$ | ::= | "*void*" | |
| $VOID\_NAME.3$ | ::= | "*void*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+ | |
| *prim_dt* | ::= | $INT\_DT \mid CHAR\_DT \mid VOID\_DT$ | |

**Grammar 3.1.2:** Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 2

### 3.1.2 Basic Lexer

## 3.2 Syntaktische Analyse

### 3.2.1 Konkrette Syntax für die Syntaktische Analyse

In 3.2.1

| | | | |
|---|---|---|---|
| *prim_exp* | ::= | *name* \| *NUM* \| *CHAR* \| "("*logic_or*")" | *L_Arith +* |
| *post_exp* | ::= | *array_subscr* \| *struct_attr* \| *fun_call* | *L_Array +* |
| | | \| *input_exp* \| *print_exp* \| *prim_exp* | *L_Pntr +* |
| *un_exp* | ::= | *un_opun_exp* \| *post_exp* | *L_Struct + L_Fun* |
| *input_exp* | ::= | "*input*""("")" | *L_Arith* |
| *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_oprel_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*<br>(","  "."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*")"";" | |

**Grammar 3.2.1:** Konkrette Syntax 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* | |

**Grammar 3.2.2:** Konkrette Syntax für die Syntaktische Analyse in EBNF, Teil 2

### 3.2.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**[1]. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 3.1 aufgelistet.

| Präzidenz | Operator | 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 ← |
| 13 | `a,b` | Komma | Links, dann rechts → |

**Tabelle 3.1:** Präzidenzregeln von PicoC

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

### 3.2.3 Derivation Tree Generierung

#### 3.2.3.1 Early Parser

#### 3.2.3.2 Codebeispiel

```
1 struct st {int *(*attr)[5][6];};
2
3 void main() {
4   struct st *(*var)[3][2];
5 }
```

**Code 3.1:** PicoC Code für Derivation Tree Generierung

```
1  file
2    ./example_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                   5
21                   6
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                    array_dims
45                      3
46                      2
```

**Code 3.2:** Derivation Tree nach Derivation Tree Generierung

### 3.2.4 Derivation Tree Vereinfachung

#### 3.2.4.1 Visitor

#### 3.2.4.2 Codebeispiel

Beispiel aus Subkapitel 3.2.3.2 wird fortgeführt.

```
1 file
2   ./example_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                  5
14                  6
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                      3
38                      2
39                    pntr_decl
40                      pntr_deg  *
41                      array_decl
42                        array_dims
43                        type_spec
44                          struct_spec
45                            name  st
46                name  var
```

**Code 3.3:** Derivation Tree nach Derivation Tree Vereinfachung

### 3.2.5 Abstrakt Syntax Tree Generierung

#### 3.2.5.1 PicoC-Knoten

| 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 ∧ b, a & b, a \| b, a && b, a \|\| b`. |
| `Eq(), NEq(), Lt(), LtE(), Gt(), GtE()` | Die **Relationen** `rel`: `a == b, a != b, a < b, a <= b, a > b, a >= b`. |
| `Const(), Writeable()` | Die **Type Qualifier** `type_qual`: `const`, was für ein **nicht beschreibbare Konstante** steht und das **nicht** Angeben von `const`, was für einen **beschreibbare** Variable steht. |
| `IntType(), CharType(), VoidType()` | Die **Type Specifier** für **Primitiven Datentypen**, die in der Abstrakten Syntax, um eine intuitive Bezeichnung zu haben einfach nur unter **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 3.2:** 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 **Pointerdatentyp** steht: `<prim_dt> *<var>`, wobei das **Attribut** `num` die **Anzahl zusammengefasster Pointer** angibt und `datatype` der Datentyp ist, auf den der oder die **Pointer** zeigen. |
| `Ref(exp, datatype, error_data)` | Container, der für die Anwendung des **Referenz-Operators** `&<var>` steht und die **Adresse** einer **Location** (Definition 3.2) 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 **Array-** oder **Pointerdatentyp**: `<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 **Arraydatentyp** steht: `<prim_dt> <var>[<i>]`, wobei das **Attribut** `nums` eine Liste von `Num('x')` ist, die die **Dimensionen** des Arrays angibt und `datatype` der Datentyp ist, der über das Anwenden von `Subscript()` auf das Array zugreifbar ist. |
| `Array(exps, datatype)` | Container für den **Initializer** eines **Arrays**, dessen Einträge `exps` weitere Initializer für eine **Array-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-Mon Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind. |
| `Subscr(exp1, exp2)` | Container für den **Indexzugriff** auf einen **Array-** oder **Pointerdatentyp**: `<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 **Structdatentyp**: `struct <name>`, wobei das **Attribut** `name` festlegt, welchen **selbst definierte** Structdatentyp dieser Container-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 3.3:** 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 **Array-Dimension** oder zu einem Initializer für ein **Struct** oder zu einem **Logischen Ausdruck**. Des Weiteren besitzt er ein versteckte Attribut `datatype`, welches für den **PicoC-Mon Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind. |
| `StructDecl(name, allocs)` | Container für die Deklaration eines **selbstdefinierten Structdatentyps**, z.B. `struct <var> {<datatype> <attr1>; <datatype> <attr2>;};`, wobei `name` der **Bezeichner** des Structdatentyps ist und `allocs` eine Liste von Bezeichnern der **Attribute** des Structdatentyps mit dazugehörigem **Datentyp**, wofür sich der **Container-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 **Container-Knoten** `Alloc(type_spec, datatype, name)` als Cotainer für die Parameter dient. |

**Tabelle 3.4:** 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 **Container-Knoten** `Alloc(type_spec, datatype, name)` als Cotainer 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 die **Jump Instruction** 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 Bezeichners des **Labels** (Definition 3.1) des Blocks ist und `stmts_instrs` eine **Liste von Statements** oder **Instructions**. Zudem besitzt er noch 3 versteckte Attribute, wobei `instrs_before` die Zahl der **Instructions** vor diesem **Block** zählt, `num_instrs` die Zahl der Instructions 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 **Abstract Syntax Trees** 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 3.5:** PicoC-Knoten Teil 4

**Definition 3.1: Label**

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

[a] **tab:picoc˙knoten˙teil˙4**.

**Definition 3.2: 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.*[a]

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

Die ausgegrauten Attribute der PicoC-Nodes sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Nodes** 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 **Token-Knoten** (Definition 3.3) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 3.4) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

**Definition 3.3: Token-Knoten**

*Ersetzt ein **Token** bei der Generierung des **Abstract Syntax Tree**, damit der Zugriff auf Knoten des Abstract Syntax Tree möglichst **simpel** ist und keine vermeidbaren Fallunterscheidungen gemacht werden müssen.*

***Token-Knoten** entsprechen im Abstract Syntax Tree **Blättern**.*[a]

[a] Thiemann, „Compilerbau“.

**Definition 3.4: Container-Knoten**

*Dient als **Container** für andere **Container-Knoten** und **Token-Knoten**. Die **Container-Knoten** werden optimalerweise immer so gewählt, dass sie **mehrere Produktionen** der **Konkretten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.*[a]

***Container-Knoten** entsprechen im Abstract Syntax Tree **Inneren Knoten**.*[b]

[a] Wie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. `1 % 3` und **Logischen Ausdrücke**, wie z.B. `1 && 2 < 3`, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.
[b] Thiemann, „Compilerbau“.

