

---

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>10</b>
1.1	RETI	10
1.2	PicoC	10
1.3	Aufgabenstellung	10
1.4	Eigenheiten der Sprache C	10
1.5	Richtlinien	10
<b>2</b>	<b>Einführung</b>	<b>11</b>
2.1	Compiler und Interpreter	11
2.1.1	T-Diagramme	14
2.2	Grammatiken	16
2.3	Grundlagen	16
2.3.1	Mehrdeutige Grammatiken	17
2.3.2	Präzidenz und Assoziativität	17
2.4	Lexikalische Analyse	18
2.5	Syntaktische Analyse	20
2.6	Code Generierung	26
2.7	Fehlermeldungen	26
2.7.1	Kategorien von Fehlermeldungen	26
<b>3</b>	<b>Implementierung</b>	<b>27</b>
3.1	Architektur	27
3.2	Lexikalische Analyse	27
3.2.1	Verwendung von Lark	27
3.2.2	Basic Parser	27
3.3	Syntaktische Analyse	27
3.3.1	Verwendung von Lark	27
3.3.2	Umsetzung von Präzidenz	27
3.3.3	Derivation Tree Generierung	30
3.3.4	Early Parser	30
3.3.5	Derivation Tree Vereinfachung	30
3.3.6	Abstrakt Syntax Tree Generierung	30
3.3.6.1	ASTNode	30
3.3.6.2	PicoC Nodes	30
3.3.6.3	RETI Nodes	30
3.4	Code Generierung	30
3.4.1	Passes	30
3.4.1.1	PicoC-Shrink Pass	30
3.4.1.2	PicoC-Blocks Pass	30
3.4.1.3	PicoC-Mon Pass	30
3.4.1.4	RETI-Blocks Pass	30
3.4.1.5	RETI-Patch Pass	30
3.4.1.6	RETI Pass	30
3.4.2	Umsetzung von Pointern	30
3.4.2.1	Referenzierung	30
3.4.2.2	Pointer Dereferenzierung durch Zugriff auf Arrayindex ersetzen	33

3.4.3	Umsetzung von Arrays . . . . .	34
3.4.3.1	Initialisierung von Arrays . . . . .	34
3.4.3.2	Zugriff auf Arrayindex . . . . .	36
3.4.3.3	Zuweisung an Arrayindex . . . . .	39
3.4.4	Umsetzung von Structs . . . . .	41
3.4.4.1	Deklaration von Structs . . . . .	41
3.4.4.2	Initialisierung von Structs . . . . .	42
3.4.4.3	Zugriff auf Structattribut . . . . .	46
3.4.4.4	Zuweisung an Structattribut . . . . .	48
3.4.5	Umsetzung der Derived Datatypes im Zusammenspiel . . . . .	50
3.4.5.1	Einleitungsteil für Globale Statische Daten und Stackframe . . . . .	50
3.4.5.2	Mittelteil für die verschiedenen Derived Datatypes . . . . .	53
3.4.5.3	Schlusssteil für die verschiedenen Derived Datatypes . . . . .	57
3.4.6	Umsetzung von Funktionen . . . . .	62
3.4.6.1	Funktionen auflösen zu RETI Code . . . . .	62
3.4.6.1.1	Sprung zur Main Funktion . . . . .	65
3.4.6.2	Funktionsdeklaration und -definition . . . . .	67
3.4.6.3	Funktionsaufruf . . . . .	68
3.4.6.3.1	Ohne Rückgabewert . . . . .	68
3.4.6.3.2	Mit Rückgabewert . . . . .	70
3.4.6.3.3	Umsetzung von Call by Sharing für Arrays . . . . .	73
3.4.6.3.4	Umsetzung von Call by Value für Structs . . . . .	76
3.4.7	Umsetzung kleinerer Details . . . . .	77
3.5	Fehlermeldungen . . . . .	77
3.5.1	Error Handler . . . . .	77
3.5.2	Arten von Fehlermeldungen . . . . .	77
3.5.2.1	Syntaxfehler . . . . .	77
3.5.2.2	Laufzeitfehler . . . . .	77
<b>4</b>	<b>Ergebnisse und Ausblick</b> . . . . .	<b>78</b>
4.1	Compiler . . . . .	78
4.2	Showmode . . . . .	78
4.3	Qualitätssicherung . . . . .	78
4.4	Kommentierter Kompilervorgang . . . . .	78
4.5	Erweiterungsideen . . . . .	78
<b>A</b>	<b>Appendix</b> . . . . .	<b>82</b>
A.1	Konkrete und Abstrakte Syntax . . . . .	82
A.2	Bedienungsanleitungen . . . . .	82
A.2.1	PicoC-Compiler . . . . .	82
A.2.2	Showmode . . . . .	82
A.2.3	Entwicklertools . . . . .	82

---

---

# Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	16
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	16
2.3	Veranschaulichung der Lexikalischen Analyse . . . . .	20
2.4	Veranschaulichung der Syntaktischen Analyse . . . . .	25
3.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	28
3.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	28
3.3	Architektur mit allen Passes ausgeschrieben . . . . .	29
4.1	Cross-Compiler als Bootstrap Compiler . . . . .	79
4.2	Iteratives Bootstrapping . . . . .	81

