

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

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	IV
Definitionsverzeichnis	VI
Grammatikverzeichnis	VII
1 Motivation	1
1.1 RETI-Architektur	2
1.2 Die Sprache PicoC	2
1.3 Eigenheiten der Sprache C	3
1.4 Gesetzte Schwerpunkte	4
1.5 Über diese Arbeit	5
1.5.1 Still der Schriftlichen Ausarbeitung	6
1.5.2 Aufbau der Schriftlichen Arbeit	6
2 Einführung	8
2.1 Compiler und Interpreter	8
2.1.1 T-Diagramme	10
2.2 Formale Sprachen	12
2.2.1 Ableitungen	15
2.2.2 Präzidenz und Assoziativität	18
2.3 Lexikalische Analyse	19
2.4 Syntaktische Analyse	22
2.5 Code Generierung	29
2.5.1 Monadische Normalform	30
2.5.2 A-Normalform	31
2.5.3 Ausgabe des Maschinencodes	33
2.6 Fehlermeldungen	34
3 Implementierung	36
3.1 Lexikalische Analyse	38
3.1.1 Konkrete Syntax für die Lexikalische Analyse	38
3.1.2 Codebeispiel	40
3.2 Syntaktische Analyse	41
3.2.1 Umsetzung von Präzidenz und Assoziativität	41
3.2.2 Konkrete Syntax für die Syntaktische Analyse	46
3.2.3 Ableitungsbaum Generierung	48
3.2.3.1 Codebeispiel	50
3.2.3.2 Ausgabe des Ableitungsbaums	51
3.2.4 Ableitungsbaum Vereinfachung	51
3.2.4.1 Codebeispiel	53
3.2.5 Abstrakt Syntax Tree Generierung	54
3.2.5.1 PicoC-Knoten	56

3.2.5.2	RETI-Knoten	61
3.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	62
3.2.5.4	Abstrakte Syntax	64
3.2.5.5	Codebeispiel	65
3.2.5.6	Ausgabe des Abstrakter Syntaxbaum	66
3.3	Code Generierung	66
3.3.1	Passes	68
3.3.1.1	PicoC-Shrink Pass	69
3.3.1.1.1	Aufgabe	69
3.3.1.1.2	Abstrakte Syntax	69
3.3.1.1.3	Codebeispiel	70
3.3.1.2	PicoC-Blocks Pass	72
3.3.1.2.1	Aufgabe	72
3.3.1.2.2	Abstrakte Syntax	72
3.3.1.2.3	Codebeispiel	74
3.3.1.3	PicoC-ANF Pass	75
3.3.1.3.1	Aufgabe	75
3.3.1.3.2	Abstrakte Syntax	76
3.3.1.3.3	Codebeispiel	78
3.3.1.4	RETI-Blocks Pass	79
3.3.1.4.1	Aufgabe	79
3.3.1.4.2	Abstrakte Syntax	79
3.3.1.4.3	Codebeispiel	80
3.3.1.5	RETI-Patch Pass	83
3.3.1.5.1	Aufgabe	83
3.3.1.5.2	Abstrakte Syntax	83
3.3.1.5.3	Codebeispiel	84
3.3.1.6	RETI Pass	87
3.3.1.6.1	Aufgabe	87
3.3.1.6.2	Konkrete und Abstrakte Syntax	87
3.3.1.6.3	Codebeispiel	89
3.3.2	Umsetzung von Pointern	92
3.3.2.1	Referenzierung	92
3.3.2.2	Dereferenzierung durch Zugriff auf Arrayindex ersetzen	94
3.3.3	Umsetzung von Arrays	95
3.3.3.1	Initialisierung von Arrays	95
3.3.3.2	Zugriff auf einen Arrayindex	100
3.3.3.3	Zuweisung an Arrayindex	105
3.3.4	Umsetzung von Structs	108
3.3.4.1	Deklaration und Definition von Structtypen	108
3.3.4.2	Initialisierung von Structs	110
3.3.4.3	Zugriff auf Structattribut	113
3.3.4.4	Zuweisung an Structattribut	117
3.3.5	Umsetzung des Zugriffs auf Derived datatypes im Allgemeinen	119
3.3.5.1	Übersicht	119
3.3.5.2	Anfangsteil	122
3.3.5.3	Mittelteil	124
3.3.5.4	Schluss teil	128
3.3.6	Umsetzung von Funktionen	132
3.3.6.1	Mehrere Funktionen	132
3.3.6.1.1	Sprung zur Main Funktion	135
3.3.6.2	Funktionsdeklaration und -definition und Umsetzung von Scopes	137
3.3.6.3	Funktionsaufruf	140
3.3.6.3.1	Rückgabewert	145

3.3.6.3.2	Umsetzung von Call by Sharing für Arrays	149
3.3.6.3.3	Umsetzung von Call by Value für Structs	153
4	Ergebnisse und Ausblick	157
4.1	Funktionsumfang	157
4.1.1	Kommandozeilenoptionen	157
4.1.2	Shell-Mode	160
4.1.3	Show-Mode	161
4.2	Qualitätssicherung	163
4.3	Erweiterungsideen	167
4.4	Fehlermeldungen	171
	Appendix	A
	Danksagungen	H
	Literatur	I

Abbildungsverzeichnis

1.1	Schritte zum Ausführen eines Programmes mit dem GCC	1
1.2	Stark vereinfachte Schritte zum Ausführen eines Programmes	1
1.3	README.md im Github Repository der Bachelorarbeit	5
2.1	Horizontale Übersetzungszwischenschritte zusammenfassen	12
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen	12
2.3	Veranschaulichung von Linksassoziativität und Rechtsassoziativität	19
2.4	Veranschaulichung von Präzidenz	19
2.5	Veranschaulichung der Lexikalischen Analyse	22
2.6	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.	26
2.7	Veranschaulichung der Syntaktischen Analyse	28
2.8	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten	31
2.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen	33
3.1	Ableitungsbäume zu den beiden Ableitungen	43
3.2	Ableitungsbaum nach Parsen eines Ausdrucks	52
3.3	Ableitungsbaum nach Vereinfachung	53
3.4	Abstrakter Syntaxbaum Generierung ohne Umdrehen	55
3.5	Abstrakter Syntaxbaum Generierung mit Umdrehen	55
3.6	Cross-Compiler Kompiliervorgang ausgeschrieben	67
3.7	Cross-Compiler Kompiliervorgang Kurzform	67
3.8	Architektur mit allen Passes ausgeschrieben	68
3.9	Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes	120
4.1	Show-Mode in der Verwendung	163
5.1	Cross-Compiler als Bootstrap Compiler	E
5.2	Iteratives Bootstrapping	G

Codeverzeichnis

3.1	PicoC-Code des Codebeispiels	41
3.2	Tokens für das Codebeispiel	41
3.3	Ableitungsbaum nach Ableitungsbaum Generierung	51
3.4	Ableitungsbaum nach Ableitungsbaum Vereinfachung	54
3.5	Aus vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum	65
3.6	PicoC Code für Codebeispiel	71
3.7	Abstrakter Syntaxbaum für Codebeispiel	72
3.8	PicoC-Blocks Pass für Codebeispiel	75
3.9	PicoC-ANF Pass für Codebeispiel	79
3.10	RETI-Blocks Pass für Codebeispiel	83
3.11	RETI-Patch Pass für Codebeispiel	87
3.12	RETI Pass für Codebeispiel	91
3.13	PicoC-Code für Pointer Referenzierung	92
3.14	Abstrakter Syntaxbaum für Pointer Referenzierung	92
3.15	Symboltabelle für Pointer Referenzierung	93
3.16	PicoC-ANF Pass für Pointer Referenzierung	93
3.17	RETI-Blocks Pass für Pointer Referenzierung	94
3.18	PicoC-Code für Pointer Dereferenzierung	94
3.19	Abstrakter Syntaxbaum für Pointer Dereferenzierung	95
3.20	PicoC-Shrink Pass für Pointer Dereferenzierung	95
3.21	PicoC-Code für Array Initialisierung	96
3.22	Abstrakter Syntaxbaum für Array Initialisierung	96
3.23	Symboltabelle für Array Initialisierung	97
3.24	PicoC-ANF Pass für Array Initialisierung	98
3.25	RETI-Blocks Pass für Array Initialisierung	100
3.26	PicoC-Code für Zugriff auf einen Arrayindex	100
3.27	Abstrakter Syntaxbaum für Zugriff auf einen Arrayindex	101
3.28	PicoC-ANF Pass für Zugriff auf einen Arrayindex	103
3.29	RETI-Blocks Pass für Zugriff auf einen Arrayindex	105
3.30	PicoC-Code für Zuweisung an Arrayindex	105
3.31	Abstrakter Syntaxbaum für Zuweisung an Arrayindex	105
3.32	PicoC-ANF Pass für Zuweisung an Arrayindex	106
3.33	RETI-Blocks Pass für Zuweisung an Arrayindex	107
3.34	PicoC-Code für die Deklaration eines Structtyps	108
3.35	Abstrakter Syntaxbaum für die Deklaration eines Structtyps	108
3.36	Symboltabelle für die Deklaration eines Structtyps	110
3.37	PicoC-Code für Initialisierung von Structs	110
3.38	Abstrakter Syntaxbaum für Initialisierung von Structs	111
3.39	PicoC-ANF Pass für Initialisierung von Structs	112
3.40	RETI-Blocks Pass für Initialisierung von Structs	113
3.41	PicoC-Code für Zugriff auf Structattribut	113
3.42	Abstrakter Syntaxbaum für Zugriff auf Structattribut	113
3.43	PicoC-ANF Pass für Zugriff auf Structattribut	116
3.44	RETI-Blocks Pass für Zugriff auf Structattribut	117
3.45	PicoC-Code für Zuweisung an Structattribut	117
3.46	Abstrakter Syntaxbaum für Zuweisung an Structattribut	117
3.47	PicoC-ANF Pass für Zuweisung an Structattribut	118

3.48	RETI-Blocks Pass für Zuweisung an Structattribut	119
3.49	PicoC-Code für den Anfangsteil	122
3.50	Abstrakter Syntaxbaum für den Anfangsteil	123
3.51	PicoC-ANF Pass für den Anfangsteil	124
3.52	RETI-Blocks Pass für den Anfangsteil	124
3.53	PicoC-Code für den Mittelteil	125
3.54	Abstrakter Syntaxbaum für den Mittelteil	125
3.55	PicoC-ANF Pass für den Mittelteil	126
3.56	RETI-Blocks Pass für den Mittelteil	128
3.57	PicoC-Code für den Schlussteil	128
3.58	Abstrakter Syntaxbaum für den Schlussteil	129
3.59	PicoC-ANF Pass für den Schlussteil	129
3.60	RETI-Blocks Pass für den Schlussteil	131
3.61	PicoC-Code für 3 Funktionen	132
3.62	Abstrakter Syntaxbaum für 3 Funktionen	133
3.63	PicoC-Blocks Pass für 3 Funktionen	134
3.64	PicoC-ANF Pass für 3 Funktionen	134
3.65	RETI-Blocks Pass für 3 Funktionen	135
3.66	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	135
3.67	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	136
3.68	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	137
3.69	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	137
3.70	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	138
3.71	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	140
3.72	PicoC-Code für Funktionsaufruf ohne Rückgabewert	140
3.73	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert	141
3.74	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert	142
3.75	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	143
3.76	RETI-Pass für Funktionsaufruf ohne Rückgabewert	145
3.77	PicoC-Code für Funktionsaufruf mit Rückgabewert	145
3.78	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert	146
3.79	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert	147
3.80	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	149
3.81	PicoC-Code für Call by Sharing für Arrays	149
3.82	Symboltabelle für Call by Sharing für Arrays	151
3.83	PicoC-ANF Pass für Call by Sharing für Arrays	152
3.84	RETI-Block Pass für Call by Sharing für Arrays	153
3.85	PicoC-Code für Call by Value für Structs	153
3.86	PicoC-ANF Pass für Call by Value für Structs	154
3.87	RETI-Block Pass für Call by Value für Structs	156
4.1	Shellaufruf und die Befehle <code>compile</code> und <code>quit</code>	160
4.2	Shell-Mode und der Befehl <code>most_used</code>	161
4.3	Typischer Test	165
4.4	Testdurchlauf	167
4.5	Beispiel für Tail Call	170
4.6	Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet	171
4.7	Fehlermeldung des GCC	173
4.8	Beispiel für typische Fehlermeldung mit 'found' und 'expected'	173
4.9	Beispiel Fehlermeldung langgestreckte fehlermeldung	173
4.10	Beispiel für Fehlermeldung mit mehreren erwarteten Tokens	174
4.11	Beispiel für Fehlermeldung ohne expected	174

Tabellenverzeichnis

3.1	Präzidenzregeln von PicoC	42
3.2	Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren	44
3.3	PicoC-Knoten Teil 1	57
3.4	PicoC-Knoten Teil 2	58
3.5	PicoC-Knoten Teil 3	59
3.6	PicoC-Knoten Teil 4	60
3.7	RETI-Knoten	62
3.8	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	63
4.1	Kommandozeilenoptionen	159
4.2	Makefileoptionen	162
4.3	Testkategorien	164
4.4	Syntaktische Fehlerarten	171
4.5	Semantische Fehlerarten	172
4.6	Laufzeit Fehlerarten	172

Definitionsverzeichnis

1.1	Deklaration	3
1.2	Definition	3
1.3	Scope	3
1.4	Call by value	3
1.5	Call by reference	3
1.6	Imperative Programmierung	4
1.7	Strukturierte Programmierung	4
1.8	Prozedurale Programmierung	4
2.1	Interpreter	8
2.2	Compiler	8
2.3	Maschinensprache	9
2.4	Immediate	9
2.5	Cross-Compiler	10
2.6	T-Diagram Programm	10
2.7	T-Diagram Übersetzer (bzw. eng. Translator)	11
2.8	T-Diagram Interpreter	11
2.9	T-Diagram Maschine	11
2.10	Symbol	12
2.11	Alphabet	13
2.12	Wort	13
2.13	Formale Sprache	13
2.14	Syntax	13
2.15	Semantik	13
2.16	Formale Grammatik	13
2.17	Chomsky Hierarchie	14
2.18	Reguläre Grammatik	15
2.19	Kontextfreie Grammatik	15
2.20	Wortproblem	15
2.21	1-Schritt-Ableitungsrelation	16
2.22	Ableitungsrelation	16
2.23	Links- und Rechtsableitungableitung	16
2.24	Linksrekursive Grammatiken	16
2.25	Formaler Ableitungsbaum	17
2.26	Mehrdeutige Grammatik	18
2.27	Assoziativität	18
2.28	Präzidenz	19
2.29	Pipe-Filter Architekturpattern	19
2.30	Pattern	20
2.31	Lexeme	20
2.32	Lexer (bzw. Scanner oder auch Tokenizer)	20
2.33	Bezeichner (bzw. Identifier)	21
2.34	Literal	22
2.35	Konkrete Syntax	23
2.36	Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)	23
2.37	Parser	23
2.38	Recognizer (bzw. Erkennen)	24
2.39	Transformer	25

2.40	Visitor	25
2.41	Abstrakte Syntax	26
2.42	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)	26
2.43	Pass	29
2.44	Reiner Ausdruck (bzw. engl. pure expression)	30
2.45	Unreiner Ausdruck	30
2.46	Monadische Normalform (bzw. engl. monadic normal form)	30
2.47	Location	31
2.48	Atomarer Ausdruck	32
2.49	Komplexer Ausdruck	32
2.50	A-Normalform (ANF)	32
2.51	Fehlermeldung	34
3.1	Metasyntax	36
3.2	Metasprache	36
3.3	Erweiterte Backus-Naur-Form (EBNF)	36
3.4	Dialekt der EBNF aus Lark	37
3.5	Abstrakte Syntax Form (ASF)	37
3.6	Earley Parser	48
3.7	Earley Recognizer	49
3.8	Label	61
3.9	Token-Knoten	61
3.10	Container-Knoten	61
3.11	Symboltabelle	76
3.12	Entarteter Baum	115
3.13	Funktionsprototyp	138
3.14	Scope (bzw. Sichtbarkeitsbereich)	139
5.1	Assemblersprache (bzw. engl. Assembly Language)	A
5.2	Assembler	A
5.3	Objectcode	A
5.4	Linker	B
5.5	Transpiler (bzw. Source-to-source Compiler)	B
5.6	Rekursiver Abstieg	B
5.7	LL(k)-Grammatik	B
5.8	Liveness Analyse	B
5.9	Live Variable	C
5.10	Graph Coloring	C
5.11	Interference Graph	C
5.12	Kontrollflussgraph	C
5.13	Kontrollfluss	C
5.14	Kontrollflussanalyse	C
5.15	Two-Space Copying Collector	D
5.16	Self-compiling Compiler	D
5.17	Minimaler Compiler	E
5.18	Bootstrap Compiler	E
5.19	Bootstrapping	F

Grammatikverzeichnis

2.1	Produktionen des Ableitungsbaums	17
3.1.1	Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF	40
3.2.1	Undurchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF	42
3.2.2	Durchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF . .	43
3.2.3	Beispiel für eine unäre rechtsassoziative Produktion	44
3.2.4	Beispiel für eine unäre linksassoziative Produktion	44
3.2.5	Beispiel für eine linksassoziative Produktion	45
3.2.6	Beispiel für eine linksassoziative Produktion	45
3.2.7	Durchdachte Konkrete Syntax für Operatorpräzidenz in EBNF	46
3.2.8	Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1	47
3.2.9	Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	48
3.2.10	Abstrakte Syntax der Sprache L_{PiocC}	64
3.3.1	Abstrakte Syntax der Sprache L_{PiocC_Shrink}	70
3.3.2	Abstrakte Syntax der Sprache L_{PiocC_Blocks}	73
3.3.3	Abstrakte Syntax der Sprache L_{PiocC_ANF}	77
3.3.4	Abstrakte Syntax der Sprache L_{RETI_Blocks}	80
3.3.5	Abstrakte Syntax der Sprache L_{RETI_Patch}	84
3.3.6	Konkrete Syntax der Sprache L_{RETI} für die Lexikalische Analyse in EBNF	88
3.3.7	Konkrete Syntax der Sprache L_{RETI} für die Syntaktische Analyse in EBNF	88
3.3.8	Abstrakte Syntax der Sprache L_{RETI}	88

1 Motivation

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

Compiler, wie der **GCC**¹ oder **Clang**² werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert³ zu **Maschinencode** zu kompilieren.

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

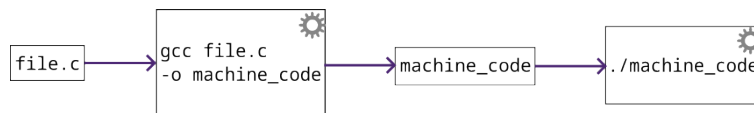


Abbildung 1.1: Schritte zum Ausführen eines Programmes mit dem GCC

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

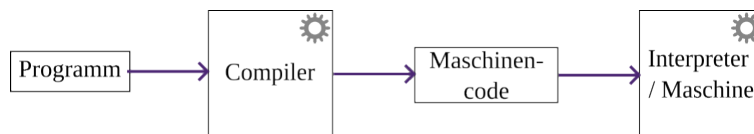


Abbildung 1.2: Stark vereinfachte Schritte zum Ausführen eines Programmes

¹GCC, the GNU Compiler Collection - GNU Project.

²clang: C++ Compiler.

³Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C .

⁴Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objectcode** (Definition 5.3) kompiliert wird. Das Kompilieren zu **Objectcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

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

Die konkrete **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC} , welche eine Untermenge der Sprache L_C ist⁵ in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zu Grunde liegt und im Unterkapitel 1.2 wird näher auf die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll.

1.1 RETI-Architektur

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

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

^aWelche unter anderem eine **Memory Map** von **EPROM**, **UART** und **SRAM** nutzt usw.

Der **Aufbau** der **Maschinensprache** L_{RETI} ist in Grammatik 3.3.6 und Grammatik 3.3.7 zusammengefasst dargestellt. Für die **Bedeutungen** der einzelnen **Register** und **Maschinenbefehle**, sowie **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

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

1.2 Die Sprache PicoC

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

- **Einzeilige Kommentare** `//` und **Mehrzeilige Kommentare** `/*` und `*/`
- die **Primitiven Datentypen** `int`, `char` und `void`
- die **Abgeleiteten Datentypen** **Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`)

⁵Die der **GCC** kompilieren kann.

- `if(cond){ }- / else{ }-Statements`⁶
- `while(cond){ }-` und `do while(cond){ };-Statements`
- **Arihmetische Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` und **unären Operatoren** `-`, `~` umgesetzt sind
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`) und **Funktionsaufrufe** (z.B. `fun(ar, st_var)`)

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

Der **Aufbau** der **Programmiersprache** L_C ist in Grammatik 3.1.1 und Grammatik 3.2.8 zusammengekommen beschrieben.

1.3 Eigenheiten der Sprache C

Definition 1.1: Deklaration

a

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

Definition 1.2: Definition

a

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

Definition 1.3: Scope

a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.4: Call by value

a

^aBast, „Programmieren in C“.

Definition 1.5: Call by reference

a

^aBast, „Programmieren in C“.

⁶Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

Definition 1.6: Imperative Programmierung

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

^aThiemann, „Einführung in die Programmierung“.

Definition 1.7: Strukturierte Programmierung

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

^aThiemann, „Einführung in die Programmierung“.

Definition 1.8: Prozedurale Programmierung

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

^aThiemann, „Einführung in die Programmierung“.

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinsprache L_{RETI} die **Syntax** und **Semantik** der Sprache L_C identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache L_{PicoC} im Vergleich zu z.B. dem **GCC**⁷ ohne merklichen Unterschied⁸ kompilieren können.

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

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

⁷Da die Sprache L_{PicoC} eine **Untermenge** von L_C ist, kann der **GCC** L_{PicoC} ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinsprache L_{RETI} .

⁸Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinsprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter [Link⁹](#) zu finden. In der Datei **README.md** (siehe Abbildung 1.3) ist unter „**Getting Started**“ ein kleines **Einführungstutorial** verlinkt. Unter „**Usage**“ ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Daneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter [Link¹⁰](#) zu finden.



Abbildung 1.3: *README.md* im Github Repository der Bachelorarbeit

Die **Schriftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schriftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter [Link¹¹](#) zu finden. Die **PDF** der Schriftliche Ausarbeitung der Bachelorarbeit wird aus dem **Latexquellcode**, welcher unter [Link¹²](#) veröffentlicht ist automatisch mithilfe der **Github Action** Nemec, *copy-file-to-another-repo-action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibliotheken** sind unter [Link¹³](#) zu finden¹⁴. Die Grafiken, die nicht mittels der Tikz Bibliothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**¹⁵ erstellt. Falls Interesse besteht **Grafiken** aus der Schriftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den **.svg**-Dateien von **Inkscape** im Ordner **/figures** zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibliotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der **README.md** unter „**References**“ bzw. direkt unter [Link¹⁶](#) zu finden.

⁹<https://github.com/matthejue/PicoC-Compiler>.

¹⁰<https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971>.

¹¹https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.

¹²<https://github.com/matthejue/Bachelorarbeit>.

¹³https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.

¹⁴Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.

¹⁵Developers, *Draw Freely — Inkscape*.

¹⁶https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

1.5.1 Still der Schriftlichen Ausarbeitung

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

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas genau so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht¹⁷ die Frage des „**warum**“.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**¹⁸ verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **konsistent** sind, wie auch das in Ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernommen, so wurde die Definition oder der entsprechende Teil in „**Anführungszeichen**“ gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox**, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

1.5.2 Aufbau der Schriftlichen Arbeit

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

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

Im Kapitel **Einführung** werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische Vorgehen**, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen in Bezug zu Compilern und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

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

Im Kapitel **Implementierung** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers**, unterteilt in die verschiedenen **Phasen der Kompilierung** nach denen das Kapitel **Einführung** ebenfalls unterteilt ist erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähnliche**

¹⁷Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.

¹⁸Das ist ein **Beispiel** für eine **Fußnote**.

Kapiteleinteilung haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständnis eines Aspekts in Kapitel **Implementierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteleinteilung** die entsprechende Definition analog im Kapitel **Einleitung** gefunden werden.

Im Kapitel **Ergebnisse und Ausblick** wird ein **Überblick** über die **wichtigsten Funktionalitäten** des PicoC-Compilers gegeben, indem anhand **kleiner Anleitungen** gezeigt wird wie man diese verwendet. Des Weiteren wird darauf eingegangen, wie die **Qualitätsicherung** für den **PicoC-Compiler** umgesetzt wurde, also wie gewährleistet wird, dass der **PicoC-Compiler** funktioniert. Zum Schluss wird noch auf **weitere Erweiterungsideen** eingegangen, die auch interessant zu implementieren wären.

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

Generell wurde in der Schriftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition 5.8) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

2 Einführung

2.1 Compiler und Interpreter

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

Definition 2.1: Interpreter

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

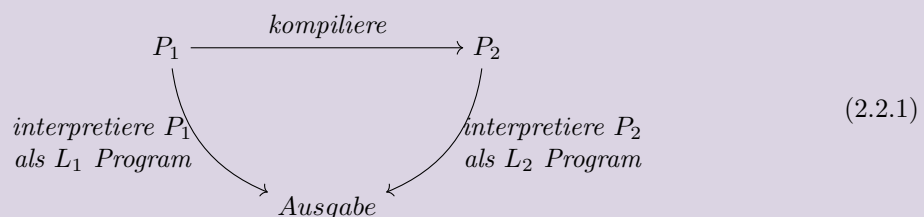
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstrakter Syntaxbaum** (Definition 2.42) und führt je nach Komposition der **Nodes** des Abstrakter Syntaxbaum, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.^a*

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

Definition 2.2: Compiler

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

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



^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 5.17) kann man das als C_{min} schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschinencode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 5.5) oder **Cross-Compiler** (Definition 2.5). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 5.1) 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 2.4) haben.

^bP. D. C. Scholl, „Betriebssysteme“.

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 **Anschaungs- und Lernwerkzeug** zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.4) 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 2.4: 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*

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 Schritt einen Befehl verarbeiten kann, da die Register und Speicherzellen des Hauptspeichers üblicherweise nur **32- bzw. 64-Bit Breite** haben.

^aLjohhuh, *What is an immediate value?*

Definition 2.5: Cross-Compiler

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

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

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

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

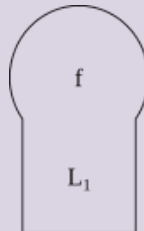
2.1.1 T-Diagramme

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

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

Definition 2.6: T-Diagramm Programm

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



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

Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.6 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 Maschinensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst nicht genug Rechenleistung besitzt, um ein Programm selbst zeitnah zu kompilieren.

Definition 2.7: 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**.

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

Definition 2.8: T-Diagramm Interpreter

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



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

Definition 2.9: 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).

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

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

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

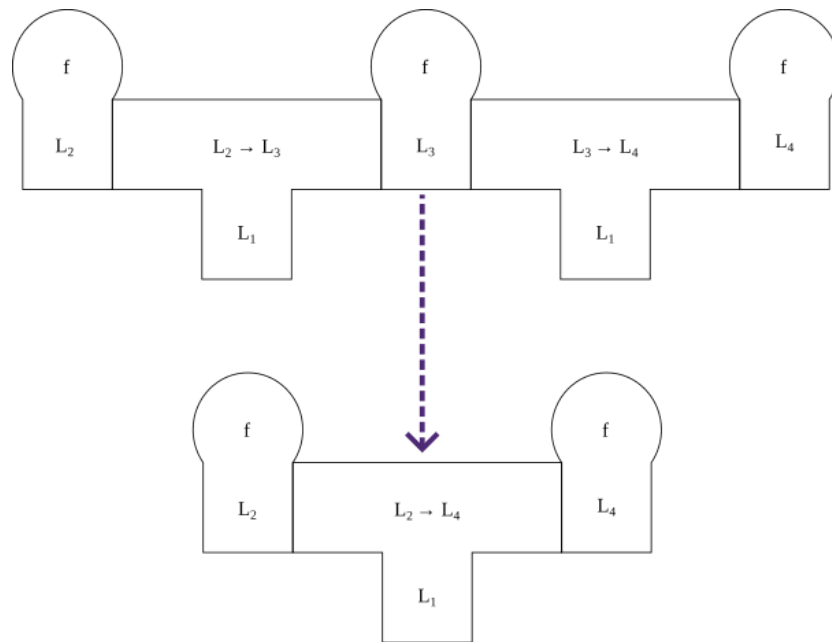


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

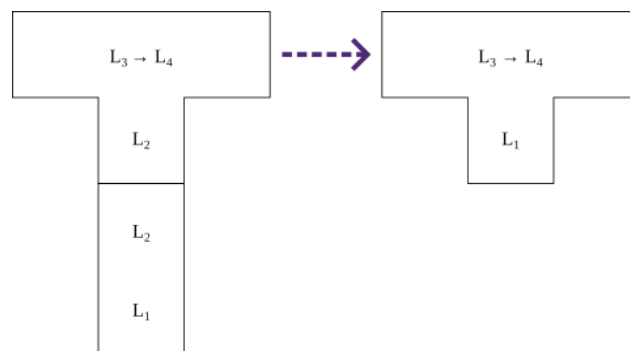


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.13) zu tun, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der Sprache L_1 in eine Sprache L_2 ist. Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen**, was die Begriffe **Symbol** (Definition 2.10), **Alphabet** (Definition 2.11), **Wort** (Definition 2.12) usw. beinhaltet vorher eingeführt zu haben.

Definition 2.10: Symbol

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

^aNebel, „Theoretische Informatik“.

Definition 2.11: Alphabet

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

^aNebel, „Theoretische Informatik“.

Definition 2.12: Wort

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

^aNebel, „Theoretische Informatik“.

Definition 2.13: Formale Sprache

„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.12) über dem **Alphabet** Σ (Definition 2.11).“^a

Das Adjektiv „**formal**“ kann dabei weggelassen werden, wenn der **Kontext** indem die **Sprache** verwendet wird **eindeutig** ist, da man das Adjektiv „**formal**“ nur verwendet um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Sprache** herauszustellen.

^aNebel, „Theoretische Informatik“.

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die **Semantik** (Definition 2.15) **gleich** bleiben. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 2.16), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 2.14) haben.

Definition 2.14: Syntax

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

^aZ.B. die **Konkrete** und **Abstrakte Syntax**, die später eingeführt werden.

^bThiemann, „Einführung in die Programmierung“.

Definition 2.15: Semantik

Die Semantik bezeichnet alles was mit der **Bedeutung** von **Formalen Sprachen** zu tun hat.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 2.16: Formale Grammatik

„Eine Formale Grammatik beschreibt wie **Wörter** einer **Sprache** abgeleitet werden können.“^a

Das Adjektiv „**formal**“ kann dabei weggelassen werden, wenn der **Kontext** indem die **Grammatik** verwendet wird **eindeutig** ist, da man das Adjektiv „**formal**“ nur verwendet um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Grammatik** herauszustellen.

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

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

- $\Sigma \hat{=} \text{Terminalsymbole}$, wobei $N \cap \Sigma = \emptyset^{b,c}$.
- $P \hat{=} \text{Menge von Produktionsregeln}$ $w \rightarrow v$, wobei $w, v \in (N \cup \Sigma)^* \wedge w \notin \Sigma^*$.^{de}
- $S \hat{=} \text{Startsymbol}$, wobei $S \in N$.

Zusätzlich ist es praktisch **Nicht-Terminalsymbole** N , **Terminalsymbole** Σ und das **leere Wort** ε allgemein als Menge der **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ zu definieren.

^aNebel, „Theoretische Informatik“.

^bWeil mit ihnen **terminiert** wird.

^cKann auch als **Alphabet** bezeichnet werden.

^d w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.

^eBzw. $w, v \in V^* \wedge w \notin \Sigma^*$.

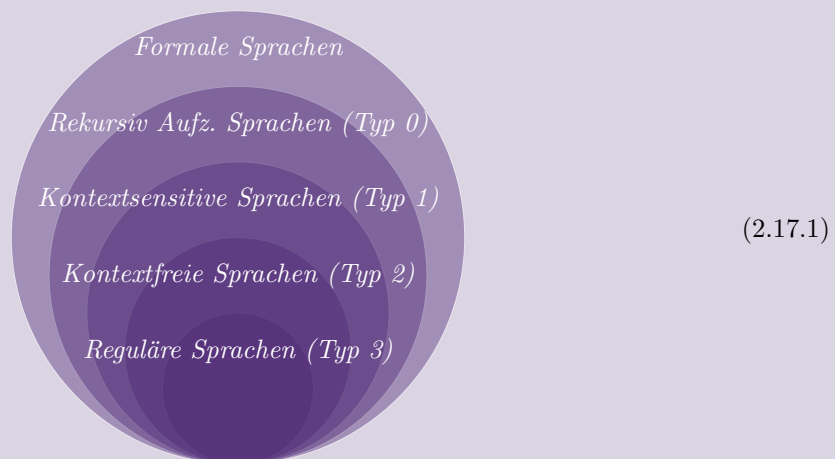
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 2.17) einteilen.

Definition 2.17: Chromsky Hierarchie

Die Chromsky Hierarchie ist eine Hierarchie in der **Formale Sprachen** nach der **Komplexität** ihrer **Formalen Grammatiken** in verschiedene **Klassen** unterteilt werden. Jede dieser Klassen hat verschiedene **Eigenschaften**, wie **Entscheidungsprobleme**, die in dieser Klasse **entscheidbar** bzw. **unentscheidbar** sind usw.

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

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



^aNebel, „Theoretische Informatik“.

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

Definition 2.18: Reguläre Grammatik

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

$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \varepsilon \quad (2.18.1)$$

haben, wobei A, B **Nicht-Terminalsymbole** sind und c ein **Terminalsymbol** ist^{a,b}.“^c

^aDiese Definition einer **Regulären Grammatik** ist **rechtsregulär**, es ist auch möglich diese Definition **linksregulär** zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.

^bDadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.

^cNebel, „Theoretische Informatik“.

Definition 2.19: Kontextfreie Grammatik

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

$$A \rightarrow v \quad (2.19.1)$$

haben, wobei A ein **Nicht-Terminalsymbol** ist und v ein beliebige Folge von **Grammatiksymbolen**^a ist.“^b

^aAlso eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

^bNebel, „Theoretische Informatik“.

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des **Wortproblems** (Definition 2.20). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise immer **entscheidbar**. Wenn das Programm ein **Wort** der **Sprache** ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es **kein Wort** der **Sprache**, die der Compiler kompiliert, wird eine **Fehlermeldung** ausgegeben.

Definition 2.20: Wortproblem

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

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

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

^aBzw. „ja“ oder „nein“ usw., es muss nicht umgedingt 1 oder 0 sein.

^bNebel, „Theoretische Informatik“.

^cAuch **Charakteristische Funktion** genannt.

2.2.1 Ableitungen

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

⁴Bzw. **Wort**.

Definition 2.21: 1-Schritt-Ableitungsrelation

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

Es gilt $u \Rightarrow v$ **genau dann wenn** $u = w_1 x w_2$, $v = w_1 y w_2$ **und** es eine Regel $x \rightarrow y \in P$ gibt, wobei $w_1, w_2, x, y \in (N \cup \Sigma)^*$ “^a

^aNebel, „Theoretische Informatik“.

