

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

*Abgabedatum:* 28<sup>th</sup> April 2022

*Author:*  
Jürgen Mattheis

*Gutachter:*  
Prof. Dr. Scholl

*Betreuung:*  
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

<b>1</b>	<b>Motivation</b>	<b>12</b>
1.1	RETI	12
1.2	PicoC	12
1.3	Aufgabenstellung	12
1.4	Eigenheiten der Sprache C	12
1.5	Richtlinien	13
<b>2</b>	<b>Einführung</b>	<b>14</b>
2.1	Compiler und Interpreter	14
2.1.1	T-Diagramme	17
2.2	Grammatiken	19
2.3	Grundlagen	19
2.3.1	Mehrdeutige Grammatiken	21
2.3.2	Präzidenz und Assoziativität	21
2.4	Lexikalische Analyse	21
2.5	Syntaktische Analyse	24
2.6	Code Generierung	30
2.7	Fehlermeldungen	30
2.7.1	Kategorien von Fehlermeldungen	30
<b>3</b>	<b>Implementierung</b>	<b>31</b>
3.1	Lexikalische Analyse	32
3.1.1	Konkrete Syntax für Lexer erstellen	32
3.1.2	Basic Lexer	33
3.2	Syntaktische Analyse	33
3.2.1	Konkrete Syntax für Parser erstellen	33
3.2.2	Umsetzung von Präzidenz	34
3.2.3	Derivation Tree Generierung	35
3.2.3.1	Early Parser	35
3.2.3.2	Codebeispiel	35
3.2.4	Derivation Tree Vereinfachung	36
3.2.4.1	Visitor	36
3.2.4.2	Codebeispiel	36
3.2.5	Abstrakt Syntax Tree Generierung	37
3.2.5.1	PicoC Nodes	37
3.2.5.2	RETI Nodes	37
3.2.5.3	Abstrakte Syntax	37
3.2.5.4	Transformer	39
3.2.5.5	Codebeispiel	39
3.3	Code Generierung	40
3.3.1	Übersicht	40
3.3.2	Passes	41
3.3.2.1	Kompositionen mit besonderer Bedeutung	41
3.3.2.2	PicoC-Shrink Pass	41
3.3.2.2.1	Codebeispiel	41
3.3.2.3	PicoC-Blocks Pass	43

3.3.2.3.1	Abstrakte Syntax	43
3.3.2.3.2	Codebeispiel	44
3.3.2.4	PicoC-Mon Pass	45
3.3.2.4.1	Abstrakte Syntax	45
3.3.2.4.2	Codebeispiel	46
3.3.2.5	RETI-Blocks Pass	48
3.3.2.5.1	Abstrakte Syntax	48
3.3.2.5.2	Codebeispiel	48
3.3.2.6	RETI-Patch Pass	50
3.3.2.6.1	Abstrakte Syntax	50
3.3.2.6.2	Codebeispiel	50
3.3.2.7	RETI Pass	52
3.3.2.7.1	Konkrete und Abstrakte Syntax	52
3.3.2.7.2	Codebeispiel	53
3.3.3	Umsetzung von Pointern	55
3.3.3.1	Referenzierung	55
3.3.3.2	Pointer Dereferenzierung durch Zugriff auf Arrayindex ersetzen	57
3.3.4	Umsetzung von Arrays	59
3.3.4.1	Initialisierung von Arrays	59
3.3.4.2	Zugriff auf Arrayindex	61
3.3.4.3	Zuweisung an Arrayindex	63
3.3.5	Umsetzung von Structs	65
3.3.5.1	Deklaration von Structs	65
3.3.5.2	Initialisierung von Structs	66
3.3.5.3	Zugriff auf Structattribut	70
3.3.5.4	Zuweisung an Structattribut	72
3.3.6	Umsetzung der Derived Datatypes im Zusammenspiel	75
3.3.6.1	Einleitungsteil für Globale Statische Daten und Stackframe	75
3.3.6.2	Mittelteil für die verschiedenen Derived Datatypes	78
3.3.6.3	Schlusssteil für die verschiedenen Derived Datatypes	81
3.3.7	Umsetzung von Funktionen	87
3.3.7.1	Funktionen auflösen zu RETI Code	87
3.3.7.1.1	Sprung zur Main Funktion	90
3.3.7.2	Funktionsdeklaration und -definition	92
3.3.7.3	Funktionsaufruf	93
3.3.7.3.1	Ohne Rückgabewert	93
3.3.7.3.2	Mit Rückgabewert	95
3.3.7.3.3	Umsetzung von Call by Sharing für Arrays	97
3.3.7.3.4	Umsetzung von Call by Value für Structs	100
3.3.8	Umsetzung kleinerer Details	102
3.4	Fehlermeldungen	102
3.4.1	Error Handler	102
3.4.2	Arten von Fehlermeldungen	102
3.4.2.1	Syntaxfehler	102
3.4.2.2	Laufzeitfehler	102
<b>4</b>	<b>Ergebnisse und Ausblick</b>	<b>103</b>
4.1	Compiler	103
4.2	Showmode	103
4.3	Qualitätssicherung	103
4.4	Kommentierter Kompiliervorgang	103
4.5	Erweiterungsideen	103
<b>A</b>	<b>Appendix</b>	<b>107</b>

A.1 Konkrete und Abstrakte Syntax . . . . .	107
A.2 Bedienungsanleitungen . . . . .	107
A.2.1 PicoC-Compiler . . . . .	107
A.2.2 Showmode . . . . .	107
A.2.3 Entwicklertools . . . . .	107

---

---

# Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	19
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	19
2.3	Veranschaulichung der Lexikalischen Analyse . . . . .	24
2.4	Veranschaulichung der Syntaktischen Analyse . . . . .	29
3.1	Cross-Compiler Kompilervorgang ausgeschrieben . . . . .	40
3.2	Cross-Compiler Kompilervorgang Kurzform . . . . .	40
3.3	Architektur mit allen Passes ausgeschrieben . . . . .	41
4.1	Cross-Compiler als Bootstrap Compiler . . . . .	104
4.2	Iteratives Bootstrapping . . . . .	106

---

---

# Codeverzeichnis

3.1	PicoC Code für Derivation Tree Generierung . . . . .	35
3.2	Derivation Tree nach Derivation Tree Generierung . . . . .	36
3.3	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	37
3.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert . . . . .	39
3.5	PicoC Code für Codebeispiel . . . . .	41
3.6	Abstract Syntax Tree für Codebeispiel . . . . .	42
3.7	PicoC Shrink Pass für Codebeispiel . . . . .	43
3.8	PicoC-Blocks Pass für Codebeispiel . . . . .	45
3.9	PicoC-Mon Pass für Codebeispiel . . . . .	48
3.10	RETI-Blocks Pass für Codebeispiel . . . . .	50
3.11	RETI-Patch Pass für Codebeispiel . . . . .	52
3.12	RETI Pass für Codebeispiel . . . . .	54
3.13	PicoC Code für Pointer Referenzierung . . . . .	55
3.14	Abstract Syntax Tree für Pointer Referenzierung . . . . .	55
3.15	Symboltabelle für Pointer Referenzierung . . . . .	56
3.16	PicoC Mon Pass für Pointer Referenzierung . . . . .	56
3.17	RETI Blocks Pass für Pointer Referenzierung . . . . .	57
3.18	PicoC Code für Pointer Dereferenzierung . . . . .	57
3.19	Abstract Syntax Tree für Pointer Dereferenzierung . . . . .	58
3.20	PicoC Shrink Pass für Pointer Dereferenzierung . . . . .	58
3.21	PicoC Code für Array Initialisierung . . . . .	59
3.22	Abstract Syntax Tree für Array Initialisierung . . . . .	59
3.23	Symboltabelle für Array Initialisierung . . . . .	60
3.24	PicoC Mon Pass für Array Initialisierung . . . . .	60
3.25	RETI Blocks Pass für Array Initialisierung . . . . .	61
3.26	PicoC Code für Zugriff auf Arrayindex . . . . .	61
3.27	Abstract Syntax Tree für Zugriff auf Arrayindex . . . . .	62
3.28	PicoC Mon Pass für Zugriff auf Arrayindex . . . . .	62
3.29	RETI Blocks Pass für Zugriff auf Arrayindex . . . . .	63
3.30	PicoC Code für Zuweisung an Arrayindex . . . . .	63
3.31	Abstract Syntax Tree für Zuweisung an Arrayindex . . . . .	64
3.32	PicoC Mon Pass für Zuweisung an Arrayindex . . . . .	64
3.33	RETI Blocks Pass für Zuweisung an Arrayindex . . . . .	65
3.34	PicoC Code für Deklaration von Structs . . . . .	65
3.35	Symboltabelle für Deklaration von Structs . . . . .	66
3.36	PicoC Code für Initialisierung von Structs . . . . .	66
3.37	Abstract Syntax Tree für Initialisierung von Structs . . . . .	68
3.38	Symboltabelle für Initialisierung von Structs . . . . .	69
3.39	PicoC Mon Pass für Initialisierung von Structs . . . . .	69
3.40	RETI Blocks Pass für Initialisierung von Structs . . . . .	70
3.41	PicoC Code für Zugriff auf Structattribut . . . . .	70
3.42	Abstract Syntax Tree für Zugriff auf Structattribut . . . . .	71
3.43	PicoC Mon Pass für Zugriff auf Structattribut . . . . .	72
3.44	RETI Blocks Pass für Zugriff auf Structattribut . . . . .	72
3.45	PicoC Code für Zuweisung an Structattribut . . . . .	73
3.46	Abstract Syntax Tree für Zuweisung an Structattribut . . . . .	73
3.47	PicoC Mon Pass für Zuweisung an Structattribut . . . . .	74

3.48 RETI Blocks Pass für Zuweisung an Structattribut . . . . .	75
3.49 PicoC Code für den Einleitungsteil . . . . .	75
3.50 Abstract Syntax Tree für den Einleitungsteil . . . . .	77
3.51 PicoC Mon Pass für den Einleitungsteil . . . . .	77
3.52 RETI Blocks Pass für den Einleitungsteil . . . . .	78
3.53 PicoC Code für den Mittelteil . . . . .	78
3.54 Abstract Syntax Tree für den Mittelteil . . . . .	79
3.55 PicoC Mon Pass für den Mittelteil . . . . .	80
3.56 RETI Blocks Pass für den Mittelteil . . . . .	81
3.57 PicoC Code für den Schlussteil . . . . .	81
3.58 Abstract Syntax Tree für den Schlussteil . . . . .	83
3.59 PicoC Mon Pass für den Schlussteil . . . . .	85
3.60 RETI Blocks Pass für den Schlussteil . . . . .	87
3.61 PicoC Code für 3 Funktionen . . . . .	87
3.62 Abstract Syntax Tree für 3 Funktionen . . . . .	88
3.63 PicoC Blocks Pass für 3 Funktionen . . . . .	88
3.64 PicoC Mon Pass für 3 Funktionen . . . . .	89
3.65 RETI Blocks Pass für 3 Funktionen . . . . .	89
3.66 PicoC Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	90
3.67 PicoC Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	90
3.68 PicoC Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	91
3.69 PicoC Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	91
3.70 PicoC Code für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	92
3.71 Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	93
3.72 PicoC Code für Funktionsaufruf ohne Rückgabewert . . . . .	93
3.73 PicoC Mon Pass für Funktionsaufruf ohne Rückgabewert . . . . .	94
3.74 RETI Blocks Pass für Funktionsaufruf ohne Rückgabewert . . . . .	94
3.75 RETI Pass für Funktionsaufruf ohne Rückgabewert . . . . .	95
3.76 PicoC Code für Funktionsaufruf mit Rückgabewert . . . . .	95
3.77 PicoC Mon Pass für Funktionsaufruf mit Rückgabewert . . . . .	96
3.78 RETI Blocks Pass für Funktionsaufruf mit Rückgabewert . . . . .	97
3.79 RETI Pass für Funktionsaufruf mit Rückgabewert . . . . .	97
3.80 PicoC Code für Call by Sharing für Arrays . . . . .	97
3.81 PicoC Mon Pass für Call by Sharing für Arrays . . . . .	98
3.82 Symboltabelle für Call by Sharing für Arrays . . . . .	99
3.83 RETI Block Pass für Call by Sharing für Arrays . . . . .	100
3.84 PicoC Code für Call by Value für Structs . . . . .	100
3.85 PicoC Mon Pass für Call by Value für Structs . . . . .	101
3.86 RETI Block Pass für Call by Value für Structs . . . . .	102



---

---

# Tabellenverzeichnis

3.1 Präzidenzregeln von PicoC . . . . .	34
---	----

---

---

# Definitionsverzeichnis

1.1	Caller-save Register	12
1.2	Callee-save Register	12
1.3	Deklaration	12
1.4	Definition	12
1.5	Call by value	12
1.6	Call by reference	13
2.1	Interpreter	14
2.2	Compiler	14
2.3	Maschiensprache	15
2.4	Assemblersprache (bzw. engl. Assembly Language)	15
2.5	Assembler	16
2.6	Objectcode	16
2.7	Linker	16
2.8	Immediate	16
2.9	Transpiler (bzw. Source-to-source Compiler)	17
2.10	Cross-Compiler	17
2.11	T-Diagram Programm	17
2.12	T-Diagram Übersetzer (bzw. eng. Translator)	18
2.13	T-Diagram Interpreter	18
2.14	T-Diagram Maschine	18
2.15	Sprache	19
2.16	Chromsky Hierarchie	20
2.17	Grammatik	20
2.18	Reguläre Sprachen	20
2.19	Kontextfreie Sprachen	20
2.20	Ableitung	20
2.21	Links- und Rechtsableitung	20
2.22	Linksrekursive Grammatiken	20
2.23	Ableitungsbaum	21
2.24	Mehrdeutige Grammatik	21
2.25	Assoziativität	21
2.26	Präzidenz	21
2.27	Wortproblem	21
2.28	LL(k)-Grammatik	21
2.29	Pattern	22
2.30	Lexeme	22
2.31	Lexer (bzw. Scanner oder auch Tokenizer)	22
2.32	Bezeichner (bzw. Identifier)	23
2.33	Literal	24
2.34	Konkrete Syntax	25
2.35	Derivation Tree (bzw. Parse Tree)	25
2.36	Parser	25
2.37	Recognizer (bzw. Erkennen)	26
2.38	Transformer	27
2.39	Visitor	27
2.40	Abstrakte Syntax	28
2.41	Abstract Syntax Tree	28

2.42	Pass	30
2.43	Monadische Normalform	30
2.44	Fehlermeldung	30
3.1	Symboltabelle	46
4.1	Self-compiling Compiler	103
4.2	Minimaler Compiler	104
4.3	Bootstrap Compiler	105
4.4	Bootstrapping	105

---

---

# Grammatikverzeichnis

3.1.1 Konkrete Syntax des Lexers in EBNF . . . . .	32
3.2.1 Konkrete Syntax des Parsers in EBNF, Teil 1 . . . . .	33
3.2.2 Konkrete Syntax des Parsers in EBNF, Teil 2 . . . . .	34
3.2.3 Abstrakte Syntax für $L_{PioC}$ . . . . .	38
3.3.1 Abstrakte Syntax für $L_{PicoC\_Blocks}$ . . . . .	43
3.3.2 Abstrakte Syntax für $L_{PicoC\_Mon}$ . . . . .	45
3.3.3 Abstrakte Syntax für $L_{RETI\_Blocks}$ . . . . .	48
3.3.4 Abstrakte Syntax für $L_{RETI\_Patch}$ . . . . .	50
3.3.5 Konkrete Syntax für $L_{RETI\_Lex}$ . . . . .	52
3.3.6 Konkrete Syntax für $L_{RETI\_Parse}$ . . . . .	53
3.3.7 Abstrakte Syntax für $L_{RETI}$ . . . . .	53

---

---

# 1 Motivation

## 1.1 RETI

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

### Definition 1.1: Caller-save Register

*a*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

### Definition 1.2: Callee-save Register

*a*

<sup>a</sup>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*

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

### Definition 1.4: Definition

*a*

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

### Definition 1.5: Call by value

*a*

<sup>a</sup>Bast, „Programmieren in C“.