---

---

# Codeverzeichnis

3.1	PicoC Code für Pointer Referenzierung . . . . .	30
3.2	Abstract Syntax Tree für Pointer Referenzierung . . . . .	31
3.3	Symboltabelle für Pointer Referenzierung . . . . .	31
3.4	PicoC Mon Pass für Pointer Referenzierung . . . . .	32
3.5	RETI Blocks Pass für Pointer Referenzierung . . . . .	32
3.6	PicoC Code für Pointer Dereferenzierung . . . . .	33
3.7	Abstract Syntax Tree für Pointer Dereferenzierung . . . . .	33
3.8	PicoC Shrink Pass für Pointer Dereferenzierung . . . . .	34
3.9	PicoC Code für Array Initialisierung . . . . .	34
3.10	Abstract Syntax Tree für Array Initialisierung . . . . .	35
3.11	Symboltabelle für Array Initialisierung . . . . .	35
3.12	PicoC Mon Pass für Array Initialisierung . . . . .	36
3.13	RETI Blocks Pass für Array Initialisierung . . . . .	36
3.14	PicoC Code für Zugriff auf Arrayindex . . . . .	36
3.15	Abstract Syntax Tree für Zugriff auf Arrayindex . . . . .	37
3.16	PicoC Mon Pass für Zugriff auf Arrayindex . . . . .	38
3.17	RETI Blocks Pass für Zugriff auf Arrayindex . . . . .	38
3.18	PicoC Code für Zuweisung an Arrayindex . . . . .	39
3.19	Abstract Syntax Tree für Zuweisung an Arrayindex . . . . .	39
3.20	PicoC Mon Pass für Zuweisung an Arrayindex . . . . .	40
3.21	RETI Blocks Pass für Zuweisung an Arrayindex . . . . .	40
3.22	PicoC Code für Deklaration von Structs . . . . .	41
3.23	Symboltabelle für Deklaration von Structs . . . . .	42
3.24	PicoC Code für Initialisierung von Structs . . . . .	42
3.25	Abstract Syntax Tree für Initialisierung von Structs . . . . .	43
3.26	Symboltabelle für Initialisierung von Structs . . . . .	44
3.27	PicoC Mon Pass für Initialisierung von Structs . . . . .	45
3.28	RETI Blocks Pass für Initialisierung von Structs . . . . .	45
3.29	PicoC Code für Zugriff auf Structattribut . . . . .	46
3.30	Abstract Syntax Tree für Zugriff auf Structattribut . . . . .	46
3.31	PicoC Mon Pass für Zugriff auf Structattribut . . . . .	47
3.32	RETI Blocks Pass für Zugriff auf Structattribut . . . . .	48
3.33	PicoC Code für Zuweisung an Structattribut . . . . .	48
3.34	Abstract Syntax Tree für Zuweisung an Structattribut . . . . .	49
3.35	PicoC Mon Pass für Zuweisung an Structattribut . . . . .	50
3.36	RETI Blocks Pass für Zuweisung an Structattribut . . . . .	50
3.37	PicoC Code für den Einleitungsteil . . . . .	51
3.38	Abstract Syntax Tree für den Einleitungsteil . . . . .	52
3.39	PicoC Mon Pass für den Einleitungsteil . . . . .	53
3.40	RETI Blocks Pass für den Einleitungsteil . . . . .	53
3.41	PicoC Code für den Mittelteil . . . . .	53
3.42	Abstract Syntax Tree für den Mittelteil . . . . .	54
3.43	PicoC Mon Pass für den Mittelteil . . . . .	55
3.44	RETI Blocks Pass für den Mittelteil . . . . .	56
3.45	PicoC Code für den Schlussteil . . . . .	57
3.46	Abstract Syntax Tree für den Schlussteil . . . . .	59
3.47	PicoC Mon Pass für den Schlussteil . . . . .	60

3.48 RETI Blocks Pass für den Schlussteil . . . . .	62
3.49 PicoC Code für 3 Funktionen . . . . .	63
3.50 Abstract Syntax Tree für 3 Funktionen . . . . .	63
3.51 PicoC Blocks Pass für 3 Funktionen . . . . .	64
3.52 PicoC Mon Pass für 3 Funktionen . . . . .	64
3.53 RETI Blocks Pass für 3 Funktionen . . . . .	65
3.54 PicoC Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	65
3.55 PicoC Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	66
3.56 PicoC Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	66
3.57 PicoC Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	67
3.58 PicoC Code für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	67
3.59 Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	68
3.60 PicoC Code für Funktionsaufruf ohne Rückgabewert . . . . .	69
3.61 PicoC Mon Pass für Funktionsaufruf ohne Rückgabewert . . . . .	69
3.62 RETI Blocks Pass für Funktionsaufruf ohne Rückgabewert . . . . .	70
3.63 RETI Pass für Funktionsaufruf ohne Rückgabewert . . . . .	70
3.64 PicoC Code für Funktionsaufruf mit Rückgabewert . . . . .	71
3.65 PicoC Mon Pass für Funktionsaufruf mit Rückgabewert . . . . .	71
3.66 RETI Blocks Pass für Funktionsaufruf mit Rückgabewert . . . . .	72
3.67 RETI Pass für Funktionsaufruf mit Rückgabewert . . . . .	73
3.68 PicoC Code für Call by Sharing für Arrays . . . . .	73
3.69 PicoC Mon Pass für Call by Sharing für Arrays . . . . .	74
3.70 Symboltabelle für Call by Sharing für Arrays . . . . .	75
3.71 RETI Block Pass für Call by Sharing für Arrays . . . . .	76
3.72 PicoC Code für Call by Value für Structs . . . . .	76
3.73 PicoC Mon Pass für Call by Value für Structs . . . . .	76
3.74 RETI Block Pass für Call by Value für Structs . . . . .	77

---

---

# Tabellenverzeichnis

3.1 Präzidenzregeln von PicoC . . . . .	29
---	----



---

---

# Definitionsverzeichnis

1.1	Caller-save Register	10
1.2	Callee-save Register	10
1.3	Deklaration	10
1.4	Definition	10
1.5	Call by value	10
1.6	Call by reference	10
2.1	Interpreter	11
2.2	Compiler	11
2.3	Maschinensprache	12
2.4	Assemblersprache (bzw. engl. Assembly Language)	12
2.5	Assembler	13
2.6	Objectcode	13
2.7	Linker	13
2.8	Immediate	13
2.9	Transpiler (bzw. Source-to-source Compiler)	13
2.10	Cross-Compiler	14
2.11	T-Diagram Programm	14
2.12	T-Diagram Übersetzer (bzw. eng. Translator)	15
2.13	T-Diagram Interpreter	15
2.14	T-Diagram Maschine	15
2.15	Sprache	16
2.16	Chromsky Hierarchie	16
2.17	Grammatik	17
2.18	Reguläre Sprachen	17
2.19	Kontextfreie Sprachen	17
2.20	Ableitung	17
2.21	Links- und Rechtsableitung	17
2.22	Linksrekursive Grammatiken	17
2.23	Ableitungsbaum	17
2.24	Mehrdeutige Grammatik	17
2.25	Assoziativität	17
2.26	Präzidenz	17
2.27	Wortproblem	17
2.28	LL(k)-Grammatik	18
2.29	Pattern	18
2.30	Lexeme	18
2.31	Lexer (bzw. Scanner)	18
2.32	Bezeichner (bzw. Identifier)	19
2.33	Literal	20
2.34	Konkrete Syntax	20
2.35	Derivation Tree (bzw. Parse Tree)	21
2.36	Parser	21
2.37	Recognizer (bzw. Erkennen)	21
2.38	Transformer	23
2.39	Visitor	23
2.40	Abstrakte Syntax	23
2.41	Abstrakt Syntax Tree	23

2.42	Pass . . . . .	26
2.43	Monadische Normalform . . . . .	26
2.44	Fehlermeldung . . . . .	26
3.1	Symboltabelle . . . . .	30
4.1	Self-compiling Compiler . . . . .	78
4.2	Minimaler Compiler . . . . .	79
4.3	Bootstrap Compiler . . . . .	80
4.4	Bootstrapping . . . . .	80

