Albert Ludwigs Universität Freiburg

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Autor: Jürgen Mattheis

Gutachter: Prof. Dr. Scholl

Betreung: M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für

Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Danksagungen

Bevor der Inhalt dieser Schrifftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie "mein aufrichtigster Dank" oder "aus tiefstem Herzen", sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das heraus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschafft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias¹ konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe², er hat sich bei der Korrektur dieser Schrifftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

¹Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

²Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil 3 weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler ausgeinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter Link⁵ zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt⁶. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁷. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiesprache kann in der Verwendung ziemlich wilkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

³Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

⁴Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes ³.

 $^{^5}$ https://github.com/michel-giehl/Reti-Emulator.

⁶Womit nicht alle Studenten so viel Glück haben.

⁷Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

\mathbf{A}	bbild	ungsverzeichnis	I
C	odeve	erzeichnis	II
Тa	abelle	enverzeichnis	\mathbf{V}
D	efinit	ionsverzeichnis	VII
\mathbf{G}	ramn	natikverzeichnis	VIII
1	Ein : 1.1	ührung RETI-Architektur	. 1 . 2
	1.1	Die Sprache PicoC	
	1.3	Eigenheiten der Sprachen C und PicoC	
	1.4	Gesetzte Schwerpunkte	
	1.5	Über diese Arbeit	
		1.5.1 Still der Schrifftlichen Ausarbeitung	
		1.5.2 Aufbau der Schrifftlichen Arbeit	. 15
2	The	oretische Grundlagen	16
	2.1	Compiler und Interpreter	
	0.0	2.1.1 T-Diagramme	
	2.2	Formale Sprachen	
		2.2.2 Präzedenz und Assoziativität	
	2.3	Lexikalische Analyse	
	2.4	Syntaktische Analyse	
	2.5	Code Generierung	
		2.5.1 Monadische Normalform	. 40
		2.5.2 A-Normalform	
		2.5.3 Ausgabe des Maschinencodes	
	2.6	Fehlermeldungen	. 45
3	_	lementierung	47
	3.1	Lexikalische Analyse	
		3.1.1 Konkrete Grammatik für die Lexikalische Analyse	
	3.2	Syntaktische Analyse	
	0.2	3.2.1 Umsetzung von Präzedenz und Assoziativität	
		3.2.2 Konkrete Grammatik für die Syntaktische Analyse	
		3.2.3 Ableitungsbaum Generierung	
		3.2.3.1 Codebeispiel	
		3.2.3.2 Ausgabe des Ableitunsgbaumes	
		3.2.4 Ableitungsbaum Vereinfachung	
		3.2.4.1 Codebeispiel	
		3.2.5 Generierung des Abstrakten Syntaxbaumes	
		3 2 5 1 PicoC-Knoten	66

		3.2.5.2 RETI-Knoten
		3.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung 7
		3.2.5.4 Abstrakte Grammatik
		3.2.5.5 Codebeispiel
		3.2.5.6 Ausgabe des Abstrakten Syntaxbaumes
3.3	Code	Generierung
	3.3.1	Passes
		3.3.1.1 PicoC-Shrink Pass
		3.3.1.1.1 Aufgabe
		3.3.1.1.2 Abstrakte Grammatik
		3.3.1.1.3 Codebeispiel
		3.3.1.2 PicoC-Blocks Pass
		3.3.1.2.1 Aufgabe
		3.3.1.2.2 Abstrakte Grammatik
		3.3.1.2.3 Codebeispiel
		3.3.1.3 PicoC-ANF Pass
		3.3.1.3.1 Aufgabe
		3.3.1.3.1 Augabe
		1
		3.3.1.4 RETI-Blocks Pass
		3.3.1.4.1 Aufgabe
		3.3.1.4.2 Abstrakte Grammatik
		3.3.1.4.3 Codebeispiel
		3.3.1.5 RETI-Patch Pass
		3.3.1.5.1 Aufgabe
		3.3.1.5.2 Abstrakte Grammatik
		3.3.1.5.3 Codebeispiel
		3.3.1.6 RETI Pass
		3.3.1.6.1 Aufgabe
		3.3.1.6.2 Konkrete und Abstrakte Grammatik
		3.3.1.6.3 Codebeispiel
	3.3.2	Umsetzung von Zeigern
		3.3.2.1 Referenzierung
		3.3.2.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen
	3.3.3	Umsetzung von Feldern
		3.3.3.1 Initialisierung eines Feldes
		3.3.3.2 Zugriff auf einen Feldindex
		3.3.3.3 Zuweisung an Feldindex
	3.3.4	Umsetzung von Verbunden
		3.3.4.1 Deklaration von Verbundstypen und Definition von Verbunden 119
		3.3.4.2 Initialisierung von Verbunden
		3.3.4.3 Zugriff auf Verbundsattribut
		3.3.4.4 Zuweisung an Verbundsattribut
	3.3.5	Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen 130
		3.3.5.1 Anfangsteil
		3.3.5.2 Mittelteil
		3.3.5.3 Schlussteil
	3.3.6	Umsetzung von Funktionen
		3.3.6.1 Mehrere Funktionen
		3.3.6.1.1 Sprung zur Main Funktion
		3.3.6.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen 15
		3.3.6.3 Funktionsaufruf
		3.3.6.3.1 Rückgabewert
		3.3.6.3.2 Umsetzung der Übergabe eines Feldes

	3.4	3.3.6.3.3 Umsetzung einer Übergabe eines Verbundes	65 69
4	Erg		7 3
	4.1	Funktionsumfang	73
		4.1.1 Kommandozeilen optionen	73
		4.1.2 Shell-Mode	76
		4.1.3 Show-Mode	78
	4.2	Qualitätssicherung	80
	4.3	Erweiterungsideen	84
A	ppen RET	dix I Architektur Details	A A
	Sons	tige Definitionen	С
	Boot	strapping	Η
Li	terat	ur	\mathbf{L}

Abbildungsverzeichnis

1.1	Schritte zum Ausführen eines Programmes mit dem GCC	1
1.2	Stark vereinfachte Schritte zum Ausführen eines Programmes	2
1.3	Speicherorganisation	4
1.4	README.md im Github Repository der Bachelorarbeit	13
2.1	Horinzontale Übersetzungszwischenschritte zusammenfassen	20
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen	
2.3		28
2.4		28
2.5	Veranschaulichung der Lexikalischen Analyse	31
2.6		38
2.7	Veranschaulichung der Syntaktischen Analyse	36
2.8	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten	11
2.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen	14
3.1	Ableitungsbäume zu den beiden Ableitungen	5 4
3.2	Ableitungsbaum nach Parsen eines Ausdrucks	32
3.3	Ableitungsbaum nach Vereinfachung	33
3.4	Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen	j5
3.5	Generierung eines Abstrakten Syntaxbaumes mit Umdrehen	
3.6		78
3.7	Cross-Compiler Kompiliervorgang Kurzform	
3.8	Architektur mit allen Passes ausgeschrieben	
3.9	Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen	31
4.1	Show-Mode in der Verwendung	3(
5.1	Datenpfade der RETI-Architektur	
5.2	Cross-Compiler als Bootstrap Compiler]
5.3		K

Codeverzeichnis

1.1	Beispiel für die Spiralregel.	7
1.2	Ausgabe des Beispiels für die Spiralregel	7
1.3	Beispiel für unterschiedliche Ausführung	7
1.4	Ausgabe des Beispiels für unterschiedliche Ausführung	7
1.5		8
1.6		8
1.7		9
1.8		9
1.9		9
	Ausgabe des Beispiels dafür, dass Zeiger auf Feld übergeben wird	
	Beispiel für Deklaration und Definition	
	Ausgabe des Beispiels für Deklaration und Definition	
	Beispiel für Sichtbarkeitsbereiche	
	Ausgabe des Beispiels für Sichtbarkeitsbereiche	
	Transparse des Besspiese für Steinesderfeitele.	
3.1	PicoC-Code des Codebeispiels	2
3.2	Tokens für das Codebeispiel	
3.3	Ableitungsbaum nach Ableitungsbaum Generierung	0
3.4	Ableitungsbaum nach Ableitungsbaum Vereinfachung	
3.5	Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum	
3.6	PicoC Code für Codebespiel	
3.7	Abstrakter Syntaxbaum für Codebespiel	
3.8	PicoC-Blocks Pass für Codebespiel	
3.9	PicoC-ANF Pass für Codebespiel	
	RETI-Blocks Pass für Codebespiel	
	RETI-Patch Pass für Codebespiel	8
	RETI Pass für Codebespiel	2
	PicoC-Code für Zeigerreferenzierung	3
	Abstrakter Syntaxbaum für Zeigerreferenzierung	3
	Symboltabelle für Zeigerreferenzierung	4
3.16	PicoC-ANF Pass für Zeigerreferenzierung	15
3.17	RETI-Blocks Pass für Zeigerreferenzierung	15
	PicoC-Code für Zeigerdereferenzierung	6
	Abstrakter Syntaxbaum für Zeigerdereferenzierung	6
3.20	PicoC-Shrink Pass für Zeigerdereferenzierung	7
	PicoC-Code für die Initialisierung eines Feldes	7
3.22	Abstrakter Syntaxbaum für die Initialisierung eines Feldes	8
3.23	Symboltabelle für die Initialisierung eines Feldes	9
	PicoC-ANF Pass für die Initialisierung eines Feldes	0
	RETI-Blocks Pass für die Initialisierung eines Feldes	1
3.26	PicoC-Code für Zugriff auf einen Feldindex	2
	Abstrakter Syntaxbaum für Zugriff auf einen Feldindex	2
	PicoC-ANF Pass für Zugriff auf einen Feldindex	4
	RETI-Blocks Pass für Zugriff auf einen Feldindex	6
	PicoC-Code für Zuweisung an Feldindex	6
	Abstrakter Syntaxbaum für Zuweisung an Feldindex	
	PicoC ANE Pass für Zuweisung an Foldinday	

3.33	RETI-Blocks Pass für Zuweisung an Feldindex	118
3.34	PicoC-Code für die Deklaration eines Verbundstyps	119
3.35	Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps	119
3.36	Symboltabelle für die Deklaration eines Verbundstyps	121
3.37	PicoC-Code für Initialisierung von Verbunden.	121
3.38	Abstrakter Syntaxbaum für Initialisierung von Verbunden	122
3.39	PicoC-ANF Pass für Initialisierung von Verbunden	123
	RETI-Blocks Pass für Initialisierung von Verbunden	124
	PicoC-Code für Zugriff auf Verbundsattribut.	124
	Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.	125
	PicoC-ANF Pass für Zugriff auf Verbundsattribut.	127
	RETI-Blocks Pass für Zugriff auf Verbundsattribut.	128
	PicoC-Code für Zuweisung an Verbundsattribut.	128
	Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.	129
	PicoC-ANF Pass für Zuweisung an Verbundsattribut.	129
	RETI-Blocks Pass für Zuweisung an Verbndsattribut.	130
	PicoC-Code für den Anfangsteil.	134
	Abstrakter Syntaxbaum für den Anfangsteil.	135
	PicoC-ANF Pass für den Anfangsteil.	135
	RETI-Blocks Pass für den Anfangsteil	136
	PicoC-Code für den Mittelteil.	136
	Abstrakter Syntaxbaum für den Mittelteil	137
	PicoC-ANF Pass für den Mittelteil	$137 \\ 137$
	RETI-Blocks Pass für den Mittelteil.	$137 \\ 139$
	PicoC-Code für den Schlussteil.	139
	Abstrakter Syntaxbaum für den Schlussteil	$140 \\ 141$
	RETI-Blocks Pass für den Schlussteil.	
	PicoC-Code für 3 Funktionen.	144
	Abstrakter Syntaxbaum für 3 Funktionen	145
	PicoC-Blocks Pass für 3 Funktionen.	146
	PicoC-ANF Pass für 3 Funktionen.	146
	RETI-Blocks Pass für 3 Funktionen.	147
	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	147
	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	148
		149
	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	
		150
	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	152
	PicoC-Code für Funktionsaufruf ohne Rückgabewert.	152
	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert	153
	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert	154
	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	155
	RETI-Pass für Funktionsaufruf ohne Rückgabewert	157
	PicoC-Code für Funktionsaufruf mit Rückgabewert.	157
	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert	158
	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert	159
	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	161
	PicoC-Code für die Übergabe eines Feldes	161
	Symboltabelle für die Übergabe eines Feldes	163
	PicoC-ANF Pass für die Übergabe eines Feldes	164
	RETI-Block Pass für die Übergabe eines Feldes	165
	PicoC-Code für die Übergabe eines Verbundes.	166
3.86	PicoC-ANF Pass für die Übergabe eines Verbundes	166

3.87	RETI-Block Pass für die Übergabe eines Verbundes	168
3.88	Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet	169
3.89	Fehlermeldung des GCC	171
3.90	Beispiel für typische Fehlermeldung mit 'found' und 'expected'	171
3.91	Beispiel für eine langgestreckte Fehlermeldung.	171
3.92	Beispiel für Fehlermeldung mit mehreren erwarteten Tokens.	172
	Beispiel für Fehlermeldung ohne expected	
4.1	Shellaufruf und die Befehle compile und quit	177
4.2	Shell-Mode und der Befehl most_used	178
4.3	Typischer Test	182
4.4	Testdurchlauf	184
4.5	Beispiel für Tail Call	187

Tabellenverzeichnis

Register der RETI-Architektur
Beispiele für Lexeme und ihre entsprechenden Tokens
Präzedenzregeln von PicoC
Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren
PicoC-Knoten Teil 1
PicoC-Knoten Teil 2
PicoC-Knoten Teil 3
PicoC-Knoten Teil 4
RETI-Knoten
Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung
Fehlerarten in der Lexikalischen und Syntaktischen Analyse
Fehlerarten in den Passes
Fehlerarten, die zur Laufzeit auftreten
Kommandozeilenoptionen, Teil 1
Kommandozeilenoptionen, Teil 2
Makefileoptionen
Testkategorien
Load und Store Befehle
Compute Befehle
Jump Befehle

Definitionsverzeichnis

1.1	Imperative Programmierung	
1.2	Strukturierte Programmierung	. (
1.3	Prozedurale Programmierung	. (
1.4	Call by Value	
1.5	Call by Reference	
1.6	Funktionsprototyp	. 10
1.7	Deklaration	. 10
1.8	Definition	
1.9	Sichtbarkeitsbereich (bzw. engl. Scope)	
2.1	Pipe-Filter Architekturpattern	
2.2	Interpreter	
2.3	Compiler	
2.4	Maschinensprache	
2.5	Immediate	
2.6	Cross-Compiler	
2.7	T-Diagram Programm	
2.8	T-Diagram Übersetzer (bzw. eng. Translator)	
2.9	T-Diagram Interpreter	
	T-Diagram Maschine	
	Symbol	
	Alphabet	
	Wort	
	Formale Sprache	
	Syntax	
	Semantik	
	Formale Grammatik	
	Chromsky Hierarchie	
	Reguläre Grammatik	
	Kontextfreie Grammatik	
	Wortproblem	
	1-Schritt-Ableitungsrelation	
	Ableitungsrelation	
2.24	Links- und Rechtsableitungableitung	. 25
	Linksrekursive Grammatiken	
	Formaler Ableitungsbaum	
	Mehrdeutige Grammatik	
	Assoziativität	
	Präzedenz	
	Lexeme	
	Token	
	Lexer (bzw. Scanner oder auch Tokenizer)	
	Literal	
	Konkrete Syntax	
	Konkrete Grammatik	
	Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)	
	Parser	
2.38	Erkenner (bzw. engl. Recognizer)	. 34

	Transformer	36
	Visitor	36
2.41	Abstrakte Syntax	36
2.42	Abstrakte Grammatik	36
2.43	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)	36
2.44	Pass	40
2.45	Reiner Ausdruck (bzw. engl. pure expression)	41
2.46	Unreiner Ausdruck	41
2.47	Monadische Normalform (bzw. engl. monadic normal form)	41
2.48	Location	42
	Atomarer Ausdruck	42
2.50	Komplexer Ausdruck	42
2.51	A-Normalform (ANF)	43
	Fehlermeldung	45
3.1	Metasyntax	47
3.2	Metasprache	47
3.3	Erweiterte Backus-Naur-Form (EBNF)	47
3.4	Dialekt der Erweiterten Backus-Naur-Form aus Lark	48
3.5	Abstrakte Syntaxform (ASF)	49
3.6	Earley Parser	59
3.7	Symboltabelle	87
3.8		127
5.1	Bezeichner (bzw. Identifier)	\mathbf{C}
5.2	Label	\mathbf{C}
5.3	Assemblersprache (bzw. engl. Assembly Language)	\mathbf{C}
5.4	Assembler	D
5.5	Objectcode	D
5.6	Linker	D
5.7	Transpiler (bzw. Source-to-source Compiler)	D
5.8	Rekursiver Abstieg	\mathbf{E}
5.9	$\mathrm{LL}(k) ext{-Grammatik}$	\mathbf{E}
5.10	Earley Erkenner	\mathbf{E}
5.11	Liveness Analyse	F
	Live Variable	F
5.13	Graph Coloring	G
	Interference Graph	G
	Kontrollflussgraph	G
5.16	Kontrollfluss	G
5.17	Kontrollflussanalyse	G
5.18	Two-Space Copying Collector	G
5.19	Self-compiling Compiler	Η
	Minimaler Compiler	I
	Boostrap Compiler	Ι
5 22	Bootstrapping	J.

Grammatikverzeichnis

2.1	Produktionen für einen Ableitungsbaum in EBNF	26
		38
2.3	Produktionen für Abstrakten Syntaxbaum in ASF	38
3.1.1	Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF	51
3.2.1	Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in	
	EBNF, die Operatorpräzidenz nicht beachtet	53
3.2.2	Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syn-	
		54
3.2.3		55
3.2.4	Beispiel für eine unäre linksassoziative Produktion in EBNF	55
3.2.5	Beispiel für eine binäre linksassoziative Produktion in EBNF	56
3.2.6	Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF	56
3.2.7	Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet	57
		58
3.2.9	Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	59
3.2.1	OAbstrakte Grammatik der Sprache L_{PiocC} in ASF $\dots \dots \dots \dots \dots \dots \dots \dots$	75
		81
3.3.2	Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF	84
		88
		91
		95
		99
		99
3.3.8	Abstrakte Grammatik der Sprache L_{RETI} in ASF \ldots 1	00

1 Einführung

Als Programmierer kommt man nicht drumherum einen Compiler zu nutzen, er ist geradezu essentiel für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprachen Python, welche als interpretierte Sprache bekannt ist, wird ein in der Programmiersprache Python geschriebenes Programm vorher zu Bytecode¹ kompiliert, bevor dieses von der Python Virtual Machine (PVM) interpretiert wird.

Anmerkung Q

Die Programmiersprache Python und jegliche andere Sprache wird fortan als L_{Python} bzw. als $L_{Name\ der\ Sprache}$ bezeichnet wird.

Compiler, wie der GCC² oder Clang³ werden üblicherweise über eine Commandline-Schnittstelle verwendet, welche es für den Benutzer unkompliziert macht ein Programm zu Maschinencode (Definition 2.4) zu kompilieren. Das Programm muss hierzu in der Sprache geschrieben sein, die der Compiler kompiliert⁴

Meist funktioniert das über schlichtes und einfaches Angeben der Datei, die das Programm enthält, welches kompiliert werden soll. Im Fall des GCC funktioiert das über pgc program.c -o machine_code 5. Als Ergebnis erhält man im Fall des GCC die mit der Option o selbst benannte Datei machine_code. Diese kann dann z.B. unter Unix-Systemen über nicht.



Abbildung 1.1: Schritte zum Ausführen eines Programmes mit dem GCC.

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

¹Dieser Begriff ist **nicht** weiter **relevant**.

² GCC, the GNU Compiler Collection - GNU Project.

 $^{^3}$ clang: C++ Compiler.

⁴Im Fall des GCC und Clang ist es die Programmiersprache L_C .

⁵Bei 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 5.5) 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.

Kapitel 1. Einführung 1.1. RETI-Architektur

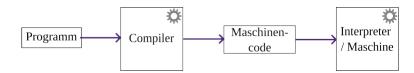


Abbildung 1.2: Stark vereinfachte Schritte zum Ausführen eines Programmes.

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 grundlegend funktioniert.

Die konkrete Aufgabe besteht darin einen sogenannten PicoC-Compiler zu implementieren, der die Programmiersprache L_{PicoC} in eine zu Lernzwecken prädestinierte, unkompliziert gehaltene Maschinensprache L_{RETI} kompilieren kann. Die Sprache L_{PicoC} ist hierbei eine Untermenge der äußerst bekannten Programmiersprache L_C , die der GCC kompilieren kann.

In dieser Einführung werden die für diese Bachelorarbeit elementare Thematiken erstmals angeschnitten und grundlgenden Informationen zu dieser Arbeit genannt. Gerade wurde das Thema dieser Bachelorarbeit veranschaulicht und die konkrette Aufgabenstellung ausformuliert. Im Unterkapitel 1.1 wird näher auf die RETI-Architektur eingegangen, die der Sprache L_{RETI} zugrunde liegt und im Unterkapitel 1.2 wird näher auf die die Sprache L_{PicoC} eingegangen, welche der PicoC-Compiler zur eben erwähnten Sprache L_{RETI} kompilieren soll. Des Weiteren wird in Unterkapitel 1.3 insbesondere auf bestimmte Eigenheiten der Sprachen L_C und L_{PicoC} eingegangen, auf welche in dieser Bachelorarbeit ein besonderes Augenmerk gerichtet wird. Danach wird in Unterkapitel 1.4 auf für diese Bachelorarbeit gesetzte Schwerpunkte eingegangen und in Unterkapitel 1.5 etwas zum Aufbau und Still dieser Schrifftlichen Ausarbeitung gesagt.

1.1 RETI-Architektur

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

Um den PicoC-Compiler zu testen war es notwendig einen RETI-Interpreter zu implementieren, der genau die Variante der RETI-Achitektur aus der Vorlesung C. Scholl, "Betriebssysteme" simuliert. Für genauere Implementierungsdetails der RETI-Architektur ist auf die Vorlesungen P. D. C. Scholl, "Technische Informatik" und C. Scholl, "Betriebssysteme" zu verweisen.

Anmerkung 9

In dieser Bachelorarbeit wird im Folgenden bei der Maschinensprache L_{RETI} immer von der Variante ausgegangen, welche durch die RETI-Architektur aus der Vorlesung C. Scholl, "Betriebssysteme" umgesetzt ist.

Die Register dieser RETI-Architektur werden in Tabelle ?? aufgezählt und erläutert. Der Befehlssatz und die Datenpfade der RETI-Architektur sind im Kapitel Appendix dokumentiert, da diese nicht explizit zum Verständnis der späteren Kapitel notwendig sind. Allerdings sind diese zum tieferen Verständnis

Kapitel 1. Einführung 1.1. RETI-Architektur

notwendig, um die später auftauchenden RETI-Befehle usw. zu verstehen. Der Aufbau der Maschinensprache L_{RETI} ist durch die Grammatiken 3.3.6 und 3.3.7 zusammengenommen beschrieben.

Register Kürzel	Register Ausgeschrieben	Aufgabe
PC	Program Counter	Zeigt auf den Maschinenbefehl, der als nächstes ausgeführt werden soll.
ACC	Accumulator	Für Operanden von Operationen oder für temporäre Werte.
IN1	Indexregister 1	Hat dieselbe Aufgabe wie das ACC-Register.
IN2	Indexregister 2	Hat dieselbe Aufgabe wie das ACC-Register.
SP	${f Stack pointer}$	Zeigt immer auf die erste freie Speicherzelle am Ende des Stacks ^a , wo als nächstes Speicher allokiert werden kann.
BAF	Begin Aktive Funktion	Zeigt auf den Beginn des Stackframes der aktuell aktiven Funktion.
CS	Codesegment	Zeigt auf den Beginn des Codesegments. Die letzten 10 Bits werden verwendet, um 22 Bit Immediates aufzufüllen. Kann dadurch dazu verwendet werden, festzulegen welcher der 3 Peripheriegeräte ^b in der Memory Map ^c angesprochen werden soll.
DS	Datensegment	Zeigt auf den Beginn des Datensegments.

^a Wird noch erläutert.

Tabelle 1.1: Register der RETI-Architektur.

Die RETI-Architektur ermöglicht es bei der Ausführung von RETI-Programmen Prozesse aufzubauen bzw. zu nutzen. In Abbildung 1.3 ist der Aufbau eines Prozesses im Hauptspeicher der RETI-Architektur zu sehen. Ein RETI-Programm nutzt dabei den Stack für temporäre Zwischenergebnisse von Berechnungen und zum Anlegen der Stackframes von Funktionen, welche die Lokalen Variablen und Parameter einer Funktion speichern. Das SP- und BAF-Register erfüllen dabei ihre in Tabelle 1.1 zugeteilten Aufgaben für den Stack.

Der Abschnitt für die Globalen Statischen Daten ist allgemein dazu da Daten zu beherbergen, die für den Rest der Programmausführung global zugänglich sein sollen, aber auch für die Lokalen Variablen der main-Funktion. Das DS-Register markiert den Anfang des Datensegments und damit auch die Anfangsadresse, ab der die Globalen Statischen Daten abgespeichert sind und kann als relativer Orientierungspunkt beim Zugriff und Abspeichern Globaler Statischer Daten dienen. Das CS-Register wird als relativer Orientierungspunkt genutzt, an dem die Ausführung von RETI-Programmen startet. Darüberhinaus wird das CS-Register dazu genutzt, die relative Startadresse zu bestimmen, an welcher der RETI-Code einer bestimmten Funktion anfängt. Der Heap ist nicht weiter relevant, da die Funktionalitäten der Sprache L_C , welche diesen nutzen in L_{PicoC} nicht enthalten sind.

^b EPROM, UART und SRAM.

^c Da die Memory Map zum Verständnis der Bachelorarbeit nicht wichtig ist, wird diese nicht mehr als nötig im weiteren Verlauf erläutert.

Kapitel 1. Einführung 1.2. Die Sprache PicoC

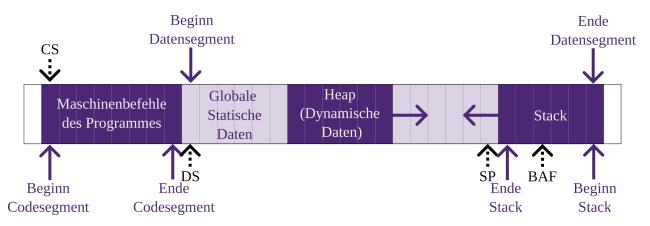


Abbildung 1.3: Speicherorganisation.

Die RETI-Architektur nutzt 3 verschiedene Peripheriegeräte, EPROM, UART und SRAM, die über eine Memory Map⁶ den über die Datenpfade der RETI-Architektur 5.1 ansprechbaren Adressraum von 2³² Adressen⁷ unter sich aufteilen.

Die Ausführung eines Programmes startet auf die einfachste Weise, indem es von einem Startprogramm im EPROM⁸ aufgerufen wird. Der EPROM wird beim Start einer RETI-CPU als erstes aufgerufen. Das liegt daran, dass bei der Memory Map der erste Adressraum von 0 bis $2^{30} - 1$ dem EPROM zugeordnet ist und das PC-Register initial den Wert 0 hat. Daher wird als erstes das Programm ausgeführt, welches an Adresse 0 im EPROM anfängt.

Die UART⁹ ist eine elektronische Schaltung mit je nach Umsetzung mehr oder weniger Pins. Es gibt allerdings immer einen RX- und einen TX-Pin, für jeweils Empfangen¹⁰ und Versenden¹¹ von Daten. Jeder der Pins wird dabei mit einer anderen von 2³ verschiedenen Adressen angsprochen. Jeweils 8-Bit können nach den Datenpfaden der RETI-CPU 5.1 auf einmal über einen Pin in ein Register der UART geschrieben werden, um versandt zu werden oder von einem Pin empfangen werden. Die UART kann z.B. genutzt werden, um Daten an einen sehr einfach gehaltenen Monitor zu senden, der diese dann anzeigt.

An letzter Stelle muss der \mathbf{SRAM}^{12} erwähnt werden, bei dem es sich um den Hauptspeicher der RETI-CPU handelt. Der Zugriff auf den Hauptspeicher ist deutlich schneller als z.B. auf ein externes Speichermedium, aber langsamer als der Zugriff auf Register. Die Datenmenge, die in einer Speicherzelle des Hauptspeichers abgespeichert ist, beträgt hierbei 32Bit = 1Byte. In der RETI-Architektur ist aufgrund dessen, dass es sich um eine 32-Bit Architektur handelt ein Datenwort 32Bit breit. Aus diesem Grund sind alle Register 32Bit groß, die Operanden der Arithmetische Logische Einheit¹³ nehmen 32Bit ein, die Befehle des Befehlssatzes sind innerhalb 32Bit codiert usw.

1.2 Die Sprache PicoC

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

⁶Da die Memory Map zum Verständnis der Bachelorarbeit nicht wichtig ist, sondern nur bei der Umsetzung des RETI-Interpreters, wird diese nicht näher erläutert als notwendig.

⁷Von 0 bis $2^{32} - 1$.

⁸Kurz für Erasable Programmable Read-Only Memory.

⁹Kurz für Universal Asynchronous Receiver Transmitter.

¹⁰Engl. Receiving, daher das R.

¹¹Engl. Transmission, daher das T.

 $^{^{12}\}mathrm{Kurz}$ für Static Random-Access Memory.

¹³Ist für Arithmetische und Logische Berechnungen zuständig.

- Einzeilige Kommentare // und Mehrzeilige Kommentare /* comment */.
- die Basisdatentypen¹⁴ int, char und void.
- die Zusammengesetzten Datentypen¹⁵ Felder (z.B. int ar[3]), Verbunde (z.B. struct st {int attr1; int attr2;}) und Zeiger (z.B. int *pntr) und ihre zugehörigen Operationen [i], .attr und * usw.
- if(cond){ }- und else{ }-Anweisungen¹⁶.
- while(cond){ }- und do while(cond){ };-Anweisungen.
- Arihmetische und Bitweise Ausdrücke, welche mithilfe der binären Operatoren +, -, *, /, %, &, |, ^, <<, >> und unären Operatoren -, ~ umgesetzt sind.¹⁷
- Logische Ausdrücke, welche mithilfe der Relationen ==, !=, <, >, <=, >= und Logischer Verknüpfungen !, &&, || umgesetzt sind.
- Zuweisungen, die mit dem Zuweisungsoperator = umgesetzt sind.
- Funktionsdeklaration (z.B. int fum(int arg1[3], struct st arg2);), Funktionsdefinition (z.B. int fum(int arg1[3], struct st arg2){}) und Funktionsaufrufe (z.B. fum(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 grundlegende Aufbau von Programmen der Programmiersprache L_{PicoC} ist durch Grammatik 3.1.1 und Grammatik 3.2.8 zusammengenommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige Eigenheiten der Programmiersprache L_C , die genauso ein Teil der Programmiersprache L_{PicoC} sind¹⁸, werden im Folgenden genauer erläutert. Diese Eigeneheiten werden in der Implementierung des PicoC-Compilers in Kapitel Implementierung noch eine wichtige Rolle spielen.

Anmerkung Q

Im Folgenden wird immer von der Programmiersprache L_{PicoC} gesprochen, da es in dieser Bachelorarbeit um diese geht und die folgenden Beispiele für die Ausgaben alle mithilfe des PicoC-Compilers und RETI-Interpreters kompiliert und daraufhin ausgeführt wurden. Aber selbiges gilt aus bereits erläutertem Grund genauso für L_C .

Bei der Programmiersprache L_{PicoC} handelt es sich im eine Imperative (Definition 1.1), Strukturierte (Definition 1.2) und Prozedurale Programmiersprache (Definition 1.3). Aufgrund dessen, dass es um eine Imperative Programmiersprache handelt ist es wichtig bei der Implementierung die Reihenfolge zu beachten.

Und aufgrund dessen, dass es um eine Strukturierte und Prozedurale Programmiersprache handelt,

 $^{^{14}\}mathrm{Bzw}$. int und char werden auch als Primitive Datentypen bezeichnet.

 $^{^{15} \}mathrm{Bzw.}$ engl. compound data types.

¹⁶Was die Kombination von if und else, nämlich else if(cond){} miteinschließt.

¹⁷Theoretisch sind die Operatoren <<, >> und ~ unnötig, da sie durch Multiplikation *, Division / und Anwendung des Xor-∧-Operators auf eine Zahl, deren binäre Repräsentation ein Folge von 1en gleicher Länge ist ersetzt werden können. 18 Da L_{PicoC} eine Untermenge von L_C ist.

ist es eine gute Methode bei der Implementierung auf Blöcke¹⁹ zu setzen, zwischen denen hin und her gesprungen werden kann. Blöcke stellen in den einzelnen Implementierungsschritten die notwendige Datenstruktur dar, um Auswahl zwischen Codestücken, Wiederholung von Codestücken und Sprünge zu Blöcken mit entsprechend zu bestimmten Bezeichnern (Definition 5.1) passenden Labeln (Definition 5.2) umzusetzen.

Definition 1.1: Imperative Programmierung

Z

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

^aThiemann, "Einführung in die Programmierung".

Definition 1.2: Strukturierte Programmierung



Wenn ein Programm anstelle von z.B. goto label-Anweisungen Kontrollstruturen, wie z.B. if (cond) {} else {}, while(cond) {} usw. verwendet, welche dem Programmcode mehr Struktur geben, weil die Auswahl zwischen Anweisungen und die Wiederholung von Anweisungen eine klare und eindeutige Struktur hat. Diese Struktur wäre bei der Umsetzung mit einer goto label-Anweisung nicht so eindeutig erkennbar und auch nicht umbedingt immer gleich aufgebaut wäre.^a

^aThiemann, "Einführung in die Programmierung".

Definition 1.3: Prozedurale Programmierung



Programme werden z.B. mittels Funktionen in überschaubare Unterprogramme^a aufgeteilt, die aufrufbar 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. Das macht den Code deutlich überschaubarer, da man die Aufgabe eines Codestücks nun nur noch mit seinem Bezeichner assoziieren muss.^b

 a Bzw. auch **Prozeduren** genannt.

 ${}^b\mathrm{Thiemann},$ "Einführung in die Programmierung".

In L_{PicoC} ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen [<i>]-Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: <remaining-datatype><var>[<i>], während andere Programmiersprachen die eckigen [<i>]-Klammern vor die Variable schreiben <remaining-datatype>[<i>]<var>.

Werden die eckigen [<i>]-Klammern hinter die Variable geschrieben, ist es schwieriger den Datentyp abzulesen, als auch ein Programm zu implementieren was diesen erkennt. Damit ein Programmierer den Datentyp ablesen kann, kann dieser die Spiralregel verwenden, die unter der Webseite Clockwise/Spiral Rule²⁰ nachgelesen werden kann. Werden die eckigen [<i>]-Klammern hinter die Variable geschrieben, wirken diese zum verwechseln ähnlich zum <var>[<ii>]-Operator für den Zugriff auf den Index eines Feldes. Wenn Ausdrücke, wie int ar[1] = {42} und var[0] = 42 vorliegen, sind var[1] und var[0] nur durch den Kontext um sie herum unterscheidbar.

In Code 1.1 ist ein Beispiel zu sehen, indem die Variable complex_var den Datentyp "Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Zeigern auf Felder der Mächtigkeit 2 von Verbunden vom Typ st" hat. Ein Vorteil davon die eckigen [<i>]-Klammern hinter die Variable zu schreiben ist in der markierten Zeile in Code 1.1 zu sehen. Will man auf ein Element dieses Datentyps zugreifen (*complex_var[0][1])[1].attr, so ist der Ausdruck fast genau gleich aufgebaut, wie der Ausdruck für den

 $^{^{19}}$ Werden später im Kapitel 3 genauer erklärt.

²⁰https://c-faq.com/decl/spiral.anderson.html

Datentyp struct st (*complex_var[1][2])[2]. Die Ausgabe des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

```
1 struct st {int attr;};
2
3 void main() {
4   struct st st_var[2] = {{.attr=314}, {.attr=42}};
5   struct st (*complex_var[1][2])[2] = {{&st_var}, &st_var}};
6   print((*complex_var[0][1])[1].attr);
7 }
```

Code 1.1: Beispiel für die Spiralregel.

```
1 42
```

Code 1.2: Ausgabe des Beispiels für die Spiralregel.

In L_{PicoC} ist die Ausführbarkeit einer Operation oder wie diese Operation ausgeführt wird davon abhängig, was für einen Datentyp die Variable im Kontext der auszuführenden Operation hat. In dem Beispiel in Code 1.3 wird in Zeile 2 ein "Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Integern" und in Zeile 3 ein "Zeiger auf Felder der Mächtigkeit 2 von Integern" erstellt. In den markierten Zeilen wird zweimal in Folge die gleiche Operation \volume var>[0] [1] ausgeführt, allerdings hat die Operation aufgrund der unterschiedlichen Datentypen der beiden Variablen einen unterschiedlichen Effekt.

In der markierten Zeile 4 wird ein normaler Zugriff auf den zweiten Eintrag im ersten Eintrag des Felds int ar[1][2] = {{314, 42}} durchgeführt. In der nachfolgend markierten Zeile 5 wird allerdings erst dem Zeiger int (*pntr)[2] = &ar[0]; gefolgt und dann ein Zugriff auf den zweiten Eintrag im ersten Eintrag des Felds int ar[1][2] = {{314, 42}} durchgeführt. Beide Operationen haben, wie in Code 1.4 zu sehen ist die gleiche Ausgabe.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(ar[0][1]);
5   print(pntr[0][1]);
6 }
```

Code 1.3: Beispiel für unterschiedliche Ausführung.

```
1 42 42
```

Code 1.4: Ausgabe des Beispiels für unterschiedliche Ausführung.

Eine weitere interessante Eigenheit, die in L_{PicoC} gültig ist, ist, dass die Operationen $\vert [0]$ [1] und $\vert (\vert)+1$) aus Code 1.3 und Code 1.5 komplett austauschbar sind. Die Ausgabe in Code 1.4 ist folglich identisch zur Ausgabe in Code 1.6.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(*(*(ar+0)+1));
5   print(*(*(pntr+0)+1));
6 }
```

Code 1.5: Beispiel mit Dereferenzierungsoperator.

```
1 42 42
```

Code 1.6: Ausgabe des Beispiels mit Dereferenzierungsoperator.