#### 3.2.5.2 RETI-Knoten

| RETI-Knoten | Beschreibung |
|---|---|
| `Program(name, instrs)` | Container für alle **Instructions**: `<name> <instrs>`, wobei `name` der **Dateiname** der Datei ist, die erstellt wird und `instrs` eine **Liste von Instructions** ist. |
| `Instr(op, args)` | Container für eine **Instruction**: `<op> <args>`, wobei `op` eine **Operation** ist und `args` eine **Liste von Argumenten** für dieser Operation. |
| `Jump(rel, im_goto)` | Container für eine **Jump-Instruction**: `JUMP<rel> <im>`, wobei `rel` eine **Relation** ist und `im_goto` ein **Immediate Value** `Im(val)` für die **Anzahl an Speicherzellen**, um die relativ zur **Jump-Instruction** 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] C. Scholl, „Betriebssysteme“

**Tabelle 3.6:** RETI-Knoten

#### 3.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Hier sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax** 3.2.1 enthalten sind.

| 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** der Instruction, die in diesem Kontext direkt nach dem Sprung zum Block einer anderen Funktion steht. |

**Tabelle 3.7:** Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 3.7 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 Syntax

| | | | |
|---|---|---|---|
| $un\_op$ | $::=$ | $Minus() \mid Not()$ | *L_Arith* |
| $bin\_op$ | $::=$ | $Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$ | |
| | $\mid$ | $Oplus() \mid And() \mid Or()$ | |
| $exp$ | $::=$ | $Name(str) \mid Num(str) \mid Char(str)$ | |
| | $\mid$ | $BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$ | |
| | $\mid$ | $UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), None)$ | |
| $exp\_stmts$ | $::=$ | $Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$ | |
| | $\mid$ | $Call(Name('print'), \langle exp\rangle)$ | |
| $un\_op$ | $::=$ | $LogicNot()$ | *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* |
| $datatype$ | $::=$ | $IntType() \mid CharType() \mid VoidType()$ | |
| $lhs$ | $::=$ | $Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str)) \mid \langle rel\_loc\rangle$ | |
| $exp\_stmts$ | $::=$ | $Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$ | |
| $stmt$ | $::=$ | $Assign(\langle lhs\rangle, \langle exp\rangle)$ | |
| | $\mid$ | $Exp(\langle exp\_stmts\rangle)$ | |
| $datatype$ | $::=$ | $PntrDecl(Num(str), \langle datatype\rangle)$ | *L_Pntr* |
| $deref\_loc$ | $::=$ | $Ref(\langle ref\_loc\rangle) \mid \langle ref\_loc\rangle$ | |
| $ref\_loc$ | $::=$ | $Name(str)$ | |
| | $\mid$ | $Deref(\langle deref\_loc\rangle, \langle exp\rangle)$ | |
| | $\mid$ | $Subscr(\langle deref\_loc\rangle, \langle exp\rangle)$ | |
| | $\mid$ | $Attr(\langle ref\_loc\rangle, Name(str))$ | |
| $exp$ | $::=$ | $Deref(\langle deref\_loc\rangle, \langle exp\rangle)$ | |
| | $\mid$ | $Ref(\langle ref\_loc\rangle)$ | |
| $datatype$ | $::=$ | $ArrayDecl(Num(str)+, \langle datatype\rangle)$ | *L_Array* |
| $exp$ | $::=$ | $Subscr(\langle deref\_loc\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$ | |
| $datatype$ | $::=$ | $StructSpec(Name(str))$ | *L_Struct* |
| $exp$ | $::=$ | $Attr(\langle ref\_loc\rangle, Name(str))$ | |
| | $\mid$ | $Struct(Assign(Name(str), \langle exp\rangle)+)$ | |
| $decl\_def$ | $::=$ | $StructDecl(Name(str),$ | |
| | | $\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$ | |
| $stmt$ | $::=$ | $If(\langle exp\rangle, \langle stmt\rangle*)$ | *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* |
| $exp\_stmts$ | $::=$ | $Call(Name(str), \langle exp\rangle*)$ | |
| $stmt$ | $::=$ | $Return(\langle exp\rangle)$ | |
| $decl\_def$ | $::=$ | $FunDecl(\langle datatype\rangle, Name(str),$ | |
| | | $\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$ | |
| | $\mid$ | $FunDef(\langle datatype\rangle, Name(str),$ | |
| | | $\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$ | |
| $file$ | $::=$ | $File(Name(str), \langle decl\_def\rangle*)$ | *L_File* |

**Grammar 3.2.3:** Abstrakte Syntax für $L_{PiocC}$

#### 3.2.5.5 Transformer

#### 3.2.5.6 Codebeispiel

Beispiel welches in Subkapitel 3.2.3.2 angefangen wurde, wird hier fortgeführt.

```
File
  Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')],
        ↪  PntrDecl(Num('1'), IntType('int')))), Name('attr'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
        ↪  PntrDecl(Num('1'), StructSpec(Name('st'))))), Name('var')))
      ]
  ]
```

**Code 3.4:** Abstract Syntax Tree aus vereinfachtem Derivarion Tree generiert

## 3.3 Code Generierung

### 3.3.1 Übersicht

Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse**, wird in diesem Kapitel aus den verschiedenen **Kompositionen** von **Container-Knoten** und **Token-Knoten** im Abstract Syntax Tree das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton 2.10). 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 3.1 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.

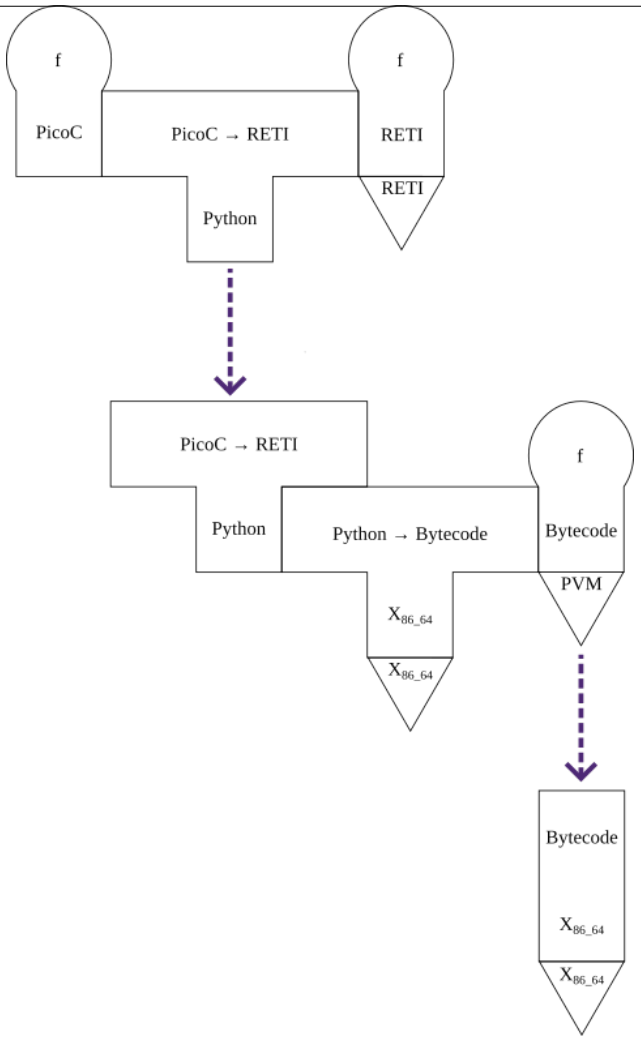



**Abbildung 3.1:** Cross-Compiler Kompiliervorgang ausgeschrieben

Dieses längliche **T-Diagram** in Abbildung 3.1 lässt sich zusammenfassen, sodass man das **T-Diagram** in Abbildung 3.2 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in $\mathtt{X}_{86\_64}$-Maschienensprache geschrieben ist.

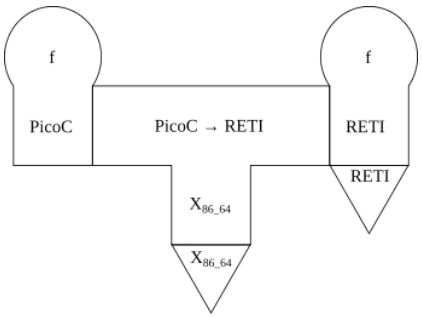



**Abbildung 3.2:** Cross-Compiler Kompiliervorgang Kurzform

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 3.3 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 **Abstract Syntax Tree** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstract Syntax Tree** in seiner eigenen **Sprache** zu generieren.

