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

Al	bbild	ıngsverzeichnis]
Co	odeve	rzeichnis	IJ
Ta	abelle	nverzeichnis	IV
D	efinit	onsverzeichnis	V
\mathbf{G}_{1}	ramn	atikverzeichnis	VI
1	Imp	ementierung	1
	1.1	Lexikalische Analyse	. 3
		1.1.1 Konkrete Grammatik für die Lexikalische Analyse	. 3
		1.1.2 Codebeispiel	. 5
	1.2	Syntaktische Analyse	. 6
		1.2.1 Umsetzung von Präzedenz und Assoziativität	. 6
		1.2.2 Konkrete Grammatik für die Syntaktische Analyse	. 11
		1.2.3 Ableitungsbaum Generierung	. 13
		1.2.3.1 Codebeispiel	. 14
		1.2.3.2 Ausgabe des Ableitunsgbaumes	. 15
		1.2.4 Ableitungsbaum Vereinfachung	
		1.2.4.1 Codebeispiel	
		1.2.5 Generierung des Abstrakten Syntaxbaumes	
		1.2.5.1 PicoC-Knoten	
		1.2.5.2 RETI-Knoten	
		1.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung	
		1.2.5.4 Abstrakte Grammatik	
		1.2.5.5 Codebeispiel	
		1.2.5.6 Ausgabe des Abstrakten Syntaxbaumes	
	1.3	Code Generierung	
		1.3.1 Passes	
		1.3.1.1 PicoC-Shrink Pass	
		1.3.1.1.1 Abstrakte Grammatik	
		1.3.1.1.2 Codebeispiel	
		1.3.1.2 PicoC-Blocks Pass	
		1.3.1.2.1 Abstrakte Grammatik	
		1.3.1.2.2 Codebeispiel	
		1.3.1.3.1 Abstrakte Grammatik	
		1.3.1.3.2 Codebeispiel	
		1.3.1.4.1 Abstrakte Grammatik	
		1.3.1.5 RETI-Patch Pass	
		1.3.1.5.1 Abstrakte Grammatik	
		1.3.1.5.2 Codebeispiel	
		1.3.1.5.2 Codebeispiel	. 50

	1.3.1.6.1 Konkrete und Abstrakte Grammatik
	1.3.1.6.2 Codebeispiel
1.3.2	Umsetzung von Zeigern
	1.3.2.1 Referenzierung
	1.3.2.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen 61
1.3.3	Umsetzung von Feldern
	1.3.3.1 Initialisierung eines Feldes
	1.3.3.2 Zugriff auf einen Feldindex
	1.3.3.3 Zuweisung an Feldindex
1.3.4	Umsetzung von Verbunden
	1.3.4.1 Deklaration von Verbundstypen und Definition von Verbunden
	1.3.4.2 Initialisierung von Verbunden
	1.3.4.3 Zugriff auf Verbundsattribut
	1.3.4.4 Zuweisung an Verbundsattribut
1.3.5	Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen 87
	1.3.5.1 Anfangsteil
	1.3.5.2 Mittelteil
	1.3.5.3 Schlussteil
1.3.6	Umsetzung von Funktionen
	1.3.6.1 Mehrere Funktionen
	1.3.6.1.1 Sprung zur Main Funktion
	1.3.6.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen 110
	1.3.6.3 Funktionsaufruf
	1.3.6.3.1 Rückgabewert
	1.3.6.3.2 Umsetzung der Übergabe eines Feldes
	1.3.6.3.3 Umsetzung der Übergabe eines Verbundes
Literatur	${f A}$

Abbildungsverzeichnis

1.2 Ableitungsbaum nach Parsen eines Ausdrucks. 1.3 Ableitungsbaum nach Vereinfachung. 1.4 Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen. 1.5 Generierung eines Abstrakten Syntaxbaumes mit Umdrehen. 1.6 Kompiliervorgang Kurzform. 1.7 Architektur mit allen Passes ausgeschrieben. 1.8 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.		8
1.4 Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen	1	6
1.5 Generierung eines Abstrakten Syntaxbaumes mit Umdrehen. 1.6 Kompiliervorgang Kurzform. 1.7 Architektur mit allen Passes ausgeschrieben. 1.8 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.	1	7
1.6 Kompiliervorgang Kurzform. 1.7 Architektur mit allen Passes ausgeschrieben. 1.8 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen. 1.8 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.	1	9
1.7 Architektur mit allen Passes ausgeschrieben	1	9
1.8 Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen	3	32
	3	3
	8	38
1.9 Veranschaulichung der Dinstanzberechnung	11	9