**Definition 1.6: Call by reference** $a$ 

---

<sup>a</sup>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**<sup>1</sup> und von **Tests** die **Beziehungen** in 2.2.1 zu belegen (siehe Subkapitel 4.3).

### Definition 2.1: Interpreter

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

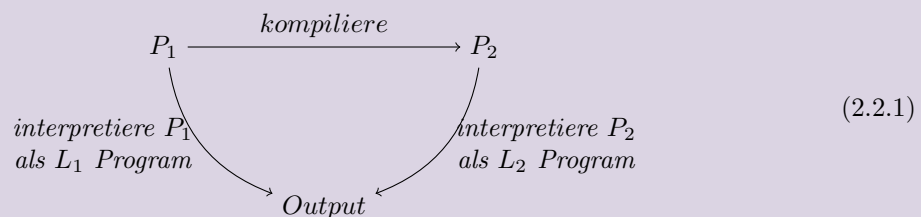
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstract Syntax Tree** (Definition 2.41) und führt je nach Komposition der **Nodes** des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.<sup>a</sup>*

<sup>a</sup>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.<sup>a</sup>*



<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

<sup>1</sup>Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **GNU Compiler Collection**

Im Folgenden wird ein voll ausgeschriebener **Compiler** als  $C_{i.w.k.min}^{o-j}$  geschrieben, wobei  $C_w$  die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache  $L_{B_i}$  einer Maschine  $M_i$  kompiliert. Fall die Notwendigkeit besteht die **Maschine**  $M_i$  anzugeben, zu dessen **Maschinensprache**  $L_{B_i}$  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  $C_{w.k}^{o-j}$  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 **Maschinenenncode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

### Definition 2.3: Maschinensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch- und Execute-Phase**, genauer gesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen konstanten** Anzahl von Fetch- und Execute Phasen im **komplexeren Fall**. Die Maschinenbefehle sind meist so designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.<sup>a,b</sup>*

<sup>a</sup>Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.8) haben.

<sup>b</sup>C. Scholl, „Betriebssysteme“.

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

*Eine sehr **hardwarenahe** Programmiersprache, deren **Instructions** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen<sup>a</sup> haben. Viele **Instructions** haben eine ähnliche übliche Struktur **Operation** <Operanden>, mit einer **Operation**, die einem **Opcode** eines Maschinenbefehls bezeichnet und keinen oder mehreren **Operanden**, wie die späteren Maschinenbefehle, denen sie entsprechen. Allerdings gibt es oftmals noch viel „syntaktischen Zucker“ innerhalb<sup>b</sup> der Instructions und drumherum<sup>c,d</sup>.*

<sup>a</sup>Instructions der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

<sup>b</sup>Z.B. erlaubt die Assemblersprache des **GCC** für die **X86\_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.

<sup>c</sup>Z.B. sind im X86\_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.

<sup>d</sup>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 **Maschinenenncode** 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 **Maschinenenncode** 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 **Maschinencode** bzw. **Objectcode** in **binärer Repräsentation**, der in **Maschiensprache** geschrieben ist.<sup>a</sup>

<sup>a</sup>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 Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.<sup>a</sup>

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

### Definition 2.7: Linker

Programm, das **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt**, sodass unter anderem kein vermeidbarer **doppelter Code** darin vorkommt.<sup>a</sup>

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

Der **Maschinencode**, denn ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschinenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschine, 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 Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenem RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, 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<sup>2</sup>.

### Definition 2.8: Immediate

**Konstanter Wert**, der als **Teil eines Maschinenbefehls** gespeichert ist und dessen **Wertebereich** dementsprechend auch durch die die Anzahl an Bits, die ihm innerhalb dieses **Maschinenbefehls** zur Verfügung gestellt sind, **beschränkter** ist als bei sonstigen Werten innerhalb des Hauptspeichers, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.<sup>a</sup>

<sup>a</sup>Ljohhuh, *What is an immediate value?*

<sup>2</sup>Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschinencode 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 Maschinenbefehle übergibt, anstatt z.B. eine unnötig **platzverbrauchenden** UTF-8 Codierung zu verwenden, die nur in sehr vielen Schritten einen Befehl verarbeiten kann, da die Register und Speicherzellen des Hauptspeichers üblicherweise nur **32- bzw. 64-Bit Breite** haben.

**Definition 2.9: Transpiler (bzw. Source-to-source Compiler)**

Kompiliert zwischen Sprachen, die ungefähr auf dem *gleichen* Level an *Abstraktion* arbeiten<sup>ab</sup>

<sup>a</sup>Die Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprache Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.

<sup>b</sup>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 **Maschinen Sprachen**  $B_1$  und  $B_2$  haben.<sup>ab</sup>

<sup>a</sup>Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler**  $C_{PicoC}^{Python}$ .

<sup>b</sup>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 **Maschinen Sprache**  $B_2$  einer Zielmaschine  $M_2$  läuft.<sup>3</sup>

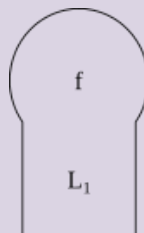
**2.1.1 T-Diagramme**

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 Programm (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschine (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.<sup>a</sup>



<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

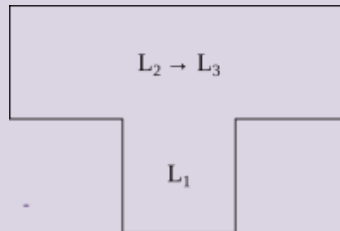
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein  $L$  dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

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

**Definition 2.12: T-Diagramm Übersetzer (bzw. eng. Translator)**

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

Für den **Übersetzer** gelten genauso, wie für einen **Compiler**<sup>a</sup> die **Beziehungen** in 2.2.1.<sup>b</sup>

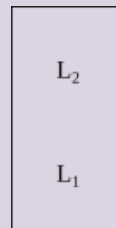


<sup>a</sup>Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 2.13: T-Diagramm Interpreter**

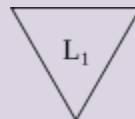
Repräsentiert einen **Interpreter**, der in der **Sprache**  $L_1$  geschrieben ist und **Programme** in der **Sprache**  $L_2$  interpretiert.<sup>a</sup>



<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 2.14: T-Diagramm Maschine**

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache**  $L_1$  ausführt.<sup>a,b</sup>



<sup>a</sup>Wenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).

<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

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

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

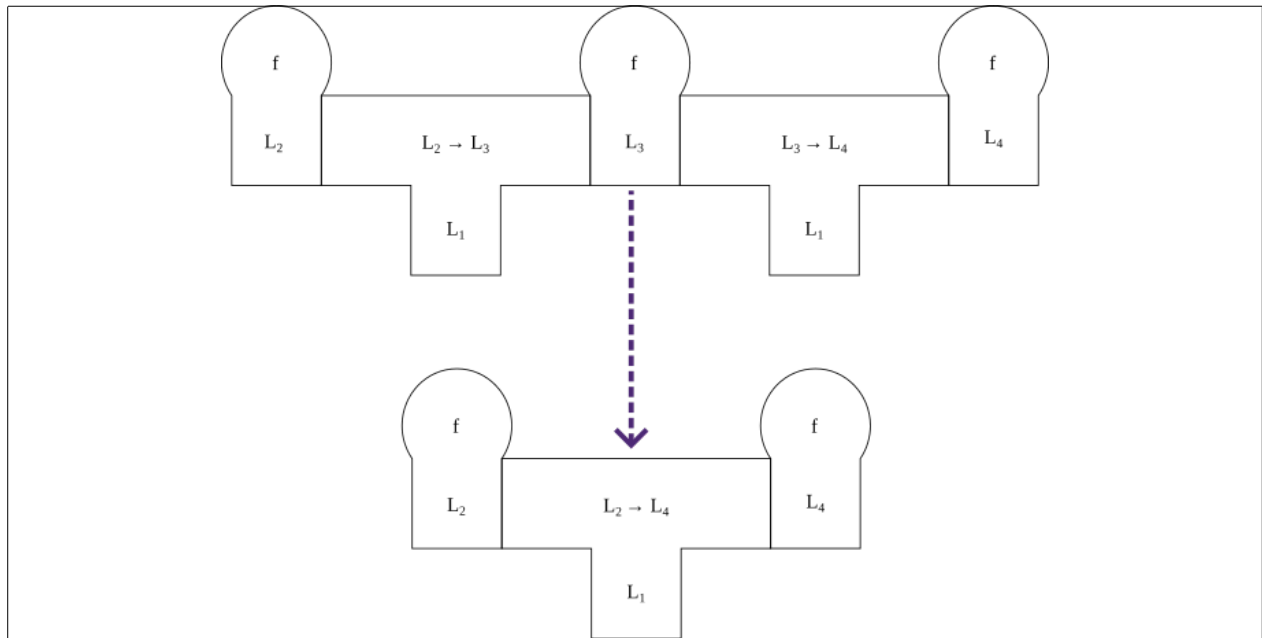


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

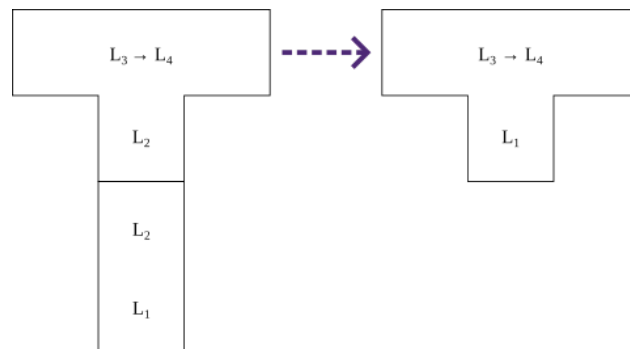


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

## 2.2 Grammatiken

## 2.3 Grundlagen

### Definition 2.15: Sprache

$a$

<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 2.16: Chomsky Hierarchie***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 2.17: Grammatik***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 2.18: Reguläre Sprachen***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 2.19: Kontextfreie Sprachen***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 2.20: Ableitung***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 2.21: Links- und Rechtsableitung***a*<sup>a</sup>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.<sup>a</sup>

<sup>a</sup>Parsing Expressions · Crafting Interpreters.

### 2.3.1 Mehrdeutige Grammatiken

#### Definition 2.23: Ableitungsbaum

*a*

<sup>a</sup>Nebel, „Theoretische Informatik“.

#### Definition 2.24: Mehrdeutige Grammatik

*a*

<sup>a</sup>Nebel, „Theoretische Informatik“.

### 2.3.2 Präzidenz und Assoziativität

#### Definition 2.25: Assoziativität

*a*

<sup>a</sup>*Parsing Expressions · Crafting Interpreters.*

#### Definition 2.26: Präzidenz

*a*

<sup>a</sup>*Parsing Expressions · Crafting Interpreters.*

#### Definition 2.27: Wortproblem

*a*

<sup>a</sup>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<sup>a</sup>. Dabei steht **LL** für *left-to-right* und *leftmost-derivation*, da das **Eingabewort** von *links nach rechts* geparsed und immer **Linksableitungen** genommen werden müssen<sup>b</sup>, damit die obige Bedingung mit den **nächsten**  $k$  Symbolen gilt.<sup>c</sup>

<sup>a</sup>Das wird auch als **Lookahead** von  $k$  bezeichnet.

<sup>b</sup>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.

<sup>c</sup>Nebel, „Theoretische Informatik“.

## 2.4 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb der **Pipe Architektur** 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 2.29) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Diese Folgen endlicher Symbole werden auch **Lexeme** (Definition 2.30) genannt.

#### Definition 2.29: 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<sup>a</sup>, die für die Beschreibung eines **Tokens**  $T$  zuständig sind.<sup>b</sup>*

<sup>a</sup>Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

<sup>b</sup>Thiemann, „Compilerbau“.

#### Definition 2.30: Lexeme

*Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token**  $T$  einer **Sprache**  $L_{Lex}$  matched.<sup>a</sup>*

<sup>a</sup>Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.31) im **Inputstring** identifiziert 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 2.31) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

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

*Ein **Lexer** ist eine **partielle Funktion**  $lex : \Sigma^* \rightarrow (N \times W)^*$ , 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.<sup>a</sup>*

<sup>a</sup>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 verzubäuen 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 2.32) von **Variablen, Konstanten und Funktionen** die Symbole<sup>a</sup>.

<sup>a</sup>Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel 3 **Symboltabelle** genannt wird.

**Definition 2.32: Bezeichner (bzw. Identifier)**

*Tokenwert, der eine Konstante, Variable, Funktion usw. **eindeutig** benennt.<sup>a,b</sup>*

<sup>a</sup>Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterscheidungsmerkmal hinzugenommen, damit es eindeutig ist.

<sup>b</sup>Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>4</sup> 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`<sup>5</sup>, 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 2.33) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99<sup>6</sup>. 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<sup>7</sup>, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 3.1.1 liefert den Beweis, dass die Sprache  $L_{PicoC\_Lex}$  des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.18 erfüllt. Einzig die Produktion `CHAR ::= "'ASCII_CHAR'"` sieht problematisch aus, kann allerdings auch als `{CHAR ::= "'CHAR2', CHAR2 ::= ASCII_CHAR'"}` **regulär** ausgedrückt werden<sup>8</sup>. Somit existiert eine **reguläre Grammatik**, welche die **Sprache**  $L_{PicoC\_Lex}$  beschreibt und damit ist die **Sprache**  $L_{PicoC\_Lex}$  **regulär**.

<sup>4</sup>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 Programmiersprache, die man zur Implementierung des Lexers nutzt wegabstrahiert.

<sup>5</sup>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.

<sup>6</sup>Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

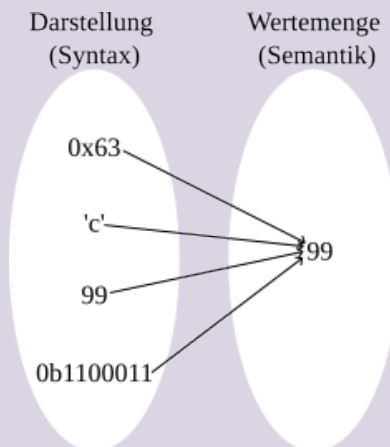
<sup>7</sup>Man nennt das auch einem **Lookahead** von 1

<sup>8</sup>Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten lassen.



**Definition 2.33: Literal**

Eine von möglicherweise vielen weiteren *Darstellungsformen* (als *Zeichenkette*) für ein und denselben *Wert* eines *Datentyps*.<sup>a</sup>



<sup>a</sup>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.

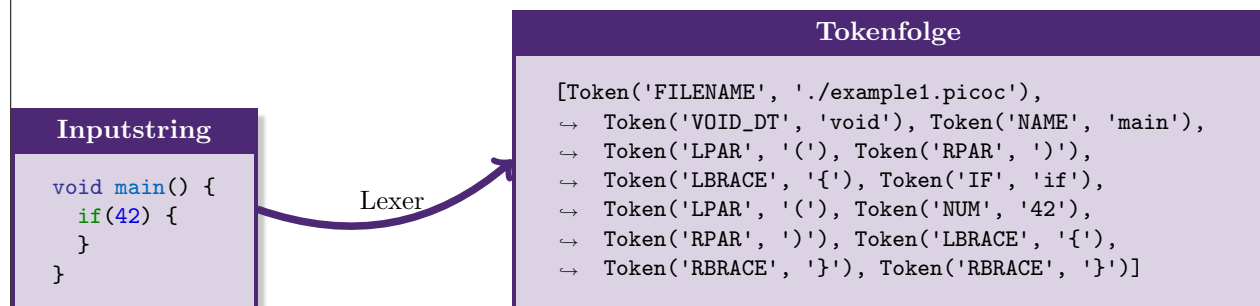


Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

## 2.5 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.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.34) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.36), ein **Derivation Tree** (Definition 2.35) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 2.41) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

**Definition 2.34: Konkrete 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 Konkreter Syntax aufgeschrieben.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.35: Derivation Tree (bzw. Parse Tree)**

*Compilerinterne Darstellung eines in Konkreter 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 verwendete Nicht-Terminalsymbol einer Produktion der Grammatik  $G_{Parse}$ , die ein Teil der Konkrete Syntax ist, bildet einen eigenen Inneren Knoten.*

*Der Derivation Tree wird optimalerweise immer so konstruiert bzw. die Konkrete Syntax immer so definiert, dass sich möglichst einfach ein Abstract Syntax Tree daraus konstruieren lässt.<sup>a</sup>*

<sup>a</sup>JSON parser - Tutorial — Lark documentation.