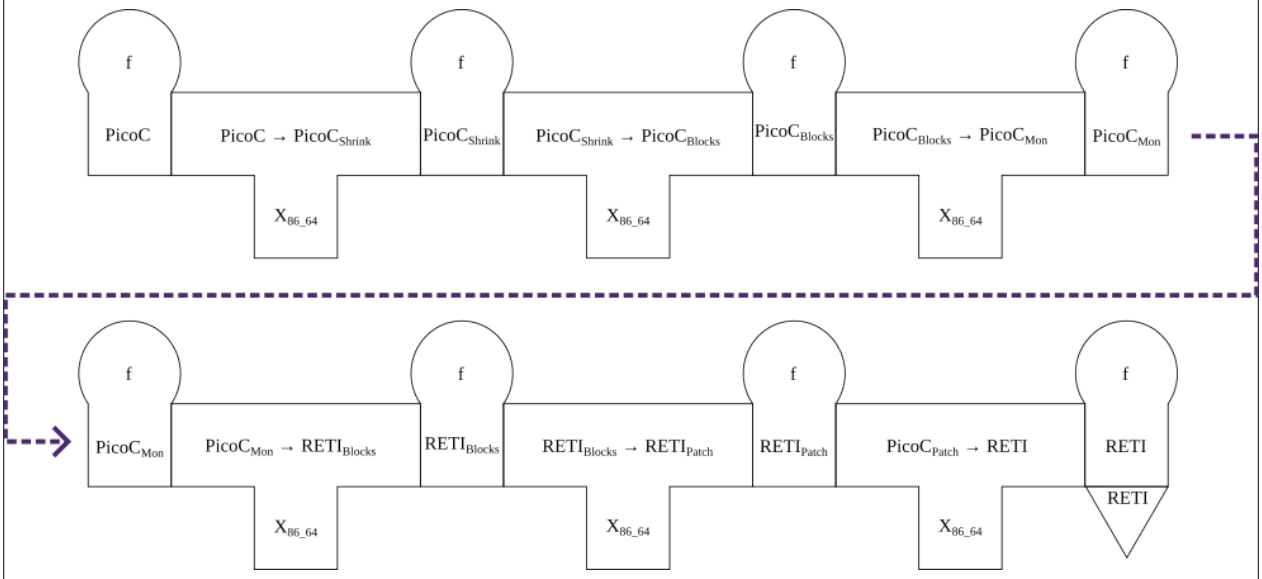



**Abbildung 3.3:** Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 3.3.2 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. Die von den **Passes** generierten **Abstract Syntax Trees** werden dabei mit jedem **Pass** der **Syntax** des **RETI-Code's** immer ähnlicher werden. Jeder Pass sollte dabei möglichst eine Aufgabe übernehmen, da der Sinn von **Passes** ist, die Kompilierung in mehrere kleinschrittige Aufgaben runterzuberechen. Wie es auch schon der Zweck des **Dervivation Tree** in der Syntaktischen Analyse war, eine Zwischenstufe zum **Abstract Syntax Tree** darzustellen, aus der sich unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstract Syntax Tree** generieren lies.

In den darauffolgenden Unterkapiteln 3.3.3, 3.3.4, 3.3.5 und 3.3.7 werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 3.3.2 nicht ausreichend vertieft wurden. Viele der verwendenten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung C. Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird[2]. 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[3] finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[4].

[2] Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

[3] https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples

[4] Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstract Syntax Trees** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

### 3.3.2 Passes

#### 3.3.2.1 PicoC-Shrink Pass

##### 3.3.2.1.1 Zweck

##### 3.3.2.1.2 Codebeispiel

```
// based on a example program from Christoph Scholl's Operating Systems lecture

void main() {
  int n = 4;
  int res = 1;
  while (1) {
    if (n == 1) {
      return;
    }
    res = n * res;
    n = n - 1;
  }
}
```

**Code 3.5:** PicoC Code für Codebespiel

```
File
  Name './example_faculty_it.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
        While
          Num '1',
          [
            If
              Atom
                Name 'n',
                Eq '==',
                Num '1',
              [
                Return(Empty())
              ]
            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
          ]
      ]
  ]
```

**Code 3.6:** Abstract Syntax Tree für Codebespiel

```
File
  Name './example_faculty_it.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
        While
          Num '1',
          [
            If
              Atom
                Name 'n',
                Eq '==',
                Num '1',
                [
                  Return(Empty())
                ]
              Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
              Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
          ]
      ]
  ]
```

**Code 3.7:** PicoC Shrink Pass für Codebespiel

#### 3.3.2.2 PicoC-Blocks Pass

##### 3.3.2.2.1 Zweck

Der Zweck dieses **Passes** ist die die Container-Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von Blöcken, `GoTo(lable)`-Statements und nur noch `IF-Else`-Container-Knoten für die **Condition** umzusetzen.

##### 3.3.2.2.2 Abstrakte Syntax

| | | | |
|---|---|---|---|
| $decl\_def$ | ::= | $FunDef(\langle datatype\rangle, Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle block\rangle*)$ | *L_Fun* |
| $block$ | ::= | $Block(Name(str), \langle stmt\rangle*)$ | *L_Blocks* |
| $stmt$ | ::= | $GoTo(Name(str)) \mid NewStackframe(Name(), GoTo(str))$ | |
| | $\mid$ | $RemoveStackframe() \mid SetScope(Name(str))$ | |
| | $\mid$ | $SingleLineComment(str, str)$ | |

**Grammar 3.3.1:** Abstrakte Syntax für $L_{PicoC\_Blocks}$

##### 3.3.2.2.3 Codebeispiel

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Block
          Name 'main.5',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.4'))
          ],
        Block
          Name 'condition_check.4',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.3'))
              ],
              [
                GoTo(Name('while_after.0'))
              ]
          ],
        Block
          Name 'while_branch.3',
          [
            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
            IfElse
              Atom
                Name 'n',
                Eq '==',
                Num '1',
              [
                GoTo(Name('if.2'))
              ],
              [
                GoTo(Name('if_else_after.1'))
              ]
          ],
        Block
          Name 'if.2',
          [
            Return(Empty())
          ],
        Block
          Name 'if_else_after.1',
          [
            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
            GoTo(Name('condition_check.4'))
          ],
        Block
```

```
58          Name 'while_after.0',
59          []
60      ]
61  ]
```

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

#### 3.3.2.3 PicoC-Mon Pass

##### 3.3.2.3.1 Zweck

##### 3.3.2.3.2 Abstrakte Syntax

| | | | |
|---|---|---|---|
| $ref\_loc$ | ::= | $Stack(Num(str)) \mid Global(Num(str))$ | *L_Assign_Alloc* |
| | $\mid$ | $Stackframe(Num(str))$ | |
| $error\_data$ | ::= | $\langle exp \rangle \mid Pos(Num(str), Num(str))$ | |
| $exp$ | ::= | $Stack(Num(str)) \mid Ref(\langle ref_loc \rangle, \langle datatype \rangle, \langle error_data \rangle)$ | |
| $stmt$ | ::= | $Exp(\langle exp \rangle)$ | |
| | $\mid$ | $Assign(Alloc(Writeable(), StructSpec(Name(str)), Name(str)),$ $Struct(Assign(Name(str), \langle exp \rangle)+, \langle datatype \rangle ))$ | |
| | $\mid$ | $Assign(Alloc(Writeable(), ArrayDecl(Num(str)+, \langle datatype \rangle)),$ $Name(str)), Array(\langle exp \rangle+, \langle datatype \rangle))$ | |
| $symbol\_table$ | ::= | $SymbolTable(\langle symbol \rangle)$ | *L_Symbol_Table* |
| $symbol$ | ::= | $Symbol(\langle type_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)$ | |
| $type\_qual$ | ::= | $Empty()$ | |
| $datatype$ | ::= | $BuiltIn() \mid SelfDefined()$ | |
| $name$ | ::= | $Name(str)$ | |
| $val$ | ::= | $Num(str) \mid Empty()$ | |
| $pos$ | ::= | $Pos(Num(str), Num(str)) \mid Empty()$ | |
| $size$ | ::= | $Num(str) \mid Empty()$ | |

**Grammar 3.3.2:** Abstrakte Syntax für $L_{PicoC\_Mon}$

**Definition 3.5: Symboltabelle**

##### 3.3.2.3.3 Codebeispiel

```
1  File
2    Name './example_faculty_it.picoc_mon',
3    [
4      Block
5        Name 'main.5',
6        [
7          // Assign(Name('n'), Num('4'))
8          Exp(Num('4'))
9          Assign(Global(Num('0')), Stack(Num('1')))
10         // Assign(Name('res'), Num('1'))
11         Exp(Num('1'))
12         Assign(Global(Num('1')), Stack(Num('1')))
```