Definition 2.22: Ableitungsrelation

„Eine **binäre Relation** \Rightarrow^* , welche der **reflexive, transitive Abschluss** der **1-Schritt-Ableitungsrelation** \Rightarrow ist. Auf der **rechten** Seite der Ableitungsrelation \Rightarrow^* steht also ein Wort aus $(N \cup \Sigma)^*$, welches durch **beliebig häufiges** Anwenden von Produktionsregeln entsteht.

Es gilt $u \Rightarrow^* v$ **genau dann wenn** $u = w_1 \Rightarrow \dots \Rightarrow w_n = v$, wobei $n \geq 1$ und $w_1, \dots, w_n \in (N \cup \Sigma)^*$.“^a

^aNebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**⁵ kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 relevant.

Definition 2.23: Links- und Rechtsableitung

„In jedem **Ableitungsschritt** wird bei **Typ-3- und Typ-2-Grammatiken** auf das am **weitesten links** (**Linksableitung**) bzw. **rechts** (**Rechtsableitung**) stehende **Nicht-Terminalsymbol** eine Produktionsregel angewandt, bei **Typ-1- und Typ-0-Grammatiken** ist es statt einem **Nicht-Terminalsymbol** die **linke** Seite einer Produktion.

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

^aNebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 2.24) ist⁶.

Definition 2.24: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

^aParsing Expressions · Crafting Interpreters.

⁵Bzw. **Wort**.

⁶Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (siehe Unterkapitel ??) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 2.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 2.36) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 2.25: Formaler Ableitungsbaum

Ist ein Baum, in dem die **Konkrete Syntax** eines **Wortes**^a nach den **Produktionen** der zugehörigen Grammatik, die angewendet werden mussten um das Wort **abzuleiten** zergliedert **hierarchisch** dargestellt wird.

Das Adjektiv „**formal**“ kann dabei weggelassen werden, wenn der **Kontext** indem der **Ableitungsbaum** verwendet wird **eindeutig** ist, da man das Adjektiv „**formal**“ nur verwendet um den Unterschied zum **compilerinternen Ableitungsbaum** herauszustellen, der den Formalen Ableitungsbaum als **Datenstruktur** zur einfachen Weiterverarbeitung umsetzt.

Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.16) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** N und die **Blätter** sind entweder **Terminalsymbole** Σ oder das **leere Wort** ε .^b

^aZ.B. **Programmcode**.

^bNebel, „Theoretische Informatik“.

In Abbildung 2.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung 2.25.1** nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition 3.3) angegebenen **Produktionen 2.1** einer ansonsten nicht näher spezifizierten **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ ergibt.

DIG_NO_0	$::=$	"1"		"2"		"3"		"4"		"5"		"6"	L_Lex
		"7"		"8"		"9"							
DIG_WITH_0	$::=$	"0"		DIG_NO_0									
NUM	$::=$	"0"		DIG_NO_0	$DIG_WITH_0^*$								
add	$::=$	add	"+"	mul		mul							L_Parse
mul	$::=$	mul	"*"	NUM		NUM							

Grammatik 2.1: *Produktionen des Ableitungsbaums*

$$add \Rightarrow mul \Rightarrow mul \text{ "*" } NUM \Rightarrow NUM \text{ "*" } NUM \Rightarrow 4 \text{ "*" } NUM \Rightarrow 4 \text{ "*" } 2 \quad (2.25.1)$$

Bei Ableitungsbäumen gibt es **keine** einheitliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 2.25.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort** ε hinzuzufügen.



Eine andere Möglichkeit ist, wie im Ableitungsbaum 2.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen⁷.



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

Definition 2.26: Mehrdeutige Grammatik

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

^aAlternativ, wenn es für w **mehrere** unterschiedliche **Linksableitungen** gibt.

^bNebel, „Theoretische Informatik“.

2.2.2 Präferenz und Assoziativität

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

Definition 2.27: Assoziativität

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

Es wird grundsätzlich zwischen **linksassoziativen** Operatoren, bei denen der **linke Operator** vor dem **rechten Operator** ausgewertet wird und **rechtsassoziativen** Operatoren, bei denen es genau anders rum ist unterschieden.^a

^aParsing Expressions · Crafting Interpreters.

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

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

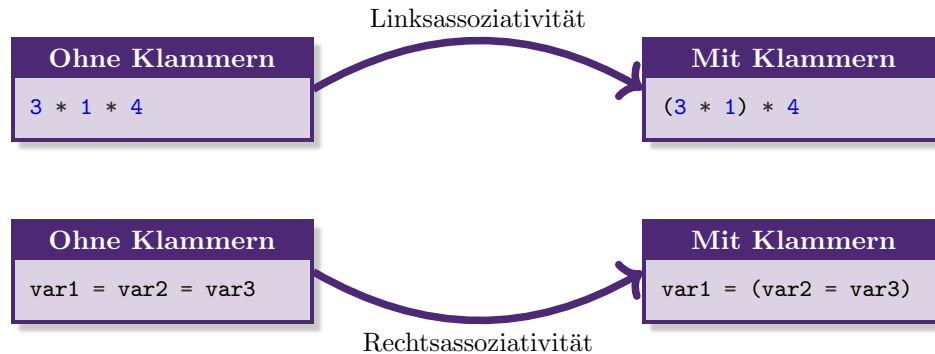


Abbildung 2.3: Veranschaulichung von Linksassoziativität und Rechtsassoziativität

Definition 2.28: Präzidenz

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

^aParsing Expressions · Crafting Interpreters.

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



Abbildung 2.4: Veranschaulichung von Präzidenz

2.3 Lexikalische Analyse

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

Definition 2.29: Pipe-Filter Architekturpattern

Ist ein **Architekturpattern**, welches aus **Pipes** und **Filtern** besteht, wobei der **Ausgang** eines **Filters** der **Eingang** des durch eine **Pipe** verbundenen adjazenten nächsten **Filters** ist, falls es einen gibt.

Ein **Filter** stellt einen Schritt dar, indem eine Eingabe **weiterverarbeitet** wird und **weitergereicht** wird. Bei der **Weiterverarbeitung** können Teile der Eingabe **entfernt**, **hinzugefügt** oder **vollständig ersetzt** werden.

Eine **Pipe** stellt ein **Bindeglied** zwischen zwei **Filtern** dar.^{a,b}



^aDas ein **Bindeglied** eine eigene Bezeichnung erhält, bedeutet allerdings nicht, dass es eine eigene wichtige **Aufgabe** erfüllt. Wie bei vielen **Pattern**, soll mit dem Namen des **Pattern**, in diesem Fall durch das **Pipe** die Anlehnung an z.B. die **Pipes aus Unix**, z.B. `cat /proc/bus/input/devices | less` zum Ausdruck gebracht werden. Und so banal es klingt, sollen manche Bezeichnungen von Pattern auch einfach nur gut klingen.

^bWestphal, „Softwaretechnik“.

Definition 2.30: Pattern

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

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

^bThiemann, „Compilerbau“.

Definition 2.31: Lexeme

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

^aThiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.32) 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 2.32) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

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

Ein **Lexer** ist eine **partielle Funktion** $lex : \Sigma^* \rightarrow (N \times W)^*$, 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 2.33) 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 **Symboltabelle** genannt wird.

Definition 2.33: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres **Scopes 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`⁸ und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** $*$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

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

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.34) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99¹⁰. 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** vorausschaut¹¹, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 3.1.1 liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.18 erfüllt. Einzig die Produktion `CHAR ::= "'ASCII_CHAR'"` sieht problematisch aus, kann allerdings auch

⁸In 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.

⁹Diese **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.

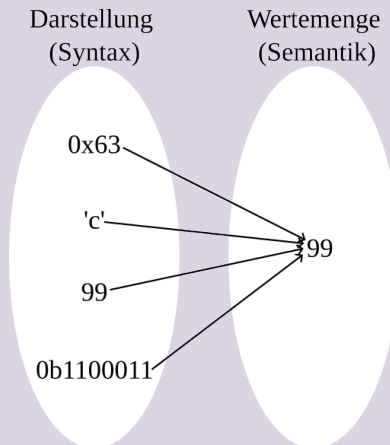
¹⁰Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

¹¹Man nennt das auch einem **Lookahead** von 1

als $\{\text{CHAR} ::= \text{" "CHAR2}, \text{CHAR2} ::= \text{ASCII_CHAR" "}\}$ **regulär** ausgedrückt werden¹². Somit existiert eine **reguläre Grammatik**, welche die **Sprache** $L_{\text{PicoC_Lex}}$ beschreibt und damit ist die **Sprache** $L_{\text{PicoC_Lex}}$ **regulär**.

Definition 2.34: 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 2.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

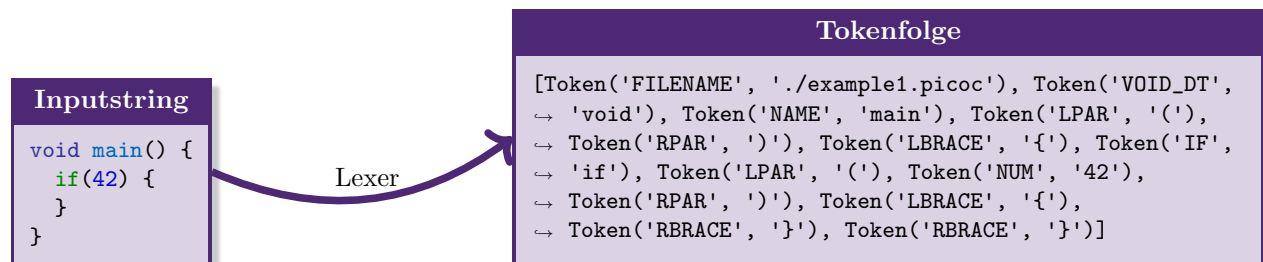


Abbildung 2.5: Veranschaulichung der Lexikalischen Analyse

2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** $\text{fun}(\text{arg})$ und **Codeblöcke** $\text{if}(1)\{\}$ 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 ein **Programm** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Defini-

¹²Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII.CHAR` **beliebig breite** Wörter ableiten lassen.

tion 2.37), ein **Ableitungsbaum** (Definition 2.36) generiert, der als Zwischenstufe hin zum einem **Abstrakter Syntaxbaum** (Definition 2.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaums** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 2.35: Konkrete Syntax

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

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

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

Definition 2.36: Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)

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

*Die **Konkrete Syntax** nach der sich der **Ableitungsbaum** richtet wird optimalerweise immer so definiert, dass sich möglichst einfach ein **Abstrakter Syntaxbaum** daraus konstruieren lässt.^a*

^aJSON parser - Tutorial — Lark documentation.

Definition 2.37: Parser

*Ein **Parser** ist ein Programm, dass aus einem Inputstring, der in **Konkreter Syntax** geschrieben ist, eine compilerinterne Darstellung, den **Ableitungsbaum** 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 2.37 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 Insekten**lexikon** und dem Aufschreiben, welchen Insekten man in welcher **Reihenfolge** begegnet ist. Zudem kann man bestimmte **Sehenswürdigkeiten** an denen man während des Ausflugs vorbeikommt ebenfalls festhalten, da es eine Rolle spielen kann in welchem örtlichen **Kontext** man den Insekten begegnet ist^a.

Der **Parser** vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich und entspricht, um auf das Beispiel zurückzukommen, dem Darstellen von **Beziehungen** zwischen den Insektenbeugnungen 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 2.38), denn ein Parser löst das **Wortproblem** (Definition 2.20) 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 **Ableitungsbaum**.

Definition 2.38: 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** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** 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. Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** (Definition 2.20) bekannt.^a*

^aThieman, „Compilerbau“.

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

- **Top-Down Parsing:** Der **Ableitungsbaum** wird von **oben-nach-unten** generiert, also von der **Wurzel** zu den **Blättern**. Dementsprechend fängt die Generierung des **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 das **Eingabewort** von **links nach rechts** eingelesen wird, was gut damit zusammenpasst, dass die **Linksableitung** die **Blätter** von **links-nach-rechts** generiert.

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

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

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.24) allerdings nicht möglich, ohne die 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 5.7) 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** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkreten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.^d
- **Chart Parsing:** Es wird **Dynamische Programmierung** verwendet und **partielle Zwischenergebnisse** werden in einer **Tabelle** (bzw. einem **Chart**) gespeichert und können **wiederverwendet** werden. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist^e. **Chart Parser** können dabei **top-down** oder **bottom-up** Ansätze umsetzen. Da die **Implementierung** von **Chart Parsern** fundamental anders ist als bei **Top-Down** und **Bottom-Up Parsern**, wird diese **Kategorie** von Parsern nochmal **speziell unterschieden** und nicht gesagt, es sei ein **Top-Down Parser** oder **Bottom-Up Parser**, der **Dynamische Programmierung** verwendet.

^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 **Abstrakter Syntaxbaum** wird mithilfe von **Transformern** (Definition 2.39) und **Visitors** (Definition 2.40) generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese ein Programm von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 2.41).

Definition 2.39: Transformer

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

^aTransformers & Visitors — Lark documentation.

Definition 2.40: Visitor

Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Ableitungsbaum** 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 **Abstrakter Syntaxbaum** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakter Syntaxbaum** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.

^bTransformers & Visitors — Lark documentation.

Definition 2.41: Abstrakte Syntax

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

Ein **Abstract Syntax Tree**, der zur Kompilierung eines Wortes^a generiert wurde, ist nach einer **Abstrakter Syntax** konstruiert.

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht. Dadurch sind die **Kompositionen**, welche die Knoten im **Abstract Syntax Tree** bilden können **syntaktisch** meist näher zur Syntax von **Maschinenbefehlen**.^b

^aZ.B. **Programmcode**.

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

Definition 2.42: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

Ist ein **compilerinterne Datenstruktur**, welche eine **Abstraktion** eines dazugehörigen **Ableitungsbaumes** darstellt, in dessen Aufbau auch das Erfordernis eines **leichten Zugriffs** und einer **leichten Weiterverarbeitbarkeit** eingeflossen ist. Bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, soll man möglichst schnell die Fragen beantworten können, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.

Im Gegensatz zum **Formalen Ableitungsbaum**, ergibt es beim **Abstrakten Syntaxbaum** keinen Sinn zusätzlich einen **Formalen Abstrakten Syntaxbaum** zu unterscheiden, da das Konzept eines **Abstrakten Syntaxbaumes** ohne eine Datenstruktur zu sein für sich allein gesehen keine Sinn hat. Wenn von Abstrakten Syntaxbäumen die Rede ist, ist immer eine **Datenstruktur** gemeint.

Die **Abstrakte Syntax** nach der sich der **Abstrakte Syntaxbaum** richtet wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten^a möglichst **einfach weiterverarbeitet** werden kann.

^aDie verschiedenen **Passes**.

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



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

Die **Baumdatenstruktur** des **Ableitungsbaumes** und **Abstrakten Syntaxbaumes** ermöglicht es die

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

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

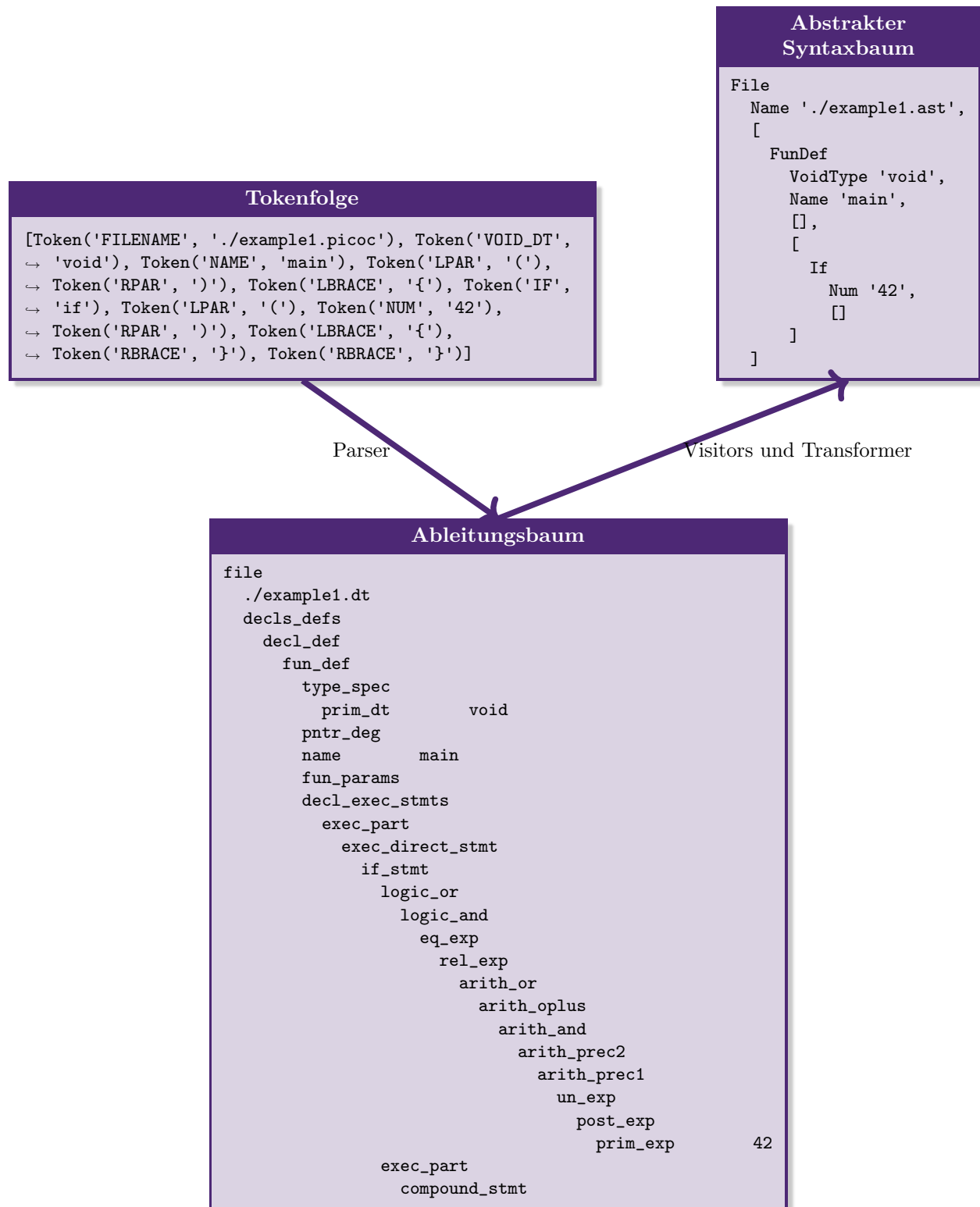


Abbildung 2.7: Veranschaulichung der Syntaktischen Analyse

2.5 Code Generierung

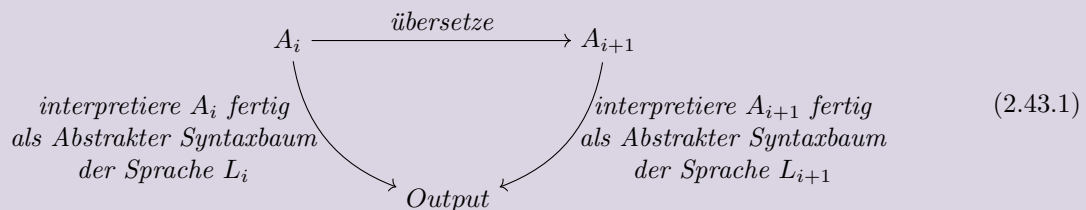
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** einer Sprache L_1 in den **Abstrakter Syntaxbaum** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.43) nennt. So wie es auch schon mit dem **Derivation Tree** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakter Syntaxbaum** konstruiert hatte. Aus dem Derivation konnte, dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

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

Definition 2.43: Pass

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

*Für jeden **Pass** und für einen beliebigen **Abstrakten Syntaxbaum** A_i gilt ähnlich, wie bei einem **vollständigen Compiler** in 2.43.1, dass:*



*wobei man hier so tut, als gäbe es zwei **Interpreter** für die zwei Sprachen L_i und L_{i+1} , welche den jeweiligen **Abstrakten Syntaxbaum** A_i bzw. A_{i+1} fertig interpretieren.^{cd}*

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

^bDer Begriff kommt aus dem **Englischen** von „passing over“, da der gesamte **Abstrakte Syntaxbaum** in einem **Pass** durchlaufen wird.

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

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

Die von den **Passes** umgeformten **Abstrakter Syntaxbaums** sollten dabei mit jedem **Pass** der **Syntax** von **Maschinenbefehlen** immer ähnlicher werden, bis es schließlich nur noch Maschinenbefehle sind.

2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tun, welche **Unreine Ausdrücke** (Definition 2.45) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.44) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 2.44: Reiner Ausdruck (bzw. engl. pure expression)

*Ein **Reiner Ausdruck** ist ein Ausdruck, der **rein** ist. Das bedeutet, dass dieser Ausdruck **keine Nebeneffekte** erzeugt. Ein **Nebeneffekt** ist eine **Bedeutung**, die ein Ausdruck hat, die sich **nicht** mit **RETI-Code** darstellen lässt.^{a,b}*

^aSondern z.B. **intern** etwas am **Kompilierprozess** ändert.

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

Definition 2.45: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definition 2.46).

Definition 2.46: Monadische Normalform (bzw. engl. monadic normal form)

*Ein **Statement** oder **Ausdruck** ist in **Monadischer Normalform**, wenn er nach einer **Konkreten Syntax** in **Monadischer Normalform** abgeleitet wurde.*

*Eine **Konkrete Syntax** ist in **Monadischer Normalform**, wenn sie **reine Ausdrücke** und **unreine Ausdrücke nicht** miteinander **mischt**, sondern voneinander **trennt**.^a*

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

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

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

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

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



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

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

2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.49) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Statements macht, in denen die **Komplexen Ausdrücke temporären Locations** zugewiesen werden (Definiton 2.47) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statement**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 2.48) enthalten.

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

Definition 2.47: Location

*Kollektiver Begriff für **Variablen**, **Attribute** bzw. **Elemente** von Variablen bestimmter Datentypen, **Speicherbereiche auf dem Stack**, die **temporäre Zwischenergebnisse** speichern und **Register**.*

*Im Grunde genommen alles, was mit einem **Programm zu tun** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.^a*

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.50). Wenn eine **Konkrete Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.50), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.48).

Definition 2.48: Atomarer Ausdruck

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

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

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

Definition 2.49: Komplexer Ausdruck

Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist, wie z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`.^{ab}

^a`int var` ist eine **Allokation**.

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

Definition 2.50: A-Normalform (ANF)

Ein **Statement** oder **Ausdruck** ist in **A-Normalform**, wenn er nach einer **Konkreten Syntax** in **A-Normalform** abgeleitet wurde.

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

Eine **Abstrakte Syntax** ist in **A-Normalform**, wenn die **Konkrete Syntax** für welche sie definiert wurde in **A-Normalform** ist.^{abc}

^aA-Normalization: *Why and How (with code)*.

^bBolingbroke und Peyton Jones, „Types are calling conventions“.

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

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

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition 5.10) inklusive **Liveness Analysis** (Definition 5.8) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.¹⁵

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

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

¹⁴Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

¹⁵Die in diesem **Paragraph** erwähnten **Begriffe** werden nur grob erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

befehlssatz es erlaubt **zwei Register** miteinander zu verrechnen¹⁶, ist es möglich **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** 'c' als **atomar** zu definieren, da sie mit einem **Maschinenbefehl** verarbeitet werden können¹⁷. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das SP-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** 'c' als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verrechnet werden können.

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

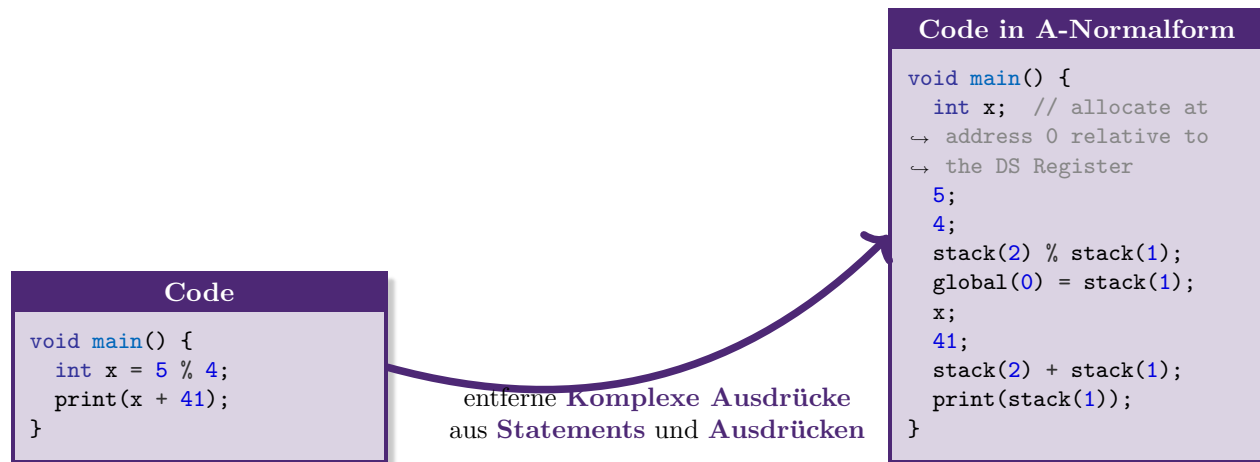


Abbildung 2.9: Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen

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

Alle weiteren denkbaren **Passes** sind zu **spezifisch** auf bestimmte **Statements** und **Ausdrücke** ausgelegt, als das sich zu diesen allgemein etwas mit einer **Theorie** dahinter sagen lässt. Alle **Passes**, die zur Implementierung des **PicoC-Compilers** geplant und ausgedacht wurden sind im Unterkapitel 3.3.1 definiert.

2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakter Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt¹⁸. Der Weg von **Abstrakter Syntax** zu **Konkreter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax**

¹⁶Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.

¹⁷Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. **MULT ACC IN2**.

¹⁸Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben.

zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden musste.

Jeder **Knoten** des **Abstrakter Syntaxbaums** erhält dazu eine Methode, welche hier `to_string` genannt wird, die eine **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **Semikolons** ; usw. ausgibt. Dabei wird nach dem **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die immer wiederum die Methode `to_string` ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgeben**.

2.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**¹⁹ eintritt:

1. der **Parser**²⁰ entscheidet das **Wortproblem** für ein **Eingabeprogramm**²¹ mit 0, also das **Eingabeprogramm** befolgt nicht die **Syntax** der **Sprache** des Compilers²².
2. in den **Passes** tritt eine Fall ein, der **nicht** in der **Semantik** der Sprache des Compilers abgedeckt ist, z.B.:
 - eine **Variable** wird **verwendet**, obwohl sie noch **nicht deklariert** ist.
 - bei einem **Funktionsaufruf** werden **mehr** Argumente oder Argumente des **falschen Datentyps** übergeben, als in der **Funktionsdeklaration** oder **Funktionsdefinition** angegeben ist.
3. Während der **Laufzeit** des Compilers tritt ein Ereignis ein, das **nicht** durch die **Semantik** der Sprache des Compilers abgedeckt ist oder das **Betriebssystem** nicht erlaubt, z.B.:
 - eine **nicht erlaubte Operation**, wie **Division durch 0** (z.B. $42 / 0$) soll ausgeführt werden.
 - **Segmentation Fault**: Wenn auf **Speicher zugegriffen** wird, der vom **Betriebssystem geschützt** ist.

oder während des des **Linkens** (Definition 5.4) etwas nicht zusammenpasst, wie z.B.:

- es gibt **keine** oder **mehr als eine** `main`-Funktion
- eine Funktion, die in einer **Objektdatei** (Definition 5.3) benötigt wird, wird von **keiner** anderen oder **mehr als einer** Objektdatei bereitgestellt

wird eine **Fehlermeldung** (Definition 2.51) ausgegeben.

Definition 2.51: Fehlermeldung

*Benachrichtigung beliebiger Form, die einen **Grund** angibt weshalb ein Programm **nicht weiter ausgeführt** werden kann^a. Das **Ausgeben** einer Fehlermeldung kann dabei auf **verschiedene Weisen** erfolgen, wie z.B.*

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

¹⁹Errors in C/C++ - GeeksforGeeks.

²⁰Bzw. der **Recognizer** im Parser.

²¹Bzw. **Wort**.

²²Bzw. das **Eingabeprogramm** lässt sich **nicht** mit der Grammatik des Compilers **ableiten**.

- über eine *Dialogbox* in einer *Graphischen Benutzerfläche*^c oder *Zeichenorientierten Benutzerschnittstelle*^d.
- in ein *Register* oder an eine *spezielle Adresse* des *Hauptspeichers* wird ein *Wert* *geschrieben*.
- *Logdatei*^e auf einem *Speichermedium*.

^aDieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler** selbst **kompiliert** hat.

^bNur unter **Linux**, **Windows** hat sowas nicht.

^cIn engl. **G**raphical **U**ser **I**nterface, kurz **GUI**.

^dIn engl. **T**ext-based **U**ser **I**nterface, kurz **TUI**.

^eIn engl. **log file**.

3 Implementierung

In diesem Kapitel wird, nachdem im Kapitel 2 die nötigen **theoretischen Grundlagen** des **Compilerbau** vermittelt wurden, nun auf die **Implementierung** des **PicoC-Compilers** eingegangen. Aufgeteilt in die selben Kategorien **Lexikalische Analyse 3.1**, **Syntaktische Analyse 3.2** und **Code Generierung 3.3**, wie in Kapitel 2, werden in den folgenden Unterkapiteln die einzelnen **Zwischenschritte** vom einem **Programm** in der **Konkreten Syntax** der Sprache L_{PicoC} hin zum einem Programm mit derselben **Semantik** in der **Konkreten Syntax** der Sprache L_{RETI} erklärt.

Für das Parsen¹ des Programmes in der **Konkreten Syntax** der Sprache L_{PicoC} wird das **Lark Parsing Toolkit**^{2 3} verwendet. Das **Lark Parsing Toolkit** ist eine Bibliothek, die es ermöglicht mittels eines in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** (Definition 3.3 bzw. für den Dialekt von Lark Definition 3.4) spezifizierten Grammatik der **Konkreten Syntax** ein Programm in ebendieser **Konkreten Syntax** zu parsen und daraus einen **Ableitungsbaum** für die kompilierintere Weiterverarbeitung zu generieren.

Definition 3.1: Metasyntax

*Steht für den **Aufbau** einer **Metasprache** (Definition 3.2), der durch eine **Grammatik** oder **Natürliche Sprache** beschrieben werden kann.*

Definition 3.2: Metasprache

*Eine Metasprache ist eine **Sprache**, die dazu genutzt wird **andere Sprachen** zu **beschreiben**^a.*

^aDas „Meta“ drückt allgemein aus, dass sich etwas auf einer **höheren Ebene** befindet. Um über die Ebene sprechen zu können, in der man sich **selbst befindet**, muss man von einer **höheren, außenstehenden Ebene** darüber reden.

Definition 3.3: Erweiterte Backus-Naur-Form (EBNF)

*Die Erweiterte Backus-Naur-Form^a ist eine **Metasyntax** (Definition 3.1) die dazu verwendet wird **Kontextfreier Grammatiken** darzustellen.^{b c}*

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

^aDer Name kommt daher, dass es eine **Erweiterung** der **Backus-Naur-Form** ist, die hier allerdings **nicht** weiter erläutert wird.

^bNebel, „Theoretische Informatik“.

^cGrammar Reference — Lark documentation.

^d<https://standards.iso.org/ittf/PubliclyAvailableStandards/>.

¹Wobei beim **Parsen** auch das **Lexen** inbegriffen ist.

²Lark - a parsing toolkit for Python.

³Shinan, lark.

Definition 3.4: Dialekt der EBNF aus Lark

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

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

^a<https://lark-parser.readthedocs.io/en/latest/grammar.html>.

^bBzw. kann der *-Quantor auch **keinmal** wiederholen bedeuten.

Das **Lark Parsing Toolkit** wurde vor allem deswegen gewählt, weil es sehr **einfach in der Verwendung** ist. Andere derartige Tools, wie z.B. **ANTLR⁴** sind **Parser Generatoren**, die zur **Konkreten Syntax** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkreten Syntax** direkt ein Programm in dieser Konkreten Syntax **parsen** und einen **Ableitungsbaum** dafür generieren zu können.

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

Neben den **Konkreten Syntaxen⁵**, die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterter Back-Naur-Form** spezifiziert sind, werden in den folgenden Unterkapiteln die **Abstrakte Syntaxen**, welche spezifizieren, welche Kompositionen für die **Abstrakter Syntaxbaums** der verschiedenen **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben, die allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit** hat.

Die Notation für die **Abstrakte Syntax** unterscheidet sich bewusst von der **Erweiterten Backus-Naur-Form**, da in der Abstrakten Syntax **Kompositionen von Knoten** beschrieben werden, die **klar auszumachen** sind, wodurch es die Grammatik nur **unnötig verkomplizieren** würde, wenn man die **Erweiterte Backus-Naur-Form** verwenden würde. Es gibt leider **keine** Standardnotation für die **Abstrakte Syntax**, die sich deutlich durchgesetzt hat, daher wird für die Abstrakte Syntaxen eine eigene **Abstract Syntax Form Notation** (Definition 3.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrete** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 3.5: Abstrakte Syntax Form (ASF)

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

Letztendlich geht es allerdings nur darum, dass aufgrund der Verwendung des **Lark Parsing Toolkit** die **Konkrete Syntax** in einem eigenen Dialekt der **Erweiterter Backus-Naur-Form** angegeben sein muss

⁴**ANTLR**.

⁵Der **Plural** von Syntax ist **Syntaxen**, wie es in Quelle *Syntax* verifiziert werden kann.

und für das Implementieren der Passes die **Abstrakte Syntax** für den **Programmierer** möglichst **einfach verständlich** sein sollte, weshalb sich die **Abstrakte Syntax Form** gut dafür eignet.

3.1 Lexikalische Analyse

Für die **Lexikalische Analyse** ist es nur notwendig eine Grammatik zu definieren, die den Teil der **Konkreten Syntax** beschreibt, der die **verschiedenen Pattern** für die verschiedenen Token der Sprache L_{PicoC} beschreibt, also den Teil der für die **Lexikalische Analyse** wichtig ist. Diese Grammatik wird dann vom **Lark Parsing Toolkit** dazu verwendet ein Programm in **Konkreter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel 2.3 erläutert ist.

3.1.1 Konkrete Syntax für die Lexikalische Analyse

In der Grammatik 3.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokennamen** oder einen **Teil der Beschreibung** eines **Tokennamen**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol NUM um einen **Tokennamen**, der durch die **Produktion** `NUM ::= "0" | DIG.NO.0 DIG.WITH.0*` beschrieben wird und beschreibt, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann. Das ist daran festzumachen, dass das Nicht-Terminalsymbol NUM in keiner anderen Produktion vorkommt, die auf der **linken Seite** des „**kann abgeleitet werden zu**“-Symbols `::=` ebenfalls ein **großgeschriebenen** Nicht-Terminalsymbol hat. Dagegen dient das **großgeschriebene** Nicht-Terminalsymbol DIG.NO.0 aus der Produktion `NUM ::= "0" | DIG.NO.0 DIG.WITH.0*` nur zu Beschreibung von NUM.