---

---

# 1 Motivation

## 1.1 RETI

Definition 1.1: Caller-save Register

Definition 1.2: Callee-save Register

## 1.2 PicoC

## 1.3 Aufgabenstellung

## 1.4 Eigenheiten der Sprache C

Definition 1.3: Deklaration

Definition 1.4: Definition

Definition 1.5: Call by value

Definition 1.6: Call by reference

## 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

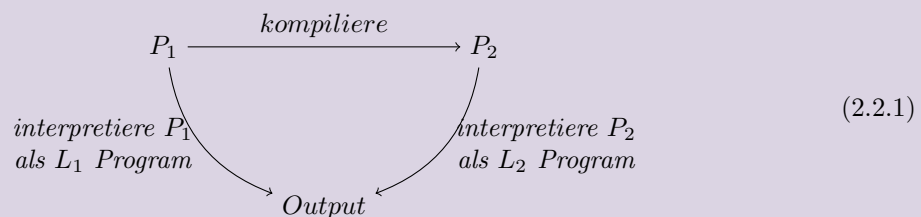
*Interpretiert die **Instructions** bzw. **Statements** eines Programmes  $P$  direkt.*

*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.*

### 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.*



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

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

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 **Maschinenncode**, der in **Maschiensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschiensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

#### Definition 2.3: Maschiensprache

*Programmiersprache, deren mögliche Programme die **hardwarenahe 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</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.

#### 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</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.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenncode** 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 **Maschinenncode** 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 **Maschinentensprache** geschrieben ist.

**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.

**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.

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 ausgeschriebenen 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?*

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

Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen Level an Abstraktion** arbeiten<sup>a</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>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 2.10: 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 **Maschinensprachen**  $B_1$  und  $B_2$  haben.<sup>a</sup>

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

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 **Maschinensprache**  $B_2$  einer Zielmaschine  $M_2$  läuft.<sup>3</sup>

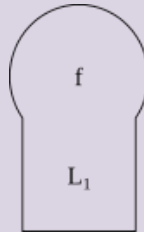
**2.1.1 T-Diagramme**

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**<sup>4</sup> besonders gut, da diese optimal darauf **zugeschnitten** sind die Eigenheiten von Compilern in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Program (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine 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.



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.

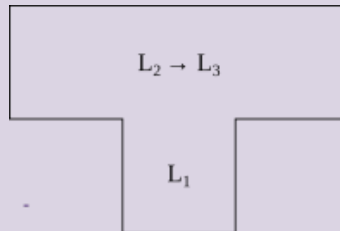
<sup>3</sup>Die an vielen Universitäten und Schulen eingesetzten programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren Microcontroller eine **C-ähnliche Sprache** in die Maschinensprache des Microcontrollers zu kompilieren, da der Microcontroller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

<sup>4</sup>Earley und Sturgis, „A formalism for translator interactions“.

**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.

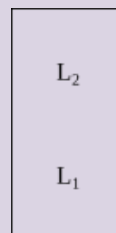
Für den **Übersetzer** gelten genauso, wie für einen **Compiler**<sup>a</sup> die **Beziehungen** in 2.2.1.



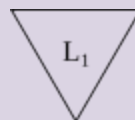
<sup>a</sup>Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren**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

**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.

**Definition 2.14: T-Diagramm Maschine**

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

Aus den verschiedenen **Blöcken** lassen sich **Kompositionen** bilden, indem man sie **adjacent** 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.



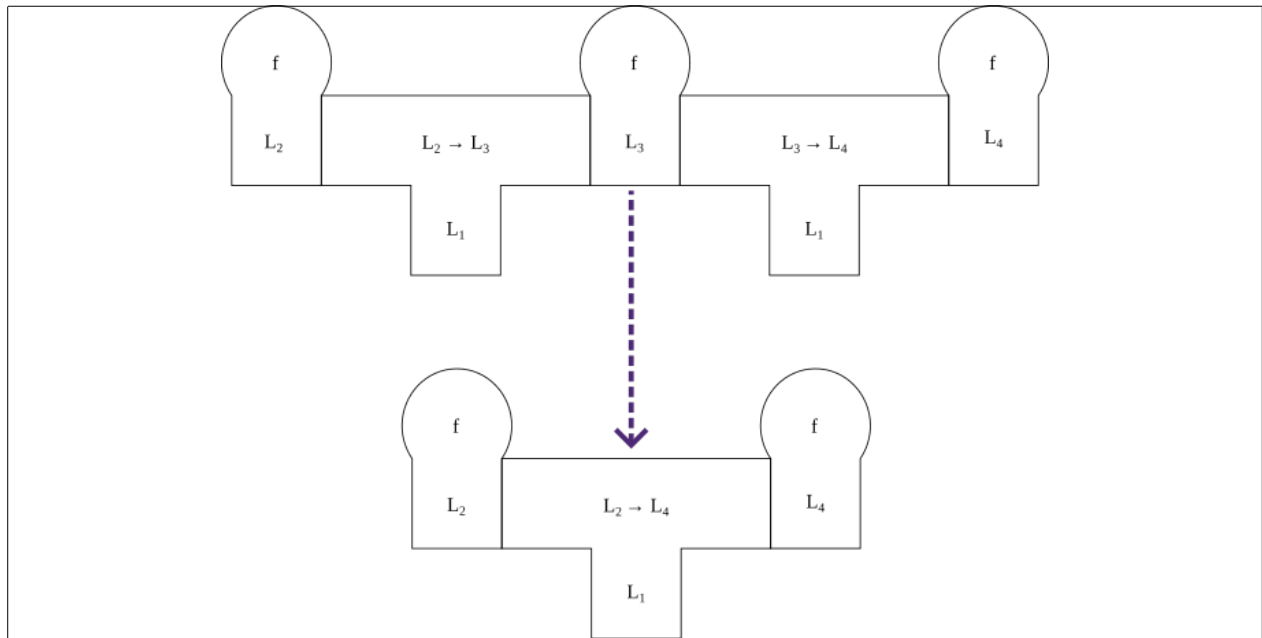


Abbildung 2.1: Horinzontale Übersetzungszwischenschritte zusammenfassen

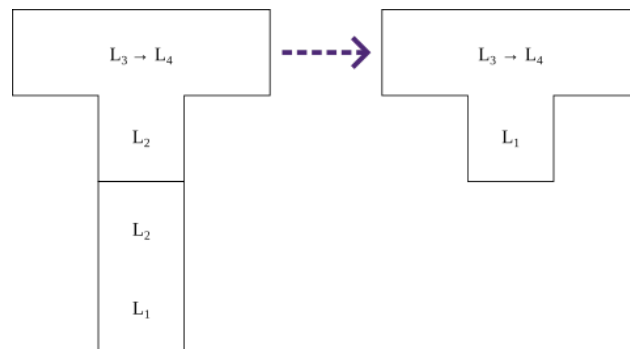


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

## 2.2 Grammatiken

## 2.3 Grundlagen

**Definition 2.15: Sprache**

**Definition 2.16: Chomsky Hierarchie**

**Definition 2.17: Grammatik****Definition 2.18: Reguläre Sprachen****Definition 2.19: Kontextfreie Sprachen****Definition 2.20: Ableitung****Definition 2.21: Links- und Rechtsableitung****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.

