
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

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

1	Motivation	12
1.1	RETI	12
1.2	PicoC	12
1.3	Aufgabenstellung	12
1.4	Eigenheiten der Sprache C	12
1.5	Richtlinien	13
2	Einführung	14
2.1	Compiler und Interpreter	14
2.1.1	T-Diagramme	17
2.2	Formale Sprachen	19
2.2.1	Mehrdeutige Grammatiken	20
2.2.2	Präzidenz und Assoziativität	21
2.3	Lexikalische Analyse	21
2.4	Syntaktische Analyse	24
2.5	Code Generierung	30
2.6	Fehlermeldungen	30
2.6.1	Kategorien von Fehlermeldungen	30
3	Implementierung	31
3.1	Lexikalische Analyse	31
3.1.1	Konkrete Syntax für die Lexikalische Analyse	31
3.1.2	Basic Lexer	32
3.2	Syntaktische Analyse	32
3.2.1	Konkrete Syntax für die Syntaktische Analyse	32
3.2.2	Umsetzung von Präzidenz	34
3.2.3	Derivation Tree Generierung	35
3.2.3.1	Early Parser	35
3.2.3.2	Codebeispiel	35
3.2.4	Derivation Tree Vereinfachung	36
3.2.4.1	Visitor	36
3.2.4.2	Codebeispiel	36
3.2.5	Abstrakt Syntax Tree Generierung	38
3.2.5.1	PicoC-Knoten	38
3.2.5.2	RETI-Knoten	43
3.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	44
3.2.5.4	Abstrakte Syntax	46
3.2.5.5	Transformer	48
3.2.5.6	Codebeispiel	48
3.3	Code Generierung	49
3.3.1	Übersicht	49
3.3.2	Passes	50
3.3.2.1	PicoC-Shrink Pass	50
3.3.2.1.1	Codebeispiel	50
3.3.2.2	PicoC-Blocks Pass	51
3.3.2.2.1	Abstrakte Syntax	51

3.3.2.2.2	Codebeispiel	52
3.3.2.3	PicoC-Mon Pass	53
3.3.2.3.1	Abstrakte Syntax	53
3.3.2.3.2	Codebeispiel	53
3.3.2.4	RETI-Blocks Pass	55
3.3.2.4.1	Abstrakte Syntax	55
3.3.2.4.2	Codebeispiel	55
3.3.2.5	RETI-Patch Pass	58
3.3.2.5.1	Abstrakte Syntax	58
3.3.2.5.2	Codebeispiel	58
3.3.2.6	RETI Pass	60
3.3.2.6.1	Konkrete und Abstrakte Syntax	60
3.3.2.6.2	Codebeispiel	61
3.3.3	Umsetzung von Pointern	64
3.3.3.1	Referenzierung	64
3.3.3.2	Dereferenzierung durch Zugriff auf Arrayindex ersetzen	65
3.3.4	Umsetzung von Arrays	67
3.3.4.1	Initialisierung von Arrays	67
3.3.4.2	Zugriff auf einen Arrayindex	71
3.3.4.3	Zuweisung an Arrayindex	75
3.3.5	Umsetzung von Structs	76
3.3.5.1	Deklaration von Structs	76
3.3.5.2	Initialisierung von Structs	78
3.3.5.3	Zugriff auf Structattribut	81
3.3.5.4	Zuweisung an Structattribut	83
3.3.6	Umsetzung der Derived Datatypes im Zusammenspiel	85
3.3.6.1	Einleitungsteil für Globale Statische Daten und Stackframe	85
3.3.6.2	Mittelteil für die verschiedenen Derived Datatypes	88
3.3.6.3	Schlusssteil für die verschiedenen Derived Datatypes	90
3.3.7	Umsetzung von Funktionen	95
3.3.7.1	Funktionen auflösen zu RETI Code	95
3.3.7.1.1	Sprung zur Main Funktion	97
3.3.7.2	Funktionsdeklaration und -definition und Umsetzung von Scopes	99
3.3.7.3	Funktionsaufruf	101
3.3.7.3.1	Ohne Rückgabewert	101
3.3.7.3.2	Mit Rückgabewert	103
3.3.7.3.3	Umsetzung von Call by Sharing für Arrays	106
3.3.7.3.4	Umsetzung von Call by Value für Structs	109
3.3.8	Umsetzung kleinerer Details	110
3.4	Fehlermeldungen	110
3.4.1	Error Handler	110
3.4.2	Arten von Fehlermeldungen	110
3.4.2.1	Syntaxfehler	110
3.4.2.2	Laufzeitfehler	110
4	Ergebnisse und Ausblick	111
4.1	Compiler	111
4.1.1	Überblick über Funktionen	111
4.1.2	Vergleich mit GCC	111
4.1.3	Showmode	111
4.2	Qualitätssicherung	111
4.3	Erweiterungsideen	111
A	Appendix	115

A.1 Konkrete und Abstrakte Syntax	115
A.2 Bedienungsanleitungen	115
A.2.1 PicoC-Compiler	115
A.2.2 Showmode	115
A.2.3 Entwicklertools	115

Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen	19
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen	19
2.3	Veranschaulichung der Lexikalischen Analyse	24
2.4	Veranschaulichung der Syntaktischen Analyse	29
3.1	Cross-Compiler Kompiliervorgang ausgeschrieben	49
3.2	Cross-Compiler Kompiliervorgang Kurzform	49
3.3	Architektur mit allen Passes ausgeschrieben	50
4.1	Cross-Compiler als Bootstrap Compiler	112
4.2	Iteratives Bootstrapping	114

Codeverzeichnis

3.1	PicoC Code für Derivation Tree Generierung	35
3.2	Derivation Tree nach Derivation Tree Generierung	36
3.3	Derivation Tree nach Derivation Tree Vereinfachung	37
3.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert	48
3.5	PicoC Code für Codebespiel	50
3.6	Abstract Syntax Tree für Codebespiel	51
3.7	PicoC Shrink Pass für Codebespiel	51
3.8	PicoC-Blocks Pass für Codebespiel	53
3.9	PicoC-Mon Pass für Codebespiel	55
3.10	RETI-Blocks Pass für Codebespiel	57
3.11	RETI-Patch Pass für Codebespiel	60
3.12	RETI Pass für Codebespiel	63
3.13	PicoC-Code für Pointer Referenzierung	64
3.14	Abstract Syntax Tree für Pointer Referenzierung	64
3.15	PicoC-Mon Pass für Pointer Referenzierung	65
3.16	RETI-Blocks Pass für Pointer Referenzierung	65
3.17	PicoC-Code für Pointer Dereferenzierung	66
3.18	Abstract Syntax Tree für Pointer Dereferenzierung	66
3.19	PicoC-Shrink Pass für Pointer Dereferenzierung	66
3.20	PicoC-Code für Array Initialisierung	67
3.21	Abstract Syntax Tree für Array Initialisierung	67
3.22	Symboltabelle für Array Initialisierung	68
3.23	PicoC-Mon Pass für Array Initialisierung	69
3.24	RETI-Blocks Pass für Array Initialisierung	71
3.25	PicoC-Code für Zugriff auf einen Arrayindex	71
3.26	Abstract Syntax Tree für Zugriff auf einen Arrayindex	71
3.27	PicoC-Mon Pass für Zugriff auf einen Arrayindex	73
3.28	RETI-Blocks Pass für Zugriff auf einen Arrayindex	74
3.29	PicoC-Code für Zuweisung an Arrayindex	75
3.30	Abstract Syntax Tree für Zuweisung an Arrayindex	75
3.31	PicoC-Mon Pass für Zuweisung an Arrayindex	75
3.32	RETI-Blocks Pass für Zuweisung an Arrayindex	76
3.33	PicoC-Code für Deklaration von Structs	76
3.34	Symboltabelle für Deklaration von Structs	77
3.35	PicoC-Code für Initialisierung von Structs	78
3.36	Abstract Syntax Tree für Initialisierung von Structs	78
3.37	Symboltabelle für Initialisierung von Structs	80
3.38	PicoC-Mon Pass für Initialisierung von Structs	80
3.39	RETI-Blocks Pass für Initialisierung von Structs	81
3.40	PicoC-Code für Zugriff auf Structattribut	81
3.41	Abstract Syntax Tree für Zugriff auf Structattribut	82
3.42	PicoC-Mon Pass für Zugriff auf Structattribut	82
3.43	RETI-Blocks Pass für Zugriff auf Structattribut	83
3.44	PicoC-Code für Zuweisung an Structattribut	83
3.45	Abstract Syntax Tree für Zuweisung an Structattribut	84
3.46	PicoC-Mon Pass für Zuweisung an Structattribut	84
3.47	RETI-Blocks Pass für Zuweisung an Structattribut	85

3.48	PicoC-Code für den Einleitungsteil	85
3.49	Abstract Syntax Tree für den Einleitungsteil	86
3.50	PicoC-Mon Pass für den Einleitungsteil	87
3.51	RETI-Blocks Pass für den Einleitungsteil	87
3.52	PicoC-Code für den Mittelteil	88
3.53	Abstract Syntax Tree für den Mittelteil	88
3.54	PicoC-Mon Pass für den Mittelteil	89
3.55	RETI-Blocks Pass für den Mittelteil	90
3.56	PicoC-Code für den Schlussteil	90
3.57	Abstract Syntax Tree für den Schlussteil	91
3.58	PicoC-Mon Pass für den Schlussteil	92
3.59	RETI-Blocks Pass für den Schlussteil	94
3.60	PicoC-Code für 3 Funktionen	95
3.61	Abstract Syntax Tree für 3 Funktionen	95
3.62	RETI-Blocks Pass für 3 Funktionen	96
3.63	PicoC-Mon Pass für 3 Funktionen	96
3.64	RETI-Blocks Pass für 3 Funktionen	97
3.65	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	97
3.66	PicoC-Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	98
3.67	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	98
3.68	PicoC-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	99
3.69	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	100
3.70	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	101
3.71	PicoC-Code für Funktionsaufruf ohne Rückgabewert	101
3.72	PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert	102
3.73	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	102
3.74	RETI-Pass für Funktionsaufruf ohne Rückgabewert	103
3.75	PicoC-Code für Funktionsaufruf mit Rückgabewert	103
3.76	PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert	104
3.77	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	105
3.78	RETI-Pass für Funktionsaufruf mit Rückgabewert	105
3.79	PicoC-Code für Call by Sharing für Arrays	106
3.80	PicoC-Mon Pass für Call by Sharing für Arrays	106
3.81	Symboltabelle für Call by Sharing für Arrays	107
3.82	RETI-Block Pass für Call by Sharing für Arrays	108
3.83	PicoC-Code für Call by Value für Structs	109
3.84	PicoC-Mon Pass für Call by Value für Structs	109
3.85	RETI-Block Pass für Call by Value für Structs	110

Tabellenverzeichnis

3.1	Präzidenzregeln von PicoC	34
3.2	PicoC-Knoten Teil 1	38
3.3	PicoC-Knoten Teil 2	39
3.4	PicoC-Knoten Teil 3	40
3.5	PicoC-Knoten Teil 4	41
3.6	RETI-Knoten	43
3.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	45

Definitionsverzeichnis

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

2.39	Visitor	27
2.40	Abstrakte Syntax	28
2.41	Abstract Syntax Tree	28
2.42	Pass	30
2.43	Monadische Normalform	30
2.44	Fehlermeldung	30
3.1	Token-Knoten	42
3.2	Container-Knoten	42
3.3	Symboltabelle	53
4.1	Self-compiling Compiler	111
4.2	Minimaler Compiler	112
4.3	Bootstrap Compiler	113
4.4	Bootstrapping	113

Grammatikverzeichnis

3.1.1 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 1	31
3.1.2 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 2	32
3.2.1 Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1	33
3.2.2 Konkrete Syntax für die Syntaktische Analyse in EBNF, Teil 2	34
3.2.3 Abstrakte Syntax für L_{PiocC}	47
3.3.1 Abstrakte Syntax für L_{PicoC_Blocks}	52
3.3.2 Abstrakte Syntax für L_{PicoC_Mon}	53
3.3.3 Abstrakte Syntax für L_{RETI_Blocks}	55
3.3.4 Abstrakte Syntax für L_{RETI_Patch}	58
3.3.5 Konkrete Syntax für L_{RETI_Lex}	60
3.3.6 Konkrete Syntax für L_{RETI_Parse}	61
3.3.7 Abstrakte Syntax für L_{RETI}	61

1 Motivation

1.1 RETI

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

Definition 1.1: Caller-save Register

a

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.2: Callee-save Register

a

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

1.2 PicoC

1.3 Aufgabenstellung

1.4 Eigenheiten der Sprache C

Definition 1.3: Deklaration

a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 1.4: Definition

a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 1.5: Allokation

a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.6: Initialisierung a ^aThiemann, „Einführung in die Programmierung“.**Definition 1.7: Scope** a ^aThiemann, „Einführung in die Programmierung“.**Definition 1.8: Call by value** a ^aBast, „Programmieren in C“.**Definition 1.9: Call by reference** a ^aBast, „Programmieren in C“.

1.5 Richtlinien

2 Einführung

2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** L_{PicoC} in die **RETI-Sprache** L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**¹ und von **Tests** die **Beziehungen** in 2.2.1 zu belegen (siehe Subkapitel 4.2).

Definition 2.1: Interpreter

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

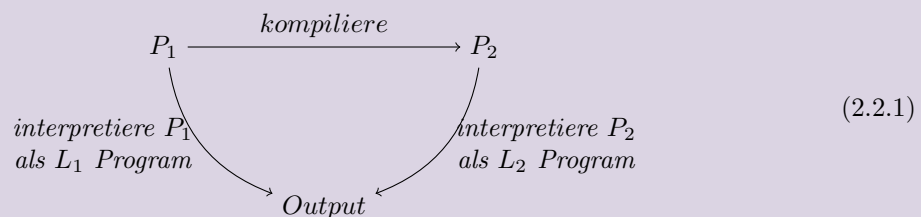
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstract Syntax Tree** (Definition 2.41) und führt je nach Komposition der **Nodes** des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.^a*

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.2: Compiler

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

*Wobei **Kompilieren** meint, dass das Program P_1 in das Program P_2 so übersetzt wird, dass bei beiden Programmen, wenn sie von **Interpretern** ihrer jeweiligen Sprachen L_1 und L_2 **interpretiert** werden, der gleiche **Output** rauskommt. Also beide Programme P_1 und P_2 die gleiche **Semantik** haben und sich nur **syntaktisch** durch die Sprachen L_1 und L_2 , in denen sie geschrieben stehen unterscheiden.^a*



^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

¹Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **GNU Compiler Collection**

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C_{i.w.k.min}^{o-j}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache L_{B_i} einer Maschine M_i kompiliert. Fall die Notwendigkeit besteht die **Maschine** M_i anzugeben, zu dessen **Maschinensprache** L_{B_i} der Compiler kompiliert, wird das als C_i geschrieben. Falls die Notwendigkeit besteht die **Sprache** L_o anzugeben, in der der Compiler selbst geschrieben ist, wird das als C^o geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert ($L_{w.k}$) oder in der er selbst geschrieben ist ($L_{o.j}$) anzugeben, wird das als $C_{w.k}^{o-j}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 4.2) kann man das als C_{min} schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschinenenncode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

Definition 2.3: Maschinensprache

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

^aViele 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.

^bC. Scholl, „Betriebssysteme“.