Codeverzeichnis

1.1	PicoC-Code des Codebeispiels
1.2	Tokens für das Codebeispiel
1.3	Ableitungsbaum nach Ableitungsbaum Generierung
1.4	Ableitungsbaum nach Ableitungsbaum Vereinfachung
1.5	Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum
1.6	PicoC Code für Codebespiel
1.7	Abstrakter Syntaxbaum für Codebespiel
1.8	PicoC-Blocks Pass für Codebespiel
1.9	Symboltabelle für Codebespiel
1.10	PicoC-ANF Pass für Codebespiel
	RETI-Blocks Pass für Codebespiel
	RETI-Patch Pass für Codebespiel
1.13	RETI Pass für Codebespiel
1.14	PicoC-Code für Zeigerreferenzierung
1.15	Abstrakter Syntaxbaum für Zeigerreferenzierung
1.16	Symboltabelle für Zeigerreferenzierung 59
1.17	PicoC-ANF Pass für Zeigerreferenzierung
	RETI-Blocks Pass für Zeigerreferenzierung
	PicoC-Code für Zeigerdereferenzierung
1.20	Abstrakter Syntaxbaum für Zeigerdereferenzierung 6
1.21	PicoC-Shrink Pass für Zeigerdereferenzierung
1.22	PicoC-Code für die Initialisierung eines Feldes
1.23	Abstrakter Syntaxbaum für die Initialisierung eines Feldes 65
1.24	Symboltabelle für die Initialisierung eines Feldes
1.25	PicoC-ANF Pass für die Initialisierung eines Feldes
1.26	RETI-Blocks Pass für die Initialisierung eines Feldes
1.27	PicoC-Code für Zugriff auf einen Feldindex
	Abstrakter Syntaxbaum für Zugriff auf einen Feldindex
	PicoC-ANF Pass für Zugriff auf einen Feldindex
	RETI-Blocks Pass für Zugriff auf einen Feldindex
	PicoC-Code für Zuweisung an Feldindex
	Abstrakter Syntaxbaum für Zuweisung an Feldindex
	PicoC-ANF Pass für Zuweisung an Feldindex
	RETI-Blocks Pass für Zuweisung an Feldindex
	PicoC-Code für die Deklaration eines Verbundstyps
	Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps
	Symboltabelle für die Deklaration eines Verbundstyps
	PicoC-Code für Initialisierung von Verbunden
	Abstrakter Syntaxbaum für Initialisierung von Verbunden
	PicoC-ANF Pass für Initialisierung von Verbunden
	RETI-Blocks Pass für Initialisierung von Verbunden
	PicoC-Code für Zugriff auf Verbundsattribut
	Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut
	PicoC-ANF Pass für Zugriff auf Verbundsattribut
	RETI-Blocks Pass für Zugriff auf Verbundsattribut
	PicoC-Code für Zuweisung an Verbundsattribut
1.47	Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut

1.48	PicoC-ANF Pass für Zuweisung an Verbundsattribut.	86
1.49	RETI-Blocks Pass für Zuweisung an Verbndsattribut	87
	PicoC-Code für den Anfangsteil.	90
	Abstrakter Syntaxbaum für den Anfangsteil.	91
	PicoC-ANF Pass für den Anfangsteil.	91
1.53	RETI-Blocks Pass für den Anfangsteil	92
1.54	PicoC-Code für den Mittelteil.	92
1.55	Abstrakter Syntaxbaum für den Mittelteil	93
	PicoC-ANF Pass für den Mittelteil	95
	RETI-Blocks Pass für den Mittelteil	97
	PicoC-Code für den Schlussteil	97
	Abstrakter Syntaxbaum für den Schlussteil	98
	PicoC-ANF Pass für den Schlussteil	99
	RETI-Blocks Pass für den Schlussteil	101
	PicoC-Code für 3 Funktionen	102
	Abstrakter Syntaxbaum für 3 Funktionen	103
	PicoC-Blocks Pass für 3 Funktionen	104
	PicoC-ANF Pass für 3 Funktionen	105
	RETI-Blocks Pass für 3 Funktionen	107
	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	107
	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	108
	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	109
	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	109
	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	110
	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	112
	PicoC-Code für Funktionsaufruf ohne Rückgabewert	112
	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert	113
	Symboltabelle für Funktionsaufruf ohne Rückgabewert.	116
	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert	117
	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	118
	RETI-Pass für Funktionsaufruf ohne Rückgabewert	120
	PicoC-Code für Funktionsaufruf mit Rückgabewert.	121
	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert	121
	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert	123
	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	124
	PicoC-Code für die Übergabe eines Feldes	125
	Symboltabelle für die Übergabe eines Feldes.	126
	PicoC-ANF Pass für die Übergabe eines Feldes	127
1.86	RETI-Block Pass für die Übergabe eines Feldes	129
	PicoC-Code für die Übergabe eines Verbundes.	129
	PicoC-ANF Pass für die Übergabe eines Verbundes.	130
1.80	RETI-Block Pass für die Übergabe eines Verbundes	139