**Definition 2.36: Parser**

*Ein Parser ist ein Programm, dass aus einem Inputstring, der in Konkreter Syntax geschrieben ist, eine compilerinterne Darstellung, den Derivation Tree generiert, was auch als Parsen bezeichnet wird.<sup>a, b</sup>*

<sup>a</sup>Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von Konkreter Syntax in Abstrakte Syntax übersetzt. Im Folgenden wird allerdings die Definition 2.36 verwendet.

<sup>b</sup>JSON parser - Tutorial — Lark documentation.

An dieser Stelle könnte möglicherweise eine Verwirrung entstehen, 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 Insektenlexikon 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<sup>a</sup>.

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 Insektenbegegnungen in einer für die **Weiterverarbeitung tauglichen Form**<sup>b</sup>.

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

<sup>a</sup>Das würde z.B. der Rolle eines **Semikolon** ; in der Sprache  $L_{PicoC}$  entsprechen.

<sup>b</sup>Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beeinflussen 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 konkreten Wert diese **Zahl** hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.6 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.37), denn ein Parser löst das **Wortproblem** (Definition 2.27) für die **Sprache**, die durch die **Konkrete 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 2.37: Recognizer (bzw. Erkennen)

*Entspricht dem Maschinenmodell 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 **Konkrete Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Syntax** beschrieben wird oder nicht<sup>ab</sup>*

<sup>a</sup>Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

<sup>b</sup>Thiemann, „Compilerbau“.

Für das **Parzen** 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.<sup>a</sup>

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 gewünschte Inputstring 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.<sup>b</sup>

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.<sup>c</sup>

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkreten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.<sup>d</sup>
- **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.<sup>e</sup>

<sup>a</sup> *What is Top-Down Parsing?*

<sup>b</sup> 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.

<sup>c</sup> 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.

<sup>d</sup> *What is Bottom-up Parsing?*

<sup>e</sup> Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 2.38) und **Visitors** (Definition 2.39) generiert und ist das Endprodukt der **Syntaktischen Analyse**. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inputstring von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 2.40).

#### Definition 2.38: 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.<sup>a</sup>

<sup>a</sup> *Transformers & Visitors — Lark documentation.*

#### Definition 2.39: 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.<sup>a,b</sup>

<sup>a</sup> 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.

<sup>b</sup> *Transformers & Visitors — Lark documentation.*

**Definition 2.40: Abstrakte Syntax**

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

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.<sup>a</sup>

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.41: 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.<sup>a</sup>

<sup>a</sup>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 2.4 fortgeführt.

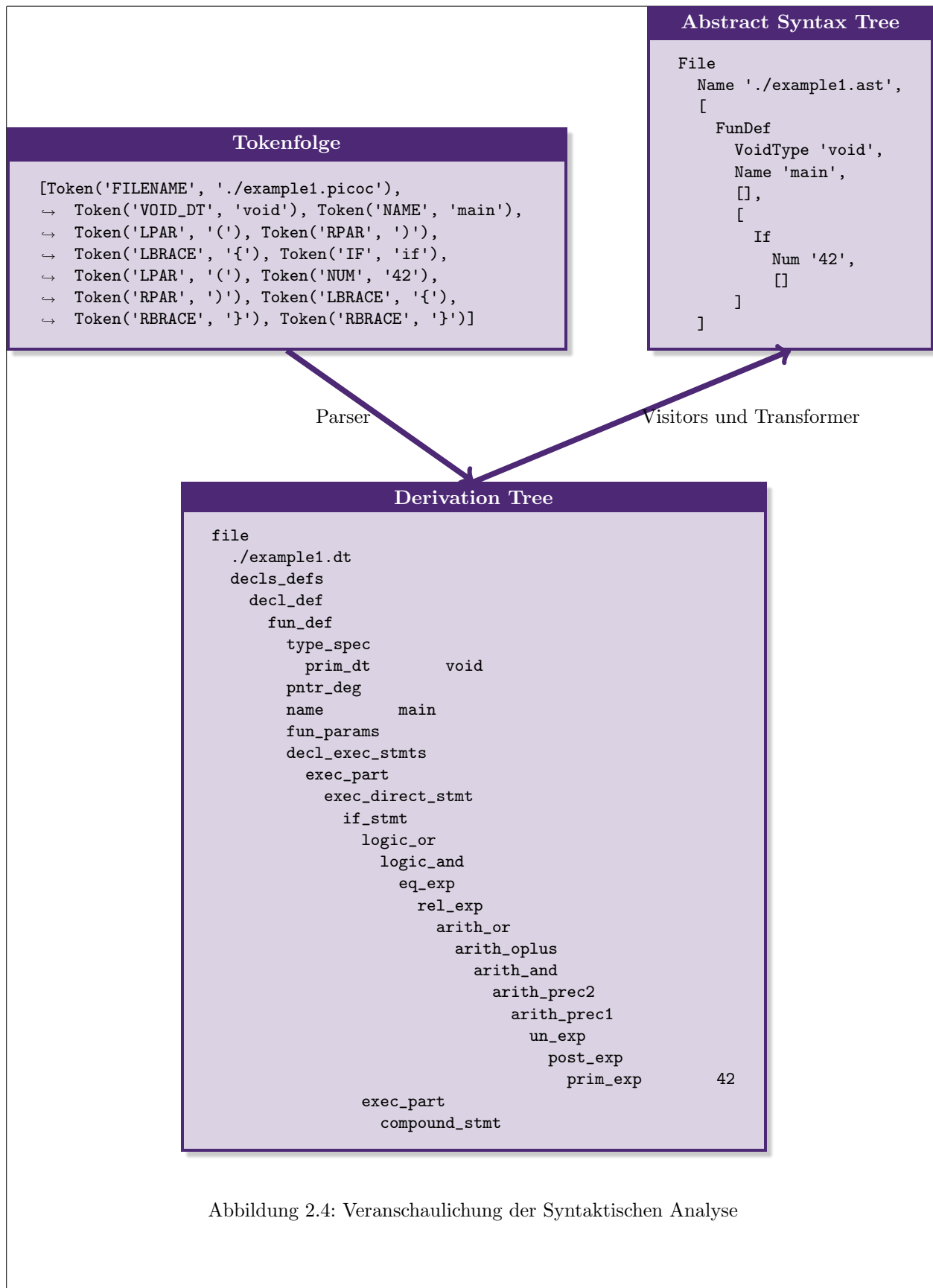


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse

## 2.6 Code Generierung

### Definition 2.42: Pass

*a*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

### Definition 2.43: Monadische Normalform

*a*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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

## 2.7 Fehlermeldungen

### Definition 2.44: Fehlermeldung

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

1. Ein Program beim **Kompilieren** von der **Konkreten Syntax** abweicht, also der **Inpustring** sich nicht mit der Konrekten Syntax **ableiten** lässt oder auf etwas **zugegriffen** werden soll, was noch **nicht** deklariert oder definiert wurde.
2. Beim Ausführen eine **verbotene** Operation ausgeführt wurde.<sup>a</sup>

<sup>a</sup>*Errors in C/C++ - GeeksforGeeks.*

### 2.7.1 Kategorien von Fehlermeldungen





# 3 Implementierung

## 3.1 Lexikalische Analyse

### 3.1.1 Konkrete Syntax für Lexer erstellen

<i>COMMENT</i>	::=	"//"/[ $\backslash$ n]*/   "/"*/ (. $\backslash$ n)*?/ "*"*/	<i>L_Comment</i>
<i>RET_COMMENT.2</i>	::=	"//""?"#[ $\backslash$ n]*/	
<i>DIG_NO_0</i>	::=	"1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"	<i>L_Arith</i>
<i>DIG_WITH_0</i>	::=	"0"   <i>DIG_NO_0</i>	
<i>NUM</i>	::=	"0"   <i>DIG_NO_0</i> <i>DIG_WITH_0</i> *	
<i>ASCII_CHAR</i>	::=	"_".~"	
<i>CHAR</i>	::=	"'" <i>ASCII_CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII_CHAR</i> + ".picoc"	
<i>LETTER</i>	::=	"a"..z"   "A"..Z"	
<i>NAME</i>	::=	( <i>LETTER</i>   "_") ( <i>LETTER</i> — <i>DIG_WITH_0</i> — "_")*	
<i>name</i>	::=	<i>NAME</i>   <i>INT_NAME</i>   <i>CHAR_NAME</i>   <i>VOID_NAME</i>	
<i>NOT</i>	::=	"~"	
<i>REF_AND</i>	::=	"&"	
<i>un_op</i>	::=	<i>SUB_MINUS</i>   <i>LOGIC_NOT</i>   <i>NOT</i>   <i>MUL_DEREF_PNTR</i>   <i>REF_AND</i>	
<i>MUL_DEREF_PNTR</i>	::=	"*"	
<i>DIV</i>	::=	"/"	
<i>MOD</i>	::=	"%"	
<i>prec1_op</i>	::=	<i>MUL_DEREF_PNTR</i>   <i>DIV</i>   <i>MOD</i>	
<i>ADD</i>	::=	"+"	
<i>SUB_MINUS</i>	::=	"-"	
<i>prec2_op</i>	::=	<i>ADD</i>   <i>SUB_MINUS</i>	
<i>LT</i>	::=	"<"	<i>L_Logic</i>
<i>LTE</i>	::=	"<="	
<i>GT</i>	::=	">"	
<i>GTE</i>	::=	">="	
<i>rel_op</i>	::=	<i>LT</i>   <i>LTE</i>   <i>GT</i>   <i>GTE</i>	
<i>EQ</i>	::=	"=="	
<i>NEQ</i>	::=	"!="	
<i>eq_op</i>	::=	<i>EQ</i>   <i>NEQ</i>	
<i>LOGIC_NOT</i>	::=	"!"	
<i>INT_DT.2</i>	::=	"int"	
<i>INT_NAME.3</i>	::=	"int" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   "_")+	<i>L_Assign_Alloc</i>
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   "_")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   "_")+	
<i>prim_dt</i>	::=	<i>INT_DT</i>   <i>CHAR_DT</i>   <i>VOID_DT</i>	

### 3.1.2 Basic Lexer

## 3.2 Syntaktische Analyse

### 3.2.1 Konkrete Syntax für Parser erstellen

In 3.2.1

<i>prim_exp</i>	::=	<i>name</i>   <i>NUM</i>   <i>CHAR</i>   "(" <i>logic_or</i> ")"	<i>L_Arith</i> +
<i>post_exp</i>	::=	<i>array_subscr</i>   <i>struct_attr</i>   <i>fun_call</i>   <i>input_exp</i>   <i>print_exp</i>   <i>prim_exp</i>	<i>L_Array</i> + <i>L_Pntr</i> +
<i>un_exp</i>	::=	<i>un_opun_exp</i>   <i>post_exp</i>	<i>L_Struct</i> + <i>L_Fun</i>
<i>input_exp</i>	::=	"input" "(" "	<i>L_Arith</i>
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i>   <i>un_exp</i>	
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i>   <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i>   <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i>   <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i>   <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i>   <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_oprel_exp</i>   <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i>   <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> "  " <i>logic_and</i>   <i>logic_and</i>	
<i>type_spec</i>	::=	<i>prim_dt</i>   <i>struct_spec</i>	<i>L_Assign_Alloc</i>
<i>alloc</i>	::=	<i>type_spec</i> <i>pntr_decl</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	
<i>initializer</i>	::=	<i>logic_or</i>   <i>array_init</i>   <i>struct_init</i>	
<i>init_stmt</i>	::=	<i>alloc</i> "=" <i>initializer</i> ";"	
<i>const_init_stmt</i>	::=	"const" <i>type_spec</i> <i>name</i> "=" <i>NUM</i> ";"	
<i>pntr_deg</i>	::=	"*" *	<i>L_Pntr</i>
<i>pntr_decl</i>	::=	<i>pntr_deg</i> <i>array_decl</i>   <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]" ) *	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name</i> <i>array_dims</i>   "(" <i>pntr_decl</i> ")" <i>array_dims</i>	
<i>array_init</i>	::=	"{" <i>initializer</i> ("," <i>initializer</i> ) * "}"	
<i>array_subscr</i>	::=	<i>post_exp</i> "[" <i>logic_or</i> "]"	
<i>struct_spec</i>	::=	"struct" <i>name</i>	<i>L_Struct</i>
<i>struct_params</i>	::=	( <i>alloc</i> ";" ) +	
<i>struct_decl</i>	::=	"struct" <i>name</i> "{" <i>struct_params</i> "}"	
<i>struct_init</i>	::=	"{" " " <i>name</i> "=" <i>initializer</i> ("," " " <i>name</i> "=" <i>initializer</i> ) * "}"	
<i>struct_attr</i>	::=	<i>post_exp</i> "." <i>name</i>	
<i>if_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_If_Else</i>
<i>if_else_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i> "else" <i>exec_part</i>	
<i>while_stmt</i>	::=	"while" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_Loop</i>
<i>do_while_stmt</i>	::=	"do" <i>exec_part</i> "while" "(" <i>logic_or</i> ")" ";"	

Grammar 3.2.1: Konkrete Syntax des Parsers in EBNF, Teil 1

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

Grammar 3.2.2: Konkrete Syntax des Parsers in EBNF, Teil 2

### 3.2.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**<sup>1</sup>. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 3.1 aufgelistet.