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

*Eine sehr **hardwarenahe** Programmiersprache, deren **Instructions** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen^a 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^b der Instructions und drumherum^{c,d}.*

^aInstructions der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

^bZ.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.

^cZ.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.

^dP. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenenncode** bzw. **Objectcode** (Definition 2.6) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschinenenncode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 2.7) zu Maschienncode zusammengesetzt wird ausführbar

ist.

Definition 2.5: Assembler

Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärer Repräsentation**, der in **Maschiensprache** geschrieben ist.^a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 2.6: Objectcode

Bei komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen wird häufig **Objectcode** erzeugt, der neben der Folge von Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.^a

^aP. Scholl, „Einführung in Embedded Systems“.

Definition 2.7: Linker

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

^aP. Scholl, „Einführung in Embedded Systems“.

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

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenem RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung².

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.^a

^aLjohhuh, *What is an immediate value?*

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

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

Kompiliert zwischen Sprachen, die ungefähr auf dem *gleichen* Level an *Abstraktion* arbeiten^{ab}

^aDie Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.

^bThiemann, „Compilerbau“.

Definition 2.10: Cross-Compiler

Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L_w geschrieben ist für eine **andere Maschine** M_2 , wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschinen Sprachen** B_1 und B_2 haben.^{ab}

^aBeim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C_{PicoC}^{Python} .

^bEarley und Sturgis, „A formalism for translator interactions“.

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

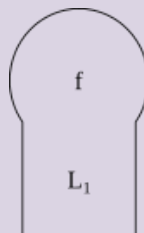
2.1.1 T-Diagramme

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

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Programm (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschine (Definition 2.14) zusammen.

Definition 2.11: T-Diagram Programm

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



^aEarley und Sturgis, „A formalism for translator interactions“.

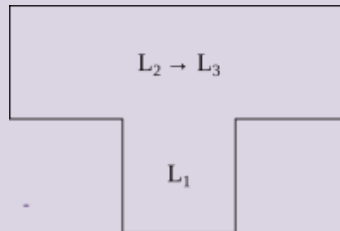
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

³Die 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 Maschinent Sprache des Microcontrollers zu kompilieren, da der Microcontroller selbst nicht genug Rechenleistung besitzt, um ein Programm selbst **zeitnah** zu kompilieren.

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

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

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

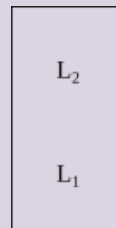


^aZwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also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

^bEarley und Sturgis, „A formalism for translator interactions“.

Definition 2.13: T-Diagramm Interpreter

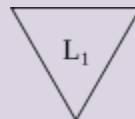
Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.^a



^aEarley und Sturgis, „A formalism for translator interactions“.

Definition 2.14: T-Diagramm Maschine

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** L_1 ausführt.^{a,b}



^aWenn 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).

^bEarley und Sturgis, „A formalism for translator interactions“.

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

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

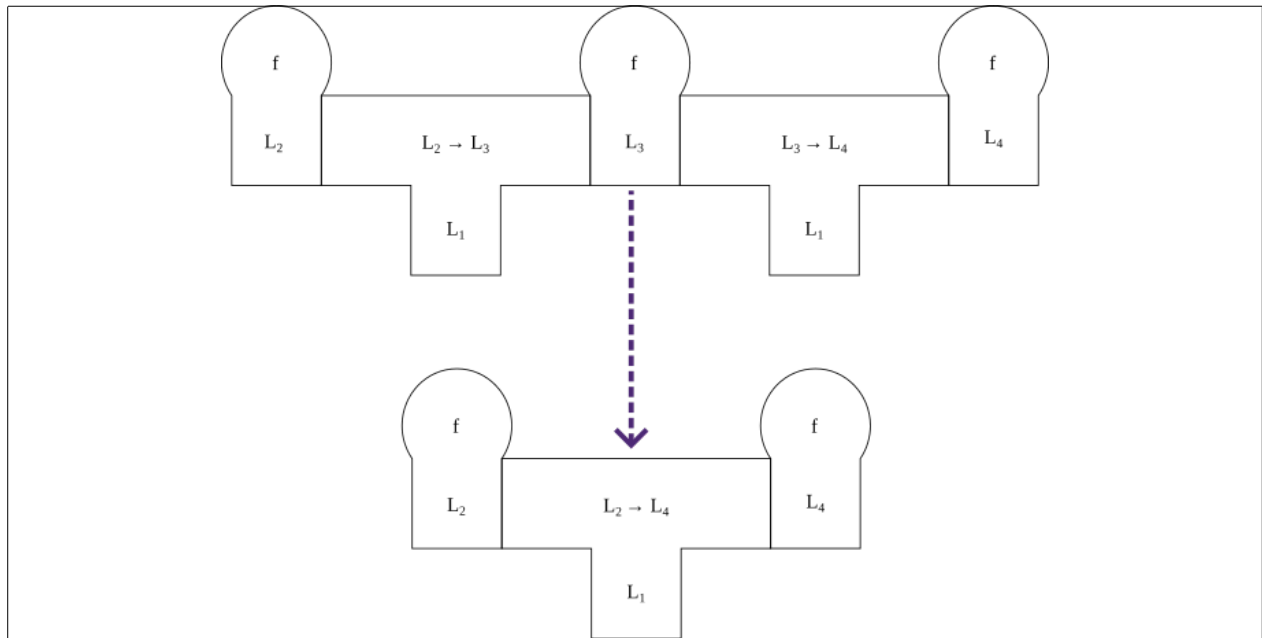


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

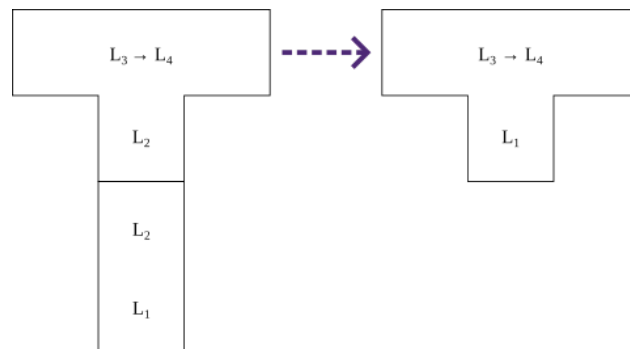


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

2.2 Formale Sprachen

Definition 2.15: Sprache

a

^aNebel, „Theoretische Informatik“.

Definition 2.16: Chomsky Hierarchie

a

^aNebel, „Theoretische Informatik“.

Definition 2.17: Grammatik*a*^aNebel, „Theoretische Informatik“.**Definition 2.18: Reguläre Sprachen***a*^aNebel, „Theoretische Informatik“.**Definition 2.19: Kontextfreie Sprachen***a*^aNebel, „Theoretische Informatik“.**Definition 2.20: Ableitung***a*^aNebel, „Theoretische Informatik“.**Definition 2.21: Links- und Rechtsableitung***a*^aNebel, „Theoretische Informatik“.**Definition 2.22: Linksrekursive Grammatiken**

Eine **Grammatik** ist **linksrekursiv**, wenn sie ein **Nicht-Terminalsymbol** enthält, dass **linksrekursiv** ist.

Ein **Nicht-Terminalsymbol** ist **linksrekursiv**, wenn das **linkeste Symbol** in einer seiner **Produktionen** es selbst ist oder zu sich selbst gemacht werden kann durch eine Folge von Ableitungen:

$$A \Rightarrow^* Aa,$$

wobei *a* eine beliebige Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen** ist.^a

^aParsing Expressions · Crafting Interpreters.**2.2.1 Mehrdeutige Grammatiken****Definition 2.23: Ableitungsbaum***a*^aNebel, „Theoretische Informatik“.

Definition 2.24: Mehrdeutige Grammatik*a*^aNebel, „Theoretische Informatik“.**2.2.2 Präzidenz und Assoziativität****Definition 2.25: Assoziativität***a*^a*Parsing Expressions · Crafting Interpreters.***Definition 2.26: Präzidenz***a*^a*Parsing Expressions · Crafting Interpreters.***Definition 2.27: Wortproblem***a*^aNebel, „Theoretische Informatik“.**Definition 2.28: LL(k)-Grammatik**

Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die nächsten k **Symbole** des **Eingabeworts** bzw. in Bezug zu Compilerbau **Token** des **Inputstrings** zu bestimmen ist^a. Dabei steht **LL** für *left-to-right* und *leftmost-derivation*, da das **Eingabewort** von **links nach rechts** geparsed und immer **Linksableitungen** genommen werden müssen^b, damit die obige Bedingung mit den **nächsten** k Symbolen gilt.^c

^aDas wird auch als **Lookahead** von k bezeichnet.^bWobei 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.^cNebel, „Theoretische Informatik“.**2.3 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^a, die für die Beschreibung eines **Tokens** T zuständig sind.^b

^aAls Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

^bThiemann, „Compilerbau“.

Definition 2.30: Lexeme

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

^aThiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.31) im **Inputstring** identifiziert und **Tokens** T zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** (Definition 2.31) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

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

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

^aThiemann, „Compilerbau“.

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

Um Verwirrung verzubäuen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

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

In der Code Generierung sind die **Bezeichner** (Definition 2.32) von **Variablen, Konstanten und Funktionen** die Symbole^a.

^aDas ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel 3 **Symboldabelle** genannt wird.

Definition 2.32: Bezeichner (bzw. Identifier)

*Tokenwert, der eine Konstante, Variable, Funktion usw. **eindeutig** benennt.^{a,b}*

^aAußer 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.

^bThiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`⁴ und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** $*$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie 42, 314 oder 12. Die **Überbegriffe** bzw. **Tokennamen** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`⁵, bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie `if` oder `}` sind die **Tokennamen** bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.33) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99⁶. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik** G_{Lex} , die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut⁷, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik ?? liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.18 erfüllt. Einzig die Produktion `CHAR ::= "'ASCII_CHAR'"` sieht problematisch aus, kann allerdings auch als `{CHAR ::= "'CHAR2', CHAR2 ::= ASCII_CHAR'"}` **regulär** ausgedrückt werden⁸. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

⁴In 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.

⁵Diese **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.

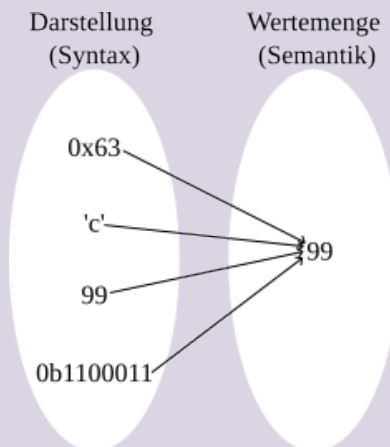
⁶Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

⁷Man nennt das auch einem **Lookahead** von 1

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

Definition 2.33: Literal

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



^aThiemann, „Einführung in die Programmierung“.

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

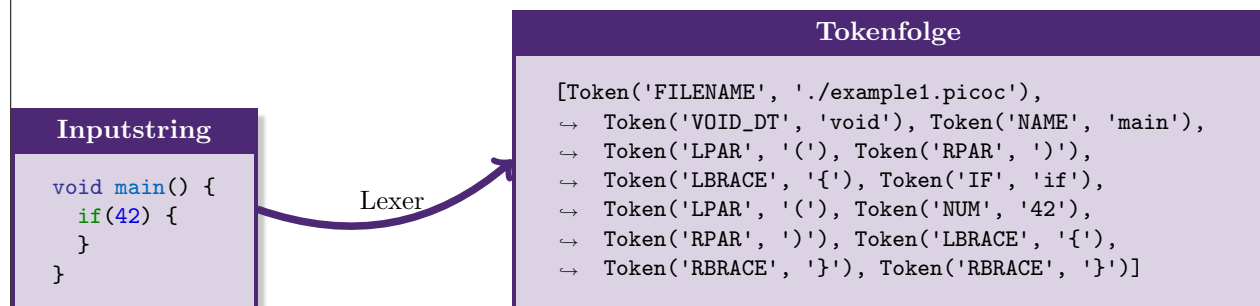


Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern '(' bzw. öffnende geschweifte Klammern '{' es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden.

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

Definition 2.34: Konkrete Syntax

Syntax einer *Sprache*, die durch die *Grammatiken* G_{Lex} und G_{Parse} zusammengenommen beschrieben wird.

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

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.35: Derivation Tree (bzw. Parse Tree)

Compilerinterne Darstellung eines in *Konkreter Syntax* geschriebenen Inputstrings als *Baumdatenstruktur*, in der *Nichtterminalsymbole* die *Inneren Knoten* der Baumdatenstruktur und *Terminalsymbole* die *Blätter* der Baumdatenstruktur bilden. Jedes zum Ableiten des Inputstrings verwendete *Nicht-Terminalsymbol* einer *Produktion* der *Grammatik* G_{Parse} , die ein Teil der *Konkrete Syntax* ist, bildet einen eigenen *Inneren Knoten*.

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

^aJSON parser - Tutorial — Lark documentation.

Definition 2.36: Parser

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

^aEs gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von *Konkreter Syntax* in *Abstrakte Syntax* übersetzt. Im Folgenden wird allerdings die Definition 2.36 verwendet.

^bJSON parser - Tutorial — Lark documentation.

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

In Bezug auf Compilerbau ist ein *Lexer* ein Teil eines *Parsers*. Der *Lexer* ist ausschließlich für die *Lexikalische Analyse* verantwortlich und entspricht z.B., wenn man bei einem Wanderausflug verschiedenen Insekten entdeckt, dem Nachschlagen in einem Insektenlexikon und dem Aufschreiben, welchen Insekten man in welcher *Reihenfolge* begegnet ist. Zudem kann man bestimmte *Sehenswürdigkeiten* an denen man während des Ausflugs vorbeikommt ebenfalls festhalten, da es eine Rolle spielen kann in welchem örtlichen *Kontext* man den Insekten begegnet ist.^a

Der *Parser* vereinigt sowohl die *Lexikalische Analyse*, als auch einen Teil der *Syntaktischen Analyse* in sich und entspricht, um auf das Beispiel zurückzukommen, dem Darstellen von *Beziehungen* zwischen den Insektenbegegnungen in einer für die *Weiterverarbeitung tauglichen Form*.^b

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

^aDas würde z.B. der Rolle eines *Semikolon* ; in der Sprache L_{PicoC} entsprechen.

^bZ.B. gibt es bestimmte *Wechselbeziehungen* zwischen Insekten, Insekten beeinflussen sich gegenseitig.

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

^aDas vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

^bThiemann, „Compilerbau“.

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

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

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

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

Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg**. Dabei wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminal** enthält.

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

Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.28) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewünschte Inputstring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass der **Inputstring** sich **nicht** mit der verwendeten Grammatik

ableiten lässt.^b

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

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

^aWhat is Top-Down Parsing?

^bDiese Form von Parsing wurde im **PicoC-Compiler** implementiert, als dieser noch auf dem Stand des **Bachelorprojektes** war, bevor er durch den nicht selbst implementierten **Earley Parser** von **Lark** (siehe *Lark - a parsing toolkit for Python*) ersetzt wurde.

^cDiese Art von Parser ist im **RETI-Interpreter** implementiert, da die **RETI-Sprache** eine besonders simple **LL(1) Grammatik** besitzt. Diese Art von **Parser** wird auch als **Predictive Parser** oder **LL(k) Recursive Descent Parser** bezeichnet, wobei **Recursive Descent** das englische Wort für **Rekursiven Abstieg** ist.