```
13        // While(Num('1'), [])
14        Exp(GoTo(Name('condition_check.4')))
15      ],
16    Block
17      Name 'condition_check.4',
18      [
19        // IfElse(Num('1'), [], [])
20        Exp(Num('1')),
21        IfElse
22          Stack
23            Num '1',
24          [
25            GoTo(Name('while_branch.3'))
26          ],
27          [
28            GoTo(Name('while_after.0'))
29          ]
30      ],
31    Block
32      Name 'while_branch.3',
33      [
34        // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35        // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36        Exp(Global(Num('0')))
37        Exp(Num('1'))
38        Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
39        IfElse
40          Stack
41            Num '1',
42          [
43            GoTo(Name('if.2'))
44          ],
45          [
46            GoTo(Name('if_else_after.1'))
47          ]
48      ],
49    Block
50      Name 'if.2',
51      [
52        Return(Empty())
53      ],
54    Block
55      Name 'if_else_after.1',
56      [
57        // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
58        Exp(Global(Num('0')))
59        Exp(Global(Num('1')))
60        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
61        Assign(Global(Num('1')), Stack(Num('1')))
62        // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
63        Exp(Global(Num('0')))
64        Exp(Num('1'))
65        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
66        Assign(Global(Num('0')), Stack(Num('1')))
67        Exp(GoTo(Name('condition_check.4')))
68      ],
69    Block
```

```
70      Name 'while_after.0',
71      [
72        Return(Empty())
73      ]
74  ]
```

**Code 3.9:** PicoC-Mon Pass für Codebespiel

#### 3.3.2.4 RETI-Blocks Pass

##### 3.3.2.4.1 Zweck

##### 3.3.2.4.2 Abstrakte Syntax

| | | | |
|---|---|---|---|
| $program$ | $::=$ | $Program(Name(str), \langle block \rangle *)$ | *L_Program* |
| $exp\_stmts$ | $::=$ | $GoTo(str)$ | *L_Blocks* |
| $instrs\_before$ | $::=$ | $Num(str)$ | |
| $num\_instrs$ | $::=$ | $Num(str)$ | |
| $block$ | $::=$ | $Block(Name(str), \langle instr \rangle *, \langle instrs\_before \rangle, \langle num\_instrs \rangle)$ | |
| $instr$ | $::=$ | $GoTo(Name(str))$ | |

**Grammar 3.3.3:** Abstrakte Syntax für $L_{RETI\_Blocks}$

##### 3.3.2.4.3 Codebeispiel

```
1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         # // Assign(Name('n'), Num('4'))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
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('res'), Num('1'))
17        # Exp(Num('1'))
18        SUBI SP 1;
19        LOADI ACC 1;
20        STOREIN SP ACC 1;
21        # Assign(Global(Num('1')), Stack(Num('1')))
22        LOADIN SP ACC 1;
23        STOREIN DS ACC 1;
24        ADDI SP 1;
25        # // While(Num('1'), [])
26        # Exp(GoTo(Name('condition_check.4')))
27        Exp(GoTo(Name('condition_check.4')))
```

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

```
85        # Exp(Global(Num('1')))
86        SUBI SP 1;
87        LOADIN DS ACC 1;
88        STOREIN SP ACC 1;
89        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
90        LOADIN SP ACC 2;
91        LOADIN SP IN2 1;
92        MULT ACC IN2;
93        STOREIN SP ACC 2;
94        ADDI SP 1;
95        # Assign(Global(Num('1')), Stack(Num('1')))
96        LOADIN SP ACC 1;
97        STOREIN DS ACC 1;
98        ADDI SP 1;
99        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
100       # Exp(Global(Num('0')))
101       SUBI SP 1;
102       LOADIN DS ACC 0;
103       STOREIN SP ACC 1;
104       # Exp(Num('1'))
105       SUBI SP 1;
106       LOADI ACC 1;
107       STOREIN SP ACC 1;
108       # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
109       LOADIN SP ACC 2;
110       LOADIN SP IN2 1;
111       SUB ACC IN2;
112       STOREIN SP ACC 2;
113       ADDI SP 1;
114       # Assign(Global(Num('0')), Stack(Num('1')))
115       LOADIN SP ACC 1;
116       STOREIN DS ACC 0;
117       ADDI SP 1;
118       # Exp(GoTo(Name('condition_check.4')))
119       Exp(GoTo(Name('condition_check.4')))
120     ],
121   Block
122     Name 'while_after.0',
123     [
124       # Return(Empty())
125       LOADIN BAF PC -1;
126     ]
127 ]
```

**Code 3.10:** RETI-Blocks Pass für Codebespiel

#### 3.3.2.5 RETI-Patch Pass

##### 3.3.2.5.1 Zweck

##### 3.3.2.5.2 Abstrakte Syntax

| | | |
|---|---|---|
| $stmt$ | $::=$ | $Exit(Num(str))$ |

**Grammar 3.3.4:** Abstrakte Syntax für $L_{RETI\_Patch}$

#### 3.3.2.5.3 Codebeispiel

```
File
  Name './example_faculty_it.reti_patch',
  [
    Block
      Name 'start.6',
      [
        # // Exp(GoTo(Name('main.5')))
        # // not included Exp(GoTo(Name('main.5')))
      ],
    Block
      Name 'main.5',
      [
        # // Assign(Name('n'), Num('4'))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # // While(Num('1'), [])
        # Exp(GoTo(Name('condition_check.4')))
        # // not included Exp(GoTo(Name('condition_check.4')))
      ],
    Block
      Name 'condition_check.4',
      [
        # // IfElse(Num('1'), [], [])
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # IfElse(Stack(Num('1')), [], [])
        LOADIN SP ACC 1;
        ADDI SP 1;
        JUMP== GoTo(Name('while_after.0'));
        # // not included Exp(GoTo(Name('while_branch.3')))
      ],
    Block
      Name 'while_branch.3',
      [
        # // If(Atom(Name('n'), Eq('==')), Num('1')), [])
        # // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
        # Exp(Global(Num('0')))
        SUBI SP 1;
```

```
56        LOADIN DS ACC 0;
57        STOREIN SP ACC 1;
58        # Exp(Num('1'))
59        SUBI SP 1;
60        LOADI ACC 1;
61        STOREIN SP ACC 1;
62        LOADIN SP ACC 2;
63        LOADIN SP IN2 1;
64        SUB ACC IN2;
65        JUMP== 3;
66        LOADI ACC 0;
67        JUMP 2;
68        LOADI ACC 1;
69        STOREIN SP ACC 2;
70        ADDI SP 1;
71        # IfElse(Stack(Num('1')), [], [])
72        LOADIN SP ACC 1;
73        ADDI SP 1;
74        JUMP== GoTo(Name('if_else_after.1'));
75        # // not included Exp(GoTo(Name('if.2')))
76      ],
77    Block
78      Name 'if.2',
79      [
80        # Return(Empty())
81        LOADIN BAF PC -1;
82      ],
83    Block
84      Name 'if_else_after.1',
85      [
86        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
87        # Exp(Global(Num('0')))
88        SUBI SP 1;
89        LOADIN DS ACC 0;
90        STOREIN SP ACC 1;
91        # Exp(Global(Num('1')))
92        SUBI SP 1;
93        LOADIN DS ACC 1;
94        STOREIN SP ACC 1;
95        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
96        LOADIN SP ACC 2;
97        LOADIN SP IN2 1;
98        MULT ACC IN2;
99        STOREIN SP ACC 2;
100       ADDI SP 1;
101       # Assign(Global(Num('1')), Stack(Num('1')))
102       LOADIN SP ACC 1;
103       STOREIN DS ACC 1;
104       ADDI SP 1;
105       # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
106       # Exp(Global(Num('0')))
107       SUBI SP 1;
108       LOADIN DS ACC 0;
109       STOREIN SP ACC 1;
110       # Exp(Num('1'))
111       SUBI SP 1;
112       LOADI ACC 1;
```