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

Tabelle 3.1: Präzidenzregeln von PicoC

<sup>1</sup>*C Operator Precedence - [cppreference.com](http://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     ptr_decl
10    ptr_deg *
11    array_decl
12    array_dims
13    5
14    6
15    ptr_decl
16    ptr_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   ptr_deg
27   name main
28   fun_params
29   decl_exec_stmts
30   decl_part
31   decl_exp_stmt
32   alloc
33   ptr_decl
34   ptr_deg *
35   array_decl
36   array_dims

```

```
37         3
38         2
39     ptr_decl
40     ptr_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 Nodes

### 3.2.5.2 RETI Nodes

### 3.2.5.3 Abstrakte Syntax

<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>BinOp</i> ( <i>&lt;exp&gt;</i> , <i>&lt;bin_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>UnOp</i> ( <i>&lt;un_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Call</i> ( <i>Name('input')</i> , <i>None</i> )	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>Call</i> ( <i>Name('print')</i> , <i>&lt;exp&gt;</i> )	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom</i> ( <i>&lt;exp&gt;</i> , <i>&lt;rel&gt;</i> , <i>&lt;exp&gt;</i> )   <i>ToBool</i> ( <i>&lt;exp&gt;</i> )	
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>	
<i>assign_lhs</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>&lt;rel_loc&gt;</i>	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )	
<i>stmt</i>	::=	<i>Assign</i> ( <i>&lt;assign_lhs&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Exp</i> ( <i>&lt;exp_stmts&gt;</i> )	
<i>datatype</i>	::=	<i>PntrDecl</i> ( <i>Num(str)</i> , <i>&lt;datatype&gt;</i> )	<i>L_Pntr</i>
<i>deref_loc</i>	::=	<i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )   <i>&lt;ref_loc&gt;</i>	
<i>ref_loc</i>	::=	<i>Name(str)</i>   <i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )	
<i>exp</i>	::=	<i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )	
<i>datatype</i>	::=	<i>ArrayDecl</i> ( <i>Num(str)</i> +, <i>&lt;datatype&gt;</i> )	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Array</i> ( <i>&lt;exp&gt;</i> +)	
<i>datatype</i>	::=	<i>StructSpec</i> ( <i>Name(str)</i> )	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )   <i>Struct</i> ( <i>Assign</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> ) +)	
<i>decl_def</i>	::=	<i>StructDecl</i> ( <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> ) +)	
<i>stmt</i>	::=	<i>If</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>IfElse</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *, <i>&lt;stmt&gt;</i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>DoWhile</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	<i>L_Fun</i>
<i>exp_stmts</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	
<i>stmt</i>	::=	<i>Return</i> ( <i>&lt;exp&gt;</i> )	
<i>decl_def</i>	::=	<i>FunDecl</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*)   <i>FunDef</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*, <i>&lt;stmt&gt;</i> *)	
<i>file</i>	::=	<i>File</i> ( <i>Name(str)</i> , <i>&lt;decl_def&gt;</i> *)	<i>L_File</i>

Grammar 3.2.3: Abstrakte Syntax für *L\_PiCC*

### 3.2.5.4 Transformer

### 3.2.5.5 Codebeispiel

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

```
1 File
2   Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           PtrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '5',
14                Num '6'
15              ],
16            PtrDecl
17              Num '1',
18              IntType 'int',
19            Name 'attr'
20          ],
21      FunDef
22        VoidType 'void',
23        Name 'main',
24        [],
25        [
26          Exp
27            Alloc
28              Writeable,
29              PtrDecl
30                Num '1',
31                ArrayDecl
32                  [
33                    Num '3',
34                    Num '2'
35                  ],
36                PtrDecl
37                  Num '1',
38                  StructSpec
39                    Name 'st',
40                  Name 'var'
41                ]
42          ]
43        ]
44      ]
45    ]
46  ]
```

Code 3.4: Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert



## 3.3 Code Generierung

### 3.3.1 Übersicht

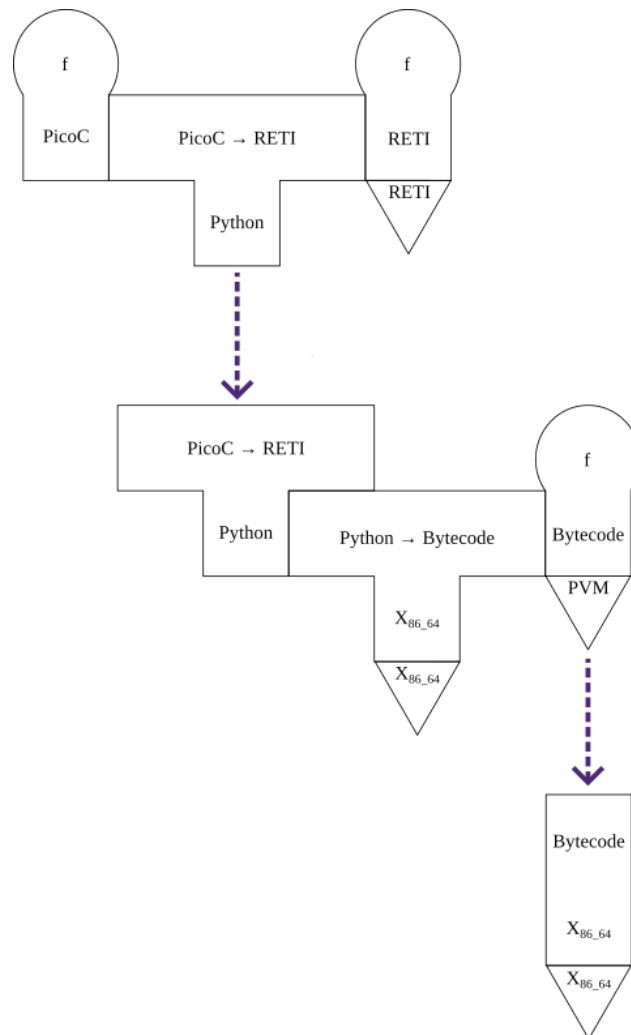


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

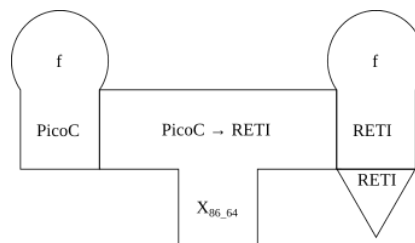


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

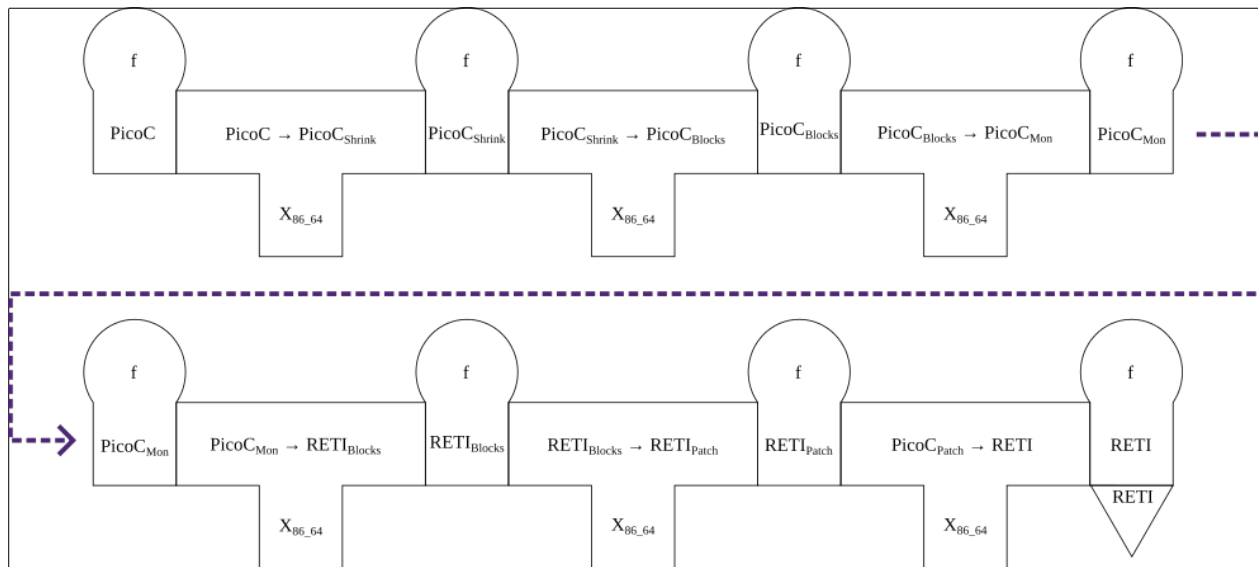


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

### 3.3.2 Passes

#### 3.3.2.1 Kompositionen mit besonderer Bedeutung

#### 3.3.2.2 PicoC-Shrink Pass

##### 3.3.2.2.1 Codebeispiel

```

1 // Author: Christoph Scholl, from the Operating Systems Lecture
2
3 void main() {
4     int n = 4;
5     int res = 1;
6     while (1) {
7         if (n == 1) {
8             return;
9         }
10        res = n * res;
11        n = n - 1;
12    }
13 }

```

Code 3.5: PicoC Code für Codebeispiel

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',

```

```

7      [],
8      [
9          Assign
10         Alloc
11         Writeable,
12         IntType 'int',
13         Name 'n',
14         Num '4',
15         Assign
16         Alloc
17         Writeable,
18         IntType 'int',
19         Name 'res',
20         Num '1',
21         While
22         Num '1',
23         [
24             If
25             Atom
26             Name 'n',
27             Eq '==',
28             Num '1',
29             [
30                 Return
31                 Empty
32             ],
33             Assign
34             Name 'res',
35             BinOp
36             Name 'n',
37             Mul '*',
38             Name 'res',
39             Assign
40             Name 'n',
41             BinOp
42             Name 'n',
43             Sub '-',
44             Num '1'
45         ]
46     ]
47 ]

```

Code 3.6: Abstract Syntax Tree für Codebespiel

```

1 File
2   Name './example_faculty_it.picoc_shrink',
3   [
4       FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9           Assign
10          Alloc

```

```

11     Writeable,
12     IntType 'int',
13     Name 'n',
14     Num '4',
15   Assign
16   Alloc
17     Writeable,
18     IntType 'int',
19     Name 'res',
20     Num '1',
21   While
22     Num '1',
23   [
24     If
25       Atom
26         Name 'n',
27         Eq '==',
28         Num '1',
29       [
30         Return
31         Empty
32       ],
33     Assign
34       Name 'res',
35       BinOp
36         Name 'n',
37         Mul '*',
38         Name 'res',
39     Assign
40       Name 'n',
41       BinOp
42         Name 'n',
43         Sub '-',
44         Num '1'
45   ]
46 ]
47 ]

```

Code 3.7: PicoC Shrink Pass für Codebeispiel

### 3.3.2.3 PicoC-Blocks Pass

#### 3.3.2.3.1 Abstrakte Syntax

<i>decl_def</i>	$::=$	<i>FunDef</i> ( $\langle datatype \rangle$ , <i>Name</i> ( <i>str</i> ), <i>Alloc</i> ( <i>Writeable</i> () , $\langle datatype \rangle$ , <i>Name</i> ( <i>str</i> ))* , $\langle block \rangle$ *)	<i>L_Fun</i>
<i>block</i>	$::=$	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), $\langle stmt \rangle$ *)	<i>L_Blocks</i>
<i>stmt</i>	$::=$	<i>Goto</i> ( <i>Name</i> ( <i>str</i> ))   <i>NewStackframe</i> ( <i>Name</i> () , <i>Goto</i> ( <i>str</i> ))   <i>RemoveStackframe</i> ()   <i>SetScope</i> ( <i>Name</i> ( <i>str</i> ))   <i>SingleLineComment</i> ( <i>str</i> , <i>str</i> )	

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

**3.3.2.3.2 Codebeispiel**

```
1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.5',
11          [
12            Assign
13              Alloc
14                Writeable,
15                IntType 'int',
16                Name 'n',
17                Num '4',
18            Assign
19              Alloc
20                Writeable,
21                IntType 'int',
22                Name 'res',
23                Num '1',
24            GoTo
25              Name 'condition_check.4'
26          ],
27        Block
28          Name 'condition_check.4',
29          [
30            IfElse
31              Num '1',
32              GoTo
33                Name 'while_branch.3',
34              GoTo
35                Name 'while_after.0'
36          ],
37        Block
38          Name 'while_branch.3',
39          [
40            IfElse
41              Atom
42                Name 'n',
43                Eq '==',
44                Num '1',
45              GoTo
46                Name 'if.2',
47              GoTo
48                Name 'if_else_after.1'
49          ],
50        Block
51          Name 'if.2',
52          [
53            Return
54              Empty
55          ],
```

```

56     Block
57     Name 'if_else_after.1',
58     [
59         Assign
60         Name 'res',
61         BinOp
62         Name 'n',
63         Mul '*',
64         Name 'res',
65         Assign
66         Name 'n',
67         BinOp
68         Name 'n',
69         Sub '-',
70         Num '1',
71         GoTo
72         Name 'condition_check.4'
73     ],
74     Block
75     Name 'while_after.0',
76     []
77 ]
78 ]

```

Code 3.8: PicoC-Blocks Pass für Codebeispiel

### 3.3.2.4 PicoC-Mon Pass

#### 3.3.2.4.1 Abstrakte Syntax

<i>ref_loc</i>	$::=$	$Tmp(Num(str)) \mid StackRead(Num(str))$ $\mid StackWrite(Num(str)) \mid GlobalRead(Num(str))$ $\mid GlobalWrite(Num(str))$	<i>L_Assign_Alloc</i>
<i>error_data</i>	$::=$	$\langle exp \rangle \mid Pos(Num(str), Num(str))$	
<i>exp</i>	$::=$	$Stack(Num(str)) \mid Ref(\langle ref\_loc \rangle, \langle datatype \rangle, \langle error\_data \rangle)$	
<i>stmt</i>	$::=$	$Exp(\langle exp \rangle)$ $\mid Assign(Alloc(Writeable(), StructSpec(Name(str)), Name(str)),$ $\quad Struct(Assign(Name(str), \langle exp \rangle +, \langle datatype \rangle))$ $\mid Assign(Alloc(Writeable(), ArrayDecl(Num(str) +, \langle datatype \rangle),$ $\quad Name(str)), Array(\langle exp \rangle +, \langle datatype \rangle))$	
<i>symbol_table</i>	$::=$	$SymbolTable(\langle symbol \rangle)$	<i>L_Symbol_Table</i>
<i>symbol</i>	$::=$	$Symbol(\langle type\_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)$	
<i>type_qual</i>	$::=$	$Empty()$	
<i>datatype</i>	$::=$	$BuiltIn() \mid SelfDefined()$	
<i>name</i>	$::=$	$Name(str)$	
<i>val</i>	$::=$	$Num(str) \mid Empty()$	
<i>pos</i>	$::=$	$Pos(Num(str), Num(str)) \mid Empty()$	
<i>size</i>	$::=$	$Num(str) \mid Empty()$	

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

## Definition 3.1: Symoltabelle

## 3.3.2.4.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'main.5',
6       [
7         Exp
8           Num '4',
9         Assign
10          GlobalWrite
11            Num '0',
12          Tmp
13            Num '1',
14        Exp
15          Num '1',
16        Assign
17          GlobalWrite
18            Num '1',
19          Tmp
20            Num '1',
21        Exp
22          GoTo
23            Name 'condition_check.4'
24      ],
25    Block
26      Name 'condition_check.4',
27      [
28        Exp
29          Num '1',
30        IfElse
31          Tmp
32            Num '1',
33          GoTo
34            Name 'while_branch.3',
35          GoTo
36            Name 'while_after.0'
37      ],
38    Block
39      Name 'while_branch.3',
40      [
41        Exp
42          GlobalRead
43            Num '0',
44        Exp
45          Num '1',
46        Exp
47          Atom
48            Tmp
49              Num '2',
50            Eq '==',

```

```
51         Tmp
52         Num '1',
53     IfElse
54         Tmp
55         Num '1',
56         GoTo
57         Name 'if.2',
58         GoTo
59         Name 'if_else_after.1'
60     ],
61 Block
62     Name 'if.2',
63     [
64         Return
65         Empty
66     ],
67 Block
68     Name 'if_else_after.1',
69     [
70         Exp
71         GlobalRead
72         Num '0',
73         Exp
74         GlobalRead
75         Num '1',
76         Exp
77         BinOp
78         Tmp
79         Num '2',
80         Mul '*',
81         Tmp
82         Num '1',
83         Assign
84         GlobalWrite
85         Num '1',
86         Tmp
87         Num '1',
88         Exp
89         GlobalRead
90         Num '0',
91         Exp
92         Num '1',
93         Exp
94         BinOp
95         Tmp
96         Num '2',
97         Sub '-',
98         Tmp
99         Num '1',
100        Assign
101        GlobalWrite
102        Num '0',
103        Tmp
104        Num '1',
105        Exp
106        GoTo
107        Name 'condition_check.4'
```



```

108 ],
109 Block
110   Name 'while_after.0',
111   [
112     Return
113     Empty
114   ]
115 ]

```

Code 3.9: PicoC-Mon Pass für Codebeispiel

### 3.3.2.5 RETI-Blocks Pass

#### 3.3.2.5.1 Abstrakte Syntax

<i>program</i>	$::=$	$Program(Name(str), \langle block \rangle^*)$	$L\_Program$
<i>exp_stmts</i>	$::=$	$Goto(str)$	$L\_Blocks$
<i>instrs_before</i>	$::=$	$Num(str)$	
<i>num_instrs</i>	$::=$	$Num(str)$	
<i>block</i>	$::=$	$Block(Name(str), \langle instr \rangle^*, \langle instrs\_before \rangle, \langle num\_instrs \rangle)$	
<i>instr</i>	$::=$	$Goto(Name(str))$	

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

#### 3.3.2.5.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         SUBI SP 1,
8         LOADI ACC 4,
9         STOREIN SP ACC 1,
10        LOADIN SP ACC 1,
11        STOREIN DS ACC 0,
12        ADDI SP 1,
13        SUBI SP 1,
14        LOADI ACC 1,
15        STOREIN SP ACC 1,
16        LOADIN SP ACC 1,
17        STOREIN DS ACC 1,
18        ADDI SP 1,
19        Exp
20          GoTo
21            Name 'condition_check.4'
22        ],
23      Block
24        Name 'condition_check.4',
25        [
26          SUBI SP 1,

```