^dWhat is Bottom-up Parsing?

^eDer **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

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

Definition 2.38: Transformer

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

^aTransformers & Visitors — Lark documentation.

Definition 2.39: Visitor

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

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

^bTransformers & Visitors — Lark documentation.

Definition 2.40: Abstrakte Syntax

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

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

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.41: Abstract Syntax Tree

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

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

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstract Syntax Tree** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.4 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

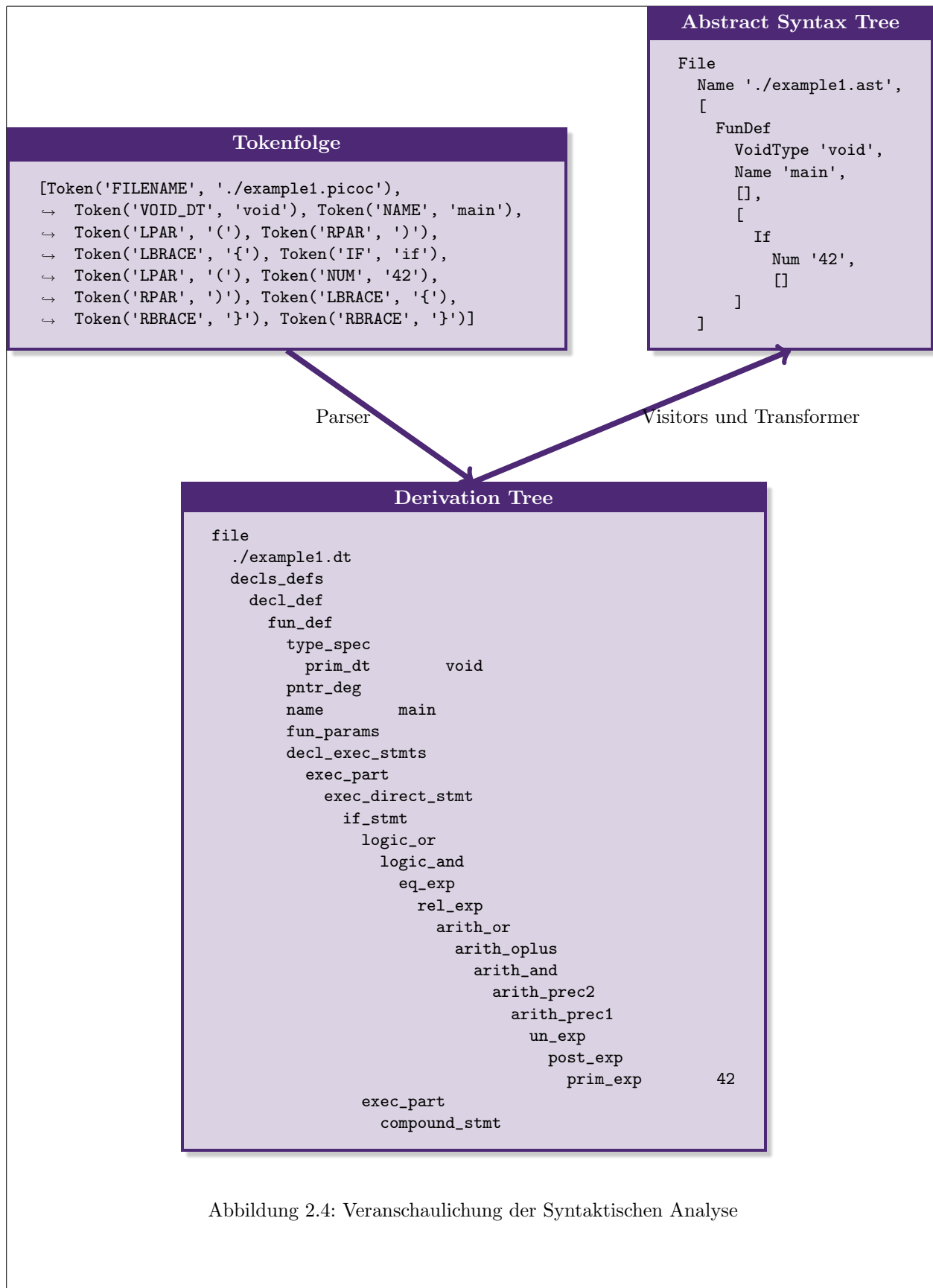


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse

2.5 Code Generierung

Definition 2.42: Pass

a

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 2.43: Monadische Normalform

a

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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

2.6 Fehlermeldungen

Definition 2.44: Fehlermeldung

Benachrichtigung beliebiger Form, die darüber informiert, dass:

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

^aErrors in C/C++ - GeeksforGeeks.

2.6.1 Kategorien von Fehlermeldungen

3 Implementierung

3.1 Lexikalische Analyse

3.1.1 Konkrete Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	"//"/[\backslash n]*"/ "/*"/(\cdot \backslash n)*?/"*/"	<i>L_Comment</i>
<i>RETI.COMMENT.2</i>	::=	"//""?"#"/[\backslash n]*/	
<i>DIG.NO_0</i>	::=	"1" "2" "3" "4" "5" "6" "7" "8" "9"	<i>L_Arith</i>
<i>DIG.WITH_0</i>	::=	"0" <i>DIG.NO_0</i>	
<i>NUM</i>	::=	"0" <i>DIG.NO_0</i> <i>DIG.WITH_0</i> *	
<i>ASCII.CHAR</i>	::=	"_".." ~ "	
<i>CHAR</i>	::=	"'" <i>ASCII.CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII.CHAR</i> + ".picoc"	
<i>LETTER</i>	::=	"a".."z" "A".."Z"	
<i>NAME</i>	::=	(<i>LETTER</i> "_") (<i>LETTER</i> — <i>DIG.WITH_0</i> — "_")*	
<i>name</i>	::=	<i>NAME</i> <i>INT.NAME</i> <i>CHAR.NAME</i> <i>VOID.NAME</i>	
<i>NOT</i>	::=	" ~ "	
<i>REF_AND</i>	::=	"&"	
<i>un_op</i>	::=	<i>SUB.MINUS</i> <i>LOGIC.NOT</i> <i>NOT</i> <i>MUL.DEREF.PNTR</i> <i>REF_AND</i>	
<i>MUL.DEREF.PNTR</i>	::=	"*"	
<i>DIV</i>	::=	"/"	
<i>MOD</i>	::=	"%"	
<i>prec1_op</i>	::=	<i>MUL.DEREF.PNTR</i> <i>DIV</i> <i>MOD</i>	
<i>ADD</i>	::=	"+"	
<i>SUB.MINUS</i>	::=	"_"	
<i>prec2_op</i>	::=	<i>ADD</i> <i>SUB.MINUS</i>	
<i>LT</i>	::=	"<"	<i>L_Logic</i>
<i>LTE</i>	::=	"<="	
<i>GT</i>	::=	">"	
<i>GTE</i>	::=	">="	
<i>rel_op</i>	::=	<i>LT</i> <i>LTE</i> <i>GT</i> <i>GTE</i>	
<i>EQ</i>	::=	"=="	
<i>NEQ</i>	::=	"!="	
<i>eq_op</i>	::=	<i>EQ</i> <i>NEQ</i>	
<i>LOGIC.NOT</i>	::=	"!"	

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

<i>INT_DT.2</i>	::=	"int"	<i>L_Assign_Alloc</i>
<i>INT_NAME.3</i>	::=	"int" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>prim_dt</i>	::=	<i>INT_DT</i> <i>CHAR_DT</i> <i>VOID_DT</i>	

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

3.1.2 Basic Lexer

3.2 Syntaktische Analyse

3.2.1 Konkrete Syntax für die Syntaktische Analyse

In 3.2.1

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

Grammar 3.2.1: Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1

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

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

3.2.2 Umsetzung von Präzidenz

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

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

Tabelle 3.1: Präzidenzregeln von PicoC

¹*C Operator Precedence - cpreference.com.*

3.2.3 Derivation Tree Generierung

3.2.3.1 Early Parser

3.2.3.2 Codebeispiel

```

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

```

Code 3.1: PicoC Code für Derivation Tree Generierung

```

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

```

```

41         array_decl
42         name var
43         array_dims
44     array_dims
45     3
46     2

```

Code 3.2: Derivation Tree nach Derivation Tree Generierung

3.2.4 Derivation Tree Vereinfachung

3.2.4.1 Visitor

3.2.4.2 Codebeispiel

Beispiel aus Subkapitel 3.2.3.2 wird fortgeführt.

```

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

```

```
37         3
38         2
39     pntr_decl
40     pntr_deg *
41     array_decl
42     array_dims
43     type_spec
44     struct_spec
45     name st
46 name var
```

Code 3.3: Derivation Tree nach Derivation Tree Vereinfachung

3.2.5 Abstrakt Syntax Tree Generierung

3.2.5.1 PicoC-Knoten

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

Tabelle 3.2: PicoC-Knoten Teil 1

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

Tabelle 3.3: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initializer eines Structs , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines Attributezeichners , zu einem weiteren Initializer für eine Array-Dimension oder zu einem Initializer für ein Struct oder zu einem Logischen Ausdruck . Des Weiteren besitzt er ein verstecktes Attribut datatype, welches für den PicoC-Mon Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Structdatentyps , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der Bezeichner des Structdatentyps ist und allocs eine Liste von Bezeichnern der Attribute des Structdatentyps mit dazugehörigem Datentyp , wofür sich der Container-Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.
If(exp, stmts_goto)	Container für ein If Statement if(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts_goto, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts_goto1, stmts_goto2)	Container für ein If-Else Statement if(<exp>) { <stmts2> } else { <stmts2> } inklusive Condition exp und 2 Branches stmts_goto1 und stmts_goto2, die zwei Alternativen darstellen in denen jeweils Listen von Statements oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts_goto)	Container für ein While-Statement while(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts_goto, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts_goto)	Container für ein Do-While-Statement do { <stmts> } while(<exp>); inklusive Condition exp und einem Branch stmts_goto, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen Funktionsaufruf : fun.name(exps), wobei name der Bezeichner der Funktion ist, die aufgerufen werden soll und exps eine Liste von Argumenten ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein Return-Statement : return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis vom Return-Statement zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine Funktionsdeklaration , z.B. <datatype> <fun.name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist und allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient.

Tabelle 3.4: PicoC-Knoten Teil 3

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts_blocks)	Container für eine Funktionsdefinition , z.B. <datatype> <fun_name>(<datatype> <param>) {<stmts>}, wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist, allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts_blocks eine Liste von Statements bzw. Blöcken ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die Erstellung eines neuen Stackframes , wobei fun_name der Bezeichner der Funktion ist, für die ein neuer Stackframe erstellt werden soll und später dazu dient den Block dieser Funktion zu finden, weil dieser für den weiteren Kompilervorang wichtige Information in seinen versteckten Attribute angehängt hat und goto_after_call ein GoTo(Name('addr@next_instr')) ist, welches später durch die Adresse der Instruction, die direkt auf die Jump Instruction folgt, ersetzt wird.
RemoveStackframe()	Container für das Entfernen des aktuellen Stackframes .
File(name, decls_defs_blocks)	Container für alle Funktionen oder Blöcke , welche eine Datei als Ursprung haben, wobei name der Dateiname der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von Funktionen bzw. Blöcken ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für Statements , der auch als Block bezeichnet wird, wobei das Attribut name der Bezeichners des Labels des Blocks ist und stmts_instrs eine Liste von Statements oder Instructions . Zudem besitzt er noch 3 versteckte Attribute, wobei instrs_before die Zahl der Instructions vor diesem Block zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die Parameter der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für ein Goto zu einem anderen Block , wobei das Attribut name der Bezeichner des Labels des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen Kommentar , den der Compiler selber während des Kompilervorangs erstellt, der im RETI-Interpreter selbst später nicht sichtbar sein wird, aber in den Immediate-Dateien , welche die Abstract Syntax Trees nach den verschiedenen Passes enthalten.
RETIComment(value)	Container für einen Kommentar im Code der Form: // # comment, der im RETI-Interpreter später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer RETI-CPU nicht umsetzbar ist und auch nicht sinnvoll wäre umzusetzen. Der Kommentar ist im Attribut value , welches jeder Knoten besitzt gespeichert.

Tabelle 3.5: PicoC-Knoten Teil 4

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

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 3.1) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 3.2) unbesetzt ist. Das **Attribut position** wird später für Fehlermeldungen gebraucht.

Definition 3.1: Token-Knoten

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

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

^aThiemann, „Compilerbau“.

Definition 3.2: Container-Knoten

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

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

^aWie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. **1 % 3** und **Logischen Ausdrücke**, wie z.B. **1 && 2 < 3**, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.

^bThiemann, „Compilerbau“.

3.2.5.2 RETI-Knoten

RETI-Knoten	Beschreibung
Program(name, instrs)	Container für alle Instructions : <name> <instrs>, wobei name der Dateiname der Datei ist, die erstellt wird und instrs eine Liste von Instructions ist.
Instr(op, args)	Container für eine Instruction : <op> <args>, wobei op eine Operation ist und args eine Liste von Argumenten für dieser Operation.
Jump(rel, im_goto)	Container für eine Jump-Instruction : JUMP<rel> <im>, wobei rel eine Relation ist und im_goto ein Immediate Value Im(val) für die Anzahl an Speicherzellen , um die relativ zur Jump-Instruction gesprungen werden soll oder ein GoTo(Name('block.xyz')) , das später im RETI-Patch Pass durch einen passenden Immediate Value ersetzt wird.
Int(num)	Container für einen Interruptaufruf : INT <im>, wobei num die Interruptvektornummer (IVN) für die passende Speicherzelle in der Interruptvektortabelle ist, in der die Adresse der Interrupt-Service-Routine (ISR) steht.
Call(name, reg)	Container für einen Prozeduraufruf : CALL <name> <reg>, wobei name der Bezeichner der Prozedur, die aufgerufen werden soll ist und reg ein Register ist, das als Argument an die Prozedur dient. Diese Operation ist in der Betriebssysteme Vorlesung ^a nicht deklariert, sondern wurde dazuerfunden, um unkompliziert ein CALL PRINT ACC oder CALL INPUT ACC im RETI-Interpreter simulieren zu können.
Name(val)	Bezeichner für eine Prozedur , z.B. PRINT oder INPUT oder den Programmnamen , z.B. PROGRAMNAME. Dieses Argument ist in der Betriebssysteme Vorlesung ^a nicht deklariert, sondern wurde dazuerfunden, um Bezeichner, wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein Register .
Im(val)	Ein Immediate Value , z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), Oplus(), Or(), And()	Compute-Memory oder Compute-Register Operationen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(), Oplusi(), Ori(), Andi()	Compute-Immediate Operationen: ADDI, SUBI, MULTI, DIVI, MODI, OPLUSI, ORI, ANDI.
Load(), Loadin(), Loadi()	Load Operationen: LOAD, LOADIN, LOADI.
Store(), Storein(), Move()	Store Operationen: STORE, STOREIN, MOVE.
Lt(), LtE(), Gt(), GtE(), Eq(), NEq(), Always(), NOP()	Relationen : <, <=, >, >=, ==, !=, _NOP.
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()	Register : PC, IN1, IN2, ACC, SP, BAF, CS, DS.

^a C. Scholl, „Betriebssysteme“

Tabelle 3.6: RETI-Knoten

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

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

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

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

Um die obige Tabelle 3.7 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch $\text{Stack}(\text{Num}('x')), x \in \mathbb{N}$ ersetzt wurde.