Die in der Grammatik 3.1.1 definierten **Nicht-Terminalsymbole** können in der Grammatik 3.2.8 der **Konkreten Syntax** für die **Syntaktischen Analyse** verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl NUM stehen darf.

Die in der Konkrete Syntax vereinzelt **kleingeschriebenen** Nicht-Terminalsymbole, wie `name` haben nur den Zweck mehrere **Tokennamen**, wie `NAME | INT_NAME | CHAR_NAME` unter einem Überbegriff zu sammeln.

In Lark steht eine Zahl `.ZAHL`, die an ein **Nicht-Terminalsymbol** angehängt ist, dass auf der linken Seite des „**kann abgeleitet werden zu**“-Symbols `::=` einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. Es gibt den Fall, dass ein Wort von mehreren Produktionen erkannt wird, z.B. wird das Wort `int` sowohl von der Produktion `NAME`, als auch von der Produktion `INT.DT` erkannt. Daher ist es notwendig für `INT.DT` eine **Priorität** `INT.DT.2` zu setzen⁶, damit das Wort `int` den **Tokennamen** `INT.DT` zugewiesen bekommt und nicht `NAME`.

Allerdings muss für den Fall, dass `int` der **Präfix** eines Wortes ist, z.B. `int_var` noch die Produktion `INT.NAME.3` definiert werden, da der im **Lark Parsing Toolkit** verwendete **Basic Lexer** sobald ein Wort von einer Produktion erkannt wird, diesem direkt einen Tokennamen zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. In diesem Fall würden aus `int_var` die Token `Token('INT_DT', 'int')`, `Token('NAME', '_var')` generiert, anstatt `Token(NAME, 'int_var')`. Daher muss die Produktion `INT.NAME.3` eingeführt werden, die immer zuerst geprüft wird. Wenn es sich nur um das Wort `int` handelt, wird zuerst die Produktion `INT.NAME.3` geprüft, es stellt sich heraus, dass `int` von der Produktion `INT.NAME.3` nicht erkannt wird, daher wird als nächstes `INT.DT.2` geprüft, welches `int` erkennt.

Die Implementierung des **Basic Lexer** aus dem **Lark Parsing Toolkit** ist unter [Link⁷](#) zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten und ist aufgrund dessen, dass sie in der Lage ist nach einer spezifizierten Grammatik zu lexen, **zu komplex**, um sie an dieser Stelle allgemein

⁶Es wird immer die **höchste** Priorität **zuerst** genommen.

⁷<https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py>.

erklären zu können.

Der **Basic Lexer** verhält sich allerdings grundlegend so, wie es im Unterkapitel 2.3 erklärt wurde, allerdings berücksichtigt der **Basic Lexer** ebenfalls **Prioritäten**, sodass für den aktuellen Index im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Wort** an dem aktuellen Index im Eingabeprogramm erkennt, bekommt es direkt den **Tokenwert** dieser Produktion zugewiesen. Weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

<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 DIG_WITH_0*</i>	
<i>ASCII_CHAR</i>	::=	"_".~"	
<i>CHAR</i>	::=	"'" <i>ASCII_CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII_CHAR</i> + ". <i>picoc</i> "	
<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>LOGIC_NOT</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>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>	

Grammatik 3.1.1: Grammatik der Konkreten Syntax der Sprache L_{Picoc} für die Lexikalische Analyse in EBNF

3.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 3.1 dazu verwendet die Konstruktion eines **Abstrakter Syntaxbaums** in seinen einzelnen **Zwischenschritten** zu erläutern.

```

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

```

Code 3.1: PicoC-Code des Codebeispiels

Die vom **Basic Lexer** des **Lark Parsing Toolkit** erkannten **Token** sind Code 3.2 zu sehen.

```

1 [Token('FILENAME', './verbose_dt_simple_ast_gen_array_decl_and_alloc.picoc'), Token('STRUCT',
  ↳ 'struct'), Token('NAME', 'st'), Token('LBRACE', '{'), Token('INT_DT', 'int'),
  ↳ Token('MUL_DEREF_PNTR', '*'), Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'),
  ↳ Token('NAME', 'attr'), Token('RPAR', ')'), Token('LSQB', '['), Token('NUM', '4'),
  ↳ Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '5'), Token('RSQB', ']'),
  ↳ Token('SEMICOLON', ';'), Token('RBRACE', '}'), Token('SEMICOLON', ';'), Token('VOID_DT',
  ↳ 'void'), Token('NAME', 'main'), Token('LPAR', '('), Token('RPAR', ')'), Token('LBRACE',
  ↳ '{'), Token('STRUCT', 'struct'), Token('NAME', 'st'), Token('MUL_DEREF_PNTR', '*'),
  ↳ Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'), Token('NAME', 'var'), Token('LSQB',
  ↳ '['), Token('NUM', '3'), Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '2'),
  ↳ Token('RSQB', ']'), Token('RPAR', ')'), Token('SEMICOLON', ';'), Token('RBRACE', '}')]

```

Code 3.2: Tokens für das Codebeispiel

3.2 Syntaktische Analyse

In der **Syntaktischen Analyse** ist es die Aufgabe des **Parsers** aus einem Programm in **Konkreter Syntax** unter Verwendung der **Tokens** aus der **Lexikalischen Analyse** einen **Ableitungsbaum** zu generieren. Es ist danach die Aufgabe möglicher **Visitors** und die Aufgabe des **Transformers** aus diesem **Ableitungsbaum** einen **Abstrakter Syntaxbaum** in **Abstrakter Syntax** zu generieren.

3.2.1 Umsetzung von Präzidenz und Assoziativität

Die Programmiersprache L_{PicoC} hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache L_C ⁸. Die **Präzidenzregeln** der Programmiersprache L_{PicoC} sind in Tabelle 3.1 aufgelistet.

⁸C Operator Precedence - cppreference.com.

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

Tabelle 3.1: Präzidenzregeln von PicoC

Würde man diese **Operatoren** ohne Beachtung von **Präzidenzregeln** (Definiton 2.28) und **Assoziativität** (Definition 2.27) in eine Grammatik verarbeiten wollen, so könnte eine Grammatik, wie Grammatik 3.2.1 dabei rauskommen.

<i>prim_exp</i>	::=	<i>name</i> <i>NUM</i> <i>CHAR</i> "(" <i>exp</i> ")"	<i>L_Arith</i> +
<i>un_op</i>	::=	"-" "~" "!" "*" "&"	
<i>un_exp</i>	::=	<i>un_op exp</i>	
<i>bin_op</i>	::=	"*" "/" "%" "+" "-" "&" "^" " " "<" "<=" ">" ">=" "!=" "==" "&&" " "	
<i>bin_exp</i>	::=	<i>exp bin_op exp</i>	
<i>exp</i>	::=	<i>prim_exp</i> <i>un_exp</i> <i>bin_exp</i>	

Grammatik 3.2.1: Undurchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF

Die Grammatik 3.2.1 ist allerdings **mehrdeutig**, d.h. verschiedene **Linksableitungen** in der Grammatik können zum selben **Wort** abgeleitet werden. Z.B. kann das Wort $3 * 1 \&\& 4$ sowohl über die **Linksableitung** 3.5.1 als auch über die **Linksableitung** 3.5.2 abgeleitet werden.

$$\begin{aligned}
 \text{exp} &\Rightarrow \text{bin_exp} \Rightarrow \text{exp bin_op exp} \Rightarrow \text{bin_exp bin_op exp} \\
 &\Rightarrow \text{exp bin_op exp bin_op exp} \Rightarrow^* 3 * 1 \&\& 4
 \end{aligned}
 \tag{3.5.1}$$

$$\begin{aligned}
 \text{exp} &\Rightarrow \text{bin_exp} \Rightarrow \text{exp bin_op exp} \Rightarrow \text{prim_exp bin_op exp} \Rightarrow \text{NUM bin_op exp} \\
 &\Rightarrow 3 \text{ bin_op exp} \Rightarrow 3 * \text{exp} \Rightarrow 3 * \text{bin_exp} \Rightarrow 3 * \text{exp bin_exp} \Rightarrow^* 3 * 1 \&\& 4
 \end{aligned}
 \tag{3.5.2}$$

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

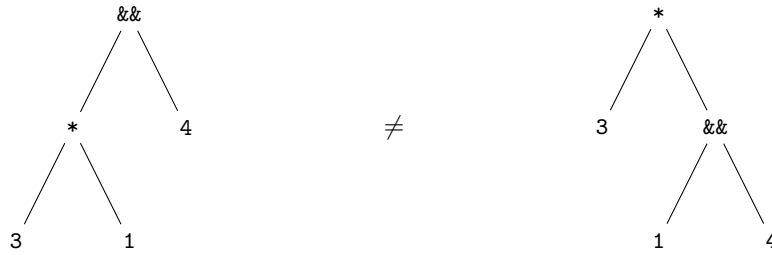


Abbildung 3.1: Ableitungsbäume zu den beiden Ableitungen

Der **linke Baum** entspricht Ableitung 3.5.1 und der **rechte Baum** entspricht Ableitung 3.5.2. Würde man in den Ausdrücken, die von diesen Bäumen dargestellt sind in **Klammern** setzen, um die **Präzidenz** sichtbar zu machen, so würde Ableitung 3.5.1 die Klammerung $(3 * 1) \&\& 4$ haben und die Ableitung 3.5.2 die Klammerung $3 * (1 \&\& 4)$ haben.

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

<code>prim.exp</code>	::=	...	<code>L_Arith + L_Array</code>
<code>post.exp</code>	::=	...	<code>+ L_Pntr + L_Struct</code>
<code>un.exp</code>	::=	...	<code>+ L_Fun</code>
<code>arith.prec1</code>	::=	...	
<code>arith.prec2</code>	::=	...	
<code>arith.and</code>	::=	...	
<code>arith.oplus</code>	::=	...	
<code>arith.or</code>	::=	...	
<code>rel.exp</code>	::=	...	<code>L_Logic</code>
<code>eq.exp</code>	::=	...	
<code>logic.and</code>	::=	...	
<code>logic.or</code>	::=	...	
<code>assign.stmt</code>	::=	...	<code>L_Assign</code>

Grammatik 3.2.2: Durchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF

Einigen **Bezeichnungen** der **Produktionen** sind in Tabelle 3.2 ihren jeweiligen **Operatoren** zugeordnet für welche sie zuständig sind.

Bezeichnung der Produktionsregel	Operatoren
post_exp	a() a[] a.b
un_exp	-a !a ~a *a &a
arith_prec1	a*b a/b a%b
arith_prec2	a+b a-b
arith_and	a<b a<=b a>b a>=b
arith_oplus	a==b a!=b
arith_or	a&b
rel_exp	a^b
eq_exp	a b
logic_and	a&&b
logic_or	a b
assign	a=b

Tabelle 3.2: Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren

Als nächstes müssen die einzelnen **Produktionen** entsprechend der **Ausdrücke** für die sie zuständig sind definiert werden. Jede der **Produktionen** soll nur Ausdrücke **erkennen** können, deren **Präzidenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzidenzstufe **höher** ist. Z.B. soll **un_op** sowohl den Ausdruck $-(3 * 14)$ als auch einfach nur $(3 * 14)$ ⁹ erkennen können, aber nicht $3 * 14$ ohne Klammern, da dieser Ausdruck eine **geringe Präzidenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

Bei z.B. der Produktion **un_exp** in 3.2.3 für die **rechtsassoziativen unären Operatoren** $-a$, $!a$, $\sim a$, $*a$ und $\&a$ ist die **Alternative** **un_op un_exp** dafür zuständig, dass diese unären Operatoren **rechtsassoziativ** geschachtelt werden können (z.B. $! \sim 42$). Die Alternative **post_exp** ist dafür zuständig, dass die Produktion auch **terminieren** kann und es auch möglich ist ausschließlich einen Ausdruck **höherer Präzidenz** (z.B. 42) zu haben.

$$un_exp ::= un_op\ un_exp \mid post_exp$$

Grammatik 3.2.3: Beispiel für eine unäre rechtsassoziative Produktion

Bei z.B. der Produktion **post_exp** in 3.2.4 für die **linksassoziativen unären Operatoren** $a()$, $a[]$ und $a.b$ sind die Alternativen **post_exp**["logic_or"] und **post_exp**."name" dafür zuständig, dass diese unären Operatoren **linksassoziativ** geschachtelt werden können (z.B. $ar[3][1].car[4]$). Die Alternative **name**("fun_args") ist für einen **einzelnen Funktionsaufruf** zuständig. Die Alternative **prim_exp** ist dafür zuständig, dass die Produktion nicht nur bei **name**("fun_args") **terminieren** kann und es auch möglich ist ausschließlich einen Ausdruck der **höchsten Präzidenz** (z.B. 42) zu haben.

$$post_exp ::= post_exp\ ["logic_or"] \mid post_exp\ ."name" \mid name\ ("fun_args") \mid prim_exp$$

Grammatik 3.2.4: Beispiel für eine unäre linksassoziative Produktion

Bei z.B. der Produktion **prec2_exp** in 3.2.5 für die **binären linksassoziativen Operatoren** $a+b$ und $a-b$ ist die **Alternative** **arith_prec2 prec2_op arith_prec1** dafür zuständig, dass **mehrere** Operationen der Präzidenzstufe 4 in Folge erkannt werden können¹⁰ (z.B. $3 + 1 - 4$, wobei $-$ und $+$ beide Präzidenzstufe 4

⁹Geklammerte Ausdrücke werden nämlich von **prim_exp** erkannt, welches eine höhere **Präzidenzstufe** hat.

¹⁰Bezogen auf Tabelle 3.1.

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

$$\text{arith_prec2} ::= \text{arith_prec2 } \text{prec2_op } \text{arith_prec1} \mid \text{arith_prec1}$$

Grammatik 3.2.5: *Beispiel für eine linksassoziative Produktion*

Manche **Parser**^a haben allerdings ein Problem mit **Linksrekursion** (Definition 2.24), wie sie z.B. in der Produktion 3.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 3.2.5 zur Produktion 3.2.6 umschreibt.

$$\text{arith_prec2} ::= \text{arith_prec1 } (\text{prec2_op } \text{arith_prec1})^*$$

Grammatik 3.2.6: *Beispiel für eine linksassoziative Produktion*

Die von Produktion 3.2.6 erkannten Ausdrücke sind dieselben, wie für die Produktion 3.2.5, allerdings ist die Produktion 3.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 3.3) definierten $*$ -Operator, um mehrere Operationen der Präzidenzstufe 4 in Folge erkennen zu können (z.B. $3 + 1 - 4$, wobei $-$ und $+$ beide Präzidenzstufe 4 haben).

Das **Nicht-Terminalsymbol** `arith_prec1` erlaubt es, dass **zwischen** der Folge von Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B. $3 + 1 / 4 - 1$, wobei $-$ und $+$ beide Präzidenzstufe 4 haben und $/$ Präzidenzstufe 3). Da der in der EBNF definierte $*$ -Operator auch bedeutet, dass das Teilpattern auf das er sich bezieht **kein einziges mal** vorkommen kann, ist es mit dem **linken Nicht-Terminalsymbol** `arith_prec1` möglich, dass ausschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B. $1 / 4$).

^aDarunter zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird **nicht**.

Alle **Operatoren** der Sprache L_{PicoC} sind also entweder **binär** und **linksassoziativ** (z.B. $a*b$, $a-b$, $a>b$ oder $a\&\&b$), **unär** und **rechtsassoziativ** (z.B. $\&a$ oder $!a$) oder **unär** und **linksassoziativ** (z.B. $a[]$ oder $a()$). Somit ergibt sich die Grammatik 3.2.7.

<i>prec1_op</i>	::=	"*" "/" "%"	<i>L_Misc</i>
<i>prec2_op</i>	::=	"+" "-"	
<i>rel_op</i>	::=	"<" "<=" ">" ">="	
<i>eq_op</i>	::=	"==" "!="	
<i>fun_args</i>	::=	[<i>logic_or</i> ("," <i>logic_or</i>)*]	
<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>post_exp</i> [" <i>logic_or</i> "] <i>post_exp</i> ." <i>name</i> " <i>name</i> (" <i>fun_args</i> ")"	+ <i>L_Array</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>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i> <i>un_exp</i>	+ <i>L_Fun</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>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	<i>L_Assign</i>

Grammatik 3.2.7: Durchdachte Konkrete Syntax für Operatorpräzedenz in EBNF

3.2.2 Konkrete Syntax für die Syntaktische Analyse

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

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrete Syntax** an die **aktuellste konsistentlos auffindbare Version** der echten Grammatik *ANSI C grammar (Yacc)* der Sprache L_C angepasst¹¹, damit es sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C , wobei z.B. die Compiler **GCC**¹² und **Clang**¹³ zu nennen wären.

In der Grammatik 3.2.8, welche die **Konkrete Syntax** der Sprache L_{PicoC} für die **Syntaktische Analyse** beschreibt, werden einige der **Tokennamen** aus der Grammatik 3.1.1 der **Konkreten Syntax** für die **Lexikalischen Analyse** verwendet, wie z.B. *NUM* aber auch *name*, welches eine Produktion ist, die mehrere **Tokennamen** unter einem Überbegriff zusammenfasst.

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

¹¹An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts verändert, wie die Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Lex)* und **Lexikalische Analyse** *noauthor'ansi'nodate-2* aus dem Jahre 1985 zeigen.

¹²*GCC, the GNU Compiler Collection - GNU Project.*

¹³*clang: C++ Compiler.*

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

Grammatik 3.2.8: Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1

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

Grammatik 3.2.9: Grammatik der Konkretten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2

In der Grammatik 3.2.8 sind alle **Grammatiksymbole** ausgegraut, die das **Bachelorprojekt** betreffen. Alle nicht ausgegrauten **Grammatiksymbole** wurden für die Implementierung der **neuen Funktionalitäten**, welche die **Bachelorarbeit** betreffen hinzugefügt.

3.2.3 Ableitungsbaum Generierung

Die in Unterkapitel 3.2.2 definierte **Konkrete Syntax**, die von der Grammatik 3.2.8 beschrieben wird lässt sich mithilfe des **Earley Parsers** (Definition 3.6) von Lark dazu verwenden Code, der in der Sprache L_{PicoC} geschrieben ist zu parsen um einen **Ableitungsbaum** zu generieren.

Definition 3.6: Earley Parser

Ist ein Algorithmus für das **Parsen** von **Wörtern** einer **Kontextfreien Sprache**, der ein **Chart Parser** ist, welcher einen mittels **Dynamischer Programmierung** und dem **Top-Down Ansatz** arbeitenden **Earley Recognizer** (Definition 3.7) nutzt, um einen **Ableitungsbaum** zu konstruieren.

Zur **Konstruktion des Ableitungsbaumes** muss dafür gesorgt werden, dass der **Earley Recognizer** bei der **Vervollständigungsoperation** Zeiger auf den **vorherigen Zustand** hinzugefügt, um durch **Rückwärtsverfolgen** dieser Zeiger die **Ableitung** wieder **nachvollziehen** zu können und so einen **Ableitungsbaum** konstruieren zu können.^a

^aJay Earley, „An efficient context-free parsing“.

Definition 3.7: Earley Recognizer

Ist ein *Recognizer*, der für *alle Kontextfreien Sprachen* das *Wortproblem* entscheiden kann und dies mittels *Dynamischer Programmierung* mit dem *Top-Down Ansatz* umsetzt.^a

Eingabe und *Ausgabe* des Algorithmus sind:

- **Eingabe:** *Eingabewort* w und **Grammatik** $G_{Parse} = \langle N, \Sigma, P, S \rangle$
- **Ausgabe:** 0 wenn $w \notin L(G_{Parse})$ ^b und 1 wenn $w \in L(G_{Parse})$

Bevor dieser *Algorithmus* erklärt wird müssen noch einige *Symbole* und *Notationen* erklärt werden:

- α, β, γ stellen eine *beliebige Folge* von *Grammatiksymbolen*^c dar
- A und B stellen *Nicht-Terminalsymbole* dar
- a stellt ein *Terminalsymbol* dar
- **Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine *Produktion*, in der α *bereits geparkt* wurde und β *noch geparkt* werden muss
- Die *Indexierung* ist informell ausgedrückt so umgesetzt, dass die *Indices zwischen Tokennamen* liegen, also *Index 0* *vor* dem ersten *Tokennamen* verortet ist, *Index 1* *nach* dem ersten *Tokennamen* verortet ist und *Index n* *nach* dem *letzten Tokennamen* verortet ist

und davor müssen noch einige *Begriffe definiert* werden:

- **Zustandsmenge:** Für jeden der $n + 1$ *Indices* j wird eine *Zustandsmenge* $Z(j)$ generiert
- **Zustand einer Zustandsmenge:** Ist ein *Tupel* $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die *aktuelle Produktion* ist, die bis *Punkt* \bullet geparkt wurde und i der *Index* ist, ab welchem der Versuch der Erkennung eines *Teilworts* des *Eingabeworts* mithilfe dieser *Produktion* begann

Der *Ablauf* des Algorithmus ist wie folgt:

1. *initialisiere* $Z(0)$ mit der *Produktion*, welches das *Startsymbol* S auf der *linken Seite* des „*kann abgeleitet werden zu*“-*Symbols* $::=$ hat
2. es werden in der *aktuellen Zustandsmenge* $Z(j)$ die folgenden *Operationen ausgeführt*:
 - **Voraussage:** Für jeden *Zustand* in der *Zustandsmenge* $Z(j)$, der die Form $(A ::= \alpha \bullet B\gamma, i)$ hat, wird für jede *Produktion* $(B ::= \beta)$ in der *Grammatik*, die ein B auf der *linken Seite* des „*kann abgeleitet werden zu*“-*Symbols* $::=$ hat ein *Zustand* $(B ::= \bullet\beta, j)$ zur *Zustandsmenge* $Z(j)$ hinzugefügt
 - **Überprüfung:** Für jeden *Zustand* in der *Zustandsmenge* $Z(j)$, der die Form $(A ::= \alpha \bullet a\gamma, i)$ hat wird der *Zustand* $(A ::= \alpha a \bullet \gamma, i)$ zur *Zustandsmenge* $Z(j + 1)$ hinzugefügt
 - **Vervollständigung:** Für jeden *Zustand* in der *Zustandsmenge* $Z(j)$, der die Form $(B ::= \beta \bullet, i)$ hat werden alle *Zustände* in $Z(i)$ gesucht, welche die Form $(A ::= \alpha \bullet B\gamma, i)$ haben und es wird der *Zustand* $(A ::= \alpha B \bullet \gamma, i)$ zur *Zustandsmenge* $Z(j)$ hinzugefügt

bis:

- der *Zustand* $(A ::= \beta \bullet, 0)$ in der *Zustandsmenge* $Z(n)$ auftaucht, wobei A das *Startsym-*

bol S ist $\Rightarrow w \in L(G_{Parse})$

- *keine Zustände* mehr hinzugefügt werden können $\Rightarrow w \notin L(G_{Parse})$

^aJay Earley, „An efficient context-free parsing“.

^b $L(G_{Parse})$ ist die **Sprache**, welche durch die **Grammatik** G_{Parse} beschrieben wird.

^cAlso eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

3.2.3.1 Codebeispiel

Der **Ableitungsbaum**, der mithilfe des **Earley Parsers** und der **Token** der **Lexikalischen Analyse** aus dem Beispiel in Code 3.1 generiert wurde, ist in Code 3.3 zu sehen. Im Code 3.3 wurden einige Zeilen **markiert**, die später in Unterkapitel 3.2.4.1 zum Vergleich wichtig sind.

```

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

```

```

42         name      var
43         array_dims
44         3
45         2
46         array_dims

```

Code 3.3: *Ableitungsbaum nach Ableitungsbaum Generierung*

3.2.3.2 Ausgabe des Ableitungsbaums

Die Ausgabe des **Ableitungsbaums** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Grammatik den **linken Seiten** des „kann abgeleitet werden zu“-Symbols $::=$ ¹⁴ entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition 2.36 eines **Ableitungsbaums** auch schon definiert ist. Die **EBNF-Grammatik 3.2.8** des **PicoC-Compilers** erlaubt es allerdings auch, dass in einem **Blatt** garnichts ε steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("NUM")*` gibt, in denen auch das **leere Wort** ε abgeleitet werden kann.

Die Ausgabe des **Abstrakter Syntaxbaum** ist bewusst so gewählt, dass sie sich optisch vom **Ableitungsbaum** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase** geschrieben sind, im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des „kann abgeleitet werden zu“-Symbols $::=$.

3.2.4 Ableitungsbaum Vereinfachung

Der **Ableitungsbaum** in Code 3.3, dessen Generierung in Unterkapitel 3.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Transformers** ein **Abstrakter Syntaxbaum** generiert werden kann. Das Problem ist, dass um den **Datentyp** einer Variable in der Programmiersprache L_C und somit auch die Programmiersprache L_{PicoC} korrekt bestimmen zu können, wie z.B. ein „**Array der Mächtigkeit 3 von Pointern auf Arrays der Mächtigkeit 2 von Integern**“ `int (*ar[3])[2]` die **Spiralregel**¹⁵ in der Implementierung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Grammatik 3.2.8 der **Konkreten Syntax** auf eine spezielle Weise passend spezifiziert.

Was man erhalten will, ist ein **entarteter Baum** von **PicoC-Knoten**, an dem man den **Datentyp** direkt ablesen kann, indem man sich einfach über den **entarteten Baum** bewegt, wie z.B. `PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], PntrDecl(Num('1'), StructSpec(Name('st')))))` für den Ausdruck `struct st *(*var[3][2])`.

Es sind hierbei mehrere Probleme zu lösen. Hat man den Ausdruck `struct st *(*var[3][2])` wird dieser zu einem **Ableitungsbaum**, wie er in Abbildung 3.2 zu sehen ist.

¹⁴ *Grammar: The language of languages (BNF, EBNF, ABNF and more).*

¹⁵ *Clockwise/Spiral Rule.*

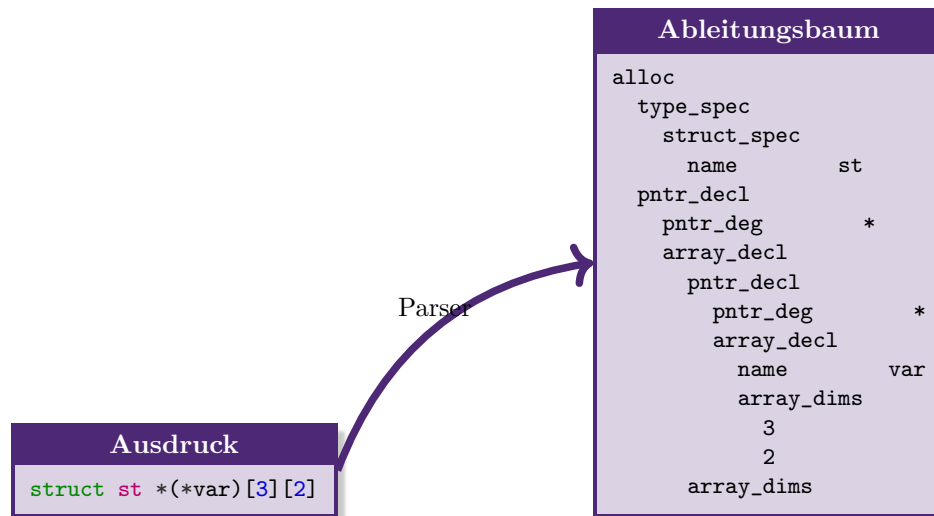


Abbildung 3.2: Ableitungsbaum nach Parsen eines Ausdrucks

Dieser **Ableitungsbaum** für den Ausdruck `struct st *(*var)[3][2]` hat allerdings einen Aufbau welcher durch die **Syntax** der **Pointerdeklaratoren** `pntr_decl(num, datatype)` und **Arraydeklaratoren** `array_decl(datatype, nums)` bestimmt ist, die **spiralähnlich** ist. Man würde allerdings gerne einen **entarteten Baum** erhalten, bei dem der Datentyp immer im **zweiten Attribut** weitergeht, anstatt abwechselnd im **zweiten** und **ersten**, wie beim **Pointerdeklarator** `pntr_decl(num, datatype)` und **Arraydeklarator** `array_decl(datatype, nums)`. Daher muss beim **ArrayDeclarator** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht werden.

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

In Abbildung 3.3 ist daher zu sehen, wie der **Ableitungsbaum** aus Abbildung 3.2 mithilfe eines **Visitors** (Definition 2.40) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des **Visitors** aus dem **Lark Parsing Toolkit** ist unter [Link¹⁶](https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py) zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Visitor** verhält sich allerdings grundlegend so, wie es in Definition 2.40 erklärt wurde.

¹⁶<https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py>.

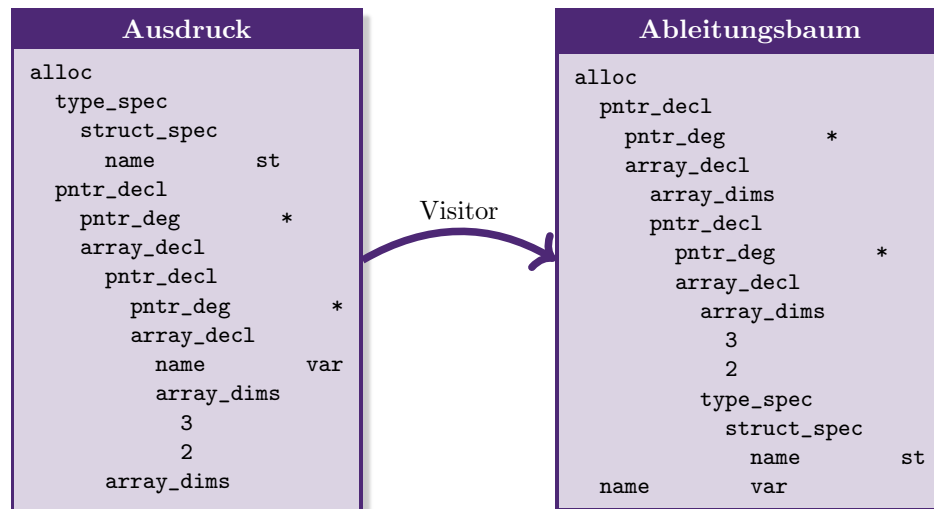


Abbildung 3.3: Ableitungsbaum nach Vereinfachung

3.2.4.1 Codebeispiel

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

```

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

```

```

30      decl_part
31      decl_exp_stmt
32      alloc
33      ptr_decl
34      ptr_deg      *
35      array_decl
36      array_dims
37      ptr_decl
38      ptr_deg      *
39      array_decl
40      array_dims
41      3
42      2
43      type_spec
44      struct_spec
45      name      st
46      name      var

```

Code 3.4: Ableitungsbaum nach Ableitungsbaum Vereinfachung

3.2.5 Abstrakt Syntax Tree Generierung

Nachdem der **Derivation Tree** in Unterkapitel 3.2.4 vereinfacht wurde, ist der **vereinfachte Ableitungsbaum** in Code 3.4 nun dazu geeignet, um mit einem **Transformer** (Definition 2.39) einen **Abstrakter Syntaxbaum** aus ihm zu generieren. Würde man den **vereinfachten Ableitungsbaum** des Ausdrucks `struct st *(*var[3][2])` auf passende Weise in einen **Abstrakter Syntaxbaum** umwandeln, so würde dabei ein **Abstrakter Syntaxbaum** wie in Abbildung 3.4 rauskommen.

Die Implementierung des **Transformers** aus dem **Lark Parsing Toolkit** ist unter [Link¹⁷](#) zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Transformer** verhält sich allerdings grundlegend so, wie es in Definition 2.39 erklärt wurde.

Den Teilbaum, der den Datentyp darstellt würde man von **oben-nach-unten¹⁸** als „**Pointer auf einen Pointer auf ein Array der Mächtigkeit 2, 3 von Structs des Typs st**“ lesen, also genau anders herum, als man den Ausdruck `struct st *(*var[3][2])` mit der **Spiralregel** lesen würde. Bei der **Spiralregel** fängt man beim Ausdruck `struct st *(*var[3][2])` bei der Variable `var` an und arbeitet sich dann auf „**Spiralbahnen**“, von **innen-nach-außen** durch den Ausdruck, um herauszufinden, dass dieser Datentyp ein „**Array der Mächtigkeit 3, 2 von Pointern auf einen Pointer auf einen Struct vom Typ st**“ ist.

¹⁷<https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py>.

¹⁸In der Informatik wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**, bzw. in diesem Beispiel von **links-nach-rechts**.

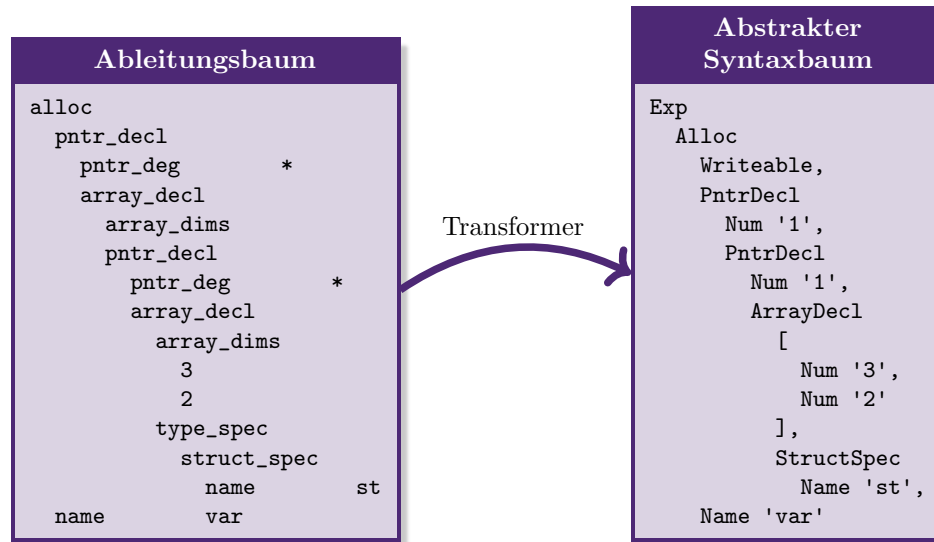


Abbildung 3.4: Abstrakter Syntaxbaum Generierung ohne Umdrehen

Dieser **Abstrakter Syntaxbaum** ist für die Weiterverarbeitung ungeeignet, denn für die **Adressberechnung** für eine Aneinandereiung von Zugriffen auf **Pointerelemente**, **Arrayelemente** oder **Structattribute**, welche in Unterkapitel 3.3.5.3 genauer erläutert wird, will man den Datentyp in **umgekehrter Reihenfolge**. Aus diesem Grund muss der **Transformer** bei der Konstruktion des **Abstrakter Syntaxbaum** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie in Abbildung 3.5 zustande, der für die Weiterverarbeitung geeignet ist.