**2.3.1 Mehrdeutige Grammatiken****Definition 2.23: Ableitungsbaum****Definition 2.24: Mehrdeutige Grammatik****2.3.2 Präzidenz und Assoziativität****Definition 2.25: Assoziativität****Definition 2.26: Präzidenz****Definition 2.27: Wortproblem**

**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>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.

## 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>What is the difference between a token and a lexeme?

**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>What is the difference between a token and a lexeme?

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)**

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>lecture-notes-2021.

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</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.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>5</sup> und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen 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>6</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>7</sup>. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

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

<sup>5</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>6</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>7</sup>Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut<sup>8</sup>, unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

#### Definition 2.33: Literal

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

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

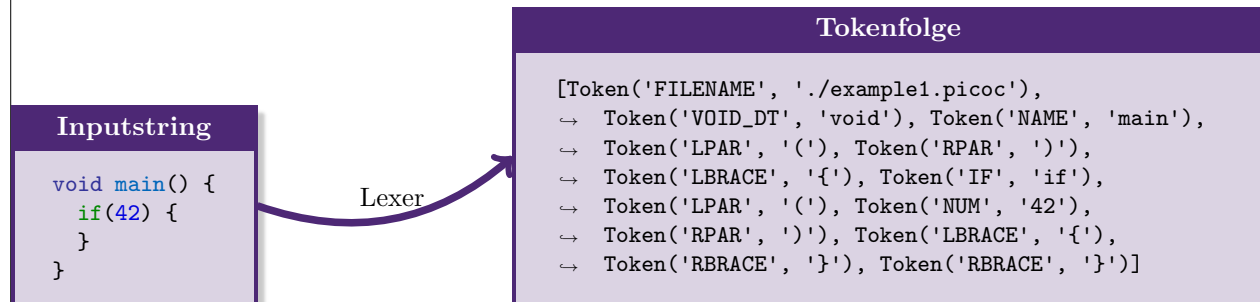


Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

## 2.5 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik**  $G_{Parse}$  notwendig, um diese Sprache 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 Klammern '(' bzw. öffnende geschweifte Klammern '{' es momentan gibt, die noch nicht durch eine entsprechende schließende 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 **Abstrakt Syntax Tree** (Definition 2.41) dient. Für einen ordentlichen Code ist es vor allem im Compilerbau förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann der **Abstrakt Syntax Tree**.

#### Definition 2.34: Konkrete Syntax

*Syntax* einer *Sprache*, die durch die *Grammatiken*  $G_{Lex}$  und  $G_{Parse}$  zusammengekommen 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>Course Webpage for Compilers (P423, P523, E313, and E513).

<sup>8</sup>Man nennt das auch einem **Lookahead** von 1

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

*Compilerinterne Darstellung eines in **Konkreter Syntax** geschriebenen Inputstrings als **Baumdatenstruktur**, in der **Nichtterminalsymbole** die **Inneren Knoten** des Baumes und **Terminalsymbole** die **Blätter** des Baumes bilden. Jede **Produktion** der **Grammatik**  $G_{Parse}$ , die ein Teil der **Konkrete Syntax** ist, wird zu einem eigenen **Knoten**.*

*Der **Derivation Tree** wird optimalerweise immer so konstruiert bzw. die **Konkrete Syntax** immer so definiert, dass sich möglichst einfach ein **Abstrakt Syntax Tree** daraus konstruieren lässt.*

**Definition 2.36: Parser**

*Ein Programm, das eine **Eingabe** in eine für die **Weiterverarbeitung** taugliche Form bringt.*

**2.36.1:** *In Bezug auf Compilerbau ist ein **Parser** ein Programm, das einen Inputstring von **Konkreter Syntax** in die compilerinterne Darstellung eines **Derivation Tree** übersetzt, was auch als **Parse** 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, das einen Inputstring von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die obige Definition 2.36.1 verwendet.

<sup>b</sup>*Compiler Design - Phases of Compiler.*

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung entstehen, ob ein **Lexer** nach der obigen Definition nicht auch ein **Parser** ist.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines Parsers. Der Parser vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich. Aber für sich isoliert, ohne Bezug zu Compilerbau betrachtet, ist ein Lexer nach Definition 2.36 ebenfalls ein Parser. Aber im Compilerbau hat **Parser** eine spezifischere Definition und hier überwiegt beim **Lexer** seine Funktionalität, dass er den Inputstring lexikalisch weiterverarbeitet, um ihn als Lexer zu bezeichnen, der Teil eines Parsers ist.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** (Definition 2.36) 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 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 relevant.

Ein **Parser** ist genauer gesagt 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. Erkenner)**

*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>a</sup>.*

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

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

- **Top-Down Parsing:** Der **Derivation Tree** wird von **oben-nach-unten** generiert, also von der **Wurzel** zu den **Blättern**. Dementsprechend fängt die Generierung des **Derivation Tree** mit dem **Startsymbol** der **Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man **Terminalsymbole** hat und der gewünschte **Inputstring** abgeleitet wurde oder es 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 das **Eingabewort** 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 Produktionsregeln dieses **Nicht-Terminalsymbols** umsetzt. Prozeduren rufen sich dabei wechselseitig gegenseitig entsprechend der **Produktionsregeln** auf, falls eine entsprechende Produktionsregel eine **Rekursion** enthält.

**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.

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<sup>b</sup>

Wenn man eine **LL(k)** Grammatik hat, kann man auf **Backtracking verzichten** und es reicht einfach nur immer  $k$  **Symbole** im **Eingabewort** bzw. in Bezug auf Compilerbau **Token** im **Inputstring** voranzuschauen. **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 Art von Parser 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. Dieser **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 **Abstrakt 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).

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstrakt Syntax Tree** ermöglicht es die Operationen, die der Compiler bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen.