In der Programmiersprache L_{PicoC} werden alle Argumente bei einem Funktionsaufruf nach der Call by Value-Strategie (Definition 1.4) übergeben. Ein Beispiel hierfür ist in Code 1.7 zu sehen. Hierbei wird ein Verbund struct st copyable_ar = {.ar={314, 314}}; ²¹ an die Funktion fun übergeben. Hierzu wird der Verbund in den Stackframe der aufgerufenen Funktion fun kopiert und an den Parameter fun gebunden.

Wie an der Ausgabe in Code 1.7 zu sehen ist hat die Zuweisung copyable_ar.ar[1] = 42 an den Parameter struct st copyable_ar in der aufgerufenen Funktion fun keinen Einfluss auf die übergebene lokale Variable struct st copyable_ar = {.ar={314, 314}} der aufrufenden Funktion. Der Eintrag an Index 1 im Feld bleibt bei 314.

Definition 1.4: Call by Value

Bei einem Funktionsaufruf wird eine Kopie des Ergebnisses eines Ausdrucks, welcher ein Argument darstellt an den entsprechenden Parameter der aufgerufenen Funktion gebunden.

Das hat zur Folge, dass bei Übergabe einer Variable als Argument an eine Funktion, diese Variable bei Änderungen am entsprechenden Parameter der aufgerufenen Funktion in der aufrufenden Funktion unverändert bleibt. a

^aBast, "Programmieren in C".

```
1 struct st {int ar[2];};
2
3 int fun(struct st copyable_ar) {
4   copyable_ar.ar[1] = 42;
5 }
6
7 void main() {
8   struct st copyable_ar = {.ar={314, 314}};
```

²¹Später wird darauf eingegangen, warum der Verbund den Bezeichner copyable_ar erhalten hat.

```
print(copyable_ar.ar[1]);
fun(copyable_ar);
print(copyable_ar.ar[1]);
}
```

Code 1.7: Beispiel dafür, dass Struct kopiert wird.

```
1 314 314
```

Code 1.8: Ausgabe des Beispiel dafür, dass Struct kopiert wird.

In der Programmiersprache L_{PicoC} gibt es kein Call by Reference (Definition 1.5), allerdings kann der Effekt von Call by Reference mittels Zeigern simuliert werden, wie es in Code 1.11^{22} bei der Funktion fun_declared_before und dem Parameter int *param zu sehen ist. Genau dieser Trick wird bei Feldern verwendet, um nicht das gesamte Feld bei einem Funktionsaufruf in den Stackframe der aufgerufenen Funktion fun kopieren zu müssen.

Wie im Beispiel in Code 1.9 zu sehen ist, wird in der markierten Zeile ein Feld int ar [2] = {314, 314} an die Funktion fun übergeben. Wie in der Ausgabe in Code 1.10 zu sehen ist, hat sich der Eintrag an Index 1 im Feld durch die Zuweisung ar [1] = 42 nach dem Funktionsuaufruf zu 42 geändert. Wird ein Feld direkt als Ausdruck ar ohne z.B. die eckigen []-Klammern für einen Indexzugriff hingeschrieben, wird die Adresse des Felds verwendet und nicht z.B. der Wert des ersten Elements des Felds.

Eine Möglichkeit ein Feld als Kopie und nicht als Referenz zu übergeben ist es, wie in Code 1.7 bei der Variable copyable_ar das Feld als Attribut eines Verbundes zu übergeben.

Definition 1.5: Call by Reference

I

Bei einem Funktionsaufruf wird eine implizite Referenz eines Arguments an den entsprechenden Parameter der aufgerufenen Funktion gebunden.

Implizit meint hier, dass der Benutzer einer Funktionalität mit Call by Reference nicht mitbekommt, dass er das Argument selbst verändert und keine lokale Kopie des Arguments.^a

^aBast, "Programmieren in C".

```
1 int fun(int ar[2]) {
2    ar[1] = 42;
3 }
4
5 void main() {
6    int ar[2] = {314, 314};
7    print(ar[1]);
8    fun(ar);
9    print(ar[1]);
10 }
```

²²Unten im Code schauen.

Code 1.9: Beispiel dafür, dass Zeiger auf Feld übergeben wird.

1 314 42

Code 1.10: Ausgabe des Beispiels dafür, dass Zeiger auf Feld übergeben wird.

Ein Programm in der Programmiersprache L_{PicoC} wird von oben-nach-unten ausgewertet. Ein Problem tritt auf, wenn z.B. eine Funktion verwendet werden soll, die aber erst unter dem entsprechenden Funktionsaufruf definiert (Definition 1.8) wird. Es ist wichtig, dass der Prototyp (Definition 1.6) einer Funktion vor dem Funktionsaufruf dieser Funktion bekannt ist. Das hat den Sinn, dass bereits während des Kompilierens überprüft werden kann, ob die beim Funktionsaufruf übergebenen Argumente den gleichen Datentyp haben, wie die Parameter des Prototyps und ob die Anzahl Argumente mit der Anzahl Parameter des Prototyps übereinstimmt.

Allerdings lassen sich Funktionen nicht immer so anordnen, dass jede in einem Funktionsaufruf aufzurufende Funktion vorher definiert sein kann. Aus diesem Grund ist es möglich den Prototyp einer Funktion vorher zu deklarieren (Definition 1.7), wie es in den markierten Zeile im Beispiel in Code 1.11 zu sehen ist. Die Ausgabe des Beispiels ist in Code 1.12 zu sehen.

Definition 1.6: Funktionsprototyp

Z

Deklaration einer Funktion, welche den Funktionsbezeichner, die Datentypen der einzelnen Funktionsparameter, die Parametereihenfolge und den Rückgabewert einer Funktion spezifiziert. Es ist nicht möglich zwei Funktionsprototypen mit dem gleichen Funktionsbezeichner zu haben. ab

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

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

Definition 1.7: Deklaration

Z

Der Datentyp bzw. Prototyp einer Variablen bzw. Funktion, sowie der Bezeichner dieser Variable bzw. Funktion wird dem Compiler mitgeteilt. ab c

 a Über das Schlüsselwort **extern** lassen sich in der Programiersprache L_C Veriablen deklarieren, ohne sie zu definieren.

 b Variablen in C und C++, Deklaration und Definition — Coder-Welten.de.

^cP. Scholl, "Einführung in Embedded Systems".

Definition 1.8: Definition



Dem Compiler wird mitgeteilt, dass zu einem bestimmten Zeitpunkt in der Programmausführung Speicher für eine Variable angelegt werden soll und wo^a dieser angelegt werden soll. Eine Funktion ist definiert, sobald ihr eine relative Anfangsadresse im Hauptspeicher zugewiesen werden kann, aber welcher die Maschinenbefehle für diese Funktion abgespeichert werden können.^{bc}

"Im Fall des PicoC-Compilers im Abschnitt für die Globalen Statischen Daten oder auf dem Stack.

 b Variablen in C und C++, Deklaration und Definition — Coder-Welten.de.

^cP. Scholl, "Einführung in Embedded Systems".

```
void fun_declared_before(int *param);

int fun_defined(int param) {
   return param + 10;
}

void main() {
   int res = fun_defined(22);
   fun_declared_before(&res);
   print(res);
}

void fun_declared_before(int *param) {
   *param = *param + 10;
}
```

Code 1.11: Beispiel für Deklaration und Definition.

```
1 42
```

Code 1.12: Ausgabe des Beispiels für Deklaration und Definition.

In L_{PicoC} lässt sich eine Variable nur innerhalb ihres Sichtbarkeitsbereichs (Definition 1.9) verwenden. Lokale Variablen und Parameter lassen sich nur innerhalb der Funktion in welcher sie definiert wurden verwenden. Der Sichtbarkeitsbereich von Lokalen Variablen und Parametern erstreckt sich hierbei von der öffnenden {-Klammer bis zur schließenden }-Klammer der Funktionsdefinition, in welcher sie definiert wurden.

Verschiedene Sichtbarkeitsbereiche können dabei identische Bezeichner besitzen. Im Beispiel in Code 1.13 kommt der markierte Bezeichner local_var in 2 verschiedenen Sichtbarkeitsbereichen vor und bezeichnet somit 2 unterschiedliche Variablen. Der Parameter param und die Lokale Variable local_var dürfen nicht den gleichen Bezeichner haben, da sie sich im gleichen Sichtbarkeitsbereich der Funktion fun_scope befinden. Die Ausgabe des Beispiels in Code 1.13 ist in Code 1.14 zu sehen.

```
Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)

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

Thiemann, "Einführung in die Programmierung".
```

```
int fun_scope(int param) {
   int local_var = 2;
   print(param);
   print(local_var);
}

void main() {
   int local_var = 4;
   fun_scope(local_var);
}
```

Code 1.13: Beispiel für Sichtbarkeitsbereiche.

1 4 2

Code 1.14: Ausgabe des Beispiels für Sichtbarkeitsbereiche.

1.4 Gesetzte Schwerpunkte

Ein Schwerpunkt dieser Bachelorarbeit war es in erster Linie bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinensprache L_{RETI} die Syntax und Semantik der Programmiersprache L_C identisch nachzuahmen. Der PicoC-Compiler sollte die Programmiersprache L_{PicoC} im Vergleich zu z.B. dem GCC^{23} ohne merklichen Unterschied²⁴ kompilieren können.

In zweiter Linie sollte dabei möglichst immer so Vorgegangen werden, wie es die RETI-Codeschnipsel aus der Vorlesung 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 geschriebenen 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.

Beim PicoC-Compiler wurde eher darauf Wert gelegt sauberen und strukturierten Code zu schreiben, den interessierte Studenten bei Interesse selber nachvollziehen können und eine unkomplizierte Bibliothek mit guter Dokumentation²⁵, nämlich das Lark Parsing Toolkit²⁶ für das Parsen (Definition 2.37) zu verwenden. Und wie man auch beim Ausführen der Tests (wie in Unterkapitel 4.2 beschrieben²⁷) sieht, macht die Laufzeit des Compilers für übliche und auch längere Programme, wie ein Student sie zu Lernzwecken mit dem Compiler kompilieren würde absolut keine Probleme.

1.5 Über diese Arbeit

Der Quellcode des PicoC-Compilers ist öffentlich unter Link²⁸ zu finden. In der Datei README.md (siehe Abbildung 1.4) 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. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der letzte Commit vor der Abgabe der Bachelorarbeit ist unter Link²⁹ zu finden.

²³Da die Sprache L_{PicoC} eine Untermenge von L_C ist, kann der GCC L_{PicoC} ebenfalls kompilieren, allerdings nicht in die gewünschte Maschinensprache L_{RETI} .

²⁴Natürlich mit Ausnahme der sich unterscheidenden Maschinensprachen zu welchen kompiliert wird und der unterschiedlichen Commandline-Optionen und Fehlermeldungen.

 $^{^{25}}$ Welcome to Lark's documentation! — Lark documentation.

²⁶Lark - a parsing toolkit for Python.

²⁷Und wie es auch im **Koloquium** gezeigt werden wird.

 $^{^{28} \}verb|https://github.com/matthejue/PicoC-Compiler.$

 $^{^{29} \}texttt{https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971.}$

Kapitel 1. Einführung 1.5. Über diese Arbeit



Abbildung 1.4: README.md im Github Repository der Bachelorarbeit.

Die Schrifftliche 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 Schrifftliche Ausarbeitung dieser Bachelorarbeit ist als PDF unter Link³⁰ zu finden. Die PDF der Schrifftliche Ausarbeitung der Bachleorararbeit wird aus dem Latexquellcode automatisch mithife der Github Action Nemec, copy_file_to_another_repo_action und der Makefile Ueda, Makefile for LaTeX generiert. Der Latexquellcode ist unter Link³¹ veröffentlicht.

Alle verwendeten Latex Bibltiotheken sind unter Link³² zu finden³³. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vektorgraphikeditors Inkscape³⁴ erstellt. Falls Interesse besteht Grafiken aus dieser Schrifftlichen 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 Link³⁵ zu finden.

Um die verschiedenen Aspekte der Bachelorarbeit besser erklären zu können, werden Codebeispiele verwendet. In diesem Kapitel Einführung werden Codebeispiele zur Anschauung verwendet. Mithilfe des in den PicoC-Compiler integrierten RETI-Interpreters werden Ausgaben erzeugt, die in dieses Dokument eingelesen wurden. In Kapitel Implementierung werden kleine repräsentative PicoC-Programme in wichtigen Zwischenstadien der Kompilierung in Form von Codebeispielen gezeigt³⁶.

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

 $^{^{30} {\}tt https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf}.$

³¹https://github.com/matthejue/Bachelorarbeit.

 $^{^{32}}$ https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.

 $^{^{33}}$ Jede einzelne verwendete Latex Bibliothek einzeln anzugeben wäre allerdings etwas zu aufwendig

 $^{^{34}}$ Developers, $Draw\ Freely$ — Inkscape.

 $^{^{35}}$ https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

³⁶Also die verschiedenen in den Passes generierten Abstrakten Syntaxbäume, sofern der Pass für den gezeigten Aspekt relevant ist. Später mehr dazu.

Kapitel 1. Einführung 1.5. Über diese Arbeit

hier zur Repräsentation verwendeten PicoC-Programme lassen sich unter dem Link³⁷ finden. Mithilfe der im Ordner /code_examples beiliegenden /Makefile und dem Befehl > make compile-all lassen sich die Codebeispiele genauso kompilieren, wie sie hier dargestellt sind³⁸.

1.5.1 Still der Schrifftlichen Ausarbeitung

In dieser Schrifftliche 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 es hat auch den Zweck, dass 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 geanu so gemacht wurde und nicht anders". 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 unter anderem zum innitialen wirklichen Verständnis eines Themas beantwortet braucht³⁹, die Frage des "warum".

Zum Verweis auf Quellen an denen sich z.B. bei der Formulierung von Definitionen in Definition's-Kästen orientiert wurde, wurden um den Lesefluss nicht zu stören Fußnoten⁴⁰ verwendet. Die meisten Definitionen wurden in eigenen Worten formuliert, damit die Definitionen untereinander 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.

In den sonstigen Fußnoten befinden sich Informationen, die vielleicht beim Verständnis helfen oder kleinere Details enthalten, die bei tiefgreifenderem Interesse interessant sein könnten. Im Allgemeinen werden die Informationen in den Fußnoten allerdings nicht zum Verständnis der Bachelorarbeit benötigt.

Des Weiteren gibt es Anmerkung 's-Kästen, welche kleine Anmerkungen enhalten, die über Konventionen aufklären sollen, vor Fallstricken warnen, die leicht zur Verwirrung führen können oder Informationen bei tiefergehenderem Interesse oder für den besseren Überblick enthalten. Der Inhalt dieser Anmerkung 's-Kästen ist allerdings zum Verständnis dieser Arbeit nicht essentiel wichtig.

Es wurde immer versucht möglichst deutsche Fachbegriffe zu verwenden, sofern sie einigermaßen geläufig sind und bei der Verwendung nicht eher verwirren⁴¹. Bei Code und anderem Text, dessen Zweck nicht dem Erklären dient, sondern der Veranschaulichung, wurde dieser konsequent in Englisch geschrieben bzw. belassen. Der Grund hierfür ist unter anderem, da die Bezeichner in der Implementierung des PicoC-Compilers, wie es mehr oder weniger Konvention beim Programmieren ist in Englisch benannt sind und diese Bezeichner in den Ausgaben des PicoC-Compilers vorkommen⁴².

 $^{^{37} {\}tt https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.}$

³⁸Es wurde zu diesem Zweck die Command-line Option -t, --thesis erstellt, die bestimmte Kommentare herausfiltert, damit die generierten Abstrakten Syntaxbäume in den verschiedenen Zwischenstufen der Kompilierung nicht zu überfüllt mit Kommentaren sind.

 $^{^{39}\}mathrm{Vor}$ allem Anfang, wo der Leser wenig über das Thema weiß.

 $^{^{40}\}mathrm{Das}$ ist ein Beispiel für eine Fußnote.

⁴¹Bei dem z.B. auch im Deutschen geläufigen Fachbegriff "Statement" war es eine schwierige Entscheidung, ob man nicht das deutsche Wort "Anweisung" verwenden soll. Da es nicht verwirrend klingt wurde sich dazu entschieden überall das deutsche Wort "Anweisung" zu verwenden.

⁴²Später werden unter anderem sogenannte Abstrakte Syntaxbäume (Definition 2.43) zur Veranschaulichung gezeigt, die vom PicoC-Compiler als Zwischenstufen der Kompilierung generiert werden. Diese Abstrakten Syntaxbäume sind in der Implementierung des PicoC-Compilers in Englisch benannt, daher wurden ihre Bezeichner in Englisch belassen.

Kapitel 1. Einführung 1.5. Über diese Arbeit

1.5.2 Aufbau der Schrifftlichen Arbeit

Der Inhalt dieser Schrifftlichen Ausarbeitung der Bachelorarbeit ist in 4 Kapitel unterteilt: Einführung, Theoretische Grundlagen, Implementierung und Ergebnisse und Ausblick. Zusätzlich gibt es noch den Appendix.

Das momentane Kapitel Einführung hatte den Zweck einen Einstieg in das Thema dieser Bachelorarbeit zu geben. Der Aufbau dieses Kapitels wurde zu Beginn bereits erläutert.

Im Kapitel Theoretische Grundlagen werden die notwendigen Theoretischen Grundlagen eingeführt, die zum Verständnis des Kapitels Implementierung notwendig sind. Die Theoretischen Grundlagen umfassen die wichtigsten Definitionen und Zusammenhänge 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 dieser Schrifftlichen 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.

Im Kapitel Implementierung werden die einzelnen Aspekte der Implementierung des PicoC-Compilers erklärt. Das Kapitel ist unterteilt in die verschiedenen Phasen der Kompilierung, nach dennen das Kapitel Einführung ebenfalls unterteilt ist. Dadurch, dass die Kapitel Theoretische Grundlagen und Implementierung eine ähliche Kapiteleinteilung haben, ist es besonders einfach zwischen beiden hin und her zu wechseln.

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 wie erwartet. Zum Schluss wird auf Erweiterungsideen eingegangen, bei denen es interessant wäre diese noch im PicoC-Compiler zu implementieren.

Im Appendix werden einige Details der RETI-Architektur, Sonstigen Definitionen und das Thema Bootstrapping angesprochen. Der Appendix dient als eine Lagerstätte für Definitionen, Tabellen, Abbildungen und ganze Unterkapitel, die bei Interesse zur weiteren Vertiefung da sind und zum Verständis der anderen Kapitel nicht notwendig sind. Damit der Rote Faden in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit erkennbar bleibt und der Lesefluss nicht gestört wird, wurden alle diese Informationen in den Appendix ausgelaggert.

Die Sonstigen Defintionen und das Thema Bootstrapping sind dazu da den Bogen von der spezifischen Implementierung des PicoC-Compilers wieder zum allgemeinen Vorgehen bei der Implementierung eines Compilers zu schlagen. Generell wurde immer versucht Parallelen zur Implementierung echter Compiler zu ziehen. Die Erklärungen und Definitionen hierfür wurden allerdings in den Appendix ausgelaggert. Der Zweck des PicoC-Compilers ist es primär ein Lerntool zu sein, weshalb Methoden, wie Liveness Analyse (Definition 5.11) usw., die in echten Compilern zur Anwendung kommen nicht umgesetzt wurden. Es sollte sich an die vorgegebenen Paradigmen aus der Vorlesung C. Scholl, "Betriebssysteme" gehalten werden.

2 Theoretische Grundlagen

In diesem Kapitel wird auf die Theoretischen Grundlagen eingegangen, die zum Verständnis der Implementierung in Kapitel 3 notwendig sind. Zuerst wird in Unterkapitel 2.1 genauer darauf eingegangen was ein Compiler und Interpreter eigentlich sind und damit in Verbindung stehende Begriffe erklärt. Danach wird in Unterkapitel 2.2 eine kleine Einführung zu einem der Grundpfeiler des Compilerbau, den Formalen Sprachen gegeben. Danach werden die einzelnen Filter des üblicherweise bei der Implementierung von Compilern genutzten Pipe-Filter-Architekturpatterns (Definition 2.1) nacheinander erklärt. Die Filter beinhalten die Lexikalische Analyse 2.3, Syntaktische Analyse 2.4 und Code Generierung 2.5. Zum Schluss wird in Unterkapitel 2.6 darauf eingegangen in welchen Situationen Fehlermeldungen auszugeben sind.

Definition 2.1: Pipe-Filter Architekturpattern

!

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

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

Eine Pipe stellt ein Bindeglied zwischen zwei Filtern dar. ab



^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 Anlehung an z.B. die Pipes aus Unix, z.B. cat /proc/bus/input/devices | less zum Ausdruck gebracht werden. Und so banal es klingt, sollen manche Bezeichnungen von Pattern auch einfach nur gut klingen.

2.1 Compiler und Interpreter

Die wohl wichtigsten zu klärenden Begriffe, sind die eines Compilers (Definition 2.3) und eines Interpreters (Definition 2.2), 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 GCC¹ und von Tests die Beziehungen in 2.3.1 zu belegen (siehe Subkapitel 4.2).

^bWestphal, "Softwaretechnik".

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

Definition 2.2: Interpreter

Z

Interpretiert die Befehle^a oder Anweisungen eines Programmes P direkt.

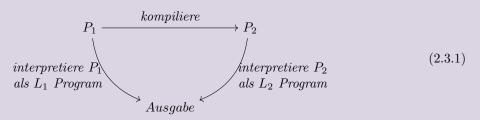
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen Sub-Bäumen des Abstrakten Syntaxbaumes (wird später eingeführt unter Definition 2.43) und führt je nach Komposition der Knoten des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.^b

^aMaschinensprache kann genauso interpretiert werden, wie auch eine Programmiersprache.

Definition 2.3: 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, sie die gleiche Ausgabe haben, wie es in Diagramm 2.3.1 dargestellt ist. Also beide Programme P_1 und P_2 die gleiche Semantik (Definition 2.16) haben und sich nur syntaktisch (Definition 2.15) durch die Sprachen L_1 und L_2 , in denen sie geschrieben stehen unterscheiden.



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

Üblicherweise kompiliert ein Compiler ein Program, das in einer Programmiersprache geschrieben ist zu Maschinencode, der in Maschinensprache (Definition 2.4) geschrieben ist, aber es gibt z.B. auch Transpiler (Definition 5.7) oder Cross-Compiler (Definition 2.6). Des Weiteren sind Maschinensprache und Assemblersprache (Definition 5.3) voneinander zu unterscheiden.

Definition 2.4: Maschinensprache

Z

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, genauergesagt 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 entworfen, dass sie sich innerhalb bestimmter Wortbreiten, die Zweierpotenzen sind kodieren lassen. Im einfachsten Fall innerhalb einer Speicherzelle des Hauptspeichers.

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

^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 2.5)

 $[^]b\mathrm{C}.$ Scholl, "Betriebssysteme".

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

Der PicoC-Compiler, der den Zweck erfüllt für Studenten ein Anschauungs- und Lernwerkzeug zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in menschenlesbarer Form mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.5) enthält. Für den RETI-Interpreter ist es ebenfalls nicht notwendig, dass der Maschinencode, den 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 2.5: Immediate

Z

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

^aLjohhuh, What is an immediate value?

Definition 2.6: Cross-Compiler



Kompiliert auf einer Maschine M_1 ein Program, dass in einer Sprache L_w geschrieben ist für eine andere Maschine M_2 , wobei beide Maschinen M_1 und M_2 unterschiedliche Maschinensprachen B_1 und B_2 haben.

^aBeim PicoC-Compiler handelt es sich um einen Cross-Compiler C_{PicoC}^{Python} , der in der Sprache L_{Python} geschrieben ist und die Sprache L_{PicoC} kompiliert.

^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 der Wissenschaftlichen Publikation 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 Program (Definition 2.7), einen Übersetzer (Definition 2.8), einen Interpreter (Definition 2.9) und eine Maschine (Definition 2.10) zusammen.

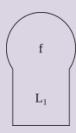
²Eine RETI-CPU zu bauen, die menschenlesbaren Maschinencode in z.B. UTF-8 Kodierung ausführen kann, wäre dagegen unnötig kompliziert und aufwändig, da Hardware binär arbeitet und man dieser daher lieber direkt die binär kodierten Maschinenbefehle übergibt, anstatt z.B. eine unnötig platzverbrauchenden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

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

Definition 2.7: T-Diagram Programm

/

Repräsentiert ein Programm, dass in der Sprache L₁ geschrieben ist und die Funktion f berechnet.^a



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

Anmerkung Q

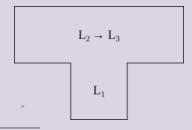
Es ist bei T-Diagrammen nicht notwendig beim entsprechenden Platzhalter, in den man die genutzte Sprache schreibt, den Namen der Sprache an ein L dranzuhängen, weil hier immer eine Sprache steht. Es würde in Definition 2.7 also reichen einfach eine 1 hinzuschreiben.

Definition 2.8: T-Diagram Übersetzer (bzw. eng. Translator)

I

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 2.3.1.^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 Übsersetzen, aber Übersetzen ist nicht immer auch Kompilieren.

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

Definition 2.9: T-Diagram Interpreter

Z

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

 L_2

 L_1

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

Definition 2.10: T-Diagram Maschine

Z

Repräsentiert eine Maschine, welche ein Programm in Maschinensprache L_1 ausführt. ab



^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).

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

Sowohl horinzontale als auch vertikale Adjazenz lassen sich, wie man in den Abbildungen 2.1 und 2.2 erkennen kann zusammenfassen.

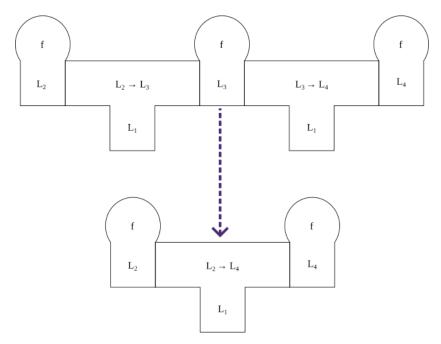


Abbildung 2.1: Horinzontale Übersetzungszwischenschritte zusammenfassen.

 $[^]b\mathrm{J}.$ Earley und Sturgis, "A formalism for translator interactions".

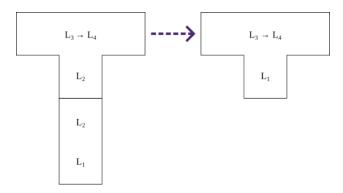


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen.

2.2 Formale Sprachen

Das Kompilieren eines Programmes hat viel mit dem Thema Formaler Sprachen (Definition 2.14) zu tuen, 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 2.11), Alphabet (Definition 2.12), Wort (Definition 2.13) beinhaltet vorher eingeführt zu haben.



Definition 2.14: Formale Sprache

Z

"Eine Formale Sprache ist eine Menge von Wörtern (Definition 2.13) über dem Alphabet Σ (Definition 2.12). "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 2.16) gleich bleiben. Beide Sprachen L_1 und L_2 haben eine Grammatik (Definition 2.17), welche diese beschreibt und können verschiedene Syntaxen (Definition 2.15) haben.

Definition 2.15: Syntax



Bezeichnet alles was mit dem Aufbau von Wörtern einer Formalen Sprache zu tuen hat. Eine Formale Grammatik, aber auch in Natürlicher Sprache ausgedrückte Regeln können die Syntax einer Sprache beschreiben. 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 2.16: Semantik



Die Semantik bezeichnet alles was mit der Bedeutung von Wörtern einer Formalen Sprache zu tuen hat.^a

^aThiemann, "Einführung in die Programmierung".

Definition 2.17: Formale Grammatik



"Eine Formale Grammatik beschriebt wie Wörter einer Sprache abgeleitet werden können.

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 = Nicht-Terminalsymbole.
- $\Sigma = Terminal symbole$, wobei $N \cap \Sigma = \emptyset$.
- $P = Menge\ von\ Produktionsregeln\ w \to v,\ wobei\ w, v \in (N \cup \Sigma)^*\ und\ w \notin \Sigma^*.^{cd}$
- S = 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.

Es ist möglich zwei Grammatiken G_1 und G_2 in einer Vereinigungsgrammatik $G_1 \uplus G_2 = \langle N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S ::= S_1 \mid S_2\}, S \rangle$ zu vereinigen. "ef

Die gerade definierten Formale Sprachen lassen sich des Weiteren in Klassen der Chromsky Hierarchie (Definition 2.18) einteilen.

Definition 2.18: Chromsky Hierarchie

Z

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 Entscheidungeprobleme, die in dieser Klasse entscheidbar bzw. unentscheidbar sind usw.

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

Zwischen den Sprachmengen benachbarter Klassen in Abbildung 2.18.1 besteht eine echte Teilmengenbeziehung: $L_3 \subset L_2 \subset L_1 \subset L_0$. Jede Reguläre Sprache ist auch eine Kontextfreie Sprache, aber nicht jede Kontextfreie Sprache ist auch eine Reguläre Sprache.^a

Formale Sprachen

Rekursiv Aufz. Sprachen (Typ 0)

Kontextsensitive Sprachen (Typ 1)

Kontextfreie Sprachen (Typ 2)

Reguläre Sprachen (Typ 3)

^aNebel, "Theoretische Informatik".

Für diese Bachelorarbeit sind allerdings nur die Spracheklassen der Chromsky-Hierarchie relevant, die von Regulären (Definition 2.19) und Kontextfreien Grammatiken (Definition 2.20) beschrieben werden.

Definition 2.19: Reguläre Grammatik

Z

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon$$
 (2.19.1)

haben, wobei A, B Nicht-Terminalsymbole sind und c ein Terminalsymbol ist^{ab}."c

^aDiese Definition einer Regulären Grammatik ist rechtsregulär, es ist auch möglich diese Definition linksregulär zu

^aWeil mit ihnen terminiert wird.

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

 $[^]cw$ muss mindestens ein Nicht-Terminalsymbol enthalten.

^dBzw. $w, v \in C^*$ und $w \notin \Sigma^*$.

^eDie Produktion $S := S_1 \mid S_2$ kann hierbei durch beliebige andere Produktionen ersetzt werden, welche die beiden Grammatiken miteinander verbinden.

^fNebel, "Theoretische Informatik".

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 2.20: Kontextfreie Grammatik

Z

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

$$A \to v \tag{2.20.1}$$

 $haben,\ wobei\ A\ ein\ Nicht-Terminal symbol\ ist\ und\ v\ ein\ beliebige\ Folge\ von\ Grammatik symbolen^a$ ist. "b

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

 b Nebel, "Theoretische Informatik".

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des Wortproblems (Definition 2.21). 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 2.21: Wortproblem



Ein Entscheidungeproblem, 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 $w \notin L$.

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

$$\mathbb{1}_L: \Sigma^* \to \{0, 1\}: w \mapsto \begin{cases} 1 & falls \ w \in L \\ 0 & sonst \end{cases}$$
 (2.21.1)

^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 **Sprach**e des Compilers abzuleiten. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 2.22) und der normalen **Ableitungsrelation** (Definition 2.23) unterschieden.

Definition 2.22: 1-Schritt-Ableitungsrelation

Z

"Eine binäre Relattion \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 unterschieden.

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".

⁴Bzw. Wort.

Definition 2.23: 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 $u = w_1 \Rightarrow \ldots \Rightarrow w_n = v$, wobei $n \geq 1$ und $w_1, \ldots, w_n \in (N \cup \Sigma)^*$. "a

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**⁵ kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.24) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines **Programmes** in Unterkapitel 2.4 relevant.

Definition 2.24: Links- und Rechtsableitungableitung



"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

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 2.25) ist⁶.

Definition 2.25: Linksrekursive Grammatiken



 $Eine\ Grammatik\ ist\ linksrekursiv,\ wenn\ sie\ ein\ Nicht-Terminal symbol\ enth\"{a}lt,\ dass\ linksrekursiv\ ist.$

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

Um herauszufinden, ob eine Grammatik mehrdeutig (Definition 2.27) ist, werden Ableitungen als Formale Ableitungsbäume (Definition 2.26) 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 2.26: Formaler Ableitungsbaum



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

^aNebel, "Theoretische Informatik".

^aNebel, "Theoretische Informatik".

^aParsing Expressions · Crafting Interpreters.

⁵Bzw. Wort.

 $^{^6}$ Für den im PicoC-Compiler verwendeten Earley Parsers stellt dies allerdings kein Problem dar.

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 Datentstruktur zur einfachen Weiterverarbeitung umsetzt.

Den Knoten dieses Baumes sind Grammatiksymbole $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.17) zugeordnet. Die Inneren Knoten des Baumes sind Nicht-Terminalsymbole N und die Blätter sind entweder Terminalsymbole Σ oder das leere Wort ε .

In Abbildung 2.26.2 ist ein Beispiel für einen Formalen Ableitungsbaum zu sehen, der sich aus der Ableitung 2.26.1 nach den im Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit (Definition 3.4) angegebenen Produktionen 2.1 einer Grammatik $G = \langle N, \Sigma, P, add \rangle$ ergibt.

DIG_NO_0	::=	"1" "2" "3" "4" "5" "6" "6" "7" "8" "9"	L _ Lex
DIG_WITH_0 NUM	::=	"0" DIG_NO_0 "0" DIG_NO_0 DIG_WITH_0*	
ADD_OP MUL_OP	::=		
$egin{array}{c} mul \ add \end{array}$::= ::=	$mul\ MUL_OP\ NUM\ \ NUM$ $add\ ADD_OP\ mul\ \ mul$	$L_{-}Parse$

Grammatik 2.1: Produktionen für einen Ableitungsbaum in EBNF

Anmerkung Q

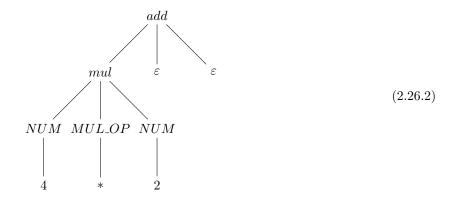
Werden die Produktionen einer Grammatik in z.B. EBNF angegeben, wie in Grammatik 3.1.1, wird die Angabe dieser Produktionen auch oft als Grammatik bezeichnet, obwohl Grammatiken eigentlich durch ein Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt sind.

$$add \Rightarrow mul \Rightarrow mul \ MUL_OP \ NUM \Rightarrow NUM \ MUL_OP \ NUM \Rightarrow "4" "*" "2"$$
 (2.26.1)

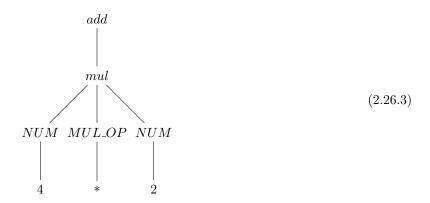
Bei Ableitungsbäumen gibt es keine einheutliche Regelung, wie damit umgegangen wird, wenn die Alternativen einer Produktion unterschiedliche viele Nicht-Terminalsymbole enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 2.26.2 von der Maximalzahl auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der Differenz zur Maximalzahl viele Blätter mit dem leeren Wort ε hinzuzufügen.

^aZ.B. Programmcode.

^bNebel, "Theoretische Informatik".



Eine andere Möglichkeit ist, wie im Ableitungsbaum 2.26.3 nur die vorhandenen Nicht-Terminalsymbole als Kinder hinzuzufügen⁷.



Für einen Compiler ist es notwendig, dass die Konkrete Grammatik keine Mehrdeutige Grammatik (Definition 2.27) ist, denn sonst können unter anderem die Präzedenzregeln der verschiedenen Operatoren nicht gewährleistet werden, wie später in Unterkapitel 3.2.1 an einem Beispiel demonstriert wird.

Definition 2.27: Mehrdeutige Grammatik

"Eine Grammatik ist mehrdeutig, wenn es ein Wort $w \in L(G)$ gibt, das mehrere Ableitungsbäume zulässt". ab

 a Alternativ, wenn es für w mehrere unterschiedliche Linksableitungen gibt.

2.2.2 Präzedenz und Assoziativität

Will man die Operatoren aus einer Programmiersprache in einer Konkreten Grammatik ausdrücken, die nicht mehrdeutig ist, so lässt sich das nach einem klaren Schema machen, wenn die Assoziativität (Definiton 2.28) und Präzedenz (Definition 2.29) dieser Operatoren festgelegt ist. Dieses Schema wird in Unterkapitel 3.2.1 genauer erklärt.

 $[^]b\mathrm{Nebel},$ "Theoretische Informatik".

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

Definition 2.28: Assoziativität

Z

"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.

Bei Assoziativität ist z.B. der Multitplikationsoperator * 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 2.29: Präzedenz



"Bestimmt, welcher Operator zuerst in einem Ausdruck, der eine Mischung verschiedener Operatoren enthält, ausgewertet wird. Operatoren mit einer höheren Präzedenz, werden vor Operatoren mit niedrigerer Präzedenz ausgewertet."

^aParsing Expressions · Crafting Interpreters.

Bei Präzedenz ist die Mischung der Operatoren für Subraktion '-' und für Multiplikation * ein Beispiel für den Einfluss von Präzedenz. 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 Präzedenz im Spiel.



Abbildung 2.4: Veranschaulichung von Präzedenz.

2.3 Lexikalische Analyse

Die Lexikalische Analyse bildet üblicherweise den ersten Filter innerhalb des Pipe-Filter Architekturpatterns (Definition 2.1) bei der Implementierung von Compilern. Die Aufgabe der Lexikalischen Analyse ist vereinfacht gesagt in einem Eingabewort⁸ endliche Folgen von Symbolen⁹ zu finden, die durch eine reguläre Grammatik erkannt werden. Diese Folgen endlicher Symoble werden auch Lexeme (Definition 2.30) genannt.

Definition 2.30: Lexeme

Z

Ein Lexeme ist ein Teilwort aus dem Eingabewort, welches unter einer Grammatik G_{Lex} abgeleitet werden kann.^a

^aThiemann, "Compilerbau".