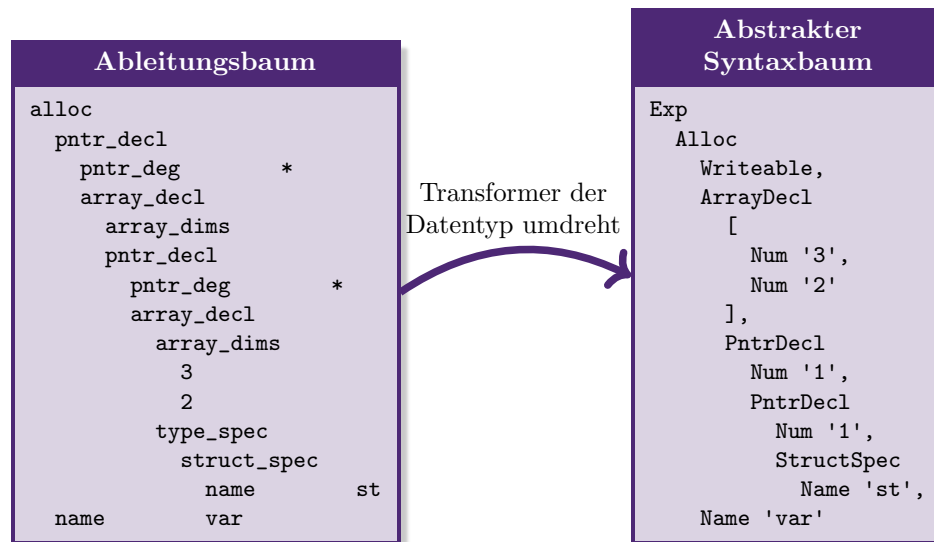


Abbildung 3.5: Abstrakter Syntaxbaum Generierung mit Umdrehen

Die Weiterverarbeitung des **Abstrakter Syntaxbaums** geschieht mithilfe von **Passes**, welche im Unterkapitel 2.5 genauer beschrieben werden. Da die Knoten des **Abstrakter Syntaxbaum** anders als beim **Ableitungsbaum** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Grammatik der **Kon-**

kretten Syntax ist es in den folgenden Unterkapiteln 3.2.5.1, 3.2.5.2 und 3.2.5.3 notwendig die **Bedeutung** der einzelnen **PicoC-Knoten**, **RETI-Knoten** und bestimmter **Kompositionen** dieser Knoten zu **dokumentieren**, die alle in den unterschiedlichen von den **Passes** umgeformten **Abstrakter Syntaxbaums** vorkommen.

Des Weiteren gibt die **Abstrakte Syntax** die durch die Grammatik 3.2.1 in Unterkapitel 3.2.5.4 beschrieben wird aufschluss darüber welche **Kompositionen von PicoC-Knoten**, neben den bereits in Tabelle 3.2.10 definierten Kompositionen mit Bedeutung insgesamt überhaupt **möglich** sind.

3.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache L_{PicoC} darstellen. Für die **PicoC-Knoten** wurden möglichst **kurze** und **leicht** verständliche Bezeichner gewählt, da auf diese Weise bei der Implementierung der einzelnen Passes möglichst **viel Code in eine Zeile** passt und dieser Code auch durch leicht verständliche Bezeichner von Knoten **intuitiv verständlich** sein sollte¹⁹. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 3.3 mit einem **Bschreibungstext** dokumentiert.

¹⁹Z.B. steht der **PicoC-Knoten** `Name(str)` für einen **Bezeichner**. Anstatt diesen Knoten in englisch `Identifizier(str)` zu nennen, wurde dieser als `Name(str)` gewählt, da `Name(str)` **kürzer** ist und **intuitiver verständlich**.

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 <code>x > 1</code> 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 3.3: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen beliebigen Ausdruck , dessen Ergebnis auf den Stack soll. Zudem besitzt er 2 versteckte Attribute, wobei datatype im RETI Blocks Pass wichtig ist und error_data für Fehlermeldungen wichtig ist.
Stack(num)	Container, der für das temporäre Ergebnis einer Berechnung, das num Speicherzellen relativ zum Stackpointer Register SP steht.
Stackframe(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Begin-Aktive-Funktion Register BAF steht.
Global(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Datensegment Register DS steht.
StackMalloc(num)	Container, der für das Allokieren von num Speicherzellen auf dem Stack steht.
PntrDecl(num, datatype)	Container, der für den 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 (Definition 2.47) 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-ANF 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 3.4: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initializer eines Structs , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines Attributezeichners , zu einem weiteren Initializer für eine 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-ANF Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Structdatentyps , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der Bezeichner des Structdatentyps ist und allocs eine Liste von Bezeichnern der Attribute des Structdatentyps mit dazugehörigem Datentyp , wofür sich der Container-Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.
If(exp, stmts)	Container für ein If Statement if(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts1, stmts2)	Container für ein If-Else Statement if(<exp>) { <stmts2> } else { <stmts2> } inklusive Condition exp und 2 Branches stmts1 und stmts2, die zwei Alternativen darstellen in denen jeweils Listen von Statements oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts)	Container für ein While-Statement while(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts)	Container für ein Do-While-Statement do { <stmts> } while(<exp>); inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen Funktionsaufruf : fun_name(exps), wobei name der Bezeichner der Funktion ist, die aufgerufen werden soll und exps eine Liste von Argumenten ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein Return-Statement : return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis vom Return-Statement zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine Funktionsdeklaration , z.B. <datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist und allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient.

Tabelle 3.5: PicoC-Knoten Teil 3

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts.blocks)	Container für eine Funktionsdefinition , z.B. <datatype> <fun.name>(<datatype> <param>) {<stmts>}, wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist, allocs die Parameter der Funktion sind, wobei der 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 und Speicherung des Werts des BAF-Registers der aufgerufenen Funktion und der Rücksprungadresse nacheinander an den Anfang des neuen Stackframes . Das Attribut fun_name steht dabei für den Bezeichner der Funktion, für die ein neuer Stackframe erstellt werden soll. Das Attribut fun_name dient später dazu den Block dieser Funktion zu finden, weil dieser für den weiteren Kompilierungsvorgang wichtige Information in seinen versteckten Attributen gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die Adresse des Befehls, der direkt auf die Jump Instruction folgt, ersetzt wird.
RemoveStackframe()	Container für das Entfernen des aktuellen Stackframes durch das Wiederherstellen des im noch aktuellen Stackframe gespeicherten Werts des BAF-Registers der aufgerufenen Funktion und das Setzen des SP-Registers auf den Wert des BAF-Registers vor der Wiederherstellung.
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 Bezeichner des Labels (Definition 3.8) 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 Kompilierungsvorgangs erstellt, der im RETI-Interpreter selbst später nicht sichtbar sein wird, aber in den Immediate-Dateien , welche die Abstrakter Syntaxbaums 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.

Definition 3.8: Label

Durch einen *Bezeichner eindeutig* zuordenbares *Sprungziel* im Programmcode.^a

^aThiemann, „Compilerbau“.

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 3.9) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 3.10) unbesetzt ist. Das **Attribut position** wird später für Fehlermeldungen gebraucht.

Definition 3.9: Token-Knoten

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

Token-Knoten entsprechen im Abstrakter Syntaxbaum *Blättern*.^a

^aThiemann, „Compilerbau“.

Definition 3.10: 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 Abstrakter Syntaxbaum *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

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache L_{RETI} darstellen. Für die **RETI-Knoten** wurden aus bereits in Unterkapitel 3.2.5.1 erläuterten Grund, genauso wie für die **RETI-Knoten** möglichst **kurze** und **leicht** verständliche Bezeichner gewählt. Alle **RETI-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 3.2.5.1 mit einem **Beschreibungstext** dokumentiert.

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 P. D. C. Scholl, „Betriebssysteme“

Tabelle 3.7: RETI-Knoten

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

In Tabelle 3.8 sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax 3.2.8** enthalten sind.

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

Tabelle 3.8: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 3.8 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch **Stack(Num('x'))**, $x \in \mathbb{N}$ ersetzt wurde.

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

3.2.5.4 Abstrakte Syntax

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

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>
<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(<exp>, <bin_op>, <exp>)</i> <i>UnOp(<un_op>, <exp>)</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i>	
<i>stmt</i>	::=	<i>Exp(<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(<exp>, <rel>, <exp>)</i> <i>ToBool(<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>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref(<exp>, <exp>)</i> <i>Ref(<exp>)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, <exp>)</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>StructAssign(Name(str), <exp>+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(<exp>, <stmt>*)</i> <i>IfElse(<exp>, <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(<exp>, <stmt>*)</i> <i>DoWhile(<exp>, <stmt>*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(<exp>)</i>	
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*)</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))* <stmt>*)</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>

Grammatik 3.2.10: Abstrakte Syntax der Sprache L_{PicoC}

Man spricht hier von der „**Abstrakten Syntax der Sprache L_{PicoC}** “ und meint hier mit der Sprache L_{PicoC} **nicht** die Sprache, welche durch die **Abstrakte Syntax** beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck die **Abstrakt Syntax** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakt Syntax** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Redeart** wurde aus der **Quelle** G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

3.2.5.5 Codebeispiel

In Code 3.5 ist der **Abstrakter Syntaxbaum** zu sehen, der aus dem **vereinfachten Ableitungsbaum** aus Code 3.4 mithilfe eines **Transformers** generiert wurde.

```

1 File
2   Name './verbose_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 '4',
14                Num '5'
15              ],
16            PtrDecl
17              Num '1',
18              IntType 'int',
19            Name 'attr'
20          ],
21      FunDef
22        VoidType 'void',
23        Name 'main',
24        [],
25        [
26          Exp
27            Alloc
28              Writeable,
29              ArrayDecl
30                [
31                  Num '3',
32                  Num '2'
33                ],
34              PtrDecl
35                Num '1',
36              PtrDecl
37                Num '1',
38              StructSpec
39                Name 'st',
40              Name 'var'
41            ]
42          ]

```

Code 3.5: *Aus vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum***3.2.5.6 Ausgabe des Abstrakter Syntaxbaums**

Ein **Knoten** eines **Abstrakter Syntaxbaum** kann entweder in der **Konkreter Syntax** der Sprache, für dessen Kompilierung er generiert wurde oder in der **Abstrakter Syntax**, die beschreibt, wie der Abstrakter Syntaxbaum selbst aufgebaut sein darf ausgegeben werden.

Das Ausgeben eines **Abstrakter Syntaxbaums** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`²⁰ der Programmiersprache **Python** umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkreten Syntax** aufgebaute **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **runden öffnenden** (und **schließenden**) **Klammern**, sowie **Kommas** ', ', **Semikolons** ; usw. zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach dem **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene **Textrepräsentation** passend **zusammenfügen** und selbst **zurückgeben**.

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

3.3 Code Generierung

Nach der Generierung eines **Abstrakter Syntaxbaum** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 2.4, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** im **Abstrakter Syntaxbaum** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** der Sprache L_{PicoC} , der durch die **Abstrakte Syntax** in Grammatik 3.2.10 spezifiziert ist in einen entsprechenden **Abstrakter Syntaxbaum** der Sprache L_{RETI} umzuformen. Das ganze lässt sich, wie in Unterkapitel 2.5 bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition 2.43) herunterbricht.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton 2.5). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel 2.1.1) in Abbildung 3.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer $X_{86,64}$ -Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer $X_{86,64}$ -Architektur laufen könnte. Und selbst dieses **T-Diagramm** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu $X_{86,64}$ kompiliert wurde usw.

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

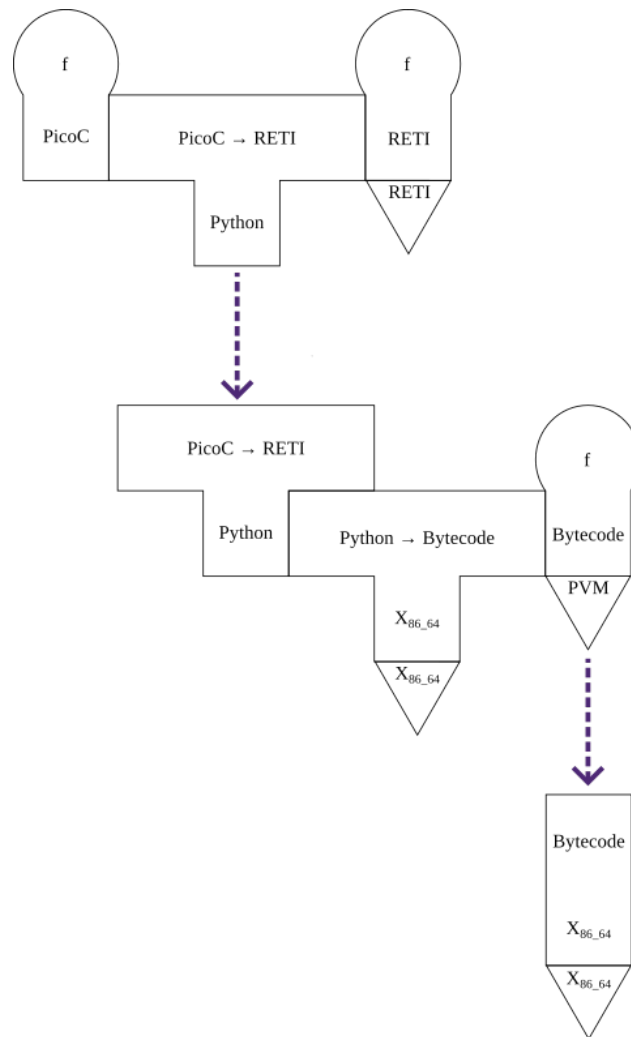


Abbildung 3.6: *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagramm** in Abbildung 3.6 lässt sich zusammenfassen, sodass man das **T-Diagramm** in Abbildung 3.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in X_{86_64} -Maschinensprache geschrieben ist.

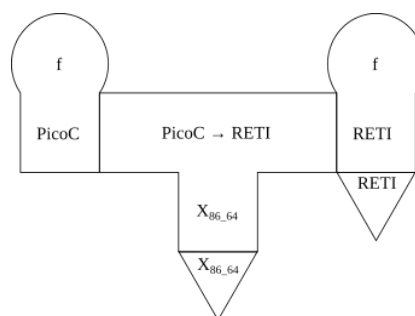


Abbildung 3.7: *Cross-Compiler Kompiliervorgang Kurzform*

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

der Kompilierprozess im Folgenden im **horizontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 3.8 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 2.29) des **PicoC-Compilers** aufeinanderfolgen. In der **Pipe-Architektur** nutzt der jeweils nächste **Pass** den generierten **Abstrakter Syntaxbaum** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstrakter Syntaxbaum** in seiner eigenen **Sprache** zu generieren.

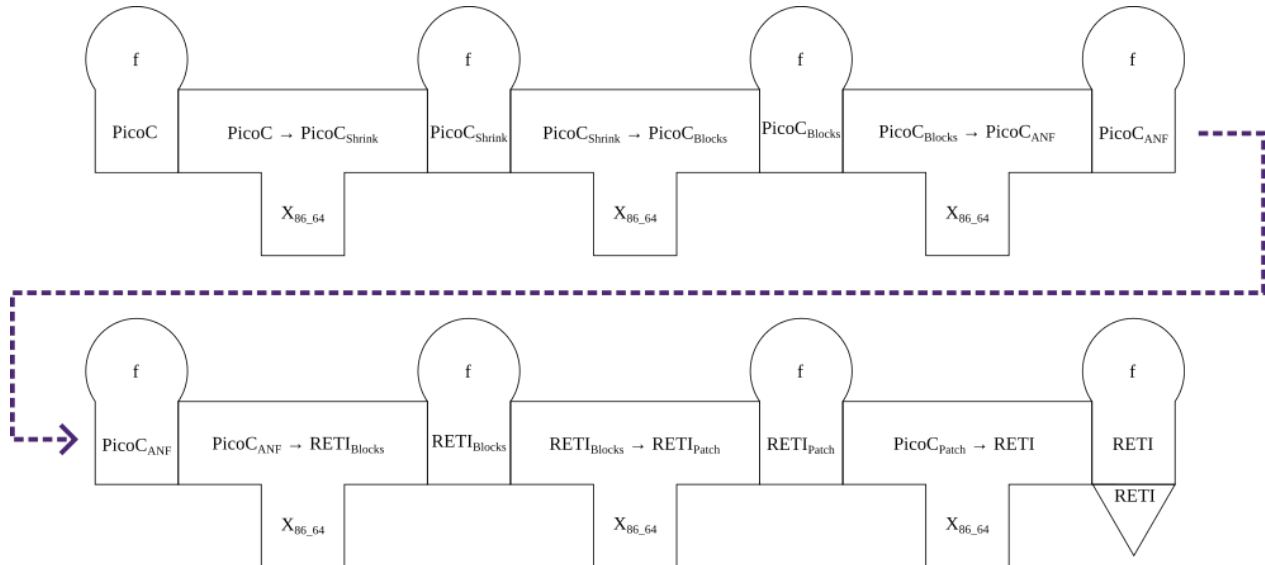


Abbildung 3.8: Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 3.3.1 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Pointern**, **Arrays**, **Structs** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 3.3.1 nicht ausreichend vertieft wurden. Viele der verwendeten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung P. D. C. Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird²¹. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem **Link**²² finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind²³.

3.3.1 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der verschiedenen Aspekte²⁴ der

²¹Also die verschiedenen in den **Passes** generierten **Abstrakter Syntaxbaums**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

²²https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.

²³Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstrakter Syntaxbaums** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

²⁴In kurz: **Pointer**, **Arrays**, **Structs** und **Funktionen**.

Bachelorarbeit wichtig.

Auf jedes Detail der einzelnen **Passes** wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Pointern**, **Arrays**, **Structs** und **Funktionen** im Detail erklärt sind und andererseits viele Aufgaben dieser **Passes** eher dem **Bachelorprojekt** zuzurechnen sind.

3.3.1.1 PicoC-Shrink Pass

3.3.1.1.1 Aufgabe

Der Aufgabe des **PicoC-Shrink Pass** ist in Unterkapitel 3.3.2 ausführlich an einem Beispiel erklärt. Kurzgefasst hat der **PicoC-Shrink Pass** die Aufgabe, die Eigenheit auszunutzen, dass der **Dereferenzierungsoperator** `*pntr` und die damit einhergehende **Pointer Arithmetik** `*(pntr + i)` sich in der Untermenge der Sprache L_C , welche die Sprache L_{PicoC} darstellt genau gleich verhält, wie der **Operator** für den **Zugriff** auf den **Index** eines **Arrays** `ar[i]`.

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** `Deref(exp, i)` im jeweiligen **Abstrakter Syntaxbaum** in **Knoten** `Subscr(exp, i)` um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung vermeiden lassen. Man lässt die **Dereferenzierung** `*(var + i)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `var[i]` übernehmen.

3.3.1.1.2 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache L_{PicoC_Shrink} in Tabelle 3.3.1 ist fast **identisch** mit der **Abstrakten Syntax** der Sprache L_{PicoC} in Tabelle 3.2.10, nach welcher der **erste** Abstrakter Syntaxbaum in der **Syntaktischen Analyse** generiert wurde. Der einzige **Unterschied** liegt darin, dass es den Knoten `Deref(exp, exp)` in Tabelle 3.3.1 **nicht** mehr gibt. Das liegt daran, dass dieser Pass alle **Vorkommnisse** des Knoten `Deref(exp, exp)` durch den Knoten `Subscr(exp, exp)` auswechselt, der ebenfalls bereits in der **Abstrakten Syntax** der Sprache L_{PicoC} definiert ist.

<i>stmt</i>	::=	<i>SingleLineComment</i> (<i>str</i> , <i>str</i>) <i>RETIComment</i> ()	<i>L_Comment</i>
<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</i> (<i>str</i>) <i>Num</i> (<i>str</i>) <i>Char</i> (<i>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</i> ('input'), <i>Empty</i> ()) <i>Call</i> (<i>Name</i> ('print'), <i><exp></i>)	
<i>stmt</i>	::=	<i>Exp</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>exp</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name</i> (<i>str</i>))	
<i>stmt</i>	::=	<i>Assign</i> (<i><exp></i> , <i><exp></i>)	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num</i> (<i>str</i>), <i><datatype></i>)	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref</i> (<i><exp></i> , <i><exp></i>) <i>Ref</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num</i> (<i>str</i>) +, <i><datatype></i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i><exp></i> , <i><exp></i>) <i>Array</i> (<i><exp></i> +)	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name</i> (<i>str</i>))	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i><exp></i> , <i>Name</i> (<i>str</i>)) <i>StructAssign</i> (<i>Name</i> (<i>str</i>), <i><exp></i>) +)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name</i> (<i>str</i>), <i>Alloc</i> (<i>Writeable</i> (), <i><datatype></i> , <i>Name</i> (<i>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</i> (<i>str</i>), <i><exp></i> *)	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return</i> (<i><exp></i>)	
<i>decl_def</i>	::=	<i>FunDecl</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>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><stmt></i> *)	
<i>file</i>	::=	<i>File</i> (<i>Name</i> (<i>str</i>), <i><decl_def></i> *)	<i>L_File</i>

Grammatik 3.3.1: Abstrakte Syntax der Sprache *L_{PiocC-Shrink}*

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

3.3.1.1.3 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 3.6 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 3.6 ist in der Funktion **faculty** ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen **Arguments** berechnet. Der Algorithmus basiert auf einem **Beispielprogramm** aus der

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

Dieser **rekursive** Algorithmus ist allerdings **kein** gutes **Anschauungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von **if**-, **if-else**-, **while**- und **do-while**-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 3.6 umgeschrieben, um **if**- und **while**-Statemtentens zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter [Link²⁵](#), unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln 3.3.2, 3.3.3, 3.3.4 und 3.3.6 zu **Pointern**, **Arrays**, **Structs** und **Funktionen** mit eigenen **Codebeispielen** erklärt werden und alle sonstigen Details dem **Bachelorprojekt** zuzurechnen sind.

```

1 // based on a example program from Christoph Scholl's Operating Systems lecture
2
3 int faculty(int n){
4     int res = 1;
5     while (1) {
6         if (n == 1) {
7             return res;
8         }
9         res = n * res;
10        n = n - 1;
11    }
12 }
13
14 void main() {
15     print(faculty(4));
16 }

```

Code 3.6: PicoC Code für Codebespiel

In Code 3.7 sieht man den **Abstrakter Syntaxbaum**, der in der **Syntaktischen Analyse** generiert wurde.

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],

```

²⁵https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```

10  [
11    Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
12    While
13      Num '1',
14      [
15        If
16          Atom(Name('n'), Eq('=='), Num('1')),
17          [
18            Return(Name('res'))
19          ]
20          Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21          Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22      ]
23  ],
24  FunDef
25    VoidType 'void',
26    Name 'main',
27    [],
28    [
29      Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30    ]
31  ]

```

Code 3.7: Abstrakter Syntaxbaum für Codebeispiel

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstrakter Syntaxbaum** in Code 3.7, da das Codebeispiel keine **Dereferenzierung** enthält.

3.3.1.2 PicoC-Blocks Pass

3.3.1.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** ist es die Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs-)`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)-Knoten` umzusetzen. Der `IfElse(exp, stmts1, stmts2)-Knoten` wird zur Umsetzung der **Bedingung** verwendet und es wird, je nachdem, ob die Bedingung **wahr** oder **falsch** ist mithilfe der `GoTo(label)-Knoten` in einen von zwei **alternativen Branches** gesprungen oder ein **Branch** erneut aufgerufen usw.

3.3.1.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache L_{PicoC_Shrink} in Tabelle 3.3.1 um die Knoten zu erweitern, die im Unterkapitel 3.3.1.2.1 erwähnt wurden. Die Knoten `If(exp, stmts)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` gibt es nicht mehr, da sie durch `Block(name, stmts_instrs-)`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)-Knoten` ersetzt wurden. Die **Funktionsdefinition** `FunDef(<datatype>, Name(str), Alloc(Writable(), <datatype>, Name(str))* , <block>*)` ist nun ein Container für **Blöcke** `Block(Name(str), <stmt>*)` und keine Statements `stmt` mehr. Das resultiert in der **Abstrakten Syntax** der Sprache L_{PicoC_Blocks} in Tabelle 3.3.2.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>
<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(<exp>, <bin_op>, <exp>)</i> <i>UnOp(<un_op>, <exp>)</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i>	
<i>stmt</i>	::=	<i>Exp(<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(<exp>, <rel>, <exp>)</i> <i>ToBool(<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>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Ref(<exp>)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, <exp>)</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>StructAssign(Name(str), <exp>)+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(<exp>, <stmt>*)</i> <i>IfElse(<exp>, <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(<exp>, <stmt>*)</i> <i>DoWhile(<exp>, <stmt>*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(<exp>)</i>	
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*)</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*</i> , <i><block>*)</i>	
<i>block</i>	::=	<i>Block(Name(str), <stmt>*)</i>	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo(Name(str))</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>

Grammatik 3.3.2: Abstrakte Syntax der Sprache L_{Pioc_Blocks}

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

Die **Abstrakte Syntax** soll im Gegensatz zur **Konkreten Syntax** meist nur vom **Programmierer**

verstanden werden, der den Compiler implementiert und sollte daher vor allem **einfach verständlich** sein und stellt daher eine **Obermenge** aller tatsächlich möglichen **Kompositionen** von **Knoten** dar^a.

Man bezeichnet hier die **Abstrakte Syntax** als „**Abstrakte Syntax der Sprache** L_{Picoc_Blocks} “. Diese Sprache L_{Picoc_Blocks} wird durch eine **Konkrete Syntax** beschrieben, die allerdings nicht weiter relevant ist, da in den **Passes** nur **Abstrakter Syntaxbaums** umgeformt werden. Es ist hierbei nur wichtig zu wissen, dass die **Abstrakte Syntax** theoretisch zur Kompilierung der Sprache L_{Picoc_Blocks} definiert ist, also die Sprache L_{PicoC_Blocks} nicht die Sprache ist, die von der **Abstrakten Syntax** beschrieben ist.

^aD.h. auch wenn dort **exp** als **Attribut** steht, kann dort **nicht** jeder Knoten, der sich aus der **Produktion** **exp** ergibt auch wirklich eingesetzt werden.

3.3.1.2.3 Codebeispiel

In Code 3.8 sieht man den **Abstract-Syntax-Tree** des **PiocoC-Blocks Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion **faculty** und die **main**-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement oder bis zum ersten **Auftauchen** eines **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-Knoten** stehen. Je nachdem, ob ein **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-Knoten** auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```

1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
10      [
11        Block
12          Name 'faculty.6',
13          [
14            Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1'))
15            // While(Num('1'), [])
16            GoTo(Name('condition_check.5'))
17          ],
18        Block
19          Name 'condition_check.5',
20          [
21            IfElse
22              Num '1',
23              [
24                GoTo(Name('while_branch.4'))
25              ],
26              [
27                GoTo(Name('while_after.1'))
28              ]
29          ],
30        Block
31          Name 'while_branch.4',

```

```

32     [
33         // If(Atom(Name('n'), Eq('=='), Num('1')), [],),
34         IfElse
35             Atom(Name('n'), Eq('=='), Num('1')),
36             [
37                 GoTo(Name('if.3'))
38             ],
39             [
40                 GoTo(Name('if_else_after.2'))
41             ]
42     ],
43     Block
44         Name 'if.3',
45         [
46             Return(Name('res'))
47         ],
48     Block
49         Name 'if_else_after.2',
50         [
51             Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52             Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53             GoTo(Name('condition_check.5'))
54         ],
55     Block
56         Name 'while_after.1',
57         []
58 ],
59 FunDef
60     VoidType 'void',
61     Name 'main',
62     [],
63     [
64         Block
65             Name 'main.0',
66             [
67                 Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
68             ]
69     ]
70 ]

```

Code 3.8: PicoC-Blocks Pass für Codebeispiel

3.3.1.3 PicoC-ANF Pass

3.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** ist es den **Abstrakter Syntaxbaum** der Sprache L_{PicoC_Blocks} in die **Abstrakte Syntax** der Sprache L_{PicoC_ANF} umzuformen, welche in **A-Normalform** (Definition 2.50) und damit auch in **Monadischer Normalform** (Definition 2.46) ist. Um Wiederholung zu vermeiden wird zur Erklärung der **A-Normalform** auf Unterkapitel 2.5.2 verwiesen.

Zudem wird eine **Symboltabelle** (Definition 3.11) eingeführt. In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** zugewiesen, die dem Wert einer von zwei **Countern** `rel_global_addr` und `rel_stack_addr` entspricht. Der Counter `rel_global_addr` ist für Variablen in den **Globalen Statischen Daten** und der Counter `rel_stack_addr` ist für Variablen auf dem **Stackframe**. Einer der beiden **Counter** wird entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle diese Variable `Name('symbol')` vor, so wird mit dem **Symbol**²⁶ als Schlüssel in der **Symboltabelle** nachgeschlagen und anstelle des `Name(str)`-Knotens die in der **Symboltabelle** nachgeschlagene Adresse in einem `Global(Num('addr'))`- bzw. `Stackframe(Num('addr'))`-Knoten eingesetzt eingefügt. Ob der `Global(Num('addr'))`- oder der `Stackframe(Num('addr'))`-Knoten zum Einsatz kommt, entscheidet sich anhand des **Scopes** (z.B. `@scope`), der in der **Symboltabelle** an den **Bezeichner** drangehängt ist (z.B. `identifizier@scope`).²⁷

Definition 3.11: Symboltabelle

*Eine über ein **Assoziatives Feld** umgesetzte **Datenstruktur**, die notwendig ist, um das Konzept einer **Variablen** in einer Sprache umzusetzen. Diese Datenstruktur ordnet jedem **Symbol**^a einer **Variablen**, **Konstanten** oder **Funktion** aus einem **Programm**, Informationen, wie die **Adresse**, die **Position** im Programmcode oder den **Datentyp** zu.*

*Die **Symboltabelle** muss nur während des **Kompilervorgangs** im **Speicher** existieren, da die Einträge in der **Symboltabelle** beeinflussen, was für **Maschinencode** generiert wird und dadurch im **Maschinencode** bereits die richtigen **Adressen** usw. angesprochen werden und es die **Symboltabelle** selbst **nicht** mehr braucht.*

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

3.3.1.3.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache L_{PicoC_Blocks} in Tabelle 3.3.2 in die **A-Normalform** zu bringen. Darunter fällt es unter anderem, dafür zu sorgen, dass **Komplexe Knoten**, wie z.B. `BinOp(exp, bin_op, exp)` nur **Atomare Knoten**, wie z.B. `Stack(Num(str))` enthalten können. Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass u.a. die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `File(Name(str), block*)`-Knoten liegen usw., was in Unterkapitel 3.3.6 genauer erklärt wird. Die **Symboltabelle** ist ebenfalls als **Abstrakter Syntaxbaum** umgesetzt, wofür in der **Abstrakten Syntax** der Sprache L_{PicoC_ANF} in Grammatik 3.3.3 neue Knoten eingeführt werden.

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

²⁶Bzw. der **Bezeichner**

²⁷Die Umsetzung von **Scopes** wird in Unterkapitel 3.3.6.2 genauer beschrieben.

<i>stmt</i>	::=	<i>SingleLineComment</i> (<i>str</i> , <i>str</i>) <i>RETIComment</i> ()	<i>L_Comment</i>
<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</i> (<i>str</i>) <i>Num</i> (<i>str</i>) <i>Char</i> (<i>str</i>) <i>Global</i> (<i>Num</i> (<i>str</i>)) <i>Stackframe</i> (<i>Num</i> (<i>str</i>)) <i>Stack</i> (<i>Num</i> (<i>str</i>)) <i>BinOp</i> (<i>Stack</i> (<i>Num</i> (<i>str</i>)), <i>bin_op</i> , <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>UnOp</i> (<i>un_op</i> , <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>Call</i> (<i>Name</i> ('input'), <i>Empty</i> ()) <i>Call</i> (<i>Name</i> ('print'), <i>exp</i>) <i>Exp</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>Stack</i> (<i>Num</i> (<i>str</i>)), <i>rel</i> , <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>ToBool</i> (<i>Stack</i> (<i>Num</i> (<i>str</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>exp</i>	::=	<i>Alloc</i> (<i>type_qual</i> , <i>datatype</i> , <i>Name</i> (<i>str</i>))	
<i>stmt</i>	::=	<i>Assign</i> (<i>Global</i> (<i>Num</i> (<i>str</i>)), <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>Assign</i> (<i>Stackframe</i> (<i>Num</i> (<i>str</i>)), <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>Assign</i> (<i>Stack</i> (<i>Num</i> (<i>str</i>)), <i>Global</i> (<i>Num</i> (<i>str</i>))) <i>Assign</i> (<i>Stack</i> (<i>Num</i> (<i>str</i>)), <i>Stackframe</i> (<i>Num</i> (<i>str</i>)))	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num</i> (<i>str</i>), <i>datatype</i>) <i>Ref</i> (<i>Global</i> (<i>str</i>)) <i>Ref</i> (<i>Stackframe</i> (<i>str</i>)) <i>Ref</i> (<i>Subscr</i> (<i>exp</i> , <i>exp</i>) <i>Ref</i> (<i>Attr</i> (<i>exp</i> , <i>Name</i> (<i>str</i>))))	<i>L_Pntr</i>
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num</i> (<i>str</i>)+, <i>datatype</i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i>exp</i> , <i>Stack</i> (<i>Num</i> (<i>str</i>))) <i>Array</i> (<i>exp</i>)+)	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name</i> (<i>str</i>))	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i>exp</i> , <i>Name</i> (<i>str</i>)) <i>Struct</i> (<i>Assign</i> (<i>Name</i> (<i>str</i>), <i>exp</i>))+)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name</i> (<i>str</i>), <i>Alloc</i> (<i>Writeable</i> (), <i>datatype</i> , <i>Name</i> (<i>str</i>))+)	
<i>stmt</i>	::=	<i>IfElse</i> (<i>Stack</i> (<i>Num</i> (<i>str</i>)), <i>stmt</i> *, <i>stmt</i> *)	<i>L_If_Else</i>
<i>exp</i>	::=	<i>Call</i> (<i>Name</i> (<i>str</i>), <i>exp</i> *)	<i>L_Fun</i>
<i>stmt</i>	::=	<i>StackMalloc</i> (<i>Num</i> (<i>str</i>)) <i>NewStackframe</i> (<i>Name</i> (<i>str</i>), <i>GoTo</i> (<i>str</i>)) <i>Exp</i> (<i>GoTo</i> (<i>Name</i> (<i>str</i>))) <i>RemoveStackframe</i> () <i>Return</i> (<i>Empty</i> ()) <i>Return</i> (<i>exp</i>)	
<i>decl_def</i>	::=	<i>FunDecl</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>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>block</i>	::=	<i>Block</i> (<i>Name</i> (<i>str</i>), <i>stmt</i> *)	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo</i> (<i>Name</i> (<i>str</i>))	
<i>file</i>	::=	<i>File</i> (<i>Name</i> (<i>str</i>), <i>block</i> *)	<i>L_File</i>
<i>symbol_table</i>	::=	<i>SymbolTable</i> (<i>symbol</i> *)	<i>L_Symbol_Table</i>
<i>symbol</i>	::=	<i>Symbol</i> (<i>type_qual</i> , <i>datatype</i> , <i>name</i> , <i>val</i> , <i>pos</i> , <i>size</i>)	
<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> ()	

3.3.1.3.3 Codebeispiel

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

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         // Assign(Name('res'), Num('1'))
8         Exp(Num('1'))
9         Assign(Stackframe(Num('1')), Stack(Num('1')))
10        // While(Num('1'), [])
11        Exp(GoTo(Name('condition_check.5')))
12      ],
13    Block
14      Name 'condition_check.5',
15      [
16        // IfElse(Num('1'), [], [])
17        Exp(Num('1')),
18        IfElse
19          Stack
20            Num '1',
21            [
22              GoTo(Name('while_branch.4'))
23            ],
24            [
25              GoTo(Name('while_after.1'))
26            ]
27        ],
28      Block
29        Name 'while_branch.4',
30        [
31          // If(Atom(Name('n'), Eq('=='), Num('1')), [])
32          // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
33          Exp(Stackframe(Num('0')))
34          Exp(Num('1'))
35          Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
36          IfElse
37            Stack
38              Num '1',
39              [
40                GoTo(Name('if.3'))
41              ],
42              [
43                GoTo(Name('if_else_after.2'))
44              ]
45        ],

```

```

46 Block
47   Name 'if.3',
48   [
49     // Return(Name('res'))
50     Exp(Stackframe(Num('1')))
51     Return(Stack(Num('1')))
52   ],
53 Block
54   Name 'if_else_after.2',
55   [
56     // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
57     Exp(Stackframe(Num('0')))
58     Exp(Stackframe(Num('1')))
59     Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
60     Assign(Stackframe(Num('1')), Stack(Num('1')))
61     // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
62     Exp(Stackframe(Num('0')))
63     Exp(Num('1'))
64     Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
65     Assign(Stackframe(Num('0')), Stack(Num('1')))
66     Exp(GoTo(Name('condition_check.5')))
67   ],
68 Block
69   Name 'while_after.1',
70   [
71     Return(Empty())
72   ],
73 Block
74   Name 'main.0',
75   [
76     StackMalloc(Num('2'))
77     Exp(Num('4'))
78     NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
79     Exp(GoTo(Name('faculty.6')))
80     RemoveStackframe()
81     Exp(ACC)
82     Exp(Call(Name('print'), [Stack(Num('1'))]))
83     Return(Empty())
84   ]
85 ]

```

Code 3.9: PicoC-ANF Pass für Codebespiel

3.3.1.4 RETI-Blocks Pass

3.3.1.4.1 Aufgabe

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

3.3.1.4.2 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache L_{RETI_Blocks} in Grammatik 3.3.4 ist verglichen mit der **Abstrakten Syntax** der Sprache L_{PicoC_ANF} in Grammatik 3.3.3 stark verändert, denn der Großteil der **PicoC-Knoten** wird in diesem Pass durch entsprechende **RETI-Knoten** ersetzt. Die einzigen verbleibenden **PicoC-Knoten**

sind $\text{Exp}(\text{GoTo}(\text{str}))$, $\text{Block}(\text{Name}(\text{str}), \langle \text{instr} \rangle^*)$ und $\text{File}(\text{Name}(\text{str}), \langle \text{block} \rangle^*)$, da das gesamte Konzept mit den **Blöcken** erst im **RETI-Pass** in Unterkapitel 3.3.8 aufgelöst wird.