#### Definition 2.38: Transformer

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** einen entsprechenden Knoten des **Abstrakt Syntax Tree** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstrakt Syntax Tree** konstruiert.

#### 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.

Kann theoretisch auch zur Konstruktion eines **Abstrakt Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakt Syntax Tree** verantwortlich ist, aber dafür ist ein **Transformer** besser geeignet.

#### Definition 2.40: Abstrakte Syntax

**Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstrakt Syntax Trees** möglich sind.<sup>a</sup>

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.

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

#### Definition 2.41: Abstrakt 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 **Abstrakt 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 ihm bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, möglichst schnell die Frage 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>Course Webpage for Compilers (P423, P523, E313, and E513).

Je weiter **unten**<sup>9</sup> und **links** ein Knoten im **Abstrakt Syntax Tree** liegt, desto eher wird dieser Knoten komplett abgearbeitet sein, da in der **Code Generierung** die Knoten nach dem **Depth First Search**

<sup>9</sup>In der Informatik wachsen **Bäume** von **oben-nach-unten**. Die **Wurzel** ist also **oben**.



Prinzip abgearbeitet werden.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.4 die Syntaktische mit dem Beispiel aus Subkapitel 2.4 fortgeführt.

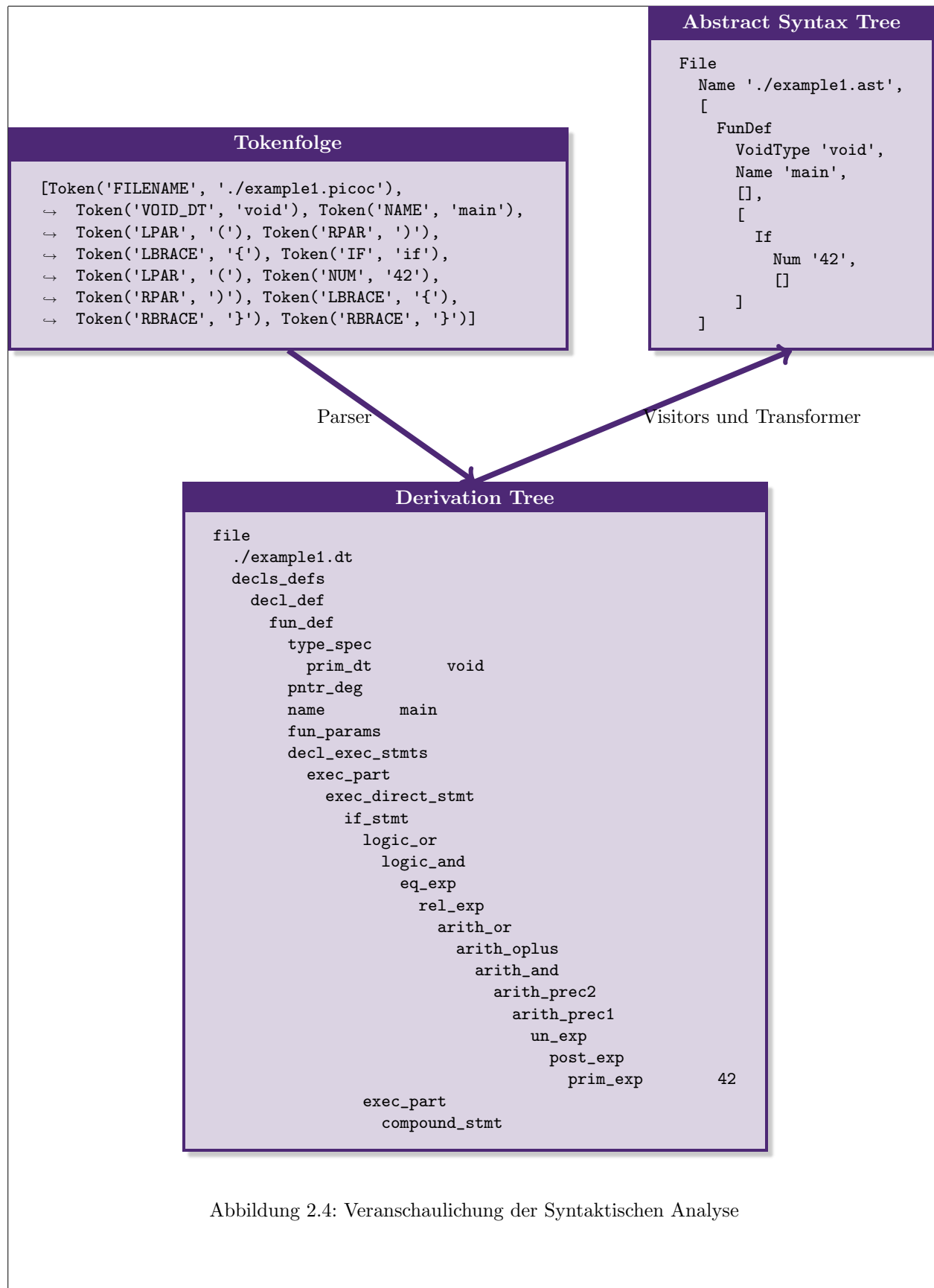


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse

## 2.6 Code Generierung

### Definition 2.42: Pass

### Definition 2.43: Monadische Normalform

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

## 2.7 Fehlermeldungen

### Definition 2.44: Fehlermeldung

### 2.7.1 Kategorien von Fehlermeldungen

---

---

# 3 Implementierung

## 3.1 Architektur

## 3.2 Lexikalische Analyse

### 3.2.1 Verwendung von Lark

### 3.2.2 Basic Parser

## 3.3 Syntaktische Analyse

### 3.3.1 Verwendung von Lark

### 3.3.2 Umsetzung von Präzidenz

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

---

<sup>1</sup>[C Operator Precedence - cppreference.com](http://c.operatorprecedence-cppreference.com).