Tabellenverzeichnis

1.1	Präzedenzregeln von PicoC
1.2	Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren
1.3	PicoC-Knoten Teil 1
1.4	PicoC-Knoten Teil 2
1.5	PicoC-Knoten Teil 3
1.6	PicoC-Knoten Teil 4
1.7	RETI-Knoten
1.8	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung
1.9	Datensegment nach der Initialisierung beider Felder
1.10	Ausschnitt des Datensegments nach der Initialisierung des Feldes in der main-Funktion 6
	Ausschnitt des Datensegments nach der Initialisierung des Feldes in der Funktion fun 6
	Ausschnitt des Datensegments bei der Adressberechnung 6
	Ausschnitt des Datensegments nach Schlussteil
	Ausschnitt des Datensegments nach Auswerten der rechten Seite
1.15	Ausschnitt des Datensegments vor Zuweisung
	Ausschnitt des Datensegments nach Zuweisung
	Datensegment mit Stackframe
1.18	Aufbau Stackframe

Definitionsverzeichnis

1.1	Metasyntax
1.2	Metasprache
1.3	Erweiterte Backus-Naur-Form (EBNF)
	Dialekt der Erweiterten Backus-Naur-Form aus Lark
1.5	Abstrakte Syntaxform (ASF)
1.6	Earley Parser
1.7	Entarteter Baum
1.8	Symboltabelle
1.9	Unterdatentyp
1.10	Stackframe

Grammatikverzeichnis

1.1.1 Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF	
1.2.1 Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in	
EBNF, die Operatorpräzidenz nicht beachtet	7
$1.2.2$ Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syn-	
taktische Analyse in EBNF, die Operatorpräzidenz beachtet	8
1.2.3 Beispiel für eine unäre rechtsassoziative Produktion in EBNF	9
1.2.4 Beispiel für eine unäre linksassoziative Produktion in EBNF	9
1.2.5 Beispiel für eine binäre linksassoziative Produktion in EBNF	10
1.2.6 Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF	10
$1.2.7$ Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet	11
1.2.8 Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1	12
1.2.9 Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	13
1.2.10Abstrakte Grammatik der Sprache L_{PiocC} in ASF	30
1.3.1 Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF	35
1 1000 25 0000	38
1.3.3 Abstrakte Grammatik der Sprache L_{PiocC_ANF} in ASF	42
1.3.4 Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF	46
1.3.5 Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF	50
1.3.6 Abstrakte Grammatik der Sprache L_{RETI} in ASF	54
1.3.7 Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF	54
1.3.8 Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF \dots	55

1 Implementierung

In diesem Kapitel wird, nachdem im Kapitel ?? die nötigen theoretischen Grundlagen des Compilerbau vermittelt wurden, nun auf die Implementierung des PicoC-Compilers eingegangen. Aufgeteilt in die selben Kategorien Lexikalische Analyse 1.1, Syntaktische Analyse 1.2 und Code Generierung 1.3, wie in Kapitel ??, 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 Toolkit² verwendet. Das Lark Parsing Toolkit ist eine Bibliothek, die es ermöglicht mittels einer in einem eigenen Dialekt der Erweiterten Back-Naur-Form (Definition 1.3 bzw. für den Dialekt von Lark Definition 1.4) spezifizierten Konkreten Grammatik ein Programm in Konkreter Syntax zu parsen und daraus einen Ableitungsbaum für die kompilerintere Weiterverarbeitung zu generieren.

Definition 1.1: Metasyntax

Z

Steht für den Aufbau einer Metasprache (Definition 1.2), der durch eine Grammatik oder Natürliche Sprache beschrieben werden kann.

Definition 1.2: Metasprache

1

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 1.3: Erweiterte Backus-Naur-Form (EBNF)



Die Erweiterte Backus-Naur-Form^a ist eine Metasyntax (Definition 1.1), 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 1.4 verwendet.

¹Wobei beim Parsen auch das Lexen inbegriffen ist.

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

³Shinan, lark.