```
113         STOREIN SP ACC 1;
114         # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
115         LOADIN SP ACC 2;
116         LOADIN SP IN2 1;
117         SUB ACC IN2;
118         STOREIN SP ACC 2;
119         ADDI SP 1;
120         # Assign(Global(Num('0')), Stack(Num('1')))
121         LOADIN SP ACC 1;
122         STOREIN DS ACC 0;
123         ADDI SP 1;
124         # Exp(GoTo(Name('condition_check.4')))
125         Exp(GoTo(Name('condition_check.4')))
126       ],
127     Block
128       Name 'while_after.0',
129       [
130         # Return(Empty())
131         LOADIN BAF PC -1;
132       ]
133   ]
```

**Code 3.11:** RETI-Patch Pass für Codebespiel

#### 3.3.2.6 RETI Pass

##### 3.3.2.6.1 Zweck

##### 3.3.2.6.2 Konkrette und Abstrakte Syntax

| | | | |
|---|---|---|---|
| *dig_no_0* | ::= | "1" \| "2" \| "3" \| "4" \| "5" \| "6" \| "7" \| "8" \| "9" | *L_Program* |
| *dig_with_0* | ::= | "0" \| *dig_no_0* | |
| *num* | ::= | "0" \| *dig_no_0dig_with_0* $*$ \| "$-$"*dig_with_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*" | |

**Grammar 3.3.5:** Konkrette Syntax für $L_{RETI\_Lex}$

| | | | |
|---|---|---|---|
| *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";")∗* | |

**Grammar 3.3.6:** Konkrette Syntax für $L_{RETI\_Parse}$

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

**Grammar 3.3.7:** Abstrakte Syntax für $L_{RETI}$

#### 3.3.2.6.3 Codebeispiel

```
1  # // Exp(GoTo(Name('main.5')))
2  # // not included Exp(GoTo(Name('main.5')))
3  # // Assign(Name('n'), Num('4'))
4  # Exp(Num('4'))
5  SUBI SP 1;
6  LOADI ACC 4;
7  STOREIN SP ACC 1;
8  # Assign(Global(Num('0')), Stack(Num('1')))
9  LOADIN SP ACC 1;
10 STOREIN DS ACC 0;
11 ADDI SP 1;
12 # // Assign(Name('res'), Num('1'))
13 # Exp(Num('1'))
14 SUBI SP 1;
15 LOADI ACC 1;
```

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

```
73 STOREIN SP ACC 2;
74 ADDI SP 1;
75 # Assign(Global(Num('1')), Stack(Num('1')))
76 LOADIN SP ACC 1;
77 STOREIN DS ACC 1;
78 ADDI SP 1;
79 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
80 # Exp(Global(Num('0')))
81 SUBI SP 1;
82 LOADIN DS ACC 0;
83 STOREIN SP ACC 1;
84 # Exp(Num('1'))
85 SUBI SP 1;
86 LOADI ACC 1;
87 STOREIN SP ACC 1;
88 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
89 LOADIN SP ACC 2;
90 LOADIN SP IN2 1;
91 SUB ACC IN2;
92 STOREIN SP ACC 2;
93 ADDI SP 1;
94 # Assign(Global(Num('0')), Stack(Num('1')))
95 LOADIN SP ACC 1;
96 STOREIN DS ACC 0;
97 ADDI SP 1;
98 # Exp(GoTo(Name('condition_check.4')))
99 JUMP -53;
100 # Return(Empty())
101 LOADIN BAF PC -1;
```

**Code 3.12:** RETI Pass für Codebespiel

### 3.3.3 Umsetzung von Pointern

#### 3.3.3.1 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 Pointer Referenzierung

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstract Syntax Tree** in Code 3.14 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Pointer `*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:** Abstract Syntax Tree für Pointer Referenzierung

Bevor man einem **Pointer** eine eine **Adresse** (z.B. `&var`) zuweisen kann, muss dieser erstmal **definiert** sein. Dafür braucht es einen Eintrag in der **Symboltabelle** in Code 3.15.

> Die **Größe** eines Pointers (z.B. eines Pointers auf ein Array 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 Pointer Referenzierung

Im **PicoC-Mon 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 3.3.6.3 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-Mon Pass für Pointer Referenzierung

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 Pointer Referenzierung

#### 3.3.3.2 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 Pointer Dereferenzierung

Der Container-Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstract Syntax Tree** in Code 3.19 eine **Dereferenzierung** `*var`. Es gibt herbei **zwei** Fälle. Bei der Anwendung von **Pointer Arithmetik**, 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:** Abstract Syntax Tree für Pointer Dereferenzierung

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[5]. 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 Pointer Dereferenzierung

## 3.3.4 Umsetzung von Arrays

### 3.3.4.1 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.

[5] 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.

```
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 **Arrays** `int ar[2][1] = {{3+1}, {4}}` wird im **Abstract Syntax Tree** 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:** Abstract Syntax Tree 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[6]. 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 3.3.4.2 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**

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

**Statischen Daten**[7].

Das **Größe** des Arrays `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_k]$, 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**: $\texttt{size}(\texttt{datatype}(\texttt{ar})) = \left(\prod_{j=1}^{n} \texttt{dim}_j\right) \cdot \texttt{size}(\texttt{datatype})$[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

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

**Code 3.23:** Symboltabelle für Array Initialisierung

[7] 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.

Im **PiocC-Mon Pass** in Code 3.24 werden zuerst die **Logischen Ausdrücke** in den Blättern des Teilbaums, der beim **Array-Initializers Container-Knoten** `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` anfängt nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben[8].

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 3.7 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-Mon 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-Mon Pass für Array Initialisierung

Im **RETI-Blocks Pass** in Code 3.25 werden die **Kompositionen** `Exp(exp)` und `Assign(Global(Num('0'))`,

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

`Stack(Num('2'))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
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'))
```

```
52          SUBI SP 1;
53          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

#### 3.3.4.2 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 **Abstract Syntax Tree** in Code 3.27 mithilfe des **Container-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:** Abstract Syntax Tree für Zugriff auf einen Arrayindex

Im **PicoC-Mon Pass** in Code 3.28 wird vom **Container-Knoten** `Subscr(Name('ar'), Num('0'))` zuerst im **Anfangsteil** 3.3.6.2 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** 3.3.6.3 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 3.7 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 **Container-Knoten** `Ref(exp, datatype)` als versteckes Attribut `datatype` angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon Pass** dem **Container-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 3.3.6.3 allgemein erklärt ist.

Im letzten Schritt, dem **Schlussteil** 3.3.6.4 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 3.3.6.4). Der **Unterdatentyp** ist dabei ein versteckes 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-Instructions** für eine Variable, die in den **Globalen Statischen Daten** liegt und eine Variable, die auf dem **Stackframe** liegt erzeugt werden müssen.

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays $\texttt{datatype ar}[\texttt{dim}_1]\dots[\texttt{dim}_n]$ 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[a] berechnet werden[b].

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] C. Scholl, „Betriebssysteme“.
[b] `ref(exp)` steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte.

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

```
26          Ref(Stackframe(Num('2')))
27          Exp(Num('1'))
28          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-Mon 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

#### 3.3.4.3 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[2] = 42;
4 }
```

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

Im **Abstract Syntax Tree** 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('2')), Num('42'))
11       ]
12   ]
```

**Code 3.31:** Abstract Syntax Tree für Zuweisung an Arrayindex

Im **PicoC-Mon Pass** in Code 3.32 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=`, bzw. des **Container-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 3.3.4.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird berechet, wie in Subkapitel 3.3.4.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`[9] die Zuweisung `=` des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Arrayindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

[9] Ist in Tabelle 3.7 genauer beschrieben ist

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

**Code 3.32:** PicoC-Mon 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('2')), 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('2'))
        SUBI SP 1;
        LOADI ACC 2;
        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;
```

```
31          ADDI SP 2;
32          STOREIN IN1 ACC 0;
33          # Return(Empty())
34          LOADIN BAF PC -1;
35      ]
36  ]
```

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

### 3.3.5 Umsetzung von Structs

#### 3.3.5.1 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 **Abstract Syntax Tree** 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:** Abstract Syntax Tree 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**[10] und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

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

[a] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache $L_{PicoC}$ nicht die Fragwürdige Designentscheidung, auch die eckigen Klammern `[]` für die Definition eines 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'))
        size:                   Num('3')
```

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

```
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

#### 3.3.5.2 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 **Abstract Syntax Tree** 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       ],
```