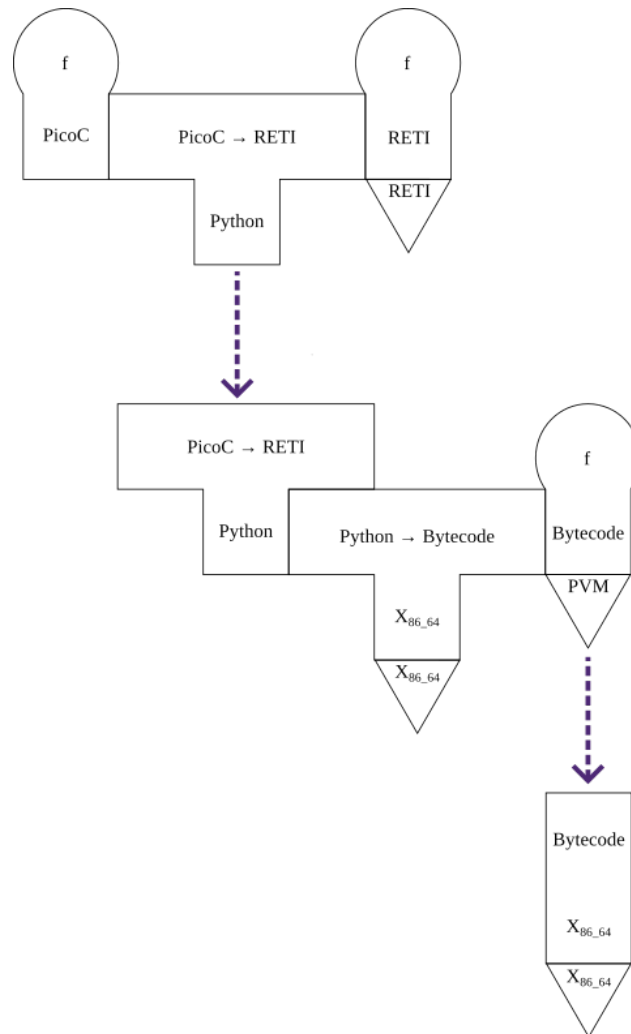


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

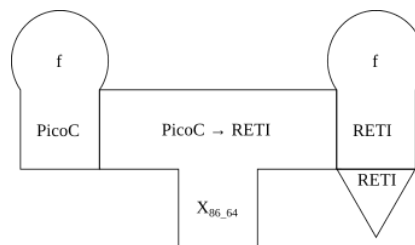


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

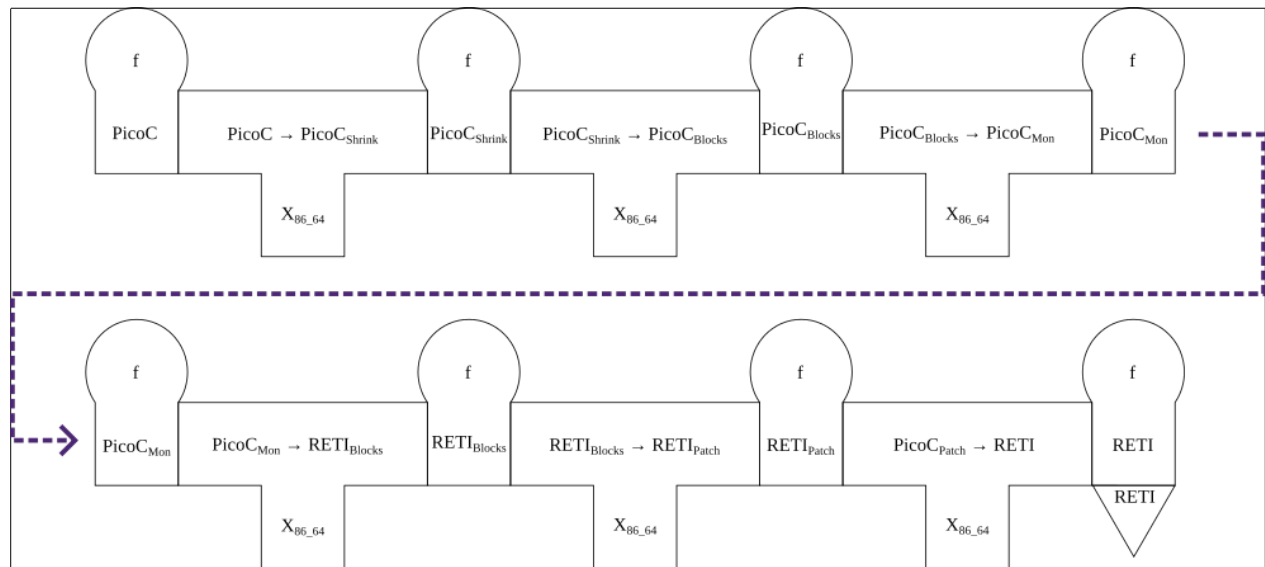


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

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

Tabelle 3.1: Präzidenzregeln von PicoC

### 3.3.3 Derivation Tree Generierung

### 3.3.4 Early Parser

### 3.3.5 Derivation Tree Vereinfachung

### 3.3.6 Abstrakt Syntax Tree Generierung

#### 3.3.6.1 ASTNode

#### 3.3.6.2 PicoC Nodes

#### 3.3.6.3 RETI Nodes

## 3.4 Code Generierung

### 3.4.1 Passes

#### 3.4.1.1 PicoC-Shrink Pass

#### 3.4.1.2 PicoC-Blocks Pass

#### 3.4.1.3 PicoC-Mon Pass

#### Definition 3.1: Symboltabelle

#### 3.4.1.4 RETI-Blocks Pass

#### 3.4.1.5 RETI-Patch Pass

#### 3.4.1.6 RETI Pass

### 3.4.2 Umsetzung von Pointern

#### 3.4.2.1 Referenzierung

```

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

```

Code 3.1: 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      PtrDecl
19      Num '1',
20      IntType 'int',
21      Name 'ptr',
22    Ref
23      Name 'var'
24  ]
25 ]

```

Code 3.2: 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:            PtrDecl(Num('1'), IntType('int'))
25       name:                Name('ptr@main')
26       value or address:    Num('1')
27       position:            Pos(Num('3'), Num('7'))
28       size:                Num('1')
29    }
30 ]

```

Code 3.3: 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 3.4: PicoC Mon Pass für Pointer Referenzierung

```
1 File
2   Name './example_pntr_ref.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 3.5: RETI Blocks Pass für Pointer Referenzierung

**3.4.2.2 Pointer Dereferenzierung durch Zugriff auf Arrayindex ersetzen**

```

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

```

Code 3.6: 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              PntrDecl
19                Num '1',
20                IntType 'int',
21                Name 'pntr',
22              Ref
23                Name 'var',
24            Exp
25              Deref
26                Name 'pntr',
27                Num '0'
28          ]
29      ]

```

Code 3.7: Abstract Syntax Tree für Pointer Dereferenzierung

```

1 File
2   Name './example_pntr_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     PntrDecl
19     Num '1',
20     IntType 'int',
21     Name 'pntr',
22   Ref
23     Name 'var',
24   Exp
25     Subscr
26     Name 'pntr',
27     Num '0'
28 ]
29 ]