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

Grammatik 3.3.4: Abstrakte Syntax der Sprache L_{RETI_Blocks}

3.3.1.4.3 Codebeispiel

In Code 3.10 sieht man den **Abstract-Syntax-Tree** des **RETI-Blocks Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem die **Statements**, die durch entsprechende **PicoC-Knoten** im **Abstrakt Syntax Tree** der Sprache L_{PicoC_ANF} in Grammatik 3.3.3 repräsentiert waren nun durch ihre entsprechenden **RETI-Knoten** ersetzt werden.

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         # // Assign(Name('res'), Num('1'))
8         # Exp(Num('1'))
9         SUBI SP 1;
10        LOADI ACC 1;
11        STOREIN SP ACC 1;
12        # Assign(Stackframe(Num('1')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN BAF ACC -3;
15        ADDI SP 1;
16        # // While(Num('1'), [])
17        # Exp(GoTo(Name('condition_check.5')))
18        Exp(GoTo(Name('condition_check.5')))
19      ],
20    Block
21      Name 'condition_check.5',
22      [
23        # // IfElse(Num('1'), [], [])
24        # Exp(Num('1'))

```

```

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

```

```

82     STOREIN SP ACC 1;
83     # Exp(Stackframe(Num('1')))
84     SUBI SP 1;
85     LOADIN BAF ACC -3;
86     STOREIN SP ACC 1;
87     # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88     LOADIN SP ACC 2;
89     LOADIN SP IN2 1;
90     MULT ACC IN2;
91     STOREIN SP ACC 2;
92     ADDI SP 1;
93     # Assign(Stackframe(Num('1')), Stack(Num('1')))
94     LOADIN SP ACC 1;
95     STOREIN BAF ACC -3;
96     ADDI SP 1;
97     # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98     # Exp(Stackframe(Num('0')))
99     SUBI SP 1;
100    LOADIN BAF ACC -2;
101    STOREIN SP ACC 1;
102    # Exp(Num('1'))
103    SUBI SP 1;
104    LOADI ACC 1;
105    STOREIN SP ACC 1;
106    # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107    LOADIN SP ACC 2;
108    LOADIN SP IN2 1;
109    SUB ACC IN2;
110    STOREIN SP ACC 2;
111    ADDI SP 1;
112    # Assign(Stackframe(Num('0')), Stack(Num('1')))
113    LOADIN SP ACC 1;
114    STOREIN BAF ACC -2;
115    ADDI SP 1;
116    # Exp(GoTo(Name('condition_check.5')))
117    Exp(GoTo(Name('condition_check.5')))
118  ],
119  Block
120    Name 'while_after.1',
121    [
122      # Return(Empty())
123      LOADIN BAF PC -1;
124    ],
125  Block
126    Name 'main.0',
127    [
128      # StackMalloc(Num('2'))
129      SUBI SP 2;
130      # Exp(Num('4'))
131      SUBI SP 1;
132      LOADI ACC 4;
133      STOREIN SP ACC 1;
134      # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
135      MOVE BAF ACC;
136      ADDI SP 3;
137      MOVE SP BAF;
138      SUBI SP 4;

```

```

139     STOREIN BAF ACC 0;
140     LOADI ACC GoTo(Name('addr@next_instr'));
141     ADD ACC CS;
142     STOREIN BAF ACC -1;
143     # Exp(GoTo(Name('faculty.6'))))
144     Exp(GoTo(Name('faculty.6'))))
145     # RemoveStackframe()
146     MOVE BAF IN1;
147     LOADIN IN1 BAF 0;
148     MOVE IN1 SP;
149     # Exp(ACC)
150     SUBI SP 1;
151     STOREIN SP ACC 1;
152     LOADIN SP ACC 1;
153     ADDI SP 1;
154     CALL PRINT ACC;
155     # Return(Empty())
156     LOADIN BAF PC -1;
157 ]
158 ]

```

Code 3.10: RETI-Blocks Pass für Codebeispiel

Wenn der **Abstrakter Syntaxbaum** ausgegeben wird, ist die Darstellung nicht ausschließlich in **Abstrakter Syntax**, da die **RETI-Knoten** aus bereits im Unterkapitel 3.2.5.6 vermitteltem Grund in **Konkreter Syntax** ausgegeben werden.

3.3.1.5 RETI-Patch Pass

3.3.1.5.1 Aufgabe

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

- das **Einfügen** eines **start.<nummer>-Blockes**, welcher ein **GoTo(Name('main'))** zur **main-Funktion** enthält, wenn in manchen Fällen die **main-Funktion** **nicht** die erste Funktion ist und daher am Anfang zur **main-Funktion** gesprungen werden muss.
- das **Entfernen** von **GoTo()**'s, deren Sprung nur **eine** Adresse weiterspringen würde.
- das **Voranstellen** von **RETI-Knoten**, die vor jeder **Division Instr(Div(), args)** prüfen, ob, nicht durch 0 geteilt wird.²⁸
- das Überprüfen darauf, ob bestimmte **Immediates Im(str)** in Befehlen, wie z.B. **Jump(rel, Im(str))**, **Instr(Loadin(), [reg, reg, Im(str)])**, **Instr(Loadi(), [reg, Im(str)])** usw. **kleiner** -2^{21} oder **größer** $2^{21} - 1$ sind. Im Fall dessen, dass es so ist, muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **bitweise Oder** berechnet werden. Im Fall, dessen, dass der **Immediate** allerdings **kleiner** $-(2^{31})$ oder **größer** $2^{31} - 1$ ist, wird eine Fehlermeldung **TooLargeLiteral** ausgegeben.

3.3.1.5.2 Abstrakte Syntax

²⁸Das fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

Die **Abstrakte Syntax** der Sprache L_{RETI_Patch} in Grammatik 3.3.5 ist im Vergleich zur **Abstrakten Syntax** der Sprache L_{RETI_Blocks} in Grammatik 3.3.4 kaum verändert. Es muss nur ein Knoten `Exit()` hinzugefügt werden, der im Falle einer **Division durch 0** die Ausführung des Programs beendet.

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

Grammatik 3.3.5: Abstrakte Syntax der Sprache L_{RETI_Patch}

3.3.1.5.3 Codebeispiel

In Code 3.11 sieht man den **Abstract-Syntax-Tree** des **Pioco-Patch Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>-Block`²⁹ eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))-Statements` entfernt³⁰, die nur einen Sprung um **eine** Position entsprochen hätten.

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.7',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'faculty.6',
12      [
13        # // Assign(Name('res'), Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Assign(Stackframe(Num('1')), Stack(Num('1')))
19        LOADIN SP ACC 1;
20        STOREIN BAF ACC -3;

```

²⁹Dieser **Block** wurde im Code 3.8 markiert.

³⁰Diese **entfernten** `GoTo(Name(str))`'s wurden ebenfalls im Code 3.8 markiert.


```

21     ADDI SP 1;
22     # // While(Num('1'), [])
23     # Exp(GoTo(Name('condition_check.5')))
24     # // not included Exp(GoTo(Name('condition_check.5')))
25 ],
26 Block
27     Name 'condition_check.5',
28     [
29         # // IfElse(Num('1'), [], [])
30         # Exp(Num('1'))
31         SUBI SP 1;
32         LOADI ACC 1;
33         STOREIN SP ACC 1;
34         # IfElse(Stack(Num('1')), [], [])
35         LOADIN SP ACC 1;
36         ADDI SP 1;
37         JUMP== GoTo(Name('while_after.1'));
38         # // not included Exp(GoTo(Name('while_branch.4')))
39     ],
40 Block
41     Name 'while_branch.4',
42     [
43         # // If(Atom(Name('n'), Eq('='), Num('1')), [])
44         # // IfElse(Atom(Name('n'), Eq('='), Num('1')), [], [])
45         # Exp(Stackframe(Num('0')))
46         SUBI SP 1;
47         LOADIN BAF ACC -2;
48         STOREIN SP ACC 1;
49         # Exp(Num('1'))
50         SUBI SP 1;
51         LOADI ACC 1;
52         STOREIN SP ACC 1;
53         LOADIN SP ACC 2;
54         LOADIN SP IN2 1;
55         SUB ACC IN2;
56         JUMP== 3;
57         LOADI ACC 0;
58         JUMP 2;
59         LOADI ACC 1;
60         STOREIN SP ACC 2;
61         ADDI SP 1;
62         # IfElse(Stack(Num('1')), [], [])
63         LOADIN SP ACC 1;
64         ADDI SP 1;
65         JUMP== GoTo(Name('if_else_after.2'));
66         # // not included Exp(GoTo(Name('if.3')))
67     ],
68 Block
69     Name 'if.3',
70     [
71         # // Return(Name('res'))
72         # Exp(Stackframe(Num('1')))
73         SUBI SP 1;
74         LOADIN BAF ACC -3;
75         STOREIN SP ACC 1;
76         # Return(Stack(Num('1')))
77         LOADIN SP ACC 1;

```

```

78     ADDI SP 1;
79     LOADIN BAF PC -1;
80 ],
81 Block
82     Name 'if_else_after.2',
83     [
84         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85         # Exp(Stackframe(Num('0')))
86         SUBI SP 1;
87         LOADIN BAF ACC -2;
88         STOREIN SP ACC 1;
89         # Exp(Stackframe(Num('1')))
90         SUBI SP 1;
91         LOADIN BAF ACC -3;
92         STOREIN SP ACC 1;
93         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94         LOADIN SP ACC 2;
95         LOADIN SP IN2 1;
96         MULT ACC IN2;
97         STOREIN SP ACC 2;
98         ADDI SP 1;
99         # Assign(Stackframe(Num('1')), Stack(Num('1')))
100        LOADIN SP ACC 1;
101        STOREIN BAF ACC -3;
102        ADDI SP 1;
103        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104        # Exp(Stackframe(Num('0')))
105        SUBI SP 1;
106        LOADIN BAF ACC -2;
107        STOREIN SP ACC 1;
108        # Exp(Num('1'))
109        SUBI SP 1;
110        LOADI ACC 1;
111        STOREIN SP ACC 1;
112        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113        LOADIN SP ACC 2;
114        LOADIN SP IN2 1;
115        SUB ACC IN2;
116        STOREIN SP ACC 2;
117        ADDI SP 1;
118        # Assign(Stackframe(Num('0')), Stack(Num('1')))
119        LOADIN SP ACC 1;
120        STOREIN BAF ACC -2;
121        ADDI SP 1;
122        # Exp(GoTo(Name('condition_check.5')))
123        Exp(GoTo(Name('condition_check.5')))
124    ],
125 Block
126     Name 'while_after.1',
127     [
128         # Return(Empty())
129         LOADIN BAF PC -1;
130     ],
131 Block
132     Name 'main.0',
133     [
134         # StackMalloc(Num('2'))

```

```

135     SUBI SP 2;
136     # Exp(Num('4'))
137     SUBI SP 1;
138     LOADI ACC 4;
139     STOREIN SP ACC 1;
140     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
141     MOVE BAF ACC;
142     ADDI SP 3;
143     MOVE SP BAF;
144     SUBI SP 4;
145     STOREIN BAF ACC 0;
146     LOADI ACC GoTo(Name('addr@next_instr'));
147     ADD ACC CS;
148     STOREIN BAF ACC -1;
149     # Exp(GoTo(Name('faculty.6')))
150     Exp(GoTo(Name('faculty.6')))
151     # RemoveStackframe()
152     MOVE BAF IN1;
153     LOADIN IN1 BAF 0;
154     MOVE IN1 SP;
155     # Exp(ACC)
156     SUBI SP 1;
157     STOREIN SP ACC 1;
158     LOADIN SP ACC 1;
159     ADDI SP 1;
160     CALL PRINT ACC;
161     # Return(Empty())
162     LOADIN BAF PC -1;
163 ]
164 ]

```

Code 3.11: RETI-Patch Pass für Codebespiel

3.3.1.6 RETI Pass

3.3.1.6.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist es die `GoTo(Name(str))`-Knoten in den den Knoten `Instr(Loadi(), [reg, GoTo(Name(str))])`, `Jump(Eq(), GoTo(Name(str)))` und `Exp(GoTo(Name(str)))` durch eine entsprechende **Adresse** zu ersetzen, die entsprechende **Distanz** oder einen entsprechenden **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))`. Die **Distanz**- und **Adressberechnung** wird in Unterkapitel 3.3.6.3 genauer mit **Formeln** erklärt.

3.3.1.6.2 Konkrete und Abstrakte Syntax

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

Ausgegeben wird der finale **Maschinencode** allerdings in **Konkreter Syntax**, die sich aus den Grammatiken 3.3.6 und 3.3.7 für jeweils die **Lexikalische** und **Syntaktische Analyse** zusammensetzt. Der Grund, warum die **Konkrete Syntax** der Sprache L_{RETI} auch nochmal in einen Teil für die **Lexikalische** und **Syntaktische Analyse** unterteilt ist, hat den Grund, dass für die Bachelorarbeit zum **Testen** des **PicoC-Compilers** ein **RETI-Interpreter** implementiert wurde, der den RETI-Code **lexen** und **parsen** muss, um ihn später **interpretieren** zu können.

<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_no_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"	

Grammatik 3.3.6: Konkrete Syntax der Sprache *L_{RETI}* für die Lexikalische Analyse in EBNF

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

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

<i>reg</i>	::=	<i>ACC</i> () <i>IN1</i> () <i>IN2</i> () <i>PC</i> () <i>SP</i> () <i>BAF</i> ()	<i>L_RETI</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>Instr</i> (<i>Loadi</i> (), [<i>Reg</i> (<i>Acc</i> ()), <i>GoTo</i> (<i>Name</i> (<i>str</i>))]) <i>Jump</i> (<i>Eq</i> (), <i>GoTo</i> (<i>Name</i> (<i>str</i>)))	
<i>program</i>	::=	<i>Program</i> (<i>Name</i> (<i>str</i>), (<i>instr</i>)*)	
<i>instr</i>	::=	<i>Exp</i> (<i>GoTo</i> (<i>str</i>)) <i>Exit</i> (<i>Num</i> (<i>str</i>))	<i>L_PicoC</i>
<i>block</i>	::=	<i>Block</i> (<i>Name</i> (<i>str</i>), (<i>instr</i>)*)	
<i>file</i>	::=	<i>File</i> (<i>Name</i> (<i>str</i>), (<i>block</i>)*)	

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

3.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkreten Syntax** ausgegeben werden, wie in Code 3.12 zu sehen ist. Es gibt **keine Blöcke** mehr und die **RETI-Befehle** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** ausschließlich aus **RETI-Ausdrücken** bestehen³¹, die sich in den **Blöcken** befunden haben, wurden durch **RETI-Befehle** ersetzt.

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

```

1 # // Exp(GoTo(Name('main.0')))
2 JUMP 67;
3 # // Assign(Name('res'), Num('1'))
4 # Exp(Num('1'))
5 SUBI SP 1;
6 LOADI ACC 1;
7 STOREIN SP ACC 1;
8 # Assign(Stackframe(Num('1')), Stack(Num('1')))
9 LOADIN SP ACC 1;
10 STOREIN BAF ACC -3;
11 ADDI SP 1;
12 # // While(Num('1'), [])
13 # Exp(GoTo(Name('condition_check.5')))
14 # // not included Exp(GoTo(Name('condition_check.5')))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4')))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0')))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;

```

³¹Wie z.B. `LOADI ACC GoTo(Name('addr@next_instr')), Exp(GoTo(Name('main.0')))` und `JUMP== GoTo(Name('if_else_after.2'))`.

```

43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3;
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;
73 # Assign(Stackframe(Num('1')), Stack(Num('1')))
74 LOADIN SP ACC 1;
75 STOREIN BAF ACC -3;
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1')))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5')))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;

```

```
00 # StackMalloc(Num('2'))
01 SUBI SP 2;
02 # Exp(Num('4'))
03 SUBI SP 1;
04 LOADI ACC 4;
05 STOREIN SP ACC 1;
06 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr'))
07 MOVE BAF ACC;
08 ADDI SP 3;
09 MOVE SP BAF;
10 SUBI SP 4;
11 STOREIN BAF ACC 0;
12 LOADI ACC 80;
13 ADD ACC CS;
14 STOREIN BAF ACC -1;
15 # Exp(GoTo(Name('faculty.6'))
16 JUMP -78;
17 # RemoveStackframe()
18 MOVE BAF IN1;
19 LOADIN IN1 BAF 0;
20 MOVE IN1 SP;
21 # Exp(ACC)
22 SUBI SP 1;
23 STOREIN SP ACC 1;
24 LOADIN SP ACC 1;
25 ADDI SP 1;
26 CALL PRINT ACC;
27 # Return(Empty())
28 LOADIN BAF PC -1;
```

Code 3.12: *RETI Pass für Codebespiel*

3.3.2 Umsetzung von Pointern

3.3.2.1 Referenzierung

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

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

Code 3.13: *PicoC-Code für Pointer Referenzierung*

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakter Syntaxbaum** in Code 3.14 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Pointer `*pntr`.

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

Code 3.14: *Abstrakter Syntaxbaum für Pointer Referenzierung*

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

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

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



```

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

```

Code 3.15: Symboltabelle für Pointer Referenzierung

Im **PicoC-ANF Pass** in Code 3.16 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')), Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp)` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var')))` ist, sind noch weitere Anweisungen zwischen den Zeilen 11 und 12 nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 3.3.5.3 kümmern.

```

1 File
2   Name './example_ptr_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('ptr'), Ref(Name('var')))
11      Ref(Global(Num('0')))
12      Assign(Global(Num('1')), Stack(Num('1')))
13      Return(Empty())
14    ]
15  ]

```

Code 3.16: PicoC-ANF Pass für Pointer Referenzierung

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

```

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 3.17: RETI-Blocks Pass für Pointer Referenzierung

3.3.2.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

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

```

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

```

Code 3.18: PicoC-Code für Pointer Dereferenzierung

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

```

1 File
2   Name './example_ptrntr_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(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11              ↪ Ref(Name('var')))
12        Exp(Deref(Name('ptr'), Num('0')))
13      ]
14    ]

```

Code 3.19: Abstrakter Syntaxbaum für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 3.20 wird ein Trick angewandt, bei dem jeder Knoten `Deref(Name('ptr'), Num('0'))` einfach durch den Knoten `Subscr(Name('ptr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Arrayindex** (z.B. `var[1]`) verhält³². Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Dereferenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Arrayindex** (z.B. `var[1]`) übernehmen lassen.

```

1 File
2   Name './example_ptrntr_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(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11              ↪ Ref(Name('var')))
12        Exp(Subscr(Name('ptr'), Num('0')))
13      ]
14    ]

```

Code 3.20: PicoC-Shrink Pass für Pointer Dereferenzierung

3.3.3 Umsetzung von Arrays

3.3.3.1 Initialisierung von Arrays

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

³²In der Sprache L_C gibt es einen Unterschied bei der Initialisierung bei z.B. `int *var = "string"` und z.B. `int var[1] = "string"`, der allerdings nichts mit den beiden Operatoren zu 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 (z.B. `var[1]`), obwohl es ein **Derived Datatype** ist.

```

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

```

Code 3.21: PicoC-Code für Array Initialisierung

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

```

1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
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 3.22: Abstrakter Syntaxbaum für Array Initialisierung

Bei der **Initialisierung** eines Arrays wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann³³. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 3.23 dargestellt ist.

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

³³Das widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.

Probleme macht, dass ein **Stackframe** in die entgegengesetzte Richtung wächst, verglichen mit den **Globalen Statischen Daten**³⁴.

Das **Größe** des Arrays datatype `ar[dim1]...[dimk]`, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Arrays multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Arrayelemente**: $\text{size}(\text{datatype}(\text{ar})) = \left(\prod_{j=1}^n \text{dim}_j\right) \cdot \text{size}(\text{datatype})^a$.

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

```

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

```

Code 3.23: *Symboltabelle für Array Initialisierung*

³⁴Wenn man beim **GCC GCC, the GNU Compiler Collection - GNU Project** einen Stackframe mittels des **GDB GCC, the GNU Compiler Collection - GNU Project** beobachtet, sieht man, dass dieser es genauso macht.

Im **Piocc-Mon Pass** in Code 3.24 werden zuerst die **Logischen Ausdrücke** in den Blättern des Teilbaums, der beim **Array-Initializers Container-Knoten** `Array([Array([BinOp(Num('3')), Add('+'), Num('1'))]), Array([Num('4')])])` anfängt nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben³⁵.

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

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

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

```

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      ],
15    Block
16      Name 'fun.0',
17      [
18        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
19        ↪   Num('6')])])
20        Exp(Num('3'))
21        Exp(Num('4'))
22        Exp(Num('5'))
23        Exp(Num('6'))
24        Assign(Stackframe(Num('3')), Stack(Num('4')))
25      ],
26    ]

```

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

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

³⁵Da der **Zuweisungsoperator** = **rechtsassoziativ** ist und auch rein **logisch**, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

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

```

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

```

```

52     SUBI SP 1;
53     LOADI ACC 6;
54     STOREIN SP ACC 1;
55     # Assign(Stackframe(Num('3')), Stack(Num('4')))
56     LOADIN SP ACC 1;
57     STOREIN BAF ACC -2;
58     LOADIN SP ACC 2;
59     STOREIN BAF ACC -3;
60     LOADIN SP ACC 3;
61     STOREIN BAF ACC -4;
62     LOADIN SP ACC 4;
63     STOREIN BAF ACC -5;
64     ADDI SP 4;
65     # Return(Empty())
66     LOADIN BAF PC -1;
67 ]
68 ]

```

Code 3.25: RETI-Blocks Pass für Array Initialisierung

3.3.3.2 Zugriff auf einen Arrayindex

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

```

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

```

Code 3.26: PicoC-Code für Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im **Abstrakter Syntaxbaum** in Code 3.27 mithilfe des **Container-Knotens** `Subscr(Name('ar'), Num('0'))` dargestellt.

```

1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
10              ↪ Array([Num('42')]))

```



```

11     ],
12     FunDef
13     VoidType 'void',
14     Name 'fun',
15     [],
16     [
17         Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
18             ↪ Array([Num('1'), Num('2'), Num('3')]))
19         Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
20     ]

```

Code 3.27: Abstrakter Syntaxbaum für Zugriff auf einen Arrayindex

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

In nächsten Schritt, dem **Mittelteil 3.3.5.3** wird die **Adresse** ab der das **Arrayelement** des Arrays auf das Zugriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Arrays, in dem dieses **Arrayelement** liegt auf den **Stack** gelegt. Da ein **Index** auf den 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 3.8 dokumentiert.

Zur **Adressberechnung** ist es notwendig auf die **Dimensionen** (z.B. `[Num('3')]`) des Arrays, auf dessen **Arrayelement** zugegriffen wird, zugreifen zu können. Daher ist der **Arraydatentyp** (z.B. `ArrayDecl([Num('3')], IntType('int'))`) dem **Container-Knoten** `Ref(exp, datatype)` als verstecktes Attribut `datatype` angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im **Piocc-Mon Pass** dem **Container-Knoten** `Ref(exp, datatype)` angehängt.

Je nachdem, ob mehrere `Subscr(exp, exp)` eine Komposition bilden (z.B. `Subscr(Subscr(Name('var'), Num('1')), Num('1'))`) ist es notwendig mehrere **Adressberechnungsschritte für den Index** `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` einzuleiten und es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und eine **Zugriff auf einen Arrayindex** `var[1]` miteinander zu kombinieren, was in Subkapitel 3.3.5.3 allgemein erklärt ist.

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

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Arrayindex** (z.B. `ar[1]`) in der `main`-

Funktion oder der Funktion `fun` erfolgt, ist eigentlich nur beim **Anfangsteil**, beim Schreiben der **Adresse** der Variable `ar` auf den **Stack** zu finden, bei dem unterschiedliche **RETI-Instructions** für eine Variable, die in den **Globalen Statischen Daten** liegt und eine Variable, die auf dem **Stackframe** liegt erzeugt werden müssen.

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays `datatype ar[dim1]...[dimn]` abgespeichert ist, kann mittels der Formel 3.3.1:

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

aus der Betriebssysteme Vorlesung^a berechnet werden^b.

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

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

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

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** `M[ref(ar[idx1]...[idxn])]` der Speicherzelle an der finalen **Adresse** `ref(ar[idx1]...[idxn])`.

^aP. D. C. Scholl, „Betriebssysteme“.

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

```

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

```

```

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

```

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

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

```

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

```

```

39 ],
40 Block
41   Name 'fun.0',
42   [
43     # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44     # Exp(Num('1'))
45     SUBI SP 1;
46     LOADI ACC 1;
47     STOREIN SP ACC 1;
48     # Exp(Num('2'))
49     SUBI SP 1;
50     LOADI ACC 2;
51     STOREIN SP ACC 1;
52     # Exp(Num('3'))
53     SUBI SP 1;
54     LOADI ACC 3;
55     STOREIN SP ACC 1;
56     # Assign(Stackframe(Num('2')), Stack(Num('3')))
57     LOADIN SP ACC 1;
58     STOREIN BAF ACC -2;
59     LOADIN SP ACC 2;
60     STOREIN BAF ACC -3;
61     LOADIN SP ACC 3;
62     STOREIN BAF ACC -4;
63     ADDI SP 3;
64     # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65     # Ref(Stackframe(Num('2')))
66     SUBI SP 1;
67     MOVE BAF IN1;
68     SUBI IN1 4;
69     STOREIN SP IN1 1;
70     # Exp(Num('1'))
71     SUBI SP 1;
72     LOADI ACC 1;
73     STOREIN SP ACC 1;
74     # Exp(Num('1'))
75     SUBI SP 1;
76     LOADI ACC 1;
77     STOREIN SP ACC 1;
78     # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79     LOADIN SP ACC 2;
80     LOADIN SP IN2 1;
81     ADD ACC IN2;
82     STOREIN SP ACC 2;
83     ADDI SP 1;
84     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85     LOADIN SP IN1 2;
86     LOADIN SP IN2 1;
87     MULTI IN2 1;
88     ADD IN1 IN2;
89     ADDI SP 1;
90     STOREIN SP IN1 1;
91     # Exp(Stack(Num('1')))
92     LOADIN SP IN1 1;
93     LOADIN IN1 ACC 0;
94     STOREIN SP ACC 1;
95     # Return(Empty())

```

```

96         LOADIN BAF PC -1;
97     ]
98 ]

```

Code 3.29: *RETI-Blocks Pass für Zugriff auf einen Arrayindex*

3.3.3.3 Zuweisung an Arrayindex

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

```

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

```

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

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

```

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

```

Code 3.31: *Abstrakter Syntaxbaum für Zuweisung an Arrayindex*

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

Danach ist das Vorgehen, bzw. sind die Kompositionen, die dieses darauffolgende Vorgehen darstellen: `Ref(Global(Num('0'))), Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum **Anfangsteil** und **Mittelteil** aus dem vorherigen Subkapitel 3.3.3.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird berechnet, wie in Subkapitel 3.3.3.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`³⁶ die Zuweisung `=` des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Arrayindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

³⁶Ist in Tabelle 3.8 genauer beschrieben ist

```

1 File
2   Name './example_array_assignment.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
8         Exp(Num('42'))
9         Ref(Global(Num('0')))
10        Exp(Num('2'))
11        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
12        Assign(Stack(Num('1')), Stack(Num('2')))
13        Return(Empty())
14      ]
15    ]

```

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

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

```

1 File
2   Name './example_array_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Ref(Global(Num('0')))
13        SUBI SP 1;
14        LOADI IN1 0;
15        ADD IN1 DS;
16        STOREIN SP IN1 1;
17        # Exp(Num('2'))
18        SUBI SP 1;
19        LOADI ACC 2;
20        STOREIN SP ACC 1;
21        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22        LOADIN SP IN1 2;
23        LOADIN SP IN2 1;
24        MULTI IN2 1;
25        ADD IN1 IN2;
26        ADDI SP 1;
27        STOREIN SP IN1 1;
28        # Assign(Stack(Num('1')), Stack(Num('2')))
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 3.33: *RETI-Blocks Pass für Zuweisung an Arrayindex*

3.3.4 Umsetzung von Structs

3.3.4.1 Deklaration und Definition von Structtypen

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

```

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

Code 3.34: PicoC-Code für die Deklaration eines Structtyps

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

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

```

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

Code 3.35: Abstrakter Syntaxbaum für die Deklaration eines Structtyps

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

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

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

Für die **Definition** einer Variable `st_var@main` mit diesem **Structtyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatype`-Attribut enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structtyp**³⁷ und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Die **Größe** einer Variable `st_var`, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st {datatype1 attr1; ... datatypen attrn;}`^a definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** `datatype1 ... datatypen` der **Attribute** `attr1, ... attrn` des **Structtyps**: $size(st) = \sum_{i=1}^n size(datatype_i)$.

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

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type_qualifier:      Empty()
6       datatype:            IntType('int')
7       name:                Name('len@st')
8       value_or_address:    Empty()
9       position:            Pos(Num('1'), Num('15'))
10      size:                 Num('1')
11    },
12    Symbol
13    {
14      type_qualifier:      Empty()
15      datatype:            ArrayDecl([Num('2')], IntType('int'))
16      name:                Name('ar@st')
17      value_or_address:    Empty()
18      position:            Pos(Num('1'), Num('24'))
19      size:                 Num('2')
20    },
21    Symbol
22    {
23      type_qualifier:      Empty()
24      datatype:            StructDecl(Name('st'), [Alloc(Writable(), IntType('int'),
25      ↪ Name('len'))Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')),
26      ↪ Name('ar'))])
27      name:                Name('st')
28      value_or_address:    [Name('len@st'), Name('ar@st')]
29      position:            Pos(Num('1'), Num('7'))
30      size:                 Num('3')

```

³⁷Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Structtyp** einnimmt.

```

29     },
30     Symbol
31     {
32         type qualifier:      Empty()
33         datatype:            FunDecl(VoidType('void'), Name('main'), [])
34         name:                 Name('main')
35         value or address:     Empty()
36         position:             Pos(Num('3'), Num('5'))
37         size:                 Empty()
38     },
39     Symbol
40     {
41         type qualifier:      Writeable()
42         datatype:            StructSpec(Name('st'))
43         name:                 Name('st_var@main')
44         value or address:     Num('0')
45         position:             Pos(Num('4'), Num('12'))
46         size:                 Num('3')
47     }
48 ]

```

Code 3.36: Symboltabelle für die Deklaration eines Structtyps

3.3.4.2 Initialisierung von Structs

Die **Initialisierung eines Structs** wird im Folgenden mithilfe des Beispiels in Code 3.37 erklärt.

```

1 struct st1 {int *attr[2];};
2
3 struct st2 {int attr1; struct st1 attr2;};
4
5 void main() {
6     int var = 42;
7     struct st2 st = {.attr1=var, .attr2={.attr={{&var, &var}}}};
8 }

```

Code 3.37: PicoC-Code für Initialisierung von Structs

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

```

1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], PtrDecl(Num('1'), IntType('int'))),
7         ↪ Name('attr'))
8     ],

```

```

9      StructDecl
10      Name 'st2',
11      [
12          Alloc(Writable(), IntType('int'), Name('attr1'))
13          Alloc(Writable(), StructSpec(Name('st1')), Name('attr2'))
14      ],
15      FunDef
16      VoidType 'void',
17      Name 'main',
18      [],
19      [
20          Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
21          Assign(Alloc(Writable(), StructSpec(Name('st2')), Name('st')),
22              ↳ Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
23              ↳ Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')),
24              ↳ Ref(Name('var'))]))]))]))]))))
25      ]
26  ]

```

Code 3.38: Abstrakter Syntaxbaum für Initialisierung von Structs

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

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

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

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

```

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

```

```

7      // Assign(Name('var'), Num('42'))
8      Exp(Num('42'))
9      Assign(Global(Num('0')), Stack(Num('1')))
10     // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
    ↪ Assign(Name('attr2'), Struct([Assign(Name('attr'),
    ↪ Array([Array([Ref(Name('var')), Ref(Name('var'))]))]))]))
11     Exp(Global(Num('0')))
12     Ref(Global(Num('0')))
13     Ref(Global(Num('0')))
14     Assign(Global(Num('1')), Stack(Num('3')))
15     Return(Empty())
16 ]
17 ]

```

Code 3.39: PicoC-ANF Pass für Initialisierung von Structs

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

```

1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
    ↪ Assign(Name('attr2'), Struct([Assign(Name('attr'),
    ↪ Array([Array([Ref(Name('var')), Ref(Name('var'))]))]))]))
17        # Exp(Global(Num('0')))
18        SUBI SP 1;
19        LOADIN DS ACC 0;
20        STOREIN SP ACC 1;
21        # Ref(Global(Num('0')))
22        SUBI SP 1;
23        LOADI IN1 0;
24        ADD IN1 DS;
25        STOREIN SP IN1 1;
26        # Ref(Global(Num('0')))
27        SUBI SP 1;
28        LOADI IN1 0;
29        ADD IN1 DS;
30        STOREIN SP IN1 1;
31        # Assign(Global(Num('1')), Stack(Num('3')))
32        LOADIN SP ACC 1;
33        STOREIN DS ACC 3;

```

```

34     LOADIN SP ACC 2;
35     STOREIN DS ACC 2;
36     LOADIN SP ACC 3;
37     STOREIN DS ACC 1;
38     ADDI SP 3;
39     # Return(Empty())
40     LOADIN BAF PC -1;
41 ]
42 ]

```

Code 3.40: RETI-Blocks Pass für Initialisierung von Structs

3.3.4.3 Zugriff auf Structattribut

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

```

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

```

Code 3.41: PicoC-Code für Zugriff auf Structattribut

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

```

1 File
2   Name './example_struct_attr_access.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc(Writable(), IntType('int'), Name('x'))
8         Alloc(Writable(), IntType('int'), Name('y'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Assign(Alloc(Writable(), StructSpec(Name('pos')), Name('st')),
16              ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
17        Exp(Attr(Name('st'), Name('y')))
18      ]
19    ]

```

Code 3.42: Abstrakter Syntaxbaum für Zugriff auf Structattribut

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

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

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

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

Sei datatype_i ein **Knoten** eines **entarteten Baumes** (siehe Definition 3.12 und Abbildung 3.3.2), dessen Wurzel datatype_1 ist. Dabei steht i für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\text{ref}(\text{datatype}_i)$ von Speicherbereichen $\text{ref}(\text{datatype}_i) \dots \text{ref}(\text{datatype}_i) + \text{size}(\text{datatype}_i)$ im **Hauptspeicher** zuordnen, wobei gilt, dass $\text{ref}(\text{datatype}_i) \leq \text{ref}(\text{datatype}_{i+1}) < \text{ref}(\text{datatype}_i) + \text{size}(\text{datatype}_i)$.^{ab}

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

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



Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen $(\text{datatype}_{1,\text{idx}_1}, \dots, \text{datatype}_{n,\text{idx}_n})$, die das Resultat einer Aneinandereiung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen

datatype_i ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 3.3.3:

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

berechnet werden.^c

Dabei darf nur der letzte Knoten datatype_n vom Datentyp **Pointer** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, der nicht der **letzte Datentyp** datatype_n in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp datatype_1 bis direkt zum Datentyp **Pointer** geht $\text{datatype}_{\text{pntr}}$ und die **zweite Adressberechnung** einen Datentyp nach dem Datentyp **Pointer** anfängt $\text{datatype}_{\text{pntr}+1}$ und bis zum letzten Datentyp datatype_n geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\text{ref}(\text{datatype}_1)$ des Summanden aus der Formel 3.3.3 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $M[\text{ref}(\text{datatype}_1, \dots, \text{datatype}_{\text{pntr}})]$ gesetzt werden.