Diese Lexeme werden vom Lexer (Definition 2.32) im Eingabewort identifziert und Tokens (Definition 2.31) zugeordnet. Das jeweils nächste Lexeme fängt dabei genau nach dem letzten Symbol des Lexemes an, das zuletzt vom Lexer erkannt wurde. Die Tokens sind es, die letztendlich an die Syntaktische Analyse weitergegeben werden.

Definition 2.31: Token



Ist ein Tupel (T, W) mit einem Tokentyp T und einem Tokenwert W. Ein Tokentyp T kann hierbei als ein Oberbegriff für eine möglicherweise unendliche Menge verschiedener Tokenwerte W verstanden werden^a.

 a Z.B. gibt es viele verschiedene Tokenwerte, wie z.B. 42, 314 oder 12, welche alle unter dem Tokentyp NUM, für Zahl zusammengefasst sind.

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



Ein Lexer ist eine partielle Funktion $lex : \Sigma^* \to (T \times W)^*$, welche ein Lexeme aus Σ^* auf ein Token (T,W) abbildet.

 $^a{\rm Thiemann},$ "Compilerbau".

Ein Lexer ist im Allgemeinen eine partielle Funktion, da es Zeichenfolgen geben kann, die sich unter der Grammatik G_{Lex} nicht ableiten lassen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine Fehlermeldung (Definition 2.52) ausgegeben.

Anmerkung Q

Um Verwirrung verzubeugen ist es wichtig folgende Unterscheidung hervorzuheben:

Wenn von Symbolen die Rede ist, so werden in der Lexikalischen Analyse, der Syntaktischen 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 Tokentypen die Symbole.

In der Code Generierung sind die Bezeichner (Definition 5.1) 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.

⁸Z.B. dem Inhalt einer Datei, welche in UTF-8 kodiert ist.

⁹Also Teilwörter des Eingabeworts.

Eine weitere Aufgabe der Lekikalischen Analyse ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen $_{-}$, Newline 10 und Tabs 10 und Eingabewort herauszufiltern. Das geschieht mittels des Lexers, der allen für die Syntaktische Analyse unwichtige Zeichen das leere Wort 10 zuordnet. Das ist auch im Sinne der Definition, denn 10 ein 10 ist immer der Fall bei der Kleeneschen Hülle 10 , wobei 10 10 10 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 Tokentyp T und Tokenwert W, ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen und auch Zahlen beliebige Zeichenfolgen sein können. Später in der Syntaktischen Analyse in Unterkapitel 2.4 wird sich nur dafür interessiert, ob an einer bestimmten Stelle ein bestimmter Tokentyp T, z.B. eine Zahl NUM steht und der Tokenwert W ist erst wieder in der Code Generierung in Unterkapitel 2.5 relevant.

Wie in Tabelle 2.1 zu sehen, gibt es für Bezeichner, wie my_fun, my_var oder my_const und verschiedenen Zahlen, wie 42, 314 oder 12 passende Tokentypen NAME¹¹ und NUM^{12 13 14}. Für Lexeme, wie if oder } sind die Tokentypen dagegen genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich IF und RBRACE.

Lexeme	Token
42, 314	Token('NUM', '42'), Token('NUM', '314')
<pre>my_fun, my_var, my_const</pre>	Token('NAME', 'my_fun'), Token('NAME', 'my_var'), Token('NAME',
	'my_const')
if , }	Token('IF', 'if'), Token('RBRACE', '}')
99, 'c'	Token('NUM', '99'), Token('CHAR', '99')

Tabelle 2.1: Beispiele für Lexeme und ihre entsprechenden Tokens.

Ein Lexeme ist nicht immer das gleiche wie der Tokenwert, denn wie in Tabelle 2.1 zu sehen ist, kann z.B. im Fall von L_{PicoC} der Wert 99 durch zwei verschiedene Literale (Definition 2.33) dargestellt werden, einmal als ASCII-Zeichen 'c', das dann als Tokenwert den entsprechenden Wert aus der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99^{15} . Der Tokenwert ist der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Anmerkung Q

Die Konkrete Grammatik G_{Lex} , die zur Beschreibung der Tokens T der Sprache L_{Lex} verwendet wird ist üblicherweise regulär, da ein typischer Lexer immer nur ein Symbol vorausschaut^a, sich nichts merkt, also unabhängig davon, was für Symbole und wie oft bestimmte Symbole davor aufgetaucht sind funktioniert. Auch für den PicoC-Compiler lässt sich aus der im Dialekt der Backus-Naur-Form des Lark Parsing Toolkit (Definition 3.4) spezifizierten Grammatik 3.1.1 schlussfolgern, dass die Sprache des PicoC-Compilers für die Lexikalische Analyse L_{PicoC_Lex} regulär ist, da alle ihre Produktionen die Definition 2.19 erfüllen.

Produktionen mit Alternative, wie z.B. DIG_WITH_0 ::= "0" | DIG_NO_0 sind unproblematisch,

¹⁰In Unix Systemen wird für Newline das ASCII Symbol line feed, in Windows hingegen die ASCII Symbole carriage return und line feed nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.

¹¹Für z.B. my_fun, my_var und my_const.

 $^{^{12}}$ Für z.B. 42, 314 und 12.

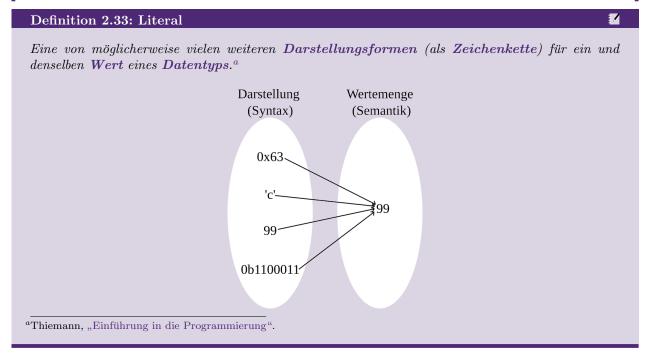
¹³Diese Tokentypen wurden im PicoC-Compiler verwendet, da man beim Programmieren möglichst kurze und leicht verständliche Bezeichner für seine Knoten haben will, damit unter anderem mehr Code in eine Zeile passt.

 $^{^{14}\}mathrm{Bzw}.$ wenn man sich nicht Kurzformen sucht <code>IDENTIFIER</code> und <code>NUMBER</code>.

 $^{^{15}}$ Die Programmiersprache L_{Python} erlaubt es z.B. den Wert 99 auch mit den Literalen 0b1100011 und 0x63 darzustellen.

denn sie können immer auch als $\{DIG_WITH_0 ::= "0", DIG_WITH_0 ::= DIG_NO_0\}$ ausgedrückt werden und z.B. DIG_WITH_0*, (LETTER | DIG_WITH_0 | "_")+ und "_"..."~" in Grammatik 3.1.1 können alle zu Alternativen umgeschrieben werden, womit diese Alternativen wie gerade gezeigt umgeformt werden können, um ebenfalls regulär zu sein. Somit existiert mit der Grammatik 3.1.1 eine reguläre Grammatik, welche die Sprache L_{PicoC_Lex} beschreibt und damit ist die Sprache L_{PicoC_Lex} nach der Chromsky Hierarchie (Definition 2.18) regulär.

^aMan nennt das auch einem Lookahead von 1.



Um eine Gesamtübersicht über die Lexikalische Analyse zu geben, ist in Abbildung 2.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

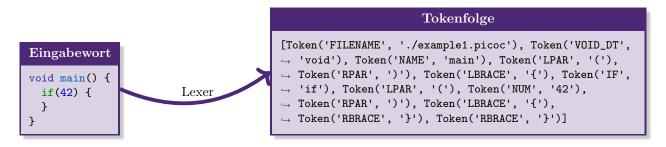


Abbildung 2.5: Veranschaulichung der Lexikalischen Analyse.

Anmerkung 9

Das Symbol \hookrightarrow zeigt im Code der Tokens in Abbildung 2.5 und in den folgenden Codes einen Zeilenumbruch an, wenn eine Zeile zu lang ist.

2.4 Syntaktische Analyse

In der Syntaktischen Analyse ist für einige Sprachen eine Kontextfreie Grammatik G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für Funktionsaufrufe fun(arg) und Codeblöcke if(1){} syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern '(' bzw. öffnende geschweifte Klammern '{'} es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden. Dies lässt sich nicht mehr mit einer Regulären Grammatik (Definition 2.19) beschreiben, sondern es braucht eine Kontextfreie Grammatik (Definition 2.20) hierfür, die es erlaubt zwischen zwei Terminalsymbolen ein Nicht-Terminalsymbol abzuleiten.

Für den PicoC-Compiler lässt sich aus der Grammatik 3.2.8 schlussfolgern, dass die Sprache des PicoC-Compilers für die Syntaktische Analyse L_{PicoC_Parse} kontextfrei, aber nicht mehr regulär ist, da alle ihre Produktionen die Definition für Kontextfreie Grammatiken 2.20 erfüllen, aber nicht die Definition für Reguläre Grammatiken 2.19.

Dass die Grammatik kontextfrei ist lässt sich auch sehr leicht erkennen, weil alle Produktionen auf der linken Seite des ::=-Symbols immer nur ein Nicht-Terminalsymbol haben und auf der rechten Seite eine beliebige Folge von Grammatiksymbolen 16 . Dass diese Grammatik aber nicht regulär sein kann, lässt sich sehr einfach an z.B. der Produktion $if_stmt ::= "if""("logic_or")" \ exec_part$ erkennen, bei der das Nicht-Terminalsymbol $logic_or$ von den Terminalsymbolen für öffnende Klammer { und schließende Klammer } eingeschlossen sein muss, was mit einer Regulären Grammatik nicht ausgedrückt werden kann

Somit existiert mit der Grammatik 3.2.8 eine Kontextfreie Grammatik und nicht Reguläre Grammatik, welche die Sprache L_{PicoC_Parse} beschreibt und damit ist die Sprache L_{PicoC_Parse} nach der Chromsky Hierarchie (Definition 2.18) kontextfrei, aber nicht regulär.

Die Syntax, in welcher ein Programm aufgeschrieben ist, wird auch als Konkrete Syntax (Definition 2.34) bezeichnet. In einem Zwischenschritt, dem Parsen wird aus diesem Programm mithilfe eines Parsers (Definition 2.37) ein Ableitungsbaum (Definition 2.36) generiert, der als Zwischenstufe hin zum einem Abstrakten Syntaxbaum (Definition 2.43) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des Ableitungsbaumes und dann erst des Abstrakten Syntaxbaumes.

Definition 2.34: Konkrete Syntax

Steht für alles, was mit dem Aufbau von nach einer Konkreten Grammatik (Definition 2.35) abgeleiteten Wörtern^a zu tuen hat.

Die Konkrete Syntax ist die Teilmenge der gesamten Syntax einer Sprache, welche für die Lexikalische und Syntaktische Analyse relevant ist. In der gesamten Syntax einer Sprache^b kann es z.B. Wörter geben, welche die gesamte Syntax nicht einhalten, die allerdings korrekt nach der Konkreten Grammatik abgeleitet sind^c.

Ein Programm in seiner Textrepräsentation, wie es in einer Textdatei nach der Konkreten Grammatik $G_{Lex} \uplus G_{Parse}^{\ d}$ abgeleitet steht, bevor man es kompiliert, ist in Konkreter Syntax aufgeschrieben.^e

^aBzw. Programmen.

^bVor allem bei **Programmiersprachen**.

^cWenn ein Programm z.B. nicht deklarierte Variablen hat und aufgrund dessen nicht kompiliert werden kann, hält dieses die gesamten Syntax nicht ein, kann allerdings so nach der Konkreten Grammatik abgeleitet werden.

 $[^]d$ Vereinigungsgrammatik wie in Definition 2.17 erklärt.

¹⁶Also eine beliebige Folge von Nicht-Terminalsymbolen und Terminalsymbolen.

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

Um einen kurzen Begriff für die Grammatik zu haben, welche die Konkrete Syntax einer Sprache beschreibt, wird diese im Folgenden als Konkrete Grammatik (Definition 2.35) bezeichnet.

Definition 2.35: Konkrete Grammatik

Grammatik, die eine Konkrete Syntax einer Sprache beschreibt und die Grammatiken G_{Lex} und G_{Parse} miteinander vereinigt: $G_{Lex} \uplus G_{Parse}^{a}$.

In der Konkreten Grammatik entsprechen die Terminalsymbole den Tokentypen, der in der Lexikalischen Analyse generierten Tokens^b und Nicht-Terminalsymbole entsprechen bei einem Ableitungsbaum den Stellen, wo ein Teilbaum eingehängt ist.

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

Compilerinterne Datenstruktur für den Formalen Ableitungsbaum (Definition 2.26) eines in Konkreter Syntax geschriebenen Programmes.

Die Blätter, die beim Formalen Ableitungsbaum Terminalsymbole einer Konkretten Grammatik $G_{Lex} \uplus G_{Parse}^a$ sind, sind in dieser Datenstruktur Tokens. In dieser Datenstruktur werden allerdings nur die Ableitungen eines Formales Ableitungsbauemes dargestellt, die sich aus den Produktionen einer Grammatik G_{Parse} ergeben. Die Tokens sind in der Syntaktischen Analyse ein atomarer Grundbaustein^b, daher sind die Ableitungen der Grammatik G_{Lex} uninteressant.

Die Konkrete Grammatik nach der Ableitungsbaum konstruiert ist, wird optimalerweise immer so definiert, dass sich möglichst einfach aus dem Ableitungsbaum ein Abstrakter Syntaxbaum konstruieren lässt.

Definition 2.37: Parser



 $Ein\ Parser\ ist\ ein\ Programm,\ dass\ aus\ einem\ Eingabewort^a,\ welches\ in\ Konkreter\ Syntax\ geschrieben\ ist\ eine\ compilerinterne\ Datenstruktur,\ den\ Ableitungsbaum\ generiert,\ was\ auch\ als\ Parsen\ bezeichnet\ wird^b.^c$

Anmerkung Q

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

^aVereinigungsgrammatik wie in Definition 2.17 erklärt.

^bWobei das Lark Parsing Toolkit, welches später bei der Implementierung verwendet wird eine spezielle Metasyntax zur Spezifikation von Grammatiken nutzt, bei der für bestimmten häufig genutzte Terminalsymbolen ein Tokenwert in die Grammatik geschrieben wird.

^aVereinigungsgrammatik wie in Definition 2.17 erklärt.

^bNicht mehr weiter teilbar.

 $[^]c$ JSON parser - Tutorial — Lark documentation.

 $[^]a$ Z.B. wiederum ein **Programm**.

^bEs gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von Konkreter Syntax in Abstrakte Syntax übersetzt. Im Folgenden wird allerdings die Definition 2.37 verwendet.

 $[^]c JSON\ parser$ - Tutorial — $Lark\ documentation$.

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

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

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

Die vom Lexer im Eingabewort identifizierten Tokens werden in der Syntaktischen Analyse vom Parser als Wegweiser verwendet, da je nachdem, in welcher Reihenfolge die Tokens auftauchen, dies einer anderen Ableitung in der Grammatik G_{Parse} entspricht. Dabei wird in der Grammatik L_{Parse} nach dem Tokentypen 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 genauergesagt ein erweiterter Erkenner (Definition 2.38), denn ein Parser löst das Wortproblem (Definition 2.21) für die Sprache, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Erkennungsalgorithmus¹⁷ gesichert wurden den Ableitungsbaum.

Definition 2.38: Erkenner (bzw. engl. Recognizer)

Entspricht einem Kellerautomaten^a, in dem Wörter bestimmter Kontextfreier Sprachen erkannt werden. Der Erkenner ist ein Algorithmus, der erkennt, ob ein Eingabewort sich mit den Produktionen der Konkreten Grammatik einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der Konkreten Grammatik beschrieben wird oder nicht. Das vom Erkenner gelöste Problem ist auch als Wortproblem (Definition 2.21) bekannt.^b

Anmerkung Q

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 Ableitungsbaumes mit dem Startsymbol der Konkreten Grammatik an und wendet in jedem Schritt eine Linksableitung auf die Nicht-Terminalsymbole an, bis man Terminalsymbole hat, die sich zum gewünschten Eingabewort abgeleitet haben oder sich herausstellt, dass dieses nicht abgeleitet werden kann.

Der Grund, warum die Linksableitung verwendet wird und nicht z.B. die Rechtsableitung, ist,

 $[^]a\mathrm{Das}$ würde z.B. der Rolle eines Semikolon ; in der Sprache L_{PicoC} entsprechen.

^bZ.B. gibt es bestimmte Wechselbeziehungen zwischen Insekten, Insekten beinflussen sich gegenseitig und ihre Umwelt.

 $[^]a$ Automat mit dem Kontextfreie Grammatiken erkannt werden.

^bThiemann, "Compilerbau".

¹⁷Bzw. engl. recognition algorithm.

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 5.8).

Mit dieser Methode ist das Parsen Linksrekursiver Grammatiken (Definition 2.25) allerdings nicht möglich, ohne die Konkrete Grammatik vorher umgeformt zu haben und jegliche Linksrekursion aus der Konkreten Grammatik entfernt zu haben, da diese zu Unendlicher Rekursion führt.

Rekursiver Abstieg kann mit Backtracking verbunden werden, um auch Konkrete Grammatiken parsen zu können, die nicht LL(k) (Definition 5.9) sind. Dabei werden meist nach dem Prinzip der Tiefensuche alle Produktionen für ein Nicht-Terminalsymbol solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle Alternativen für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle Alternativen abgesucht sind, was dann bedeutet, dass das Eingabewort sich nicht mit der verwendeten Konkreten Grammatik ableiten lässt.^b

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

- Bottom-Up Parsing: Es wird mit dem Eingabewort gestartet und versucht Rechtsableitungen entsprechend der Produktionen einer Konkreten Grammatik 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.

Der Abstrakte 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).

a What 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 Webseite 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.

Definition 2.39: Transformer



Ein Programm, das von unten-nach-oben^a nach dem Prinzip der Breitensuche alle Knoten des Ableitungsbaum besucht und beim Antreffen eines bestimmten Knoten des Ableitungsbaumes je nach Kontext einen entsprechenden Knoten des Abstrakten Syntaxbaumes erzeugt und diesen anstelle des Knotens des Ableitungsbaumes setzt und so Stück für Stück den Abstrakten Syntaxbaum konstruiert.^b

 a In der Informatik wachsen Bäume von oben-nach-unten, von der Wurzel zur den Blättern.

Definition 2.40: Visitor



Ein Programm, das von unten-nach-oben^a, nach dem Prinzip der Breitensuche alle Knoten des Ableitungsbaumes besucht und beim Antreffen eines bestimmten Knoten des Ableitungsbaumes, diesen in-place mit anderen Knoten tauscht oder manipuliert, um den Ableitungbaum für die weitere Verarbeitung durch z.B. einen Transformer zu vereinfachen.^{bc}

^aIn der Informatik wachsen Bäume von oben-nach-unten, von der Wurzel zur den Blättern.

^bKann theoretisch auch zur Konstruktion eines Abstrakten Syntaxbaumes verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des Abstrakten Syntaxbaumes verantwortlich ist. Aber dafür ist ein Transformer besser geeignet.

 c Transformers $\, \& \, Visitors - Lark \, documentation.$

Definition 2.41: Abstrakte Syntax



Steht für alles, was mit dem Aufbau von Abstrakten Syntaxbäumen zu tuen hat.

Ein Abstrakter Syntaxbaum, der zur Kompilierung eines Wortes^a generiert wurde befindet sich in Abstrakter Syntax.^b

 a Z.B. Programmcode.

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

Um einen kurzen Begriff für die Grammatik, welche die Abstrakte Syntax einer Sprache beschreibt zu haben, wird diese im Folgenden als Abstrakte Grammatik (Definition 2.42) bezeichnet.

Definition 2.42: Abstrakte Grammatik



Grammatik, die eine Abstrakte Syntax beschreibt, also beschreibt was für Arten von Kompositionen mit den Knoten eines Abstrakten Syntaxbaumes möglich sind.

Jene Produktionen, die in der Konkreten Grammatik für die Umsetzung von Präzedenz notwendig waren, sind in der Abstrakten Grammatik abgeflacht. Dadurch sind die Kompositionen, welche die Knoten im Abstrakten Syntaxbaum bilden können syntaktisch meist näher an der Syntax von Maschinenbefehlen.

Definition 2.43: 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.

 $^{{}^}b\mathit{Transformers}\ \&\ \mathit{Visitors}\ -\mathit{Lark}\ \mathit{documentation}.$

Die Knoten des Abstrakten Syntaxbaumes enthalten dabei verschiedene Attribute, welche wichtigen Informationen für den Kompiliervorang und Fehlermeldungen enthalten.^a

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

Anmerkung Q

In dieser Bachelorarbeit wird häufig von der "Abstrakten Syntax", der "Abstrakten Grammatik" bzw. dem "Abstrakten Syntaxbaum" einer "Sprache" L gesprochen. Gemeint ist hier mit der Sprache L nicht die Sprache, welche durch die Abstrakte Grammatik, 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 Abstrakte Syntaxbaum überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die Abstrakte Grammatik beschrieben wird, interessiert man sich nie wirklich explizit. Diese Konvention wurde aus dem Buch G. Siek, Course Webpage for Compilers (P423, P523, E313, and E513) übernommen.

^aBzw. es ist die Sprache, welche durch die Konkrete Grammatik beschrieben wird.

Im Abstrakten Syntaxbaum können theoretisch auch die Tokens aus der Lexikalischen Analyse weiterverwendet werden, allerdings ist dies nicht empfehlenswert. Es ist zum empfehlen die Tokens durch eigene entsprechende Knoten umzusetzen, damit der Zugriff auf Knoten des Abstrakten Syntaxbaumes immer einheitlich erfolgen kann und auch, da manche Tokens des Abstrakten Syntaxbaum noch nicht optimal benannt sind. Manche "Symbole" werden in der Lexikalischen Analyse mehrfach verwendet, wie z.B. das Symbol - in L_{PicoC} , welches für die binäre Subtraktionsoperation als auch die unäre Minusoperation verwendet wurde. Der verwendete Tokentyp dieses Symbols lautet im PicoC-Compiler SUB_MINUS. Da in der Syntaktischen Analyse beide Operationen nur in bestimmten Kontexten vorkommen, lassen sie sich unterscheiden und dementsprechend können für beide Operationen jeweils zwei seperate Knoten erstellt werden. Im Fall des PicoC-Compilers sind es die Knoten Sub() und Minus().

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

Die Abstrakte Grammatik nach der ein Abstrakter Syntaxbaum konstruiert ist wird optimalerweise immer so definiert, dass der Abstrakte Syntaxbaum in den darauffolgenden Verarbeitungsschritten¹⁸ möglichst einfach weiterverarbeitet werden kann.

Auf der linken Seite in Abbildung 2.6 wird das Beispiel 2.26.2 aus Unterkapitel 2.2.1 fortgeführt. Dieses Beispiel stellt den Arithmetischen Ausdruck 4 * 2 in Bezug auf die Konkrete Grammatik 2.2^{19} , welche die höhere Präzedenz der Multipikation * berücksichtigt in einem Ableitungsbaum dar. Allerdings handelt es sich bei diesem Ableitungsbaum nicht um einen Formalen Ableitungsbaum, sondern um eine compilerinterne Datenstruktur für einen solchen. Dementsprechend sind die Blätter nun Tokens, die mithilfe der Grammatik L_{Lex} generiert wurden, womit die Darstellung von Ableitungen sich auf die Grammatik L_{Parse} beschränkt.

Auf der rechten Seite in Abbildung 2.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum abstrahiert, der nach der Abstrakten Grammatik 2.3 konstruiert ist. Die Abstrakte Grammatik ist hierbei in Abstrakter Syntaxform (Definition 3.5) angegeben. In der Abstrakten Grammatik 2.3 sind jegliche Produktionen wegabstrahiert, die in der Konkreten Grammatik 2.2 so umgesetzt sind, damit diese Präzidenz beachtet und nicht mehrdeutig ist. Aus diesem Grund gibt es nur noch einen allgemeinen

 $^{^{18}\}mathrm{Den}$ verschiedenen Passes.

¹⁹Die Konkrette Grammatik ist hierbei im Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit (Definition 3.4) angegeben.

Knoten für binäre Operationen $BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle)$.

DIG_NO_0	::=	"1" "2" "3" "4" "5" "6" "7" "8" "9"	L_Lex
DIG_WITH_0 NUM	::=	"0" DIG_NO_0	
ADD_OP MUL_OP	::=	"+"	
$egin{array}{c} mul \ add \end{array}$::=	$mul\ MUL_OP\ NUM\ \ NUM\ add\ ADD_OP\ mul\ \ mul$	L_Parse

Grammatik 2.2: Produktionen für Ableitungsbaum in EBNF

Grammatik 2.3: Produktionen für Abstrakten Syntaxbaum in ASF

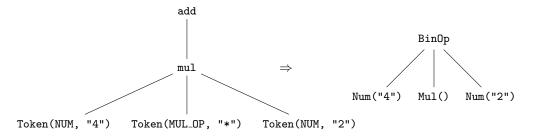


Abbildung 2.6: Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.

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, sind in Abbildung 2.7 die einzelnen Zwischenschritte von den Tokens der Lexikalischen Analyse zum Abstrakten Syntaxbaum anhand des fortgeführten Beispiels aus Subkapitel 2.3 veranschaulicht. In Abbildung 2.7 werden die Darstellungen des Ableitungsbaumes und des Abstrakten Syntaxbaumes verwendet, wie sie vom PicoC-Compiler ausgegeben werden. In der Darstellung des PicoC-Compilers stellen die verschiedenen Einrückungen die verschiedenen Ebenen dieser Bäume dar. Die Bäume wachsen von der Wurzel von links-nach-rechts zu den Blättern.

Abstrakter Syntaxbaum File Name './example1.ast', FunDef VoidType 'void', Tokenfolge Name 'main', [], [Token('FILENAME', './example1.picoc'), Token('VOID_DT', Ε → 'void'), Token('NAME', 'main'), Token('LPAR', '('), Ιf → Token('RPAR', ')'), Token('LBRACE', '{'), Token('IF', Num '42', $_{\hookrightarrow}$ 'if'), Token('LPAR', '('), Token('NUM', '42'), → Token('RPAR', ')'), Token('LBRACE', '{'),] → Token('RBRACE', '}'), Token('RBRACE', '}')]] Parser Visitors und Transformer Ableitungsbaum file ./example1.dt decls_defs decl_def fun_def type_spec prim_dt void pntr_deg name main fun_params decl_exec_stmts exec_part exec_direct_stmt if_stmt logic_or logic_and eq_exp rel_exp arith_or arith_oplus arith_and arith_prec2 arith_prec1 un_exp post_exp 42 prim_exp exec_part compound_stmt

Abbildung 2.7: Veranschaulichung der Syntaktischen Analyse.

2.5 Code Generierung

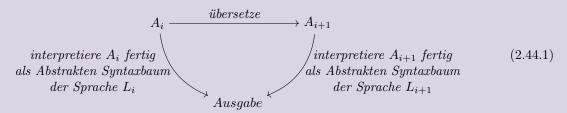
In der Code Generierung steht man nun dem Problem gegenüber einen Abstrakten Syntaxbaum einer Sprache L_1 in den Abstrakten Syntaxbaum einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man Passes (Definition 2.44) nennt. So wie es auch schon mit dem Ableitungsbaum in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum Abstrakten Syntaxbaum kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit Transformern und Visitors ein Abstrakter Syntaxbaum generiert werden.

Definition 2.44: Pass

/

Einzelner Übersetzungsschritt in einem Kompiliervorgang 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.

Für jeden Pass und für einen beliebigen Abstrakten Syntaxbaum A_i gilt ähnlich, wie bei einem vollständigen Compiler in 2.44.1, 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

Die von den Passes umgeformten Abstrakten Syntaxbäume 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 tuen, welche Unreine Ausdrücke (Definition 2.46) besitzt, so ist es sinnvoll einen Pass einzuführen, der Reine (Definition 2.45) und Unreine Ausdrücke voneinander trennt. Das wird erreicht, indem man aus den Unreinen Ausdrücken vorangestellte Anweisungen 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.

^aEin Pass kann mit einem Transpiler 5.7 (Definition 5.7) 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 auschließlich 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.

^cInterpretieren 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).

Definition 2.45: Reiner Ausdruck (bzw. engl. pure expression)

Z

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. ^{ab}

 a Sondern z.B. intern etwas am Kompilierprozess ändert.

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

Definition 2.46: Unreiner Ausdruck

Z

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

Auf diese Weise sind alle Anweisungen und Ausdrücke in Monadischer Normalform (Definiton 2.47).

Definition 2.47: Monadische Normalform (bzw. engl. monadic normal form)

Z

Eine Anweisung oder Ausdruck ist in Monadischer Normalform, wenn es oder er nach einer Konkreten Grammatik in Monadischer Normalform abgeleitet wurde.

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

Eine Abstrakte Grammatik ist in Monadischer Normalform, wenn die Konkrete Grammatik 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 Syntax²⁰ 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 eigenständige Anweisung vorangestellt werden.



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

Die Aufgabe eines solchen Passes ist es, den Abstrakten Syntaxbaum der Syntax von Maschinenbefehlen anzunähren, 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

²⁰Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

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 2.50) aus Anweisungen und Ausdrücken entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken vorangestellte Anweisungen macht, in denen die Komplexen Ausdrücke temporären Locations zugewiesen werden (Definiton 2.48) und dann anstelle des Komplexen Ausdrucks auf die jeweilige temporäre Location zugegriffen wird.

Sollte in der Anweisung, in welcher 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 Anweisungen zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur Atomare Ausdrücke (Definiton 2.49) 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 eine vorangestellte Anweisung gemacht, welches einfach nur den Nebeneffekt dieses Unreinen Ausdrucks ausführt.

Definition 2.48: Location

Z

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

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

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

Auf diese Weise sind alle Anweisungen und Ausdrücke in A-Normalform (Definition 2.51). Wenn eine Konkrete Grammatik in A-Normalform ist, ist diese auch automatisch in Monadischer Normalform (Definition 2.51), genauso, wie ein Atomarer Ausdruck auch ein Reiner Ausdruck ist (nach Definition 2.49).

Definition 2.49: 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).

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

Definition 2.50: Komplexer Ausdruck



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

^aint var ist eine Allokation.

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

Definition 2.51: A-Normalform (ANF)

Z

Eine Anweisung oder ein Ausdruck ist in A-Normalform, wenn es oder er nach einer Konkreten Grammatik in A-Normalform abgeleitet wurde.

Eine Konkrete Grammatik 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 Grammatik ist in A-Normalform, wenn die Konkrete Grammatik 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 Syntax²¹ aufgeschrieben wurden.

Der PicoC-Compiler nutzt, anders als es geläufig ist keine Register und Graph Coloring (Definition 5.13) inklusive Liveness Analysis (Definition 5.11) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den Hauptspeicher, wobei temporäre Zwischenergebnisse auf den Stack gespeichert werden.²²

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 Abstrakte 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 Ausdrück 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 Maschinenbefehlssatz es erlaubt zwei Register miteinander zu verechnen²³, ist es möglich Ausdrücke für Zahl 4, Variable var und ASCII-Zeichen c' als atomar zu definieren, da sie mit einem Maschinenbefehl verarbeitet werden können²⁴. 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 verechnet werden können.

Die Anweisungen 4, x, usw. für sich sind in diesem Fall Anweisungen, 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

²¹Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

²²Die 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.

 $^{^{23}{\}rm Z.B.}$ Addieren oder Subtraktion von zwei Registerinhalten.

²⁴Mit dem RETI-Befehlssatz wäre das durchaus möglich, durch z.B. MULT ACC IN2.

zu verrechnen und wiederum einer Speicherzelle des Stack zuzuweisen.

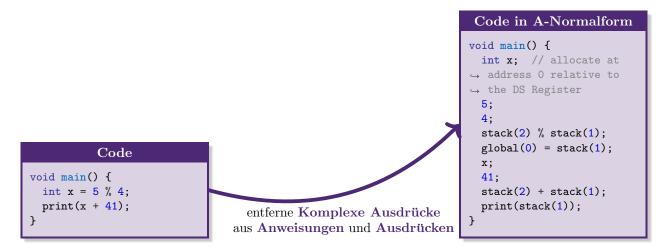


Abbildung 2.9: Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.

Ein solcher Pass hat vor allem in erster Linie die Aufgabe den Abstrakten Syntaxbaum der Syntax von Maschinenbefehlen besonders dadurch anzunähren, dass er die Anweisungen weniger komplex macht und diese dadurch den ziemlich simplen Maschinenbefehlen syntaktisch ähnlicher sind. Des Weiteren vereinfacht dieser Pass die Implementierung der nachfolgenden Passes enorm, da Anweisungen 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 Anweisungen 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 Abstrakten Syntaxbaum den eigentlichen Maschinencode in Konkreter Syntax zu generieren. In üblichen Compilern wird hier für den Maschinencode eine binäre Repräsentation gewählt. Da der PicoC-Compiler vor allem zu Lernzwecken konzipiert ist, wird bei diesem der Maschinencode allerdings in einer menschenlesbaren Repräsentation ausgegeben. Der Weg von der Abstrakten Syntax zur Konkreten Syntax ist allerdings wesentlich einfacher, als der Weg von der Konkreten Syntax zur Abstrakten Syntax, für die eine gesamte Syntaktische Analyse, die eine Lexikalische Analyse beinhaltet durchlaufen werden musste.

Jeder Knoten des Abstrakten Syntaxbaumes 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 Prinzip der Tiefensuche der gesamte Abstrakte Syntaxbaum 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 zurückgebeben.

2.6 Fehlermeldungen

Wenn bei einem Compiler ein unerwünschtes Verhalten der folgenden Kategorien²⁵ eintritt:

- 1. in der Lexikalischen oder Syntaktischen Analyse tritt eine Fall ein, der nicht in der Syntax der Sprache des Compilers abgedeckt ist, z.B.:
 - der Lexer kann eine Zeichenfolge nicht nach der Grammatik G_{Lex} ableiten. Der Lexer ist genaugenommen ein Teil des Parsers und ist damit bereits durch den nachfolgenden Punkt "Parser" abgedeckt. Um die unterschiedlichen Ebenen, Lexikalische und Syntaktische Analyse gesondert zu betrachten wurde der Lexer an dieser Stelle ebenfalls kurz eingebracht.
 - der Parser²⁶ entscheidet das Wortproblem für ein Eingabeprogramm²⁷ mit 0, also das Eingabeprogramm lässt sich nicht durch die Konkrete Grammatik $G_{Lex} \uplus G_{Parse}$ des Compilers ableiten.
- 2. in den Passes tritt ein Fall ein, der nicht in der Syntax 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 5.6) etwas nicht zusammenpasst, wie z.B.:

- es gibt keine oder mehr als eine main-Funktion.
- eine Funktion, die in einer Objektdatei (Definition 5.5) benötigt wird, wird von keiner anderen oder mehr als einer Objektdatei bereitsgestellt.

wird eine Fehlermeldung (Definition 2.52) ausgegeben.

Definition 2.52: 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.
- ullet über eine Dialogbox in einer Graphischen Benutzerfläche^c oder Zeichenorientierten Benutzerschnittstelle^d.

 $^{^{25}}Errors\ in\ C/C++$ - Geeks for Geeks.

 $^{^{26}\}mathrm{Bzw.}$ der **Erkenner** innerhalb des Parsers.

²⁷Bzw. Wort.

- in ein Register oder an eine spezielle Adresse des Hauptspeichers wird ein Wert geschrie-
- Logdatei^e auf einem Speichermedium.

 $[^]a$ Dieses Programm kann z.B. ein Compiler sein oder ein Programm, dass dieser Compiler selbst kompiliert hat.

 $^{{}^}b$ Nur unter Linux, Windows hat sowas nicht.

^cIn engl. Graphical User Interface, kurz GUI.

 $[^]d\mathrm{In}$ engl. Text-based User Interface, kurz TUI. $^e\mathrm{In}$ 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 einer in einem eigenen Dialekt der Erweiterten Back-Naur-Form (Definition 3.3 bzw. für den Dialekt von Lark Definition 3.4) spezifizierten Konkreten Grammatik ein Programm in Konkreter Syntax zu parsen und daraus einen Ableitungsbaum für die kompilerintere Weiterverarbeitung zu generieren.

Definition 3.1: Metasyntax

Z

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

Definition 3.2: Metasprache

Z

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 Kontextfreie Grammatiken darzustellen.

Am grundlegensten lässt sich die Erweiterte Backus-Naur-Form in Kürze wie folgt beschreiben. bc

- Terminalsymbole werden in Anführungszeichen "" geschrieben (z.B. "term").
- Nicht-Terminalsymbole werden normal hingeschrieben (z.B. non-term).
- Leerzeichen dienen zur visuellen Abtrennung von Grammatiksymbolen^d.

Weitere Details sind in der Spezifikation des Standards unter Link^e zu finden. Allerdings werden in der Praxis, wie z.B. in Lark oft eigene abgewandelte Notationen wie in Definition 3.4 verwendet.

¹Wobei beim Parsen auch das Lexen inbegriffen ist.

 $^{^2\}mathit{Lark}$ - a parsing toolkit for Python.

³Shinan, lark.

Definition 3.4: Dialekt der Erweiterten Backus-Naur-Form aus Lark

1

Das Lark Parsing Toolkit verwendet eine eigene Notation für die Erweiterte Backus-Naur-Form (Definition 3.3), die sich teilweise in einzelnen Aspekten von der Syntax aus dem Standard unterscheidet und unter Link^a dokumentiert ist.