```
    StructDecl
      Name 'st2',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
        ↪  Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
        ↪  Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')),
        ↪  Ref(Name('var'))])])])])])
      ]
  ]
```

**Code 3.38:** Abstract Syntax Tree für Initialisierung von Structs

Im **PicoC-Mon 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 3.3.4.1 daher wird um keine Wiederholung zu betreiben auf Subkapitel 3.3.4.1 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 Container-Knoten** anfängt, wird auf dieselbe Weise nach dem **Depth-First-Search** Prinzip von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Arrays** in Subkapitel 3.3.4.1 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 **Depth-First-Search** Schema von **links-nach-rechts** iteriert und die Ergebnisse der **Logischen Ausdrücken** in den **Blättern** auf den **Stack** gespeichert. Der Fall, dass ein **Logischer Ausdruck** vorliegt erübrigt sich damit.

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
```

```
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([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-Mon 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([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

#### 3.3.5.3 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 **Abstract Syntax Tree** 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:** Abstract Syntax Tree für Zugriff auf Structattribut

Im **PicoC-Mon 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 3.3.4.2 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel 3.3.4.2 verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Subkapitel 3.3.4.2 durch Kompositionen ersetzt, die sich in **Anfangsteil** 3.3.6.2, **Mittelteil** 3.3.6.3 und **Schlussteil** 3.3.6.4 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 3.3.4.2.

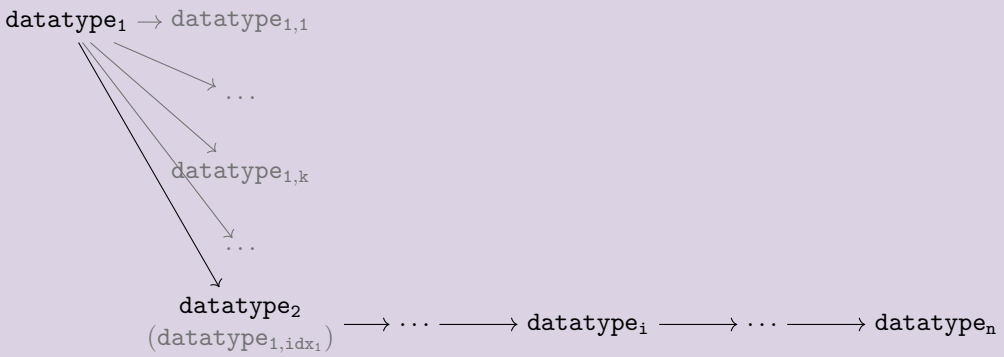
Nur für den **Mittelteil** wird eine andere Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` gebraucht. Diese Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` erfüllt die Aufgabe die **Adresse**, ab der das **Attribut** auf das zugegriffen wird anfängt zu berechnen. Dabei wurde die **Anfangsadresse** 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 3.3.4.2, 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 **Container-Knoten** `Ref(exp, datatype)` angehängt.

Sei $\texttt{datatype}_\texttt{i}$ ein **Knoten** eines **entarteten Baumes** (siehe Definition 3.6 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 $\texttt{ref}(\texttt{datatype}_\texttt{i}) \ \ldots \ \texttt{ref}(\texttt{datatype}_\texttt{i}) + \texttt{size}(\texttt{datatype}_\texttt{i})$ 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 **Pointerelemente**, **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen

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

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

berechnet werden.[c]

Dabei darf nur der letzte Knoten $\texttt{datatype}_\texttt{n}$ vom Datentyp **Pointer** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, 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 Pointer geht $\texttt{datatype}_\texttt{pntr}$ und die **zweite Adressberechnung** einen Dantentyp nach dem Datentyp Pointer 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 **Pointerelemente**, **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 $\sum_{k=1}^{idx_i-1} \text{size}(\texttt{datatype}_{\texttt{i},\texttt{k}})$ in der Formel.

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

[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 **Pointerelmente**, **Arrayelmente** 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 3.6: 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*.

```
1 File
2   Name './example_struct_attr_access.picoc_mon',
```

```
3  [
4    Block
5      Name 'main.0',
6      [
7        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
         ↪  Num('2'))]))
8        Exp(Num('4'))
9        Exp(Num('2'))
10       Assign(Global(Num('0')), Stack(Num('2')))
11       // Exp(Attr(Name('st'), Name('y')))
12       Ref(Global(Num('0')))
13       Ref(Attr(Stack(Num('1')), Name('y')))
14       Exp(Stack(Num('1')))
15       Return(Empty())
16     ]
17 ]
```

**Code 3.43:** PicoC-Mon 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.

```
1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Exp(Attr(Name('st'), Name('y')))
23        # Ref(Global(Num('0')))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        STOREIN SP IN1 1;
28        # Ref(Attr(Stack(Num('1')), Name('y')))
29        LOADIN SP IN1 1;
30        ADDI IN1 1;
```

```
31          STOREIN SP IN1 1;
32          # Exp(Stack(Num('1')))
33          LOADIN SP IN1 1;
34          LOADIN IN1 ACC 0;
35          STOREIN SP ACC 1;
36          # Return(Empty())
37          LOADIN BAF PC -1;
38        ]
39    ]
```

**Code 3.44:** RETI-Blocks Pass für Zugriff auf Structattribut

#### 3.3.5.4 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:** Abstract Syntax Tree für Zuweisung an Structattribut

Im **PicoC-Mon 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 3.3.4.3 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel 3.3.4.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 3.3.4.3 muss hier für das Auswerten des **linken** Container-Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Subkapitel 3.3.5.3 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-Mon 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;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # 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

### 3.3.6 Umsetzung des Zugriffs auf Derived datatypes im Allgemeinen

#### 3.3.6.1 Übersicht

In den Unterkapiteln 3.3.3, 3.3.4 und 3.3.5 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln beschriebenen Datentypen (**Pointer**, **Array** und **Struct**) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem **Anfangsteil**, **Mittelteil** und **Schlussteil** darin erkennen.

Dieses Vorgehen ist in Abbildung 3.4 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der Datenypen **Pointer**, **Array** und **Struct** 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)`-Container-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 **Pointerelmements**, **Arrayelements** oder **Structattributs**.

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

Die **Zugriffsoperation** muss dabei zum **Datentyp** im versteckten Attribut `datatype` passen, ansonsten gibt

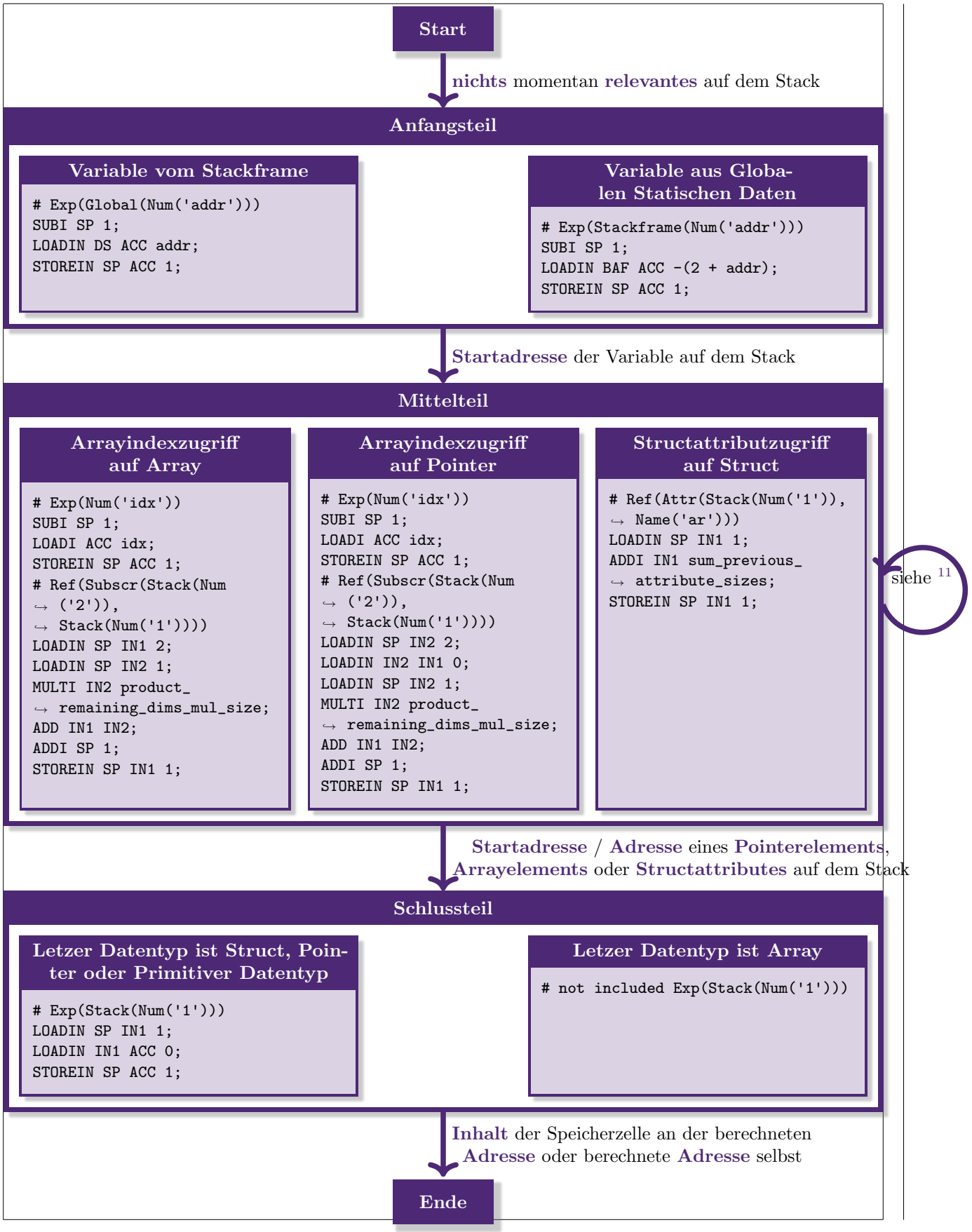



**Abbildung 3.4:** Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

es eine DatatypeMismatch-**Fehlermeldung**. Ein **Zugriff auf ein Arrayindex** `Subscr(exp1, epp2)` kann dabei mit den Datentypen **Array** `ArrayDecl(nums, datatype)` und **Pointer** `PntrDecl(num, datatype)` kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Pointer** `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.

> 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 3.3.3 bereits erklärt wurde, wurde der **Container-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 **Pointerelement**, **Arrayelement** oder **Structattribut** 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 **Container-Knoten** oder **Token-Knoten** in diesem **Unterdatentyp** interessiert und die möglicherweise unter diesem momentan betrachteten **Knoten** liegenden **Container-Knoten** und **Token-Knoten** in einem anderen `Ref(exp, versteckte Attribut)`-Container-Knoten dem versteckte Attribut 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[12], 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.

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

[12] Und nicht auf ein **Element** des Arrays.
[12] **Startadresse** / **Adresse** eines **Pointerelements**, **Arrayelements** oder **Structattributes** auf dem Stack.

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 **Container-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** 3.3.6.2, **Mittelteil** 3.3.6.3 und **Schlussteil** 3.3.6.4 bei der Kompilierung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente**, **Structattribute** bei gemischten Ausdrücken, wie `(*st_first.ar)[0];` einzeln isoliert betrachtet und erläutert.

#### 3.3.6.2 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 **Abstract Syntax Tree** 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:** Abstract Syntax Tree für den Anfangsteil

Im **PicoC-Mon 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')))
        Return(Empty())
      ],
    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-Mon 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

#### 3.3.6.3 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinandereihung von Zugriffen auf **Pointerelemente**, **Arrayelemente** oder **Structattribute** 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 **Abstract Syntax Tree** in Code 3.54 wird die Aneinandererihung von Zugriffen auf **Pointerelemente**, **Arrayelemente** und **Structattribute** `(*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:** Abstract Syntax Tree für den Mittelteil

Im **PicoC-Mon 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 Container-Knoten einfach dem `exp` Attribut des `Ref(exp)`-Container 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-Mon 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)`-Container-Knoten berücksichtigt werden, was in Unterkapitel 3.3.6.1 zusammen mit der Abbildung 3.4 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

#### 3.3.6.4 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 3.3.6.2 und **Mittelteil** 3.3.6.3 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird[13], 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 **Abstract Syntax Tree** 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'))
```