Definition 1.4: Dialekt der Erweiterten Backus-Naur-Form aus Lark

Das Lark Parsing Toolkit verwendet eine eigene Notation für die Erweiterte Backus-Naur-Form (Definition 1.3), 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. ()*). Die Verwendung des *-Quantors kann wie in Umformung 1.0.1 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b*\} \quad \Rightarrow \quad \{a := b_tmp, \ b_tmp := b \ b_tmp \ \mid \ \varepsilon\} \tag{1.0.1}$$

• 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 +-Quantors kann wie in Umformung 1.0.2 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b+\} \quad \Rightarrow \quad \{a := b \ b_tmp, \ b_tmp := b \ b_tmp \mid \varepsilon\} \tag{1.0.2}$$

• 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 1.0.3 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a ::= "ascii1" ... "ascii2"\} \Rightarrow \{a ::= "ascii1" \mid ... \mid "ascii2"\}$$
 (1.0.3)

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. Lark besitzt des Weiteren eine sehr gute Dokumentation Welcome to Lark's documentation! — Lark documentation.

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 Unter-

^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.

^dAlso von Terminalsymbolen und Nicht-Terminalsymbolen.

ehttps://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_IS0_IEC_14977_1996(E).zip.

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

 $[^]b$ Der *-Quantor bedeutet im Gegensatz zum +-Quantor auch keinmal wiederholen.

 $^{^{4}}ANTLR$.

kapiteln die Abstrakten Grammatiken, welche spezifzieren, welche Kompositionen für die Abstrakten Syntaxbäume der verschiedenden Passes erlaubt sind in einer bewusst anderen Notation aufgeschrieben. Diese Notation hat allerdings Ähnlichkeit mit dem Dialekt der Erweiterten Backus-Naur-Form aus dem Lark Parsing Toolkit.

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. Hierdurch würde die Abstrakten Grammatiken nur unnötig verkompliziert, 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 1.5) 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 1.5: 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 1.4) 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 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.

1.1 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 ?? erläutert ist.

1.1.1 Konkrete Grammatik für die Lexikalische Analyse

In der Konkreten Grammatik 1.1.1 für die Lexikalische Analyse stehen großgeschriebene Nicht-Terminalsymbole entweder für einen Tokentyp oder einen Teil der Beschreibung des Aufbaus der zum Tokentyp gehörenden möglichen Tokenswerte. Zum Beispiel handelt es sich bei dem großgeschriebenen Nicht-Terminalsymbol NUM um einen Tokentyp, dessen zugeordnete mögliche Tokenwerte durch die Produktion NUM ::= "0" | DIG_NO_O DIG_WITH_O* beschrieben werden. Diese Produktionen beschreiben, wie ein möglicher Tokenwert, in diesem Fall eine Zahl aufgebaut sein kann.

Die in der Konkreten Grammatik 1.1.1 für die Lexikalische Analyse definierten Nicht-Terminalsymbole können in der Konkreten Grammatik 1.2.8 für die Syntaktischen Analayse verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl NUM stehen darf.

Die in der Konkreten Grammatik vereinzelt kleingeschriebenen Nicht-Terminalsymbole, wie z.B. name haben nur den Zweck mehrere Tokentypen, wie z.B. NAME | INT_NAME | CHAR_NAME unter einem Überbegriff zu sammeln.

In Lark steht eine Zahl .<number>, die an ein Nicht-Terminalsymbol angehängt ist (z.B. NONTERM.<number>), dass auf der linken Seite des ::=-Symbols einer Produktion steht für die Priorität der Produktion dieses Nicht-Terminalsymbols. Es wird immer die Produktion mit der höchste Priorität, also der höchsten Zahl <number> zuerst genommen.

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 setzen, 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. Ansonsten würden aus int_var die Tokens Token('INT_DT', 'int'), Token('NAME', '_var') generiert, anstatt dem TokenToken(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 Link⁵ 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 ?? erklärt wurde, nur berücksichtigt der Basic Lexer ebenfalls Priortiäten, sodass für den aktuellen Index⁶ im Eingabeprogramm zuerst alle Produktionen der höchsten Priorität geprüft werden. Sobald eine dieser Produktionen ein Lexeme an dem aktuellen Index im Eingabeprogramm ableiten kann, wird hieraus direkt ein Token mit dem entsprechenden Tokentyp dieser Produktion und dem abgeleiteten Tokenwert erstellt. Weitere Produktionen werden nicht mehr geprüft. Ansonsten werden alle Produktionen der nächstniedrigeren Priorität geprüft usw.