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

3.2.5.4 Abstrakte Syntax

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

Grammar 3.2.3: Abstrakte Syntax für *L_Piocc*

3.2.5.5 Transformer

3.2.5.6 Codebeispiel

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

```
1 File
2   Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8         Writeable,
9         PtrDecl
10        Num '1',
11        ArrayDecl
12        [
13          Num '5',
14          Num '6'
15        ],
16        PtrDecl
17        Num '1',
18        IntType 'int',
19        Name 'attr'
20      ],
21    FunDef
22      VoidType 'void',
23      Name 'main',
24      [],
25      [
26        Exp(Alloc(Writeable(), PtrDecl(Num('1')), ArrayDecl([Num('3'), Num('2')],
27          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var')))
28      ]
29    ]
```

Code 3.4: Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert

3.3 Code Generierung

3.3.1 Übersicht

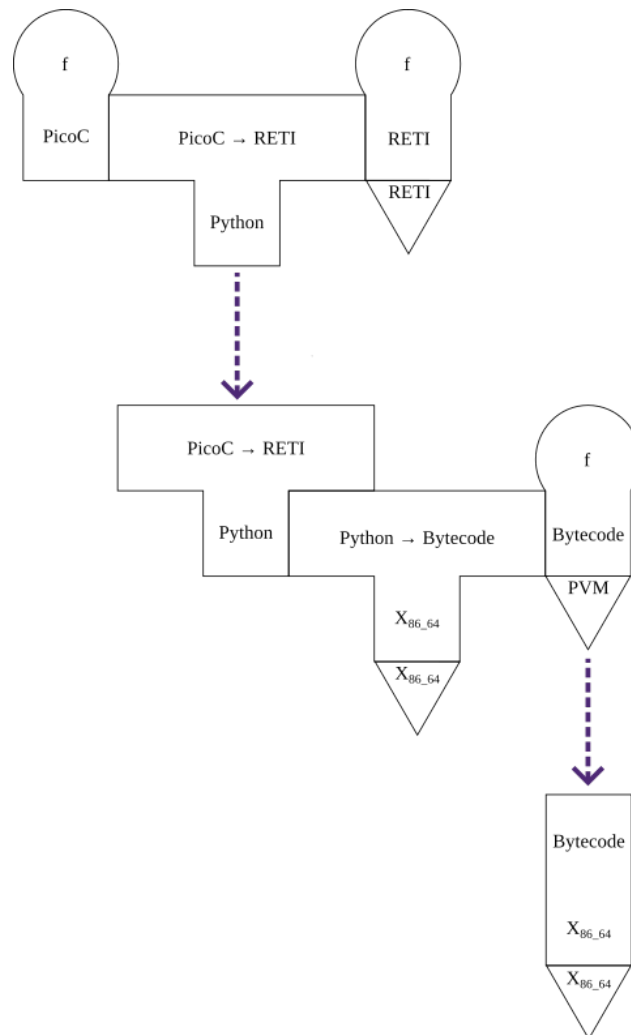


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

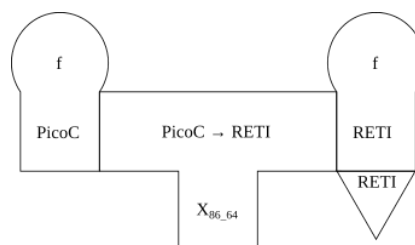


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

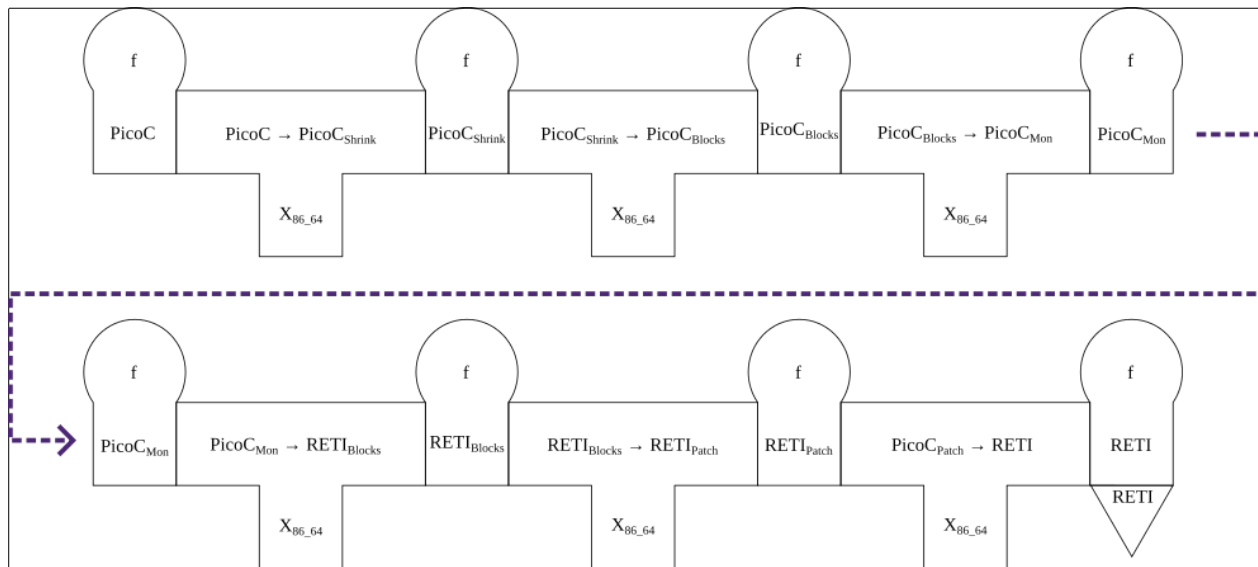


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

3.3.2 Passes

3.3.2.1 PicoC-Shrink Pass

3.3.2.1.1 Codebeispiel

```

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

```

Code 3.5: PicoC Code für Codebeispiel

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [

```

```

9      Assign(Alloc(Writable(), IntType('int'), Name('n')), Num('4'))
10     Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
11     While
12       Num '1',
13       [
14         If
15         Atom
16           Name 'n',
17           Eq '==',
18           Num '1',
19           [
20             Return(Empty())
21           ]
22         Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23         Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24       ]
25   ]
26 ]

```

Code 3.6: Abstract Syntax Tree für Codebeispiel

```

1 File
2   Name './example_faculty_it.picoc_shrink',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), IntType('int'), Name('n')), Num('4'))
10        Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
11        While
12          Num '1',
13          [
14            If
15            Atom
16              Name 'n',
17              Eq '==',
18              Num '1',
19              [
20                Return(Empty())
21              ]
22            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24          ]
25        ]
26      ]

```

Code 3.7: PicoC Shrink Pass für Codebeispiel

3.3.2.2 PicoC-Blocks Pass

3.3.2.2.1 Abstrakte Syntax

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

Grammar 3.3.1: Abstrakte Syntax für *LPicoC_Blocks*

3.3.2.2.2 Codebeispiel

```

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

```

```

44     ],
45     Block
46     Name 'if.2',
47     [
48         Return(Empty())
49     ],
50     Block
51     Name 'if_else_after.1',
52     [
53         Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
54         Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
55         GoTo(Name('condition_check.4'))
56     ],
57     Block
58     Name 'while_after.0',
59     []
60 ]
61 ]

```

Code 3.8: PicoC-Blocks Pass für Codebeispiel

3.3.2.3 PicoC-Mon Pass

3.3.2.3.1 Abstrakte Syntax

<i>ref_loc</i>	::= <i>Stack</i> (<i>Num</i> (<i>str</i>)) <i>Global</i> (<i>Num</i> (<i>str</i>))	<i>L_Assign_Alloc</i>
	<i>Stackframe</i> (<i>Num</i> (<i>str</i>))	
<i>error_data</i>	::= $\langle exp \rangle$ <i>Pos</i> (<i>Num</i> (<i>str</i>), <i>Num</i> (<i>str</i>))	
<i>exp</i>	::= <i>Stack</i> (<i>Num</i> (<i>str</i>)) <i>Ref</i> ($\langle ref_loc \rangle$, $\langle datatype \rangle$, $\langle error_data \rangle$)	
<i>stmt</i>	::= <i>Exp</i> ($\langle exp \rangle$)	
	<i>Assign</i> (<i>Alloc</i> (<i>Writeable</i> ()), <i>StructSpec</i> (<i>Name</i> (<i>str</i>), <i>Name</i> (<i>str</i>)),	
	<i>Struct</i> (<i>Assign</i> (<i>Name</i> (<i>str</i>), $\langle exp \rangle$)+, $\langle datatype \rangle$))	
	<i>Assign</i> (<i>Alloc</i> (<i>Writeable</i> ()), <i>ArrayDecl</i> (<i>Num</i> (<i>str</i>)+, $\langle datatype \rangle$),	
	<i>Name</i> (<i>str</i>), <i>Array</i> ($\langle exp \rangle$ +, $\langle datatype \rangle$))	
<i>symbol_table</i>	::= <i>SymbolTable</i> ($\langle symbol \rangle$)	<i>L_Symbol_Table</i>
<i>symbol</i>	::= <i>Symbol</i> ($\langle type_qual \rangle$, $\langle datatype \rangle$, $\langle name \rangle$, $\langle val \rangle$, $\langle pos \rangle$, $\langle size \rangle$)	
<i>type_qual</i>	::= <i>Empty</i> ()	
<i>datatype</i>	::= <i>BuiltIn</i> () <i>SelfDefined</i> ()	
<i>name</i>	::= <i>Name</i> (<i>str</i>)	
<i>val</i>	::= <i>Num</i> (<i>str</i>) <i>Empty</i> ()	
<i>pos</i>	::= <i>Pos</i> (<i>Num</i> (<i>str</i>), <i>Num</i> (<i>str</i>)) <i>Empty</i> ()	
<i>size</i>	::= <i>Num</i> (<i>str</i>) <i>Empty</i> ()	

Grammar 3.3.2: Abstrakte Syntax für L_{PicoC_Mon}

Definition 3.3: Symboltabelle

3.3.2.3.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'main.5',
6       [
7         // Assign(Name('n'), Num('4'))
8         Exp(Num('4'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('res'), Num('1'))
11        Exp(Num('1'))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        // While(Num('1'), [])
14        Exp(GoTo(Name('condition_check.4'))))
15      ],
16    Block
17      Name 'condition_check.4',
18      [
19        // IfElse(Num('1'), [], [])
20        Exp(Num('1')),
21        IfElse
22          Stack
23            Num '1',
24            [
25              GoTo(Name('while_branch.3'))
26            ],
27            [
28              GoTo(Name('while_after.0'))
29            ]
30        ],
31      Block
32        Name 'while_branch.3',
33        [
34          // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35          // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36          Exp(Global(Num('0')))
37          Exp(Num('1'))
38          Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
39          IfElse
40            Stack
41              Num '1',
42              [
43                GoTo(Name('if.2'))
44              ],
45              [
46                GoTo(Name('if_else_after.1'))
47              ]
48        ],
49      Block
50        Name 'if.2',
51        [
52          Return(Empty())
53        ],
54      Block
55        Name 'if_else_after.1',
56        [
57          // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))

```

```

58     Exp(Global(Num('0')))
59     Exp(Global(Num('1')))
60     Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
61     Assign(Global(Num('1')), Stack(Num('1')))
62     // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
63     Exp(Global(Num('0')))
64     Exp(Num('1'))
65     Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
66     Assign(Global(Num('0')), Stack(Num('1')))
67     Exp(GoTo(Name('condition_check.4')))
68 ],
69 Block
70   Name 'while_after.0',
71   [
72     Return(Empty())
73   ]
74 ]

```

Code 3.9: PicoC-Mon Pass für Codebeispiel

3.3.2.4 RETI-Blocks Pass

3.3.2.4.1 Abstrakte Syntax

<i>program</i>	$::=$	$Program(Name(str), \langle block \rangle^*)$	$L_Program$
<i>exp_stmts</i>	$::=$	$GoTo(str)$	L_Blocks
<i>instrs_before</i>	$::=$	$Num(str)$	
<i>num_instrs</i>	$::=$	$Num(str)$	
<i>block</i>	$::=$	$Block(Name(str), \langle instr \rangle^*, \langle instrs_before \rangle, \langle num_instrs \rangle)$	
<i>instr</i>	$::=$	$GoTo(Name(str))$	

Grammar 3.3.3: Abstrakte Syntax für L_{RETI_Blocks}

3.3.2.4.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         # // Assign(Name('n'), Num('4'))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('res'), Num('1'))
17        # Exp(Num('1'))

```



```

18     SUBI SP 1;
19     LOADI ACC 1;
20     STOREIN SP ACC 1;
21     # Assign(Global(Num('1')), Stack(Num('1')))
22     LOADIN SP ACC 1;
23     STOREIN DS ACC 1;
24     ADDI SP 1;
25     # // While(Num('1'), [])
26     # Exp(GoTo(Name('condition_check.4')))
27     Exp(GoTo(Name('condition_check.4')))
28 ],
29 Block
30   Name 'condition_check.4',
31   [
32     # // IfElse(Num('1'), [], [])
33     # Exp(Num('1'))
34     SUBI SP 1;
35     LOADI ACC 1;
36     STOREIN SP ACC 1;
37     # IfElse(Stack(Num('1')), [], [])
38     LOADIN SP ACC 1;
39     ADDI SP 1;
40     JUMP== GoTo(Name('while_after.0'));
41     Exp(GoTo(Name('while_branch.3')))
42   ],
43 Block
44   Name 'while_branch.3',
45   [
46     # // If(Atom(Name('n'), Eq('='), Num('1')), [])
47     # // IfElse(Atom(Name('n'), Eq('='), Num('1')), [], [])
48     # Exp(Global(Num('0')))
49     SUBI SP 1;
50     LOADIN DS ACC 0;
51     STOREIN SP ACC 1;
52     # Exp(Num('1'))
53     SUBI SP 1;
54     LOADI ACC 1;
55     STOREIN SP ACC 1;
56     # Exp(Atom(Stack(Num('2')), Eq('='), Stack(Num('1'))))
57     LOADIN SP ACC 2;
58     LOADIN SP IN2 1;
59     SUB ACC IN2;
60     JUMP== 3;
61     LOADI ACC 0;
62     JUMP 2;
63     LOADI ACC 1;
64     STOREIN SP ACC 2;
65     ADDI SP 1;
66     # IfElse(Stack(Num('1')), [], [])
67     LOADIN SP ACC 1;
68     ADDI SP 1;
69     JUMP== GoTo(Name('if_else_after.1'));
70     Exp(GoTo(Name('if.2')))
71   ],
72 Block
73   Name 'if.2',
74   [

```