[13] 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:** Abstract Syntax Tree für den Schlussteil

Im **PicoC-Mon 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-Mon 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)`-Container-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 3.3.6.1 zusammen mit der Abbildung 3.4 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

### 3.3.7 Umsetzung von Funktionen

#### 3.3.7.1 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 **Abstract Syntax Tree** 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 Container-Knoten sind in Tabelle 3.5 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:** Abstract Syntax Tree 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)`[14] 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>`[15].

Es werden parallel dazu neue Zuordnungen im **Dictionary** `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[16]. 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'))
          ]
      ]
  ]
```

[14] Eine Erklärung dazu ist in Unterkapitel 3.3.2.2.1 zu finden.
[15] Der **Grund** dafür kann im Unterkapitel 3.3.2.2.1 nachgelesen werden.
[16] Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

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

Im **PicoC-Mon Pass** in Code 3.64 werden die `FunDef(datatype, name, allocs, stmts)`-Container-Knoten komplett aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Container-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-Mon Pass für 3 Funktionen

Nach dem **RETI Pass** in Code 3.65 gibt es nur noch **RETI-Instructions**, die Blöcke wurden entfernt und die **RETI-Instructions** 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 der **Instruction** 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-Instructions**, 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

#### 3.3.7.1.1 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',
```

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

**Code 3.67:** 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)`-Container-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>'))`-Container-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;
```

```
      ],
    Block
      Name 'fun2.1',
      [
        # // 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;
      ],
    Block
      Name 'main.0',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

**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.

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.0')))
JUMP 8;
# // Block(Name('fun1.2'), [])
# Return(Empty())
LOADIN BAF PC -1;
# // Block(Name('fun2.1'), [])
# // 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;
# // Block(Name('main.0'), [])
# Return(Empty())
LOADIN BAF PC -1;
```

**Code 3.69:** RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

#### 3.3.7.2 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 **Fehler-**

**meldungen** ausgegeben werden können, wenn der **Prototyp** (Definition 3.7) 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.

**Definition 3.7: 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?*

```
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 3.8). 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**[17] Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** ein **Funktionswechsel** eintritt und

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

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 **Dictionary** 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** 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 **Dictionary 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.

**Definition 3.8: Scope (bzw. Sichtbarkeitsbereich)**

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

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

```
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

#### 3.3.7.3 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 3.3.7.3.1 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 **Abstract Syntax Tree** in Code 3.73 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

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

**Code 3.73:** Abstract Syntax Tree für Funktionsaufruf ohne Rückgabewert

Im **PicoC-Mon 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 3.7 und 3.2 genauer erklärt sind.

Der Container-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 Container-Knoten `StackMalloc(Num('2'))` umgesetzt,

indem das SP-Register einfach um zwei Speicherzellen **dekrementiert** wird und somit Speicher auf dem **Stack** belegt wird[18].

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

**Code 3.74:** PicoC-Mon 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-Instructions** 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 **Dictionary** `fun_name_to_block_name`[19] mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Container-Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

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

[18] Wobei hier "**reserviert**"besser passen würde.
[19] Dieses Dictionary wurde in Unterkapitel 3.3.7.1 eingeführt.

```
14        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        MOVE BAF ACC;
16        ADDI SP 3;
17        MOVE SP BAF;
18        SUBI SP 4;
19        STOREIN BAF ACC 0;
20        LOADI ACC GoTo(Name('addr@next_instr'));
21        ADD ACC CS;
22        STOREIN BAF ACC -1;
23        # Exp(GoTo(Name('stack_fun.0')))
24        Exp(GoTo(Name('stack_fun.0')))
25        # RemoveStackframe()
26        MOVE BAF IN1;
27        LOADIN IN1 BAF 0;
28        MOVE IN1 SP;
29        # Return(Empty())
30        LOADIN BAF PC -1;
31      ],
32    Block
33      Name 'stack_fun.0',
34      [
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ]
38  ]
```

**Code 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[20] ersetzt: `LOADI ACC 14`. Und auch der **Container-Knoten**, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den **Container-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.

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:

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

- 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>` 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:

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

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 verstecktes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und $idx$ haben die **gleiche Bedeutung** wie in der Formel 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

#### 3.3.7.3.1 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 **Abstract Syntax Tree** 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:** Abstract Syntax Tree für Funktionsaufruf mit Rückgabewert

Im **PicoC-Mon 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 3.7 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-Mon Pass** eines hinzufügt wird in Form der `Komposition Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine `MissingReturn`-**Fehlermeldung** ausgelöst.

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

**Code 3.79:** PicoC-Mon 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.

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        # Exp(Num('21'))
```