Wichtige Unterschiede dieses Dialekts sind hierbei z.B.:

• für die Darstellung von Optionaler Wiederholung wird der aus regulären Ausdrücken bekannte *-Quantor zusammen mit optionalen runden Klammern () verwendet (z.B. ()*).^b Die Verwendung des *-Symbols kann wie in Umformung 3.4.1 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b*\} \quad \Rightarrow \quad \{a := b_tmp, \ b_tmp := b \ b_tmp \ \mid \ \varepsilon\} \tag{3.4.1}$$

• für die Darstellung von mindestents 1-Mal Wiederholung wird der ebenfalls aus regulären Ausdrücken bekannte +-Operator zusammen mit optionalen runden Klammern () verwendet (z.B. ()+). Die Verwendung des +-Symbols kann wie in Umformung 3.4.2 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b+\} \quad \Rightarrow \quad \{a := b \ b_tmp, \ b_tmp := b \ b_tmp \mid \varepsilon\} \tag{3.4.2}$$

• für alle ASCII-Symbole zwischen z.B. _ und ~ als Alternative aufgeschrieben kann auch die Abkürzung "_"..."~" verwendet werden. Die Verwendung dieser Schreibweise kann wie in Umformung 3.4.3 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a ::= "ascii1" ... "ascii2"\} \Rightarrow \{a ::= "ascii1" \mid ... \mid "ascii2"\}$$
 (3.4.3)

Um bei einer Produktion auszudrücken, wozu die linke Seite abgeleitet werden kann, wird das ::=-Symbol verwendet. Dieses Symbol wird als "kann abgeleitet werden zu" gelesen.

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 Grammatik einer Sprache einen Parser in einer vorher bestimmten Programmiersprache generieren, anstatt wie das Lark Parsing Toolkit bei Angabe einer Konkreten Grammatik direkt ein Programm in dieser Konkreten Grammatik 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}

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

^bNebel, "Theoretische Informatik".

 $[^]c$ Grammar Reference — Lark documentation.

 $[^]d$ Also von Terminalsymbolen und Nicht-Terminalsymbolen.

 $[^]e$ https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_IS0_IEC_14977_1996(E).zip.

^ahttps://lark-parser.readthedocs.io/en/latest/grammar.html.

 $[^]b\mathrm{Der}$ *-Quantor bedeutet im Gegensatz zum +-Operator auch keinmal wiederholen.

 $^{^{4}}ANTLR$.

zur Maschinensprache L_{RETI} funktioniert. Eine ausführliche Diskussion zur Priorisierung Laufziet wurde in Unterkapitel 1.4 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 Grammatiken, die aufgrund der Verwendung des Lark Parsing Toolkit in einem eigenen Dialekt der Erweiterten Back-Naur-Form spezifiziert sind, werden in den folgenden Unterkapiteln die Abstrakten Grammatiken, welche spezifizieren, welche Kompositionen für die Abstrakten Syntaxbäume der verschiedenden 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 Abstrakten Grammatiken nur unnötig verkomplizieren würde, wenn man die Erweiterte Backus-Naur-Form verwenden würde. Es gibt leider keine Standardnotation für Abstrakte Grammatiken, die sich deutlich durchgesetzt hat, daher wird für Abstrakte Grammatiken eine eigene Abstrakte Syntaxform 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 Syntaxform (ASF)

Z

Ist eine eigene Metasyntax für Abstrakte Grammatiken, die für diese Bachelorarbeit definiert wurde. Sie unterscheidet sich vom Dialekt der Backus-Naur-Form des Lark Parsing Toolkit (Definition 3.4) nur durch:

- Terminalsymbole müssen nicht von "" engeschlossen sein, da die Knoten in der Abstrakten Syntax sowieso schon klar auszumachen sind und von anderen Symbolen der Metasprache leicht zu unterscheiden sind (z.B. Node(<non-term>), <non-term>)).
- dafür müssen allerdings Nicht-Terminalsymbole von <>-Klammern eingeschlossen sein (z.B. <non-term>).

Letztendlich geht es allerdings nur darum, dass aufgrund der Verwendung des Lark Parsing Toolkit die Konkrete Grammatik in einem eigenen Dialekt der Erweiterter Backus-Naur-Form angegeben sein muss und für das Implementieren der Passes die Abstrakte Grammatik für den Programmierer möglichst einfach verständlich sein sollte, weshalb sich die Abstrake Syntax Form gut dafür eignet.

3.1 Lexikalische Analyse

Für die Lexikalische Analyse ist es nur notwendig eine Konkrete Grammatik zu definieren, die den Teil der Konkreten Syntax beschreibt, der für die Lexikalische Analyse wichtig ist. Diese Konkrete 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 Grammatik für die Lexikalische Analyse

In der Konkreten Grammatik 3.1.1 für die Lexikalische Analyse stehen großgeschriebene Nicht-Terminalsymbole entweder für einen Tokentyp oder einen Teil der Beschreibung eines Tokentyps. Zum Beispiel handelt es sich bei dem großgeschriebenen Nicht-Terminalsymbol NUM um einen Tokentyp, der durch die Produktion NUM ::= "0" | DIG_NO_O DIG_WITH_O* 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 ::=-Symbols ebenfalls ein großgeschriebenen Nicht-Terminalsymbol hat. Dagegen dient das großgeschriebene Nicht-Terminalsymbol DIG_NO_O aus der Produktion NUM ::= "O" | DIG_NO_O DIG_WITH_O* nur zu Beschreibung von NUM.

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

Die in der Konkrete Grammatik vereinzelt kleingeschriebenen Nicht-Terminalsymbole, wie name haben nur den Zweck mehrere Tokentypen, 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 ::=-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 setzen⁵, damit das Wort int den Tokentyp 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 Tokentyp zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. In diesem Fall würden aus int_var die Tokens 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 Link⁶ 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 Konkreten Grammatik zu lexen, zu komplex, um sie an dieser Stelle allgemein 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 Priortiäten, 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, wird hieraus direkt ein Token mit dem entsprechenden Tokenwert dieser Produktion erstellt. Weitere Produktionen werden nicht mehr geprüft. Ansonsten werden alle Produktionen der nächstniedrigeren Priorität geprüft usw.

⁵Es wird immer die höchste Priorität zuerst genommen.

 $^{^6}$ https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.

```
/[\wedge \backslash n]*/
COMMENT
                                                  /(. | \n)*? / "*/"
                                                                           L_{-}Comment
                       ::=
                            "//""_{-}"?"#"/[\wedge \setminus n]*/
RETI\_COMMENT.2
                       ::=
                                           "3"
                                   "2"
DIG\_NO\_0
                       ::=
                            "1"
                                                                           L_Arith_Bit
                            "7"
                                    "8"
                                           "9"
DIG\_WITH\_0
                            "0"
                                    DIG\_NO\_0
                       ::=
NUM
                            "0"
                                   DIG\_NO\_0 DIG\_WITH\_0*
                       ::=
                            "_"…"∼"
CHAR
                       ::=
FILENAME
                            CHAR + ".picoc"
                       ::=
LETTER
                            "a"..."z"
                                     | "A".."Z"
                       ::=
                            (LETTER | "_")
NAME
                       ::=
                                (LETTER | DIG_WITH_0 | "_")*
                            NAME | INT_NAME | CHAR_NAME
name
                       ::=
                            VOID\_NAME
                            "!"
LOGIC_NOT
                       ::=
                            " \sim "
NOT
                       ::=
                            "&"
REF\_AND
                       ::=
                            SUB_MINUS | LOGIC_NOT |
                                                               NOT
un\_op
                       ::=
                            MUL\_DEREF\_PNTR \mid REF\_AND
MUL\_DEREF\_PNTR
                            "*"
                       ::=
                            " /"
DIV
                       ::=
                            "%"
MOD
                       ::=
                            MUL\_DEREF\_PNTR \mid DIV \mid MOD
prec1\_op
                       ::=
                            "+"
ADD
                       ::=
SUB\_MINUS
                       ::=
                            ADD
                                     SUB\_MINUS
prec2\_op
                       ::=
                            "<<"
L\_SHIFT
                       ::=
                            ">>"
R\_SHIFT
                       ::=
shift\_op
                            L\_SHIFT
                                          R\_SHIFT
                       ::=
LT
                            "<"
                                                                           L\_Logic
                       ::=
                            "<="
LTE
                       ::=
                            ">"
GT
                       ::=
                            ">="
GTE
                       ::=
rel\_op
                            LT
                                   LTE
                                            GT
                       ::=
EQ
                            "=="
                       ::=
                            "!="
NEQ
                       ::=
                                    NEQ
                            EQ
eq\_op
                       ::=
                            "int"
INT\_DT.2
                       ::=
                                                                           L_{-}Assign_{-}Alloc
INT\_NAME.3
                            "int"
                                  (LETTER \mid DIG\_WITH\_0 \mid
                       ::=
                            "char"
CHAR\_DT.2
                       ::=
CHAR\_NAME.3
                            "char" (LETTER
                                                 DIG\_WITH\_0
                       ::=
VOID\_DT.2
                       ::=
                            "void"
VOID\_NAME.3
                            "void" (LETTER
                                                 DIG\_WITH\_0
                       ::=
prim_{-}dt
                            INT\_DT
                                         CHAR\_DT
                                                        VOID\_DT
                       ::=
```

Grammatik 3.1.1: Konkrete Grammatik 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 Abstrakten Syntaxbaumes 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 Tokens sind Code 3.2 zu sehen.

```
Image: Ito the image: Ito the
```

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 Abstrakten Syntaxbaum in Abstrakter Syntax zu generieren.

3.2.1 Umsetzung von Präzedenz und Assoziativität

In diesem Unterkapitel wird eine ähnliche Erklärweise, wie in Quelle Parsing Expressions · Crafting Interpreters verwendet. Die Programmiersprache L_{PicoC} hat dieselben Präzedenzregeln implementiert, wie die Programmiersprache L_{C} . Die Präzedenzregeln der verschiedenen Operatoren der Programmiersprache L_{PicoC} sind in Tabelle 3.1 aufgelistet.

⁷C Operator Precedence - cppreference.com.

Präzedenz	zstuf@peratoren	Beschreibung	Assoziativität	
1	a()	Funktionsaufruf		
	a[]	Indexzugriff	Links, dann rechts \rightarrow	
	a.b	Attributzugriff		
2	-a	Unäres Minus		
	!a ~a	Logisches NOT und Bitweise NOT	Pachta dann linka	
	*a &a	Dereferenz und Referenz, auch	Rechts, dann links \leftarrow	
		Adresse-von		
3	a*b a/b a%b	Multiplikation, Division und Modulo		
4	a+b a-b	Addition und Subtraktion		
5	a< <b a="">>b	S a b Bitweise Linksshift und Rechtsshift		
6	a <b a<="b</td"><td colspan="2">a a b a<=b Kleiner, Kleiner Gleich, Größer, Größer</td>	a a b a<=b Kleiner, Kleiner Gleich, Größer, Größer		
	a>b a>=b	Gleich		
7	a==b a!=b	Gleichheit und Ungleichheit Links, dann rechts		
8	a&b	a&b Bitweise UND		
9	a^b	a^b Bitweise XOR (exclusive or)		
10	a b	a b Bitweise ODER (inclusive or)		
11	a&&b	Logiches UND		
12	a b	Logisches ODER		
13	a=b	Zuweisung	Rechts, dann links \leftarrow	

Tabelle 3.1: Präzedenzregeln von PicoC.

Würde man diese Operatoren ohne Beachtung von Präzedenzreglen (Definition 2.29) und Assoziativität (Definition 2.28) in eine Konkrete Grammatik verarbeiten wollen, so könnte eine Konkrete Grammatik $G = \langle N, \Sigma, P, exp \rangle$ mit Produktionen P 3.2.1 dabei rauskommen.

```
"'"CHAR"'"
                                                        "("exp")"
                           NUM
                                                                                   L_-Arith_-Bit
prim_{-}exp
                 exp"["exp"]"
                                               | name"("fun\_args")"
                                  exp"."name
                                                                                   +L_{-}Logic
                 [exp("," exp)*]
fun\_args
                                                                                   + L_-Pntr
           ::=
                                                                                   + L_Array
un\_op
                                                                                   + L_Struct
un\_exp
           ::=
                 un\_op \ exp
                                       | "+" | "-"
bin\_op
                                              "<="
                                  "&&"
bin_{-}exp
           ::=
                 exp bin_op exp
                              un\_exp \mid bin\_exp
exp
                 prim_{-}exp
```

Grammatik 3.2.1: Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet

Die Konkrete Grammatik 3.2.1 ist allerdings mehrdeutig, d.h. verschiedene Linksableitungen in der Konkreten 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.

$$exp \Rightarrow bin_exp \Rightarrow exp \ bin_op \ exp \Rightarrow bin_exp \ bin_op \ exp$$

 $\Rightarrow exp \ bin_op \ exp \ bin_op \ exp \ \Rightarrow^* \ "3" "*" "1" "&&" "4"$

```
exp \Rightarrow bin\_exp \Rightarrow exp \ bin\_op \ exp \Rightarrow prim\_exp \ bin\_op \ exp
\Rightarrow NUM \ bin\_op \ exp \Rightarrow "3" \ bin\_op \ exp \Rightarrow "3" "*" \ exp
\Rightarrow "3" "*" \ bin\_exp \Rightarrow "3" "*" \ exp \ bin\_exp \ exp \Rightarrow "3" "*" "1" "&&" "4"
```

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 darsgestellt sind in Klammern setzen, um die Präzedenz 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 Präzedenzregeln und die Assoziativität der Operatoren beim Erstellen der Konkreten Grammatik miteinzubeziehen. Hierzu wird nun Tabelle 3.1 betrachtet. Für jede Präzedenzstufe 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 Präzedenzstufe, welche Literale, wie 'c', 5 oder var und geklammerte Ausdrücke wie (3 && 14) abdeckt.

$prim_exp$::=	 $L_Arith_Bit + L_Array$
$post_exp$::=	 $+$ $L_{-}Pntr$ $+$ $L_{-}Struct$
un_exp	::=	 $+ L_{-}Fun$
$arith_prec1$::=	
$arith_prec2$::=	
$arith_shift$::=	
$arith_and$::=	
$arith_xor$::=	
$arith_or$::=	
rel_exp	::=	 L_Logic
eq_exp	::=	
$logic_and$::=	
$logic_or$::=	
$\overline{assign_stmt}$::=	 L_Assign

Grammatik 3.2.2: Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet

Einige Bezeichnungen von Nicht-Terminalsymbolen auf der linken Seite des ::=-Operators 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
$arith_shift$	a< <b a="">>b
arith_and	a&b
arith_xor	a^b
arith_or	a b
rel_exp	a <b a="" a<="b">b a>=b
eq_exp	a==b 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 **Präzedenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzedenzstufe **höher** ist. Z.B. soll un_op sowohl den Ausdruck -(3 * 14) als auch einfach nur (3 * 14)⁸ erkennen können, aber nicht 3 * 14 ohne Klammern, da dieser Ausdruck eine **geringe Präzedenz** 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 um_exp in 3.2.3 für die rechtsassoziativen unären Operatoren -a, !a ~a, *a und &a ist die Alternative um_op um_exp dafür zuständig, dass diese unären Operatoren rechtsassoziativ geschachtelt werden können (z.B. !-~42). Die Alternative post_exp ist dafür zuständig, dass die Produktion auch terminieren kann und es auch möglich ist auschließlich einen Ausdruck höherer Präzedenz (z.B. 42) zu haben.

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

Grammatik 3.2.3: Beispiel für eine unäre rechtsassoziative Produktion in EBNF

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 auschließlich einen Ausdruck der höchsten Präzedenz (z.B. 42) zu haben.

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

Grammatik 3.2.4: Beispiel für eine unäre linksassoziative Produktion in EBNF

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 Präzedenzstufe 4 in Folge erkannt werden können⁹ (z.B. 3 + 1 - 4, wobei - und + beide Präzedenzstufe 4

⁸Geklammerte Ausdrücke werden nämlich von prim_exp erkannt, welches eine höhere Präzedenzstufe hat.

⁹Bezogen auf Tabelle 3.1.

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

 $arith_prec2$::= $arith_prec2$ $prec2_op$ $arith_prec1$ | $arith_prec1$

Grammatik 3.2.5: Beispiel für eine binäre linksassoziative Produktion in EBNF

Anmerkung Q

Manche Parser^a haben allerdings ein Problem mit Linksrekursion (Definition 2.25), 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.

 $arith_prec2$::= $arith_prec1$ ($prec2_op$ $arith_prec1$)*

Grammatik 3.2.6: Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF

Die von der Grammatik 3.2.6 erkannten Ausdrücke sind dieselben, wie für die Grammatik 3.2.5, allerdings ist die Grammatik 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 Präzedenzstufe 4 in Folge erkennen zu können (z.B. 3 + 1 - 4, wobei - und + beide Präzedenzstufe 4 haben).

Das Nicht-Terminalsymbol arith_prec1 erlaubt es, dass zwischen der Folge von Operationen der Präzedenzstufe 4 auch Operationen der Präzedenzstufe 3 auftauchen können (z.B. 3 + 1 / 4 - 1, wobei - und + beide Präzedenzstufe 4 haben und / Präzedenzstufe 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 auschließlich ein Ausdruck höherer Präzedenz 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 Konkrete Grammatik 3.2.7.

```
"*"
                                                                                                  L_{-}Misc
prec1\_op
               ::=
                     "+"
prec2\_op
               ::=
                     "<<"
shift\_op
rel\_op
                ::=
eq\_op
                     [logic\_or("," logic\_or)*]
fun\_args
                ::=
                                                          "("logic\_or")'
                                NUM
                                            CHAR
prim_{-}exp
                                                                                                  L_Arith_Bit
               ::=
                     post\_exp"["logic\_or"]"
                                              | post_exp"."name | name"("fun_args")"
                                                                                                  + L_Array
post\_exp
               ::=
                     prim_{-}exp
                                                                                                  + L_-Pntr
                                                                                                  + L_Struct
un_{-}exp
                     un\_op \ un\_exp \mid post\_exp
               ::=
                     arith_prec1 prec1_op un_exp
                                                                                                  + L_Fun
arith\_prec1
                ::=
                                                      un_{-}exp
arith\_prec2
                ::=
                     arith_prec2 prec2_op arith_prec1 | arith_prec1
arith\_shift
                     arith\_shift\ shift\_op\ arith\_prec2 | arith\_prec2
                ::=
arith\_and
               ::=
                     arith_and "&" arith_shift | arith_shift
arith\_xor
                     arith\_xor "\land" arith\_and
                                                  | arith\_and
               ::=
                     arith_or "|" arith_xor
arith\_or
                                                  arith\_xor
               ::=
rel\_exp
                     rel_exp rel_op arith_or
                                                   arith\_or
                                                                                                  L_{-}Logic
               ::=
                     eq_exp eq_op rel_exp |
                                                rel_exp
eq_exp
               ::=
                     logic\_and "&&" eq\_exp
                                                | eq_{-}exp
logic\_and
               ::=
                                                  logic\_and
                     logic_or "||" logic_and
logic\_or
               ::=
                     un_exp "=" logic_or";"
assign\_stmt
               ::=
                                                                                                  L_Assign
```

Grammatik 3.2.7: Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet

3.2.2 Konkrete Grammatik für die Syntaktische Analyse

Die gesamte Konkrete Grammatik 3.2.8 ergibt sich wenn man die Konkrete 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 Grammatik an die aktuellste kostenlos auffindbare Version der echten Konkreten Grammatik der Sprache L_C , zusammengesetzt aus einer Grammatik für die Syntaktische Analyse $ANSI\ C\ grammar\ (Yacc)$ und Lexikalische Analyse $ANSI\ C\ grammar\ (Lex)$ angepasst¹⁰, 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 GCC¹¹ und Clang¹² zu nennen wären.

In der Konkreten Grammatik 3.2.8 für die Syntaktischen Analyse werden einige der Tokentypen aus der Konkreten Grammatik 3.1.1 für die Lexikalischen Analyse verwendet, wie z.B. NUM aber auch name, welches eine Produktion ist, die mehrere Tokentypen unter einem Überbegriff zusammenfasst.

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

 $^{^{10}}$ An der für die Programmiersprache L_{PicoC} relevanten Syntax hat sich allerdings über die Jahre nichts verändert, wie die Konkreten Grammatiken für die Syntaktische Analyse ANSI C grammar (Yacc) old und Lexikalische Analyse ANSI C grammar (Lex) old aus dem Jahre 1985 zeigen.

 $^{^{11}}GCC$, the $\stackrel{\frown}{GNU}$ Compiler Collection - $\stackrel{\frown}{GNU}$ Project.

 $^{^{12}}$ clang: C++ Compiler.

prim_exp post_exp un_exp	::= ::= ::=	name NUM CHAR "("logic_or")" array_subscr struct_attr fun_call input_exp print_exp prim_exp un_op un_exp post_exp	$L_Arith_Bit + L_Array + L_Pntr + L_Struct + L_Fun$
input_exp print_exp arith_prec1 arith_prec2 arith_shift arith_and arith_xor arith_or	::= ::= ::= ::= ::= ::=	"input""("")" "print""("logic_or")" arith_prec1 prec1_op un_exp un_exp arith_prec2 prec2_op arith_prec1 arith_prec1 arith_shift shift_op arith_prec2 arith_prec2 arith_and "&" arith_shift arith_shift arith_xor "\" arith_and arith_and arith_or " " arith_xor arith_xor	L_Arith_Bit
rel_exp eq_exp logic_and logic_or	::= ::= ::=	rel_exp rel_op arith_or arith_or eq_exp eq_op rel_exp rel_exp logic_and "&&" eq_exp eq_exp logic_or " " logic_and logic_and	L_Logic
type_spec alloc assign_stmt initializer init_stmt const_init_stmt	::= ::= ::= ::= ::=	<pre>prim_dt struct_spec type_spec pntr_decl un_exp "=" logic_or";" logic_or array_init struct_init alloc "=" initializer";" "const" type_spec name "=" NUM";"</pre>	L_Assign_Alloc
$pntr_deg$ $pntr_decl$::=	"*"* pntr_deg array_decl array_decl	$L_{-}Pntr$
array_dims array_decl array_init array_subscr	::= ::= ::=	("["NUM"]")* name array_dims "("pntr_decl")"array_dims "{"initializer("," initializer) *"}" post_exp"["logic_or"]"	L_Array
struct_spec struct_params struct_decl struct_init struct_attr	::= ::= ::= ::=	"struct" name (alloc";")+ "struct" name "{"struct_params"}" "{""."name"="initializer ("," "."name"="initializer)*"}" post_exp"."name	L_Struct
if_stmt if_else_stmt	::=	"if""("logic_or")" exec_part "if""("logic_or")" exec_part "else" exec_part	L_If_Else
while_stmt do_while_stmt	::=	"while""("logic_or")" exec_part "do" exec_part "while""("logic_or")"";"	$L_{-}Loop$

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

```
alloc";"
                                                                                                    L\_Stmt
decl\_exp\_stmt
                    ::=
decl\_direct\_stmt
                          assign_stmt | init_stmt | const_init_stmt
                    ::=
decl\_part
                          decl\_exp\_stmt \mid decl\_direct\_stmt \mid RETI\_COMMENT
                    ::=
                          "{"exec\_part *"}"
compound\_stmt
                    ::=
                          logic\_or";"
exec\_exp\_stmt
                    ::=
exec\_direct\_stmt
                          if\_stmt \mid if\_else\_stmt \mid while\_stmt \mid do\_while\_stmt
                    ::=
                          assign\_stmt \quad | \quad fun\_return\_stmt
                          compound\_stmt \mid exec\_exp\_stmt \mid exec\_direct\_stmt
exec\_part
                    ::=
                          RETI\_COMMENT
                          decl\_part * exec\_part *
decl\_exec\_stmts
                    ::=
                                                                                                    L_{-}Fun
fun\_args
                          [logic\_or("," logic\_or)*]
                    ::=
                          name"("fun\_args")"
fun\_call
                    ::=
fun\_return\_stmt
                          "return" [logic_or]";"
                    ::=
                          [alloc("," alloc)*]
fun\_params
                    ::=
fun\_decl
                          type_spec pntr_deg name" ("fun_params")"
                    ::=
                          type_spec_pntr_deg_name"("fun_params")" "{"decl_exec_stmts"}"
fun_{-}def
                    ::=
                          (struct\_decl
                                            fun\_decl)";"
decl\_def
                                                              fun_{-}def
                                                                                                    L_File
                    ::=
                          decl\_def*
decls\_defs
                    ::=
file
                    ::=
                          FILENAME decls_defs
```

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

Anmerkung Q

In der Konkreten 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 Grammatik 3.2.8 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 Erkenner (Defintion 5.10 im Kapitel Appendix) nutzt, um einen Ableitungsbaum zu konstruieren.

Zur Konstruktion des Ableitungsbaumes muss dafür gesorgt werden, dass der Earley Erkenner 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

 $[^]a$ Jay Earley, "An efficient context-free parsing".

3.2.3.1 Codebeispiel

Der Ableitungsbaum, der mithilfe des Earley Parsers und der Tokens 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
     ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt
     decls_defs
       decl_def
         struct_decl
 6
           name
                        st
           struct_params
 8
9
             alloc
                type_spec
10
                  prim_dt
                                  int
11
                pntr_decl
12
                  pntr_deg
13
                  array_decl
14
                    pntr_decl
15
                      pntr_deg
                      array_decl
17
                        name
                                     attr
18
                        array_dims
19
                    array_dims
20
                      4
                      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
33
                    type_spec
34
                      struct_spec
35
                        name
                                     st
36
                    pntr_decl
37
                      pntr_deg
38
                      array_decl
39
                        pntr_decl
40
                          pntr_deg
                          array_decl
                            name
                                          var
                             array_dims
44
                               3
45
                               2
                        array_dims
```

Code 3.3: Ableitungsbaum nach Ableitungsbaum Generierung.

3.2.3.2 Ausgabe des Ableitunsgbaumes

Die Ausgabe des Ableitungsbaumes wird komplett vom Lark Parsing Toolkit übernommen. Für die Inneren Knoten werden die Nicht-Terminalsymbole, welche in der Konkreten Grammatik den linken Seiten des ::=-Symbols¹³ entsprechen hergenommen und die Blätter sind Terminalsymbole, genauso, wie es in der Definition 2.36 eines Ableitungsbaumes 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 Abstrakten Syntaxbaumes ist bewusst so gewählt, dass sie sich optisch vom Ableitungsbaum unterscheidet, indem die Bezeichner der Knoten in UpperCamelCase¹⁴ geschrieben sind, im Gegensatz zum Ableitungsbaum, dessen Innere Knoten im snake_case geschrieben sind, wie auch die Nicht-Terminalsymbole auf den linken Seiten des ::=-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 Tramsformers 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 "Feld der Mächtigkeit 3 von Zeigern auf Felder der Mächtigkeit 2 von Integern" int (*ar[3])[2] die Spiralregel¹⁵ in der Implementeirung des PicoC-Compilers umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden Produktionen in der Konkreten Grammatik 3.2.8 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.

¹³ Grammar: The language of languages (BNF, EBNF, ABNF and more).

¹⁴Naming convention (programming).

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 Zeigerdeklaratoren pntr_decl(num, datatype) und Felddeklaratoren 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 Zeigerdeklarator pntr_decl(num, datatype) und Felddeklarator array_decl(datatype, nums). Daher muss beim FeldDeclarator 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 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.

```
file
     ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
     decls_defs
 4
5
       decl_def
         struct_decl
           name
                        st
 7
8
9
           struct_params
             alloc
               pntr_decl
10
                  pntr_deg
                  array_decl
                    array_dims
                      4
14
                      5
                    pntr_decl
16
                      pntr_deg
17
                      array_decl
18
                        array_dims
19
                        type_spec
20
                          prim_dt
                                          int
               name
                             attr
22
       decl_def
23
         fun_def
24
           type_spec
25
             prim_dt
                              void
26
           pntr_deg
27
           name
                        main
28
           fun_params
           decl_exec_stmts
```

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

Code 3.4: Ableitungsbaum nach Ableitungsbaum Vereinfachung.

3.2.5 Generierung des Abstrakten Syntaxbaumes

Nachdem der Ableitungsbaum 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 Abstrakten Syntaxbaum aus ihm zu generieren. Würde man den vereinfachten Ableitungsbaum des Ausdrucks struct st *(*var[3][2]) auf passende Weise in einen Abstrakten 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 von oben-nach-unten¹⁸ als "Zeiger auf einen Zeiger auf ein Feld der Mächtigkeit 2 von Feldern der Mächtigkeit 3 von Verbunden 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 "Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Zeigern auf einen Zeiger auf einen Verbund vom Typ st" ist.

¹⁷https://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.

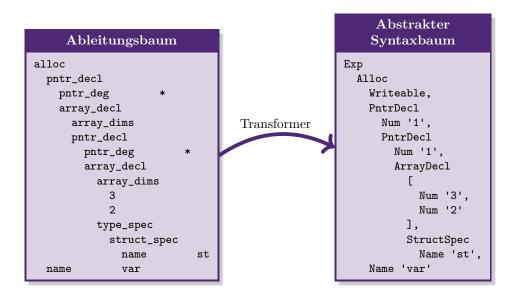


Abbildung 3.4: Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen.

Dieser Abstrakte Syntaxbaum ist für die Weiterverarbeitung ungeeignet, denn für die Adressberechnung für eine Aneinandereihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundattribute, welche in Unterkapitel 3.3.5.2 genauer erläutert wird, will man den Datentyp in umgekehrter Reihenfolge. Aus diesem Grund muss der Transformer bei der Konstruktion des Abstrakten Syntaxbaumes 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.

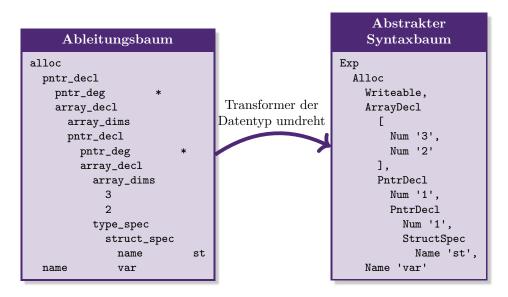


Abbildung 3.5: Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.

Die Weiterverarbeitung des Abstrakten Syntaxbaumes geschieht mithilfe von Passes, welche im Unterkapitel 2.5 genauer beschrieben werden. Da die Knoten des Abstrakten Syntaxbaumes anders als beim Ableitungsbaum nicht die gleichen Bezeichnungen haben wie Produktionen der Konkreten Grammatik

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 Abstrakten Syntaxbäumen vorkommen.

Des Weiteren gibt die Abstrakte Grammatik 3.2.10 in Unterkapitel 3.2.5.4 Aufschluss darüber welche Kompositionen von PicoC-Knoten neben den bereits in Tabelle 3.8 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 Abstrakten Syntaxbäumen vorkommen sind in Tabelle 3.3 mit einem Beschreibungstext dokumentiert.

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

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

Tabelle 3.3: PicoC-Knoten Teil 1.

PiocC-Knoten	Beschreibung
<pre>Exp(exp, datatype, error_data)</pre>	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 Zeigerdatentyp steht: <prim_dt> *<var>, wobei das Attribut num die Anzahl zusammen- gefasster Zeiger angibt und datatype der Datentyp ist, auf den der oder die Zeiger zeigen.</var></prim_dt>
Ref(exp, datatype, error_data)	Container, der für die Anwendung des Referenz-Operators & <var> steht und die Adresse einer Location (Definition 2.48) 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.</var>
Deref(lhs, exp)	Container für den Indexzugriff auf einen Feld- oder Zei- gerdatentyp: <var>[<i>], wobei exp1 eine angehängte weite- re 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.</i></var>
ArrayDecl(nums, datatype)	Container, der für den Felddatentyp steht: <prim_dt> <var>[<i>], wobei das Attribut nums eine Liste von Num('x') ist, die die Dimensionen des Feld angibt und datatype der Datentyp ist, der über das Anwenden von Subscript() auf das Feld zugreifbar ist.</i></var></prim_dt>
Array(exps, datatype)	Container für den Initializer eines Feldes, dessen Einträge exps weitere Initializer für eine Feld-Dimension oder ein Initializer für einen Verbund 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 Indexzugriff auf einen Feld- oder Zeigerdatentyp: <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.</i></var>
StructSpec(name)	Container für einen selbst definierten Verbundstyp: struct <name>, wobei das Attribut name festlegt, welchen selbst definierten Verbundstyp dieser Knoten repräsentiert.</name>
Attr(exp, name)	Container für den Attributzugriff auf einen Verbundsdatentyp: <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.</attr></var>

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initializer eines Verbundes, z.B {. <attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(1hs, exp) ist mit einer Zuordnung eines Attributezeichners, zu einem weiteren Initializer für eine Feld-Dimension oder zu einem Initializer für einen Verbund 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.</attr2></attr1>
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Verbundstyps, z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;};, wobei name der Bezeichner des Verbundstyps ist und allocs eine Liste von Bezeichnern der Attribute des Verbundstyps mit dazugehörigem Datentyp, wofür sich der Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.</attr2></datatype></attr1></datatype></var>
If(exp, stmts)	Container für ein If-Anweisung if(<exp>) { <stmts> } in- klusive Condition exp und einem Branch stmts, indem eine Liste von Anweisungen stehen kann oder ein einzelnes GoTo(Name('block.xyz')).</stmts></exp>
<pre>IfElse(exp, stmts1, stmts2)</pre>	Container für ein If-Else Anweisung if (<exp>) { <stmts2> } else { <stmts2> } inklusive Codition exp und 2 Branches stmts1 und stmts2, die zwei Alternativen Darstellen in denen jeweils Listen von Anweisungen oder GoTo(Name('block.xyz'))'s stehen können.</stmts2></stmts2></exp>
While(exp, stmts)	Container für ein While-Anweisung while(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Anweisungen stehen kann oder ein einzelnes GoTo(Name('block.xyz')).</stmts></exp>
DoWhile(exp, stmts)	Container für ein Do-While-Anweisung do { <stmts> } while(<exp>); inklusive Condition exp und einem Branch stmts, indem eine Liste von Anweisungen stehen kann oder ein einzelnes GoTo(Name('block.xyz')).</exp></stmts>
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-Anweisung: return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis vom Return-Anweisung zurückgegeben wird.</exp>
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 Knoten Alloc(type_spec, datatype, name) als Cotainer für die Parameter dient.</param2></datatype></param1></datatype></fun_name></datatype>

Tabelle 3.5: PicoC-Knoten Teil 3.

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs,	Container für eine Funktionsdefinition, z.B. <datatype></datatype>
stmts_blocks)	<pre><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 Knoten Alloc(type_spec, datatype, name) als Con- tainer für die Parameter dient und stmts_blocks eine Liste von Statemetns bzw. Blöcken ist, welche diese Funktion beinhaltet.</stmts></datatype></fun_name></pre>
NewStackframe(fun_name, goto_after_call)	Container für die Erstellung eines neuen Stackframes und Speicherung des Werts des BAF-Registers der aufrufenden Funktion und der Rücksprungadresse nacheinander an den Anfang des neuen Stackframes. Das Attribut fun_name stehte dabei für den Bezeichner der Funktion, für die ein neuer Stackframe erstellt werden soll. Das Attribut fun_name dient später dazu den Block dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attributen gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die Adresse des Befehls, der direkt auf den Sprungbefehl folgt,
RemoveStackframe()	ersetzt wird. Container für das Entfernen des aktuellen Stackframes durch das Wiederherstellen des im noch aktuellen Stackframe gespeicherten Werts des BAF-Registes der aufrufenden Funktion und das Setzen des SP-Registers auf den Wert des BAF-Registesr vor der Wiederherstellung.
File(name, decls_defs_blocks)	Container für alle Funkionen oder Blöcke, welche eine Datei als Ursprung haben, wobei name der Dateiname der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von Funktionen bzw. Blöcken ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für Anweisungen, der auch als Block bezeichnet wird, wobei das Attribut name der Bezeichner des Labels (Definition 5.2) des Blocks ist und stmts_instrs eine Liste von Anweisungen oder Befehlen. Zudem besitzt er noch 3 versteckte Attribute, wobei instrs_before die Zahl der Befehle vor diesem Block zählt, num_instrs die Zahl der Befehle ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die Parameter der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für ein Goto zu einem anderen Block, wobei das Attribut name der Bezeichner des Labels des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen Kommentar, den der Compiler selber während des Kompiliervorangs erstellt, der im RETI-Interpreter selbst später nicht sichtbar sein wird, aber in den Immediate-Dateien, welche die Abstrakten Syntaxbäume nach den verschiedenen Passes enthalten.
RETIComment(value)	Container für einen Kommentar im Code der Form: // # comment, der im RETI-Intepreter später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer RETI-CPU nicht umsetzbar ist und auch nicht sinnvoll wäre umzusetzen. Der Kommentar ist im Attribut value, welches jeder Knoten besitzt gespeichert.

Anmerkung Q

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

Jeder Knoten hat darüberhinaus auch noch 2 Attribute value und position. Das Attribut value entspricht bei einem Blatt dem Tokenwert des Tokens welches es ersetzt. Bei Inneren Knoten ist das Attribut value hingegen unbesetzt. Das Attribut position wird für Fehlermeldungen gebraucht.

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 erläutertem 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 Abstrakten Syntaxbäumen vorkommen sind in Tabelle 3.2.5.1 mit einem Beschreibungstext dokumentiert.

RETI-Knoten	Beschreibung
Program(name, instrs)	Container für alle Befehle: <name> <instrs>, wobei name</instrs></name>
	der Dateiname der Datei ist, die erstellt wird und instrs
	eine Liste von Befehlen ist.
<pre>Instr(op, args)</pre>	Container für einen Befehl: <op> <args>, wobei op eine</args></op>
	Operation ist und args eine Liste von Argumenten
	für dieser Operation.
Jump(rel, im_goto)	Container für einen Sprungbefehl: JUMP <rel> <im>, wo-</im></rel>
	bei rel eine Relation ist und im_goto ein Immediate
	Value Im(val) für die Anzahl an Speicherzellen, um
	die relativ zum Sprungbefehl 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</im>
	die Interrruptvektornummer (IVN) für die passende
	Speicherzelle in der Interruptvektortabelle ist, in der
	die Adresse der Interrupt-Service-Routine (ISR) steht.
Call(name, reg)	Container für einen Prozeduraufruf: CALL <name> <reg>,</reg></name>
	wobei name der Bezeichner der Prozedur, die aufgerufen
	werden soll ist und reg ein Register ist, das als Argu-
	ment an die Prozedur dient. Diese Operation ist in der
	Betriebssysteme Vorlesung ^a nicht deklariert, sondern wur-
	de dazuerfunden, um unkompliziert ein CALL PRINT ACC
	oder CALL INPUT ACC im RETI-Interpreter simulieren zu
	können.
Name(val)	Bezeichner für eine Prozedur, z.B. PRINT oder INPUT oder
	den Programnamen, z.B. PROGRAMNAME. Dieses Argu-
	ment ist in der Betriebssysteme Vorlesung ^a nicht dekla-
	riert, sondern wurde dazuerfunden, um Bezeichner, wie
Post(most)	PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg) Im(val)	Container für ein Register. Ein Immediate Value, z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), Xor(),	Compute-Memory oder Compute-Register Operatio-
Or(), And()	nen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(),	Compute-Immediate Operationen: ADDI, SUBI, MULTI,
Xori(), Ori(), Andi()	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(),	Relationen: <, <=, >, >=, ==, !=, _NOP.
Always(), NOp()	
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(),	Register: PC, IN1, IN2, ACC, SP, BAF, CS, DS.
Cs(), Ds()	

^a 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.

Komposition	Beschreibung
Ref(Global(Num('addr')))	Speichert Adresse der Speicherzelle, die Num ('addr') Spei-
	cherzellen relativ zum Datensegment Register DS steht
	auf den Stack.
<pre>Ref(Stackframe(Num('addr')))</pre>	Speichert Adresse der Speicherzelle, die Num ('addr') Spei-
	cherzellen relativ zum Begin-Aktive-Funktion Regis-
	ter BAF steht auf den Stack.
<pre>Ref(Subscr(Stack(Num('addr1')),</pre>	Berechnet die nächste Adresse aus der Adresse, die an
<pre>Stack(Num('addr2'))))</pre>	Speicherzelle Stack(Num('addr1')) steht und dem Subs-
	<pre>cript Index, der an Speicherzelle Stack(Num('addr2'))</pre>
	steht und speichert diese auf den Stack. Die Berechnung
	ist abhängig davon ob der Datentyp ArrayDecl(datatype)
	oder PntrDecl(datatype) ist. Der Datentyp ist ein ver-
	stecktes Attribut von Ref(exp).
<pre>Ref(Attr(Stack(Num('addr1')),</pre>	Berechnet die nächste Adresse aus der Adresse, die
<pre>Name('attr')))</pre>	an Speicherzelle Stack(Num('addr1')) steht und dem At-
	tributnamen Name('attr') und speichert diese auf den
	Stack. Zur Berechnung ist der Name des Verbundes
	in StructSpec(Name('st')) notwendig, dessen Attribut
	Name('attr') ist. StructSpec(Name('st')) ist ein versteck-
	tes Attribut von Ref(exp).
<pre>Assign(Stack(Num('size'))),</pre>	Schreibt Num('size') viele Speicherzellen, die ab
<pre>Global(Num('addr')))</pre>	Global(Num('addr')) relativ zum Datensegment Regis-
	ter DS stehen, versetzt genauso auf den Stack.
<pre>Assign(Stack(Num('size')),</pre>	Schreibt Num('size') viele Speicherzellen, die ab
<pre>Stackframe(Num('addr')))</pre>	Stackframe(Num('addr')) relativ zum Begin-Aktive-
	Funktion Register BAF stehen, versetzt genauso auf den
	Stack.
<pre>Exp(Global(Num('addr'))</pre>	Speichert Inhalt der Speicherzelle, die Num('addr') Spei-
	cherzellen relativ zum Datensegment Register DS steht
	auf den Stack.
<pre>Exp(Stackframe(Num('addr'))</pre>	Speichert Inhalt der Speicherzelle, die Num('addr') Spei-
	cherzellen relativ zum Begin-Aktive-Funktion Regis-
	ter BAF steht auf den Stack.
<pre>Exp(Stack(Num('addr')))</pre>	Speichert Inhalt der Speicherzelle, die Num('addr') Spei-
	cherzellen relativ zum Stackpointer Register SP steht
	auf den Stack.
Assign(Stack(Num('addr1')),	Speichert Inhalt der Speicherzelle Stack(Num('addr2')),
Stack(Num('addr2')))	die Num('addr2') Speicherzellen relativ zum Stackpoin-
	ter Register SP steht an der Adresse in der Speicherzelle,
	die Num('addr1') Speicherzellen relativ zum Stackpoin-
	ter Register SP steht.
Assign(Global(Num('addr')),	Schreibt Num('size') viele Speicherzellen, die auf dem
<pre>Stack(Num('size')))</pre>	Stack stehen, versetzt genauso auf die Speicherzellen ab
	Num('addr') relativ zum Datensegment Register DS.
Assign(Stackframe(Num('addr')),	Schreibt Num('size') viele Speicherzellen, die auf dem
Stack(Num('size')))	Stack stehen, versetzt genauso auf die Speicherzellen ab
	Num('addr') relativ zum Begin-Aktive-Funktion Re-
	gister BAF.
<pre>Exp(Reg(reg))</pre>	Schreibt den aktuellen Wert des Registers reg auf den
	Stack.
<pre>Instr(Loadi(), [Reg(Acc()),</pre>	Lädt in das Register ACC die Adresse des Befehls, der in
GoTo(Name('addr@next_instr'))])	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.

Anmerkung Q

Um die obige Tabelle 3.8 nicht mit unnötig viel repetetiven Inhalt zu füllen wurden die zahlreichen Kompostionen 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 Grammatik

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

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle) RETIComment()$	$L_Comment$
un_op bin_op	::=	$egin{array}{c c c c c c c c c c c c c c c c c c c $	L_Arith_Bit
exp $stmt$::=	$Name(\langle str \rangle) \mid Num(\langle str \rangle) \mid Char(\langle str \rangle)$ $BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle)$ $UnOp(\langle un_op \rangle, \langle exp \rangle) \mid Call(Name('input'), Empty())$ $Call(Name('print'), \langle exp \rangle)$ $Exp(\langle exp \rangle)$	
un_op rel bin_op exp	::= ::= ::=	$\begin{array}{c cccc} LogicNot() & & & \\ Eq() & NEq() & Lt() & LtE() & Gt() & GtE() \\ LogicAnd() & LogicOr() & & & \\ Atom(\langle exp \rangle, \langle rel \rangle, \langle exp \rangle) & & ToBool(\langle exp \rangle) \end{array}$	L_Logic
type_qual datatype exp stmt	::= ::= ::=	$Const() \mid Writeable() \\ IntType() \mid CharType() \mid VoidType() \\ Alloc(\langle type_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) \\ Assign(\langle exp \rangle, \langle exp \rangle)$	L_Assign_Alloc
$\begin{array}{c} datatype \\ exp \end{array}$::=	$PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)$ $Deref(\langle exp \rangle, \langle exp \rangle) \mid Ref(\langle exp \rangle)$	L_Pntr
$\begin{array}{c} datatype \\ exp \end{array}$::=	$\begin{array}{c c} ArrayDecl(Num(\langle str \rangle)+,\langle datatype \rangle) \\ Subscr(\langle exp \rangle,\langle exp \rangle) & Array(\langle exp \rangle+) \end{array}$	L_Array
datatype exp decl_def	::= ::= ::=	$StructSpec(Name(\langle str \rangle)) \\ Attr(\langle exp \rangle, Name(\langle str \rangle)) \\ Struct(Assign(Name(\langle str \rangle), \langle exp \rangle) +) \\ StructDecl(Name(\langle str \rangle), \\ Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) +) \\$	L_Struct
stmt	::=	$If(\langle exp \rangle, \langle stmt \rangle *)$ $IfElse(\langle exp \rangle, \langle stmt \rangle *, \langle stmt \rangle *)$	L_If_Else
stmt	::=	$While(\langle exp \rangle, \langle stmt \rangle *) $ $DoWhile(\langle exp \rangle, \langle stmt \rangle *)$	L_Loop
exp $stmt$ $decl_def$::= ::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *)$ $Return(\langle exp \rangle)$ $FunDecl(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *)$ $FunDef(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle stmt \rangle *)$	L_Fun
file	::=	$File(Name(\langle str \rangle), \langle decl_def \rangle *)$	L _ $File$

Grammatik 3.2.10: Abstrakte Grammatik der Sprache L_{PiocC} in ASF

3.2.5.5 Codebeispiel

In Code 3.5 ist der Abstrakte 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',
```

```
StructDecl
          Name 'st',
 7
8
            Alloc
              Writeable.
 9
              PntrDecl
10
                 Num '1',
11
                 ArrayDecl
12
13
                     Num '4',
14
                     Num '5'
15
                   ],
16
                   PntrDecl
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
                   PntrDecl
35
                     Num '1',
                     PntrDecl
36
37
                        Num '1',
38
                        StructSpec
39
                          Name 'st',
40
                 Name 'var'
41
          ]
42
     ]
```