Die Formel 3.3.3 stellt dabei eine **Verallgemeinerung** der Formel 3.3.1 dar, die für alle möglichen Aneinanderreihungen von Zugriffen auf **Pointerelemente**, **Arrayelementen** und **Structattribute** funktioniert (z.B. `(*complex_var.attr2)[3]`). Da die Formel **allgemein** sein muss, lässt sie sich nicht so elegant mit einem Produkt \prod schreiben, wie die Formel 3.3.1, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben^d.

Die Komposition $\text{Exp}(\text{Attr}(\text{Stack}(\text{Num}('1')), \text{name}))$ repräsentiert dabei den Summanden $\text{ref}(\text{datatype}_1)$ in der Formel.

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

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

^aEs ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.

^b $\text{ref}(\text{datatype})$ steht dabei für das Schreiben der **Startadresse**, die dem **Datentyp** datatype zugeordnet ist auf den **Stack**.

^cDie **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Pointerelemente**, **Arrayelemente** oder **Structattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** datatype_i , die vor dem **Element** / **Attribut** $\text{datatype}_{i,\text{idx}_i}$ liegen.

^dStructattribute haben **unterschiedliche** Größen.

Definition 3.12: Entarteter Baum

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

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

Der Baum entspricht also einer **verketteten Liste**.^a

^aBäume.

```
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 3.43: PicoC-ANF Pass für Zugriff auf Structattribut

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

```

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;
29        # Ref(Attr(Stack(Num('1')), Name('y')))
30        LOADIN SP IN1 1;
31        ADDI IN1 1;

```



```

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

```

Code 3.44: RETI-Blocks Pass für Zugriff auf Structattribut

3.3.4.4 Zuweisung an Structattribut

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

```

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

```

Code 3.45: PicoC-Code für Zuweisung an Structattribut

Im **Abstract Syntax Tree** wird eine **Zuweisung an ein Structattribut** (z.B. `st.y = 42`) durch die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` dargestellt.

```

1 File
2   Name './example_struct_attr_assignment.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc(Writable(), IntType('int'), Name('x'))
8         Alloc(Writable(), IntType('int'), Name('y'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Assign(Alloc(Writable(), StructSpec(Name('pos')), Name('st')),
16              ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
17        Assign(Attr(Name('st'), Name('y')), Num('42'))
18      ]
19    ]

```

Code 3.46: Abstrakter Syntaxbaum für Zuweisung an Structattribut

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

Im Gegensatz zum Vorgehen in Unterkapitel 3.3.3.3 muss hier für das Auswerten des **linken** Container-Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Subkapitel 3.3.4.3 vorgegangen werden.

```

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      ]
18    ]

```

Code 3.47: *PicoC-ANF Pass für Zuweisung an Structattribut*

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

```

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')))

```

```

17      LOADIN SP ACC 1;
18      STOREIN DS ACC 1;
19      LOADIN SP ACC 2;
20      STOREIN DS ACC 0;
21      ADDI SP 2;
22      # // Assign(Attr(Name('st'), Name('y')), Num('42'))
23      # Exp(Num('42'))
24      SUBI SP 1;
25      LOADI ACC 42;
26      STOREIN SP ACC 1;
27      # Ref(Global(Num('0')))
28      SUBI SP 1;
29      LOADI IN1 0;
30      ADD IN1 DS;
31      STOREIN SP IN1 1;
32      # Ref(Attr(Stack(Num('1')), Name('y')))
33      LOADIN SP IN1 1;
34      ADDI IN1 1;
35      STOREIN SP IN1 1;
36      # Assign(Stack(Num('1')), Stack(Num('2')))
37      LOADIN SP IN1 1;
38      LOADIN SP ACC 2;
39      ADDI SP 2;
40      STOREIN IN1 ACC 0;
41      # Return(Empty())
42      LOADIN BAF PC -1;
43  ]
44 ]

```

Code 3.48: RETI-Blocks Pass für Zuweisung an Structattribut

3.3.5 Umsetzung des Zugriffs auf Derived datatypes im Allgemeinen

3.3.5.1 Übersicht

In den Unterkapiteln 3.3.2, 3.3.3 und 3.3.4 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln beschriebenen Datentypen (**Pointer**, **Array** und **Struct**) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem **Anfangsteil**, **Mittelteil** und **Schlusssteil** darin erkennen.

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

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

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

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

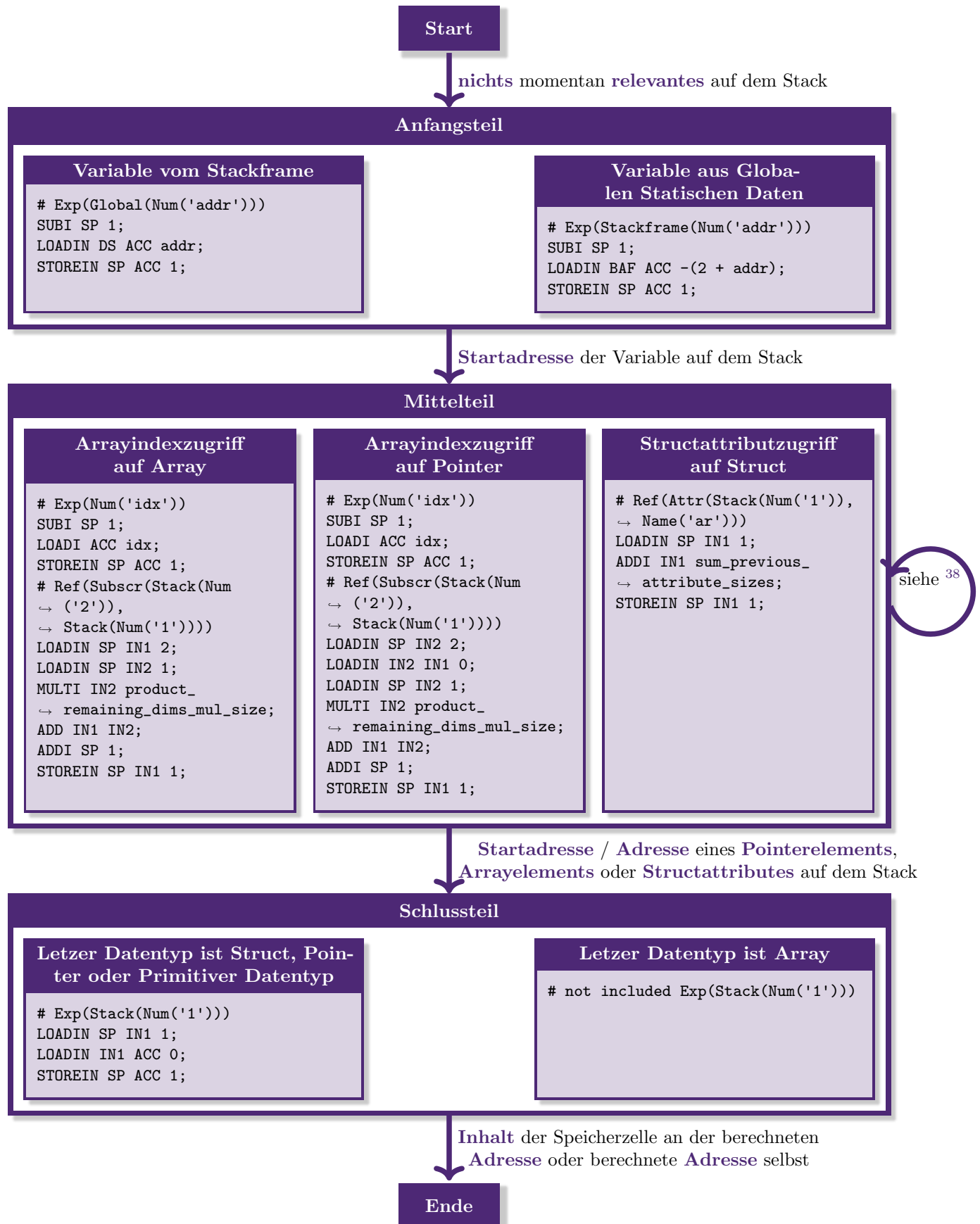


Abbildung 3.9: Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

es eine **DatatypeMismatch-Fehlermeldung**. Ein **Zugriff auf ein Arrayindex** `Subscr(exp1, exp2)` kann dabei mit den Datentypen **Array** `ArrayDecl(nums, datatype)` und **Pointer** `PntrDecl(num, datatype)` kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Pointer** `PntrDecl(num, datatype)` die **Adresse**, die auf dem **Stack** liegt auf eine Speicherzelle mit einer weiteren **Adresse** zeigt und das gewünschte Element erst zu finden ist, wenn man der letzteren **Adresse** folgt. Ein **Zugriff auf ein Structattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.

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

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

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

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

Arrays haben in der Sprache L_C und somit auch in L_{PicoC} die Eigenheit, dass wenn auf ein gesamtes **Array** zugegriffen wird³⁹, die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache L_{PicoC} implementieren Datentypen wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

Implementieren lässt sich dieses Vorgehen, indem beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)` Ausdrucks ein `Exp(Stack(Num('1')))` an die Spitze einer **Liste der generierten Ausdrücke** gesetzt wird und der Ausdruck selbst als `exp`-Attribut des `Ref(exp)`-Knotens gesetzt wird und hinter dem `Exp(Stack(Num('1')))`-Container-Knoten in der Liste eingefügt wird. Beim Antreffen eines `Ref(exp)` wird fast gleich vorgegangen, wie beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)`, nur, dass kein `Exp(Stack(Num('1')))` vorne an die Spitze der **Liste der generierten Ausdrücke** gesetzt wird. Und ein `Ref(exp)` bei dem `exp` direkt ein `Name(str)` ist, wird dieser einfach direkt durch `Ref(Global(num))` bzw. `Ref(Stackframe(num))` ersetzt.

³⁹Und nicht auf ein **Element** des Arrays.

³⁹**Startadresse** / **Adresse** eines **Pointerelements**, **Arrayelements** oder **Structattributes** auf dem **Stack**.

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

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

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

3.3.5.2 Anfangsteil

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

```

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

Code 3.49: PicoC-Code für den Anfangsteil

Im **Abstrakter Syntaxbaum** in Code 3.50 wird die **Referenzierung** `&complex_var` mit der Komposition `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird aber einfach nur `Ref(Name('complex_var'))` geschrieben, aber da beim Erstellen des **Abstract Syntax Tree** jeder **Logischer Ausdruck** in ein `Exp(exp)` eingebettet wird, ist das `Ref(Name('complex_var'))` in ein `Exp()` eingebettet. Man müsste an vielen Stellen eine gesonderte **Fallunterscheidung** aufstellen, um von `Exp(Ref(Name('complex_var')))` das `Exp()` zu entfernen,

obwohl das `Exp()` in den darauffolgenden **Passes** so oder so herausgefiltert wird. Daher wurde darauf verzichtet den Code ohne triftigen Grund **komplexer** zu machen.

```

1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     StructDecl
5       Name 'ar_with_len',
6       [
7         Alloc(Writable(), IntType('int'), Name('len'))
8         Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
16          ↪ Name('st_ar')))
17        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
18          ↪ IntType('int'))), Name('complex_var')))
19        Exp(Ref(Name('complex_var')))
20      ],
21    FunDef
22      VoidType 'void',
23      Name 'fun',
24      [],
25      [
26        Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
27          ↪ Name('st_ar')))
28        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
29          ↪ Name('complex_var')))
30        Exp(Ref(Name('complex_var')))
31      ]
32    ]
33  ]

```

Code 3.50: Abstrakter Syntaxbaum für den Anfangsteil

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

```

1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Exp(Ref(Name('complex_var')))
8         Ref(Global(Num('9')))
9         Return(Empty())
10      ],
11    Block

```

```

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

```

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

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

```

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

```

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

3.3.5.3 Mittelteil

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


```

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

```

Code 3.53: PicoC-Code für den Mittelteil

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

```

1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
9           ↪ Name('ar'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14        Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
16          ↪ Array([Num('42')]))
15        Assign(Alloc(Writable(), StructSpec(Name('st')), Name('complex_var')),
17          ↪ Struct([Assign(Name('ar'), Ref(Name('var')))]))
18        Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
19          ↪ Sub('-', Num('2')))))
20      ]
21  ]

```

Code 3.54: Abstrakter Syntaxbaum für den Mittelteil

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

```

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('complex_var'), 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('complex_var'), Name('ar')), Num('0')),
14        ↪   BinOp(Num('2'), Sub('-',), Num('2'))))
15        Ref(Global(Num('1')))
16        Ref(Attr(Stack(Num('1')), Name('ar')))
17        Exp(Num('0'))
18        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
19        Exp(Num('2'))
20        Exp(Num('2'))
21        Exp(BinOp(Stack(Num('2')), Sub('-',), Stack(Num('1'))))
22        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23        Exp(Stack(Num('1')))
24        Return(Empty())
25      ]
26    ]

```

Code 3.55: PicoC-ANF Pass für den Mittelteil

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

```

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('complex_var'), 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('complex_var'), Name('ar')), Num('0')),
    ↪   BinOp(Num('2'), Sub('-'), Num('2'))))
27     # Ref(Global(Num('1')))
28     SUBI SP 1;
29     LOADI IN1 1;
30     ADD IN1 DS;
31     STOREIN SP IN1 1;
32     # Ref(Attr(Stack(Num('1')), Name('ar')))
33     LOADIN SP IN1 1;
34     ADDI IN1 0;
35     STOREIN SP IN1 1;
36     # Exp(Num('0'))
37     SUBI SP 1;
38     LOADI ACC 0;
39     STOREIN SP ACC 1;
40     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41     LOADIN SP IN2 2;
42     LOADIN IN2 IN1 0;
43     LOADIN SP IN2 1;
44     MULTI IN2 1;
45     ADD IN1 IN2;
46     ADDI SP 1;
47     STOREIN SP IN1 1;
48     # Exp(Num('2'))
49     SUBI SP 1;
50     LOADI ACC 2;
51     STOREIN SP ACC 1;
52     # Exp(Num('2'))
53     SUBI SP 1;
54     LOADI ACC 2;
55     STOREIN SP ACC 1;
56     # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
57     LOADIN SP ACC 2;
58     LOADIN SP IN2 1;
59     SUB ACC IN2;
60     STOREIN SP ACC 2;
61     ADDI SP 1;
62     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63     LOADIN SP IN1 2;
64     LOADIN SP IN2 1;
65     MULTI IN2 1;
66     ADD IN1 IN2;
67     ADDI SP 1;
68     STOREIN SP IN1 1;
69     # Exp(Stack(Num('1')))
70     LOADIN SP IN1 1;
71     LOADIN IN1 ACC 0;
72     STOREIN SP ACC 1;
73     # Return(Empty())

```

```

74     LOADIN BAF PC -1;
75 ]
76 ]

```

Code 3.56: RETI-Blocks Pass für den Mittelteil

3.3.5.4 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 3.3.5.2 und **Mittelteil** 3.3.5.3 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird⁴⁰, wird im Folgenden mithilfe des Beispiels in Code 3.57 erklärt.

```

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

```

Code 3.57: PicoC-Code für den Schlussteil

Das Generieren des **Abstrakter Syntaxbaum** in Code 3.58 verläuft wie üblich.

```

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

```

⁴⁰Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

```

17     Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('ptr1')),
18           ↪ Ref(Name('var')))
19     Assign(Alloc(Writable(), PtrDecl(Num('2'), IntType('int')), Name('complex_var3')),
20           ↪ Ref(Name('ptr1')))
21     Exp(Subscr(Name('complex_var1'), Num('0')))
22     Exp(Subscr(Name('complex_var2'), Num('0')))
23     Exp(Deref(Name('complex_var3'), Num('0')))
24 ]
25 ]

```

Code 3.58: Abstrakter Syntaxbaum für den Schlussteil

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

```

1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('4')), Stack(Num('1')))
10        // Assign(Name('ptr1'), Ref(Name('var')))
11        Ref(Global(Num('4')))
12        Assign(Global(Num('5')), Stack(Num('1')))
13        // Assign(Name('complex_var3'), Ref(Name('ptr1')))
14        Ref(Global(Num('5')))
15        Assign(Global(Num('6')), Stack(Num('1')))
16        // Exp(Subscr(Name('complex_var1'), Num('0')))
17        Ref(Global(Num('0')))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        // Exp(Subscr(Name('complex_var2'), Num('0')))
22        Ref(Global(Num('2')))
23        Exp(Num('0'))
24        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25        Exp(Stack(Num('1')))
26        // Exp(Subscr(Name('complex_var3'), Num('0')))
27        Ref(Global(Num('6')))
28        Exp(Num('0'))
29        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30        Exp(Stack(Num('1')))
31        Return(Empty())
32      ]
33    ]

```

Code 3.59: PicoC-ANF Pass für den Schlussteil

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

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

```

1 File
2   Name './example_derived_dts_final_part.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('4')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 4;
15        ADDI SP 1;
16        # // Assign(Name('pntr1'), Ref(Name('var')))
17        # Ref(Global(Num('4')))
18        SUBI SP 1;
19        LOADI IN1 4;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('5')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 5;
25        ADDI SP 1;
26        # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
27        # Ref(Global(Num('5')))
28        SUBI SP 1;
29        LOADI IN1 5;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Assign(Global(Num('6')), Stack(Num('1')))
33        LOADIN SP ACC 1;
34        STOREIN DS ACC 6;
35        ADDI SP 1;
36        # // Exp(Subscr(Name('complex_var1'), Num('0')))
37        # Ref(Global(Num('0')))
38        SUBI SP 1;
39        LOADI IN1 0;
40        ADD IN1 DS;
41        STOREIN SP IN1 1;
42        # Exp(Num('0'))
43        SUBI SP 1;
44        LOADI ACC 0;
45        STOREIN SP ACC 1;
46        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47        LOADIN SP IN1 2;
48        LOADIN SP IN2 1;
49        MULTI IN2 2;
50        ADD IN1 IN2;
51        ADDI SP 1;
52        STOREIN SP IN1 1;

```

```

53      # // not included Exp(Stack(Num('1')))
54      # // Exp(Subscr(Name('complex_var2'), Num('0')))
55      # Ref(Global(Num('2')))
56      SUBI SP 1;
57      LOADI IN1 2;
58      ADD IN1 DS;
59      STOREIN SP IN1 1;
60      # Exp(Num('0'))
61      SUBI SP 1;
62      LOADI ACC 0;
63      STOREIN SP ACC 1;
64      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65      LOADIN SP IN1 2;
66      LOADIN SP IN2 1;
67      MULTI IN2 2;
68      ADD IN1 IN2;
69      ADDI SP 1;
70      STOREIN SP IN1 1;
71      # Exp(Stack(Num('1')))
72      LOADIN SP IN1 1;
73      LOADIN IN1 ACC 0;
74      STOREIN SP ACC 1;
75      # // Exp(Subscr(Name('complex_var3'), Num('0')))
76      # Ref(Global(Num('6')))
77      SUBI SP 1;
78      LOADI IN1 6;
79      ADD IN1 DS;
80      STOREIN SP IN1 1;
81      # Exp(Num('0'))
82      SUBI SP 1;
83      LOADI ACC 0;
84      STOREIN SP ACC 1;
85      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86      LOADIN SP IN2 2;
87      LOADIN IN2 IN1 0;
88      LOADIN SP IN2 1;
89      MULTI IN2 1;
90      ADD IN1 IN2;
91      ADDI SP 1;
92      STOREIN SP IN1 1;
93      # Exp(Stack(Num('1')))
94      LOADIN SP IN1 1;
95      LOADIN IN1 ACC 0;
96      STOREIN SP ACC 1;
97      # Return(Empty())
98      LOADIN BAF PC -1;
99  ]
100 ]

```

Code 3.60: RETI-Blocks Pass für den Schlussteil

3.3.6 Umsetzung von Funktionen

3.3.6.1 Mehrere Funktionen

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

```

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

```

Code 3.61: *PicoC-Code für 3 Funktionen*

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

```

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

```


Code 3.62: *Abstrakter Syntaxbaum für 3 Funktionen*

Im **PicoC-Blocks Pass** in Code 3.63 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`⁴¹ beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`⁴².

Es werden parallel dazu neue Zuordnungen im **Assoziativen Feld** `fun_name_to_block_name` hinzugefügt. Das **Dicionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu⁴³. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```

1 File
2   Name './verbose_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 ]

```

⁴¹Eine Erklärung dazu ist in Unterkapitel 3.3.1.2.1 zu finden.

⁴²Der **Grund** dafür kann im Unterkapitel 3.3.1.2.1 nachgelesen werden.

⁴³Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

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

Im **PicoC-ANF Pass** in Code 3.64 werden die `FunDef(datatype, name, allocs, stmts)`-Container-Knoten komplett aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Container-Knoten nur noch Blöcke befinden.

```

1 File
2   Name './verbose_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 3.64: *PicoC-ANF Pass für 3 Funktionen*

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

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

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.2'))))
3 # // not included Exp(GoTo(Name('main.2'))))
4 # // Block(Name('main.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.1'), [])
8 # Return(Empty())

```

```

9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1'))))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 LOADIN BAF PC -1;

```

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

3.3.6.1.1 Sprung zur Main Funktion

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

Im Beispiel in Code 3.66 ist die `main`-Funktion allerdings **nicht** die **erste** Funktion. Daher muss dafür gesorgt werden, dass die `main`-Funktion die erste Funktion ist, die ausgeführt wird.

```

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

```

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

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

```

1 File
2   Name './verbose_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

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

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

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

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

```

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

```

```

15     ],
16     Block
17     Name 'fun2.1',
18     [
19         # // Return(Num('1'))
20         # Exp(Num('1'))
21         SUBI SP 1;
22         LOADI ACC 1;
23         STOREIN SP ACC 1;
24         # Return(Stack(Num('1')))
25         LOADIN SP ACC 1;
26         ADDI SP 1;
27         LOADIN BAF PC -1;
28     ],
29     Block
30     Name 'main.0',
31     [
32         # Return(Empty())
33         LOADIN BAF PC -1;
34     ]
35 ]

```

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

Im **RETI Pass** in Code 3.69 wird das `GoTo(Name('main.<nummer>'))` durch den entsprechenden Sprung `JUMP <distanz_zur_main_funktion>` ersetzt und die Blöcke entfernt.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.0'))))
3 JUMP 8;
4 # // Block(Name('fun1.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun2.1'), [])
8 # // Return(Num('1'))
9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Return(Stack(Num('1')))
14 LOADIN SP ACC 1;
15 ADDI SP 1;
16 LOADIN BAF PC -1;
17 # // Block(Name('main.0'), [])
18 # Return(Empty())
19 LOADIN BAF PC -1;

```

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

3.3.6.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

In der Programmiersprache L_C und somit auch L_{PicoC} ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehler-**

meldungen ausgegeben werden können, wenn der **Prototyp** (Definition 3.13) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

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

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

Definition 3.13: Funktionsprototyp

*Deklaration einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parameterreihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktionendeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.^{a,b}*

^aDer **Funktionsprototyp** ist von der **FunktionsSignatur** zu unterscheiden, die in Programmiersprache wie C++ und Java für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentypen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentypen** usw. unterscheiden.

^bWhat is the difference between function prototype and function signature?

```

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

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

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

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 3.14). Die **Scopes** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**⁴⁴ Durchhiterieren über den **Abstrakter Syntaxbaum** des aktuellen **Passes** ein **Funktionswechsel**

⁴⁴D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

eintritt und über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

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

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

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

Definition 3.14: Scope (bzw. Sichtbarkeitsbereich)

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

^aThiemann, „Einführung in die Programmierung“.

```

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'), [])

```

```

25     name:                Name('main')
26     value or address:    Empty()
27     position:            Pos(Num('6'), Num('5'))
28     size:                Empty()
29 },
30 Symbol
31 {
32     type qualifier:      Writeable()
33     datatype:            IntType('int')
34     name:                Name('var@main')
35     value or address:    Num('0')
36     position:            Pos(Num('7'), Num('6'))
37     size:                Num('1')
38 },
39 Symbol
40 {
41     type qualifier:      Writeable()
42     datatype:            IntType('int')
43     name:                Name('var@fun2')
44     value or address:    Num('0')
45     position:            Pos(Num('12'), Num('13'))
46     size:                Num('1')
47 }
48 ]

```

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

3.3.6.3 Funktionsaufruf

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

```

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

```

Code 3.72: PicoC-Code für Funktionsaufruf ohne Rückgabewert

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


```

1 File
2   Name './example_fun_call_no_return_value.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writable(), IntType('int'), Name('attr1'))
8         Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9       ],
10    FunDecl
11      VoidType 'void',
12      Name 'stack_fun',
13      [
14        Alloc
15          Writable,
16          ArrayDecl
17            [
18              Num '2',
19              Num '3'
20            ],
21          StructSpec
22            Name 'st',
23            Name 'param'
24        ],
25      FunDef
26        VoidType 'void',
27        Name 'main',
28        [],
29        [
30          Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
31              ↪ Name('local_var')))
32          Exp(Call(Name('stack_fun'), [Name('local_var')]))
33          Return(Empty())
34        ],
35      FunDef
36        VoidType 'void',
37        Name 'stack_fun',
38        [
39          Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
40              ↪ Name('param'))
41        ],
42        [
43          Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
44        ]
45      ]
46    ]
47  ]

```

Code 3.73: Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert

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

Der Container-Knoten `StackMalloc(Num('2'))` ist notwendig, weil auf dem **Stackframe** für den Wert des BAF-Registers der **aufrufenden Funktion** und die **Rücksprungadresse** 2 Speicherzellen Platz am **Anfang** des **Stackframes** gelassen werden muss. Das wird durch den Container-Knoten `StackMalloc(Num('2'))` umgesetzt,

indem das SP-Register einfach um zwei Speicherzellen **dekrementiert** wird und somit Speicher auf dem **Stack** belegt wird⁴⁵.

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc(Num('2'))
8         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
10        Exp(GoTo(Name('stack_fun.0')))
11        RemoveStackframe()
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]

```

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

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

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

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Assoziativen Feld** `fun_name.to.block_name`⁴⁶ mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Container-Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```

1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # StackMalloc(Num('2'))
8         SUBI SP 2;
9         # Ref(Global(Num('0')))
10        SUBI SP 1;
11        LOADI IN1 0;
12        ADD IN1 DS;
13        STOREIN SP IN1 1;

```

⁴⁵Wobei hier "reserviert" besser passen würde.

⁴⁶Dieses Assoziative Feld wurde in Unterkapitel 3.3.6.1 eingeführt.

```

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

```

Code 3.75: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI Pass** in Code 3.75 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist⁴⁷ ersetzt: `LOADI ACC 14`. Und auch der **Container-Knoten**, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den **Container-Knoten** `JUMP 5` ersetzt.