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

```
66          MOVE SP BAF;
67          SUBI SP 2;
68          STOREIN BAF ACC 0;
69          LOADI ACC GoTo(Name('addr@next_instr'));
70          ADD ACC CS;
71          STOREIN BAF ACC -1;
72          # Exp(GoTo(Name('fun_no_return_value.1')))
73          Exp(GoTo(Name('fun_no_return_value.1')))
74          # RemoveStackframe()
75          MOVE BAF IN1;
76          LOADIN IN1 BAF 0;
77          MOVE IN1 SP;
78          # Return(Empty())
79          LOADIN BAF PC -1;
80        ]
81    ]
```

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

#### 3.3.7.3.2 Umsetzung von Call by Sharing für Arrays

Die **Call by Reference** (Definition 1.9) 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-Mon Pass** wird im Fall dessen, dass der **oberste Container-Knoten** im Teilbaum, der den Datentyp darstellt und an die Funktion übergeben wird ein **Array** `ArrayDecl(nums, datatype)` ist, dieser zu einem **Pointer** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, an das `datatype`-Attribut des **Pointers** `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 **Pointer** nur eine **Speicherzelle** braucht.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
        ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('param'))])
        name:                   Name('fun_array_from_stackframe')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param@fun_array_from_stackframe')
        value or address:       Num('0')
        position:               Pos(Num('1'), Num('37'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
        ↪  [Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
        name:                   Name('fun_array_from_global_data')
        value or address:       Empty()
        position:               Pos(Num('4'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param@fun_array_from_global_data')
        value or address:       Num('0')
        position:               Pos(Num('4'), Num('36'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
        name:                   Name('local_var@fun_array_from_global_data')
        value or address:       Num('6')
        position:               Pos(Num('5'), Num('6'))
        size:                   Num('6')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        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-Mon 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 **Container-Knoten** `Global(num)` bzw. `Stackframe(num)` die **relative Adressen** relativ zum `DS`-Register bzw. `SP`-Register dar, die aus der **Symboltabelle** entnommen sind.

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

**Code 3.83:** PicoC-Mon 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')))
        SUBI SP 1;
        MOVE BAF IN1;
        SUBI IN1 8;
        STOREIN SP IN1 1;
        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 3;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('fun_array_from_stackframe.2')))
        Exp(GoTo(Name('fun_array_from_stackframe.2')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
```

```
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

#### 3.3.7.3.3 Umsetzung von Call by Value für Structs

Die **Call by Value** (Definition 1.8) 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-Mon 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-Mon 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-Mon 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

### 3.4.1 Error Handler

### 3.4.2 Arten von Fehlermeldungen

#### 3.4.2.1 Syntaxfehler

#### 3.4.2.2 Laufzeitfehler

# 4 Ergebnisse und Ausblick

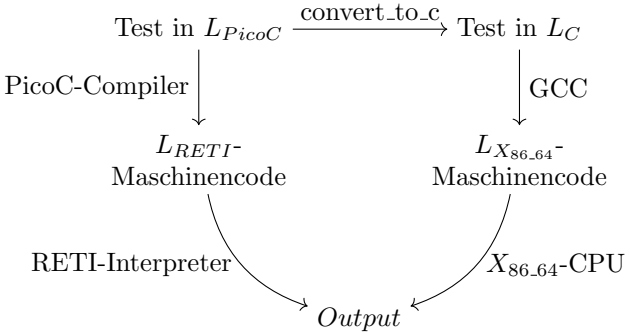
## 4.1 Compiler

### 4.1.1 Überblick über Funktionen

### 4.1.2 Vergleich mit GCC

### 4.1.3 Showmode

## 4.2 Qualitätssicherung




(4.0.1)

## 4.3 Erweiterungsideen

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 4.1) 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 4.1: 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] 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 4.4) 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**[1] 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 4.3) 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.[2]

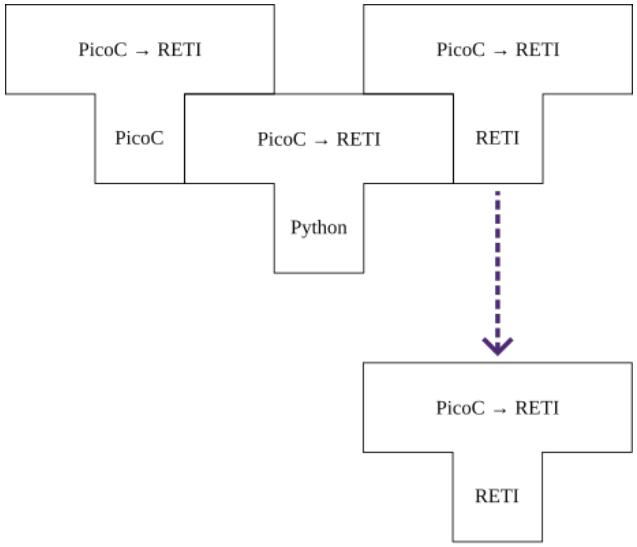



**Abbildung 4.1:** Cross-Compiler als Bootstrap Compiler

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** $C^o_w$ 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** $C^{Python}_{PicoC}$.

[1] 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.

[2] 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.

**Definition 4.2: 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 4.3: 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** $C_{RETI\_PicoC}^{PicoC}$, der einen **minimalen Compiler** (Definition 4.2) 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.[3]

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine $M_{RETI}$ selbst, aufbauend auf diesem **minimalen Compiler** $C_{RETI\_PicoC}^{PicoC}$, wie in Subdefinition 4.4.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** $C_C$ weiterentwickelt.

[3] Natürlich könnte man aber auch einfach den **Cross-Compiler** $C_{PicoC}^{Python}$ um weitere Funktionalitäten von $L_C$ erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache $L_{Python}$.

**Definition 4.4: 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 4.4.1 nochmal gesondert hervorzuheben:*

**4.4.1:** *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] Earley und Sturgis, „A formalism for translator interactions“.

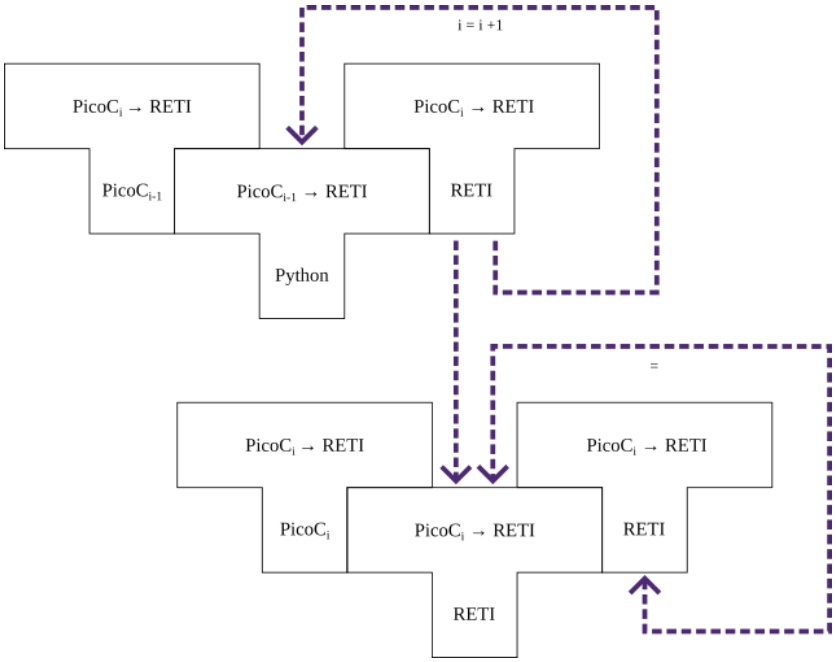



**Abbildung 4.2:** Iteratives Bootstrapping

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 4.4.1 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 4.4.1 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.

# A Appendix

## A.1 Konkrette und Abstrakte Syntax

## A.2 Bedienungsanleitungen

### A.2.1 PicoC-Compiler

### A.2.2 Showmode

### A.2.3 Entwicklertools

# Literatur

## Online


- *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).
- *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).
- *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).
- *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).

## Artikel


- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: 10.1145/355598.362740.


## Vorlesungen


- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Nebel, 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, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19. 07. 2022).


## Sonstige Quellen


- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).