```

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

```

```

84      ADDI SP 1,
85      SUBI SP 1,
86      LOADIN DS ACC 0,
87      STOREIN SP ACC 1,
88      SUBI SP 1,
89      LOADI ACC 1,
90      STOREIN SP ACC 1,
91      LOADIN SP ACC 2,
92      LOADIN SP IN2 1,
93      SUB ACC IN2,
94      STOREIN SP ACC 2,
95      ADDI SP 1,
96      LOADIN SP ACC 1,
97      STOREIN DS ACC 0,
98      ADDI SP 1,
99      Exp
100     GoTo
101     Name 'condition_check.4'
102 ],
103 Block
104   Name 'while_after.0',
105   [
106     LOADIN BAF PC -1
107   ]
108 ]

```

Code 3.10: RETI-Blocks Pass für Codebeispiel

### 3.3.2.6 RETI-Patch Pass

#### 3.3.2.6.1 Abstrakte Syntax

$$\text{stmt} ::= \text{Exit}(\text{Num}(\text{str}))$$
Grammar 3.3.4: Abstrakte Syntax für  $L_{RETI\_Patch}$ 

#### 3.3.2.6.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.6',
6       [],
7     Block
8       Name 'main.5',
9       [
10        SUBI SP 1,
11        LOADI ACC 4,
12        STOREIN SP ACC 1,
13        LOADIN SP ACC 1,
14        STOREIN DS ACC 0,
15        ADDI SP 1,

```

```
16     SUBI SP 1,
17     LOADI ACC 1,
18     STOREIN SP ACC 1,
19     LOADIN SP ACC 1,
20     STOREIN DS ACC 1,
21     ADDI SP 1
22 ],
23 Block
24   Name 'condition_check.4',
25   [
26     SUBI SP 1,
27     LOADI ACC 1,
28     STOREIN SP ACC 1,
29     LOADIN SP ACC 1,
30     ADDI SP 1,
31     JUMP== GoTo
32       Name 'while_after.0';
33   ],
34 Block
35   Name 'while_branch.3',
36   [
37     SUBI SP 1,
38     LOADIN DS ACC 0,
39     STOREIN SP ACC 1,
40     SUBI SP 1,
41     LOADI ACC 1,
42     STOREIN SP ACC 1,
43     LOADIN SP ACC 2,
44     LOADIN SP IN2 1,
45     SUB ACC IN2,
46     JUMP== 3;,
47     LOADI ACC 0,
48     JUMP 2;,
49     LOADI ACC 1,
50     STOREIN SP ACC 2,
51     ADDI SP 1,
52     LOADIN SP ACC 1,
53     ADDI SP 1,
54     JUMP== GoTo
55       Name 'if_else_after.1';
56   ],
57 Block
58   Name 'if.2',
59   [
60     LOADIN BAF PC -1
61   ],
62 Block
63   Name 'if_else_after.1',
64   [
65     SUBI SP 1,
66     LOADIN DS ACC 0,
67     STOREIN SP ACC 1,
68     SUBI SP 1,
69     LOADIN DS ACC 1,
70     STOREIN SP ACC 1,
71     LOADIN SP ACC 2,
72     LOADIN SP IN2 1,
```

```

73      MULT ACC IN2,
74      STOREIN SP ACC 2,
75      ADDI SP 1,
76      LOADIN SP ACC 1,
77      STOREIN DS ACC 1,
78      ADDI SP 1,
79      SUBI SP 1,
80      LOADIN DS ACC 0,
81      STOREIN SP ACC 1,
82      SUBI SP 1,
83      LOADI ACC 1,
84      STOREIN SP ACC 1,
85      LOADIN SP ACC 2,
86      LOADIN SP IN2 1,
87      SUB ACC IN2,
88      STOREIN SP ACC 2,
89      ADDI SP 1,
90      LOADIN SP ACC 1,
91      STOREIN DS ACC 0,
92      ADDI SP 1,
93      Exp
94      GoTo
95      Name 'condition_check.4'
96  ],
97  Block
98      Name 'while_after.0',
99      [
100      LOADIN BAF PC -1
101      ]
102  ]

```

Code 3.11: RETI-Patch Pass für Codebeispiel

### 3.3.2.7 RETI Pass

#### 3.3.2.7.1 Konkrete und Abstrakte Syntax

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

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



```
16 LOADIN SP ACC 1;
17 ADDI SP 1;
18 JUMP== 49;
19 SUBI SP 1;
20 LOADIN DS ACC 0;
21 STOREIN SP ACC 1;
22 SUBI SP 1;
23 LOADI ACC 1;
24 STOREIN SP ACC 1;
25 LOADIN SP ACC 2;
26 LOADIN SP IN2 1;
27 SUB ACC IN2;
28 JUMP== 3;
29 LOADI ACC 0;
30 JUMP 2;
31 LOADI ACC 1;
32 STOREIN SP ACC 2;
33 ADDI SP 1;
34 LOADIN SP ACC 1;
35 ADDI SP 1;
36 JUMP== 2;
37 LOADIN BAF PC -1;
38 SUBI SP 1;
39 LOADIN DS ACC 0;
40 STOREIN SP ACC 1;
41 SUBI SP 1;
42 LOADIN DS ACC 1;
43 STOREIN SP ACC 1;
44 LOADIN SP ACC 2;
45 LOADIN SP IN2 1;
46 MULT ACC IN2;
47 STOREIN SP ACC 2;
48 ADDI SP 1;
49 LOADIN SP ACC 1;
50 STOREIN DS ACC 1;
51 ADDI SP 1;
52 SUBI SP 1;
53 LOADIN DS ACC 0;
54 STOREIN SP ACC 1;
55 SUBI SP 1;
56 LOADI ACC 1;
57 STOREIN SP ACC 1;
58 LOADIN SP ACC 2;
59 LOADIN SP IN2 1;
60 SUB ACC IN2;
61 STOREIN SP ACC 2;
62 ADDI SP 1;
63 LOADIN SP ACC 1;
64 STOREIN DS ACC 0;
65 ADDI SP 1;
66 JUMP -53;
67 LOADIN BAF PC -1;
```

Code 3.12: RETI Pass für Codebeispiel

### 3.3.3 Umsetzung von Pointern

#### 3.3.3.1 Referenzierung

```

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

```

Code 3.13: PicoC Code für Pointer Referenzierung

```

1 File
2   Name './example_pntr_ref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign
10          Alloc
11            Writeable,
12            IntType 'int',
13            Name 'var',
14            Num '42',
15          Assign
16            Alloc
17              Writeable,
18              PntrDecl
19                Num '1',
20                IntType 'int',
21                Name 'pntr',
22              Ref
23                Name 'var'
24            ]
25      ]

```

Code 3.14: Abstract Syntax Tree für Pointer Referenzierung

```

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

```

1 File
2   Name './example_pntr_ref.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         Exp
8           Num '42',
9         Assign
10          GlobalWrite
11            Num '0',
12          Tmp
13            Num '1',
14        Ref
15          GlobalRead
16            Num '0',
17        Assign
18          GlobalWrite
19            Num '1',
20          Tmp
21            Num '1',
22        Return
23          Empty
24      ]
25 ]

```

Code 3.16: PicoC Mon Pass für Pointer Referenzierung

```

1 File
2   Name './example_pntr_ref.reti_blocks',
3   [
4     Block
5       Name 'main.O',
6       [
7         SUBI SP 1,
8         LOADI ACC 42,
9         STOREIN SP ACC 1,
10        LOADIN SP ACC 1,
11        STOREIN DS ACC 0,
12        ADDI SP 1,
13        SUBI SP 1,
14        LOADI IN1 0,
15        ADD IN1 DS,
16        STOREIN SP IN1 1,
17        LOADIN SP ACC 1,
18        STOREIN DS ACC 1,
19        ADDI SP 1,
20        LOADIN BAF PC -1
21      ]
22    ]

```

Code 3.17: RETI Blocks Pass für Pointer Referenzierung

### 3.3.3.2 Pointer Dereferenzierung durch Zugriff auf Arrayindex ersetzen

```

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

```

Code 3.18: PicoC Code für Pointer Dereferenzierung

```

1 File
2   Name './example_pntr_deref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign
10          Alloc
11            Writeable,
12            IntType 'int',
13            Name 'var',
14            Num '42',
15          Assign
16            Alloc
17            Writeable,

```

```
18         PtrDecl
19         Num '1',
20         IntType 'int',
21         Name 'ptr',
22     Ref
23         Name 'var',
24     Exp
25         Deref
26         Name 'ptr',
27         Num '0'
28 ]
29 ]
```

Code 3.19: Abstract Syntax Tree für Pointer Dereferenzierung

```
1 File
2   Name './example_ptr_deref.picoc_shrink',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign
10          Alloc
11          Writeable,
12          IntType 'int',
13          Name 'var',
14          Num '42',
15        Assign
16          Alloc
17          Writeable,
18          PtrDecl
19          Num '1',
20          IntType 'int',
21          Name 'ptr',
22        Ref
23          Name 'var',
24      Exp
25        Subscr
26          Name 'ptr',
27          Num '0'
28    ]
29  ]
```

Code 3.20: PicoC Shrink Pass für Pointer Dereferenzierung

### 3.3.4 Umsetzung von Arrays

#### 3.3.4.1 Initialisierung von Arrays

```

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

```

Code 3.21: PicoC Code für Array Initialisierung

```

1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign
10          Alloc
11          Writeable,
12          ArrayDecl
13            [
14              Num '2',
15              Num '1'
16            ],
17          IntType 'int',
18          Name 'ar',
19          Array
20            [
21              Array
22                [
23                  Num '4'
24                ],
25              Array
26                [
27                  Num '2'
28                ]
29            ]
30          ]
31 ]

```

Code 3.22: Abstract Syntax Tree für Array Initialisierung

```

1 SymbolTable
2   [
3     Symbol(
4       {
5         type qualifier:      Empty()
6         datatype:            FunDecl(VoidType('void'), Name('main'), [])

```

```

7      name:                Name('main')
8      value or address:    Empty()
9      position:            Pos(Num('1'), Num('5'))
10     size:                Empty()
11  },
12  Symbol(
13  {
14      type qualifier:      Writeable()
15      datatype:            ArrayDecl([Num('2'), Num('1')], IntType('int'))
16      name:                Name('ar@main')
17      value or address:    Num('0')
18      position:            Pos(Num('2'), Num('6'))
19      size:                Num('2')
20  }
21 ]

```

Code 3.23: Symboltabelle für Array Initialisierung

```

1 File
2   Name './example_array_init.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         Exp
8           Num '4',
9         Exp
10          Num '2',
11        Assign
12          GlobalWrite
13            Num '0',
14          Tmp
15            Num '2',
16        Return
17          Empty
18      ]
19 ]

```

Code 3.24: PicoC Mon Pass für Array Initialisierung

```

1 File
2   Name './example_array_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         SUBI SP 1,
8         LOADI ACC 4,
9         STOREIN SP ACC 1,
10        SUBI SP 1,
11        LOADI ACC 2,

```

```

12     STOREIN SP ACC 1,
13     LOADIN SP ACC 1,
14     STOREIN DS ACC 1,
15     LOADIN SP ACC 2,
16     STOREIN DS ACC 0,
17     ADDI SP 2,
18     LOADIN BAF PC -1
19 ]
20 ]

```

Code 3.25: RETI Blocks Pass für Array Initialisierung

### 3.3.4.2 Zugriff auf Arrayindex

Der Zugriff auf einen bestimmten Index eines Arrays ist wie folgt umgesetzt:

```

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

```

Code 3.26: PicoC Code für Zugriff auf Arrayindex

```

1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign
10          Alloc
11            Writeable,
12            ArrayDecl
13              [
14                Num '2'
15              ],
16              IntType 'int',
17              Name 'ar',
18            Array
19              [
20                Num '1',
21                Num '2'
22              ],
23            Exp
24              Subscr
25                Name 'ar',
26                Num '2'
27          ]
28      ]

```

Code 3.27: Abstract Syntax Tree für Zugriff auf Arrayindex

```

1 File
2   Name './example_array_access.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         Exp
8           Num '1',
9         Exp
10          Num '2',
11        Assign
12          GlobalWrite
13            Num '0',
14          Tmp
15            Num '2',
16        Ref
17          GlobalRead
18            Num '0',
19        Exp
20          Num '2',
21        Ref
22          Subscr
23            Tmp
24              Num '2',
25            Tmp
26              Num '1',
27        Exp
28          Subscr
29            Tmp
30              Num '1',
31            Num '0',
32        Return
33          Empty
34      ]
35  ]

```

Code 3.28: PicoC Mon Pass für Zugriff auf Arrayindex

```

1 File
2   Name './example_array_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         SUBI SP 1,
8         LOADI ACC 1,
9         STOREIN SP ACC 1,
10        SUBI SP 1,
11        LOADI ACC 2,

```

```

12     STOREIN SP ACC 1,
13     LOADIN SP ACC 1,
14     STOREIN DS ACC 1,
15     LOADIN SP ACC 2,
16     STOREIN DS ACC 0,
17     ADDI SP 2,
18     SUBI SP 1,
19     LOADI IN1 0,
20     ADD IN1 DS,
21     STOREIN SP IN1 1,
22     SUBI SP 1,
23     LOADI ACC 2,
24     STOREIN SP ACC 1,
25     LOADIN SP IN1 2,
26     LOADIN SP IN2 1,
27     MULTI IN2 1,
28     ADD IN1 IN2,
29     ADDI SP 1,
30     STOREIN SP IN1 1,
31     LOADIN SP IN1 1,
32     LOADIN IN1 ACC 0,
33     STOREIN SP ACC 1,
34     LOADIN BAF PC -1
35 ]
36 ]

```

Code 3.29: RETI Blocks Pass für Zugriff auf Arrayindex

### 3.3.4.3 Zuweisung an Arrayindex

```

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

```

Code 3.30: PicoC Code für Zuweisung an Arrayindex

```

1 File
2   Name './example_array_assignment.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Exp
10          Alloc
11          Writeable,
12          ArrayDecl
13            [
14              Num '2'
15            ],

```



```

16         IntType 'int',
17         Name 'ar',
18     Assign
19         Subscr
20             Name 'ar',
21             Num '2',
22             Num '42'
23     ]
24 ]

```

Code 3.31: Abstract Syntax Tree für Zuweisung an Arrayindex

```

1 File
2   Name './example_array_assignment.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         Exp
8           Num '42',
9         Ref
10          GlobalRead
11            Num '0',
12        Exp
13          Num '2',
14        Ref
15          Subscr
16            Tmp
17              Num '2',
18            Tmp
19              Num '1',
20        Assign
21          Subscr
22            Tmp
23              Num '1',
24            Num '0',
25          Tmp
26            Num '2',
27        Return
28          Empty
29      ]
30  ]

```

Code 3.32: PicoC Mon Pass für Zuweisung an Arrayindex

```

1 File
2   Name './example_array_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [

```

```

7      SUBI SP 1,
8      LOADI ACC 42,
9      STOREIN SP ACC 1,
10     SUBI SP 1,
11     LOADI IN1 0,
12     ADD IN1 DS,
13     STOREIN SP IN1 1,
14     SUBI SP 1,
15     LOADI ACC 2,
16     STOREIN SP ACC 1,
17     LOADIN SP IN1 2,
18     LOADIN SP IN2 1,
19     MULTI IN2 1,
20     ADD IN1 IN2,
21     ADDI SP 1,
22     STOREIN SP IN1 1,
23     LOADIN SP IN1 1,
24     LOADIN SP ACC 2,
25     ADDI SP 2,
26     STOREIN IN1 ACC 0,
27     LOADIN BAF PC -1
28 ]
29 ]

```

Code 3.33: RETI Blocks Pass für Zuweisung an Arrayindex

### 3.3.5 Umsetzung von Structs

#### 3.3.5.1 Deklaration von Structs

```

1 struct st1 {int *ar[3];};
2
3 struct st2 {struct st1 st;};
4
5 void main() {
6 }

```

Code 3.34: PicoC Code für Deklaration von Structs