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

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

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

$$adr_{danach} = \#Bef_{vor\ akt. Bl.} + idx + 4 \quad (3.3.1)$$

wobei:

⁴⁷Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

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

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

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

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und zu dem gesprungen werden soll. Es handelt sich hierbei um ein verstecktes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung** wie in der Formel 3.3.1.

```

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

```

```

22 MOVE BAF IN1;
23 LOADIN IN1 BAF 0;
24 MOVE IN1 SP;
25 # Return(Empty())
26 LOADIN BAF PC -1;
27 # Return(Empty())
28 LOADIN BAF PC -1;

```

Code 3.76: RETI-Pass für Funktionsaufruf ohne Rückgabewert

3.3.6.3.1 Rückgabewert

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

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

```

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

```

Code 3.77: PicoC-Code für Funktionsaufruf mit Rückgabewert

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

```

1 File
2   Name './example_fun_call_with_return_value.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'fun_with_return_value',
7       [],
8       [
9         Return(BinOp(Num('21'), Mul('*'), Num('2'))))
10      ],

```

```

11  FunDef
12      VoidType 'void',
13      Name 'fun_no_return_value',
14      [],
15      [
16          Return(Empty())
17      ],
18  FunDef
19      VoidType 'void',
20      Name 'main',
21      [],
22      [
23          Assign(Alloc(Writable(), IntType('int'), Name('var')),
24              ↪ Call(Name('fun_with_return_value'), []))
25          Exp(Call(Name('fun_no_return_value'), []))
26      ]

```

Code 3.78: Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert

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

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

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

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

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

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` kein

`return`-Statement explizit ans Ende geschrieben wird, im **PicoC-ANF Pass** eines hinzugefügt wird in Form der Komposition `Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine **MissingReturn-Fehlermeldung** ausgelöst.

```

1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         Exp(Num('21'))
9         Exp(Num('2'))
10        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
11        Return(Stack(Num('1')))
12      ],
13    Block
14      Name 'fun_no_return_value.1',
15      [
16        Return(Empty())
17      ],
18    Block
19      Name 'main.0',
20      [
21        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
22        StackMalloc(Num('2'))
23        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
24        Exp(GoTo(Name('fun_with_return_value.2')))
25        RemoveStackframe()
26        Exp(ACC)
27        Assign(Global(Num('0')), Stack(Num('1')))
28        StackMalloc(Num('2'))
29        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
30        Exp(GoTo(Name('fun_no_return_value.1')))
31        RemoveStackframe()
32        Return(Empty())
33      ]
34    ]

```

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

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

```

1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
7         # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8         # Exp(Num('21'))

```

```

9      SUBI SP 1;
10     LOADI ACC 21;
11     STOREIN SP ACC 1;
12     # Exp(Num('2'))
13     SUBI SP 1;
14     LOADI ACC 2;
15     STOREIN SP ACC 1;
16     # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17     LOADIN SP ACC 2;
18     LOADIN SP IN2 1;
19     MULT ACC IN2;
20     STOREIN SP ACC 2;
21     ADDI SP 1;
22     # Return(Stack(Num('1')))
23     LOADIN SP ACC 1;
24     ADDI SP 1;
25     LOADIN BAF PC -1;
26 ],
27 Block
28   Name 'fun_no_return_value.1',
29   [
30     # Return(Empty())
31     LOADIN BAF PC -1;
32   ],
33 Block
34   Name 'main.0',
35   [
36     # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
37     # StackMalloc(Num('2'))
38     SUBI SP 2;
39     # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
40     MOVE BAF ACC;
41     ADDI SP 2;
42     MOVE SP BAF;
43     SUBI SP 2;
44     STOREIN BAF ACC 0;
45     LOADI ACC GoTo(Name('addr@next_instr'));
46     ADD ACC CS;
47     STOREIN BAF ACC -1;
48     # Exp(GoTo(Name('fun_with_return_value.2')))
49     Exp(GoTo(Name('fun_with_return_value.2')))
50     # RemoveStackframe()
51     MOVE BAF IN1;
52     LOADIN IN1 BAF 0;
53     MOVE IN1 SP;
54     # Exp(ACC)
55     SUBI SP 1;
56     STOREIN SP ACC 1;
57     # Assign(Global(Num('0')), Stack(Num('1')))
58     LOADIN SP ACC 1;
59     STOREIN DS ACC 0;
60     ADDI SP 1;
61     # StackMalloc(Num('2'))
62     SUBI SP 2;
63     # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
64     MOVE BAF ACC;
65     ADDI SP 2;

```



```

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

```

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

3.3.6.3.2 Umsetzung von Call by Sharing für Arrays

Die **Call by Reference** (Definition 1.5) Übergabe eines Arrays an eine andere Funktion, wird im Folgenden mithilfe des Beispiels in Code 3.81 erklärt.

```

1 void fun_array_from_stackframe(int (*param)[3]) {
2 }
3
4 void fun_array_from_global_data(int param[2][3]) {
5     int local_var[2][3];
6     fun_array_from_stackframe(local_var);
7 }
8
9 void main() {
10     int local_var[2][3];
11     fun_array_from_global_data(local_var);
12 }

```

Code 3.81: *PicoC-Code für Call by Sharing für Arrays*

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

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

```

1 SymbolTable
2 [
3   Symbol
4   {
5     type qualifier:      Empty()
6     datatype:            FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
7     ↪ [Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8     ↪ Name('param'))])
9     name:                Name('fun_array_from_stackframe')
10    value or address:     Empty()
11    position:             Pos(Num('1'), Num('5'))
12    size:                 Empty()
13  },
14  Symbol
15  {
16    type qualifier:      Writable()
17    datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
18    name:                Name('param@fun_array_from_stackframe')
19    value or address:     Num('0')
20    position:            Pos(Num('1'), Num('37'))
21    size:                Num('1')
22  },
23  Symbol
24  {
25    type qualifier:      Empty()
26    datatype:            FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
27    ↪ [Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
28    name:                Name('fun_array_from_global_data')
29    value or address:     Empty()
30    position:            Pos(Num('4'), Num('5'))
31    size:                Empty()
32  },
33  Symbol
34  {
35    type qualifier:      Writable()
36    datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
37    name:                Name('param@fun_array_from_global_data')
38    value or address:     Num('0')
39    position:            Pos(Num('4'), Num('36'))
40    size:                Num('1')
41  },
42  Symbol
43  {
44    type qualifier:      Writable()
45    datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
46    name:                Name('local_var@fun_array_from_global_data')
47    value or address:     Num('6')
48    position:            Pos(Num('5'), Num('6'))
49    size:                Num('6')
50  },
51  Symbol
52  {
53    type qualifier:      Empty()
54    datatype:            FunDecl(VoidType('void'), Name('main'), [])
55    name:                Name('main')
56    value or address:     Empty()
57    position:            Pos(Num('9'), Num('5'))

```

```

55     size:                Empty()
56   },
57   Symbol
58   {
59     type qualifier:      Writeable()
60     datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
61     name:                 Name('local_var@main')
62     value or address:     Num('0')
63     position:             Pos(Num('10'), Num('6'))
64     size:                 Num('6')
65   }
66 ]

```

Code 3.82: *Symboltabelle für Call by Sharing für Arrays*

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

Die Komposition `Ref(Global(Num('0')))` ist für Variablen in den **Globalen Statischen Daten** und die Komposition `Ref(Stackframe(Num('6')))` ist für Variablen aus dem **Stackframe**. Dabei stellen die Zahlen in den **Container-Knoten** `Global(num)` bzw. `Stackframe(num)` die **relative Adressen** relativ zum DS-Register bzw. SP-Register dar, die aus der **Symboltabelle** entnommen sind.

```

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

```

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

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

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'fun_array_from_stackframe.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun_array_from_global_data.1',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Ref(Stackframe(Num('6'))))
16        SUBI SP 1;
17        MOVE BAF IN1;
18        SUBI IN1 8;
19        STOREIN SP IN1 1;
20        # NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr'))))
21        MOVE BAF ACC;
22        ADDI SP 3;
23        MOVE SP BAF;
24        SUBI SP 3;
25        STOREIN BAF ACC 0;
26        LOADI ACC GoTo(Name('addr@next_instr'));
27        ADD ACC CS;
28        STOREIN BAF ACC -1;
29        # Exp(GoTo(Name('fun_array_from_stackframe.2'))))
30        Exp(GoTo(Name('fun_array_from_stackframe.2'))))
31        # RemoveStackframe()
32        MOVE BAF IN1;
33        LOADIN IN1 BAF 0;
34        MOVE IN1 SP;
35        # Return(Empty())
36        LOADIN BAF PC -1;
37      ],
38    Block
39      Name 'main.0',
40      [
41        # StackMalloc(Num('2'))
42        SUBI SP 2;
43        # Ref(Global(Num('0'))))
44        SUBI SP 1;
45        LOADI IN1 0;
46        ADD IN1 DS;
47        STOREIN SP IN1 1;
48        # NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr'))))
49        MOVE BAF ACC;

```

```

50      ADDI SP 3;
51      MOVE SP BAF;
52      SUBI SP 9;
53      STOREIN BAF ACC 0;
54      LOADI ACC GoTo(Name('addr@next_instr'));
55      ADD ACC CS;
56      STOREIN BAF ACC -1;
57      # Exp(GoTo(Name('fun_array_from_global_data.1')))
58      Exp(GoTo(Name('fun_array_from_global_data.1')))
59      # RemoveStackframe()
60      MOVE BAF IN1;
61      LOADIN IN1 BAF 0;
62      MOVE IN1 SP;
63      # Return(Empty())
64      LOADIN BAF PC -1;
65  ]
66 ]

```

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

3.3.6.3.3 Umsetzung von Call by Value für Structs

Die **Call by Value** (Definition 1.4) Übergabe eines **Structs** wird im Folgenden mithilfe des Beispiels in Code 3.85 erklärt.

```

1 struct st {int attr1; int attr2[2];};
2
3
4 void fun_struct_from_stackframe(struct st param) {
5 }
6
7 void fun_struct_from_global_data(struct st param) {
8     fun_struct_from_stackframe(param);
9 }
10
11
12 void main() {
13     struct st local_var;
14     fun_struct_from_global_data(local_var);
15 }

```

Code 3.85: PicoC-Code für Call by Value für Structs

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

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

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

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

```

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

```

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

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

```

1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',

```

```

6      [
7      # Return(Empty())
8      LOADIN BAF PC -1;
9      ],
10     Block
11     Name 'fun_struct_from_global_data.1',
12     [
13     # StackMalloc(Num('2'))
14     SUBI SP 2;
15     # Assign(Stack(Num('3')), Stackframe(Num('2')))
16     SUBI SP 3;
17     LOADIN BAF ACC -4;
18     STOREIN SP ACC 1;
19     LOADIN BAF ACC -3;
20     STOREIN SP ACC 2;
21     LOADIN BAF ACC -2;
22     STOREIN SP ACC 3;
23     # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
24     MOVE BAF ACC;
25     ADDI SP 5;
26     MOVE SP BAF;
27     SUBI SP 5;
28     STOREIN BAF ACC 0;
29     LOADI ACC GoTo(Name('addr@next_instr'));
30     ADD ACC CS;
31     STOREIN BAF ACC -1;
32     # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
33     Exp(GoTo(Name('fun_struct_from_stackframe.2')))
34     # RemoveStackframe()
35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Return(Empty())
39     LOADIN BAF PC -1;
40     ],
41     Block
42     Name 'main.0',
43     [
44     # StackMalloc(Num('2'))
45     SUBI SP 2;
46     # Assign(Stack(Num('3')), Global(Num('0')))
47     SUBI SP 3;
48     LOADIN DS ACC 0;
49     STOREIN SP ACC 1;
50     LOADIN DS ACC 1;
51     STOREIN SP ACC 2;
52     LOADIN DS ACC 2;
53     STOREIN SP ACC 3;
54     # NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
55     MOVE BAF ACC;
56     ADDI SP 5;
57     MOVE SP BAF;
58     SUBI SP 5;
59     STOREIN BAF ACC 0;
60     LOADI ACC GoTo(Name('addr@next_instr'));
61     ADD ACC CS;
62     STOREIN BAF ACC -1;

```

```
63      # Exp(GoTo(Name('fun_struct_from_global_data.1')))  
64      Exp(GoTo(Name('fun_struct_from_global_data.1')))  
65      # RemoveStackframe()  
66      MOVE BAF IN1;  
67      LOADIN IN1 BAF 0;  
68      MOVE IN1 SP;  
69      # Return(Empty())  
70      LOADIN BAF PC -1;  
71  ]  
72  ]
```

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

4 Ergebnisse und Ausblick

Zum Schluss soll ein **Überblick** über das gegeben werden, was im Kapitel **Implementierung** implementiert wurde. Im Unterkapitel 4.1 wird darauf eingegangen ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel **Motivation** alle implementiert werden konnten und daraufhin mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten Zugriffen werden kann, aber auch auf Funktionalitäten **anderer mitimplementierter Tools**. Im Unterkapitel 4.2 wird aufgezeigt, was zur **Qualitätssicherung** implementiert wurde, um zu gewährleisten, dass der **PicoC-Compiler** die Kompilierung der **Programmiersprache** L_{PicoC} in **Syntax** und **Semantik identisch** zur entsprechenden **Untermenge** der Programmiersprache L_C umsetzt. Als allerletztes wird im Unterkapitel 4.3 ein Ausblick gegeben, wie der PicoC-Compiler **erweitert** werden könnte.

4.1 Funktionsumfang

In Kapitel **Implementierung** konnten **alle** Funktionalitäten, die in Kapitel **Motivation** erläutert wurden implementiert werden. Während der **Funktionsumfang** des **PicoC-Compiler** zum Stand des **Bachelorprojektes** noch sehr beschränkt war und einzig eine **Strukturierte Programmierung** mit `if(cond) { } else { }`, `while(cond) { }` usw. erlaubte und komplexere Programme nur mit **viel Aufwand** und **unübersichtlichen Spaghetticode** implementierbar waren, erlaubt es der **PicoC-Compiler** nachdem er in der **Bachelorarbeit** um **Felder**, **Zeiger**, **Verbunde** und **Funktionen** erweitert wurde mittels der **Funktionen** eine **Prozedurale Programmierung** umzusetzen. **Prozedurale Programmierung** zusammen mit der Möglichkeit **Felder**, **Zeiger** und **Verbunde** zu verwenden trägt zu einem **geordneteren, intuitiv verständlicheren** und **übersichtlicheren** Code bei.

Bei der Implementierung des **PicoC-Compilers** wurden verschiedene **Kommandozeilenoptionen** und **Modes** implementiert. Diese werden in den folgenden Kapiteln 4.1.1, 4.1.2 und 4.1.3 mithilfe **kurzer Anleitungen** erklärt.

Die kurzen **Anleitungen** in dieser **Schriftlichen Ausarbeitung** der Bachelorarbeit sollen nur zu einem **schnellen, grundlegenden Verständnis** der Verwendung des **PicoC-Compilers** und seiner **Kommandozeilenoptionen** und **Befehle** beihelfen, sowie zum Verständnis der **weiteren implementierten Tools**. Alle weiteren **Kommandozeilenoptionen** und **Befehle** sind für die Verwendung des PicoC-Compilers **unwichtig** und erweisen sich nur in **speziellen Situationen** als nützlich, weshalb für diese auf die **ausführlichere Dokumentation** unter [Link](https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt)¹ verwiesen wird.

4.1.1 Kommandozeilenoptionen

Will man einfach nur ein **Programm** `program.picoc` kompilieren ist das mit dem **PicoC-Compiler** genauso **unkompliziert** wie mit dem **GCC** durch einfaches **Angeben der Datei**, die kompiliert werden soll: `> picoc_compiler program.picoc`. Als Ergebnis des Kompiliervorgangs wird eine Datei `program.reti` mit dem entsprechenden **RETI-Code** erstellt, wobei für die **Benennung der Datei** einfach nur der

¹https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt

Basisname der Datei `program` an eine neue **Dateiendung** `.ret` angehängt wird².

Daneben gibt es allerdings auch die Möglichkeit **Kommandozeilenoptionen** `<cli-options>` in der Form `> picoc_compiler <cli-options> program.picoc` mitanzugeben, von denen die **wichtigsten** in Tabelle 4.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumentation** unter [Link](#) nachgelesen werden.

²Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der festen Namen `a.out` erstellt.

Kommandozeilenoption	Beschreibung	Standardwert
<code>-i, --intermediate_stages</code>	Gibt Zwischenschritte der Kompilierung in Form der verschiedenen Tokens , Ableitungsbäume , Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateieindungen aber gleichem Basinamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal .	false , most_used: true
<code>-p, --print</code>	Gibt alle Dateiausgaben auch im Terminal aus. Diese Option ist im Shell-Mode dauerhaft aktiviert.	false (true im Shell-Mode und für den most_used- Befehl)
<code>-v, --verbose</code>	Fügt den verschiedenen Zwischenschritten der Kompilierung , unter anderem auch dem finalen RETI-Code Kommentare hinzu, welche ein Statement oder Befehl aus einem vorherigen Pass beinhalten, der durch die darunterliegenden Statements oder Befehle ersetzt wurde. Wenn die <code>--run</code> -Option aktiviert ist, wird der Zustand der virtuellen RETI-CPU vor und nach jedem Befehl angezeigt.	false
<code>-vv, --double_verbose</code>	Hat dieselben Effekte , wie die <code>--verbose</code> -Option, aber bewirkt zusätzlich weitere Effekte . PicoC-Knoten erhalten bei der Ausgabe in den Abstrakten Syntaxbäumen zusätzliche runde Klammern , sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In Fehlermeldungen werden mehr Tokens angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der <code>--intermediate_stages</code> -Option werden in den dadurch ausgegebenen Abstrakten Syntaxbäumen ebenfalls versteckte Attribute , die Informationen zu Datentypen und für Fehlermeldungen beinhalten angezeigt.	false
<code>-h, --help</code>	Zeigt die Dokumentation , welche ebenfalls unter Link gefunden werden kann im Terminal an. Mit der <code>--color</code> -Option kann die Dokumentation mit farblicher Hervorhebung im Terminal angezeigt werden.	false
<code>-l</code>	Es lässt sich einstellen, wieviele Zeilen rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	
<code>-c</code>	Aktiviert farbige Ausgabe .	
<code>-R, --run</code>	Führt die RETI-Befehle , die das Ergebnis der Kompilierung sind mit einer virtuellen RETI-CPU aus. Wenn die <code>--intermediate_stages</code> -Option aktiviert ist, wird eine Datei <code><basename>.reti_states</code> erstellt, welche den Zustand der RETI-CPU nach dem letzten ausgeführten RETI-Befehl enthält. Wenn die <code>--verbose</code> - oder <code>--double_verbose</code> -Option aktiviert ist, wird der Zustand der RETI-CPU vor und nach jedem Befehl auch noch zusätzlich in die Datei <code><basename>.reti_states</code> ausgegeben.	false , most_used: true
<code>-B, --process_begin</code>	Setzt die relative Adresse , wo der Prozess bzw. das Codesegment für das ausgeführte Programm beginnt.	3
<code>-D, --datasegment_size</code>	Setzt die Größe des Datensegments . Diese Option muss mit Vorsicht gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die Globalen Statischen Daten und der Stack miteinander kollidieren.	32

Alle **kleingeschriebenen** Kommandozeilenoptionen, wie `-i`, `-p`, `-v` usw. betreffen dabei den **PicoC-Compiler** und alle **großgeschriebenen** Kommandozeilenoptionen, wie `-R`, `-B`, `-D` usw. betreffen den **RETI-Interpreter**.

4.1.2 Shell-Mode

Will man z.B. eine **Folge von Statements** in der Programmiersprache L_{PicoC} **schnell** kompilieren ohne eine Datei erstellen zu müssen, so kann der **PicoC-Compiler** im sogenannten **Shell-Mode** aufgerufen werden. Hierzu wird der PicoC-Compiler **ohne Argumente** `> picoc_compiler` aufgerufen, wie es in Code 4.1 zu sehen ist. Die angegebene **Folge von Statements** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem `> compile <cli-options> <filename>`-Befehl (oder der **Abkürzung** `cpl`) kann **PicoC-Code** zu **RETI-Code** kompiliert werden. Die Kommandozeilenoptionen `<cli-options>` sind dieselben, wie wenn der Compiler **direkt** mit Kommandozeilenoptionen aufgerufen wird. Die **wichtigsten** dieser **Kommandozeilenoptionen** sind in Tabelle 4.1 angegeben.

Mit dem Befehl `> quit` kann der **Shell-Mode** wieder **verlassen** werden.

```
> picoc_compiler
PicoC Shell. Enter `help` (shortcut `?`) to see the manual.
PicoC> cpl "6 * 7;";
----- RETI -----
SUBI SP 1;
LOADI ACC 6;
STOREIN SP ACC 1;
SUBI SP 1;
LOADI ACC 7;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
LOADIN BAF PC -1;

Compilation successfull

PicoC> quit
```

Code 4.1: Shellaufruf und die Befehle *compile* und *quit*

Wenn man möglichst alle nützlichen **Kommandozeilenoptionen** direkt aktiviert haben will, bei denen es **keinen** Grund gibt, sie nicht mitanzugeben, kann der Befehl `> most_used <cli-options> <filename>` (oder seine **Abkürzung** `mu`) genutzt werden, um diese Kommandozeilenoptionen mit dem `compile`-Befehl **nicht** jedes mal **selbst** Angeben zu müssen. In der Tabelle 4.1 sind in grau die Werte der einzelnen **Kommandozeilenoptionen** angegeben, die bei dem Befehl `most_used` gesetzt werden. In Code 4.2 ist der `most_used`-Befehl in seiner Verwendung zu sehen.

Dadurch, dass die `--intermediate_stages`- und die `--run`-Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie **Tokens**, **Derivation Tree** usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls angezeigt. Aus **Platzgründen** ist das meiste allerdings mit `'...'` ausgelassen.






```


PicoC> mu "int var = 42;";
----- Code -----
// stdin.picoc:
void main() {int var = 42;}
----- Tokens -----
...
----- Derivation Tree -----
...
----- Derivation Tree Simple -----
...
----- Abstract Syntax Tree -----
...
----- PicoC Shrink -----
...
----- PicoC Blocks -----
...
----- PicoC Mon -----
...
----- Symbol Table -----
...
----- RETI Blocks -----
...
----- RETI Patch -----
...
----- RETI -----
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
LOADIN BAF PC -1;
----- RETI Run -----
...

Compilation successfull

```

Code 4.2: Shell-Mode und der Befehl *most_used*

Im **Shell-Mode** kann der **Cursor** mit den  und  Pfeiltasten bewegt werden. In der **Befehlshistorie** kann sich mit den  und  Pfeiltasten **rückwärts** und **vorwärts** bewegt werden. Mit  kann ein Befehl **automatisch vervollständigt** werden.

Es gibt für den **Shell-Mode** noch **weitere Befehle**, wie `color_toggle`, `history` etc. und **kleinere Funktionalitäten** für die Shell, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wird allerdings auf die **Dokumentation** unter [Link](#) verwiesen, welche auch über den Befehl  angezeigt werden kann.

4.1.3 Show-Mode

Der **Show-Mode** ist ein Nebenprodukt der Implementierung des **PicoC-Compilers**. Dieser **Mode** wurde eigentlich nur implementiert, um beim **Testen** des PicoC-Compilers **Bugs** bei der Generierung des **RETI-Code** zu finden, indem im Terminal eine **virtuelle RETI-CPU** angezeigt wird, welches den **kompletten**

Zustand einer virtuell ausgeführten RETI mit allen **Registern**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen** anzeigt.

Allerdings bringt die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen** auch einen großen **Lerneffekt** mit sich, weshalb der **Show-Mode** noch **weiterentwickelt** wurde, sodass auch **Studenten** ihn auf unkomplizierte Weise nutzen können.

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen**, die z.B. unter `<more-options>` noch für die **Makefile** gesetzt werden können sind in Tabelle 4.2 aufgelistet.

Kommandozeilenoption	Beschreibung	Standardwert
FILEPATH	Pfad zur Datei, die im Show-Mode angezeigt werden soll	<code>()</code>
TESTNAME	Name des Tests. Alles andere als der Basisname , wie die Dateiendung wird abgeschnitten	<code>()</code>
EXTENSION	Dateiendung , die an TESTNAME angehängt werden soll zu <code>./tests/TESTNAME.EXTENSION</code>	<code>reti_states</code>
NUM_WINDOWS	Anzahl Fenster auf die ein Dateiinhalte verteilt werden soll	<code>5</code>
VERBOSE	Möglichkeit die Kommandozeilenoption <code>-v</code> oder <code>-vv</code> zu aktivieren für eine ausführlichere Ausgabe	<code>()</code>
DEBUG	Möglichkeit die Kommandozeilenoption <code>-d</code> zu aktivieren, um bei <code>make test-show TESTNAME=<testname></code> den Debugger für den entsprechenden Test <code><testname></code> zu starten	<code>()</code>

Tabelle 4.2: Makefileoptionen

Alternativ kann der **Show-Mode** mit dem Befehl `make test-show TESTNAME=<testname> <more-options>` auch für einen der geschriebenen **Tests** im Ordner `/tests` gestartet werden. Der **Test** wird bei diesem Befehl **erst ausgeführt** und dann der **Show-Mode** gestartet.

Der **Show-Mode** nutzt den Terminal Texteditor **Neovim**³ um einen **Dateiinhalte** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 4.1 zu sehen ist. Für den **Show-Mode** wird eine eigene **Konfiguration für Neovim** verwendet, welche in der **Konfigurationsdatei** `/interpr_showcase.vim` spezifiziert ist.

Gedacht ist der **Show-Mode** vor allem dafür etwas ähnliches wie ein **RETI-Debugger** zu sein und wird daher standardmäßig bei **Nicht-Angabe** einer **EXTENSION** auf die Datei `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen anzuzeigen, indem **EXTENSION** auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandt, indem die verschiedenen **Zustände der RETI-CPU nicht zur Laufzeit** des **Show-Mode** berechnet werden, sondern schon berechnet wurden und nacheinander in die Datei `<program>.reti_states` ausgegeben wurden. Der **Show-Mode** macht nichts anderes, als immer an die Stelle zu springen, an welcher der nächste Zustand anfängt. Durch Drücken von `Tab` und `↑ -Tab` können auf diese Weise die **verschiedenen Zuständen** der **RETI-CPU vor** und **nach** der Ausführung eines Befehls **angezeigt** werden.

³Home - Neovim.

index: 43	00019 ADDI SP 1;	00057 LOADIN DS ACC 0;	00095 STOREIN SP ACC 2;	00133 0
instruction: ADDI SP 1;	00020 LOADIN SP ACC 1;	00058 STOREIN SP ACC 1;	00096 ADDI SP 1;	00134 0
ACC: 1	00021 STOREIN DS ACC 0;	00059 LOADIN SP ACC 1;	00097 LOADIN SP ACC 1;	00135 0
ACC_SIMPLE: 1	00022 ADDI SP 1;	00060 ADDI SP 1;	00098 STOREIN DS ACC 0;	00136 0
IN1: 0	00023 SUBI SP 1;	00061 CALL PRINT ACC;	00099 ADDI SP 1;	00137 0
IN1_SIMPLE: 0	00024 LOADIN DS ACC 0;	00062 SUBI SP 1;	00100 JUMP -32;	00138 0
IN2: 4	00025 STOREIN SP ACC 1;	00063 LOADI ACC 0;	00101 SUBI SP 1;	00139 0
IN2_SIMPLE: 4	00026 SUBI SP 1;	00064 STOREIN SP ACC 1;	00102 LOADIN DS ACC 0;	00140 0
PC: 2147483686	00027 LOADI ACC 4;	00065 LOADIN SP ACC 1;	00103 STOREIN SP ACC 1;	00141 0
PC_SIMPLE: 38	00028 STOREIN SP ACC 1;	00066 STOREIN DS ACC 0;	00104 LOADIN SP ACC 1;	00142 0
SP: 2147483792	00029 LOADIN SP ACC 2;	00067 ADDI SP 1;	00105 ADDI SP 1;	00143 0
SP_SIMPLE: 144	00030 STOREIN SP IN2 1;	00068 SUBI SP 1;	00106 JUMP== 7;	00144 4 <- SP
BAF: 2147483650	00031 SUB ACC IN2;	00069 LOADIN DS ACC 0;	00107 SUBI SP 1;	00145 1
BAF_SIMPLE: 2	00032 JUMP< 3;	00070 STOREIN SP ACC 1;	00108 LOADIN DS ACC 0;	UART:
CS: 2147483651	00033 LOADI ACC 0;	00071 SUBI SP 1;	00109 STOREIN SP ACC 1;	00000 0
CS_SIMPLE: 3	00034 JUMP 2;	00072 LOADI ACC 2;	00110 LOADIN SP ACC 1;	00001 0
DS: 2147483762	00035 LOADI ACC 1;	00073 STOREIN SP ACC 1;	00111 ADDI SP 1;	00002 0
DS_SIMPLE: 114	00036 STOREIN SP ACC 2;	00074 LOADIN SP ACC 2;	00112 CALL PRINT ACC;	00003 0
SRAM:	00037 ADDI SP 1;	00075 LOADIN SP IN2 1;	00113 LOADIN BAF PC -1;	EPROM:
00000 JUMP 0;	00038 LOADIN SP ACC 1; <- PC	00076 SUB ACC IN2;	00114 3 <- DS	00000 LOADI DS -2097152; <- IN1
00001 2147483648	00039 ADDI SP 1;	00077 JUMP< 3;	00115 0	00001 MULTI DS 1024; <- ACC
00002 0 <- BAF	00040 JUMP== 2;	00078 LOADI ACC 0;	00116 0	00002 MOVE DS SP;
00003 CALL INPUT ACC; <- CS	00041 JUMP -32;	00079 JUMP 2;	00117 0	00003 MOVE DS BAF;
00004 SUBI SP 1;	00042 SUBI SP 1;	00080 LOADI ACC 1;	00118 0	00004 MOVE DS CS; <- IN2
00005 STOREIN SP ACC 1;	00043 LOADIN DS ACC 0;	00081 STOREIN SP ACC 2;	00119 0	00005 ADDI SP 145;
00006 LOADIN SP ACC 1;	00044 STOREIN SP ACC 1;	00082 ADDI SP 1;	00120 0	00006 ADDI BAF 2;
00007 STOREIN DS ACC 0;	00045 SUBI SP 1;	00083 LOADIN SP ACC 1;	00121 0	00007 ADDI CS 3;
00008 ADDI SP 1;	00046 LOADI ACC 2;	00084 ADDI SP 1;	00122 0	00008 ADDI DS 114;
00009 SUBI SP 1;	00047 STOREIN SP ACC 1;	00085 JUMP== 16;	00123 0	00009 MOVE CS PC;
00010 LOADIN DS ACC 0;	00048 LOADIN SP ACC 2;	00086 SUBI SP 1;	00124 0	
00011 STOREIN SP ACC 1;	00049 LOADIN SP IN2 1;	00087 LOADIN DS ACC 0;	00125 0	
00012 SUBI SP 1;	00050 SUB ACC IN2;	00088 STOREIN SP ACC 1;	00126 0	index: 44
00013 LOADI ACC 1;	00051 STOREIN SP ACC 2;	00089 SUBI SP 1;	00127 0	instruction: LOADIN SP ACC 1;
00014 STOREIN SP ACC 1;	00052 ADDI SP 1;	00090 LOADI ACC 1;	00128 0	ACC: 1
00015 LOADIN SP ACC 2;	00053 LOADIN SP ACC 1;	00091 STOREIN SP ACC 1;	00129 0	ACC_SIMPLE: 1
00016 LOADIN SP IN2 1;	00054 ADDI SP 1;	00092 LOADIN SP ACC 2;	00130 0	IN1: 0
00017 ADD ACC IN2;	00055 JUMP== 13;	00093 LOADIN SP IN2 1;	00131 0	IN1_SIMPLE: 0
00018 STOREIN SP ACC 2;	00056 SUBI SP 1;	00094 ADD ACC IN2;	00132 0	IN2: 4
00019 ADDI SP 1;	00057 LOADIN DS ACC 0;	00095 STOREIN SP ACC 2;	00133 0	IN2_SIMPLE: 4
				PC: 2147483687

Abbildung 4.1: Show-Mode in der Verwendung

Zur **besseren Orientierung** wird für alle Register ebenfalls ein mit der Registerbezeichnung beschrifteter **Zeiger** <- REG an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert im Register** nach der **Memory Map** dem **Adressbereich** von **EPROM**, **UART** oder **SRAM** entspricht.

Durch Drücken von **Esc** oder **q** kann der **Show-Mode** wieder verlassen werden. Es gibt für den **Show-Mode** noch viele weitere **Tastenkürzel**, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wieder allerdings auf die **Dokumentation** unter [Link](#) verwiesen. Des Weiteren stehen durch die Nutzung des Terminal Texteditors **Neovim** auch alle **Funktionalitäten** dieses mächtigen Terminal Texteditors zur Verfügung, welche mittels der Eingabe von **:help** **nachgelesen** werden können oder mittels der Eingabe von **:Tutor** mithilfe einer kurzen **Einführungsanleitung** **erlernt** werden können.

4.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm 2.2.1 in Unterkapitel 2.1 genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 4.2.1 definieren. Ein **beliebiges** Testprogramm P_{PicoC} in der Sprache L_{PicoC} muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm P_{RETI} in der Sprache L_{RETI} , trotz der **unterschiedlichen Syntax**.

Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis **/tests** bzw. unter [Link](#)⁴ zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 4.3.

⁴https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

Testkategorie	Beschreibung
basic	Einfache Tests, welche die grundlegenden Funktionalitäten des Compilers testen
advanced	Tests, die Spezialfälle und Kombinationen verschiedener Funktionalitäten des Compilers testen
hard	Tests, die längere, komplexe Programme testen, für welche die Funktionalitäten des Compilers in perfekter Harmonie miteinander funktionieren müssen
example	Tests, die bekannte Algorithmen darstellen und daher als gutes, repräsentatives Beispiel für die Funktionsfähigkeit des PicoC-Compilers dienen
error	Tests, die Fehlermeldungen testen. Für diese Tests wird keine Verifikation ausgeführt
exclude	Tests, für welche aufgrund vielfältiger Gründe keine Verifikation ausgeführt werden soll

Tabelle 4.3: Testkategorien

Dass die Programme in beiden Sprachen die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn beide die **gleiche Ausgabe** haben und es sehr **unwahrscheinlich** ist zufällig bei der gewählten Eingabe die spezifische Ausgabe zu erhalten. Wenn **immer mehr Tests**, die alle einen unterschiedlichen Teil der Semantik der Sprache L_{PicoC} abdecken vorliegen, bei denen die jeweiligen Programme P_{PicoC} und P_{RETI} interpretiert die gleiche **Ausgabe** haben, dann kann mit **immer höherer Wahrscheinlichkeit** von einem **funktionierenden** Compiler ausgegangen werden.

Die Kante vom Testprogramm P_{PicoC} zur Ausgabe aus Diagramm 4.2.1 drückt aus, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat, in welcher der **Schreiber des Tests** die Rolle des entsprechenden **Interpreters**⁵ aus Diagramm 2.2.1 übernimmt und die **erwartete Ausgabe** seiner eigenen Interpretation des **PicoC-Codes** anstelle von `<space_seperated_output>` hineinschreibt.

Ein Beispiel für einen **Test** ist in Code 4.3 zu sehen. Sobald die Tests mithilfe der `/Makefile` mit dem Befehl `> make test` ausgeführt werden, wird als erstes für **jeden** Test das Bashscript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert⁶ und die entsprechenden Werte in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt.

Die Datei `<program>.in` enthält **Eingaben**, welche durch `input()`-Funktionsaufrufe eingelesen werden, die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben** der `print(<exp>)`-Funktionsaufrufe, die später eingeführte Datei `<program>.out` enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Tests** und die Datei `<program>.datasegment_size` enthält die **Größe des Datensegments** für die Ausführung des entsprechenden Tests.

⁵Der die **Semantik** des Tests umsetzt.

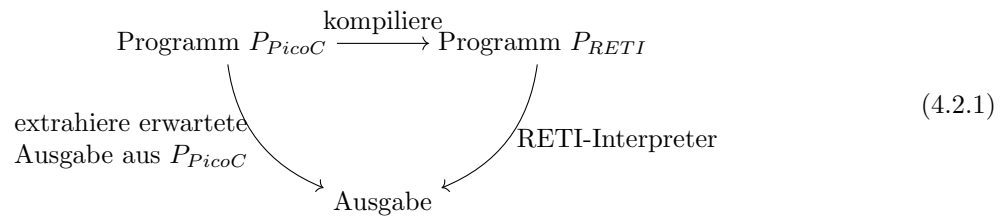
⁶Falls vorhanden.