```

Code 3.8: PicoC Shrink Pass für Pointer Dereferenzierung

### 3.4.3 Umsetzung von Arrays

#### 3.4.3.1 Initialisierung von Arrays

```

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

```

Code 3.9: 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 3.10: 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 3.11: 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 3.12: 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 3.13: RETI Blocks Pass für Array Initialisierung

### 3.4.3.2 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 3.14: 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 3.15: 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 3.16: 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 3.17: RETI Blocks Pass für Zugriff auf Arrayindex

### 3.4.3.3 Zuweisung an Arrayindex

```

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

```

Code 3.18: 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 3.19: 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 3.20: 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 3.21: RETI Blocks Pass für Zuweisung an Arrayindex

### 3.4.4 Umsetzung von Structs

#### 3.4.4.1 Deklaration von Structs

```

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

```

Code 3.22: 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(Writeable(),
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(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('3')
40    },
41   Symbol(

```

```

40     {
41         type qualifier:      Empty()
42         datatype:           FunDecl(VoidType('void'), Name('main'), [])
43         name:                Name('main')
44         value or address:    Empty()
45         position:            Pos(Num('5'), Num('5'))
46         size:                Empty()
47     }
48 ]

```

Code 3.23: Symboltabelle für Deklaration von Structs

### 3.4.4.2 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 3.24: 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 3.25: Abstract Syntax Tree für Initialisierung von Structs

```

1 SymbolTable
2 [
3   Symbol(
4     {
5       type qualifier:      Empty()
6       datatype:            ArrayDecl([Num('1')], PntrDecl(Num('1'), IntType('int')))
7       name:                Name('pntr@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(Writable(),
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(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('1')
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         Symbol(
51           {
52             type qualifier: Writable()
53             datatype:       IntType('int')
54             name:           Name('var@main')
55             value or address: Num('0')
56             position:       Pos(Num('6'), Num('6'))
57             size:           Num('1')
58           },
59           Symbol(
60             {
61               type qualifier: Writable()
62               datatype:       StructSpec(Name('st1'))
63               name:           Name('st@main')
64               value or address: Num('1')
65               position:       Pos(Num('7'), Num('13'))
66               size:           Num('1')
67             }
68           )
69         )
70       )
71     )
72   )
73 ]

```

Code 3.26: Symoltabelle 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 3.27: 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 3.28: RETI Blocks Pass für Initialisierung von Structs

## 3.4.4.3 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 3.29: 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 3.30: 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 3.31: 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 3.32: RETI Blocks Pass für Zugriff auf Structattribut

#### 3.4.4.4 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 3.33: 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 3.34: 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 3.35: 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 3.36: RETI Blocks Pass für Zuweisung an Structattribut

### 3.4.5 Umsetzung der Derived Datatypes im Zusammenspiel

#### 3.4.5.1 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 3.37: 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 'pntr2',
48     Exp
49     Name 'pntr2'
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     PntrDecl
70     Num '1',
71     ArrayDecl
72     [
73         Num '3'
74     ],
75     IntType 'int',
76     Name 'pntr1',
77     Exp
78     Name 'pntr1'
79 ]
80 ]

```

Code 3.38: 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 3.39: 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 3.40: RETI Blocks Pass für den Einleitungsteil

### 3.4.5.2 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 3.41: 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 3.42: 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 3.43: 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 3.44: RETI Blocks Pass für den Mittelteil

**3.4.5.3    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 3.45: 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     Assign
60     Alloc
61     Writeable,
62     IntType 'int',
63     Name 'var',
64     Num '42',
65     Assign
66     Alloc
67     Writeable,
68     PntrDecl
69     Num '1',
70     IntType 'int',
71     Name 'pntr1',
72     Ref
73     Name 'var',
74     Assign
75     Alloc
76     Writeable,
77     PntrDecl
78     Num '2',
79     IntType 'int',
80     Name 'pntr2',
81     Ref
82     Name 'pntr1',
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 'pntr2',
```

```
94         Num '0'
95     ]
96 ]
```

Code 3.46: 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 3.47: PicoC Mon Pass für den Schlussteil

```
1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5       Name 'main.O',
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 3.48: RETI Blocks Pass für den Schlussteil

## 3.4.6 Umsetzung von Funktionen

### 3.4.6.1 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 3.49: 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 3.50: 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 3.51: 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 3.52: 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 3.53: RETI Blocks Pass für 3 Funktionen

**Sprung zur Main Funktion:**

```

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

```

Code 3.54: 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 3.55: 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 3.56: 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 3.57: PicoC Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

### 3.4.6.2 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 3.58: 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(Writable(),
7         ↪ IntType('int'), Name('var'))])
8         name:                Name('fun2')
9         value or address:    Empty()

```

```

9      position:      Pos(Num('1'), Num('4'))
10     size:          Empty()
11   },
12   Symbol(
13   {
14     type qualifier:  Empty()
15     datatype:        FunDecl(VoidType('void'), Name('fun1'), [])
16     name:            Name('fun1')
17     value or address: Empty()
18     position:        Pos(Num('3'), Num('5'))
19     size:            Empty()
20   },
21   Symbol(
22   {
23     type qualifier:  Empty()
24     datatype:        FunDecl(VoidType('void'), Name('main'), [])
25     name:            Name('main')
26     value or address: Empty()
27     position:        Pos(Num('6'), Num('5'))
28     size:            Empty()
29   },
30   Symbol(
31   {
32     type qualifier:  Writeable()
33     datatype:        IntType('int')
34     name:            Name('var@main')
35     value or address: Num('0')
36     position:        Pos(Num('7'), Num('6'))
37     size:            Num('1')
38   },
39   Symbol(
40   {
41     type qualifier:  Writeable()
42     datatype:        IntType('int')
43     name:            Name('var@fun2')
44     value or address: Num('0')
45     position:        Pos(Num('11'), Num('13'))
46     size:            Num('1')
47   }
48 ]

```

Code 3.59: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

### 3.4.6.3 Funktionsaufruf

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 3.60: 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 3.61: 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 3.62: 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 3.63: RETI Pass für Funktionsaufruf ohne Rückgabewert

**Mit Rückgabewert:**