```

75     # Return(Empty())
76     LOADIN BAF PC -1;
77 ],
78 Block
79     Name 'if_else_after.1',
80     [
81         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
82         # Exp(Global(Num('0')))
83         SUBI SP 1;
84         LOADIN DS ACC 0;
85         STOREIN SP ACC 1;
86         # Exp(Global(Num('1')))
87         SUBI SP 1;
88         LOADIN DS ACC 1;
89         STOREIN SP ACC 1;
90         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
91         LOADIN SP ACC 2;
92         LOADIN SP IN2 1;
93         MULT ACC IN2;
94         STOREIN SP ACC 2;
95         ADDI SP 1;
96         # Assign(Global(Num('1')), Stack(Num('1')))
97         LOADIN SP ACC 1;
98         STOREIN DS ACC 1;
99         ADDI SP 1;
100        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
101        # Exp(Global(Num('0')))
102        SUBI SP 1;
103        LOADIN DS ACC 0;
104        STOREIN SP ACC 1;
105        # Exp(Num('1'))
106        SUBI SP 1;
107        LOADI ACC 1;
108        STOREIN SP ACC 1;
109        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
110        LOADIN SP ACC 2;
111        LOADIN SP IN2 1;
112        SUB ACC IN2;
113        STOREIN SP ACC 2;
114        ADDI SP 1;
115        # Assign(Global(Num('0')), Stack(Num('1')))
116        LOADIN SP ACC 1;
117        STOREIN DS ACC 0;
118        ADDI SP 1;
119        # Exp(GoTo(Name('condition_check.4')))
120        Exp(GoTo(Name('condition_check.4')))
121    ],
122    Block
123        Name 'while_after.0',
124        [
125            # Return(Empty())
126            LOADIN BAF PC -1;
127        ]
128 ]

```

Code 3.10: RETI-Blocks Pass für Codebespiel

3.3.2.5 RETI-Patch Pass

3.3.2.5.1 Abstrakte Syntax

$$\text{stmt} ::= \text{Exit}(\text{Num}(\text{str}))$$

Grammar 3.3.4: Abstrakte Syntax für L_{RETI_Patch}

3.3.2.5.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.6',
6       [
7         # // Exp(GoTo(Name('main.5')))
8         # // patched out Exp(GoTo(Name('main.5')))
9       ],
10    Block
11      Name 'main.5',
12      [
13        # // Assign(Name('n'), Num('4'))
14        # Exp(Num('4'))
15        SUBI SP 1;
16        LOADI ACC 4;
17        STOREIN SP ACC 1;
18        # Assign(Global(Num('0')), Stack(Num('1')))
19        LOADIN SP ACC 1;
20        STOREIN DS ACC 0;
21        ADDI SP 1;
22        # // Assign(Name('res'), Num('1'))
23        # Exp(Num('1'))
24        SUBI SP 1;
25        LOADI ACC 1;
26        STOREIN SP ACC 1;
27        # Assign(Global(Num('1')), Stack(Num('1')))
28        LOADIN SP ACC 1;
29        STOREIN DS ACC 1;
30        ADDI SP 1;
31        # // While(Num('1'), [])
32        # Exp(GoTo(Name('condition_check.4')))
33        # // patched out Exp(GoTo(Name('condition_check.4')))
34      ],
35    Block
36      Name 'condition_check.4',
37      [
38        # // IfElse(Num('1'), [], [])
39        # Exp(Num('1'))
40        SUBI SP 1;
41        LOADI ACC 1;
42        STOREIN SP ACC 1;
43        # IfElse(Stack(Num('1')), [], [])
44        LOADIN SP ACC 1;
45        ADDI SP 1;

```

```

46     JUMP== GoTo(Name('while_after.0'));
47     # // patched out Exp(GoTo(Name('while_branch.3')))
48 ],
49 Block
50     Name 'while_branch.3',
51     [
52         # // If(Atom(Name('n'), Eq('=='), Num('1')), [], [])
53         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
54         # Exp(Global(Num('0')))
55         SUBI SP 1;
56         LOADIN DS ACC 0;
57         STOREIN SP ACC 1;
58         # Exp(Num('1'))
59         SUBI SP 1;
60         LOADI ACC 1;
61         STOREIN SP ACC 1;
62         # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
63         LOADIN SP ACC 2;
64         LOADIN SP IN2 1;
65         SUB ACC IN2;
66         JUMP== 3;
67         LOADI ACC 0;
68         JUMP 2;
69         LOADI ACC 1;
70         STOREIN SP ACC 2;
71         ADDI SP 1;
72         # IfElse(Stack(Num('1')), [], [])
73         LOADIN SP ACC 1;
74         ADDI SP 1;
75         JUMP== GoTo(Name('if_else_after.1'));
76         # // patched out Exp(GoTo(Name('if.2')))
77     ],
78 Block
79     Name 'if.2',
80     [
81         # Return(Empty())
82         LOADIN BAF PC -1;
83     ],
84 Block
85     Name 'if_else_after.1',
86     [
87         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
88         # Exp(Global(Num('0')))
89         SUBI SP 1;
90         LOADIN DS ACC 0;
91         STOREIN SP ACC 1;
92         # Exp(Global(Num('1')))
93         SUBI SP 1;
94         LOADIN DS ACC 1;
95         STOREIN SP ACC 1;
96         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
97         LOADIN SP ACC 2;
98         LOADIN SP IN2 1;
99         MULT ACC IN2;
100        STOREIN SP ACC 2;
101        ADDI SP 1;
102        # Assign(Global(Num('1')), Stack(Num('1')))

```

```

103     LOADIN SP ACC 1;
104     STOREIN DS ACC 1;
105     ADDI SP 1;
106     # // Assign(Name('n'), BinOp(Name('n'), Sub('-', Num('1'))))
107     # Exp(Global(Num('0'))))
108     SUBI SP 1;
109     LOADIN DS ACC 0;
110     STOREIN SP ACC 1;
111     # Exp(Num('1'))
112     SUBI SP 1;
113     LOADI ACC 1;
114     STOREIN SP ACC 1;
115     # Exp(BinOp(Stack(Num('2')), Sub('-', Stack(Num('1')))))
116     LOADIN SP ACC 2;
117     LOADIN SP IN2 1;
118     SUB ACC IN2;
119     STOREIN SP ACC 2;
120     ADDI SP 1;
121     # Assign(Global(Num('0')), Stack(Num('1'))))
122     LOADIN SP ACC 1;
123     STOREIN DS ACC 0;
124     ADDI SP 1;
125     # Exp(GoTo(Name('condition_check.4'))))
126     Exp(GoTo(Name('condition_check.4'))))
127 ],
128 Block
129   Name 'while_after.0',
130   [
131     # Return(Empty())
132     LOADIN BAF PC -1;
133   ]
134 ]

```

Code 3.11: RETI-Patch Pass für Codebeispiel

3.3.2.6 RETI Pass

3.3.2.6.1 Konkrete und Abstrakte Syntax

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

Grammar 3.3.5: Konkrete Syntax für L_{RETI_Lex}

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

Grammar 3.3.6: Konkrete Syntax für L_{RETI_Parse}

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

Grammar 3.3.7: Abstrakte Syntax für L_{RETI}

3.3.2.6.2 Codebeispiel

```

1 # // Exp(GoTo(Name('main.5')))
2 # // patched out Exp(GoTo(Name('main.5')))
3 # // Assign(Name('n'), Num('4'))
4 # Exp(Num('4'))
5 SUBI SP 1;
6 LOADI ACC 4;
7 STOREIN SP ACC 1;
8 # Assign(Global(Num('0')), Stack(Num('1')))
9 LOADIN SP ACC 1;
10 STOREIN DS ACC 0;
11 ADDI SP 1;
12 # // Assign(Name('res'), Num('1'))
13 # Exp(Num('1'))
14 SUBI SP 1;
15 LOADI ACC 1;

```

```

16 STOREIN SP ACC 1;
17 # Assign(Global(Num('1')), Stack(Num('1')))
18 LOADIN SP ACC 1;
19 STOREIN DS ACC 1;
20 ADDI SP 1;
21 # // While(Num('1'), [])
22 # Exp(GoTo(Name('condition_check.4'))))
23 # // patched out Exp(GoTo(Name('condition_check.4'))))
24 # // IfElse(Num('1'), [], [])
25 # Exp(Num('1'))
26 SUBI SP 1;
27 LOADI ACC 1;
28 STOREIN SP ACC 1;
29 # IfElse(Stack(Num('1')), [], [])
30 LOADIN SP ACC 1;
31 ADDI SP 1;
32 JUMP== 49;
33 # // patched out Exp(GoTo(Name('while_branch.3'))))
34 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36 # Exp(Global(Num('0'))))
37 SUBI SP 1;
38 LOADIN DS ACC 0;
39 STOREIN SP ACC 1;
40 # Exp(Num('1'))
41 SUBI SP 1;
42 LOADI ACC 1;
43 STOREIN SP ACC 1;
44 # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
45 LOADIN SP ACC 2;
46 LOADIN SP IN2 1;
47 SUB ACC IN2;
48 JUMP== 3;
49 LOADI ACC 0;
50 JUMP 2;
51 LOADI ACC 1;
52 STOREIN SP ACC 2;
53 ADDI SP 1;
54 # IfElse(Stack(Num('1')), [], [])
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 JUMP== 2;
58 # // patched out Exp(GoTo(Name('if.2'))))
59 # Return(Empty())
60 LOADIN BAF PC -1;
61 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
62 # Exp(Global(Num('0'))))
63 SUBI SP 1;
64 LOADIN DS ACC 0;
65 STOREIN SP ACC 1;
66 # Exp(Global(Num('1'))))
67 SUBI SP 1;
68 LOADIN DS ACC 1;
69 STOREIN SP ACC 1;
70 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
71 LOADIN SP ACC 2;
72 LOADIN SP IN2 1;

```

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

Code 3.12: RETI Pass für Codebespiel

3.3.3 Umsetzung von Pointern

3.3.3.1 Referenzierung

Die **Referenzierung** `&<var>` wird im Folgenden anhand des Beispiels in Code 3.13 erklärt.

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

Code 3.13: PicoC-Code für Pointer Referenzierung

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakt Syntax Tree** in Code 3.14 eine **Referenzierung** `&<var>`.

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

Code 3.14: Abstract Syntax Tree für Pointer Referenzierung

Im **PicoC-Mon Pass** in Code 3.15 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')), Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp)` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var')))` ist, sind noch weitere Anweisungen zwischen den Zeilen 11 und 12 nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 3.3.6.2 kümmern.

```
1 File
2   Name './example_pntr_ref.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('pntr'), Ref(Name('var')))
11        Ref(Global(Num('0')))
```

```

12     Assign(Global(Num('1')), Stack(Num('1')))
13     Return(Empty())
14 ]
15 ]

```

Code 3.15: PicoC-Mon Pass für Pointer Referenzierung

Im **RETI-Blocks Pass** in Code 3.16 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_pntr_ref.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('pntr'), Ref(Name('var')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29    ]

```

Code 3.16: RETI-Blocks Pass für Pointer Referenzierung

3.3.3.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

Die **Dereferenzierung** `*<var>` wird im Folgenden anhand des Beispiels in Code 3.17 erklärt.

```

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

```

Code 3.17: PicoC-Code für Pointer Dereferenzierung

Der Knoten `Deref(Name('var'))` repräsentiert im **Abstrakt Syntax Tree** in Code 3.18 eine **Dereferenzierung** `*(<var>)`.

```

1 File
2   Name './example_pntr_deref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
11              ↪ Ref(Name('var')))
12        Exp(Deref(Name('pntr'), Num('0')))
13      ]
14    ]

```

Code 3.18: Abstract Syntax Tree für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 3.19 wird ein Trick angewendet, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** `*(<var> + <i>)` sich identisch zum **Operator für den Zugriff auf einen Arrayindex** `<var>[<i>]` verhält². Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Dereferenzierung** `*(<var> + <i>)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `<var>[<i>]` übernehmen lassen.

```

1 File
2   Name './example_pntr_deref.picoc_shrink',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
11              ↪ Ref(Name('var')))
12        Exp(Subscr(Name('pntr'), Num('0')))
13      ]
14    ]

```

Code 3.19: PicoC-Shrink Pass für Pointer Dereferenzierung

²In der Sprache L_C gibt es einen Unterschied bei der Initialisierung bei z.B. `<datatype> *(<var>) = "string"` und `<datatype> <var>[<i>] = "string"`, der allerdings nichts mit den beiden Operatoren zu tun hat, sondern mit der **Initialisierung**, bei der die Sprache L_C verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Arrayindex**, vor den Bezeichner schreibt: `<var>[<i>]`, obwohl es ein **Derived Datatype** ist.

3.3.4 Umsetzung von Arrays

3.3.4.1 Initialisierung von Arrays

Die **Initialisierung** eines **Arrays** (`<datatype> <var>[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 3.20 erklärt.

```

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

```

Code 3.20: PicoC-Code für Array Initialisierung

Die **Initialisierung** eines **Arrays** `<datatype> <var>[2][1] = {{3+1}, {4}}` wird im **Abstrakt Syntax Tree** in Code 3.21 mithilfe der Komposition `Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')), Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])]))` dargestellt.

```

1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
10          ↪ Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
11          ↪ Array([Num('4')])]))
12       ],
13     FunDef
14       VoidType 'void',
15       Name 'fun',
16       [],
17       [
18         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
19          ↪ Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
20       ]
21   ]

```

Code 3.21: Abstract Syntax Tree für Array Initialisierung

Bei der **Initialisierung** eines **Arrays** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann³. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 3.22 dargestellt ist.

³Das widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** tritt allerdings erst später in Aktion.

Bei Variablen auf dem **Stackframe** wird ein Array **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Arrays genommen. Dies macht den **Zugriff auf einen Arrayindex** in Subkapitel 3.3.4.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf einen Arrayindex** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die Entgegengesetzt Richtung der **Globalen Statischen Daten** wächst⁴.

```

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

```

Code 3.22: Symboltabelle für Array Initialisierung

Im **PiocC-Mon Pass** in Code 3.23 werden zuerst die **Ausdrücke** im **Array-Initializer** `Array([Array([BinOp(Num('3'), Add('++'), Num('1'))]), Array([Num('4')])])` nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben.

⁴Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

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

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

```

1 File
2   Name './example_array_init.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Assign(Name('ar'), Array([Array([BinOp(Num('3')), Add('+'), Num('1'))]),
8         ↪   Array([Num('4')]))])
9         Exp(Num('3'))
10        Exp(Num('1'))
11        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
12        Exp(Num('4'))
13        Assign(Global(Num('0')), Stack(Num('2')))
14        Return(Empty())
15      ],
16    Block
17      Name 'fun.0',
18      [
19        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
20        ↪   Num('6')]))])
21        Exp(Num('3'))
22        Exp(Num('4'))
23        Exp(Num('5'))
24        Exp(Num('6'))
25        Assign(Stackframe(Num('3')), Stack(Num('4')))
26        Return(Empty())
27      ]
28    ]
29  ]

```

Code 3.23: PicoC-Mon Pass für Array Initialisierung

Im **RETI-Blocks Pass** in Code 3.24 werden die **PicoC-Knoten** bzw. **Kompositionen** für die Ausdrücke `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_array_init.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Array([BinOp(Num('3')), Add('+'), Num('1'))]),
8         ↪   Array([Num('4')]))])

```

```

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

```

```

64     ADDI SP 4;
65     # Return(Empty())
66     LOADIN BAF PC -1;
67 ]
68 ]

```

Code 3.24: RETI-Blocks Pass für Array Initialisierung

3.3.4.2 Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im Folgenden anhand des Beispiels in Code 3.25 erklärt.

```

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

```

Code 3.25: PicoC-Code für Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im **Abstract Syntx Tree** in Code 3.26 mithilfe des **Container-Knotens** `Subscr(Name('ar'), Num('0'))` dargestellt.

```

1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
10              ↪ Array([Num('42')]))
11         Exp(Subscr(Name('ar'), Num('0')))
12       ],
13     FunDef
14       VoidType 'void',
15       Name 'fun',
16       [],
17       [
18         Assign(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
19              ↪ Array([Num('1'), Num('2'), Num('3')]))
20         Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
21       ]
22   ]

```


Code 3.26: Abstract Syntax Tree für Zugriff auf einen Arrayindex

Im **PicoC-Mon Pass** in Code 3.27 wird beim **Container-Knoten** `Subscr(Name('ar'), Num('0'))` zuerst im **Einleitungsteil** die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktion `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil** wird die Adresse des **Index**, des Arrays auf das Zugriffen werden soll berechnet. Da der **Index** auf den Zugriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `<ar>[1 + <var>]` bestimmt sein kann, indem auch **Variablen** vorkommen können, kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle 3.7 dokumentiert.