Code 3.5: Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum.

3.2.5.6 Ausgabe des Abstrakten Syntaxbaumes

Ein Teilbaum eines Abstrakten Syntaxbaumes kann entweder in der Konkreten Syntax der Sprache, für dessen Kompilierung er generiert wurde oder in der Abstrakten Syntax, die beschreibt, wie der Abstrakte Syntaxbaum selbst aufgebaut sein darf ausgegeben werden.

Das Ausgeben eines Abstrakten Syntaxbaumes wird im PicoC-Compiler über die Magische Methode $_repr_()^{20}$ der Programmiersprache L_{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

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.

; usw. zur Darstellung der Hierarchie und zur Abtrennung zurück. Dabei wird nach dem Prinzip der Tiefensuche der gesamte Abstrakte Syntaxbaum 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.

Beim PicoC-Compiler wurden Abstrakte und Konkrete Syntax miteinander gemischt. Für PicoC-Knoten wurde die Abstrakte Syntax verwendet, da Passes schließlich auf Abstrakten Syntaxbäumen 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 gescheifte 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 Abstrakten Syntaxbaumes als Ergebnis der Lexikalischen und Syntaktischen Analyse in Unterkapitel 2.4, wird in diesem Kapitel auf Basis der verschiedenen Kompositionen von PicoC-Knoten und RETI-Knoten im Abstrakten Syntaxbaum das gewünschte Endprodukt des PicoC-Compilers, der RETI-Code generiert.

Man steht nun dem Problem gegenüber einen Abstrakten Syntaxbaum der Sprache L_{PicoC} , der durch die Abstrakte Grammatik 3.2.10 spezifiziert ist in einen entsprechenden Abstrakten 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 2.44) herunterbricht.

Beim PicoC-Compiler handelt es sich um einen Cross-Compiler (Definiton 2.6). Damit RETI-Code erzeugt werden kann, der auf der RETI-Architektur läuft, muss erst, wie im T-Diagram (siehe Unterkapitel 2.1.1) in Abbildung 3.6 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-Diagram 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.

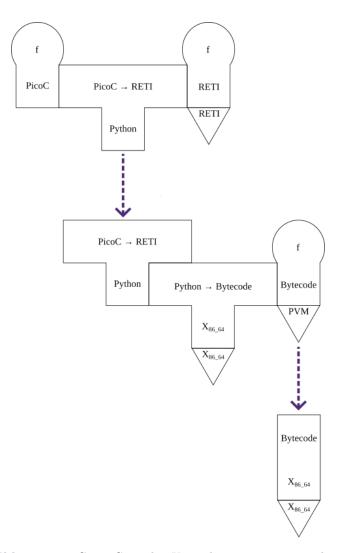


Abbildung 3.6: Cross-Compiler Kompiliervorgang ausgeschrieben.

Dieses längliche T-Diagram in Abbildung 3.6 lässt sich zusammenfassen, sodass man das T-Diagram in Abbildung 3.7 erhält, in welcher direkt angegeben ist, dass der PicoC-Compiler in $X_{86.64}$ -Maschinensprache geschrieben ist.

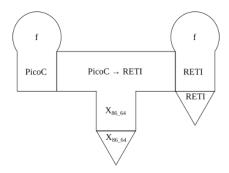


Abbildung 3.7: Cross-Compiler Kompiliervorgang Kurzform.

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

Kompilierprozess im Folgenden im horinzontalen, 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 2.1) des PicoC-Compilers aufeinanderfolgen. In der Pipe-Architektur nutzt der jeweils nächste Pass den generierten Abstrakten Syntaxbaum des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen Abstrakten Syntaxbaum in seiner eigenen Sprache zu generieren.

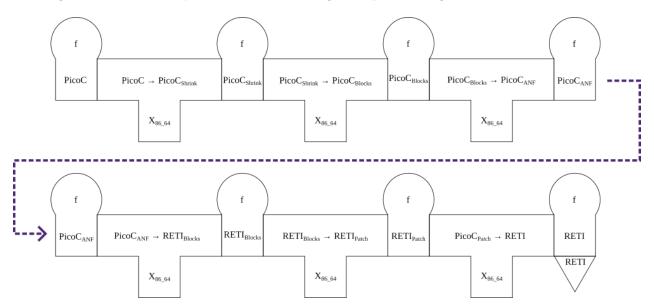


Abbildung 3.8: Architektur mit allen Passes ausgeschrieben.

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 Zeigern, Feldern, Verbunden 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 verwendenten Ansätze zur Lösung dieser Probleme basieren auf der Vorlesung C. Scholl, "Betriebssysteme" und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem PicoC-Compiler auch in der Praxis implementiert werden konnten.

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 veschiedenen Aspekte²¹ der 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 Zeigern, Feldern, Verbunden 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 3.3.2.2 ausführlich an einem Beispiel erklärt. Kurzgefasst hat der PicoC-Shrink Pass die Aufgabe, die Eigenheit auszunutzen, dass der Dereferenzierungoperator *pntr und die damit einhergehende Zeigerarithmetik *(pntr + i) sich in der Untermenge

²¹In kurz: Zeiger, Felder, Verbunde und Funktionen.

der Sprache L_C , welche die Sprache L_{PicoC} darstellt genau gleich verhält, wie der Operator für den Zugriff auf den Index eines Feldes ar[i].

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

3.3.1.1.2 Abstrakte Grammatik

Die Abstrakte Grammatik 3.3.1 der Sprache L_{PicoC_Shrink} ist fast identisch mit der Abstrakten Grammatik 3.2.10 der Sprache L_{PicoC} , nach welcher der erste Abstrakte Syntaxbaum in der Syntaktischen Analyse generiert wurde. Der einzige Unterschied liegt darin, dass es den Knoten Deref(exp, exp) in Abstrakten Grammatik 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 nach der Abstrakten Grammatik der Sprache L_{PicoC} definiert ist.

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle) \mid RETIComment()$	$L_Comment$
un_op bin_op exp	::= ::= ::= 	$\begin{array}{c cccc} Minus() & & Not() \\ Add() & & Sub() & & Mul() & & Div() & & Mod() \\ Oplus() & & And() & & Or() \\ Name(\langle str \rangle) & & Num(\langle str \rangle) & & Char(\langle str \rangle) \\ BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle) & & Call(Name('input'), Empty()) \\ UnOp(\langle un_op \rangle, \langle exp \rangle) & & Call(Name('input'), Empty()) \\ Call(Name('print'), \langle exp \rangle) & & Exp(\langle exp \rangle) \end{array}$	L_Arith_Bit
un_op rel bin_op exp	::=	$\begin{array}{c cccc} LogicNot() & \\ Eq() & & NEq() & & Lt() & & LtE() & & Gt() & & GtE() \\ LogicAnd() & & LogicOr() & \\ Atom(\langle exp \rangle, \langle rel \rangle, \langle exp \rangle) & & ToBool(\langle exp \rangle) & \end{array}$	L_Logic
type_qual datatype exp stmt	::= ::= ::=	$Const() \mid Writeable() \\ IntType() \mid CharType() \mid VoidType() \\ Alloc(\langle type_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) \\ Assign(\langle exp \rangle, \langle exp \rangle)$	L_Assign_Alloc
$\begin{array}{c} datatype \\ exp \end{array}$::=	$\begin{array}{c c} PntrDecl(Num(\langle str \rangle), \langle datatype \rangle) \\ Deref(\langle exp \rangle, \langle exp \rangle) & Ref(\langle exp \rangle) \end{array}$	L_Pntr
$\begin{array}{c} datatype \\ exp \end{array}$::=	$ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle) Subscr(\langle exp \rangle, \langle exp \rangle) Array(\langle exp \rangle+)$	L_Array
datatype exp decl_def	::= ::= ::=	$StructSpec(Name(\langle str \rangle)) \\ Attr(\langle exp \rangle, Name(\langle str \rangle)) \\ Struct(Assign(Name(\langle str \rangle), \langle exp \rangle) +) \\ StructDecl(Name(\langle str \rangle), \\ Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) +) \\$	L_Struct
stmt	::=	$If(\langle exp \rangle, \langle stmt \rangle *)$ $IfElse(\langle exp \rangle, \langle stmt \rangle *, \langle stmt \rangle *)$	L_If_Else
stmt	::=	$While(\langle exp \rangle, \langle stmt \rangle *) \\ DoWhile(\langle exp \rangle, \langle stmt \rangle *)$	L_Loop
exp stmt decl_def	::= ::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *)$ $Return(\langle exp \rangle)$ $FunDecl(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *)$ $FunDef(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle stmt \rangle *)$	L_Fun
file	::=	$File(Name(\langle str \rangle), \langle decl_def \rangle *)$	$L_{-}File$

Grammatik 3.3.1: Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF

Anmerkung Q

Ein rot markierter Knoten bedeutet, dass dieser im Vergleich zur vorherigen Abstrakten Grammatik 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 C. Scholl, "Betriebssysteme", welcher in der Vorlesung allerdings rekursiv implementiert ist.

Dieser rekursive Algoirthmus ist allerdings kein gutes Anschaungsbeispiel, 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-Anweisungen 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-Statemtens zu enthalten.

Beide Varianten des Algorithmus wurden zum Testen des PicoC-Compilers verwendet und sind als Tests im Ordner /tests unter Link²², 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 Zeigern, Feldern, Verbunden und Funktionen mit eigenen Codebeispielen erklärt werden und alle sonstigen Details dem Bachelorprojekt zuzurechnen sind.

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

Code 3.6: PicoC Code für Codebespiel.

In Code 3.7 sieht man den Abstrakten 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  [
```

 $^{^{22}}$ https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```
Alloc(Writeable(), IntType('int'), Name('n'))
 9
         ],
10
         Γ
11
           Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
12
           While
13
             Num '1',
14
             Γ
               Ιf
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
         Γ
29
           Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])))
30
     ]
```

Code 3.7: Abstrakter Syntaxbaum für Codebespiel.

Im PicoC-Shrink-Pass ändert sich nichts im Vergleich zum Abstrakten 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(lable)- 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 Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 3.3.1 der Sprache L_{PicoC_Shrink} 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(lable)-und IfElse(exp, stmts1, stmts2)-Knoten ersetzt wurden. Die Funktionsdefinition FunDef($\langle datatype \rangle$, Name(str), Alloc(Writeable(), $\langle datatype \rangle$, Name(str))*, $\langle block \rangle$ *) ist nun ein Container für Blöcke Block(Name(str), $\langle stmt \rangle$ *) und keine Anweisungen stmt mehr. Das resultiert in der Abstrakten Grammatik 3.3.2 der Sprache L_{PicoC_Blocks} .

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle) RETIComment()$	$L_Comment$
un_op bin_op	::=	$Minus() \mid Not()$ $Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$ $Oplus() \mid And() \mid Or()$	L_Arith_Bit
exp	::=	$Name(\langle str \rangle) \mid Num(\langle str \rangle) \mid Char(\langle str \rangle)$ $BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle)$ $UnOp(\langle un_op \rangle, \langle exp \rangle) \mid Call(Name('input'), Empty())$ $Call(Name('print'), \langle exp \rangle)$	
stmt	::=	$Exp(\langle exp \rangle)$	
un_op rel bin_op exp	::= ::= ::=	$LogicNot() \\ Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE() \\ LogicAnd() \mid LogicOr() \\ Atom(\langle exp \rangle, \langle rel \rangle, \langle exp \rangle) \mid ToBool(\langle exp \rangle)$	$L_{-}Logic$
type_qual datatype exp stmt	::= ::= ::=	$Const() \mid Writeable() \\ IntType() \mid CharType() \mid VoidType() \\ Alloc(\langle type_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) \\ Assign(\langle exp \rangle, \langle exp \rangle)$	L_Assign_Alloc
$\begin{array}{c} datatype \\ exp \end{array}$::= ::=	$PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)$ $Ref(\langle exp \rangle)$	L_Pntr
$\begin{array}{c} datatype \\ exp \end{array}$::=	$\begin{array}{c c} ArrayDecl(Num(\langle str \rangle)+,\langle datatype \rangle) \\ Subscr(\langle exp \rangle,\langle exp \rangle) & Array(\langle exp \rangle+) \end{array}$	L_Array
datatype exp decl_def	::= ::= ::=	$StructSpec(Name(\langle str \rangle))$ $Attr(\langle exp \rangle, Name(\langle str \rangle))$ $Struct(Assign(Name(\langle str \rangle), \langle exp \rangle)+)$ $StructDecl(Name(\langle str \rangle), \langle exp \rangle)$	L_Struct
stmt	::=	$Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))+)$ $If(\langle exp \rangle, \langle stmt \rangle *)$ $IfElse(\langle exp \rangle, \langle stmt \rangle *, \langle stmt \rangle *)$	L_If_Else
stmt	::=	$While(\langle exp \rangle, \langle stmt \rangle *) \ DoWhile(\langle exp \rangle, \langle stmt \rangle *)$	L_Loop
exp $stmt$ $decl_def$::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *)$ $Return(\langle exp \rangle)$ $FunDecl(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *)$ $FunDef(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle block \rangle *)$	L_Fun
block $stmt$::=	$Block(Name(\langle str \rangle), \langle stmt \rangle *) \ GoTo(Name(\langle str \rangle))$	L_Blocks

Grammatik 3.3.2: Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF

Anmerkung 9

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

Die Abstrakte Grammatik soll im Gegensatz zur Konkreten Grammatik 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.

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

3.3.1.2.3 Codebeispiel

In Code 3.8 sieht man den Abstrakten Syntaxbaum des PiocC-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 Anweisungen der jeweiligen Funktionen bis zur letzten Anweisung oder bis zum ersten Auftauchen eines If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(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.

```
Name './example_faculty_it.picoc_blocks',
 4
       FunDef
 5
         IntType 'int',
 6
         Name 'faculty',
           Alloc(Writeable(), IntType('int'), Name('n'))
 9
         ],
10
         Γ
11
           Block
12
             Name 'faculty.6',
13
14
               Assign(Alloc(Writeable(), 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
             Γ
21
               IfElse
22
                 Num '1',
23
                 Γ
24
                    GoTo(Name('while_branch.4'))
25
                 ],
26
                 Γ
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')),
```

```
GoTo(Name('if.3'))
38
                  ],
39
                  Ε
                    GoTo(Name('if_else_after.2'))
                  ]
42
             ],
43
           Block
44
              Name 'if.3',
45
46
                Return(Name('res'))
47
              ],
48
           Block
49
              Name 'if_else_after.2',
50
              Γ
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
              Name 'while_after.1',
56
57
58
         ],
59
       FunDef
60
         VoidType 'void',
61
         Name 'main',
62
         [],
63
         Γ
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 Codebespiel.

3.3.1.3 PicoC-ANF Pass

3.3.1.3.1 Aufgabe

Die Aufgabe des PicoC-ANF Passes ist es den Abstrakten Syntaxbaum der Sprache L_{PicoC_Blocks} in die Abstrakte Grammatik der Sprache L_{PicoC_ANF} umzuformen, welche in A-Normalform (Definition 2.51) und damit auch in Monadischer Normalform (Definition 2.47) ist. Um Wiederholung zu vermeiden wird zur Erklärung der A-Normalform auf Unterkapitel 2.5.2 verwiesen.

Zudem wird eine Symboltabelle (Definition 3.7) 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 Symbol²³ als Schlüssel in der Symboltabelle nachgeschlagen und anstelle des Name(str)-Knotens die in

²³Bzw. der Bezeichner

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 Sichtbarkeitsbereichs (z.B. @scope), der in der Symboltabelle an den Bezeichner drangehängt ist (z.B. identifier@scope).²⁴

Definition 3.7: 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 Kompiliervorgangs 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 Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 3.3.2 der Sprache L_{PicoC_Blocks} 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 Grammatik 3.3.3 der Sprache L_{PicoC_ANF} der Sprache L_{PicoC_ANF} neue Knoten eingeführt werden.

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

²⁴Die Umsetzung von Sichtbarkeitsbereichen wird in Unterkapitel 3.3.6.2 genauer beschrieben.

```
RETIComment()
                                                                                                                                                  L_{-}Comment
stmt
                               SingleLineComment(\langle str \rangle, \langle str \rangle)
                      ::=
                                                                                                                                                  L_Arith_Bit
un\_op
                      ::=
                              Minus()
                                                   Not()
bin\_op
                      ::=
                               Add()
                                          Sub()
                                                             Mul() \mid Div() \mid
                                                                                              Mod()
                                                             |Or()
                               Oplus()
                                            And()
                              Name(\langle str \rangle) \mid Num(\langle str \rangle)
                                                                                Char(\langle str \rangle)
                                                                                                         Global(Num(\langle str \rangle))
exp
                               Stackframe(Num(\langle str \rangle))
                                                                       | Stack(Num(\langle str \rangle))|
                               BinOp(Stack(Num(\langle str \rangle)), \langle bin\_op \rangle, Stack(Num(\langle str \rangle)))
                               UnOp(\langle un\_op \rangle, Stack(Num(\langle str \rangle))) \mid Call(Name('input'), Empty())
                               Call(Name('print'), \langle exp \rangle)
                               Exp(\langle exp \rangle)
                              LogicNot()
                                                                                                                                                  L\_Logic
un\_op
                      ::=
                               Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt()
rel
                                                                                                         GtE()
                      ::=
                               LogicAnd()
                                                      LogicOr()
bin\_op
                      ::=
                               Atom(Stack(Num(\langle str \rangle)), \langle rel \rangle, Stack(Num(\langle str \rangle)))
exp
                      ::=
                              ToBool(Stack(Num(\langle str \rangle)))
                              Const()
                                                 Writeable()
                                                                                                                                                  L\_Assign\_Alloc
type\_qual
                      ::=
                              IntType() \mid CharType() \mid VoidType()
datatype
                      ::=
                              Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle))
exp
                      ::=
                              Assign(Global(Num(\langle str \rangle)), Stack(Num(\langle str \rangle)))
stmt
                      ::=
                               Assign(Stackframe(Num(\langle str \rangle)), Stack(Num(\langle str \rangle)))
                               Assign(Stack(Num(\langle str \rangle)), Global(Num(\langle str \rangle)))
                               Assign(Stack(Num(\langle str \rangle)), Stackframe(Num(\langle str \rangle)))
                               PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)
                                                                                                                                                  L_{-}Pntr
datatype
                      ::=
                               Ref(Global(\langle str \rangle)) \mid Ref(Stackframe(\langle str \rangle))
                               Ref(Subscr(\langle exp \rangle, \langle exp \rangle \mid Ref(Attr(\langle exp \rangle, Name(\langle str \rangle)))))
                               ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle)
                                                                                                                                                  L_-Array
datatupe
                      ::=
                               Subscr(\langle exp \rangle, Stack(Num(\langle str \rangle)))
                                                                                        Array(\langle exp \rangle +)
exp
                      ::=
datatype
                               StructSpec(Name(\langle str \rangle))
                                                                                                                                                  L\_Struct
                      ::=
                               Attr(\langle exp \rangle, Name(\langle str \rangle))
exp
                      ::=
                               Struct(Assign(Name(\langle str \rangle), \langle exp \rangle)+)
decl\_def
                               StructDecl(Name(\langle str \rangle),
                      ::=
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) +)
                               IfElse(Stack(Num(\langle str \rangle)), \langle stmt \rangle *, \langle stmt \rangle *)
                                                                                                                                                  L_If_Else
stmt
                      ::=
                              Call(Name(\langle str \rangle), \langle exp \rangle *)
                                                                                                                                                  L-Fun
                      ::=
exp
                               StackMalloc(Num(\langle str \rangle)) \mid NewStackframe(Name(\langle str \rangle), GoTo(\langle str \rangle))
stmt
                      ::=
                               Exp(GoTo(Name(\langle str \rangle))) \mid RemoveStackframe()
                               Return(Empty()) \mid Return(\langle exp \rangle)
decl\_def
                               FunDecl(\langle datatype \rangle, Name(\langle str \rangle))
                      ::=
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))*)
                               FunDef(\langle datatype \rangle, Name(\langle str \rangle),
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))*, \langle block \rangle*)
block
                               Block(Name(\langle str \rangle), \langle stmt \rangle *)
                                                                                                                                                  L\_Blocks
                      ::=
stmt
                               GoTo(Name(\langle str \rangle))
                      ::=
                                                                                                                                                  L_File
file
                               File(Name(\langle str \rangle), \langle block \rangle *)
symbol\_table
                               SymbolTable(\langle symbol \rangle *)
                                                                                                                                                  L\_Symbol\_Table
                      ::=
                               Symbol(\langle type\_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)
symbol
                      ::=
                              Empty()
type\_qual
                      ::=
datatype
                      ::=
                               BuiltIn()
                                                    SelfDefined()
                               Name(\langle str \rangle)
name
                      ::=
val
                               Num(\langle str \rangle)
                                                   | Empty()
                      ::=
                               Pos(Num(\langle str \rangle), Num(\langle str \rangle))
                                                                                  Empty()
pos
                      ::=
                               Num(\langle str \rangle)
                                                     Empty()
size
                                                                                                                                                                88
```

3.3.1.3.3 Codebeispiel

In Code 3.9 sieht man den Abstrakten Syntaxbaum des PiocC-ANF Passes für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem alle Anweisungen 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',
 4
       Block
         Name 'faculty.6',
 6
 7
8
           // Assign(Name('res'), Num('1'))
           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')),
           IfElse
18
19
             Stack
20
                Num '1',
21
             Ε
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
             ]
         ],
```

```
Block
47
         Name 'if.3',
48
           // Return(Name('res'))
           Exp(Stackframe(Num('1')))
           Return(Stack(Num('1')))
51
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',
71
           Return(Empty())
72
         ],
73
       Block
         Name 'main.0',
74
75
           StackMalloc(Num('2'))
           Exp(Num('4'))
           NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
           Exp(GoTo(Name('faculty.6')))
80
           RemoveStackframe()
81
           Exp(ACC)
           Exp(Call(Name('print'), [Stack(Num('1'))]))
82
83
           Return(Empty())
84
         ]
85
     ]
```

Code 3.9: Pico C-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 Anweisungen in den Blöcken, die durch PicoC-Knoten im Abstrakten Syntaxbaum der Sprache L_{PicoC_ANF} dargestellt sind durch ihren entsprechenden RETI-Knoten zu ersetzen.

3.3.1.4.2 Abstrakte Grammatik

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

Block(Name(str), (instr)*) und File(Name(str), (block)*), da das gesamte Konzept mit den Blöcken erst im RETI-Pass in Unterkapitel 3.3.8 aufgelöst wird.

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

Grammatik 3.3.4: Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF

3.3.1.4.3 Codebeispiel

In Code 3.10 sieht man den Abstrakten Syntaxbaum des RETI-Blocks Passes für das aus Unterkapitel 3.6 weitergeführte Beispiel, indem die Anweisungen, die durch entsprechende PicoC-Knoten im Abstrakten Syntaxbaum der Sprache $L_{PicoC_ANF}^{25}$ repräsentiert waren nun durch ihre entsprechennden RETI-Knoten ersetzt werden.

```
File
 2
    Name './example_faculty_it.reti_blocks',
     Γ
       Block
         Name 'faculty.6',
 6
7
8
9
           # // Assign(Name('res'), Num('1'))
           # Exp(Num('1'))
           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',
         [
```

²⁵Beschrieben durch die Grammatik 3.3.3.

```
# // 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
         Name 'while_branch.4',
36
         Γ
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;
           STOREIN SP ACC 1;
42
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
         Ε
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',
           # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
           # Exp(Stackframe(Num('0')))
```

```
SUBI SP 1;
81
           LOADIN BAF ACC -2;
82
           STOREIN SP ACC 1;
83
           # Exp(Stackframe(Num('1')))
           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;
00
           LOADIN BAF ACC -2;
L01
           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;
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
113
           LOADIN SP ACC 1;
           STOREIN BAF ACC -2;
114
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',
L21
         Ε
           # Return(Empty())
123
           LOADIN BAF PC -1;
124
         ],
L25
       Block
126
         Name 'main.0',
L27
128
           # StackMalloc(Num('2'))
L29
           SUBI SP 2;
130
           # Exp(Num('4'))
131
           SUBI SP 1;
           LOADI ACC 4;
132
           STOREIN SP ACC 1;
133
134
           # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
L35
           MOVE BAF ACC;
           ADDI SP 3;
```

```
MOVE SP BAF;
           SUBI SP 4;
.39
           STOREIN BAF ACC 0;
           LOADI ACC GoTo(Name('addr@next_instr'));
           ADD ACC CS;
           STOREIN BAF ACC -1;
43
           # Exp(GoTo(Name('faculty.6')))
L44
           Exp(GoTo(Name('faculty.6')))
L45
           # RemoveStackframe()
146
           MOVE BAF IN1;
L47
           LOADIN IN1 BAF 0;
48
           MOVE IN1 SP;
49
           # Exp(ACC)
150
           SUBI SP 1;
151
           STOREIN SP ACC 1;
152
           LOADIN SP ACC 1;
153
           ADDI SP 1;
154
           CALL PRINT ACC;
L55
            # Return(Empty())
156
           LOADIN BAF PC -1;
L57
158
```

Code 3.10: RETI-Blocks Pass für Codebespiel.

Anmerkung 9

Wenn der Abstrakte Syntaxbaum ausgegeben wird, ist die Darstellung nicht auschließlich in Abstrakter Syntax, da die RETI-Knoten aus bereits im Unterkapitel 3.2.5.6 vermitteltem Grund in Konkreter Syntax ausgeben 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 Abstrakten Syntaxbaumes, 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.²⁶
- 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 -2²¹ oder größer 2²¹ 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 -(2³¹) oder größer 2³¹ 1 ist, wird eine Fehlermeldung TooLargeLiteral ausgegeben.

²⁶Das fällt unter die Themenbereiche des Bachelorprojekts und wird daher nicht genauer erläutert.

3.3.1.5.2 Abstrakte Grammatik

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

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

Grammatik 3.3.5: Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF

3.3.1.5.3 Codebeispiel

In Code 3.11 sieht man den Abstrakten Syntaxbaum des PiocC-Patch Passes für das aus Unterkapitel 3.6 weitergeführte Beispiel. Durch den RETI-Patch Pass wurde hier ein start. <nummer>-Block²⁷ eingesetzt, da die main-Funktion nicht die erste Funktion ist. Des Weiteren wurden durch diesen Pass einzelne GoTo(Name(str))-Anweisungen entfernt²⁸, die nur einen Sprung um eine Position entsprochen hätten.

```
File
 2
    Name './example_faculty_it.reti_patch',
     Γ
       Block
         Name 'start.7',
 7
8
9
           # // Exp(GoTo(Name('main.0')))
           Exp(GoTo(Name('main.0')))
         ],
10
       Block
11
         Name 'faculty.6',
12
13
           # // Assign(Name('res'), Num('1'))
           # Exp(Num('1'))
15
           SUBI SP 1;
16
           LOADI ACC 1:
           STOREIN SP ACC 1;
17
18
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
```

 $^{^{27}\}mathrm{Dieser}$ Block wurde im Code 3.8 markiert.

²⁸Diese entfernten GoTo(Name(str))'s' wurden ebenfalls im Code 3.8 markiert.

```
LOADIN SP ACC 1;
20
           STOREIN BAF ACC -3;
21
           ADDI SP 1;
           # // While(Num('1'), [])
           # 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'))
           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;
           # 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'))
           # Exp(Stackframe(Num('1')))
           SUBI SP 1;
           LOADIN BAF ACC -3;
           STOREIN SP ACC 1;
```

```
76
           # Return(Stack(Num('1')))
           LOADIN SP ACC 1;
77
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;
01
           STOREIN BAF ACC -3:
           ADDI SP 1:
102
103
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104
           # Exp(Stackframe(Num('0')))
           SUBI SP 1;
106
           LOADIN BAF ACC -2;
L07
           STOREIN SP ACC 1;
108
           # Exp(Num('1'))
109
           SUBI SP 1;
           LOADI ACC 1;
L10
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;
17
           ADDI SP 1;
18
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
119
           LOADIN SP ACC 1;
120
           STOREIN BAF ACC -2;
l21
           ADDI SP 1;
122
           # Exp(GoTo(Name('condition_check.5')))
123
           Exp(GoTo(Name('condition_check.5')))
124
         ],
125
       Block
L26
         Name 'while_after.1',
L27
128
           # Return(Empty())
L29
           LOADIN BAF PC -1;
130
         ],
l31
       Block
132
         Name 'main.0',
```

```
Γ
            # StackMalloc(Num('2'))
            SUBI SP 2;
            # Exp(Num('4'))
            SUBI SP 1;
137
138
            LOADI ACC 4;
139
            STOREIN SP ACC 1;
L40
            # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
L41
            MOVE BAF ACC;
42
            ADDI SP 3;
143
           MOVE SP BAF;
44
            SUBI SP 4;
45
            STOREIN BAF ACC 0;
146
           LOADI ACC GoTo(Name('addr@next_instr'));
47
            ADD ACC CS;
148
            STOREIN BAF ACC -1;
149
            # Exp(GoTo(Name('faculty.6')))
150
            Exp(GoTo(Name('faculty.6')))
L51
            # RemoveStackframe()
            MOVE BAF IN1;
152
153
           LOADIN IN1 BAF O;
154
            MOVE IN1 SP;
155
            # Exp(ACC)
156
            SUBI SP 1;
            STOREIN SP ACC 1;
157
158
            LOADIN SP ACC 1;
            ADDI SP 1;
159
160
            CALL PRINT ACC;
161
            # Return(Empty())
62
            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 Grammatik

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

Ausgegeben wird der finale Maschinencode allerdings in Konkreter Syntax, die durch die Konkreten Grammatiken 3.3.6 und 3.3.7 für jeweils die Lexikalische und Syntaktische Analyse beschrieben wird. Der Grund, warum die Konkrete Grammatik 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.