```

1 SymbolTable
2 [
3     Symbol(
4         {
5             type qualifier:    Empty()
6             datatype:          ArrayDecl([Num('3')], PtrDecl(Num('1'), IntType('int')))
7             name:               Name('ar@st1')
8             value or address:   Empty()
9             position:           Pos(Num('1'), Num('17'))
10            size:                Num('3')
11        },
12    Symbol(

```

```

13 {
14   type qualifier:      Empty()
15   datatype:            StructDecl(Name('st1'), [Alloc(Writable(),
16     ↪ ArrayDecl([Num('3')], PtrDecl(Num('1'), IntType('int'))), Name('ar'))])
17   name:                Name('st1')
18   value or address:    [Name('ar@st1')]
19   position:            Pos(Num('1'), Num('7'))
20   size:                Num('3')
21 },
22 Symbol(
23 {
24   type qualifier:      Empty()
25   datatype:            StructSpec(Name('st1'))
26   name:                Name('st@st2')
27   value or address:    Empty()
28   position:            Pos(Num('3'), Num('23'))
29   size:                Num('3')
30 },
31 Symbol(
32 {
33   type qualifier:      Empty()
34   datatype:            StructDecl(Name('st2'), [Alloc(Writable(),
35     ↪ StructSpec(Name('st1')), Name('st'))])
36   name:                Name('st2')
37   value or address:    [Name('st@st2')]
38   position:            Pos(Num('3'), Num('7'))
39   size:                Num('3')
40 },
41 Symbol(
42 {
43   type qualifier:      Empty()
44   datatype:            FunDecl(VoidType('void'), Name('main'), [])
45   name:                Name('main')
46   value or address:    Empty()
47   position:            Pos(Num('5'), Num('5'))
48   size:                Empty()
49 }
50 ]

```

Code 3.35: Symboltabelle für Deklaration von Structs

### 3.3.5.2 Initialisierung von Structs

```

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

```

Code 3.36: PicoC Code für Initialisierung von Structs

```
1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc
8           Writeable,
9           ArrayDecl
10            [
11              Num '1'
12            ],
13            PtrDecl
14              Num '1',
15              IntType 'int',
16            Name 'pntr'
17          ],
18      StructDecl
19        Name 'st2',
20        [
21          Alloc
22            Writeable,
23            StructSpec
24              Name 'st1',
25              Name 'st'
26          ],
27      FunDef
28        VoidType 'void',
29        Name 'main',
30        [],
31        [
32          Assign
33            Alloc
34              Writeable,
35              IntType 'int',
36              Name 'var',
37              Num '42',
38          Assign
39            Alloc
40              Writeable,
41              StructSpec
42                Name 'st1',
43                Name 'st',
44              Struct
45                [
46                  Assign
47                    Name 'st',
48                    Struct
49                      [
50                        Assign
51                          Name 'pntr',
52                          Array
53                            [
54                              Array
55                                [
56                                  Ref
57                                    Name 'var'
```

```

58         ]
59     ]
60 ]
61 ]
62 ]
63 ]

```

Code 3.37: Abstract Syntax Tree für Initialisierung von Structs

```

1 SymbolTable
2 [
3     Symbol(
4         {
5             type qualifier:      Empty()
6             datatype:            ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int')))
7             name:                Name('ptr@st1')
8             value or address:    Empty()
9             position:            Pos(Num('1'), Num('17'))
10            size:                Num('1')
11        },
12     Symbol(
13         {
14             type qualifier:      Empty()
15             datatype:            StructDecl(Name('st1'), [Alloc(Writeable(),
16             ↪ ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int'))), Name('ptr'))])
17             name:                Name('st1')
18             value or address:    [Name('ptr@st1')]
19             position:            Pos(Num('1'), Num('7'))
20             size:                Num('1')
21        },
22     Symbol(
23         {
24             type qualifier:      Empty()
25             datatype:            StructSpec(Name('st1'))
26             name:                Name('st@st2')
27             value or address:    Empty()
28             position:            Pos(Num('3'), Num('23'))
29             size:                Num('1')
30        },
31     Symbol(
32         {
33             type qualifier:      Empty()
34             datatype:            StructDecl(Name('st2'), [Alloc(Writeable(),
35             ↪ StructSpec(Name('st1')), Name('st'))])
36             name:                Name('st2')
37             value or address:    [Name('st@st2')]
38             position:            Pos(Num('3'), Num('7'))
39             size:                Num('1')
40        },
41     Symbol(
42         {
43             type qualifier:      Empty()
44             datatype:            FunDecl(VoidType('void'), Name('main'), [])
45             name:                Name('main')

```

```

44     value or address:    Empty()
45     position:           Pos(Num('5'), Num('5'))
46     size:               Empty()
47 },
48 Symbol(
49 {
50     type qualifier:      Writeable()
51     datatype:            IntType('int')
52     name:                Name('var@main')
53     value or address:    Num('0')
54     position:            Pos(Num('6'), Num('6'))
55     size:                Num('1')
56 },
57 Symbol(
58 {
59     type qualifier:      Writeable()
60     datatype:            StructSpec(Name('st1'))
61     name:                Name('st@main')
62     value or address:    Num('1')
63     position:            Pos(Num('7'), Num('13'))
64     size:                Num('1')
65 }
66 ]

```

Code 3.38: Symboltabelle für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.picoc_mon',
3   [
4     Block
5     Name 'main.0',
6     [
7       Exp
8       Num '42',
9       Assign
10      GlobalWrite
11      Num '0',
12      Tmp
13      Num '1',
14      Ref
15      GlobalRead
16      Num '0',
17      Assign
18      GlobalWrite
19      Num '1',
20      Tmp
21      Num '1',
22      Return
23      Empty
24    ]
25  ]

```

Code 3.39: PicoC Mon Pass für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         SUBI SP 1,
8         LOADI ACC 42,
9         STOREIN SP ACC 1,
10        LOADIN SP ACC 1,
11        STOREIN DS ACC 0,
12        ADDI SP 1,
13        SUBI SP 1,
14        LOADI IN1 0,
15        ADD IN1 DS,
16        STOREIN SP IN1 1,
17        LOADIN SP ACC 1,
18        STOREIN DS ACC 1,
19        ADDI SP 1,
20        LOADIN BAF PC -1
21      ]
22    ]

```

Code 3.40: RETI Blocks Pass für Initialisierung von Structs

### 3.3.5.3 Zugriff auf Structattribut

```

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

```

1 File
2   Name './example_struct_attr_access.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'x',
11        Alloc
12          Writeable,
13          IntType 'int',
14          Name 'y'

```

```

15 ],
16 FunDef
17   VoidType 'void',
18   Name 'main',
19   [],
20   [
21     Assign
22       Alloc
23       Writeable,
24       StructSpec
25         Name 'pos',
26         Name 'st',
27       Struct
28         [
29           Assign
30             Name 'x',
31             Num '4',
32           Assign
33             Name 'y',
34             Num '2'
35         ],
36       Exp
37         Attr
38           Name 'st',
39           Name 'y'
40     ]
41 ]

```

Code 3.42: Abstract Syntax Tree für Zugriff auf Structattribut

```

1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         Exp
8           Num '4',
9         Exp
10          Num '2',
11        Assign
12          GlobalWrite
13            Num '0',
14          Tmp
15            Num '2',
16        Ref
17          GlobalRead
18            Num '0',
19        Ref
20          Attr
21            Tmp
22              Num '1',
23              Name 'y',
24        Exp

```



```

25     Subscr
26     Tmp
27     Num '1',
28     Num '0',
29     Return
30     Empty
31 ]
32 ]

```

Code 3.43: PicoC Mon Pass für Zugriff auf Structattribut

```

1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5     Name 'main.0',
6     [
7       SUBI SP 1,
8       LOADI ACC 4,
9       STOREIN SP ACC 1,
10      SUBI SP 1,
11      LOADI ACC 2,
12      STOREIN SP ACC 1,
13      LOADIN SP ACC 1,
14      STOREIN DS ACC 1,
15      LOADIN SP ACC 2,
16      STOREIN DS ACC 0,
17      ADDI SP 2,
18      SUBI SP 1,
19      LOADI IN1 0,
20      ADD IN1 DS,
21      STOREIN SP IN1 1,
22      LOADIN SP IN1 1,
23      ADDI IN1 1,
24      STOREIN SP IN1 1,
25      LOADIN SP IN1 1,
26      LOADIN IN1 ACC 0,
27      STOREIN SP ACC 1,
28      LOADIN BAF PC -1
29    ]
30  ]

```

Code 3.44: RETI Blocks Pass für Zugriff auf Structattribut

### 3.3.5.4 Zuweisung an Structattribut

```

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

```

1 File
2   Name './example_struct_attr_assignment.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'x',
11        Alloc
12          Writeable,
13          IntType 'int',
14          Name 'y'
15      ],
16    FunDef
17      VoidType 'void',
18      Name 'main',
19      [],
20      [
21        Assign
22          Alloc
23            Writeable,
24            StructSpec
25              Name 'pos',
26              Name 'st',
27            Struct
28              [
29                Assign
30                  Name 'x',
31                  Num '4',
32                Assign
33                  Name 'y',
34                  Num '2'
35              ],
36            Assign
37              Attr
38                Name 'st',
39                Name 'y',
40                Num '42'
41          ]
42      ]

```

Code 3.46: Abstract Syntax Tree für Zuweisung an Structattribut

```

1 File
2   Name './example_struct_attr_assignment.picoc_mon',
3   [
4     Block

```

```

5      Name 'main.0',
6      [
7          Exp
8              Num '4',
9          Exp
10             Num '2',
11         Assign
12             GlobalWrite
13                 Num '0',
14             Tmp
15                 Num '2',
16         Exp
17             Num '42',
18         Ref
19             GlobalRead
20                 Num '0',
21         Ref
22             Attr
23                 Tmp
24                     Num '1',
25                 Name 'y',
26         Assign
27             Subscr
28                 Tmp
29                     Num '1',
30                 Num '0',
31             Tmp
32                 Num '2',
33         Return
34             Empty
35     ]
36 ]

```

Code 3.47: PicoC Mon Pass für Zuweisung an Structattribut

```

1 File
2     Name './example_struct_attr_assignment.reti_blocks',
3     [
4         Block
5             Name 'main.0',
6             [
7                 SUBI SP 1,
8                 LOADI ACC 4,
9                 STOREIN SP ACC 1,
10                SUBI SP 1,
11                LOADI ACC 2,
12                STOREIN SP ACC 1,
13                LOADIN SP ACC 1,
14                STOREIN DS ACC 1,
15                LOADIN SP ACC 2,
16                STOREIN DS ACC 0,
17                ADDI SP 2,
18                SUBI SP 1,
19                LOADI ACC 42,

```

```

20     STOREIN SP ACC 1,
21     SUBI SP 1,
22     LOADI IN1 0,
23     ADD IN1 DS,
24     STOREIN SP IN1 1,
25     LOADIN SP IN1 1,
26     ADDI IN1 1,
27     STOREIN SP IN1 1,
28     LOADIN SP IN1 1,
29     LOADIN SP ACC 2,
30     ADDI SP 2,
31     STOREIN IN1 ACC 0,
32     LOADIN BAF PC -1
33 ]
34 ]

```

Code 3.48: RETI Blocks Pass für Zuweisung an Structattribut

### 3.3.6 Umsetzung der Derived Datatypes im Zusammenspiel

#### 3.3.6.1 Einleitungsteil für Globale Statische Daten und Stackframe

```

1 struct ar_with_len {int len; int ar[2];};
2
3 void main() {
4     struct ar_with_len st_ar[3];
5     int (*pntr2)[3];
6     pntr2;
7 }
8
9 void fun() {
10    struct ar_with_len st_ar[3];
11    int (*pntr1)[3];
12    pntr1;
13 }

```

Code 3.49: PicoC Code für den Einleitungsteil

```

1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     StructDecl
5       Name 'ar_with_len',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'len',
11        Alloc
12          Writeable,
13        ArrayDecl

```

```
14      [
15          Num '2'
16      ],
17      IntType 'int',
18      Name 'ar'
19  ],
20  FunDef
21      VoidType 'void',
22      Name 'main',
23      [],
24      [
25          Exp
26              Alloc
27                  Writeable,
28                  ArrayDecl
29                      [
30                          Num '3'
31                      ],
32                  StructSpec
33                      Name 'ar_with_len',
34                      Name 'st_ar',
35          Exp
36              Alloc
37                  Writeable,
38                  PtrDecl
39                      Num '1',
40                  ArrayDecl
41                      [
42                          Num '3'
43                      ],
44                  PtrDecl
45                      Num '1',
46                      IntType 'int',
47                  Name 'ptr2',
48          Exp
49              Name 'ptr2'
50      ],
51  FunDef
52      VoidType 'void',
53      Name 'fun',
54      [],
55      [
56          Exp
57              Alloc
58                  Writeable,
59                  ArrayDecl
60                      [
61                          Num '3'
62                      ],
63                  StructSpec
64                      Name 'ar_with_len',
65                      Name 'st_ar',
66          Exp
67              Alloc
68                  Writeable,
69                  PtrDecl
70                      Num '1',
```

```

71         ArrayDecl
72         [
73             Num '3'
74         ],
75         IntType 'int',
76         Name 'pntr1',
77     Exp
78     Name 'pntr1'
79 ]
80 ]

```

Code 3.50: Abstract Syntax Tree für den Einleitungsteil

```

1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4       Block
5         Name 'main.1',
6         [
7             Exp
8             GlobalRead
9             Num '9',
10            Return
11            Empty
12        ],
13        Block
14        Name 'fun.0',
15        [
16            Exp
17            StackRead
18            Num '9',
19            Return
20            Empty
21        ]
22    ]

```

Code 3.51: PicoC Mon Pass für den Einleitungsteil

```

1 File
2   Name './example_derived_dts_introduction_part.reti_blocks',
3   [
4       Block
5         Name 'main.1',
6         [
7             SUBI SP 1,
8             LOADIN DS ACC 9,
9             STOREIN SP ACC 1,
10            LOADIN BAF PC -1
11        ],
12        Block
13        Name 'fun.0',

```

```

14     [
15         SUBI SP 1,
16         LOADIN BAF ACC -11,
17         STOREIN SP ACC 1,
18         LOADIN BAF PC -1
19     ]
20 ]

```

Code 3.52: RETI Blocks Pass für den Einleitungsteil

### 3.3.6.2 Mittelteil für die verschiedenen Derived Datatypes

```

1 struct st1 {int (*ar)[1]};;
2
3 void main() {
4     int var[1] = {42};
5     struct st1 st_first = {.ar=&var};
6     (*st_first.ar)[0];
7 }

```

Code 3.53: PicoC Code für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc
8           Writeable,
9           PtrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '1'
14              ],
15            IntType 'int',
16            Name 'ar'
17          ],
18      FunDef
19        VoidType 'void',
20        Name 'main',
21        [],
22        [
23          Assign
24            Alloc
25              Writeable,
26              ArrayDecl
27                [
28                  Num '1'
29                ],
30              IntType 'int',

```

```

31     Name 'var',
32   Array
33   [
34     Num '42'
35   ],
36   Assign
37   Alloc
38   Writeable,
39   StructSpec
40   Name 'st1',
41   Name 'st_first',
42   Struct
43   [
44     Assign
45     Name 'ar',
46     Ref
47     Name 'var'
48   ],
49   Exp
50   Subscr
51   Deref
52   Attr
53   Name 'st_first',
54   Name 'ar',
55   Num '0',
56   Num '0'
57 ]
58 ]

```

Code 3.54: Abstract Syntax Tree für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.picoc_mon',
3   [
4     Block
5     Name 'main.0',
6     [
7       Exp
8       Num '42',
9       Assign
10      GlobalWrite
11      Num '0',
12      Tmp
13      Num '1',
14      Ref
15      GlobalRead
16      Num '0',
17      Assign
18      GlobalWrite
19      Num '1',
20      Tmp
21      Num '1',
22      Ref
23      GlobalRead

```