Je nachdem, ob mehrere `Subscr(exp, exp)` eine Komposition bilden (z.B. `Subscr(Subscr(Name('<var>'), Num('<i>')), Num('<i>'))`), ist es notwendig mehrere **Adressberechnungsschritte für den Index** `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` einzuleiten und es muss auch möglich sein, z.B. einen **Attributzugriff** `<var>.<attr>` und eine **Zugriff auf einen Arrayindex** `<var>[<i>]` miteinander zu kombinieren, was in Subkapitel 3.3.6.2 allgemein erklärt ist.

```

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

```

```

31     Exp(Stack(Num('1')))
32     Return(Empty())
33 ]
34 ]

```

Code 3.27: PicoC-Mon Pass für Zugriff auf einen Arrayindex

Im **RETI-Blocks Pass** in Code 3.24 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_array_access.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Exp(Subscr(Name('ar'), Num('0')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Exp(Num('0'))
23        SUBI SP 1;
24        LOADI ACC 0;
25        STOREIN SP ACC 1;
26        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        LOADIN SP IN1 2;
28        LOADIN SP IN2 1;
29        MULTI IN2 1;
30        ADD IN1 IN2;
31        ADDI SP 1;
32        STOREIN SP IN1 1;
33        # Exp(Stack(Num('1')))
34        LOADIN SP IN1 1;
35        LOADIN IN1 ACC 0;
36        STOREIN SP ACC 1;
37        # Return(Empty())
38        LOADIN BAF PC -1;
39      ],
40     Block
41       Name 'fun.0',
42       [
43         # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44         # Exp(Num('1'))

```

```

45     SUBI SP 1;
46     LOADI ACC 1;
47     STOREIN SP ACC 1;
48     # Exp(Num('2'))
49     SUBI SP 1;
50     LOADI ACC 2;
51     STOREIN SP ACC 1;
52     # Exp(Num('3'))
53     SUBI SP 1;
54     LOADI ACC 3;
55     STOREIN SP ACC 1;
56     # Assign(Stackframe(Num('2')), Stack(Num('3')))
57     LOADIN SP ACC 1;
58     STOREIN BAF ACC -2;
59     LOADIN SP ACC 2;
60     STOREIN BAF ACC -3;
61     LOADIN SP ACC 3;
62     STOREIN BAF ACC -4;
63     ADDI SP 3;
64     # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65     # Ref(Stackframe(Num('2')))
66     SUBI SP 1;
67     MOVE BAF IN1;
68     SUBI IN1 4;
69     STOREIN SP IN1 1;
70     # Exp(Num('1'))
71     SUBI SP 1;
72     LOADI ACC 1;
73     STOREIN SP ACC 1;
74     # Exp(Num('1'))
75     SUBI SP 1;
76     LOADI ACC 1;
77     STOREIN SP ACC 1;
78     # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79     LOADIN SP ACC 2;
80     LOADIN SP IN2 1;
81     ADD ACC IN2;
82     STOREIN SP ACC 2;
83     ADDI SP 1;
84     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85     LOADIN SP IN1 2;
86     LOADIN SP IN2 1;
87     MULTI IN2 1;
88     ADD IN1 IN2;
89     ADDI SP 1;
90     STOREIN SP IN1 1;
91     # Exp(Stack(Num('1'))))
92     LOADIN SP IN1 1;
93     LOADIN IN1 ACC 0;
94     STOREIN SP ACC 1;
95     # Return(Empty())
96     LOADIN BAF PC -1;
97 ]
98 ]

```

Code 3.28: RETI-Blocks Pass für Zugriff auf einen Arrayindex

3.3.4.3 Zuweisung an Arrayindex

```

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

```

Code 3.29: 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(Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
10        Assign(Subscr(Name('ar'), Num('2')), Num('42'))
11      ]
12 ]

```

Code 3.30: 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(Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
8         // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
9         Exp(Num('42'))
10        Ref(Global(Num('0')))
11        Exp(Num('2'))
12        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
13        Assign(Stack(Num('1')), Stack(Num('2')))
14        Return(Empty())
15      ]
16 ]

```

Code 3.31: 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          # // Exp(Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
8          # // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
9          # Exp(Num('42'))
10         SUBI SP 1;
11         LOADI ACC 42;
12         STOREIN SP ACC 1;
13         # Ref(Global(Num('0')))
14         SUBI SP 1;
15         LOADI IN1 0;
16         ADD IN1 DS;
17         STOREIN SP IN1 1;
18         # Exp(Num('2'))
19         SUBI SP 1;
20         LOADI ACC 2;
21         STOREIN SP ACC 1;
22         # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23         LOADIN SP IN1 2;
24         LOADIN SP IN2 1;
25         MULTI IN2 1;
26         ADD IN1 IN2;
27         ADDI SP 1;
28         STOREIN SP IN1 1;
29         LOADIN SP IN1 1;
30         LOADIN SP ACC 2;
31         ADDI SP 2;
32         STOREIN IN1 ACC 0;
33         # Return(Empty())
34         LOADIN BAF PC -1;
35     ]
36 ]

```

Code 3.32: RETI-Blocks Pass für Zuweisung an Arrayindex

3.3.5 Umsetzung von Structs

3.3.5.1 Deklaration von Structs

```

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

```

Code 3.33: 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
42     {
43         type qualifier:      Empty()
44         datatype:           FunDecl(VoidType('void'), Name('main'), [])
45         name:               Name('main')
46         value or address:    Empty()
47         position:           Pos(Num('5'), Num('5'))
48         size:               Empty()
49     }
50 ]

```

Code 3.34: Symboltabelle für Deklaration von Structs

3.3.5.2 Initialisierung von Structs

```

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

```

Code 3.35: 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(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
33          Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')),
34            ↳ Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
35            ↳ Array([Array([Ref(Name('var'))]))]))]))))
36        ]
37    ]
38  ]

```

Code 3.36: Abstract Syntax Tree für Initialisierung von Structs

```

1 SymbolTable
2 [
3   Symbol
4   {
5     type qualifier:      Empty()
6     datatype:            ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int')))
7     name:                Name('ptr@st1')
8     value or address:    Empty()
9     position:            Pos(Num('1'), Num('17'))
10    size:                 Num('1')
11  },
12  Symbol
13  {
14    type qualifier:      Empty()
15    datatype:            StructDecl(Name('st1'), [Alloc(Writeable(),
16      ↪ ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int'))), Name('ptr')))]
17    name:                Name('st1')
18    value or address:    [Name('ptr@st1')]
19    position:            Pos(Num('1'), Num('7'))
20    size:                 Num('1')
21  },
22  Symbol
23  {
24    type qualifier:      Empty()
25    datatype:            StructSpec(Name('st1'))
26    name:                Name('st@st2')
27    value or address:    Empty()
28    position:            Pos(Num('3'), Num('23'))
29    size:                 Num('1')
30  },
31  Symbol
32  {
33    type qualifier:      Empty()
34    datatype:            StructDecl(Name('st2'), [Alloc(Writeable(),
35      ↪ StructSpec(Name('st1')), Name('st')))]
36    name:                Name('st2')
37    value or address:    [Name('st@st2')]
38    position:            Pos(Num('3'), Num('7'))
39    size:                 Num('1')
40  },
41  Symbol
42  {
43    type qualifier:      Empty()
44    datatype:            FunDecl(VoidType('void'), Name('main'), [])
45    name:                Name('main')
46    value or address:    Empty()
47    position:            Pos(Num('5'), Num('5'))
48    size:                 Empty()
49  },
50  Symbol
51  {
52    type qualifier:      Writeable()
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')

```



```

56     },
57     Symbol
58     {
59         type qualifier:      Writeable()
60         datatype:           StructSpec(Name('st1'))
61         name:               Name('st@main')
62         value or address:    Num('1')
63         position:           Pos(Num('7'), Num('13'))
64         size:               Num('1')
65     }
66 ]

```

Code 3.37: Symboltabelle für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.picoc_mon',
3   [
4     Block
5     Name 'main.0',
6     [
7       // Assign(Name('var'), Num('42'))
8       Exp(Num('42'))
9       Assign(Global(Num('0')), Stack(Num('1')))
10      // Assign(Name('st'), Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
11      ↪   Array([Array([Ref(Name('var'))]))]))]))
12      Ref(Global(Num('0')))
13      Assign(Global(Num('1')), Stack(Num('1')))
14      Return(Empty())
15    ]
16  ]

```

Code 3.38: 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       # // Assign(Name('var'), Num('42'))
8       # Exp(Num('42'))
9       SUBI SP 1;
10      LOADI ACC 42;
11      STOREIN SP ACC 1;
12      # Assign(Global(Num('0')), Stack(Num('1')))
13      LOADIN SP ACC 1;
14      STOREIN DS ACC 0;
15      ADDI SP 1;
16      # // Assign(Name('st'), Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
17      ↪   Array([Array([Ref(Name('var'))]))]))]))
18      # Ref(Global(Num('0')))

```

```

18     SUBI SP 1;
19     LOADI IN1 0;
20     ADD IN1 DS;
21     STOREIN SP IN1 1;
22     # Assign(Global(Num('1')), Stack(Num('1')))
23     LOADIN SP ACC 1;
24     STOREIN DS ACC 1;
25     ADDI SP 1;
26     # Return(Empty())
27     LOADIN BAF PC -1;
28 ]
29 ]

```

Code 3.39: RETI-Blocks Pass für Initialisierung von Structs

3.3.5.3 Zugriff auf Structattribut

```

1 struct pos {int x; int y;};
2
3 void main() {
4     struct pos st = {.x=4, .y=2};
5     st.y;
6 }

```

Code 3.40: 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(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
22               ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
23         Exp(Attr(Name('st'), Name('y')))
24       ]
25     ]
26   ]

```

Code 3.41: 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         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8           ↪ Num('2'))]))
9         Exp(Num('4'))
10        Exp(Num('2'))
11        Assign(Global(Num('0')), Stack(Num('2')))
12        // Exp(Attr(Name('st'), Name('y')))
13        Ref(Global(Num('0')))
14        Ref(Attr(Stack(Num('1')), Name('y')))
15        Exp(Stack(Num('1')))
16        Return(Empty())
17      ]
18    ]

```

Code 3.42: 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         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8           ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # Assign(Global(Num('0')), Stack(Num('2')))
18        LOADIN SP ACC 1;
19        STOREIN DS ACC 1;
20        LOADIN SP ACC 2;
21        STOREIN DS ACC 0;
22        ADDI SP 2;
23        # // Exp(Attr(Name('st'), Name('y')))
24        # Ref(Global(Num('0')))
25        SUBI SP 1;
26        LOADI IN1 0;
27        ADD IN1 DS;
28        STOREIN SP IN1 1;

```

```

28     # Ref(Attr(Stack(Num('1')), Name('y')))
29     LOADIN SP IN1 1;
30     ADDI IN1 1;
31     STOREIN SP IN1 1;
32     # Exp(Stack(Num('1')))
33     LOADIN SP IN1 1;
34     LOADIN IN1 ACC 0;
35     STOREIN SP ACC 1;
36     # Return(Empty())
37     LOADIN BAF PC -1;
38 ]
39 ]

```

Code 3.43: RETI-Blocks Pass für Zugriff auf Structattribut

3.3.5.4 Zuweisung an Structattribut

```

1 struct pos {int x; int y;};
2
3 void main() {
4     struct pos st = {.x=4, .y=2};
5     st.y = 42;
6 }

```

Code 3.44: 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(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
22           ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
23         Assign(Attr(Name('st'), Name('y')), Num('42'))
24       ]
25     ]
26   ]

```

Code 3.45: 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         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8           ↪ Num('2'))]))
9         Exp(Num('4'))
10        Exp(Num('2'))
11        Assign(Global(Num('0')), Stack(Num('2')))
12        // Assign(Attr(Name('st'), Name('y')), Num('42'))
13        Exp(Num('42'))
14        Ref(Global(Num('0')))
15        Ref(Attr(Stack(Num('1')), Name('y')))
16        Assign(Stack(Num('1')), Stack(Num('2')))
17        Return(Empty())
18      ]
19    ]

```

Code 3.46: 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         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8           ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # Assign(Global(Num('0')), Stack(Num('2')))
18        LOADIN SP ACC 1;
19        STOREIN DS ACC 1;
20        LOADIN SP ACC 2;
21        STOREIN DS ACC 0;
22        ADDI SP 2;
23        # // Assign(Attr(Name('st'), Name('y')), Num('42'))
24        # Exp(Num('42'))
25        SUBI SP 1;
26        LOADI ACC 42;
27        STOREIN SP ACC 1;

```

```

27     # Ref(Global(Num('0')))
28     SUBI SP 1;
29     LOADI IN1 0;
30     ADD IN1 DS;
31     STOREIN SP IN1 1;
32     # Ref(Attr(Stack(Num('1')), Name('y')))
33     LOADIN SP IN1 1;
34     ADDI IN1 1;
35     STOREIN SP IN1 1;
36     LOADIN SP IN1 1;
37     LOADIN SP ACC 2;
38     ADDI SP 2;
39     STOREIN IN1 ACC 0;
40     # Return(Empty())
41     LOADIN BAF PC -1;
42 ]
43 ]

```

Code 3.47: RETI-Blocks Pass für Zuweisung an Structattribut

3.3.6 Umsetzung der Derived Datatypes im Zusammenspiel

3.3.6.1 Einleitungsteil für Globale Statische Daten und Stackframe

```

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

```

Code 3.48: 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(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
26         ↪ Name('st_ar')))
27     Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
28         ↪ IntType('int')))), Name('ptr2')))
29     Exp(Name('ptr2'))
30 ],
31 FunDef
32 VoidType 'void',
33 Name 'fun',
34 [],
35 [
36     Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
37         ↪ Name('st_ar')))
38     Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
39         ↪ Name('ptr1')))
40     Exp(Name('ptr1'))
41 ]
42 ]

```

Code 3.49: 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(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
8         ↪ Name('st_ar')))
9         // Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
10         ↪ IntType('int')))), Name('ptr2')))
11         // Exp(Name('ptr2'))
12         Exp(Global(Num('9')))
13         Return(Empty())
14     ],
15     Block
16     Name 'fun.0',
17     [
18         // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
19         ↪ Name('st_ar')))

```

```

17      // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
18      ↪ Name('ptr1'))
19      // Exp(Name('ptr1'))
20      Exp(Stackframe(Num('9')))
21      Return(Empty())
22  ]