```
"6"
dig\_no\_0
                                                                       L_Program
                 "7"
                          "8"
                                  "<sub>9"</sub>
                 "0"
dig_with_0
            ::=
                          dig\_no\_0
                 "0"
                         dig\_no\_0 dig\_with\_0* | "-"dig\_no\_0*
num
            ::=
                 "a"..."Z"
letter
            ::=
                 letter(letter \mid dig\_with\_0 \mid \_)*
name
                 "ACC"
                             "IN1" | "IN2"
                                                |"PC""|"SP"
reg
            ::=
                             "CS" | "DS"
                 "BAF"
arg
            ::=
                 reg
                         num
                            "!=" | "<" | "<=" | ">"
rel
            ::=
                  ">="
                            "\_NOP"
```

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

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

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

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

Grammatik 3.3.8: Abstrakte Grammatik der Sprache L_{RETI} in ASF

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 auschließlich aus RETI-Ausdrücken bestehen²⁹, 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 Abstrakte Syntaxbaum, wenn er in eine Datei ausgegeben wird, direkt RETI-Code in menschenlesbarer Repräsentation erzeugt.

```
# // 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;
```

²⁹Wie z.B. LOADI ACC GoTo(Name('addr@next_instr')), Exp(GoTo(Name('main.0'))) und JUMP== GoTo(Name('if_else_after.2')).

```
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;
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;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
06 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
107 MOVE BAF ACC;
08 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
11 STOREIN BAF ACC 0;
12 LOADI ACC 80;
13 ADD ACC CS;
14 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6')))
116 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;
125 ADDI SP 1;
26 CALL PRINT ACC;
27 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 3.12: RETI Pass für Codebespiel.

3.3.2 Umsetzung von Zeigern

Die Umsetzung von Zeigern ist in diesem Unterkapitel schnell erklärt, auch Dank eines kleinen Taschenspielertricks³⁰. Hierbei sind nur die Operationen für Referenzierung und Dereferenzierung in den Unterkapiteln 3.3.2.1 und 3.3.2.2 zu erläutern. Referenzierung kann dazu genutzt werden einen Zeiger zu initialisieren und Dereferenzierung kann dazu genutzt werden, um auf diesen später zuzugreifen.

3.3.2.1 Referenzierung

Referenzierung (z.B. &var) ist eine Operation bei der ein Zeiger auf eine Location in Form der Anfangsadresse dieser Location als Ergebnis zurückgegeben wird. Die Implementierung der Referenzierung 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 Zeigerreferenzierung.

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

```
1
  File
2
    Name './example_pntr_ref.ast',
     Γ
      FunDef
         VoidType 'void',
6
7
8
9
         Name 'main',
         [],
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10
           Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),

→ Ref(Name('var')))
11
    ]
```

Code 3.14: Abstrakter Syntaxbaum für Zeigerreferenzierung.

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

Anmerkung Q

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

³⁰Später mehr dazu.

```
SymbolTable
     Γ
       Symbol
 4
         {
           type qualifier:
                                     Empty()
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
                                     Name('main')
           name:
                                     Empty()
           value or address:
 9
                                     Pos(Num('1'), Num('5'))
           position:
10
           size:
                                     Empty()
11
         },
12
       Symbol
13
         {
14
           type qualifier:
                                     Writeable()
           datatype:
                                     IntType('int')
16
                                     Name('var@main')
           name:
17
                                     Num('0')
           value or address:
18
                                     Pos(Num('2'), Num('6'))
           position:
19
                                     Num('1')
           size:
20
         },
21
       Symbol
22
         {
23
                                     Writeable()
           type qualifier:
24
                                     PntrDecl(Num('1'), IntType('int'))
           datatype:
25
                                     Name('pntr@main')
           name:
26
           value or address:
                                     Num('1')
27
           position:
                                     Pos(Num('3'), Num('7'))
28
                                     Num('1')
           size:
29
         }
30
    ]
```

Code 3.15: Symboltabelle für Zeigerreferenzierung.

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, deren Bedeutung in Unterkapitel 3.2.5 erklärt wurde. Im Fall, dass in Ref(exp)) das exp vielleicht nicht direkt ein Name('var') enthält und exp z.B. ein Subscr(Attr(Name('var'), Name('attr'))) 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, name) nach dem Schema in Subkapitel 3.3.5.2 kümmern.

```
Name './example_pntr_ref.picoc_mon',
4
      Block
5
        Name 'main.0',
           // Assign(Name('var'), Num('42'))
8
           Exp(Num('42'))
9
           Assign(Global(Num('0')), Stack(Num('1')))
10
           // Assign(Name('pntr'), Ref(Name('var')))
11
           Ref(Global(Num('0')))
12
           Assign(Global(Num('1')), Stack(Num('1')))
13
           Return(Empty())
```

```
14 ]
15 ]
```

Code 3.16: PicoC-ANF Pass für Zeigerreferenzierung.

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 semantisch entsprechenden RETI-Knoten ersetzt, deren Bedeutungen im Appendix in Unterkapitel 4.3 erklärt sind.

```
1
  File
 2
    Name './example_pntr_ref.reti_blocks',
 4
       Block
         Name 'main.0',
 6
           # // 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 Zeigerreferenzierung.

3.3.2.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

Dereferenzierung (z.B. *var) ist eine Operation bei der einem Zeiger zur Location hin gefolgt wird, auf welche dieser zeigt und das Ergebnis z.B. der Inhalt der ersten Speicherzelle der referenzierten Location ist. Die Implementierung von Dereferenzierung 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 Zeigerdereferenzierung.

Der Knoten Deref(Name('var'), Num('0'))) repräsentiert im Abstrakten Syntaxbaum in Code 3.19 eine Dereferenzierung *var. Es gibt hierbei zwei Fälle. Bei der Anwendung von Zeigerarithmetik, wie z.B. *(var + 2 - 1) übersetzt sich diese zu Deref(Name('var'), BinOp(Num('2'), Sub(), Num('1'))). Bei einer normalen Dereferenzierung, wie z.B. *var, übersetzt sich diese zu Deref(Name('var'), Num('0'))³¹.

```
File
2
    Name './example_pntr_deref.ast',
    Ε
4
      FunDef
5
        VoidType 'void',
6
        Name 'main',
        [],
8
9
          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10
          Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),

→ Ref(Name('var')))
11
          Exp(Deref(Name('pntr'), Num('0')))
12
13
    ]
```

Code 3.19: Abstrakter Syntaxbaum für Zeigerdereferenzierung.

Im PicoC-Shrink Pass in Code 3.20 wird ein Trick angewandet, bei dem jeder Knoten Deref(Name('pntr'), Num('0')) einfach durch den Knoten Subscr(Name('pntr'), Num('0')) ersetzt wird. Die Bedeutung des letzteren Knoten wurde in Unterkapitel 3.2.5 erklärt. Der Trick besteht darin, dass der Dereferenzierungsoperator (z.B. *(var + 1)) sich identisch zum Operator für den Zugriff auf einen Feldindex (z.B. var[1]) verhält, wie es bereits im Unterkapitel 1.3 erläutert wurde. Damit spart man sich viele vermeidbare Fallunterscheidungen und doppelten Code und kann die Derefenzierung (z.B. *(var + 1)) einfach von den Routinen für einen Zugriff auf einen Feldindex (z.B. var[1]) übernehmen lassen.

```
File
Name './example_pntr_deref.picoc_shrink',

[
FunDef
VoidType 'void',
Name 'main',
[],
[],
[]
Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
Ref(Name('var')))
Exp(Subscr(Name('pntr'), Num('0')))
```

³¹Das Num('0') steht dafür, dass dem Zeiger gefolgt wird, aber danach nicht noch mit einem Versatz von der Größe des Datentyps in diesem Kontext auf eine nebenliegende Location zugegriffen wird.

```
12 ]
13 ]
```

Code 3.20: PicoC-Shrink Pass für Zeigerdereferenzierung.

3.3.3 Umsetzung von Feldern

Bei Feldern ist in diesem Unterkapitel die Umsetzung der Innitialisierung eines Feldes 3.3.3.1, des Zugriffs auf einen Feldindex 3.3.3.2 und der Zuweisung an einen Feldindex 3.3.3.3 zu klären.

3.3.3.1 Initialisierung eines Feldes

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

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

void fun() {
  int ar[2][2] = {{3, 4}, {5, 6}};
}
```

Code 3.21: PicoC-Code für die Initialisierung eines Feldes.

Die Initialisierung eines Feldes intar[2][1]={{3+1},{4}} wird im Abstrakten 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
2
    Name './example_array_init.ast',
    Γ
4
      FunDef
         VoidType 'void',
6
        Name 'main',
7
8
         [],
         Γ
           Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
           → Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
              Array([Num('4')])))
        ],
10
11
      FunDef
12
         VoidType 'void',
        Name 'fun',
13
14
         [],
         [
16
           Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
               Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
17
    ]
```

Code 3.22: Abstrakter Syntaxbaum für die Initialisierung eines Feldes.

Bei der Initialisierung eines Feldes wird zuerst Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int'))) ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann³². Das Definieren der Variable ar erfolgt mittels der Symboltabelle, die in Code 3.23 dargestellt ist.

Bei Variablen auf dem Stackframe wird ein Feld rückwärts auf das Stackframe geschrieben und auch die Adresse des ersten Elements als Adresse des Feldes genommen. Dies macht den Zugriff auf einen Feldindex in Subkapitel 3.3.3.2 deutlich unkomplizierter, da man so nicht mehr zwischen Stackframe und Globalen Statischen Daten beim Zugriff auf einen Feldindex unterscheiden muss, da es Probleme macht, dass ein Stackframe in die entgegengesetzte Richtung wächst, verglichen mit den Globalen Statischen Daten³³.

Anmerkung Q

Das Größe des Feldes datatype $ar[dim_1]\dots[dim_k]$, die ihm size-Attribut des Symboltabelleneintrags eingetragen ist, berechnet sich dabei aus der Mächtigkeit der einzelnen Dimensionen des Feldes multipliziert mit der Größe des grundlegenden Datentyps der einzelnen Feldelemente: $size(datatype(ar)) = \left(\prod_{j=1}^n dim_j\right) \cdot size(datatype)^a$.

^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

```
SymbolTable
 2
     Γ
 3
       Symbol
           type qualifier:
                                     Empty()
           datatype:
                                     FunDecl(VoidType('void'), Name('main'), [])
7
8
                                     Name('main')
           name:
                                     Empty()
           value or address:
 9
           position:
                                     Pos(Num('1'), Num('5'))
10
           size:
                                     Empty()
11
         },
12
       Symbol
13
         {
14
           type qualifier:
                                     Writeable()
15
                                     ArrayDecl([Num('2'), Num('1')], IntType('int'))
           datatype:
16
                                     Name('ar@main')
           name:
17
                                     Num('0')
           value or address:
18
                                     Pos(Num('2'), Num('6'))
           position:
19
                                     Num('2')
           size:
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                     Empty()
24
                                     FunDecl(VoidType('void'), Name('fun'), [])
           datatype:
25
           name:
                                     Name('fun')
26
           value or address:
                                     Empty()
```

³²Das Widerspricht der üblichen Auswertungsreihenfolge beim Zuweisungsoperator =, der rechtsassoziativ ist. Der Zuweisungsoperator = tritt allerdings erst später in Aktion.

³³Wenn 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.

```
Pos(Num('5'), Num('5'))
           position:
28
           size:
                                     Empty()
29
         },
30
       Symbol
31
         {
           type qualifier:
32
                                     Writeable()
33
                                     ArrayDecl([Num('2'), Num('2')], IntType('int'))
           datatype:
34
                                     Name('ar@fun')
           name:
35
           value or address:
                                     Num('3')
36
           position:
                                     Pos(Num('6'), Num('6'))
37
                                     Num('4')
           size:
38
39
     ]
```

Code 3.23: Symboltabelle für die Initialisierung eines Feldes.

Im PiocC-ANF Pass in Code 3.24 werden zuerst die Logischen Ausdrücke in den Blättern des Teilbaumes, der beim Feld-Initializers Knoten Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])]) anfängt nach dem Prinzip der Tiefensuche, von links-nach-rechts ausgewertet und auf den Stack geschrieben³⁴.

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 grundlegenden Datentyp³⁵ das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man einfach die Größe des Feldes 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 Feldes 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.

```
File
    Name './example_array_init.picoc_mon',
    Γ
4
      Block
        Name 'main.1',
          // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
           → Array([Num('4')])))
          Exp(Num('3'))
9
          Exp(Num('1'))
10
          Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
11
          Exp(Num('4'))
12
          Assign(Global(Num('0')), Stack(Num('2')))
          Return(Empty())
```

³⁴Da der Zuweisungsoperator = rechtsassoziativ ist und auch rein logisch, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

 $^{^{35}}$ Z.B. ein Verbund, sodass es ein "Feld von Verbunden" ist.

```
],
15
       Block
16
         Name 'fun.0',
17
           // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
19
           Exp(Num('3'))
20
           Exp(Num('4'))
21
           Exp(Num('5'))
22
           Exp(Num('6'))
23
           Assign(Stackframe(Num('3')), Stack(Num('4')))
24
           Return(Empty())
25
         ]
26
    1
```

Code 3.24: PicoC-ANF Pass für die Initialisierung eines Feldes.

Im RETI-Blocks Pass in Code 3.25 werden die Kompositionen Exp(exp) und Assign(Global(Num('0')), Stack(Num('2'))) bzw. Assign(Stackframe(Num('3')), Stack(Num('4'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
1 File
 2
    Name './example_array_init.reti_blocks',
    Ε
 4
       Block
 5
         Name 'main.1',
 6
           # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
           → Array([Num('4')])]))
           # Exp(Num('3'))
 9
           SUBI SP 1;
10
           LOADI ACC 3;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('1'))
13
           SUBI SP 1;
14
           LOADI ACC 1;
15
           STOREIN SP ACC 1;
16
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
17
           LOADIN SP ACC 2;
18
           LOADIN SP IN2 1;
19
           ADD ACC IN2;
20
           STOREIN SP ACC 2;
21
           ADDI SP 1;
22
           # Exp(Num('4'))
23
           SUBI SP 1;
24
           LOADI ACC 4;
25
           STOREIN SP ACC 1;
26
           # Assign(Global(Num('0')), Stack(Num('2')))
27
           LOADIN SP ACC 1;
28
           STOREIN DS ACC 1;
29
           LOADIN SP ACC 2;
30
           STOREIN DS ACC 0;
           ADDI SP 2;
           # Return(Empty())
```

```
LOADIN BAF PC -1;
34
         ],
35
       Block
36
         Name 'fun.0',
37
38
           # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
39
           # Exp(Num('3'))
           SUBI SP 1;
40
41
           LOADI ACC 3;
42
           STOREIN SP ACC 1;
43
           # Exp(Num('4'))
44
           SUBI SP 1;
45
           LOADI ACC 4;
46
           STOREIN SP ACC 1;
47
           # Exp(Num('5'))
48
           SUBI SP 1;
49
           LOADI ACC 5;
50
           STOREIN SP ACC 1;
51
           # 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;
           STOREIN BAF ACC -2;
57
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 die Initialisierung eines Feldes.

3.3.3.2 Zugriff auf einen Feldindex

Der Zugriff auf einen Feldindex (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 Feldindex.

Der Zugriff auf einen Feldindex ar[0] wird im Abstrakten Syntaxbaum in Code 3.27 mithilfe des Knotens Subscr(Name('ar'), Num('0')) dargestellt.

```
1
  File
 2
    Name './example_array_access.ast',
     Ε
 4
       FunDef
         VoidType 'void',
         Name 'main',
         [],
 8
 9
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
           → Array([Num('42')]))
           Exp(Subscr(Name('ar'), Num('0')))
10
11
         ],
12
       FunDef
13
         VoidType 'void',
14
         Name 'fun',
15
         [],
16
17
           Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),

→ Array([Num('1'), Num('2'), Num('3')]))
18
           Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19
         ٦
20
    ]
```

Code 3.27: Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.

Im PicoC-ANF Pass in Code 3.28 wird vom Knoten Subscr(Name('ar'), Num('0')) zuerst im Anfangsteil 3.3.5.1 die Adresse der Variable Name('ar') auf den Stack geschrieben. Bei den Globalen Statischen Daten der main-Funktion wird das durch die Komposition Ref(Global(Num('0'))) dargestellt und beim Stackframe der Funktionm fun wird das durch die Komposition Ref(Stackframe(Num('2'))) dargestellt.

In nächsten Schritt, dem Mittelteil 3.3.5.2 wird die Adresse ab der das Feldelement des Feldes auf das Zugegriffen werden soll anfängt berechnet. Dabei wurde im Anfangsteil bereits die Anfangsadresse des Feldes, in dem dieses Feldelement liegt auf den Stack gelegt. Da ein Index auf den Zugegriffen werden soll auch durch das Ergebnis eines komplexeren Ausdrucks, z.B. ar[1 + var] bestimmt sein kann, indem auch Variablen vorkommen können, kann dieser nicht während des Kompilierens berechnet werden, sondern muss zur Laufzeit berechnet werden.

Daher muss zuerst der Wert des Index, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch Exp(Num('0')) und dann muss die Adresse des Index berechnet werden, was durch die Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) dargestellt wird. Die Bedeutung der Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) ist in Tabelle 3.8 dokumentiert.

Zur Adressberechnung ist es notwendig auf die Dimensionen (z.B. [Num('3')]) des Feldes, auf dessen Feldelement zugegriffen wird, zugreifen zu können. Daher ist der Felddatentyp (z.B. ArrayDecl([Num('3')], IntType('int'))) dem Knoten Ref(exp, datatype) als verstecktes Attribut datatype angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) angehängt.

Je nachdem, ob mehrere Subscr(exp, exp) eine Komposition bilden (z.B. Subscr(Subscr(Name('var'), Num('1')), Num('1'))) ist es notwendig mehrere Adressberechnungsschritte für den Index Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) einzuleiten und es muss auch möglich sein, z.B. einen Attributzugriff var.attr und eine Zugriff auf einen Arryindex var[1] miteinander zu kombinieren, was in Subkapitel 3.3.5.2 allgemein erklärt ist.

Im letzten Schritt, dem Schlussteil 3.3.5.3 wird der Inhalt des Index, dessen Adresse in den vorherigen Schritten berechnet wurde, nun auf den Stack geschrieben, wobei dieser die Adresse auf dem Stack ersetzt, die es zum Finden des Index brauchte. Dies wird durch den Knoten Exp(Stack(Num('1'))) dargestellt. Je nachdem, welchen Datentyp die Variable ar hat und auf welchen Unterdatentyp folglich im Kontext zuletzt zugegriffen wird, abhängig davon wird der Schlussteil Exp(Stack(Num('1'))) auf eine andere Weise verarbeitet (siehe Subkapitel 3.3.5.3). Der Unterdatentyp ist dabei ein verstecktes Attribut des Exp(Stack(Num('1')))-Knoten.

Der einzige Unterschied, je nachdem, ob der Zugriff auf einen Feldindex (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-Befehle für eine Variable, die in den Globalen Statischen Daten liegt und eine Variable, die auf dem Stackframe liegt erzeugt werden müssen.

Anmerkung 9

Die Berechnung der Adresse, ab der ein Feldelement eines Feldes datatype $ar[dim_1]...[dim_n]$ abgespeichert ist, kann mittels der Formel 3.3.1:

$$\operatorname{ref}(\operatorname{ar}[\operatorname{idx_1}] \dots [\operatorname{idx_n}]) = \operatorname{ref}(\operatorname{ar}) + \left(\sum_{i=1}^n \left(\prod_{j=i+1}^n \operatorname{dim}_j\right) \cdot \operatorname{idx_i}\right) \cdot \operatorname{size}(\operatorname{datatype}) \tag{3.3.1}$$

aus der Betriebssysteme Vorlesung C. Scholl, "Betriebssysteme" berechnet werden^a.

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 Feldelement, der als Faktor idxi in der Formel auftaucht.

Der Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) repräsentiert dabei einen ausmultiplizierten Summanden $\left(\prod_{j=i+1}^n \dim_j\right) \cdot \mathrm{id} x_i \cdot \mathrm{size}(\mathrm{datatpye})$ in der Formel.

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

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

```
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
         Name 'fun.0',
18
19
20
           // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
21
           Exp(Num('1'))
           Exp(Num('2'))
22
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 Feldindex.

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

```
2
     Name './example_array_access.reti_blocks',
     Γ
 4
       Block
         Name 'main.1',
           # // Assign(Name('ar'), Array([Num('42')]))
 8
           # Exp(Num('42'))
           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;
           STOREIN SP IN1 1;
22
           # Exp(Num('0'))
           SUBI SP 1;
```

```
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;
           LOADI ACC 2;
50
51
           STOREIN SP ACC 1;
52
           # Exp(Num('3'))
53
           SUBI SP 1;
           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;
           SUBI IN1 4;
68
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;
           LOADI ACC 1;
76
           STOREIN SP ACC 1;
78
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
           LOADIN SP ACC 2;
           LOADIN SP IN2 1;
```

```
ADD ACC IN2;
82
           STOREIN SP ACC 2;
83
           ADDI SP 1;
84
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
           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;
           STOREIN SP ACC 1;
94
95
           # Return(Empty())
96
           LOADIN BAF PC -1;
97
         ]
    ]
```

Code 3.29: RETI-Blocks Pass für Zugriff auf einen Feldindex.

3.3.3.3 Zuweisung an Feldindex

Die Zuweisung eines Wertes an einen Feldindex (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[1] = 42;
4 }
```

Code 3.30: PicoC-Code für Zuweisung an Feldindex.

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

```
File
Name './example_array_assignment.ast',

[
FunDef
VoidType 'void',
Name 'main',
[],
[]
Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
Assign(Subscr(Name('ar'), Num('1')), Num('42'))
]
]
]
```

Code 3.31: Abstrakter Syntaxbaum für Zuweisung an Feldindex.

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

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

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 Feldindex, dessen Adresse im Schritt danach berechnet wurde dar.

```
File
    Name './example_array_assignment.picoc_mon',
4
5
6
      Block
         Name 'main.0',
7
8
9
           // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
           Exp(Num('42'))
           Ref(Global(Num('0')))
10
           Exp(Num('1'))
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 Feldindex.

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.

```
Name './example_array_assignment.reti_blocks',
     Γ
4
5
      Block
        Name 'main.0',
           # // Assign(Subscr(Name('ar'), Num('1')), 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;
           STOREIN SP IN1 1;
```

³⁶Ist in Tabelle 3.8 genauer beschrieben ist

```
# Exp(Num('1'))
           SUBI SP 1;
19
          LOADI ACC 1;
20
          STOREIN SP ACC 1;
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
          LOADIN SP IN1 2;
          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;
          ADDI SP 2;
          STOREIN IN1 ACC 0;
33
          # Return(Empty())
34
          LOADIN BAF PC -1;
         ]
36
    ]
```

Code 3.33: RETI-Blocks Pass für Zuweisung an Feldindex.

3.3.4 Umsetzung von Verbunden

Bei Verbunden wird in diesem Unterkapitel zunächst geklärt, wie die **Deklaration von Verbundstypen** umgesetzt ist. Ist ein Verbundstyp deklariert, kann damit einhergehend ein Verbund mit diesem Verbundstyp definiert werden. Die Umsetzung von beidem wird in Unterkapitel 3.3.4.1 erläutert. Des Weiteren ist die Umsetzung der Innitialisierung eines Verbundes 3.3.4.2, des **Zugriffs auf ein Verbundsattribut** 3.3.4.3, der **Zuweisung an ein Verbundsattribut** 3.3.4.4 zu klären.

3.3.4.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Deklaration (Definition 1.7) eines neuen Verbundstyps (z.B. struct st {int len; int ar[2];}) und die Definition (Definition 1.8) eines Verbundes mit diesem Verbundstyp (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 Verbundstyps.

Bevor irgendwas definiert werden kann, muss erstmal ein Verbundstyp deklariert werden. Im Abstrakten Syntaxbaum in Code 3.36 wird die Deklaration eines Verbundstyps 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 **Verbundstyp** struct st st_var; wird durch die Komposition Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')) dargestellt.

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

Code 3.35: Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.

Für den Verbundstyp 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 Verbundstyps [Name('len@st'), Name('ar@st')] aufgelistet, sodass man über den Verbundstyp st die Attribute des Verbundstyps in der Symboltabelle nachschlagen kann. Die Schlüssel der Attribute haben einen Suffix @st angehängt, der eine Art Sichtbarkeitsbereich innerhalb des Verbundtyps für seine Attribut darstellt. Es gilt foglich, dass innerhalb eines Verbundstyps zwei Attribute nicht gleich benannt werden können, aber dafür zwei unterschiedliche Verbundstypen ihre Attribute gleich benennen können.

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

Für die Definition einer Variable st_var@main mit diesem Verbundstyp st wird ein Eintrag in der Symboltabelle angelegt. Das datatyp-Attribut enthält dabei den Namen des Verbundstyps als Komposition StructSpec(Name('st')), wodurch jederzeit alle wichtigen Informationen zu diesem Verbundstyp³⁷ und seinen Attributen in der Symboltabelle nachgeschlagen werden können.

Anmerkung Q

Die Größe einer Variable st_var, die ihm size-Attribut des Symboltabelleneintrags eingetragen ist und mit dem Verbundstyp struct st {datatype₁ attr₁; ... datatype_n attr_n; };^a definiert ist (struct st st_var;), berechnet sich dabei aus der Summe der Größen der einzelnen Datentypen datatype₁ ... datatype_n der Attribute attr₁, ... attr_n des Verbundstyps: 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 Feldes vor die Variable zu schreiben von Lc übernommen. Es wird so getann, als würde der komplette Datentyp immer hinter der Variable stehen: datatype var.

```
SymbolTable
 2
     Ε
       Symbol
         {
            type qualifier:
                                      Empty()
 6
7
8
9
           datatype:
                                      IntType('int')
           name:
                                      Name('len@st')
                                      Empty()
           value or address:
                                      Pos(Num('1'), Num('15'))
           position:
10
                                      Num('1')
           size:
11
         },
12
       Symbol
13
         {
14
            type qualifier:
15
                                      ArrayDecl([Num('2')], IntType('int'))
           datatype:
16
                                      Name('ar@st')
           name:
17
                                      Empty()
           value or address:
18
           position:
                                      Pos(Num('1'), Num('24'))
19
           size:
                                      Num('2')
20
         },
21
       Symbol
22
         {
            type qualifier:
                                      Empty()
```

 $^{^{37}}$ Wie z.B. vor allem die Größe bzw. Anzahl an Speicherzellen, die dieser Verbundstyp einnimmt.

```
datatype:
                                    StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
               Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
               Name('ar'))])
                                    Name('st')
                                     [Name('len@st'), Name('ar@st')]
           value or address:
27
           position:
                                    Pos(Num('1'), Num('7'))
28
           size:
                                    Num('3')
29
         },
30
       Symbol
31
32
           type qualifier:
                                    Empty()
33
                                    FunDecl(VoidType('void'), Name('main'), [])
           datatype:
34
                                    Name('main')
           name:
35
                                    Empty()
           value or address:
36
                                    Pos(Num('3'), Num('5'))
           position:
37
           size:
                                    Empty()
38
         },
39
       Symbol
40
         {
41
                                    Writeable()
           type qualifier:
42
                                    StructSpec(Name('st'))
           datatype:
43
                                    Name('st_var@main')
44
                                    Num('0')
           value or address:
45
                                    Pos(Num('4'), Num('12'))
           position:
46
                                    Num('3')
           size:
47
    ]
```

Code 3.36: Symboltabelle für die Deklaration eines Verbundstyps.

3.3.4.2 Initialisierung von Verbunden

Die Initialisierung eines Verbundes 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 Verbunden.

Im Abstrakten Syntaxbaum in Code 3.38 wird die Initialisierung eines Verbundes 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',
```

```
StructDecl
5
        Name 'st1',
6
           Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
           → Name('attr'))
8
        ],
9
      StructDecl
10
        Name 'st2',
11
12
           Alloc(Writeable(), IntType('int'), Name('attr1'))
13
           Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
14
        ],
15
      FunDef
16
         VoidType 'void',
17
         Name 'main',
18
         [],
19
20
           Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
           Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
21

    Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
               Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))]))])))
23
    ]
```

Code 3.38: Abstrakter Syntaxbaum für Initialisierung von Verbunden.

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

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

Beim Iterieren über den Teilbaum, muss beim Verbund-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 Feldes oder Verbundes im Teilbaum auf der rechten Seite, der beim jeweiligen obersten Initializer anfängt immer nur 3 Fällte, man hat es auf der rechten Seite entweder mit einem Verbund-Initialiser, einem Feld-Initialiser oder einem Logischen Ausdruck zu tuen. Bei Feld- und Verbund-Initialisier wird einfach über diese nach dem Prinzip der Tiefensuche 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.

```
Name './example_struct_init.picoc_mon',
4
      Block
        Name 'main.0',
6
7
8
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
9
           Assign(Global(Num('0')), Stack(Num('1')))
           // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
10
           → Assign(Name('attr2'), Struct([Assign(Name('attr'), 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: Pico C-ANF Pass für Initialisierung von Verbunden.

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.

```
File
    Name './example_struct_init.reti_blocks',
       Block
         Name 'main.0',
 7
8
           # // Assign(Name('var'), Num('42'))
           # 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([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;
```

```
LOADI IN1 0;
29
           ADD IN1 DS;
           STOREIN SP IN1 1;
30
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 Verbunden.

3.3.4.3 Zugriff auf Verbundsattribut

Der Zugriff auf ein Verbundsattribut (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 Verbundsattribut.

Im Abstrakten Syntaxbaum in Code 3.42 wird der Zugriff auf ein Verbundsattribut st.y mithilfe der Komposition Exp(Attr(Name('st'), Name('y'))) dargestellt.

```
File
    Name './example_struct_attr_access.ast',
 4
5
       StructDecl
         Name 'pos',
6
7
8
9
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
         ],
       FunDef
11
         VoidType 'void',
12
         Name 'main',
13
         [],
14
         [
15
           Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),

    Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
```

```
16 Exp(Attr(Name('st'), Name('y')))
17 ]
18 ]
```

Code 3.42: Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.

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

Die Komposition Exp(Attr(Name('st'), Name('y'))) wird genauso, wie in Subkapitel 3.3.3.2 durch Kompositionen ersetzt, die sich in Anfangsteil 3.3.5.1, Mittelteil 3.3.5.2 und Schlussteil 3.3.5.3 aufteilen lassen. In diesem Fall sind es Ref(Global(Num('0'))) (Anfangsteil), Ref(Attr(Stack(Num('1')), Name('y'))) (Mittelteil) und Exp(Stack(Num('1'))) (Schlussteil). Der Anfangsteil und Schlussteil sind genau gleich, wie in Subkapitel 3.3.3.2.

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

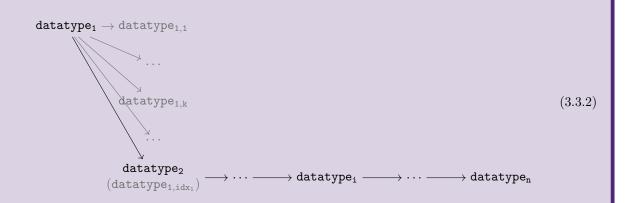
Im Gegensatz zur Komposition Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) beim Zugriff auf einen Feldindex in Subkapitel 3.3.3.2, muss hier vorher nichts anderes als die Anfangsadresse des Verbundes auf dem Stack liegen. Das Verbundsattribut auf welches zugegriffen wird steht bereits in der Komposition Ref(Attr(Stack(Num('1')), Name('y'))), nämlich Name('y'). Den Verbundstyp, dem dieses Attribut gehört, kann man aus dem versteckten Attribut datatype herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) angehängt.

Anmerkung Q

Sei datatype_i ein Knoten eines entarteten Baumes (siehe Definition 3.8 und Abbildung 3.3.2), dessen Wurzel datatype_i ist. Dabei steht i für eine Ebene des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich Startadressen ref(datatype_i) von Speicherbereichen ref(datatype_i) ... ref(datatype_i) + size(datatype_i) im Hauptspeicher zuordnen, wobei gilt, dass ref(datatype_i) \leq ref(datatype_{i+1}) < ref(datatype_i) + size(datatype_i).

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

Sei datatype_{i,idx_i} ein beliebiges Element / Attribut des Datentyps datatype_i, sodass gilt: datatype_{i,idx_i} = datatype_{i+1}.



Die Berechnung der Adresse für eine beliebige Folge verschiedener Datentypen ($\mathtt{datatype_{1,idx_1}}, \ldots, \mathtt{datatype_{n,idx_n}}$), die das Resultat einer Aneinandereihung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente** und **Verbundsattributte** unterschiedlicher Datentypen $\mathtt{datatype_i}$ ist (z.B. *complex_var.attr3[2]), kann mittels der Formel 3.3.3:

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

berechnet werden.

Dabei darf nur der letzte Knoten datatype_n vom Datentyp Zeiger sein. Ist in einer Folge von Datentypen ein Knoten vom Datentyp Zeiger, 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 datatype₁ bis direkt zum Dantentyp Zeiger geht datatype_{pntr} und die zweite Adressberechnung einen Dantentyp nach dem Datentyp Zeiger anfängt datatype_{pntr+1} und bis zum letzten Datenyp datatype_n geht. Bei der zweiten Adressberechnung muss dabei die Adresse ref(datatype₁) des Summanden aus der Formel 3.3.3 auf den Inhalt der Speicherzelle an der gerade in der zweiten Adressberechnung berechneten Adresse M [ref(datatype₁, ..., datatype_{pntr})] gesetzt werden.

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

Die Komposition Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentiert dabei den Summanden ref(datatype₁) in der Formel.

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

Die Komposition Exp(Stack(Num('1'))) repräsentiert dabei das Lesen des Inhalts $M[\text{ref}(\text{datatype}_{1,idx_1}, \ldots, \text{datatype}_{n,idx_n})]$ der Speicherzelle an der finalen Adresse $\text{ref}(\text{datatype}_{1,idx_1}, \ldots, \text{datatype}_{n,idx_n})$.

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

^bref (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 Zeigerelmente, Feldelemente oder Verbundsattribute resultiert. Die innere Schleife iteriert über alle Elemente oder Attribute des momentan betrachteten Datentyps datatype_i, die vor dem Element / Attribut datatype_{i,idx_i} liegen.

^dVerbundsattribute haben unterschiedliche Größen.

Definition 3.8: Entarteter Baum

Z

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

Oder alternativ: Baum beim dem jeder Knoten des Baumes maximal eine eingehende Kante hat, also maximal Innengrad^a 1.

Der Baum entspricht also einer verketteten Liste.^b

^aDer Innengrad ist die Anzahl eingehener Kanten.

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

Code 3.43: PicoC-ANF Pass für Zugriff auf Verbundsattribut.

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.

 $^{{}^{}b}B\ddot{a}ume.$

```
LOADI ACC 4;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('2'))
13
           SUBI SP 1;
14
           LOADI ACC 2;
15
           STOREIN SP ACC 1;
16
           # Assign(Global(Num('0')), Stack(Num('2')))
17
           LOADIN SP ACC 1;
18
           STOREIN DS ACC 1;
19
           LOADIN SP ACC 2;
20
           STOREIN DS ACC 0;
21
           ADDI SP 2;
           # // Exp(Attr(Name('st'), Name('y')))
22
23
           # Ref(Global(Num('0')))
24
           SUBI SP 1;
25
           LOADI IN1 0;
26
           ADD IN1 DS;
27
           STOREIN SP IN1 1;
28
           # Ref(Attr(Stack(Num('1')), Name('y')))
29
           LOADIN SP IN1 1;
30
           ADDI IN1 1;
31
           STOREIN SP IN1 1;
32
           # Exp(Stack(Num('1')))
33
           LOADIN SP IN1 1;
34
           LOADIN IN1 ACC O;
35
           STOREIN SP ACC 1;
36
           # Return(Empty())
37
           LOADIN BAF PC -1;
38
         ]
39
    ]
```

Code 3.44: RETI-Blocks Pass für Zugriff auf Verbundsattribut.

3.3.4.4 Zuweisung an Verbundsattribut

Die **Zuweisung an ein Verbundsattribut** (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 Verbundsattribut.

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

```
Name './example_struct_attr_assignment.ast',
4
      StructDecl
        Name 'pos',
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
9
        ],
10
      FunDef
11
         VoidType 'void',
12
        Name 'main',
13
         [],
14
15
           Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),

    Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
           Assign(Attr(Name('st'), Name('y')), Num('42'))
16
17
18
    ]
```

Code 3.46: Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.

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 Feldelement 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 ansonsnten 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 Knoten Attr(Name('st'), Name('y')) von Assign(Attr(Name('st'), Name('y')), Num('42')) wie in Subkapitel 3.3.4.3 vorgegangen werden.