```

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

```

Code 3.64: 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 3.65: 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 3.66: 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 3.67: RETI Pass für Funktionsaufruf mit Rückgabewert

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 3.68: 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 3.69: 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(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param1')), Alloc(Writeable(), 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:      Writeable()
18        datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
19        name:                 Name('param1@stack_fun')
20        value or address:     Num('0')
21        position:             Pos(Num('1'), Num('21'))
22        size:                 Num('1')
23      },
24      Symbol(
25        {
26          type qualifier:      Writeable()
27          datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
28          name:                 Name('param2@stack_fun')
29          value or address:     Num('1')
30          position:             Pos(Num('1'), Num('37'))
31          size:                 Num('1')
32        },
33        Symbol(
34          {
35            type qualifier:      Empty()
36            datatype:            FunDecl(VoidType('void'), Name('main'), [])
37            name:                 Name('main')
38            value or address:     Empty()
39            position:             Pos(Num('4'), Num('5'))
40            size:                 Empty()
41          },
42          Symbol(
43            {
44              type qualifier:      Writeable()
45              datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
46              name:                 Name('local_var1@main')
47              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 3.70: 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 3.71: RETI Block Pass für Call by Sharing für Arrays

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 3.72: 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 3.73: 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 3.74: RETI Block Pass für Call by Value für Structs

### 3.4.7 Umsetzung kleinerer Details

## 3.5 Fehlermeldungen

### 3.5.1 Error Handler

### 3.5.2 Arten von Fehlermeldungen

#### 3.5.2.1 Syntaxfehler

#### 3.5.2.2 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.

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 Compiler**  $C_{RETI\_PicoC}^{PicoC}$  für die Maschine  $M_{RETI}$  kompiliert bzw. **bootstraped** und man den kompilierten

<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.

**RETI-Maschiendencode** dann einfach von der Maschine  $M_{other}$  auf die Maschine  $M_{RETI}$  kopiert.<sup>2</sup>

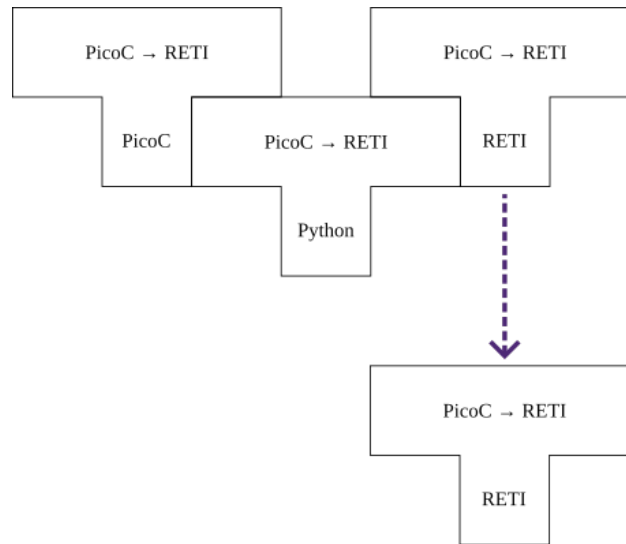


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

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**  $C_w^o$  oder einer **ersten Version**  $C_{w_i}^{w_i}$  des Self-compiling Compilers  $C_w^w$  wichtig sind.<sup>a</sup>

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

<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 **zirkuläre Abhängigkeit**, dass initial ein **Self-compiling Compiler**  $C_w^w$  bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.

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

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 **Lügendgeschichten 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  $L_{Python}$ .

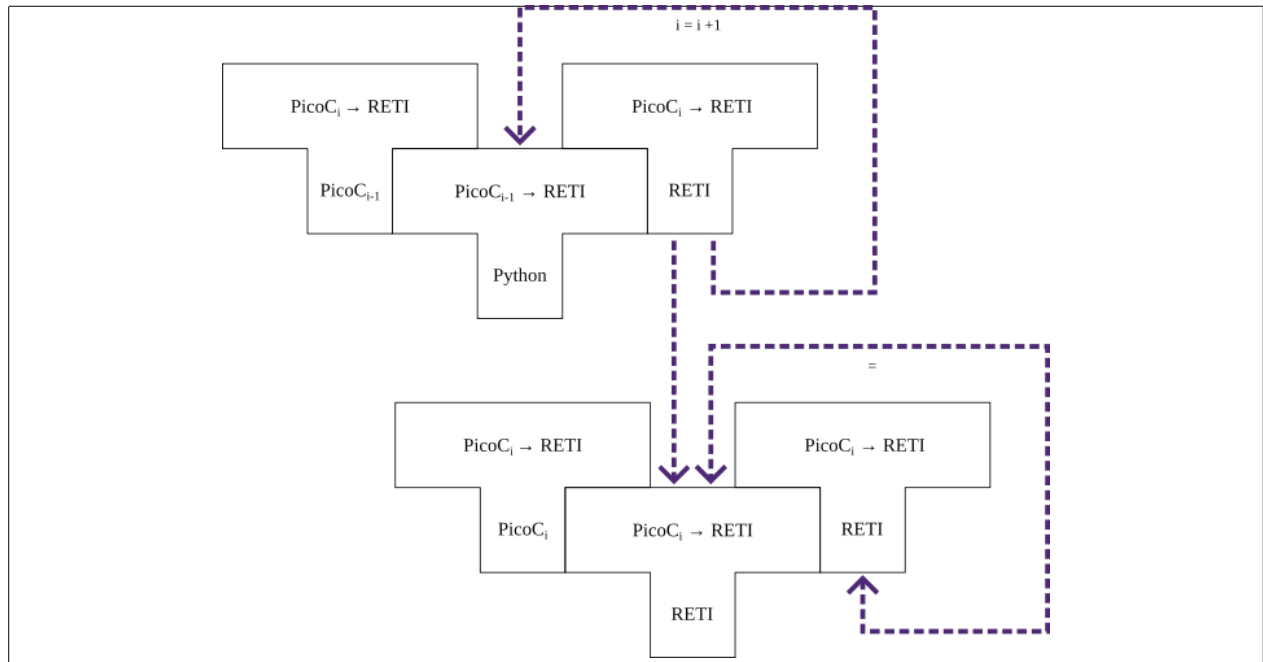


Abbildung 4.2: Iteratives Bootstrapping

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).
- *Compiler Design - Phases of Compiler*. URL: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_phases\\_of\\_compiler.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm) (besucht am 19.06.2022).
- *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).
- *lecture-notes-2021*. 20. Jan. 2022. URL: <https://github.com/Compiler-Construction-Uni-Freiburg/lecture-notes-2021/blob/56300e6649e32f0594bbbd046a2e19351c57dd0c/material/lexical-analysis.pdf> (besucht am 28.04.2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: <https://reverseengineering.stackexchange.com/q/17671> (besucht am 13.04.2022).
- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *What is the difference between a token and a lexeme?* NewbeDEV. URL: <http://newbedev.com/what-is-the-difference-between-a-token-and-a-lexeme> (besucht am 17.06.2022).
- *What is Top-Down Parsing?* URL: <https://www.tutorialspoint.com/what-is-top-down-parsing> (besucht am 22.06.2022).