```

Code 3.50: 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         # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
8         ↪ Name('st_ar'))
9         # // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')],
10        ↪ PtrDecl(Num('1'), IntType('int'))), Name('ptr2'))
11        # // Exp(Name('ptr2'))
12        # Exp(Global(Num('9')))
13        SUBI SP 1;
14        LOADIN DS ACC 9;
15        STOREIN SP ACC 1;
16        # Return(Empty())
17        LOADIN BAF PC -1;
18      ],
19    Block
20      Name 'fun.0',
21      [
22        # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
23        ↪ Name('st_ar'))
24        # // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')],
25        ↪ IntType('int'))), Name('ptr1'))
26        # // Exp(Name('ptr1'))
27        # Exp(Stackframe(Num('9')))
28        SUBI SP 1;
29        LOADIN BAF ACC -11;
30        STOREIN SP ACC 1;
31        # Return(Empty())
32        LOADIN BAF PC -1;
33      ]
34    ]
35  ]

```

Code 3.51: RETI-Blocks Pass für den Einleitungsteil

3.3.6.2 Mittelteil für die verschiedenen Derived Datatypes

```

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

```

Code 3.52: 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           PntrDecl
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(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
24            ↪ Array([Num('42')]))
25          Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st_first')),
26            ↪ Struct([Assign(Name('ar'), Ref(Name('var')))]))
27          Exp(Subscr(Deref(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
28        ]
29      ]
30   ]

```

Code 3.53: 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         // Assign(Name('var'), Array([Num('42')]))

```

```

8      Exp(Num('42'))
9      Assign(Global(Num('0')), Stack(Num('1')))
10     // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11     Ref(Global(Num('0')))
12     Assign(Global(Num('1')), Stack(Num('1')))
13     // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
14     Ref(Global(Num('1')))
15     Ref(Attr(Stack(Num('1')), Name('ar')))
16     Exp(Num('0'))
17     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18     Exp(Num('0'))
19     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20     Exp(Stack(Num('1')))
21     Return(Empty())
22 ]
23 ]

```

Code 3.54: 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         # // Assign(Name('var'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
27        # Ref(Global(Num('1')))
28        SUBI SP 1;
29        LOADI IN1 1;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Ref(Attr(Stack(Num('1')), Name('ar')))
33        LOADIN SP IN1 1;
34        ADDI IN1 0;
35        STOREIN SP IN1 1;

```

```

36      # Exp(Num('0'))
37      SUBI SP 1;
38      LOADI ACC 0;
39      STOREIN SP ACC 1;
40      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41      LOADIN SP IN2 2;
42      LOADIN IN2 IN1 0;
43      LOADIN SP IN2 1;
44      MULTI IN2 1;
45      ADD IN1 IN2;
46      ADDI SP 1;
47      STOREIN SP IN1 1;
48      # Exp(Num('0'))
49      SUBI SP 1;
50      LOADI ACC 0;
51      STOREIN SP ACC 1;
52      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
53      LOADIN SP IN1 2;
54      LOADIN SP IN2 1;
55      MULTI IN2 1;
56      ADD IN1 IN2;
57      ADDI SP 1;
58      STOREIN SP IN1 1;
59      # Exp(Stack(Num('1')))
60      LOADIN SP IN1 1;
61      LOADIN IN1 ACC 0;
62      STOREIN SP ACC 1;
63      # Return(Empty())
64      LOADIN BAF PC -1;
65  ]
66 ]

```

Code 3.55: RETI-Blocks Pass für den Mittelteil

3.3.6.3 Schlussteil für die verschiedenen Derived Datatypes

```

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

```

Code 3.56: 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(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
22            ↪ Name('ar1')), Array([Array([Num('42'), Num('314')])]))
23          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
24            ↪ Name('ar2')), Struct([Assign(Name('attr'), Array([Num('42'), Num('314')])]))))
25          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
26          Assign(Alloc(Writeable(), PtrDecl(Num('1'), IntType('int')), Name('pntr1')),
27            ↪ Ref(Name('var')))
28          Assign(Alloc(Writeable(), PtrDecl(Num('2'), IntType('int')), Name('pntr2')),
29            ↪ Ref(Name('pntr1')))
30          Exp(Subscr(Name('ar1'), Num('0')))
31          Exp(Subscr(Name('ar2'), Num('0')))
32          Exp(Deref(Name('pntr2'), Num('0')))
33        ]
34      ]
35    ]

```

Code 3.57: Abstract Syntax Tree für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.O',
6       [
7         // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')])]))
8         Exp(Num('42'))
9         Exp(Num('314'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
12          ↪ Num('314')])]))))
13        Exp(Num('42'))
14        Exp(Num('314'))
15        Assign(Global(Num('2')), Stack(Num('2')))
16        // Assign(Name('var'), Num('42'))
17        Exp(Num('42'))

```

```

17     Assign(Global(Num('4')), Stack(Num('1')))
18     // Assign(Name('pntr1'), Ref(Name('var')))
19     Ref(Global(Num('4')))
20     Assign(Global(Num('5')), Stack(Num('1')))
21     // Assign(Name('pntr2'), Ref(Name('pntr1')))
22     Ref(Global(Num('5')))
23     Assign(Global(Num('6')), Stack(Num('1')))
24     // Exp(Subscr(Name('ar1'), Num('0')))
25     Ref(Global(Num('0')))
26     Exp(Num('0'))
27     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
28     Exp(Stack(Num('1')))
29     // Exp(Subscr(Name('ar2'), Num('0')))
30     Ref(Global(Num('2')))
31     Exp(Num('0'))
32     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
33     Exp(Stack(Num('1')))
34     // Exp(Subscr(Name('pntr2'), Num('0')))
35     Ref(Global(Num('6')))
36     Exp(Num('0'))
37     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
38     Exp(Stack(Num('1')))
39     Return(Empty())
40 ]
41 ]

```

Code 3.58: PicoC-Mon Pass für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')]))))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Exp(Num('314'))
13        SUBI SP 1;
14        LOADI ACC 314;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
23        ↪ Num('314')]))]))
24        # Exp(Num('42'))
25        SUBI SP 1;
26        LOADI ACC 42;

```

```
26     STOREIN SP ACC 1;
27     # Exp(Num('314'))
28     SUBI SP 1;
29     LOADI ACC 314;
30     STOREIN SP ACC 1;
31     # Assign(Global(Num('2')), Stack(Num('2')))
32     LOADIN SP ACC 1;
33     STOREIN DS ACC 3;
34     LOADIN SP ACC 2;
35     STOREIN DS ACC 2;
36     ADDI SP 2;
37     # // Assign(Name('var'), Num('42'))
38     # Exp(Num('42'))
39     SUBI SP 1;
40     LOADI ACC 42;
41     STOREIN SP ACC 1;
42     # Assign(Global(Num('4')), Stack(Num('1')))
43     LOADIN SP ACC 1;
44     STOREIN DS ACC 4;
45     ADDI SP 1;
46     # // Assign(Name('pntr1'), Ref(Name('var')))
47     # Ref(Global(Num('4')))
48     SUBI SP 1;
49     LOADI IN1 4;
50     ADD IN1 DS;
51     STOREIN SP IN1 1;
52     # Assign(Global(Num('5')), Stack(Num('1')))
53     LOADIN SP ACC 1;
54     STOREIN DS ACC 5;
55     ADDI SP 1;
56     # // Assign(Name('pntr2'), Ref(Name('pntr1')))
57     # Ref(Global(Num('5')))
58     SUBI SP 1;
59     LOADI IN1 5;
60     ADD IN1 DS;
61     STOREIN SP IN1 1;
62     # Assign(Global(Num('6')), Stack(Num('1')))
63     LOADIN SP ACC 1;
64     STOREIN DS ACC 6;
65     ADDI SP 1;
66     # // Exp(Subscr(Name('ar1'), Num('0')))
67     # Ref(Global(Num('0')))
68     SUBI SP 1;
69     LOADI IN1 0;
70     ADD IN1 DS;
71     STOREIN SP IN1 1;
72     # Exp(Num('0'))
73     SUBI SP 1;
74     LOADI ACC 0;
75     STOREIN SP ACC 1;
76     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
77     LOADIN SP IN1 2;
78     LOADIN SP IN2 1;
79     MULTI IN2 2;
80     ADD IN1 IN2;
81     ADDI SP 1;
82     STOREIN SP IN1 1;
```

```

83      # Exp(Stack(Num('1')))
84      # // Exp(Subscr(Name('ar2'), Num('0')))
85      # Ref(Global(Num('2')))
86      SUBI SP 1;
87      LOADI IN1 2;
88      ADD IN1 DS;
89      STOREIN SP IN1 1;
90      # Exp(Num('0'))
91      SUBI SP 1;
92      LOADI ACC 0;
93      STOREIN SP ACC 1;
94      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
95      LOADIN SP IN1 2;
96      LOADIN SP IN2 1;
97      MULTI IN2 2;
98      ADD IN1 IN2;
99      ADDI SP 1;
100     STOREIN SP IN1 1;
101     # Exp(Stack(Num('1')))
102     LOADIN SP IN1 1;
103     LOADIN IN1 ACC 0;
104     STOREIN SP ACC 1;
105     # // Exp(Subscr(Name('pntr2'), Num('0')))
106     # Ref(Global(Num('6')))
107     SUBI SP 1;
108     LOADI IN1 6;
109     ADD IN1 DS;
110     STOREIN SP IN1 1;
111     # Exp(Num('0'))
112     SUBI SP 1;
113     LOADI ACC 0;
114     STOREIN SP ACC 1;
115     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
116     LOADIN SP IN2 2;
117     LOADIN IN2 IN1 0;
118     LOADIN SP IN2 1;
119     MULTI IN2 1;
120     ADD IN1 IN2;
121     ADDI SP 1;
122     STOREIN SP IN1 1;
123     # Exp(Stack(Num('1')))
124     # Return(Empty())
125     LOADIN BAF PC -1;
126 ]
127 ]

```

Code 3.59: RETI-Blocks Pass für den Schlussteil

3.3.7 Umsetzung von Funktionen

3.3.7.1 Funktionen auflösen zu RETI Code

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

Code 3.60: 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(Empty())  
10      ],  
11     FunDef  
12       VoidType 'void',  
13       Name 'fun1',  
14       [],  
15       [],  
16     FunDef  
17       IntType 'int',  
18       Name 'fun2',  
19       [],  
20       [  
21         Return(Num('1'))  
22      ]  
23   ]
```

Code 3.61: 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(Empty())
13         ]
14     ],
15     FunDef
16     VoidType 'void',
17     Name 'fun1',
18     [],
19     [
20         Block
21             Name 'fun1.1',
22             []
23     ],
24     FunDef
25     IntType 'int',
26     Name 'fun2',
27     [],
28     [
29         Block
30             Name 'fun2.0',
31             [
32                 Return(Num('1'))
33             ]
34     ]
35 ]

```

Code 3.62: RETI-Blocks Pass für 3 Funktionen

```

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

```

Code 3.63: PicoC-Mon Pass für 3 Funktionen

```

1 File
2   Name './example_3_funs.reti_blocks',
3   [
4     Block
5       Name 'main.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun1.1',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun2.0',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ]
29  ]

```

Code 3.64: RETI-Blocks Pass für 3 Funktionen

3.3.7.1.1 Sprung zur Main Funktion

```

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

```

Code 3.65: 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(Empty())
8      ],
9    Block
10     Name 'fun2.1',
11     [
12       // Return(Num('1'))
13       Exp(Num('1'))
14       Return(Stack(Num('1')))
15     ],
16    Block
17     Name 'main.0',
18     [
19       Return(Empty())
20     ]
21 ]

```

Code 3.66: 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         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun2.1',
12      [
13        # // Return(Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Return(Stack(Num('1')))
19        LOADIN SP ACC 1;
20        ADDI SP 1;
21        LOADIN BAF PC -1;
22      ],
23    Block
24      Name 'main.0',
25      [
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29 ]

```

Code 3.67: RETI-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(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun2.1',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ],
29    Block
30      Name 'main.0',
31      [
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ]
35  ]

```

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

3.3.7.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

```

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

Bei mehreren Funktionen werden die **Scopes** der unterschiedlichen **Funktionen** mittels eines **Suffix** "<fun_name>@" umgesetzt, der an den **Variablen**namen <var> drangehängt wird: <var>@<fun_name>. Dieser **Suffix** wird geändert sobald beim **Top-Down**⁵ Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** nach dem **Depth-First-Search** Schema über den

```

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

```

⁵D.h. von der Wurzel zu den Blättern eines Baumes

```

46     size:          Num('1')
47   }
48 ]

```

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

3.3.7.3 Funktionsaufruf

3.3.7.3.1 Ohne Rückgabewert

```

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

```

Code 3.71: 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         // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
8         //   ↳ Name('local_var')))
9         // Exp(Call(Name('stack_fun'), [Name('local_var')]))
10        StackMalloc(Num('2'))
11        Ref(Global(Num('0')))
12        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
13        Exp(GoTo(Name('stack_fun.0')))
14        RemoveStackframe()
15        Return(Empty())
16      ],
17    Block
18      Name 'stack_fun.0',
19      [
20        // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
21        //   ↳ Name('param')))
22        // Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
23        Return(Empty())
24      ]
25    ]
26  ]

```

Code 3.72: 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         # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
8         ↪   Name('local_var'))
9         # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
10        # StackMalloc(Num('2'))
11        SUBI SP 2;
12        # Ref(Global(Num('0')))
13        SUBI SP 1;
14        LOADI IN1 0;
15        ADD IN1 DS;
16        STOREIN SP IN1 1;
17        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
18        MOVE BAF ACC;
19        ADDI SP 3;
20        MOVE SP BAF;
21        SUBI SP 4;
22        STOREIN BAF ACC 0;
23        LOADI ACC GoTo(Name('addr@next_instr'));
24        ADD ACC CS;
25        STOREIN BAF ACC -1;
26        # Exp(GoTo(Name('stack_fun.0')))
27        Exp(GoTo(Name('stack_fun.0')))
28        # RemoveStackframe()
29        MOVE BAF IN1;
30        LOADIN IN1 BAF 0;
31        MOVE IN1 SP;
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ],
35    Block
36      Name 'stack_fun.0',
37      [
38        # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
39        ↪   Name('param'))
40        # // Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
41        # Return(Empty())
42        LOADIN BAF PC -1;
43      ]
44    ]
45  ]

```

Code 3.73: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

```

1 # // Exp(GoTo(Name('main.1')))
2 # // patched out Exp(GoTo(Name('main.1')))

```

```

3 # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
  ↳ Name('local_var'))))
4 # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
5 # StackMalloc(Num('2'))
6 SUBI SP 2;
7 # Ref(Global(Num('0'))))
8 SUBI SP 1;
9 LOADI IN1 0;
10 ADD IN1 DS;
11 STOREIN SP IN1 1;
12 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
13 MOVE BAF ACC;
14 ADDI SP 3;
15 MOVE SP BAF;
16 SUBI SP 4;
17 STOREIN BAF ACC 0;
18 LOADI ACC 14;
19 ADD ACC CS;
20 STOREIN BAF ACC -1;
21 # Exp(GoTo(Name('stack_fun.0'))))
22 JUMP 5;
23 # RemoveStackframe()
24 MOVE BAF IN1;
25 LOADIN IN1 BAF 0;
26 MOVE IN1 SP;
27 # Return(Empty())
28 LOADIN BAF PC -1;
29 # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))), Name('param'))))
30 # // Exp(Alloc(Writable(), IntType('int'), Name('local_var'))))
31 # Return(Empty())
32 LOADIN BAF PC -1;

```

Code 3.74: RETI-Pass für Funktionsaufruf ohne Rückgabewert

3.3.7.3.2 Mit Rückgabewert