```
Name './example_struct_attr_assignment.picoc_mon',
    Γ
      Block
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
9
           Exp(Num('2'))
10
           Assign(Global(Num('0')), Stack(Num('2')))
11
           // Assign(Attr(Name('st'), Name('y')), Num('42'))
12
           Exp(Num('42'))
13
           Ref(Global(Num('0')))
14
           Ref(Attr(Stack(Num('1')), Name('y')))
           Assign(Stack(Num('1')), Stack(Num('2')))
16
           Return(Empty())
17
        ]
18
    ]
```

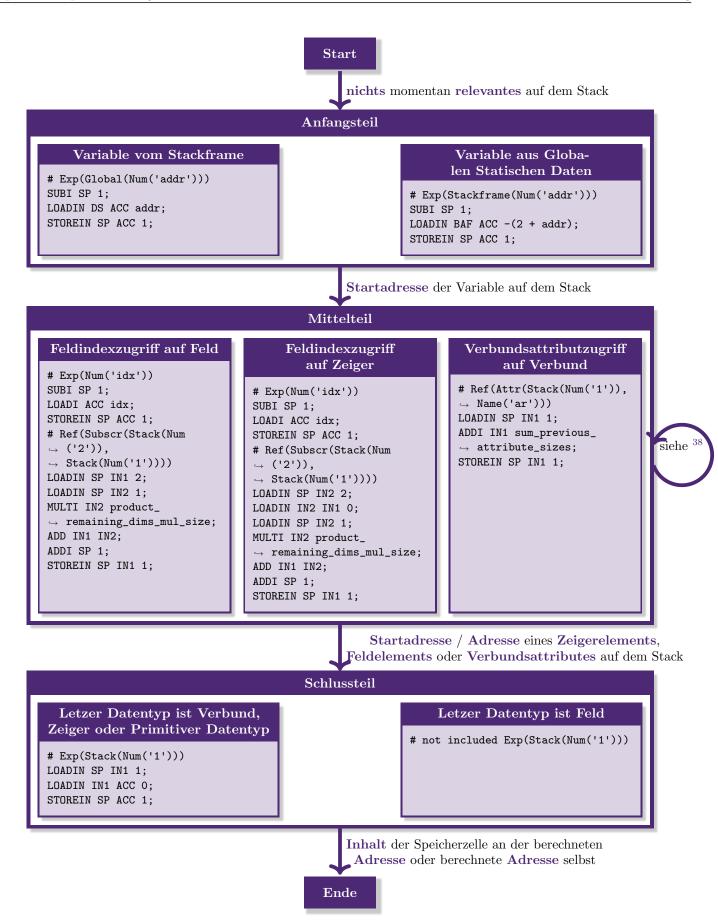
Code 3.47: PicoC-ANF Pass für Zuweisung an Verbundsattribut.

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
     Name './example_struct_attr_assignment.reti_blocks',
       Block
 5
         Name 'main.0',
 6
 7
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           # Exp(Num('4'))
 9
           SUBI SP 1;
10
           LOADI ACC 4;
11
           STOREIN SP ACC 1;
12
           # Exp(Num('2'))
13
           SUBI SP 1;
14
           LOADI ACC 2;
15
           STOREIN SP ACC 1;
16
           # Assign(Global(Num('0')), Stack(Num('2')))
17
           LOADIN SP ACC 1;
18
           STOREIN DS ACC 1:
19
           LOADIN SP ACC 2;
20
           STOREIN DS ACC 0;
21
           ADDI SP 2;
22
           # // 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
         ]
    ]
```

Code 3.48: RETI-Blocks Pass für Zuweisung an Verbndsattribut.

3.3.5 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen



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 (Zeiger, Feld und Verbund) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem Anfangsteil, Mittelteil und Schlussteil 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 Datenypen Zeiger, Feld und Verbund 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)-Knotens ein ArrayDecl(nums, datatype), ein PntrDecl(num, datatype) oder StructSpec(name) beinhaltet und die dazu passende Zugriffsoperation Subscr(exp1, exp2) oder Attr(exp, name) vorliegt, einen anderen RETI-Code generiert wird. Dieser RETI-Code berechet die Startadresse eines gewünschten Zeigerelements, Feldelements oder Verbundsattributs.

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

Die Zugriffsoperation muss dabei zum Datentyp im versteckten Attribut datatype passen, ansonsten gibt es eine DatatypeMismatch-Fehlermeldung. Ein Zugriff auf ein Feldindex Subscr(exp1, epp2) kann dabei mit den Datentypen Feld ArrayDecl(nums, datatype) und Zeiger PntrDecl(num, datatype) kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen RETI-Code. Das liegt daran, dass bei einem Zeiger 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 Verbundsattribut Attr(exp, name) kann nur mit dem Datentyp Struct StructSpec(name) kombiniert werden.

Anmerkung Q

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 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 Feldelement jeweils gegenseitig austauschbar sind. Der Zugriff auf einen Feldindex steht also gleichermaßen auch für eine Dereferenzierung.

Das versteckte Attribut datatype beinhaltet den Unterdatentyp, in welchem der Zugriff auf ein Zeigerelment, Feldelement oder Verbundsattribut 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 Knoten in diesem Unterdatentyp interessiert und die möglicherweise unter diesem momentan betrachteten Knoten liegenden Knoten in einem anderen Ref(exp, versteckte Attribut)-Knoten, dem jeweiligen versteckten Attribut datatype zugeordnet sind. Das versteckte Attribut datatype enthält also die Information auf welchen Unterdatentyp im dem momentanen Kontext gerade zugegriffen wird.

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

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

gewünschte Ergebnis dar.

Felder haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes Feld 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.

Anmerkung Q

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')))-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.

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 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 3.3.5.1, Mittelteil 3.3.5.2 und Schlussteil 3.3.5.3 bei der Kompilierung von Zugriffen auf Zeigerelemente, Feldelemente, Verbundsattribute bei gemischten Ausdrücken, wie (*st_first.ar)[0]; einzeln isoliert betrachtet und erläutert.

3.3.5.1 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() {
```

³⁹Und nicht auf ein Element des Feldes.

³⁹Startadresse / Adresse eines Zeigerelements, Feldelements oder Verbundsattributes auf dem Stack.

```
4    struct ar_with_len st_ar[3];
5    int *(*complex_var)[3];
6    &complex_var;
7 }
8    y 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 Abstrakten Syntaxbaum in Code 3.50 wird die Refererenzierung &complex_var mit der Komposition Exp(Ref(Name('complex_var'))) dargestellt. Üblicherweise wird aber einfach nur Ref(Name('complex_var') geschrieben, aber da beim Erstellen des Abstract Syntx Tree jeder Logischer Ausdruck in ein Exp(exp) eingebettet wird, ist das Ref(Name('complex_var')) in ein Exp() eingebettet. Man müsste an vielen Stellen eine gesonderte Fallunterschiedung aufstellen, um von Exp(Ref(Name('complex_var'))) das Exp() zu entfernen, obwohl das Exp() in den darauffolgenden Passes so oder so herausgefiltet wird. Daher wurde darauf verzichtet den Code ohne triftigen Grund komplexer zu machen.

```
File
1
    Name './example_derived_dts_introduction_part.ast',
     Γ
      StructDecl
        Name 'ar_with_len',
          Alloc(Writeable(), IntType('int'), Name('len'))
          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9
        ],
10
      FunDef
11
         VoidType 'void',
12
        Name 'main',
13
         [],
14
15
          Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           → Name('st_ar')))
          Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
16
           → IntType('int'))), Name('complex_var')))
17
          Exp(Ref(Name('complex_var')))
18
         ],
19
      FunDef
20
         VoidType 'void',
21
        Name 'fun',
22
         [],
23
24
          Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           → Name('st_ar')))
          Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
25

→ Name('complex_var')))
          Exp(Ref(Name('complex_var')))
26
         ]
27
    ]
```

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
    Name './example_derived_dts_introduction_part.picoc_mon',
     Ε
       Block
         Name 'main.1',
7
8
9
           // Exp(Ref(Name('complex_var')))
           Ref(Global(Num('9')))
           Return(Empty())
10
         ],
       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
 2
    Name './example_derived_dts_introduction_part.reti_blocks',
     Γ
       Block
         Name 'main.1',
 6
7
8
9
           # // Exp(Ref(Name('complex_var')))
           # Ref(Global(Num('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.

3.3.5.2 Mittelteil

Der Mittelteil, bei dem die Startadresse / Adresse einer Aneinandereihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute 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 Abstrakten Syntaxbaum in Code 3.54 wird die Aneinandererihung von Zugriffen auf Zeigerelemente, Feldelemente und Verbundsattribute (*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',
4
      StructDecl
        Name 'st',
6
           Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
           → Name('ar'))
        ],
8
9
      FunDef
10
        VoidType 'void',
11
        Name 'main',
12
         [],
13
         Ε
14
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
           → Array([Num('42')]))
           Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),

    Struct([Assign(Name('ar'), Ref(Name('var')))]))

16
           Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),

    Sub('-'), Num('2'))))
```

```
17 ]
18 ]
```

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')))) und Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) und Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) ersetzt. Bei Subscr(exp1, exp2) wird dieser Knoten einfach dem exp Attribut des Ref(exp)-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')))).

```
2
3
    Name './example_derived_dts_main_part.picoc_mon',
     Γ
 4
5
6
7
8
       Block
         Name 'main.0',
           // Assign(Name('var'), Array([Num('42')]))
           Exp(Num('42'))
           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')),
               BinOp(Num('2'), Sub('-'), Num('2'))))
           Ref(Global(Num('1')))
15
           Ref(Attr(Stack(Num('1')), Name('ar')))
16
           Exp(Num('0'))
17
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18
           Exp(Num('2'))
19
           Exp(Num('2'))
20
           Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
21
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22
           Exp(Stack(Num('1')))
23
           Return(Empty())
24
         ]
25
    ]
```

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(Num('2')), 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)-Knoten berücksichtigt werden, was in Unterkapitel 3.3.5 zusammen mit der Abbildung 3.9 bereits erklärt wurde.

```
Name './example_derived_dts_main_part.reti_blocks',
     Γ
 4
       Block
         Name 'main.0',
           # // Assign(Name('var'), Array([Num('42')]))
           # 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'))))
           # Ref(Global(Num('1')))
27
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'))))
```

```
LOADIN SP ACC 2;
58
           LOADIN SP IN2 1;
           SUB ACC IN2;
59
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())
           LOADIN BAF PC -1;
75
         1
    ]
```

Code 3.56: RETI-Blocks Pass für den Mittelteil.

3.3.5.3 Schlussteil

Der Schlussteil, bei dem der Inhalt der Speicherzelle an der Adresse, die im Anfangsteil 3.3.5.1 und Mittelteil 3.3.5.2 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 Abstrakten Syntaxbaumes in Code 3.58 verläuft wie üblich.

```
1 File
2 Name './example_derived_dts_final_part.ast',
```

 $^{^{40}}$ Und dabei die Speicherzelle der Adresse selbst überschreibt.

```
4
      StructDecl
5
        Name 'st',
7
8
          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
        ],
9
      FunDef
10
         VoidType 'void',
11
        Name 'main',
12
         [],
13
14
          Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),

→ Name('complex_var1')))
15
          Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),

→ Name('complex_var2')))
          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
16
17
          Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),

→ Ref(Name('var')))
18
          Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),

→ Ref(Name('pntr1')))
19
          Exp(Subscr(Name('complex_var1'), Num('0')))
          Exp(Subscr(Name('complex_var2'), Num('0')))
20
          Exp(Deref(Name('complex_var3'), Num('0')))
22
    ]
```

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.

```
2
    Name './example_derived_dts_final_part.picoc_mon',
     Γ
 4
       Block
         Name 'main.0',
 7
8
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
           Assign(Global(Num('4')), Stack(Num('1')))
10
           // Assign(Name('pntr1'), Ref(Name('var')))
11
           Ref(Global(Num('4')))
12
           Assign(Global(Num('5')), Stack(Num('1')))
13
           // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14
           Ref(Global(Num('5')))
15
           Assign(Global(Num('6')), Stack(Num('1')))
16
           // Exp(Subscr(Name('complex_var1'), Num('0')))
17
           Ref(Global(Num('0')))
18
           Exp(Num('0'))
19
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20
           Exp(Stack(Num('1')))
21
           // Exp(Subscr(Name('complex_var2'), Num('0')))
22
           Ref(Global(Num('2')))
           Exp(Num('0'))
```

```
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,datatpye)-Knoten kein Feld ArrayDecl(nums, datatype) enthält, ansonsten ist bei einem Feld die Adresse auf dem Stack bereits das gewünschte Ergebnis. Genaueres wurde in Unterkapitel 3.3.5 zusammen mit der Abbildung 3.9 bereits erklärt.

```
1 File
    Name './example_derived_dts_final_part.reti_blocks',
     Γ
       Block
         Name 'main.0',
           # // Assign(Name('var'), Num('42'))
           # Exp(Num('42'))
           SUBI SP 1:
10
           LOADI ACC 42;
           STOREIN SP ACC 1;
11
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;
           LOADI IN1 5;
29
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;
           ADDI SP 1;
```

```
# // Exp(Subscr(Name('complex_var1'), Num('0')))
           # Ref(Global(Num('0')))
37
38
           SUBI SP 1;
39
           LOADI IN1 0;
           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;
           LOADI ACC 0;
62
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;
           LOADIN IN1 ACC O;
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;
           # Exp(Num('0'))
81
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;
           STOREIN SP IN1 1;
```

Code 3.60: RETI-Blocks Pass für den Schlussteil.

3.3.6 Umsetzung von Funktionen

Um die Umsetzung von Funktionen zu verstehen, ist es erstmal wichtig zu verstehen, wie Funktionen später im RETI-Code aussehen (Unterkapitel 3.3.6.1), wie Funktionen deklariert (Definition 1.7) und definiert (Definition 1.8) werden können und hierbei Sichtbarkeitsbereiche (Definition 1.9) umgesetzt sind (Unterkapitel 3.3.6.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines Funktionsaufrufes erklärt werden (Unterkapitel 3.3.6.3). Beim Thema Funktionsaufruf muss im speziellen nochmal darauf eingegangen werden, wie Rückgabewerte (Unterkapitel 3.3.6.3.1) umgesetzt sind und die Übergabe von Zusammengesetzten Datentypen, die mehr als eine Speicherzelle belegen, wie Verbunden (Unterkapitel 3.3.6.3.3) und Feldern (Unterkapitel 3.3.6.3.2) umgesetzt ist.

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 Abstrakten Syntaxbaum in Code 3.62 werden die 3 Funktionen durch entsprechende Knoten dargestellt. Am Beispiel der Funktion void fun2() {return 1;} wäre der hierzu passende Knoten FunDef(VoidType(), Name('fun2'), [], [Return(Num('1'))]). Die einzelnen Attribute dieses FunDef(datatype, name, allocs, stmts_blocks)-Knoten sind in Tabelle 3.6 erklärt.

```
File
 2
     Name './verbose_3_funs.ast',
       FunDef
         VoidType 'void',
         Name 'main',
 7
8
          [],
          Γ
 9
           Return
10
              Empty
11
         ],
       FunDef
13
         VoidType 'void',
14
         Name 'fun1',
         [],
          [],
```

```
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 Anweisungen der Funktion in Blöcke Block(name, stmts_instrs) aufgeteilt. Dabei bekommt ein Block Block(name, stmts_instrs), der die Anweisungen 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)-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 Assoziative Feld fun_name_to_block_name ordnet einem Funktionsnamen den Blocknamen des Blockes, der die erste Anweisung der Funktion enthält zu. Der Bezeichner des Blockes <name>.<nummer> ist dabei bis auf die angehängte Nummer identisch zu dem der Funktion. Diese Zuordnung ist nötig, da Blöcke eben noch eine Nummer an ihren Bezeichner <name>.<nummer> angehängt haben.

```
1
   File
 2
     Name './verbose_3_funs.picoc_blocks',
     Γ
       {\tt FunDef}
          VoidType 'void',
          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',
```

 $^{^{41}\}mathrm{Eine}$ Erklärung dazu ist in Unterkapitel3.3.1.2.1zu finden.

 $^{^{42}\}mathrm{Der}$ Grund dafür kann im Unterkapitel3.3.1.2.1nachgelesen werden.

```
27 [],
28 [
29 Block
30 Name 'fun2.0',
31 [
32 Return(Num('1'))
33 ]
34 ]
35 ]
```

Code 3.63: PicoC-Blocks Pass für 3 Funktionen.

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

```
File
     Name './verbose_3_funs.picoc_mon',
     Γ
       Block
         Name 'main.2',
           Return(Empty())
         ],
 9
       Block
10
         Name 'fun1.1',
11
12
           Return(Empty())
13
         ],
14
       Block
         Name 'fun2.0',
16
17
           // Return(Num('1'))
18
           Exp(Num('1'))
           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. Die Sprünge werden nämlich dorthin gemacht, wo die RETI-Befehle, die aus den Anweisungen einer Funktion kompiliert wurden anfangen.

```
# // 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 [
```

```
Block
         Name 'fun1.2',
           # Return(Empty())
 8
           LOADIN BAF PC -1;
 9
         ],
10
       Block
         Name 'fun2.1',
11
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
         ٦.
23
       Block
         Name 'main.0',
24
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)-Knoten muss dazu im decls_defs-Attribut, welches eine Liste von Funktionen ist, die main-Funktion an Index 0 geschoben werden.

Die Möglichkeit für die sich in der Implementierung des PicoC-Compilers entschieden wurde, ist es, wenn die main-Funktion nicht die erste auftauchende Funktion ist, einen start.<nummer>-Block als ersten Block einzufügen. Dieser start.<nummer>-Block enthält einen GoTo(Name('main.<nummer>'))-Knoten, 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 Verwender⁴³ des PicoC-Compilers vermutlich am intuitivsten ist, wenn der RETI-Code für die Funktionen an denselben Stellen relativ zueinander verortet ist, wie die Funktionsdefinitionen im PicoC-Code.

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

```
File
Name './verbose_3_funs_main.reti_patch',

Block
Name 'start.3',
```

⁴³Also die kommenden Studentengenerationen.

```
# // Exp(GoTo(Name('main.0')))
           Exp(GoTo(Name('main.0')))
 9
         ],
10
       Block
11
         Name 'fun1.2',
12
13
           # Return(Empty())
14
           LOADIN BAF PC -1;
         ],
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.<number>')) durch den entsprechenden Sprung JUMP <distance_to_main_function> ersetzt und es werden 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 Sichtbarkeitsbereichen

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 1.6) der Funktion nicht mit den Datentypen der Argumente oder der Anzahl Argumente übereinstimmt, die beim Funktionsaufruf an die Funktion in einer festen Reihenfolge übergeben werden.

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

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

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

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

Die Variablen var@main und var@fun2 der main-Funktion und der Funktion fun2 haben unterschiedliche Sichtbarkeitsbereiche (Definition 1.9). Die Sichtbarkeitsbereiche der Funktionen werden mittels eines Suffix "@<fun_name>" umgesetzt, der an den Bezeichner var angehängt wird: var@<fun_name>. Dieser Suffix wird geändert, sobald beim top-down⁴⁴ iterieren über den Abstrakten Syntaxbaum des aktuellen Passes eine neuer FunDef(datatype, name, allocs, stmts_blocks)-Knoten betreten wird und über die Anweisungen im stmts-Attribut iteriert wird. Für das Iterieren über die Anweisungen dieses neuen Funktionsknotens

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

wird der Suffix der neuen Funktion FunDef(name, datatype, params, stmts_blocks) angehängt, der aus dem name-Attribut entnommen wird.

Ein Grund, warum Sichtbarkeitsbereiche 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 einer Variable direkt ihr Sichtbarkeitsbereich in dem sie definiert wurde abgelesen werden kann, ist der Suffix ebenfalls im Name(str)-Knoten des name-Attribubtes eines Symbols der Symboltabelle angehängt. Dies ist in Code 3.71 markiert.

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

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

```
32
           type qualifier:
                                     Writeable()
33
           datatype:
                                     IntType('int')
34
           name:
                                     Name('var@main')
35
                                     Num('0')
           value or address:
36
           position:
                                     Pos(Num('7'), Num('6'))
37
           size:
                                     Num('1')
38
         },
39
       Symbol
40
           type qualifier:
41
                                     Writeable()
42
                                     IntType('int')
           datatype:
43
                                     Name('var@fun2')
           name:
44
                                     Num('0')
           value or address:
45
                                     Pos(Num('12'), Num('13'))
           position:
46
           size:
                                     Num('1')
47
         }
```

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. Zudem wurde, um die Adressberechnung anschaulicher zu machen als Datentyp für den Parameter param der Funktion stack_fun ein Verbund gewählt, der mehrere Speicherzellen im Hauptspeicher einnimmt. 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: Pico C-Code für Funktionsaufruf ohne Rückgabewert.

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

```
Name './example_fun_call_no_return_value.ast',
 4
       StructDecl
         Name 'st',
           Alloc(Writeable(), IntType('int'), Name('attr1'))
           Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
 9
         ],
10
       FunDecl
11
         VoidType 'void',
12
         Name 'stack_fun',
13
         Γ
14
           Alloc
15
             Writeable,
16
             ArrayDecl
17
               Γ
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(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
           → Name('local_var')))
           Exp(Call(Name('stack_fun'), [Name('local_var')]))
31
32
           Return(Empty())
33
         ],
34
       {\tt FunDef}
35
         VoidType 'void',
36
         Name 'stack_fun',
37
         Γ
           Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
38
           → Name('param'))
39
         ],
40
         Γ
41
           Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
42
     ]
```

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

Code 3.74: Pico C-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 Kompostionen LOADI ACC GoTo(Name('addr@next_instr')) und Exp(GoTo(Name('stack_fun.0'))) noch keine fertigen RETI-Befehlen 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_name⁴⁶ mit dem Schlüssel stack_fun⁴⁷, der im Knoten NewStackframe(Name('stack_fun')) gespeichert ist, nachgeschlagen.

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

 $[\]overline{^{45}}$ Wobei hier "reserviert" besser passen würde.

⁴⁶Dieses Assoziative Feld wurde in Unterkapitel 3.3.6.1 eingeführt.

⁴⁷Dem Bezeichner der Funktion.

```
# 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 O;
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
     ]
```

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 ist⁴⁸ ersetzt: LOADI ACC 14. Und auch der Knoten, der den Sprung Exp(GoTo(Name('stack_fun.0'))) darstellt wird durch den Knoten JUMP 5 ersetzt.

Die Distanz 5 im RETI-Knoten JUMP 5 wird mithilfe des instrs_before-Attributs des Zielblocks Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)⁴⁹ 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 Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size) berechnet. Es handelt sich bei bei 14 um eine relative Adresse, die relativ zum CS-Register⁵⁰ berechnet wird.

Anmerkung 9

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

⁴⁸Also der Befehl, der bisher durch die Komposition Exp(GoTo(Name('stack_fun.0'))) dargestellt wurde.

⁴⁹Welcher den ersten Befehl der gewünschten Funktion enthält.

⁵⁰Welches im RETI-Interpreter von einem Startprogramm im EPROM immer so gesetzt wird, dass es die Adresse enthält, an der das Codesegment anfängt.

folgenden Formel:

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

wobei:

- es sich bei bei adr_{danach} um eine relative Adresse handelt, die relativ zum CS-Register berechnet wird.
- #Bef_{vor akt. Bl.} 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 = relativer Index des Befehls LOADI ACC <addr@next_instr> selbst im Block.
- 4 = Distanz, die zwischen den in Code 3.76 markierten Befehlen LOADI ACC <im> und JUMP <im> liegt und noch eins mehr, weil man ja zum nächsten Befehl will.

Die Berechnug der Distanz distanz für den Sprung JUMP distanz zum ersten Befehl eines im vorhergehenden Pass existenten Blockes erfolgt dabei nach der folgenden Formel:

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

wobei:

- #Bef_{vor Zielbl.} 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.

```
# // 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;
```

^aIm **RETI-Pass** gibt es keine Blöcke mehr.

```
LOADI ACC 14;

ADD ACC CS;

Exp(GoTo(Name('stack_fun.0')))

JUMP 5;

# RemoveStackframe()

MOVE BAF IN1;

LOADIN IN1 BAF 0;

MOVE IN1 SP;

# Return(Empty())

LOADIN BAF PC -1;

# Return(Empty())

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_valu e()) 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.

```
int fun_with_return_value() {
   return 21 * 2;
}

void fun_no_return_value() {
   return;
}

void main() {
   int var = fun_with_return_value();
   fun_no_return_value();
}
```

Code 3.77: PicoC-Code für Funktionsaufruf mit Rückgabewert.

Im Abstrakten Syntaxbaum in Code 3.78 wird eine Return-Anweisung mit Rückgabewert return 21 * 2 mit der Komposition Return(BinOp(Num('21'), Mul('*'), Num('2'))) dargestellt, eine Return-Anweisung ohne Rückgabewert return mit der Komposition Return(Empty()) und ein Funktionsaufruf inklusive Zuweisung des Rückgabewertes int var = fun_with_return_value() durch die Komposition Assign(Alloc (Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[])).

```
1 File
2 Name './example_fun_call_with_return_value.ast',
3 [
4 FunDef
```

```
IntType 'int',
        Name 'fun_with_return_value',
        [],
8
        Γ
          Return(BinOp(Num('21'), Mul('*'), Num('2')))
10
        ],
11
      FunDef
12
        VoidType 'void',
13
        Name 'fun_no_return_value',
14
15
        16
          Return(Empty())
17
        ],
18
      FunDef
19
        VoidType 'void',
20
        Name 'main',
21
        [],
22
23
          Assign(Alloc(Writeable(), IntType('int'), Name('var')),
          Exp(Call(Name('fun_no_return_value'), []))
24
25
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. Als letztes wird die Rücksprungadresse in das PC-Register geladen, die durch den NewStackframe()-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)⁵¹ definiert werden.

Dieser Trick mit dem Speichern des Rückgabewertes im ACC-Register ist notwendidg, weil durch das Entfernen des Stackframes der aufgerufenen Funktion das SP-Register nicht mehr an der gleichen Stelle steht. Daher sind alle temporären Werte, die in der aufgerufenen Funktion auf den Stack geschrieben wurden unzugänglich, weil man nicht wissen kann, um wieviel die Adresse im SP-Register verglichen zu vorher verschoben ist, weil der Stackframe von unterschiedlichen aufgerufenen Funktionen unterschiedlich groß sein kann.

Die Komposition Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[])) wird nach dem allokieren der Variable Name('var') durch die Komposition Assign(Global(Num ('0')),Stack(Num('1'))) ersetzt, welche den Rückgabewert der Funktion Name('fun_with_return_value'), welcher durch die Komposition Exp(Acc) aus dem ACC-Register auf den Stack geschrieben wurde nun vom

⁵¹Diese Komposition schreibt den aktuellen Wert des Registers ACC auf den Stack

Stack in die Speicherzelle der Variable Name('var') in den Globalen Statischen Daten 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, sie stellt nur das Laden der Rücksprungsadresse in das PC-Register dar.

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem Rückgabedatentyp void keine return-Anweisung explizit ans Ende geschrieben wird, im PicoC-ANF Pass eines hinzufügt wird in Form der Komposition Return(Empty()). Beim Nicht-Angeben im Falle eines Dantentyps, der nicht void ist, wird allerdings eine MissingReturn-Fehlermeldung ausgelöst.

```
File
 2
    Name './example_fun_call_with_return_value.picoc_mon',
 4
       Block
 5
         Name 'fun_with_return_value.2',
           // Return(BinOp(Num('21'), Mul('*'), Num('2')))
           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
         Γ
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')))
           Exp(GoTo(Name('fun_no_return_value.1')))
31
           RemoveStackframe()
32
           Return(Empty())
         ]
33
```

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.

```
Name './example_fun_call_with_return_value.reti_blocks',
     Γ
 4
       Block
         Name 'fun_with_return_value.2',
           # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
           # 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'))))
           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',
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 O;
53
           MOVE IN1 SP;
54
           # Exp(ACC)
           SUBI SP 1;
56
           STOREIN SP ACC 1;
           # Assign(Global(Num('0')), Stack(Num('1')))
```

```
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
```

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

3.3.6.3.2 Umsetzung der Übergabe eines Feldes

Die Eigenheit, dass bei der Übergabe eines Feldes an eine andere Funktion, dieses als Zeiger übergeben wird, wurde bereits im Unterkapitel 1.3 erläutert. Die Umsetzung der Übergabe eines Feldes an eine andere Funktion wird im Folgenden mithilfe des Beispiels in Code 3.81 erklärt.

```
void fun_array_from_stackframe(int (*param)[3]) {

void fun_array_from_global_data(int param[2][3]) {
    int local_var[2][3];
    fun_array_from_stackframe(local_var);
}

void main() {
    int local_var[2][3];
    fun_array_from_global_data(local_var);
}
```

Code 3.81: PicoC-Code für die Übergabe eines Feldes.

Im PicoC-ANF Pass muss im Fall dessen, dass der oberste Knoten im Teilbaum, der den Datentyp darstellt und an die Funktion übergeben wird ein Feld ArrayDecl(nums, datatype) ist auf spezielle Weise vorgegangen werden. Dieser oberste Knoten des Teilbaums, der den Datentyp darstellt muss zu einem

Zeiger PntrDecl(num, datatype) umgewandelt und der Rest des Teilbaumes, der am datatype-Attribut hängt, muss an das datatype-Attribut des Zeigers PntrDecl(num, datatype) gehängt werden.

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 Zeiger nur eine Speicherzelle braucht.

```
SymbolTable
     Γ
       Symbol
 4
         {
 5
           type qualifier:
                                    Empty()
 6
                                    FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
           datatype:
               [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
           → Name('param'))])
           name:
                                    Name('fun_array_from_stackframe')
           value or address:
                                    Empty()
 9
           position:
                                    Pos(Num('1'), Num('5'))
10
                                    Empty()
           size:
11
         },
12
       Symbol
13
14
                                    Writeable()
           type qualifier:
15
                                    PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
           datatype:
16
                                    Name('param@fun_array_from_stackframe')
           name:
17
                                    Num('0')
           value or address:
18
                                    Pos(Num('1'), Num('37'))
           position:
19
           size:
                                    Num('1')
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                    Empty()
24
                                    FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
           datatype:
                                    ArrayDecl([Num('3')], IntType('int')), Name('param'))])
               [Alloc(Writeable(),
25
                                    Name('fun_array_from_global_data')
26
           value or address:
                                    Empty()
27
                                    Pos(Num('4'), Num('5'))
           position:
28
           size:
                                    Empty()
29
         },
30
       Symbol
31
         {
32
           type qualifier:
                                    Writeable()
33
                                    PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
           datatype:
34
                                    Name('param@fun_array_from_global_data')
           name:
35
                                    Num('0')
           value or address:
36
                                    Pos(Num('4'), Num('36'))
           position:
37
                                    Num('1')
           size:
38
         },
39
       Symbol
40
         {
41
           type qualifier:
                                    Writeable()
42
                                    ArrayDecl([Num('2'), Num('3')], IntType('int'))
           datatype:
43
                                    Name('local_var@fun_array_from_global_data')
           name:
           value or address:
                                    Num('6')
```

```
position:
                                     Pos(Num('5'), Num('6'))
46
           size:
                                     Num('6')
47
         },
48
       Symbol
49
         {
           type qualifier:
50
                                     Empty()
51
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
52
                                     Name('main')
           name:
53
                                     Empty()
           value or address:
54
                                     Pos(Num('9'), Num('5'))
           position:
55
                                     Empty()
           size:
56
         },
57
       Symbol
58
         {
59
                                     Writeable()
           type qualifier:
60
                                     ArrayDecl([Num('2'), Num('3')], IntType('int'))
           datatype:
61
           name:
                                     Name('local_var@main')
62
                                     Num('0')
           value or address:
63
                                     Pos(Num('10'), Num('6'))
           position:
64
                                     Num('6')
           size:
65
         }
66
     ]
```

Code 3.82: Symboltabelle für die Übergabe eines Feldes.

Im PicoC-ANF Pass in Code 3.83 ist zu sehen, dass zur Übergabe der beiden Felder die Adresse des jeweiligen Feldes auf den Stack geschrieben wird. Die Adressen der beiden Felder 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 die Variable local_var aus der main-Funktion, da diese in den Globalen Statischen Daten liegt und die Komposition Ref(Stackframe(Num('6'))) ist für die Variable local_var aus der Funktion fun_array_from_global_data, da diese auf dem Stackframe dieser Funktion liegt. Dabei stellen die Zahlen in den 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
    Name './example_fun_call_by_sharing_array.picoc_mon',
 4
       Block
         Name 'fun_array_from_stackframe.2',
 6
7
8
9
           Return(Empty())
         ],
       Block
10
         Name 'fun_array_from_global_data.1',
           StackMalloc(Num('2'))
           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()
           Return(Empty())
         ],
```

```
19
       Block
20
         Name 'main.0',
21
22
           StackMalloc(Num('2'))
           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 die Übergabe eines Feldes.

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
     Name './example_fun_call_by_sharing_array.reti_blocks',
 4
       Block
         Name 'fun_array_from_stackframe.2',
           # Return(Empty())
           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;
         ],
```

```
Block
39
         Name 'main.0',
40
           # StackMalloc(Num('2'))
           SUBI SP 2;
           # 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 die Übergabe eines Feldes.

3.3.6.3.3 Umsetzung einer Übergabe eines Verbundes

Die Eigenheit, dass ein Verbund als Argument beim Funktionsaufruf einer anderen Funktion in den Stackframe der aufgerufenen Funktion kopiert wird, wurde bereits im Unterkapitel 1.3 erläutert. Die Umsetzung der Übergabe eines Verbundes wird im Folgenden mithilfe des Beispiels in Code 3.85 erklärt.

```
struct st {int attr1; int attr2[2];};

void fun_struct_from_stackframe(struct st param) {

void fun_struct_from_global_data(struct st param) {

fun_struct_from_stackframe(param);

}

void main() {

struct st local_var;

fun_struct_from_global_data(local_var);
}
```

Code 3.85: PicoC-Code für die Übergabe eines Verbundes.

Im PicoC-ANF Pass in Code 3.86 wird zur Übergabe eines Verbundes, der komplette Verbund 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 einen gesamten Verbund anders gehandhabt als sonst. Normalerweise wird beim Zugriff auf einen Verbund die Adresse des ersten Attributs dieses Verbundes auf den Stack geschrieben. Bei der Übergabe an eine Funktion wird dagegen der gesamte Verbund 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. die 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')), Global(Num('0'))) wird verwendet, da die Verbundsvariable local_var der main-Funktion in den Globalen Statischen Daten liegt und die Komposition Assign(Stack(Num('3')), Stackframe(Num('2'))) wird verwendet, da die Verbundsvariable local_var der Funktion fun_struct_from_global_data im Stackframe liegt.

```
File
 2
    Name './example_fun_call_by_value_struct.picoc_mon',
 4
       Block
 5
         Name 'fun_struct_from_stackframe.2',
 6
7
8
9
           Return(Empty())
         ],
       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',
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
         ]
    ]
```

Code 3.86: PicoC-ANF Pass für die Übergabe eines Verbundes.

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.

```
Name './example_fun_call_by_value_struct.reti_blocks',
 4
       Block
         {\tt Name 'fun\_struct\_from\_stackframe.2',}\\
           # Return(Empty())
           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 O;
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;
           LOADIN DS ACC 1;
```

```
STOREIN SP ACC 2;
           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 O;
68
           MOVE IN1 SP;
69
           # Return(Empty())
70
           LOADIN BAF PC -1;
71
    ]
```

Code 3.87: RETI-Block Pass für die Übergabe eines Verbundes.

3.4 Fehlermeldungen

Die Fehlerarten, die der PicoC-Compiler ausgeben kann sind in den Tabellen 3.9, 3.10 und 3.11 und eingeteilt nach den Kategorien "Fehlerarten in der Lexikalischen und Syntaktischen Analyse", "Fehlerarten in den Passes", "Fehlerarten, die zur Laufzeit auftreten" aus Unterkapitel 2.6. Da der PicoC-Compiler nicht in der Lage ist mehrere Dateien zu kompilieren und somit keinen Linker nötig hat, mussten Fehler, die normalerweise beim Linken aufgefunden werden würden in den Passes umgesetzt werden und sind somit der Kategorie "Fehlerarten in den Passes" zuzuordnen.

Fehlerarten, wie z.B. UninitialisedVariable beim Verwenden einer uninitialisierten Variable oder IndexOutOfBound bei einem 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 3.88 läuft z.B. ohne Fehlermeldungen durch.

```
1 #include <stdio.h>
2
3 void main() {
4   int var;
5   printf("\n%d", var);
6 }
```

Code 3.88: Beispiel für C-Programm, dass eine uninitialisierte Variable verwendet.

Fehlerart	Beschreibung
UnexpectedCharacter	Der Lexer ist auf eine unerwartete Zeichenfolge gestossen, die in
	der Konkretten Grammatik für die Lexikalische Analyse 3.1.1
	nicht abgeleitet werden kann.
UnexpectedToken	Der Parser hat ein unerwartetes Token erhalten, das in dem
	Kontext in dem es sich befand in der Konkretten Grammatik für
	die Syntaktische Analyse 3.2.10 nicht abgeleitet werden kann.
UnexpectedEOF	Der Parser hat in dem Kontext in dem er sich befand bestimmte
	Tokens erwartet, aber die Eingabe endete abrupt.

Tabelle 3.9: Fehlerarten in der Lexikalischen und Syntaktischen Analyse.

Fehlerart	Beschreibung
UnknownIdentifier	Es wird ein Zugriff auf einen Bezeichner gemacht (z.B. unknown_var + 1), der noch nicht deklariert und ist daher nicht in der Symboltabelle aufgefunden werden kann.
UnknownAttribute	Der Verbundstyp (z.B. struct st {int attr1; int attr2;}) auf dessen Attribut im momentanen Kontext zugegriffen wird (z.B. var[3].unknown_attr) besitzt das Attribut (z.B. unknown_attr) auf das zugegriffen werden soll nicht.
ReDeclarationOrDefinition	Ein Bezeichner ^a der bereits deklariert oder definiert ist (z.B. int var) wird erneut deklariert oder definiert (z.B. int var[2]). 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 intialisierten Konstante (z.B. const int const_var = 42) ein Wert zugewiesen wird (z.B. const_var = 41). Der einzige Weg, wie eine Konstante einen Wert erhält ist bei ihrere Initialisierung.
TooLargeLiteral	Der Wert eines Literals ist größer als $2^{31} - 1$ oder kleiner als -2^{31} .
${\tt NotExactlyOneMainFunction}$	Das Programm besitzt keine oder mehr als eine main-Funktion.
${\tt PrototypeMismatch}$	Der Prototyp einer deklarierten Funktion (z.B. int fun(int arg1, int arg2[3])) stimmt nicht mit dem Prototyp der späteren Definition dieser Funktion (z.B. void fun(int arg1[2], int arg2) { })) überein.
ArgumentMismatch	Wenn die Argumente eines Funktionsaufrufs (z.B. fun(42, 314)) nicht mit dem Prototyp der Funktion die aufgerufen werden soll (z.B. void fun(int arg[2]) { })) nach Datentypen oder Anzahl Argumente bzw. Parameter übereinstimmt.
MissingReturn	Wenn eine Funktion, die ihrem Prototyp zufolge einen Rückgabewert hat, der nicht vom Datentyp void ist (z.B. int fun() $\{\}$) als letzte Anweisung keine return-Anweisung hat, dass einen Wert des entsprechenden Datentyps zurückgibt ^b .

^a Z.B. von einer Funktion oder Variable.

Tabelle 3.10: Fehlerarten in den Passes.