```

24     Num '1',
25   Ref
26     Attr
27       Tmp
28       Num '1',
29       Name 'ar',
30   Exp
31     Num '0',
32   Ref
33     Subscr
34       Tmp
35       Num '2',
36       Tmp
37       Num '1',
38   Exp
39     Num '0',
40   Ref
41     Subscr
42       Tmp
43       Num '2',
44       Tmp
45       Num '1',
46   Exp
47     Subscr
48       Tmp
49       Num '1',
50       Num '0',
51   Return
52     Empty
53 ]
54 ]

```

Code 3.55: PicoC Mon Pass für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         SUBI SP 1,
8         LOADI ACC 42,
9         STOREIN SP ACC 1,
10        LOADIN SP ACC 1,
11        STOREIN DS ACC 0,
12        ADDI SP 1,
13        SUBI SP 1,
14        LOADI IN1 0,
15        ADD IN1 DS,
16        STOREIN SP IN1 1,
17        LOADIN SP ACC 1,
18        STOREIN DS ACC 1,
19        ADDI SP 1,
20        SUBI SP 1,

```

```

21     LOADI IN1 1,
22     ADD IN1 DS,
23     STOREIN SP IN1 1,
24     LOADIN SP IN1 1,
25     ADDI IN1 0,
26     STOREIN SP IN1 1,
27     SUBI SP 1,
28     LOADI ACC 0,
29     STOREIN SP ACC 1,
30     LOADIN SP IN2 2,
31     LOADIN IN2 IN1 0,
32     LOADIN SP IN2 1,
33     MULTI IN2 1,
34     ADD IN1 IN2,
35     ADDI SP 1,
36     STOREIN SP IN1 1,
37     SUBI SP 1,
38     LOADI ACC 0,
39     STOREIN SP ACC 1,
40     LOADIN SP IN1 2,
41     LOADIN SP IN2 1,
42     MULTI IN2 1,
43     ADD IN1 IN2,
44     ADDI SP 1,
45     STOREIN SP IN1 1,
46     LOADIN SP IN1 1,
47     LOADIN IN1 ACC 0,
48     STOREIN SP ACC 1,
49     LOADIN BAF PC -1
50 ]
51 ]

```

Code 3.56: RETI Blocks Pass für den Mittelteil

### 3.3.6.3 Schlussteil für die verschiedenen Derived Datatypes

```

1 struct st {int attr[2];};
2
3 void main() {
4     int ar1[1][2] = {{42, 314}};
5     struct st ar2[1] = {.attr={42, 314}};
6     int var = 42;
7     int *pntr1 = &var;
8     int **pntr2 = &pntr1;
9
10    ar1[0];
11    ar2[0];
12    *pntr2;
13 }

```

Code 3.57: PicoC Code für den Schlussteil

```
1 File
2   Name './example_derived_dts_final_part.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           ArrayDecl
10            [
11              Num '2'
12            ],
13            IntType 'int',
14            Name 'attr'
15        ],
16      FunDef
17        VoidType 'void',
18        Name 'main',
19        [],
20        [
21          Assign
22            Alloc
23              Writeable,
24              ArrayDecl
25                [
26                  Num '1',
27                  Num '2'
28                ],
29                IntType 'int',
30                Name 'ar1',
31              Array
32                [
33                  Array
34                    [
35                      Num '42',
36                      Num '314'
37                    ]
38                ],
39            Assign
40              Alloc
41                Writeable,
42                ArrayDecl
43                  [
44                    Num '1'
45                  ],
46                StructSpec
47                  Name 'st',
48                  Name 'ar2',
49              Struct
50                [
51                  Assign
52                    Name 'attr',
53                    Array
54                      [
55                        Num '42',
56                        Num '314'
57                      ]
58                ]
59        ]
60      ]
61    ]
62  ]
```

```

58     ],
59     Assign
60     Alloc
61     Writeable,
62     IntType 'int',
63     Name 'var',
64     Num '42',
65     Assign
66     Alloc
67     Writeable,
68     PtrDecl
69     Num '1',
70     IntType 'int',
71     Name 'ptr1',
72     Ref
73     Name 'var',
74     Assign
75     Alloc
76     Writeable,
77     PtrDecl
78     Num '2',
79     IntType 'int',
80     Name 'ptr2',
81     Ref
82     Name 'ptr1',
83     Exp
84     Subscr
85     Name 'ar1',
86     Num '0',
87     Exp
88     Subscr
89     Name 'ar2',
90     Num '0',
91     Exp
92     Deref
93     Name 'ptr2',
94     Num '0'
95 ]
96 ]

```

Code 3.58: Abstract Syntax Tree für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5     Name 'main.0',
6     [
7       Exp
8       Num '42',
9       Exp
10      Num '314',
11      Assign
12      GlobalWrite

```

```
13         Num '0',
14         Tmp
15         Num '2',
16     Exp
17         Num '42',
18     Exp
19         Num '314',
20     Assign
21         GlobalWrite
22         Num '2',
23         Tmp
24         Num '2',
25     Exp
26         Num '42',
27     Assign
28         GlobalWrite
29         Num '4',
30         Tmp
31         Num '1',
32     Ref
33         GlobalRead
34         Num '4',
35     Assign
36         GlobalWrite
37         Num '5',
38         Tmp
39         Num '1',
40     Ref
41         GlobalRead
42         Num '5',
43     Assign
44         GlobalWrite
45         Num '6',
46         Tmp
47         Num '1',
48     Ref
49         GlobalRead
50         Num '0',
51     Exp
52         Num '0',
53     Ref
54         Subscr
55         Tmp
56         Num '2',
57         Tmp
58         Num '1',
59     Exp
60         Subscr
61         Tmp
62         Num '1',
63         Num '0',
64     Ref
65         GlobalRead
66         Num '2',
67     Exp
68         Num '0',
69     Ref
```

```

70     Subscr
71     Tmp
72     Num '2',
73     Tmp
74     Num '1',
75     Exp
76     Subscr
77     Tmp
78     Num '1',
79     Num '0',
80     Ref
81     GlobalRead
82     Num '6',
83     Exp
84     Num '0',
85     Ref
86     Subscr
87     Tmp
88     Num '2',
89     Tmp
90     Num '1',
91     Exp
92     Subscr
93     Tmp
94     Num '1',
95     Num '0',
96     Return
97     Empty
98 ]
99 ]

```

Code 3.59: PicoC Mon Pass für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5     Name 'main.0',
6     [
7       SUBI SP 1,
8       LOADI ACC 42,
9       STOREIN SP ACC 1,
10      SUBI SP 1,
11      LOADI ACC 314,
12      STOREIN SP ACC 1,
13      LOADIN SP ACC 1,
14      STOREIN DS ACC 1,
15      LOADIN SP ACC 2,
16      STOREIN DS ACC 0,
17      ADDI SP 2,
18      SUBI SP 1,
19      LOADI ACC 42,
20      STOREIN SP ACC 1,
21      SUBI SP 1,

```

```
22      LOADI ACC 314,  
23      STOREIN SP ACC 1,  
24      LOADIN SP ACC 1,  
25      STOREIN DS ACC 3,  
26      LOADIN SP ACC 2,  
27      STOREIN DS ACC 2,  
28      ADDI SP 2,  
29      SUBI SP 1,  
30      LOADI ACC 42,  
31      STOREIN SP ACC 1,  
32      LOADIN SP ACC 1,  
33      STOREIN DS ACC 4,  
34      ADDI SP 1,  
35      SUBI SP 1,  
36      LOADI IN1 4,  
37      ADD IN1 DS,  
38      STOREIN SP IN1 1,  
39      LOADIN SP ACC 1,  
40      STOREIN DS ACC 5,  
41      ADDI SP 1,  
42      SUBI SP 1,  
43      LOADI IN1 5,  
44      ADD IN1 DS,  
45      STOREIN SP IN1 1,  
46      LOADIN SP ACC 1,  
47      STOREIN DS ACC 6,  
48      ADDI SP 1,  
49      SUBI SP 1,  
50      LOADI IN1 0,  
51      ADD IN1 DS,  
52      STOREIN SP IN1 1,  
53      SUBI SP 1,  
54      LOADI ACC 0,  
55      STOREIN SP ACC 1,  
56      LOADIN SP IN1 2,  
57      LOADIN SP IN2 1,  
58      MULTI IN2 2,  
59      ADD IN1 IN2,  
60      ADDI SP 1,  
61      STOREIN SP IN1 1,  
62      SUBI SP 1,  
63      LOADI IN1 2,  
64      ADD IN1 DS,  
65      STOREIN SP IN1 1,  
66      SUBI SP 1,  
67      LOADI ACC 0,  
68      STOREIN SP ACC 1,  
69      LOADIN SP IN1 2,  
70      LOADIN SP IN2 1,  
71      MULTI IN2 2,  
72      ADD IN1 IN2,  
73      ADDI SP 1,  
74      STOREIN SP IN1 1,  
75      LOADIN SP IN1 1,  
76      LOADIN IN1 ACC 0,  
77      STOREIN SP ACC 1,  
78      SUBI SP 1,
```

```

79     LOADI IN1 6,
80     ADD IN1 DS,
81     STOREIN SP IN1 1,
82     SUBI SP 1,
83     LOADI ACC 0,
84     STOREIN SP ACC 1,
85     LOADIN SP IN2 2,
86     LOADIN IN2 IN1 0,
87     LOADIN SP IN2 1,
88     MULTI IN2 1,
89     ADD IN1 IN2,
90     ADDI SP 1,
91     STOREIN SP IN1 1,
92     LOADIN BAF PC -1
93 ]
94 ]

```

Code 3.60: RETI Blocks Pass für den Schlussteil

### 3.3.7 Umsetzung von Funktionen

#### 3.3.7.1 Funktionen auflösen zu RETI Code

```

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

```

1 File
2   Name './example_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

```
1 File
2   Name './example_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.2',
11          [
12            Return
13              Empty
14          ]
15        ],
16        FunDef
17          VoidType 'void',
18          Name 'fun1',
19          [],
20          [
21            Block
22              Name 'fun1.1',
23              []
24          ],
25        FunDef
26          IntType 'int',
27          Name 'fun2',
28          [],
29          [
30            Block
31              Name 'fun2.0',
32              [
33                Return
34                  Num '1'
35              ]
36          ]
37    ]
```

Code 3.63: PicoC Blocks Pass für 3 Funktionen

```
1 File
2   Name './example_3_funs.picoc_mon',
3   [
4     Block
5       Name 'main.2',
6       [
7         Return
8         Empty
9       ],
10    Block
11      Name 'fun1.1',
12      [
13        Return
14        Empty
15      ],
16    Block
17      Name 'fun2.0',
18      [
19        Exp
20        Num '1',
21        Return
22        Tmp
23        Num '1'
24      ]
25  ]
```

Code 3.64: PicoC Mon Pass für 3 Funktionen

```
1 File
2   Name './example_3_funs.reti_blocks',
3   [
4     Block
5       Name 'main.2',
6       [
7         LOADIN BAF PC -1
8       ],
9     Block
10      Name 'fun1.1',
11      [
12        LOADIN BAF PC -1
13      ],
14    Block
15      Name 'fun2.0',
16      [
17        SUBI SP 1,
18        LOADI ACC 1,
19        STOREIN SP ACC 1,
20        LOADIN SP ACC 1,
21        ADDI SP 1,
22        LOADIN BAF PC -1
23      ]
24  ]
```

Code 3.65: RETI Blocks Pass für 3 Funktionen

### 3.3.7.1.1 Sprung zur Main Funktion

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

Code 3.66: PicoC Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
1 File
2   Name './example_3_funs_main.picoc_mon',
3   [
4     Block
5       Name 'fun1.2',
6       [
7         Return
8         Empty
9       ],
10    Block
11      Name 'fun2.1',
12      [
13        Exp
14          Num '1',
15        Return
16          Tmp
17          Num '1'
18      ],
19    Block
20      Name 'main.0',
21      [
22        Return
23        Empty
24      ]
25  ]
```

Code 3.67: PicoC Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
1 File
2   Name './example_3_funs_main.reti_blocks',
3   [
4     Block
5       Name 'fun1.2',
6       [
7         LOADIN BAF PC -1
```

```

8      ],
9      Block
10     Name 'fun2.1',
11     [
12         SUBI SP 1,
13         LOADI ACC 1,
14         STOREIN SP ACC 1,
15         LOADIN SP ACC 1,
16         ADDI SP 1,
17         LOADIN BAF PC -1
18     ],
19     Block
20     Name 'main.0',
21     [
22         LOADIN BAF PC -1
23     ]
24 ]

```

Code 3.68: PicoC Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```

1 File
2   Name './example_3_funs_main.reti_patch',
3   [
4       Block
5       Name 'start.3',
6       [
7           Exp
8           GoTo
9           Name 'main.0'
10      ],
11      Block
12      Name 'fun1.2',
13      [
14          LOADIN BAF PC -1
15      ],
16      Block
17      Name 'fun2.1',
18      [
19          SUBI SP 1,
20          LOADI ACC 1,
21          STOREIN SP ACC 1,
22          LOADIN SP ACC 1,
23          ADDI SP 1,
24          LOADIN BAF PC -1
25      ],
26      Block
27      Name 'main.0',
28      [
29          LOADIN BAF PC -1
30      ]
31 ]

```

Code 3.69: PicoC Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

**3.3.7.2 Funktionsdeklaration und -definition**

```

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

```

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

```

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

#### 3.3.7.3.1 Ohne Rückgabewert

```

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

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc
8           Num '2',
9         Ref
10          GlobalRead
11            Num '0',
12          NewStackframe
13            Name 'stack_fun',
14            GoTo
15              Name 'addr@next_instr',

```

```

16     Exp
17     GoTo
18         Name 'stack_fun.0',
19     RemoveStackframe,
20     Return
21     Empty
22 ],
23 Block
24     Name 'stack_fun.0',
25     [
26         Return
27         Empty
28     ]
29 ]

```

Code 3.73: PicoC Mon Pass für Funktionsaufruf ohne Rückgabewert

```

1 File
2     Name './example_fun_call_no_return_value.reti_blocks',
3     [
4         Block
5             Name 'main.1',
6             [
7                 SUBI SP 2,
8                 SUBI SP 1,
9                 LOADI IN1 0,
10                ADD IN1 DS,
11                STOREIN SP IN1 1,
12                MOVE BAF ACC,
13                ADDI SP 3,
14                MOVE SP BAF,
15                SUBI SP 4,
16                STOREIN BAF ACC 0,
17                LOADI ACC GoTo
18                Name 'addr@next_instr',
19                ADD ACC CS,
20                STOREIN BAF ACC -1,
21            Exp
22            GoTo
23                Name 'stack_fun.0',
24            MOVE BAF IN1,
25            LOADIN IN1 BAF 0,
26            MOVE IN1 SP,
27            LOADIN BAF PC -1
28        ],
29        Block
30            Name 'stack_fun.0',
31            [
32                LOADIN BAF PC -1
33            ]
34    ]

```

Code 3.74: RETI Blocks Pass für Funktionsaufruf ohne Rückgabewert

```

1 SUBI SP 2;
2 SUBI SP 1;
3 LOADI IN1 0;
4 ADD IN1 DS;
5 STOREIN SP IN1 1;
6 MOVE BAF ACC;
7 ADDI SP 3;
8 MOVE SP BAF;
9 SUBI SP 4;
10 STOREIN BAF ACC 0;
11 LOADI ACC 14;
12 ADD ACC CS;
13 STOREIN BAF ACC -1;
14 JUMP 5;
15 MOVE BAF IN1;
16 LOADIN IN1 BAF 0;
17 MOVE IN1 SP;
18 LOADIN BAF PC -1;
19 LOADIN BAF PC -1;

```

Code 3.75: RETI Pass für Funktionsaufruf ohne Rückgabewert

### 3.3.7.3.2 Mit Rückgabewert

```

1 void stack_fun() {
2     return 42;
3 }
4
5 void main() {
6     int var = stack_fun();
7 }

```

Code 3.76: PicoC Code für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Exp
8           Num '42',
9         Return
10          Tmp
11            Num '1'
12        ],
13     Block
14       Name 'main.0',
15     [

```