```

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

```

Code 3.75: 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          // Return(Num('42'))
8          Exp(Num('42'))
9          Return(Stack(Num('1')))
10     ],
11     Block
12     Name 'main.0',
13     [
14         // Assign(Name('var'), Call(Name('stack_fun'), []))
15         StackMalloc(Num('2'))
16         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
17         Exp(GoTo(Name('stack_fun.1'))))
18         RemoveStackframe()
19         Assign(Global(Num('0')), Stack(Num('1')))
20         Return(Empty())
21     ]
22 ]

```

Code 3.76: 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           # // Return(Num('42'))
8           # Exp(Num('42'))
9           SUBI SP 1;
10          LOADI ACC 42;
11          STOREIN SP ACC 1;
12          # Return(Stack(Num('1')))
13          LOADIN SP ACC 1;
14          ADDI SP 1;
15          LOADIN BAF PC -1;
16      ],
17      Block
18      Name 'main.0',
19      [
20          # // Assign(Name('var'), Call(Name('stack_fun'), []))
21          # StackMalloc(Num('2'))
22          SUBI SP 2;
23          # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
24          MOVE BAF ACC;
25          ADDI SP 2;
26          MOVE SP BAF;
27          SUBI SP 2;
28          STOREIN BAF ACC 0;
29          LOADI ACC GoTo(Name('addr@next_instr'));
30          ADD ACC CS;
31          STOREIN BAF ACC -1;
32          # Exp(GoTo(Name('stack_fun.1')))
33          Exp(GoTo(Name('stack_fun.1')))
34          # RemoveStackframe()

```

```

35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Assign(Global(Num('0')), Stack(Num('1')))
39     LOADIN SP ACC 1;
40     STOREIN DS ACC 0;
41     ADDI SP 1;
42     # Return(Empty())
43     LOADIN BAF PC -1;
44 ]
45 ]

```

Code 3.77: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1 # // Exp(GoTo(Name('main.0'))))
2 JUMP 7;
3 # // Return(Num('42'))
4 # Exp(Num('42'))
5 SUBI SP 1;
6 LOADI ACC 42;
7 STOREIN SP ACC 1;
8 # Return(Stack(Num('1')))
9 LOADIN SP ACC 1;
10 ADDI SP 1;
11 LOADIN BAF PC -1;
12 # // Assign(Name('var'), Call(Name('stack_fun'), []))
13 # StackMalloc(Num('2'))
14 SUBI SP 2;
15 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
16 MOVE BAF ACC;
17 ADDI SP 2;
18 MOVE SP BAF;
19 SUBI SP 2;
20 STOREIN BAF ACC 0;
21 LOADI ACC 17;
22 ADD ACC CS;
23 STOREIN BAF ACC -1;
24 # Exp(GoTo(Name('stack_fun.1'))))
25 JUMP -15;
26 # RemoveStackframe()
27 MOVE BAF IN1;
28 LOADIN IN1 BAF 0;
29 MOVE IN1 SP;
30 # Assign(Global(Num('0')), Stack(Num('1')))
31 LOADIN SP ACC 1;
32 STOREIN DS ACC 0;
33 ADDI SP 1;
34 # Return(Empty())
35 LOADIN BAF PC -1;

```

Code 3.78: RETI-Pass für Funktionsaufruf mit Rückgabewert

3.3.7.3.3 Umsetzung von Call by Sharing für Arrays

```

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

```

Code 3.79: 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         // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param1')))
9         // Exp(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
10        Return(Empty())
11      ],
12    Block
13      Name 'main.0',
14      [
15        // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
16        ↪ Name('local_var1')))
17        // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
18        ↪ Name('local_var2')))
19        // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
20        StackMalloc(Num('2'))
21        Ref(Global(Num('0')))
22        Ref(Global(Num('6')))
23        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
24        Exp(GoTo(Name('stack_fun.1')))
25        RemoveStackframe()
26        Return(Empty())
27      ]
28    ]
29  ]

```

Code 3.80: 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'),
   ↪ [Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
   ↪ Name('param1')), Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')),
   ↪ Name('param2'))])
7      name:          Name('stack_fun')
8      value or address: Empty()
9      position:      Pos(Num('1'), Num('5'))
10     size:          Empty()
11 },
12 Symbol
13 {
14     type qualifier: Writable()
15     datatype:      PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16     name:          Name('param1@stack_fun')
17     value or address: Num('0')
18     position:      Pos(Num('1'), Num('21'))
19     size:          Num('1')
20 },
21 Symbol
22 {
23     type qualifier: Writable()
24     datatype:      PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
25     name:          Name('param2@stack_fun')
26     value or address: Num('1')
27     position:      Pos(Num('1'), Num('37'))
28     size:          Num('1')
29 },
30 Symbol
31 {
32     type qualifier: Empty()
33     datatype:      FunDecl(VoidType('void'), Name('main'), [])
34     name:          Name('main')
35     value or address: Empty()
36     position:      Pos(Num('4'), Num('5'))
37     size:          Empty()
38 },
39 Symbol
40 {
41     type qualifier: Writable()
42     datatype:      ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:          Name('local_var1@main')
44     value or address: Num('0')
45     position:      Pos(Num('5'), Num('6'))
46     size:          Num('6')
47 },
48 Symbol
49 {
50     type qualifier: Writable()
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 3.81: 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         # // Exp(Alloc(Writable(), PntrDecl(Num('1')), ArrayDecl([Num('3')],
8           ↪ IntType('int'))), Name('param1'))
9         # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param2'))
10        # Return(Empty())
11        LOADIN BAF PC -1;
12      ],
13    Block
14      Name 'main.0',
15      [
16        # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
17          ↪ Name('local_var1'))
18        # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
19          ↪ Name('local_var2'))
20        # // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
21        # StackMalloc(Num('2'))
22        SUBI SP 2;
23        # Ref(Global(Num('0'))))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        STOREIN SP IN1 1;
28        # Ref(Global(Num('6'))))
29        SUBI SP 1;
30        LOADI IN1 6;
31        ADD IN1 DS;
32        STOREIN SP IN1 1;
33        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
34        MOVE BAF ACC;
35        ADDI SP 4;
36        MOVE SP BAF;
37        SUBI SP 4;
38        STOREIN BAF ACC 0;
39        LOADI ACC GoTo(Name('addr@next_instr'));
40        ADD ACC CS;
41        STOREIN BAF ACC -1;
42        # Exp(GoTo(Name('stack_fun.1'))
43        Exp(GoTo(Name('stack_fun.1'))
44        # RemoveStackframe()
45        MOVE BAF IN1;
46        LOADIN IN1 BAF 0;
47        MOVE IN1 SP;
48        # Return(Empty())
49        LOADIN BAF PC -1;
50      ]
51    ]
52  ]

```

Code 3.82: RETI-Block Pass für Call by Sharing für Arrays

3.3.7.3.4 Umsetzung von Call by Value für Structs

```

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

```

Code 3.83: 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         // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('param')))
8         Return(Empty())
9       ],
10    Block
11      Name 'main.0',
12      [
13        // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('local_var')))
14        // Exp(Call(Name('stack_fun'), [Name('local_var')]))
15        StackMalloc(Num('2'))
16        Assign(Stack(Num('3')), Global(Num('0')))
17        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
18        Exp(GoTo(Name('stack_fun.1')))
19        RemoveStackframe()
20        Return(Empty())
21      ]
22  ]

```

Code 3.84: 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         # // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('param')))
8         # Return(Empty())
9         LOADIN BAF PC -1;
10      ],
11    Block

```

```

12     Name 'main.0',
13     [
14         # // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('local_var')))
15         # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
16         # StackMalloc(Num('2'))
17         SUBI SP 2;
18         # Assign(Stack(Num('3')), Global(Num('0')))
19         SUBI SP 3;
20         LOADIN DS ACC 0;
21         STOREIN SP ACC 1;
22         LOADIN DS ACC 1;
23         STOREIN SP ACC 2;
24         LOADIN DS ACC 2;
25         STOREIN SP ACC 3;
26         # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
27         MOVE BAF ACC;
28         ADDI SP 5;
29         MOVE SP BAF;
30         SUBI SP 5;
31         STOREIN BAF ACC 0;
32         LOADI ACC GoTo(Name('addr@next_instr'));
33         ADD ACC CS;
34         STOREIN BAF ACC -1;
35         # Exp(GoTo(Name('stack_fun.1')))
36         Exp(GoTo(Name('stack_fun.1')))
37         # RemoveStackframe()
38         MOVE BAF IN1;
39         LOADIN IN1 BAF 0;
40         MOVE IN1 SP;
41         # Return(Empty())
42         LOADIN BAF PC -1;
43     ]
44 ]

```

Code 3.85: RETI-Block Pass für Call by Value für Structs

3.3.8 Umsetzung kleinerer Details

3.4 Fehlermeldungen

3.4.1 Error Handler

3.4.2 Arten von Fehlermeldungen

3.4.2.1 Syntaxfehler

3.4.2.2 Laufzeitfehler

4 Ergebnisse und Ausblick

4.1 Compiler

4.1.1 Überblick über Funktionen

4.1.2 Vergleich mit GCC

4.1.3 Showmode

4.2 Qualitätssicherung

4.3 Erweiterungsideen

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ (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.^a

^aEarley und Sturgis, „A formalism for translator interactions“.

Will man nun für eine Maschine M_{RETI} , auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 4.4) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ implementieren und hat bereits den gesamten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**¹ 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**

¹Beschreibt 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.

(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. **bootstrapped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} kopiert.²

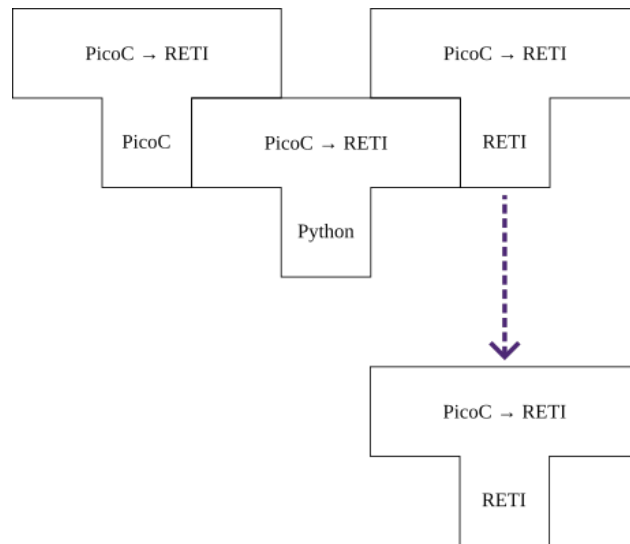


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

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^a 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.

^aIn diesem Fall, dem **Cross-Compiler** C_{PicoC}^{Python} .

Definition 4.2: Minimaler Compiler

Compiler C_{w_min} , der nur die **notwendigsten Funktionalitäten** einer Wunschsprache L_w , wie **Schleifen, Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** C_w^w oder einer **ersten Version** $C_{w_i}^{w_i}$ des Self-compiling Compilers C_w^w wichtig sind.^{a,b}

^aDen **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.

^bThiemann, „Compilerbau“.

²Im 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^a. Der **Bootstrapping Compiler** stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** C_w^w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.^b

^aDabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.

^bThiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$, der einen **minimalen Compiler** (Definition 4.2) für eine Teilmenge der **Programmiersprache** C bzw. L_C darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw. L_C für die Maschine M_{RETI} mittels **Bootstrapping** implementieren.³

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

Definition 4.4: Bootstrapping

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

^aZ.B. mithilfe eines **Bootstrap Compilers**.

^bDer Begriff hat seinen Ursprung in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügend Geschichten des Freiherrn von Münchhausen** bekannten Redewendung „sich am eigenen Schopf aus dem Sumpf ziehen“ entspricht.

^cHat 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.

^dEinen 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.

^eEs 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.

^fDer 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.

^gEarley und Sturgis, „A formalism for translator interactions“.

³Natürlich 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} .

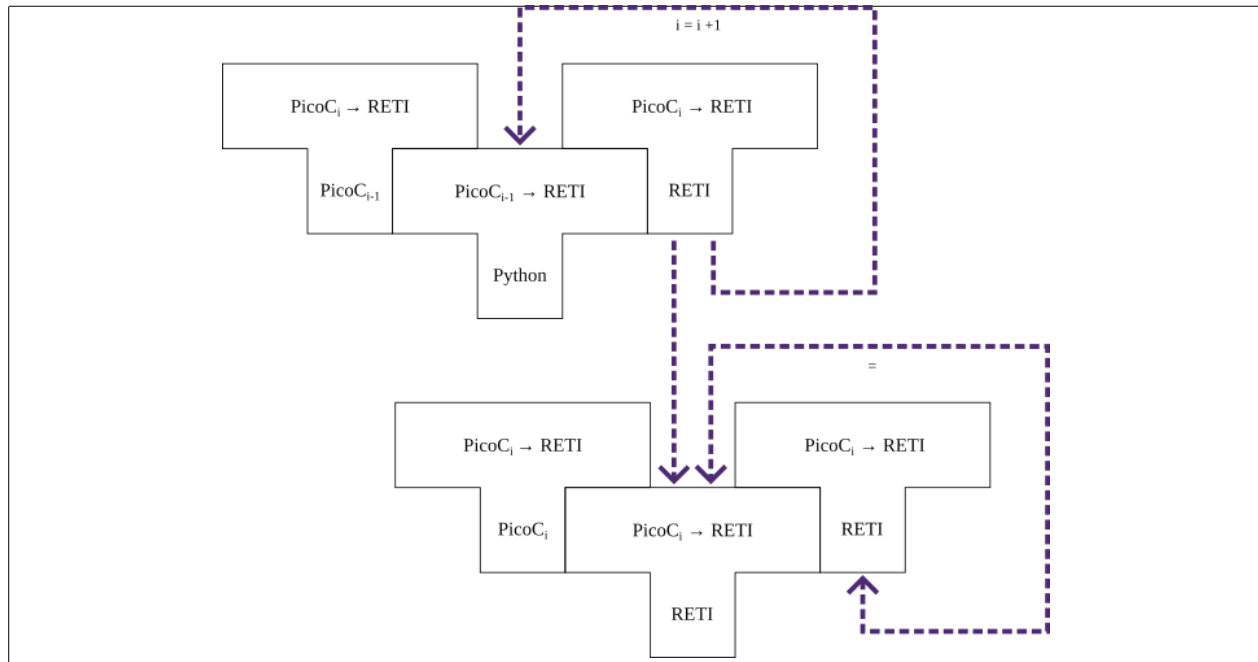


Abbildung 4.2: Iteratives Bootstrapping

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 (besucht am 27.04.2022).
- *Errors in C/C++* - *GeeksforGeeks*. URL: <https://www.geeksforgeeks.org/errors-in-cc/> (besucht am 10.05.2022).
- *GCC, the GNU Compiler Collection* - *GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *JSON parser - Tutorial* — *Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09.07.2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: <https://reverseengineering.stackexchange.com/q/17671> (besucht am 13.04.2022).
- *Parsing Expressions · Crafting Interpreters*. URL: <https://www.craftinginterpreters.com/parsing-expressions.html> (besucht am 09.07.2022).
- *Transformers & Visitors* — *Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/visitors.html> (besucht am 09.07.2022).
- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *What is Top-Down Parsing?* URL: <https://www.tutorialspoint.com/what-is-top-down-parsing> (besucht am 22.06.2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).

Artikel

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

Vorlesungen

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09.07.2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).

Sonstige Quellen

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