Fehlerart	Beschreibung
DivisionByZero	Wenn bei einer Division durch 0 geteilt wird (z.B. var / 0).

Tabelle 3.11: Fehlerarten, die zur Laufzeit auftreten.

In Code 3.90 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 3.90 ist in Code 3.89 zu sehen. Nacheinander stehen in Code 3.90 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 ^~~~
```

^b Der entsprechende Datentyp müsste auf das Beispiel von davor void fun(int arg[2]) {...} bezogen z.B. return 42

Code 3.89: 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 angzeigt. 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 Lexeme, welches an der Stelle des Fehlers vorgefunden wurde und das Symbol ~ soll einen Pfeil symbolisieren, der auf eine Position zeigt an der ein anderer Tokentyp, ein anderer Datentyp usw. erwartet worden wäre und in der Zeile darunter eine Beschriftung an sich hängen hat, die konkret angibt, was dort eingentlich erwartet worden wäre.

Code 3.90: Beispiel für typische Fehlermeldung mit 'found' und 'expected'.

Bei Fehlermeldungen, wie in Code 3.91, die ihre Ursache an einer anderen Stelle im Code haben, wird einmal ein Header mit Programmauschnitt für die Stelle an welcher der Fehler aufgetreten ist erstellt und ein weiterer Header mit Programmauschnitt für die Stelle welche die Ursache für das Auftreten dieses Fehlers ist.

```
/tests/error_redefinition.picoc:6:6: Redefinition: Redefinition of 'var'.
2
    void main() {
       int var = 42;
4
5
       int var = 41;
6
7
8
     ./tests/error_redefinition.picoc:5:6: Note: Already defined here:
9
    void main() {
10
       int var = 42;
11
12
       int var = 41;
13
    }
```

Code 3.91: Beispiel für eine langgestreckte Fehlermeldung.

Bei manchen Fehlermeldungen, wie in Code 3.92 ist es garnicht möglich mit ~ ein Lexeme 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 Lexeme gibt, welches man markieren könnte. Des Weiteren ist in Code 3.92 interessant, dass in markierten Zeile in Code 3.92 mehrere Tokens angegeben werden,

die nach der Konkreten 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.

Code 3.92: Beispiel für Fehlermeldung mit mehreren erwarteten Tokens.

Bei wiederum anderen Fehlermeldungen, wie in Code 3.93 ist es nicht möglich ein erwartetes Token anzugeben, da das Programm in Code 3.93 eigenlich korrekt nach der Konkretten Grammatik 3.2.8 abgeleitet ist, weshalb sich hier keine erwarteten Tokens angeben lassen. Es liegt auf das konkrete Beispiel in Code 3.93 bezogen nämlich daran, dass die Variable unknown_identifier nicht definiert ist, weshalb dieses Programm nicht in der gesamten Syntax der Sprache L_{PicoC} sein kann.

Code 3.93: 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.

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 Einführung alle implementiert werden konnten und daraufhin mithilfe kurzer Anleitungen ein grober Einblick gegeben, wie auf diese Funktionalitäten Zugegriffen 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 Einführung 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 Schrifftlichen 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 nütztlich, weshalb für diese auf die ausführlichere Dokumentation unter Link¹ 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

¹ https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt

Basisname der Datei program an eine neue Dateiendung .reti 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 Dokumenation unter Link nachgelesen werden.

 $^{^2}$ Beim GCC wird bei Nicht-Angabe eines Dateinamen mit der -o Option dagegen eine Datei mit der festen Namen a. out erstellt.

Kommandozeilen- option	Beschreibung	Standard- wert
-i, intermediate_stages	Gibt Zwischenschritte der Kompilierung in Form der verschiedenen Tokens, Ableitungsbäume, Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateiendungen aber gleichem Basinamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal.	false, most_used: true
-p,print	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)
-v,verbose	Fügt den verschiedenen Zwischenschritten der Kompilierung, unter anderem auch dem finalen RETI-Code Kommentare hinzu, welche eine Anweisung oder einen Befehl aus einem vorherigen Pass beinhalten, der durch die darunterliegenden Anweisungen oder Befehle ersetzt wurde. Wenn dierun-Option aktivert ist, wird der Zustand der virtuellen RETI-CPU vor und nach jedem Befehl angezeigt.	false
-vv,double_verbose	Hat dieselben Effekte, wie die -verbose-Option, aber bewirkt zusätzlich weitere Effekte. PicoC-Knoten erhalten bei der Ausgabe in den Abstrakten Syntaxbäumen zustätzliche 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 derintermediate_stages-Option werden in den dadurch ausgegebenen Abstrakten Syntaxbäumen ebenfalls versteckte Attribute, die Informationen zu Datentypen und für Fehlermeldungen beinhalten angezeigt.	false
-h,help	Zeigt die Dokumentation , welche ebenfalls unter Link gefunden werden kann im Terminal an. Mit der color -Option kann die Dokumentation mit farblicher Hervorhebung im Terminal angezeigt werden.	false
-1	Es lässt sich einstellen, wieviele Zeilen rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	2
-c	Aktiviert farbige Ausgabe.	false, most_used: true
-t,thesis	Filtert für die Codebeispiele in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit bestimmte Kommentare in den Abstrakten Syntaxbäumen heraus, damit alles übersichtlich bleibt.	false

Tabelle 4.1: Kommandozeilenoptionen, Teil 1.

Kommandozeilen- option	Beschreibung	Standard- wert		
-R,run	Führt die RETI-Befehle, die das Ergebnis der Kompilierung sind	false,		
	mit einer virtuellen RETI-CPU aus. Wenn die	$most_used$:		
	intermediate_stages-Option aktiviert ist, wird eine Datei	true		
	<pre><basename>.reti_states erstellt, welche den Zustsand der</basename></pre>			
	RETI-CPU nach dem letzten ausgeführten RETI-Befehl enthält.			
	Wenn dieverbose- oderdouble_verbose-Option aktiviert ist,			
	wird der Zustand der RETI-CPU vor und nach jedem Befehl auch			
	noch zusätlich in die Datei <basename>.reti_states ausgegeben.</basename>			
-B,process_begin	Setzt die relative Adresse, wo der Prozess bzw. das	3		
	Codesegment für das ausgeführte Programm beginnt.			
-D,	-D, Setzt die Größe des Datensegments. Diese Option muss mit			
datasegment_size	Vorsicht gesetzt werden, denn wenn der Wert zu niedrig gesetzt			
	wird, dann können die Globalen Statischen Daten und der			
	Stack miteinander kollidieren.			

Tabelle 4.2: Kommandozeilenoptionen, Teil 2.

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 Anweisungen 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 \rightarrow picoc_compiler aufgerufen, wie es in Code 4.1 zu sehen ist. Die angegebene Folge von Anweisungen \leftarrow wird dabei automatisch in eine main-Funktion eingefügt: void main() \leftarrow seq-of-stmts \rightarrow .

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, Ableitungsbaum 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 \leftarrow und \rightarrow Pfeiltasten bewegt werden. In der Befehlshistorie kann sich mit den \uparrow und \downarrow Pfeiltasten rückwarts und vorwärts bewegt werden. Mit Tab 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 help 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 4.3 aufgelistet.

Kommandozeilenoption	Beschreibung	Standardwert
FILEPATH	Pfad zur Datei, die im Show-Mode angezeigt werden soll	Ø
TESTNAME	Name des Tests. Alles andere als der Basisname, wie die Dateiendung wird abgeschnitten	Ø
EXTENSION	Dateiendung, die an TESTNAME angehängt werden soll zu ./tests/TESTNAME.EXTENSION	reti_states
NUM_WINDOWS	Anzahl Fenster auf die ein Dateiinhalt verteilt werden soll	5
VERBOSE	Möglichkeit die Kommandozeilenoption -v oder -vv zu aktivieren für eine ausführlichere Ausgabe	Ø
DEBUG	Möglichkeit die Kommandozeilenoption -d zu aktivieren, um bei make test-show TESTNAME= <testname> den Debugger für den entsprechenden Test <testname> zu starten</testname></testname>	Ø

Tabelle 4.3: 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 Neovim³ um einen Dateiinhalt 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.

 $^{^3}Home$ - Neovim.

```
0021 JUMP 44;
0022 MOVE BAF IN1;
0023 LOADIN IN1 BAF 0;
                                                                                                                                                       059 ADD ACC IN2;
060 STOREIN SP ACC 2;
061 ADDI SP 1; <- PC
      STMPLE:
                                                                            0024 MOVE IN1 SP;
0025 SUBI SP 1;
0026 STOREIN SP ACC 1;
IN1 SIMPLE:
                                                                                                                                                      062 LOADIN SP ACC 1;
                                                                                                                                                                                                                         00100 LOADI ACC 101;
                                                                                                                                                      063 ADDI SP 1;
064 LOADIN BAF PC -1;
                                                                                                                                                                                                                         00101 ADD ACC CS;
00102 STOREIN BAF ACC -1;
                                                                                                                                                                                                                                                                                               00139 2
00140 42
N2 SIMPLE:
                                                                          00027 LOADIN SP ACC 1;
00028 STOREIN DS ACC 1;
00029 ADDI SP 1;
00030 SUBI SP 1;
00031 LOADIN DS ACC 1;
00032 STOREIN SP ACC 1;
                                                                                                                                                      065 SUBI SP 1;
066 LOADI ACC 2;
067 STOREIN SP ACC 1;
                                                                                                                                                                                                                         00103 JUMP -58;
00104 MOVE BAF IN1;
00105 LOADIN IN1 BAF 0;
                         2147483709
                                                                                                                                                                                                                                                                                                00141 2
                                                                                                                                                                                                                                                                                                00143 2147483752
                                                                                                                                                       1068 LOADIN SP ACC 1;
1069 STOREIN BAF ACC
1070 ADDI SP 1;
                                                                                                                                                                                                                         00106 MOVE IN1 SP;
00107 SUBI SP 1;
00108 STOREIN SP ACC 1;
     STMPLE:
                                                                                                                                                                                                                                                                                                00144 2147483797 <- BAF
                         2147483651
                                                                                                                                                 00071 SUBI SP 1;
00072 LOADIN BAF ACC -2;
00073 STOREIN SP ACC 1;
                                                                                                                                                                                                                         00109 LOADIN SP ACC 1;
00110 ADDI SP 1;
00111 CALL PRINT ACC;
                                                                           00033 SUBT SP 1:
                                                                                                                                                                                                                                                                                                00147 38
                                                                              0034 LOADI ACC 2;
0035 STOREIN SP ACC 1;
                                                                                                                                                                                                                                                                                                00149 2147483656
                                                                                                                                                                                                                        00112 SUBI SP 1;
00113 LOADIN BAF ACC -4;
00114 STOREIN SP ACC 1;
00115 LOADIN SP ACC 1;
00116 ADDI SP 1;
00117 LOADIN BAF PC -1;
                                                                                                                                                  00074 SUBI SP 1;
00075 LOADIN BAF ACC -3;
00076 STOREIN SP ACC 1;
     SIMPLE:
                                                                           00036 LOADIN SP ACC 2;
00037 LOADIN SP IN2 1;
                                                                           00038 ADD ACC IN2:
   00001 2147483648
                                                                                                                                                 00077 LOADIN SP ACC 2;
00078 LOADIN SP IN2 1;
                                                                           00039 STOREIN SP ÁCC 2:
                                                                          00040 ADDI SP 1;
00041 LOADIN SP ACC 1;
   00002 0
00003 CALL INPUT ACC; <- CS
                                                                                                                                                      079 ADD ACC IN2:
                                                                          00042 ADDI SP 1;
00043 CALL PRINT ACC;
00044 LOADIN BAF PC -1;
                                                                                                                                                      080 STOREIN SP ACC 2;
081 ADDI SP 1;
082 LOADIN SP ACC 1;
      0004 SUBI SP 1;
0005 STOREIN SP ACC 1;
                                                                                                                                                                                                                                                                                                 00000 LOADI DS -2097152; <- IN
00001 MULTI DS 1024;
                                                                                                                                                                                                                                                                                                 00002 MOVE DS SP; <- IN2
      0006 LOADIN SP ACC 1;
         06 LUADIN SP ACC 1;

07 STOREIN DS ACC 0;

08 ADDI SP 1;

09 SUBI SP 2;

10 SUBI SP 1;

11 LUADIN DS ACC 0;
                                                                                                                                                      082 LUADIN SF ACC 1;
083 STOREIN BAF ACC
084 ADDI SP 1;
085 SUBI SP 1;
                                                                          00045 SUBI SP 1;
00046 LOADI ACC 2;
00047 STOREIN SP ACC 1;
                                                                                                                                                                                                                                                                                                    0003 MOVE DS BAF:
                                                                             0048 LOADIN SP ACC 1;
0049 STOREIN BAF ACC -3;
                                                                                                                                                      086 LOADIN BAF ACC -4;
087 STOREIN SP ACC 1;
                                                                              050 ADDI SP 1;
051 SUBI SP 1;
052 LOADIN BAF ACC -2;
              STOREIN SP ACC 1:
                                                                                                                                                      088 LOADIN SP ACC 1:
                                                                                                                                                       089 ADDI SP 1;
090 CALL PRINT ACC;
                                                                                                                                                       91 SUBI SP 2;
                                                                                      STOREIN SP ACC 1:
                                                                                      SUBI SP 1;
LOADIN BAF ACC
```

Abbildung 4.1: Show-Mode in der Verwendung.

Zur besseren Orientierung wird für alle Register ebenfalls ein mit der Registerbezeichnung beschriffteter 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 2.3.1 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 Link⁴ zu finden. Eingeteilt sind die Tests in die folgenden Kategorien in Tabelle 4.4.

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
	Funktionaliäten 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 Verfikation
	ausgeführt.
exclude	Tests, für welche aufgrund vielfältiger Gründe keine Verifikation ausgeführt
	werden soll.
thesis	Tests, die eigentlich vorher Codebeispiele für diese Schrifftliche
	Ausarbeitung der Bachelorarbeit waren.
tobias	Tests, die der Betreuer dieser Bachelorarbeit, Tobias geschrieben hat.

Tabelle 4.4: 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 Interpreters⁵ aus Diagramm 2.3.1 übernimmt und die erwartete Ausgabe seiner eigenen Interpretation des PicoC-Codes anstelle von <space_seperated_output> hinein-schreibt.

Ein Beispiel für einen Test ist in Code 4.3 zu sehen. Sobald die Tests mithilfe des Bashcripts /run_tests.sh ausgeführt werden oder dieses mithilfe der /Makefile mit dem Befehl > make test ausgeführt wird, 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 cprogram>.in, <program>.out_expected und cprogram>.datasegment_size</code> schreibt. Das letztere Skript kann ebenfalls mit dem Befehl > make extract ausgeführt werden.

Die Datei
cprogram>.in enthält Eingaben, welche durch input()-Funktionsaufrufe eingelesen werden, die Datei
cprogram>.out_expected enthält zu erwartende Ausgaben der print(<exp>)-Funktionaufrufe, die später eingeführte Datei
cprogram>.out enthält die tatsächlichen Ausgaben der print(<exp>)-Funktionsaufrufe bei der Ausführung des Tests und die Datei
cprogram>.datasegment_size
enthält die Größe des Datensegments für die Ausführung des entsprechenden Tests.

⁵Der die **Semantik** des Tests umsetzt.

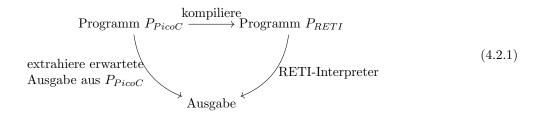
⁶Falls vorhanden.

```
// 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.

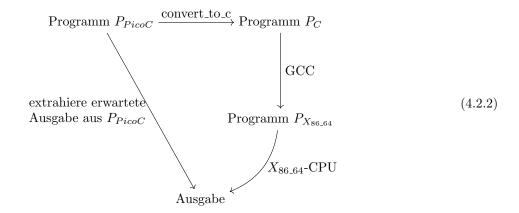


Allerdings gibt es bei dem Testverfahren, welches in Diagramm 4.2.1 dargestellt ist ein Problem, denn der Schreiber der Tests ist in diesem Fall die gleiche Person, die auch den Compiler implementiert. Wenn der Schreiber der Tests ein falsches Verständnis davon hat, wie das Ergebnis eines Ausdrucks berechnet 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_{C} 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 \blacktriangleright 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 $P_{X_{86_64}}$. 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 tätsächliche 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.

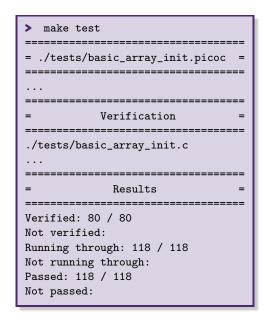
 $^{^9} History$ - $GCC\ Wiki$.

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.



Code 4.4: Testdurchlauf.

Der Befehl make test <more-options> lässt sich ebenfalls mit den Makefileoptionen <more-options> TESTNAME, VERBOSE und DEBUG aus Tabelle 4.3 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 #includestdio.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-Compiler¹⁴ 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 (Definition 2.48) zuzuweisen wird mithilfe einer Liveness Analyse (Defintion 5.11) ein Interferenzgraph

¹⁴Möglicherweise ja im Rahmen eines Masterprojektes ².

(Definition 5.14) aufgebaut. Auf den Interferenzgraph wird ein Graph Coloring Algorithmus (Definition 5.13) 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 Kontrollflussnalayse (Definition 5.17) iterativ unter Verwendung eines Kontrollflussgraphen (Definition 5.15) auf die verschiedenen Blöcke angewendet werden, bis sich an den Live Variablen nichts mehr ändert.¹⁵

- Tail Call: Wenn ein Funktionsaufruf die letzte Anweisung 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 partiell zu 2 + input(), input() und input() * 2 berechnet werden. Dies 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 L_{PicoC-Eval} umformt. In der Abstrakten Syntax der Sprache L_{PicoC-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 L_{C++} 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 5.6) können die kompilierten .o-Objektdateien (Definition 5.5) 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 Bibltiothek stdlib¹⁸ implementiert, deren .h-Headerdatei mittels #include "malloc_and_free.h" eingebunden wer-

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.

 $^{^{17}}$ Es gibt hierfür leider keinen deutschen Begriff, der geläufig ist.

¹⁸Auch engl. General Purpose Standard Library genannt.

den muss. Es braucht eine neue Kommandozeilenoption -1 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 Zeiger 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 5.18) 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 C. Scholl, "Betriebssysteme" festgelegt wurde.
- PicoPython: Da das Lark Parsing Toolkit verwendet wurde, welches das Parsen über eine selbst angegebene Konkrete Grammatik übernimmt, könnte mit relativ geringem Aufwand ein Konkrete Grammatik defininiert werden, die eine zur Programmiersprache L_{Python} ähnliche Konkrete Syntax beschreibt. Die Konkrete Syntax einer Programmiersprache lässt sich durch Austauschen der Konkreten Grammatik sehr einfach ändern, nur die Semanatik zu ändern kann deutlich aufwändiger sein. Viele der PicoC-Knoten könnten für die Programmiersprache $L_{PicocPython}$ wiederverwendet werden und viele Passes müssten nur erweitert werden.
- Call by Reference: Über das wiederverwenden des &-Symbols für Parameter bei Funktiondeklaration und Funktionsdefinition, wie es in der Vorlesung 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 eine Anweisungen beginnen und wo sie aufhören usw., damit der Debugger weiß, bis wohin er die RETI-Befehle ausführen soll, damit er eine Anweisung abgearbeitet hat.
- Bootstrapping: Mittels Bootstrapping lässt sich der PicoC-Compiler unabängig von der Sprache L_{Python} und der Maschine, die das cross-compilen (Definition 2.6) übernimmt machen. Im Unterkapitel 4.3 wird genauer hierauf eingegangen. Hierdurch wird der PicoC-Compiler zum einem Compiler für die RETI-CPU gemacht, der auf der RETI-CPU selbst läuft.

```
in:42
      expected:0
 3
   int ret0() {
    return 0;
 6 }
 8 int ret1() {
    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
  void main() {
19
20
    print(tail_call_fun(input()));
21 }
```

Code 4.5: Beispiel für Tail Call.

Anmerkung Q

Partielle Evaluation und Lazy Evaluation wurden im PicoC-Compiler nicht impelementiert, da dieser als Lerntool gedacht ist und dieses 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.

Appendix

Dieses Kapitel dient als Lagerstätte für Definitionen, Tabellen, Abbildungen und ganze Unterkapitel, die zum Erhalt des roten Fadens und des Leseflusses in den vorangegangenen Kapiteln hierher ausgelaggert wurden. Im Unterkapitel RETI Architektur Details können einige Details der RETI-Architektur nachgeschaut werden, die im Kapitel Einführung den Lesefluss stören würden und zum Verständnis nur bedingt wichtig sind. Im Unterkapitel Sonstige Definitionen sind einige Definitionen ausgelaggert, die zum Verständnis der Implementierung des PicoC-Compilers nicht wichtig sind, aber z.B. an einer bestimmten Stelle in den vorangegangenen Kapiteln kurz Erwähnung fanden. Im Unterkapitel Bootstrapping wird ein Vorgehen, das Bootsrapping erklärt, welches beim PicoC-Compiler nicht umgesetzt wurde, es aber erlauben würde aus dem PicoC-Compiler einen Compiler für die RETI-CPU zu machen, der auf der RETI-CPU selbst läuft.

RETI Architektur Details

Typ	Modus	Befehl	Wirkung
01	00	LOAD D i	$D := M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
01	01	LOADIN S D i	$D := M(\langle S \rangle + i), \langle PC \rangle := \langle PC \rangle + 1$
01	11	LOADI D i	$D := 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1, \text{ bei } D = PC \text{ wird der PC}$
			nicht inkrementiert
10	00	STORE S i	$M(\langle i \rangle) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	01	STOREIN D S i	$M(\langle D \rangle + i) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	11	MOVE S D	$D := S, \langle PC \rangle := \langle PC \rangle + 1$, Move: Bei $D = PC$ wird der
			PC nicht inkrementiert

Tabelle 5.1: Load und Store Befehle.

Typ	\mathbf{M}	RO	${f F}$	Befehl	Wirkung
00	0	0	000	ADDI D i	$[D] := [D] + [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	001	SUBI D i	$[D] := [D] - [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	010	MULI D i	$[D] := [D] * [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	011	DIVI D i	$[D] := [D] / [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	100	MODI D i	$[D] := [D] \% [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	OPLUSI D i	$[D] := [D] \oplus 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	110	ORI D i	$[D] := [D] \lor 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	ANDI D i	$[D] := [D] \wedge 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	000	ADD D i	$[D] := [D] + [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	001	SUB D i	$[D] := [D] - [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	010	MUL D i	$[D] := [D] * [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	011	DIV D i	$[D] := [D] / [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	100	MOD D i	$[D] := [D] \% [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	OPLUS D i	$D := D \oplus M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	110	OR D i	$D := D \lor M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	AND D i	$D := D \land M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	000	ADD D S	$[D] := [D] + [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	001	SUB D S	$[D] := [D] - [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	010	MUL D S	$[D] := [D] * [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	011	DIV D S	$[D] := [D] / [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	100	MOD D S	$[D] := [D] \% [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	OPLUS D S	$D := D \oplus S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	110	OR D S	$D := D \lor S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	AND D S	$D := D \land S, \langle PC \rangle := \langle PC \rangle + 1$

Tabelle 5.2: Compute Befehle.

Type	Condition	J	Befehl	Wirkung
11	000	00	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11	001	00	$\mathrm{JUMP}_{>}\mathrm{i}$	Falls $[ACC] > 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	010	00	$JUMP_{=}i$	Falls $[ACC] = 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	011	00	$\mathrm{JUMP}_{\geq}\mathrm{i}$	Falls $[ACC] \ge 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	100	00	$JUMP_{<}i$	Falls $[ACC] < 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	101	00	$\mathrm{JUMP}_{ eq}\mathrm{i}$	Falls $[ACC] \neq 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	110	00	$JUMP \le i$	Falls $[ACC] \le 0$: $\langle PC \rangle := \langle PC \rangle + [i]$, sonst: $\langle PC \rangle := \langle PC \rangle + 1 \langle PC \rangle := \langle PC \rangle + [i]$
11	111	00	JUMPi	$\langle PC \rangle := \langle PC \rangle + [i]$
11	*	01	INT i	$\langle PC \rangle := IVT[i]$ Interrupt Nr.i wird Ausgeführt
11	*	10	RTI	Rücksprungadresse vom Stack entfernt, in PC geladen, Wechsel in Usermodus

Tabelle 5.3: Jump Befehle.

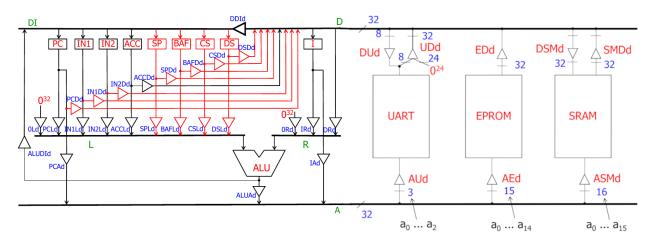


Abbildung 5.1: Datenpfade der RETI-Architektur.

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 tuen haben.

Definition 5.1: Bezeichner (bzw. Identifier)

Z

Zeichenfolge^a, die eine Konstante, Variable, Funktion usw. innerhalb ihres Sichtbarkeitsbereichs eindeutig benennt. ^{b c}

Definition 5.2: Label

7

Durch einen Bezeichner eindeutig zuordenbares Sprungziel im Programmcode.^a

Definition 5.3: Assemblersprache (bzw. engl. Assembly Language)

1

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

^aBzw. Tokenwert.

 $[^]b$ Außer wenn z.B. bei Funktionen die Programmiersprache das Überladen erlaubt usw. In diesem Fall wird die Signatur der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.

^cThiemann, "Einführung in die Programmierung".

^aThiemann, "Compilerbau".

 $[^]a$ Befehle der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als Pseudo-Befehle bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

 $^{{}^{}b}$ Z.B. erlaubt die Assemblersprache des GCC für die X_{86_64} -Architektur für manche Operanden die Syntax n(%r), die einen Speicherzugriff mit Offset n zur Adresse, die im Register %r steht durchführt, wobei z.B. die Klammern () usw. nur "syntaktischer Zucker" sind und natürlich nicht mitkodiert werden.

 c Z.B. sind im $X_{86.64}$ Assembler die Befehle in Blöcken untergebracht, die ein Label haben und zu denen mittels jmp <label> gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher. d P. Scholl, "Einführung in Embedded Systems".

Anmerkung Q

Ein Assembler (Definition 5.4) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt Maschinencode bzw. Objectcode (Definition 5.5) 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 die Ausgabe liefern, welche er in den allermeisten Fällen haben will, nämlich den Maschinencode bzw. Objectcode, der direkt ausführbar ist bzw. wenn er später mit dem Linker (Definition 5.6) zu Maschienencode zusammengesetzt wird ausführbar ist.

Definition 5.4: Assembler

Z

Übersetzt im allgemeinen Assemblercode, der in Assemblersprache geschrieben ist zu Maschinencode bzw. Objectcode in binärerer Repräsentation, der in Maschinensprache geschrieben ist.^a

^aP. Scholl, "Einführung in Embedded Systems".

Definition 5.5: 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 Maschiendencode nicht mehr enthalten sind, sobald der Linker die Objektdateien zum Maschinencode zusammengesetzt hat.^a

^aP. Scholl, "Einführung in Embedded Systems".

Definition 5.6: Linker



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

^aP. Scholl, "Einführung in Embedded Systems".

Definition 5.7: Transpiler (bzw. Source-to-source Compiler)



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

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

^bThiemann, "Compilerbau".

Definition 5.8: Rekursiver Abstieg

Z

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.

Anmerkung Q

Bei manchen Ansätzen für das Parsen eines Programmes, ist es notwendig eine LL(k)-Grammatik (Definition 5.9) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des Rekursiven Abstiegs (Definition 5.8) verwenden lässt sich eine bessere minimale Laufzeit garantieren, da aufgrund der LL(k)-Eigenschafft ausgeschlossen werden kann, dass Backtracking notwendig ist^a.

^aMehr Erklärung hierzu findet sich im Unterkapitel 2.4.

Definition 5.9: LL(k)-Grammatik

Z

Eine Grammatik ist LL(k) für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die nächsten k Tokens des Eingabeworts 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

Definition 5.10: Earley Erkenner

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

Eingabe und Ausgabe des Algorithmus sind:

- Eingabe: Eingabewort w und Konkrete Grammatik $G_{Parse} = \langle N, \Sigma, P, S \rangle$.
- Ausgabe: 0 wenn $w \notin L(G_{Parse})^d$ 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^e 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 geparst wurde und β noch geparst werden muss.
- Die Indexierung ist informell ausgedrückt so umgesetzt, dass die Indices zwischen Tokentypen liegen, also Index 0 vor dem ersten Tokentyp verortet ist, Index 1 nach dem ersten Tokentyp verortet ist und Index n nach dem letzten Tokentyp verortet ist.

und davor müssen noch einige Begriffe definiert werden:

 $[^]a$ Das wird auch als **Lookahead** von k bezeichnet.

 $[^]b$ Wobei sich das mit den Linksableitungen automatisch ergibt, wenn man das Eingabewort von links-nach-rechts parsed und jeder der nächsten k Ableitungsschritte eindeutig sein soll.

^cNebel, "Theoretische Informatik".

- 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 geparst 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 ::=-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 Konkreten Grammatik, die ein B auf der linken Seite des ::=-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 $(A ::= \alpha \bullet \alpha \gamma, i)$ 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 ::= β•,i) hat werden alle Zustände in Z(i) gesucht, welche die Form (A ::= α•Bγ,i)
 haben und es wird der Zustand (A ::= αB•γ,i) zur Zustandsmenge Z(j) hinzugefügt.

bis:

- der Zustand $(A := \beta \bullet, 0)$ in der Zustandsmenge Z(n) auftaucht, wobei A das Startsymbol S ist $\Rightarrow w \in L(G_{Parse})$.
- keine Zustände mehr hinzugefügt werden können $\Rightarrow w \notin L(G_{Parse})$.

Definition 5.11: Liveness Analyse

1

Findet heraus, welche Variablen in welchen Regionen eines Programmes verwendet werden.^a

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

Definition 5.12: Live Variable

1

Eine Location, deren momentaner Wert später im Programmablauf noch verwendet wird. Man sagt auch die Location ist live. ab

^aJay Earley, "An efficient context-free parsing".

^bErklärweise wurde von der Webseite Earley parser übernommen.

^cEarley Parser.

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

^eAlso eine Folge von Terminalsymbolen und Nicht-Terminalsymbolen.

 $[^]a\mathrm{Es}$ gibt leider kein allgemein verwendetes deutsches Wort für Live Variable.

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

Definition 5.13: Graph Coloring

Z

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.
- b Bzw. Farben.
- ^cEs gibt leider kein allgemein verwendetes deutsches Wort für Graph Coloring.
- ^dG. Siek, Course Webpage for Compilers (P423, P523, E313, and E513).

Definition 5.14: Interference Graph

7

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

Definition 5.15: Kontrollflussgraph



Gerichteter Graph, der den Kontrollfluss eines Programmes beschreibt.^a

Definition 5.16: Kontrollfluss

7

Die Reihenfolge in der z.B. Anweisungen, Funktionsaufrufe usw. eines Programmes ausgewertet werden^a.

Definition 5.17: Kontrollflussanalyse

Analyse des Kontrollflusses (Defintion 5.16) eines Programmes, um herauszufinden zwischen welchen Teilen des Programms Daten ausgetauscht werden und welche Abhängigkeiten sich daraus ergeben.

Der simpelste Ansatz ist es in einen Kontrollflussgraph iterativ einen Algorithmus^a anzuwenden, bis sich an den Werten der Knoten nichts mehr $\ddot{a}ndert^b$.

- ^aIm Bezug zu Compilerbau die Linveness Analayse.
- ^bBis diese sich **stabilisiert** haben
- ^cG. Siek, Course Webpage for Compilers (P423, P523, E313, and E513).

Definition 5.18: Two-Space Copying Collector

Z

Ein Garbabe 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

^aBzw. Farbe.

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

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

^aMan geht hier von einem **imperativen** Programm aus.

^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 $C_{RETI_PicoC}^{PicoC}$ (Defintion 5.19) 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. Mittels Bootrapping wird aus dem PicoC-Compiler ein "richtiger Compiler" für die RETI-CPU gemacht, der auf der RETI-CPU selbst läuft.

Anmerkung Q

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. Falls 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 5.20) kann man das als C_{min} schreiben.

Definition 5.19: Self-compiling Compiler

Z

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 5.22) zum laufen gebracht wurden, den gerade beschriebenen Self-compiling Compiler $C_{RETI_PicoC}^{PicoC}$ implementieren und hat bereits den gesamtem Self-compiling Compiler $C_{RETI_PicoC}^{PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das Henne-Ei-Problem² 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 5.21) 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

¹Ein üblicher Compiler, wie ihn ein Programmierer verwendet, wie GCC oder Clang läuft üblicherweise selbst auf der Maschine für welche er kompiliert.

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.

 $^{^3}$ Im Fall, dass auf der Maschine M_{RETI} die Programmiersprache L_{Python} bereits mittels Bootstrapping zum Laufen gebracht wurde, könnte der Self-compiling Compiler $C_{RETI_PicoC}^{PicoC}$ auch mithife des Cross-Compilers C_{PicoC}^{Python} als externe Entität und der Programmiersprache L_{Python} auf der Maschine M_{RETI} selbst kompiliert werden.

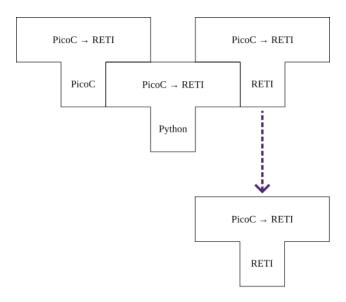


Abbildung 5.2: Cross-Compiler als Bootstrap Compiler.

Anmerkung 9

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 allersten Compiler C_{first} anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

Definition 5.20: Minimaler Compiler

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

Definition 5.21: Boostrap Compiler

1

Compiler C_w^o , der es ermöglicht einen Self-compiling Compiler C_w^w zu boostrapen, indem der Self-compiling Compiler C_w^o mit dem Bootstrap Compiler C_w^o kompiliert wird. Der Bootstrapping Compiler stellt die externe Entität dar, die es ermöglicht die zirkulare Abhängikeit, dass initial ein Self-compiling Compiler C_w^o bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.

 $^{{}^{}a}$ In diesem Fall, dem Cross-Compiler C_{PicoC}^{Python}

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

^bThiemann, "Compilerbau".

^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 5.20) 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 5.22.1 den minimalen Compiler schrittweise zu einem immer vollständigeren C-Compiler C_C weiterentwickelt.

Definition 5.22: Bootstrapping

Z

Wenn man einen Self-compiling Compiler C_w^w einer Wunschsprache L_w auf einer Zielmaschine M zum laufen bringt^{abcd}. Dabei ist die Art von Bootstrapping in 5.22.1 nochmal gesondert hervorzuheben:

5.22.1: Wenn man die aktuelle Version eines Self-compiling Compilers $C_{w_i}^{w_i}$ der Wunschsprache L_{w_i} mithilfe von früheren Versionen seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C_{w_{i-1}}^{w_{i-1}}$ kompiliert wird und schafft es so iterativ immer umfangreichere Compiler zu bauen. efg

^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 Lügengeschichten 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 weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache L_o , in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.

 d Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute Probe aufs Exempel darstellen, dass der Compiler auch wirklich funktioniert.

^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} .

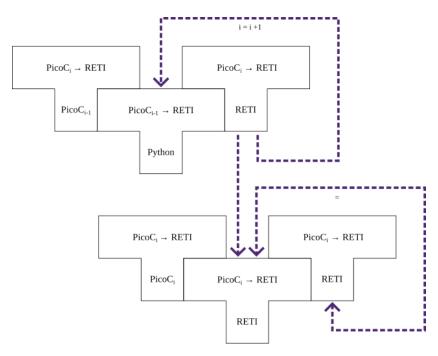


Abbildung 5.3: Iteratives Bootstrapping.

Anmerkung Q

Auch wenn ein Self-compiling Compiler $C_{w_i}^{w_i}$ in der Subdefinition 5.22.1 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 5.22.1 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.

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.quut.com/c/ANSI-C-grammar-1-2011.html (besucht am 15.08.2022).
- ANSI C grammar (Lex) old. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (besucht am 15.08.2022).
- ANSI C grammar (Yacc). URL: https://www.quut.com/c/ANSI-C-grammar-y.html (besucht am 15.08.2022).
- ANSI C grammar (Yacc) old. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html (besucht am 15.08.2022).
- 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).
- Earley Parser. URL: https://rahul.gopinath.org/post/2021/02/06/earley-parsing/ (besucht am 20.06.2022).
- 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).
- Variablen in C und C++, Deklaration und Definition Coder-Welten.de. URL: https://www.coder-welten.de/einstieg/variablen-in-c-3.html (besucht am 11.08.2022).
- 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.
- 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, 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, Bernhard. "Theoretische Informatik". Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09.07.2022).
- Scholl, Christoph. "Betriebssysteme". Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- Scholl, Philipp. "Einführung in Embedded Systems". Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09.07.2022).
- Scholl, Prof. Dr. Christoph. "Technische Informatik". Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03.08.2022).
- Thiemann, Peter. "Compilerbau". Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09.07.2022).
- — "Einführung in die Programmierung". Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09.07.2022).
- Westphal, Dr. Bernd. "Softwaretechnik". Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19.07.2022).

Sonstige Quellen

- 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. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23.07.2022).
- Earley parser. In: Wikipedia. Page Version ID: 1090848932. 31. Mai 2022. URL: https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=1090848932 (besucht am 15.08.2022).
- Lark a parsing toolkit for Python. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28.04.2022).
- Naming convention (programming). In: Wikipedia. Page Version ID: 1100066005. 24. Juli 2022. URL: https://en.wikipedia.org/w/index.php?title=Naming_convention_(programming)&oldid=1100066005 (besucht am 30.07.2022).
- 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).
- 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).