```

16      StackMalloc
17      Num '2',
18      NewStackframe
19      Name 'stack_fun',
20      GoTo
21      Name 'addr@next_instr',
22      Exp
23      GoTo
24      Name 'stack_fun.1',
25      RemoveStackframe,
26      Assign
27      GlobalWrite
28      Num '0',
29      Tmp
30      Num '1',
31      Return
32      Empty
33  ]
34 ]

```

Code 3.77: PicoC Mon Pass für Funktionsaufruf mit Rückgabewert

```

1 File
2 Name './example_fun_call_with_return_value.reti_blocks',
3 [
4   Block
5     Name 'stack_fun.1',
6     [
7       SUBI SP 1,
8       LOADI ACC 42,
9       STOREIN SP ACC 1,
10      LOADIN SP ACC 1,
11      ADDI SP 1,
12      LOADIN BAF PC -1
13    ],
14   Block
15     Name 'main.0',
16     [
17       SUBI SP 2,
18       MOVE BAF ACC,
19       ADDI SP 2,
20       MOVE SP BAF,
21       SUBI SP 2,
22       STOREIN BAF ACC 0,
23       LOADI ACC GoTo
24       Name 'addr@next_instr',
25       ADD ACC CS,
26       STOREIN BAF ACC -1,
27       Exp
28       GoTo
29       Name 'stack_fun.1',
30       MOVE BAF IN1,
31       LOADIN IN1 BAF 0,
32       MOVE IN1 SP,

```

```

33     LOADIN SP ACC 1,
34     STOREIN DS ACC 0,
35     ADDI SP 1,
36     LOADIN BAF PC -1
37 ]
38 ]

```

Code 3.78: RETI Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1  JUMP 7;
2  SUBI SP 1;
3  LOADI ACC 42;
4  STOREIN SP ACC 1;
5  LOADIN SP ACC 1;
6  ADDI SP 1;
7  LOADIN BAF PC -1;
8  SUBI SP 2;
9  MOVE BAF ACC;
10 ADDI SP 2;
11 MOVE SP BAF;
12 SUBI SP 2;
13 STOREIN BAF ACC 0;
14 LOADI ACC 17;
15 ADD ACC CS;
16 STOREIN BAF ACC -1;
17 JUMP -15;
18 MOVE BAF IN1;
19 LOADIN IN1 BAF 0;
20 MOVE IN1 SP;
21 LOADIN SP ACC 1;
22 STOREIN DS ACC 0;
23 ADDI SP 1;
24 LOADIN BAF PC -1;

```

Code 3.79: RETI Pass für Funktionsaufruf mit Rückgabewert

### 3.3.7.3.3 Umsetzung von Call by Sharing für Arrays

```

1 void stack_fun(int (*param1)[3], int param2[2][3]) {
2 }
3
4 void main() {
5     int local_var1[2][3];
6     int local_var2[2][3];
7     stack_fun(local_var1, local_var2);
8 }

```

Code 3.80: PicoC Code für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Return
8         Empty
9       ],
10    Block
11      Name 'main.0',
12      [
13        StackMalloc
14        Num '2',
15        Ref
16        GlobalRead
17        Num '0',
18        Ref
19        GlobalRead
20        Num '6',
21        NewStackframe
22        Name 'stack_fun',
23        GoTo
24        Name 'addr@next_instr',
25        Exp
26        GoTo
27        Name 'stack_fun.1',
28        RemoveStackframe,
29        Return
30        Empty
31      ]
32    ]

```

Code 3.81: PicoC Mon Pass für Call by Sharing für Arrays

```

1 SymbolTable
2   [
3     Symbol(
4       {
5         type qualifier:      Empty()
6         datatype:            FunDecl(VoidType('void'), Name('stack_fun'),
7         ↪ [Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param1')), Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')),
9         ↪ Name('param2'))])
10        name:                Name('stack_fun')
11        value or address:     Empty()
12        position:             Pos(Num('1'), Num('5'))
13        size:                 Empty()
14      },
15      Symbol(
16        {
17          type qualifier:      Writable()
18          datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
19          name:                Name('param1@stack_fun')

```

```

17     value or address:    Num('0')
18     position:           Pos(Num('1'), Num('21'))
19     size:               Num('1')
20 },
21 Symbol(
22 {
23     type qualifier:      Writeable()
24     datatype:            PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
25     name:                Name('param2@stack_fun')
26     value or address:    Num('1')
27     position:            Pos(Num('1'), Num('37'))
28     size:               Num('1')
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('4'), Num('5'))
37     size:               Empty()
38 },
39 Symbol(
40 {
41     type qualifier:      Writeable()
42     datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:                Name('local_var1@main')
44     value or address:    Num('0')
45     position:            Pos(Num('5'), Num('6'))
46     size:               Num('6')
47 },
48 Symbol(
49 {
50     type qualifier:      Writeable()
51     datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
52     name:                Name('local_var2@main')
53     value or address:    Num('6')
54     position:            Pos(Num('6'), Num('6'))
55     size:               Num('6')
56 }
57 ]

```

Code 3.82: Symboltabelle für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         LOADIN BAF PC -1
8       ],
9     Block
10      Name 'main.0',

```

```

11  [
12      SUBI SP 2,
13      SUBI SP 1,
14      LOADI IN1 0,
15      ADD IN1 DS,
16      STOREIN SP IN1 1,
17      SUBI SP 1,
18      LOADI IN1 6,
19      ADD IN1 DS,
20      STOREIN SP IN1 1,
21      MOVE BAF ACC,
22      ADDI SP 4,
23      MOVE SP BAF,
24      SUBI SP 4,
25      STOREIN BAF ACC 0,
26      LOADI ACC GoTo
27          Name 'addr@next_instr',
28      ADD ACC CS,
29      STOREIN BAF ACC -1,
30      Exp
31      GoTo
32      Name 'stack_fun.1',
33      MOVE BAF IN1,
34      LOADIN IN1 BAF 0,
35      MOVE IN1 SP,
36      LOADIN BAF PC -1
37  ]
38 ]

```

Code 3.83: RETI Block Pass für Call by Sharing für Arrays

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

```

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

```

Code 3.84: PicoC Code für Call by Value für Structs

```

1 File
2     Name './example_fun_call_by_value_struct.picoc_mon',
3     [
4         Block
5             Name 'stack_fun.1',
6             [

```

```

7      Return
8      Empty
9  ],
10 Block
11   Name 'main.0',
12   [
13     StackMalloc
14     Num '2',
15     Assign
16     Tmp
17     Num '3',
18     GlobalRead
19     Num '0',
20     NewStackframe
21     Name 'stack_fun',
22     GoTo
23     Name 'addr@next_instr',
24     Exp
25     GoTo
26     Name 'stack_fun.1',
27     RemoveStackframe,
28     Return
29     Empty
30   ]
31 ]

```

Code 3.85: PicoC Mon Pass für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         LOADIN BAF PC -1
8       ],
9     Block
10      Name 'main.0',
11      [
12        SUBI SP 2,
13        SUBI SP 3,
14        LOADIN DS ACC 0,
15        STOREIN SP ACC 1,
16        LOADIN DS ACC 1,
17        STOREIN SP ACC 2,
18        LOADIN DS ACC 2,
19        STOREIN SP ACC 3,
20        MOVE BAF ACC,
21        ADDI SP 5,
22        MOVE SP BAF,
23        SUBI SP 5,
24        STOREIN BAF ACC 0,
25        LOADI ACC GoTo
26        Name 'addr@next_instr',

```

```
27      ADD ACC CS,  
28      STOREIN BAF ACC -1,  
29      Exp  
30      GoTo  
31          Name 'stack_fun.1',  
32      MOVE BAF IN1,  
33      LOADIN IN1 BAF 0,  
34      MOVE IN1 SP,  
35      LOADIN BAF PC -1  
36  ]  
37  ]
```

Code 3.86: RETI Block Pass für Call by Value für Structs

### 3.3.8 Umsetzung kleinerer Details

## 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

## 4.2 Showmode

## 4.3 Qualitätssicherung

## 4.4 Kommentierter Kompiliervorgang

## 4.5 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**  $C_{RETI\_PicoC}^{PicoC}$  (Definition 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 Maschine**, 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.<sup>a</sup>

<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

Will man nun für eine Maschine  $M_{RETI}$ , auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 4.4) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$  implementieren und hat bereits den gesamten **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$  in der Sprache  $L_{PicoC}$  geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**<sup>1</sup> reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$  auf der **Maschine**  $M_{RETI}$  zu kompilieren bereits einen kompilierten **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$ , der mit der Maschinensprache  $B_{RETI}$  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**  $C_{RETI\_PicoC}^{PicoC}$  nicht selbst komplett in der Maschinensprache  $B_{RETI}$  schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler**  $C_{PicoC}^{Python}$ , den man bereits in der Programmiersprache  $L_{Python}$  implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 4.3) darstellt, auf einer anderen Maschine  $M_{other}$  dafür nutzt, damit dieser den **Self-compiling**

<sup>1</sup>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.



**Compiler**  $C_{RETI\_PicoC}^{PicoC}$  für die Maschine  $M_{RETI}$  kompiliert bzw. **bootstrapped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine  $M_{other}$  auf die Maschine  $M_{RETI}$  kopiert.<sup>2</sup>

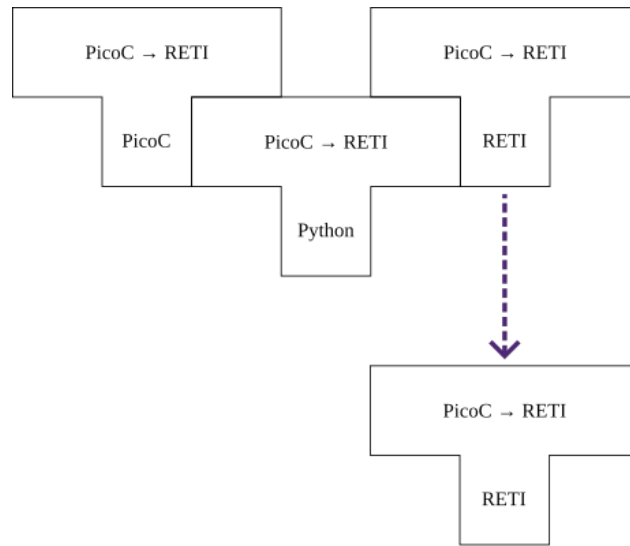


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

Einen ersten **minimalen Compiler**  $C_{2-w.min}$  für eine Maschine  $M_2$  und Wunschsprache  $L_w$  kann man entweder mittels eines **externen Bootstrap Compilers**  $C_w^o$  kompilieren<sup>a</sup> oder man schreibt ihn direkt in der **Maschinensprache**  $B_2$  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 allerersten 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.

<sup>a</sup>In diesem Fall, dem **Cross-Compiler**  $C_{PicoC}^{Python}$ .

#### 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.<sup>a,b</sup>

<sup>a</sup>Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.

<sup>b</sup>Thiemann, „Compilerbau“.

<sup>2</sup>Im Fall, dass auf der Maschine  $M_{RETI}$  die Programmiersprache  $L_{Python}$  bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$  auch mithilfe des **Cross-Compilers**  $C_{PicoC}^{Python}$  als **externe Entität** und der Programmiersprache  $L_{Python}$  auf der Maschine  $M_{RETI}$  selbst kompiliert werden.

**Definition 4.3: Bootstrap Compiler**

Compiler  $C_w^o$ , der es ermöglicht einen **Self-compiling Compiler**  $C_w^w$  zu **bootstrappen**, indem der **Self-compiling Compiler**  $C_w^w$  mit dem **Bootstrap Compiler**  $C_w^o$  **kompiliert** wird<sup>a</sup>. 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.<sup>b</sup>

<sup>a</sup>Dabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.

<sup>b</sup>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 Maschine  $M_{RETI}$  mittels **Bootstrapping** implementieren.<sup>3</sup>

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.

**Definition 4.4: Bootstrapping**

Wenn man einen **Self-compiling Compiler**  $C_w^w$  einer Wunschsprache  $L_w$  auf einer **Zielmaschine**  $M$  zum laufen bringt<sup>a,b,c,d</sup>. 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.<sup>e,f,g</sup>

<sup>a</sup>Z.B. mithilfe eines **Bootstrap Compilers**.

<sup>b</sup>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.

<sup>c</sup>Hat man einmal einen solchen **Self-compiling Compiler**  $C_w^w$  auf der Maschine  $M$  zum laufen gebracht, so kann man den Compiler auf der Maschine  $M$  weiterentwickeln, 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.

<sup>d</sup>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.

<sup>e</sup>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.

<sup>f</sup>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.

<sup>g</sup>Earley und Sturgis, „A formalism for translator interactions“.

<sup>3</sup>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  $L_{Python}$ .

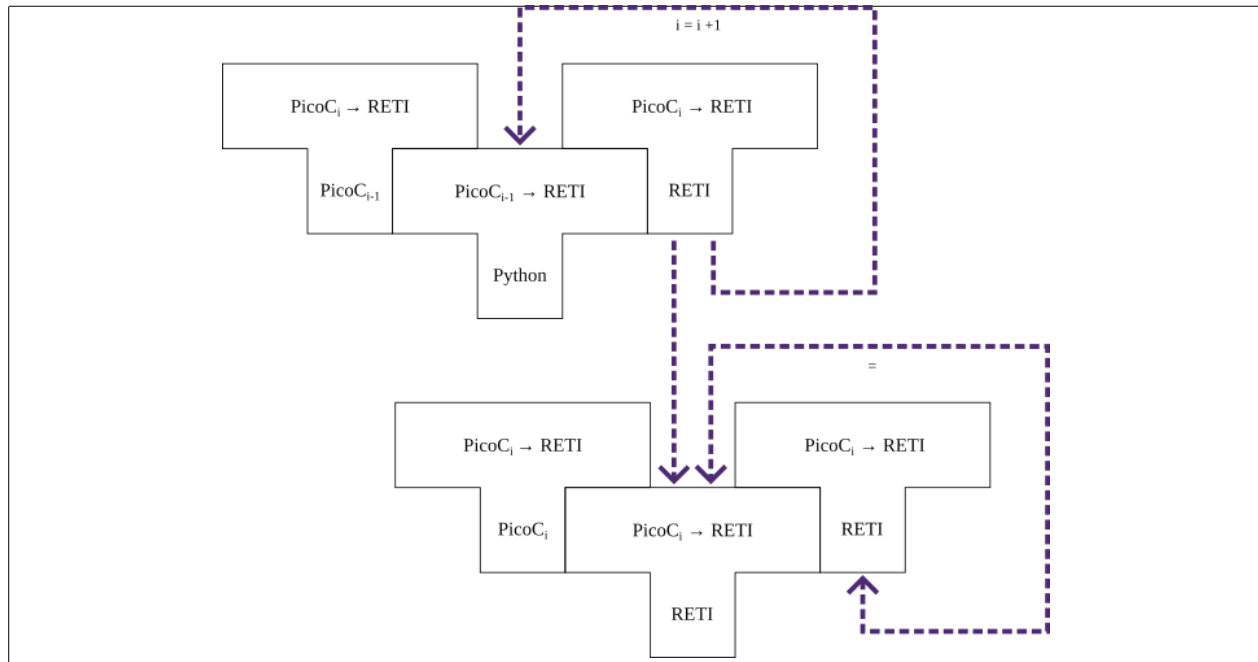


Abbildung 4.2: Iteratives Bootstrapping

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 Konkrete und Abstrakte Syntax

## A.2 Bedienungsanleitungen

### A.2.1 PicoC-Compiler

### A.2.2 Showmode

### A.2.3 Entwicklertools

---

---

# Literatur

## Online

- *C Operator Precedence* - *cppreference.com*. URL: [https://en.cppreference.com/w/c/language/operator\\_precedence](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).
- *JSON parser - Tutorial* — *Lark documentation*. URL: [https://lark-parser.readthedocs.io/en/latest/json\\_tutorial.html](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 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](https://doi.org/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](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](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).

## Sonstige Quellen

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).