```
// in:21 2 6 7
// expected:42 42
// datasegment:4

void main() {
    print(input() * input());
    print(input() * input());
}
```

Code 4.3: Typischer Test

Die Kante vom Programm P_{RETI} zur Ausgabe aus Abbildung 4.2.1 ist dadurch erfüllt, dass das Programm P_{RETI} vom **RETI-Interpreter** interpretiert wird und jedes mal beim Antreffen des **RETI-Befehls** `CALL PRINT ACC` der entsprechende **Inhalt** des ACC-Registers in die Datei `<program>.out` ausgegeben wird. Ein Test kann mit einer bestimmten Wahrscheinlichkeit die **Korrektheit** des **Teils der Semantik** der Sprache L_{PicoC} , die er abdeckt **verifizieren**, wenn der Inhalt von `<program>.out_expected` und `<program>.out` **identisch** ist.

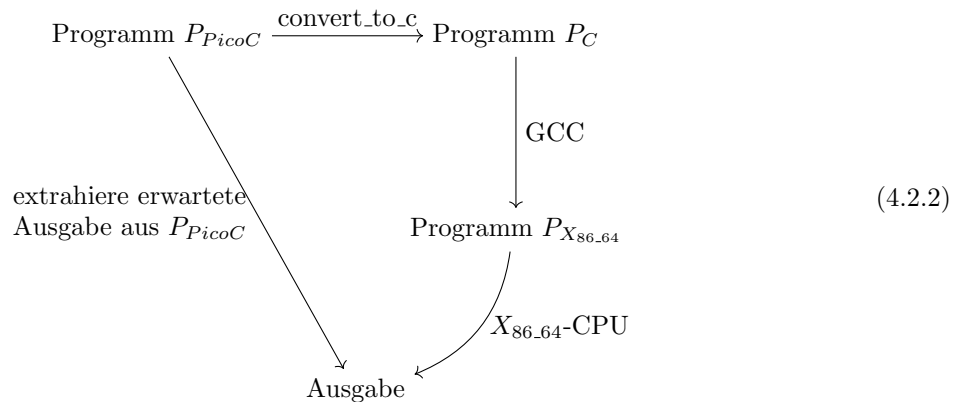


Allerdings gibt es bei dem Testverfahren, welches in Diagramm 4.2.1 dargestellt ist ein **Problem**, denn der **Schreiber** der Tests ist in diesem Fall die **gleiche Person**, die auch den **Compiler implementiert**. Wenn der **Schreiber** der Tests ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl im **Test** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht⁷.

Aus diesem Grund muss hier eine **weitere Maßnahme**, welche in Diagramm 4.2.2 dargestellt ist eingeführt werden, die gewährleistet, dass die **Ausgabe** in Diagramm 4.2.1 sich auf jeden Fall aus der **Semantik** der Sprache L_{PicoC} ⁸ ergibt. Das wird erreicht, indem wie in Diagramm 4.2.2 dargestellt ist, überprüft wird, ob die **Ausgabe** des Pfades von P_{PicoC} zur Ausgabe mit der **Ausgabe** des Pfades von P_C über $P_{X_{86.64}}$ **identisch** ist.

⁷Welche ja identisch zu der von L_C sein sollte.

⁸Die eine **Untermenge** von L_C ist.



Das Programm P_C ergibt sich dabei aus dem Testprogramm P_{PicoC} durch **Ausführen** des Pythonscripts `/convert_to_c.py`, welches **später näher erläutert** wird. Mithilfe der `/Makefile` und dem Befehl `> make convert` lässt sich dieses Pythonscript auf **alle** Tests anwenden.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante von P_C zu P_{X86_64} . Beim **GCC** handelt es sich um einen Compiler der Sprache L_C , der somit auch mit Ausnahme der `print()` und `input()`-Funktionen auch die Sprache L_{PicoC} kompilieren kann. Der **GCC** setzt aufgrund seiner bekanntermaßen **vielfachen Verwendung** auf der Welt und seinem **sehr langem Bestehen** seit 1987⁹ ¹⁰ die **Semantik** der Sprache L_C , vor allem für die kleine Untermenge, welche L_{PicoC} darstellt mit sehr hoher Wahrscheinlichkeit **korrekt** um.

Durch das **Abgleichen** mit dem **GCC** in Diagramm 4.2.2 kann nun **sichergestellt** werden, dass die Tests **nicht** nur die Interpretation, die der Schreiber der Tests und Implementierer des **PicoC-Compilers** von der Semantik der Sprache L_{PicoC} hat **bestätigen**, sondern die tatsächliche **Einhaltung der Semantik** der Sprache L_{PicoC} testen.

Dazu durchläuft jeder Test, wie in Diagramm 4.2.2 dargestellt ist eine **Verifikation**, in der **verifiziert** wird, ob bei der Kompilierung des Testprogramms P_C mit dem **GCC** und Ausführung des hieraus generierten X_{86_64} -Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe `// expected:<space_seperated_output>` des Testschreibers ist. Erst dann ist ein Test **verifiziert**, d.h. man kann, wenn der Test **vernünftig definiert** ist mit **hoher Wahrscheinlichkeit** sagen¹¹, dass wenn dieser Test für den **PicoC-Compiler** durchläuft, der **Teil der Semantik** der Sprache L_{PicoC} , den dieser Test testet vom PicoC-Compiler **korrekt umgesetzt** ist.

Für diese **Verifikation** ist das Bashscript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt wird. Beim Befehl `> make test` wird dieses Bashscript **vor** dem eigentlichen Testen¹² durchgeführt. In Code 4.4 ist ein Testdurchlauf mit `> make test` zu sehen. Wobei **Verified: 50/50** anzeigt, wieviele der Tests **verifizierbar** sind¹³, also beim **GCC** ohne Fehlermeldung durchlaufen, **Not verified:** die **nicht verifizierbaren** Tests angibt, **Running through: 88 / 88** anzeigt wieviele Tests mit dem **PicoC-Compiler** durchlaufen, **Not running through:** die **nicht** durchlaufenden Tests angibt, **Passed: 88 / 88** zeigt bei wievielen Tests die Ausgabe mit der erwarteten Ausgabe **identisch** ist, **Not passed:** die Tests anzeigt, bei denen das **nicht** der Fall ist.

⁹History - GCC Wiki.

¹⁰In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.

¹¹Es besteht allerdings immer eine **Chance**, dass die Ausgabe für den Test nur **zufällig** übereinstimmt. Diese Chance kann allerdings durch **vernünftige Definition** des Tests sehr **gering** gehalten werden.

¹²Prüfen, ob der interpretierte RETI-Code des PicoC-Compilers die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.

¹³Also **alle** Tests aus den **Kategorien basic, advanced, hard** und **example**.

```

> make test
=====
= ./tests/basic_array_init.picoc =
=====
...
=====
=          Verification          =
=====
./tests/basic_array_init.c
...
=====
=          Results              =
=====
Verified: 50 / 50
Not verified:
Running through: 88 / 88
Not running through:
Passed: 88 / 88
Not passed:

```

Code 4.4: Testdurchlauf

Der Befehl `make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` TESTNAME, VERBOSE und DEBUG aus Tabelle 4.2 kombinieren.

Das Pythonscript `/convert_to_c.py` ist notwendig, da L_{PicoC} sich bei den Funktionen `print()` und `input()` von der **Syntax** der Sprache L_C unterscheidet, bei der z.B. `printf("%d", 12)` anstelle von `print(12)` geschrieben werden muss. Für die Sprache L_{PicoC} erfüllen die Funktionen `print()` und `input()` allerdings nur den **Zweck**, dass sie zum **Testen des Compilers** gebraucht werden, um über die Funktion `input()` für eine bestimmte **Eingabe** die **Ausgabe** über die Funktion `print()` testen zu können. Aus diesem Grund ist es notwendig die **Syntax** dieser Funktionen in L_C zu übersetzen.

Die Funktion `print(<exp>)` wird vom Pythonscript `convert_to_c.py` zu `printf("%d", <exp>)` übersetzt. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibliothek** `<stdio.h>` eingebunden werden. Bei der Funktion `input()` wurde **nicht** der aufwändige **Umweg** genommen die Funktion `input()` durch ihre entsprechende Funktion in der Sprache L_C zu ersetzen. Es geht viel direkter, indem **nacheinander** die `input()`-Funktionen durch entsprechende Eingaben aus der Datei `<program>.in` ersetzt werden. Man schreibt einfach **direkt** den Wert hin, den die `input()`-Funktionen normalerweise einlesen sollten.

4.3 Erweiterungsideen

Mit dem **Funktionsumfang** des **PicoC-Compilers**, der in Unterkapitel 4.2 erläutert wurde muss allerdings das Ende der Fahnenstange noch **nicht** erreicht sein. Weitere Ideen, die im **PicoC-Compiler**¹⁴ implementiert werden könnten, wären:

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern** und erst wenn alle Register **voll** sind werden Variablen an Adressen auf dem **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller** ist, als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations**

¹⁴Möglicherweise ja im Rahmen eines **Masterprojektes** 😊.

(Definition 2.47) zuzuweisen wird mithilfe einer **Liveness Analyse** (Definition 5.8) ein **Interferenzgraph** (Definition 5.11) aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition 5.10) angewandt, der den **Locations** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Des Weiteren muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussanalyse** (Definition 5.14) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition 5.12) auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.¹⁵

- **Tail Call:** Wenn ein Funktionsaufruf das **letzte** Statement in einem Funktionsblock ist, wird der Stackframe dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion zurückgekehrt werden muss¹⁶. Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden, **bevor** der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die nur Tail Calls ausführt nur eine **konstante Menge** an **Speicherplatz** auf dem Stack verbraucht. In Code 4.5 sind **zwei Tail Calls** markiert.
- **Partielle Evaluation:** Bei Ausdrücken wie `4 + input() - 2`, `input() * 1` oder `0 + input() * 2` können **Teilausdrücke** bereits **während** des **Kompilierens** zu `2 + input()`, `input()` und `input() * 2` **partiell** berechnet werden. Die kann durch einen neuen **PicoC-Eval Pass** umgesetzt werden, der **vor** oder **nach** dem **PicoC-Shrink Pass** den Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache *LPicoC-Eval* umformt. In der Abstrakten Syntax der Sprache *LPicoC-Eval* sind **binäre Operationen** zwischen zwei `Num(str)`-PicoC-Knoten **nicht möglich**. Diese **partielle Vorberechnung** kann auch auf **Konstanten** und **Variablen** ausgeweitet werden. Der **Vorteil** ist, dass hierdurch weniger **RETI-Code** produziert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.
- **Lazy Evaluation:** Bei Ausdrücken wie `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei `var1 = 0` und `var2 = 1` müssen diese Ausdrücke nur **soweit** berechnet werden, wie es **benötigt** wird. Sobald bei einer Aneinanderreihung von `&&`-Operationen einmal eine 0 auftaucht, muss der Rest des Ausdrucks **nicht** mehr berechnet werden, da mit dem Auftauchen der 0 bereits klar ist, dass dieser Ausdruck sich zu 0 auswertet. Genauso für eine Aneinanderreihung von `||`-Operationen und dem Auftauchen einer 1. Daher kommt es aufgrund der Division durch 0 nicht zu einer **DivisionByZero-Fehlermeldung**, da die Ausdrücke garnicht so weit ausgewertet werden. Im Unterschied zur **Partiellen Evaluation** läuft **Lazy Evaluation**¹⁷ zur **Laufzeit** ab.
- **Objektorientierung:** Wie in der Programmiersprache *LC++* müssen **Klassen** und `new`-, `new[]`-, `delete`-, `delete[]`- und `::`-Operatoren eingeführt werden. Die Speicherung eines **Objekts** ist ähnlich wie bei **Verbunden**.
- **Mehrere Dateien:** **Funktionen** werden zusammen mit **Attributen** in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden **.h-Headerdateien** verwendet, für die Definition sind **.c-Quellcodedateien** da. Hierbei ist der **Basisname** einer **.h-Headerdatei** **identisch** zur entsprechenden **.c-Quellcodedatei** mit den entsprechenden Definitionen. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei entspricht. Über einen **Linker** (Definition 5.4) können die **kompilierten .o-Objektdaten** (Definition 5.3) zusammengefügt werden, wobei der **Linker** darauf achtet **keinen doppelten Code** zuzulassen.
- **malloc und free:** Es wird eine **Bibltiothek** mit den Funktionen `malloc` und `free`, wie in der Bibliothek

¹⁵Die in diesem **Unterpunkt** erwähnten **Begriffe** werden nur **grob** erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

¹⁶Was der Grund ist, warum ein **Stackframe** überhaupt angelegt wird, damit später beim **Rücksprung** aus der **aufgerufenen Funktion** die Ausführung mit allen Variablen, wie **vor der Ausführung** fortgesetzt werden kann.

¹⁷Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

`stdlib`¹⁸ implementiert, deren `.h`-Headerdatei mittels `#include "malloc_and_free.h"` eingebunden werden muss. Es braucht eine neue **Kommandozeilenoption** `-l` um dem **Linker** verwendete Bibliotheken mitzuteilen. Aufgrund der Einführung von `malloc` und `free` wird im **Datensegment** der Abschnitt nach den **Globalen Statischen Daten** als **Heap** bezeichnet, der mit dem **Stack** kollidieren kann. Im **Heap** wird von der `malloc`-Funktion **Speicherplatz allokiert** und ein **Pointer** auf diesen **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz im **Heap** zur **Allokation** frei ist, muss dies in einer **Datenstruktur** abgespeichert werden.

- **Garbage Collector:** Anstelle der `free`-Funktion kann auch einfach die `malloc`-Funktion direkt so implementiert werden, dass sobald der Speicherplatz auf dem **Heap** knapp wird, Speicherplatz, der sonst unmöglich in der Zukunft mehr genutzt werden würde freigegeben wird. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition 5.15) implementieren.
- **stdio.h:** Die Funktionen `print` und `input` werden nicht über den **Trick** einen eigenen **RETI-Befehl** `CALL (PRINT | INPUT) ACC` für den **RETI-Interpreter** zu definieren, der einfach **direkt** das **Ausgeben** und **Eingaben entgegennehmen** übernimmt gelöst, sondern über eine eigene **stdio-Bibliothek** mit `print`- und `input`-Funktionen, welche die **UART** verwenden, um z.B. an einem simpel gehaltenen simulierten **Monitor** Daten zu übertragen, die dieser anzeigt.
- **Feld mit Länge:** Man könnte in einer **Bibliothek** einen eigenen **Felddatentyp**, wie in der Programmiersprache L_{C++} mit dem Datentyp `std::vector` über eine **Klasse** implementieren, der seine **Anzahl Elemente** an den **Anfang** des Felds speichert, sodass über eine **Methode** `size` die **Anzahl Elemente** direkt über die **Variable des Felds** selbst ausgelesen werden kann (z.B. `vec.var.size`) und **nicht** in einer **seperaten Variable** gespeichert werden muss.
- **Maschinencode in binärer Repräsentation:** Maschinencode wird nicht, wie momentan beim **PicoC-Compiler** in **menschenlesbarer Repräsentation** ausgegeben, sondern in **binärer Repräsentation** nach dem **Intruktionsformat**, welches in der Vorlesung P. D. C. Scholl, „Betriebssysteme“ festgelegt wurde.
- **PicoPython:** Da das **Lark Parsing Toolkit** verwendet wurde, welches das **Parsen** über eine selbst angegebene **Grammatik** übernimmt, könnte mit **relativ geringem Aufwand** ein Grammatik definiert werden, die eine zur Programmiersprache L_{Python} **ähnliche Syntax** beschreibt. Die **Syntax** einer Programmiersprache lässt sich durch Austauschen der Grammatik **sehr einfach** ändern, nur die **Semantik** zu ändern kann **deutlich aufwändiger** sein. Viele der **PicoC-Knoten** könnten für die Programmiersprache $L_{PicoPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.
- **Call by Reference:** Über das wiederverwenden des `&`-Symbols für **Parameter** bei **Funktionsdeklaration** und **Funktionsdefinition**, wie es in der Vorlesung P. D. P. Scholl, „Einführung in Embedded Systems“ erklärt wurde.
- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt durch welche spezielle **Informationen** in den RETI-Code geschrieben werden, die einem **Debugger** unter anderem mitteilen, wo die **RETI-Befehle** für ein Statement **beginnen** und wo sie **aufhören** usw., damit der **Debugger** weiß, bis wohin er die **RETI-Befehle** ausführen soll, damit er ein Statement abgearbeitet hat.
- **Bootstrapping:** Mittels **Bootstrapping** lässt sich der **PicoC-Compiler** unabhängig von der Sprache L_{Python} und der **Maschine**, die das **cross-compilen** (Definition 2.5) übernimmt machen. Im Unterkapitel 4.4 wird genauer hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zum einem **Compiler** für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

¹⁸Auch engl. **General Purpose Standard Library** genannt.

```
1 // in:42
2 // expected:0
3
4 int ret0() {
5     return 0;
6 }
7
8 int ret1() {
9     return 1;
10 }
11
12 int tail_call_fun(int bool_val) {
13     if (bool_val) {
14         return ret0();
15     }
16     return ret1();
17 }
18
19 void main() {
20     print(tail_call_fun(input()));
21 }
```

Code 4.5: *Beispiel für Tail Call*

Partielle Evaluation und Lazy Evaluation wurden im PicoC-Compiler **nicht** implementiert, da dieser als **Lerntool** gedacht ist und diese Funktionalitäten den **RETI-Code** für Studenten **schwerer verständlich** machen könnten, da die **Codeschnipsel** und damit verbundene **Paradigmen** aus der Vorlesung **nicht** mehr so einfach **nachvollzogen** werden können und das **schwerere Ausmachen** können von **Orientierungspunkten** und **Fehlen erwarteter Codeschnipsel** leichter zur **Verwirrung** bei den Studenten führen könnte.

4.4 Fehlermeldungen

Die **Fehlerarten**, die der **PicoC-Compiler** ausgeben kann sind in den Tabellen 4.4, 4.5 und 4.6 und eingeteilt nach den Kategorien **Syntax**, **Semantik** und **Laufzeit** aus Unterkapitel 4.4.

Fehlerarten, wie z.B. `UninitialisedVariable` beim Verwenden einer **uninitialisierten Variable** oder `IndexOutOfBounds` bei **Feldzugriff** auf einen **Index**, der **außerhalb** des **Feldes** liegt gibt es für die Sprache L_{PicoC} **nicht**, da da bei der Programmiersprache L_C , die eine Obermenge der Programmiersprache L_{PicoC} ist diese Fehlermeldungen auch **nicht** gibt. Das Programm in Code 4.6 läuft z.B. **ohne Fehlermeldungen** durch.

```
1 #include <stdio.h>
2
3 void main() {
4     int var;
5     printf("\n%d", var);
6 }
```

Code 4.6: Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet

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

Tabelle 4.4: Syntaktische Fehlerarten

Fehlerart	Beschreibung
UnknownIdentifier	Es wird ein Zugriff auf einen Bezeichner gemacht (z.B. <code>unknown_var + 1</code>), der noch nicht deklariert und ist daher nicht in der Symboltabelle aufgefunden werden kann.
UnknownAttribute	Der Structtyp (z.B. <code>struct st {int attr1; int attr2;}</code>) auf dessen Attribut im momentanen Kontext zugegriffen wird (z.B. <code>var[3].unknown_attr</code>) besitzt das Attribut (z.B. <code>unknown_attr</code>) auf das zugegriffen werden soll nicht .
ReDeclarationDefinition	Ein Bezeichner ^a der bereits deklariert oder definiert ist (z.B. <code>int var</code>) wird erneut deklariert oder definiert (z.B. <code>int var[2]</code>). Dieser Fehler ist leicht festzustellen, indem geprüft wird ob das Assoziative Feld durch welches die Symboltabelle umgesetzt ist diesen Bezeichner bereits als Schlüssel besitzt.
ConstAssign	Wenn einer initialisierten Konstante (z.B. <code>const int const_var = 42</code>) ein Wert zugewiesen wird (z.B. <code>const_var = 41</code>). Der einzige Weg , wie eine Konstante einen Wert erhält ist bei ihrer Initialisierung .
TooLargeLiteral	Der Wert eines Literals ist größer als $2^{31} - 1$ oder kleiner als -2^{31} .
NotExactlyOneMainFunction	Das Programm besitzt keine oder mehr als eine main-Funktion .
PrototypeMismatch	Der Prototyp einer deklarierten Funktion (z.B. <code>int fun(int arg1, int arg2[3])</code>) stimmt nicht mit dem Prototyp der späteren Definition dieser Funktion (z.B. <code>void fun(int arg1[2], int arg2) { }</code>) überein.
ArgumentMismatch	Wenn die Argumente eines Funktionsaufrufs (z.B. <code>fun(42, 314)</code>) nicht mit dem Prototyp der Funktion die aufgerufen werden soll (z.B. <code>void fun(int arg[2]) { }</code>) nach Datentypen oder Anzahl Argumente bzw. Parameter übereinstimmt.
MissingReturn	Wenn eine Funktion, die ihrem Prototyp zufolge einen Rückgabewert hat, der nicht vom Datentyp <code>void</code> ist (z.B. <code>int fun() { ... }</code>) als letztes Statement kein return-Statement hat, dass einen Wert des entsprechenden Datentyps zurückgibt ^b .

^a Z.B. von einer **Funktion** oder **Variable**.

^b Der **entsprechende Datentyp** müsste auf das Beispiel von davor `void fun(int arg[2]) { ... }` bezogen z.B. `return 42` sein.

Tabelle 4.5: Semantische Fehlerarten

Fehlerart	Beschreibung
DivisionByZero	Wenn bei einer Division durch 0 geteilt wird (z.B. <code>var / 0</code>).

Tabelle 4.6: Laufzeit Fehlerarten

In Code 4.8 ist eine **typische Fehlermeldung** zu sehen. Eine **Fehlermeldung** fängt immer mit einem **Header** an, bei dem sich an den Fehlermeldungen des **GCC** orientiert wurde. Ein analoges Beispiel für eine **GCC-Fehlermeldung** für Code 4.8 ist in Code 4.7 zu sehen. Nacheinander stehen in Code 4.8 im **Header** der **Dateiname**, die **Position** des Fehlers in der Datei in der das fehlerhafte Programm steht, die **Fehlerart** und ein **Beschreibungstext**.

```

1 ./tests/error_wrong_written_keyword.c:8:5: error: expected 'while' before 'wile'
2   } wile (True);
3     ^~~~

```


Code 4.7: Fehlermeldung des GCC

Unter dem **Header** wird beim **PicoC-Compiler** ein kleiner **Ausschnitt des Programmes** um die Stelle herum an welcher der **Fehler aufgetreten** ist angezeigt. Die Kommandozeilenoptionen `-1` und `-c`, welche in Tabelle 4.1 erläutert werden könnten in diesem Zusammenhang interessant sein.

Das Symbol `~` bzw. eine Folge von `~` kennzeichnet beim **PicoC-Compiler** das **Token**, welches an der Stelle des Fehlers **vorgefunden** wurde und das Symbol `^` soll einen **Pfeil** symbolisieren, der auf eine **Position zeigt** an der ein **anderes Token**, ein **anderer Datentyp** usw. **erwartet** worden wäre und **in der Zeile darunter** eine **Beschriftung** an sich hängen hat, die konkret angibt, was dort eigentlich **erwartet** worden wäre.

```

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

Code 4.8: Beispiel für typische Fehlermeldung mit 'found' und 'expected'

Bei **Fehlermeldungen**, wie in Code 4.9, die ihre **Ursache** an einer **anderen Stelle im Code** haben, wird einmal ein **Header** mit **Programmausschnitt** für die **Stelle** an welcher der **Fehler aufgetreten** ist erstellt und ein weiterer **Header** mit **Programmausschnitt** für die **Stelle** welche die **Ursache** für das Auftreten dieses Fehlers ist.

```

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

Code 4.9: Beispiel Fehlermeldung langgestreckte fehlermeldung

Bei manchen Fehlermeldungen, wie in Code 4.10 ist es **garnicht möglich** mit `~` ein Token an der **Stelle** zu markieren, an welcher der **Fehler vorgefunden** wurde, da z.B. beim **UnexpectedEOF**-Fehler das **Ende der Programmes** erreicht wurde, wo es **kein sichtbares Token** gibt, welches man markieren könnte. Des Weiteren ist in Code 4.10 interessant, dass in markierten Zeile in Code 4.10 **mehrere Tokens** angegeben

werden, die nach der Grammatik 3.2.8 an dieser Stelle erwartet werden können. Es werden **standardmäßig** nur die **ersten 5 erwarteten Tokens** angegeben, aber mittels der Kommandozeilenoptionen `-vv` kann auch aktiviert werden, dass **alle möglichen Tokens** in einer solchen `or`-Kette angegeben werden.

```

1 ./tests/error_unexpected_eof.picoc:4:13: UnexpectedEOF: Unexpected end-of-file, expected e.g.
  ↳ reti comment or '}' or '(' or 'print' or '*' or 'if'.
2
3 void main() {
4     ^
5     reti comment or '}' or '(' or 'print' or '*' or 'if'

```

Code 4.10: *Beispiel für Fehlermeldung mit mehreren erwarteten Tokens*

Bei wiederum anderen Fehlermeldungen, wie in Code 4.11 ist es **nicht möglich** ein **erwartetes Token** anzugeben, da es sich um einen **Semantische Fehler** handelt und der Fehler **nicht** durch **brechen** mit der **Syntax** zustande gekommen ist, sondern auf das konkrete Beispiel in Code 4.11 bezogen daran liegt, dass die Variable `unknown_identifier` nicht definiert ist, weshalb sich hier keine **erwarteten Tokens** angeben lassen, da auf der **semantischen** Seite verlangt ist, dass diese Variable `unknown_identifier` eine **Bedeutung** hat, welche sie aufgrund dessen, dass sie **nicht definiert** ist nicht hat.

```

1 ./tests/error_unknown_identifier_ref.picoc:5:18: UnknownIdentifier: Identifier
  ↳ 'unknown_identifier' wasn't declared yet.
2
3 void main() {
4     int var = 42 + unknown_identifier;
5                 ~~~~~
6 }

```

Code 4.11: *Beispiel für Fehlermeldung ohne expected*

Bei z.B. dem **Laufzeit-Fehler** `DivisionByZero` wird beim Auftreten einer **Division durch 0** mit entsprechendem **RETI-Code** gecheckt, ob der **rechte Operand** einer **Divisionsoperation** eine 0 ist und wenn dies der Fall ist in das **ACC-Register** der Wert 1 geschrieben und die Programmausführung beendet. Der Wert 1 im **ACC-Register** stellt eine `DivisionByZero`-Fehlermeldung dar. Wenn es noch weitere **Laufzeit-Fehlerarten** gebe, dann würde eine 2 im **ACC-Register** für einen anderen **Laufzeit-Fehler** stehen usw.

Appendix

Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die zur **Erklärung** der Vorgehensweise zur Implementierung eines **üblichen Compilers** referenziert werden, aber **nichts** mit dem Vorgehen zur Implementierung des **PicoC-Compilers** zu tun haben.

Definition 5.1: 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 **Opcod**e 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. D. P. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 5.2) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinencode** bzw. **Objectcode** (Definition 5.3) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschinencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 5.4) zu Maschiendencod zusammengesetzt wird ausführbar ist.

Definition 5.2: Assembler

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

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

Definition 5.3: Objectcode

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

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

Definition 5.4: Linker

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

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

Definition 5.5: Transpiler (bzw. Source-to-source Compiler)

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

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

^bThiemann, „Compilerbau“.

Definition 5.6: Rekursiver Abstieg

Es wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminalsymbol** enthält.

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 5.7) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 5.6) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschaft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist¹.

Definition 5.7: 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“.

Definition 5.8: Liveness Analyse

Findet heraus, welche **Variablen** in welchen **Regionen** eines Programmes **verwendet** werden.^a

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

¹Mehr **Erklärung** hierzu findet sich im Unterkapitel 2.4.

Definition 5.9: Live Variable

Eine Location, deren momentaner Wert *später* im Programmablauf noch *verwendet* wird. Man sagt auch die Location ist *live*.^{a,b}

^aEs gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Live Variable**.

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

Definition 5.10: Graph Coloring

Problem bei dem den **Knoten** eines Graphen^a **Zahlen**^b zugewiesen werden sollen, sodass **keine** zwei **adjazente Knoten** die **gleiche Zahl** haben und **möglichst wenige** unterschiedliche Zahlen gebraucht werden.^{c,d}

^aIn Bezug zu Compilerbau ein **Ungerichteter Graph**.

^bBzw. **Farben**.

^cEs gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Graph Coloring**.

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

Definition 5.11: Interference Graph

Ein **ungerichteter Graph** mit **Locations** als **Knoten**, der eine **Kante** zwischen zwei Locations hat, wenn es sich bei beiden Locations **zu dem Zeitpunkt** um **Live Locations** handelt. In Bezug auf **Graph Coloring** bedeutet eine **Kante**, dass diese zwei Locations **nicht** die **gleiche Zahl**^a zugewiesen bekommen dürfen.^b

^aBzw. **Farbe**.

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

Definition 5.12: Kontrollflussgraph

Gerichteter Graph, der den Kontrollfluss eines Programmes beschreibt.^a

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

Definition 5.13: Kontrollfluss

Die **Reihenfolge** in der z.B. **Statements**, **Funktionsaufrufe** usw. eines Programmes ausgewertet werden^a.

^aMan geht hier von einem **imperativen** Programm aus.

Definition 5.14: Kontrollflussanalyse

Analyse des **Kontrollflusses** (Definition 5.13) eines **Programmes**, um herauszufinden zwischen welchen Teilen des Programms **Daten ausgetauscht** werden und welche **Abhängigkeiten** sich daraus ergeben.

Der **simpleste Ansatz** ist es in einen Kontrollflussgraph **iterativ** einen Algorithmus^a anzuwenden, bis sich an den Werten der Knoten **nichts** mehr **ändert**.^{b,c}

^aIm Bezug zu Compilerbau die **Linveness Analyse**.

^bBis diese sich **stabilisiert** haben

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

Definition 5.15: Two-Space Copying Collector

Ein *Garbage Collector* bei dem der *Heap* in *FromSpace* und *ToSpace* unterteilt wird und bei *nicht ausreichendem* Speicherplatz auf dem *Heap* alle Variablen, die in Zukunft noch verwendet werden vom *FromSpace* zum *ToSpace* kopiert werden. Der aktuelle *ToSpace* wird danach zum neuen *FromSpace* und der aktuelle *FromSpace* wird danach zum neuen *ToSpace*.^a

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

Bootstrapping

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ (Definition 5.16) 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 5.16: 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

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

Will man nun für eine Maschine M_{RETI} , auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 5.19) 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**² reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ auf der Maschine M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$, der mit der Maschinensprache B_{RETI} läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ nicht selbst komplett in der Maschinensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C_{PicoC}^{Python} , den man bereits in der Programmiersprache L_{Python} implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 5.18) darstellt, auf einer anderen Maschine M_{other} dafür nutzt, damit dieser den **Self-compiling Compiler** $C_{RETI_PicoC}^{PicoC}$ für die Maschine M_{RETI} kompiliert bzw. **bootstraped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine M_{other} auf die Maschine M_{RETI} kopiert.³

²Beschreibt 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.

³Im 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.

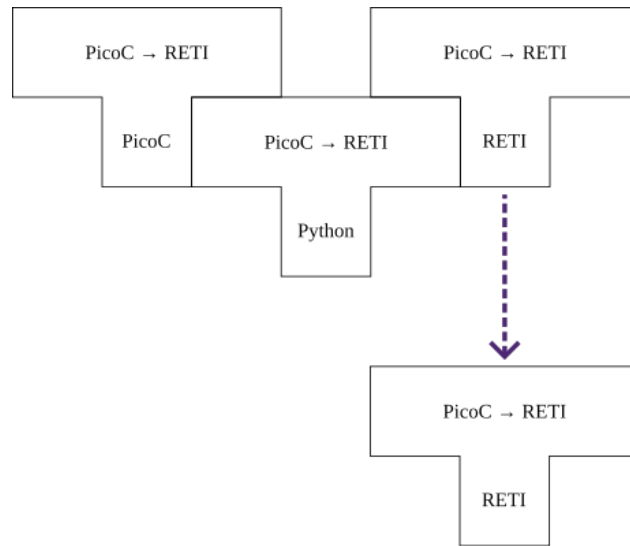


Abbildung 5.1: Cross-Compiler als Bootstrap Compiler

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 5.17: 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** $C_{w_i}^w$ des Self-compiling Compilers C_w^w wichtig sind.^{a,b}

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

^bThiemann, „Compilerbau“.

Definition 5.18: 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 5.17) 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.⁴

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem **minimalen Compiler** $C_{RETI_PicoC}^{PicoC}$, wie in Subdefinition 5.19.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Definition 5.19: 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 5.19.1 nochmal gesondert hervorzuheben:

5.19.1: 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.

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

⁴Natürlich 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} .

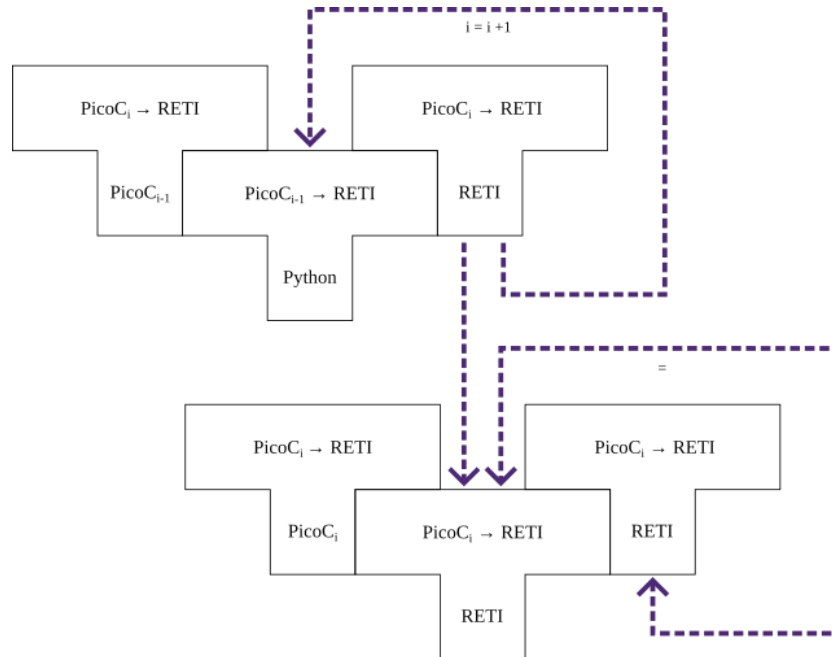


Abbildung 5.2: Iteratives Bootstrapping

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 5.19.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 5.19.1 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.

Danksagungen

Literatur

Online

- *A-Normalization: Why and How (with code)*. URL: <https://matt.might.net/articles/a-normalization/> (besucht am 23.07.2022).
- *ANSI C grammar (Lex)*. URL: <https://www.lysator.liu.se/c/ANSI-C-grammar-1.html> (besucht am 29.07.2022).
- *ANSI C grammar (Yacc)*. URL: <http://www.quot.com/c/ANSI-C-grammar-y.html> (besucht am 29.07.2022).
- *ANTLR*. URL: <https://www.antlr.org/> (besucht am 31.07.2022).
- *Bäume*. URL: <https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/> (besucht am 17.07.2022).
- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27.04.2022).
- *clang: C++ Compiler*. URL: <http://clang.org/> (besucht am 29.07.2022).
- *Clockwise/Spiral Rule*. URL: <https://c-faq.com/decl/spiral.anderson.html> (besucht am 29.07.2022).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: <https://inkscape.org/> (besucht am 03.08.2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: <https://www.geeksforgeeks.org/errors-in-cc/> (besucht am 10.05.2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *Grammar Reference — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/grammar.html> (besucht am 31.07.2022).
- *Grammar: The language of languages (BNF, EBNF, ABNF and more)*. URL: <https://matt.might.net/articles/grammars-bnf-ebnf/> (besucht am 30.07.2022).
- *History - GCC Wiki*. URL: <https://gcc.gnu.org/wiki/History> (besucht am 06.08.2022).
- *Home - Neovim*. URL: <http://neovim.io/> (besucht am 04.08.2022).
- *JSON parser - Tutorial — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09.07.2022).

- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: <https://reverseengineering.stackexchange.com/q/17671> (besucht am 13.04.2022).
- *Parsing Expressions · Crafting Interpreters*. URL: <https://www.craftinginterpreters.com/parsing-expressions.html> (besucht am 09.07.2022).
- *Transformers & Visitors — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/visitors.html> (besucht am 09.07.2022).
- *Welcome to Lark's documentation! — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/> (besucht am 31.07.2022).
- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: <https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/> (besucht am 18.07.2022).
- *What is Top-Down Parsing?* URL: <https://www.tutorialspoint.com/what-is-top-down-parsing> (besucht am 22.06.2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).
- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand*. 1. Aufl. Wiley, 20. Nov. 2012.

Artikel

- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: [10.1145/355598.362740](https://doi.org/10.1145/355598.362740).
- Earley, Jay. „An efficient context-free parsing“. In: 13 (1968). URL: <https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf> (besucht am 10.08.2022).

Vorlesungen

- Bast, Prof. Dr. Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09.07.2022).

- Scholl, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03.08.2022).
- Scholl, Prof. Dr. Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Prof. Dr. Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl> (besucht am 19.07.2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: [10.1145/1596638.1596640](https://doi.org/10.1145/1596638.1596640). URL: <http://portal.acm.org/citation.cfm?doid=1596638.1596640> (besucht am 23.07.2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).
- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03.08.2022).
- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: <https://github.com/lark-parser/lark> (besucht am 31.07.2022).
- *Syntax*. In: *Wiktionary*. Page Version ID: 9196998. 7. Juni 2022. URL: <https://de.wiktionary.org/w/index.php?title=Syntax&oldid=9196998> (besucht am 31.07.2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: <https://github.com/tueda/makefile4latex> (besucht am 03.08.2022).