 $^{^5 \}text{https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.}$

⁶Ein Lexer bewegt sich über das Eingabeprogramm und erstellt, wenn ein Lexeme sich in der Konkreten Grammatik ableiten lässt ein Token und bewegt sich danach um die Länge des Lexemes viele Indices weiter.

```
/[\wedge \backslash n]*/
COMMENT
                                                  /(. | \n)*? / "*/"
                                                                           L_{-}Comment
                       ::=
                            "//""_{-}"?"#"/[\wedge \setminus n]*/
RETI\_COMMENT.2
                       ::=
                                   "2"
                                           "3"
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 1.1.1: Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF

1.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 1.1 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 1.1: PicoC-Code des Codebeispiels.

Die vom Basic Lexer des Lark Parsing Toolkit generierten Tokens sind Code 1.2 zu sehen.

Code 1.2: Tokens für das Codebeispiel.

1.2 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.

1.2.1 Umsetzung von Präzedenz und Assoziativität

In diesem Unterkapitel wird eine ähnliche Erklärweise, wie in dem Buch Nystrom, Parsing Expressions. Crafting Interpreters verwendet. Die Programmiersprache L_{PicoC} hat dieselben Präzedenzregeln implementiert, wie die Programmiersprache L_C . Die Präzedenzregeln sind von der Webseite C Operator Precedence - cppreference.com übernommen. Die Präzedenzregeln der verschiedenen Operatoren der Programmiersprache L_{PicoC} sind in Tabelle 1.1 aufgelistet.

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	Rechts, dann links \leftarrow
	*a &a	Dereferenz und Referenz, auch	neciits, daiii iiiks ←
		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	Bitweise Linksshift und Rechtsshift	
6	a <b a<="b</th"><th>Kleiner, Kleiner Gleich, Größer, Größer</th><th></th>	Kleiner, Kleiner Gleich, Größer, Größer	
	a>b a>=b	Gleich	
7	a==b a!=b	Gleichheit und Ungleichheit	Links, dann rechts \rightarrow
8	a&b	Bitweise UND	
9	a^b	Bitweise XOR (exclusive or)	
10	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 1.1: Präzedenzregeln von PicoC.

Würde man diese Operatoren ohne Beachtung von Präzedenzreglen (Definition ??) und Assoziativität (Definition ??) in eine Konkrete Grammatik verarbeiten wollen, so könnte eine Konkrete Grammatik $G = \langle N, \Sigma, P, exp \rangle$ 1.2.1 dabei rauskommen.

```
NUM
                                      "'"CHAR"'"
                                                        "("exp")"
                                                                                    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 1.2.1: Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet

Die Konkrete Grammatik 1.2.1 ist allerdings mehrdeutig (Definition ??), 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 1.2.1 als auch über die Linksableitung 1.2.2 abgeleitet werden. Ab dem Moment, wo der Trick klar ist, wird das Ableiten mit der ⇒*-Relation beschleunigt.

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

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

```
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\_op \ exp \Rightarrow "3" "*" "1" "&&" "4"
```

Die beiden abgeleiteten Wörter sind gleich, allerdings sind die Ableitungsbäume unterschiedlich, wie in Abbildung 1.1 zu sehen ist. Da hier nur ein Konzept vermittelt werden soll, entsprechen die beiden Ableitungsbäume in Abbildung 1.1 nicht 1-zu-1 den Ableitungen 1.2.1 und 1.2.2, sondern sind vereinfacht.



Abbildung 1.1: Ableitungsbäume zu den beiden Ableitungen.

Der linke Baum entspricht Ableitung 1.2.1 und der rechte Baum entspricht Ableitung 1.2.2. Würde man in den Ausdrücken, die von diesen Bäumen darsgestellt sind Klammern setzen, um die Präzedenz sichtbar zu machen, so würde Ableitung 1.2.1 die Klammerung (3 * 1) & 4 haben und die Ableitung 1.2.2 die Klammerung 3 * (1 & 4) haben. Es ist wichtig die Präzedenzregeln und die Assoziativität von Operatoren beim Erstellen der Konkreten Grammatik miteinzubeziehen, da das Ergebnis des gleichen Ausdrucks sich bei unterschiedlicher Klammerung unterscheiden kann.

Hierzu wird nun Tabelle 1.1 betrachtet. Für jede **Präzedenzstufe** in der Tabelle 1.1 wird eine eigene Produktion erstellt, wie es in Grammatik 1.2.2 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.

```
L_Arith_Bit + L_Array
prim_{-}exp
                 ::=
                              + L_-Pntr + L_-Struct
post\_exp
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
                 ::=
                       . . .
assign\_stmt
                              L\_Assign
                 ::=
```

Grammatik 1.2.2: 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 ::=-Symbols in Grammatik 1.2.2 sind in Tabelle 1.2 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
$\operatorname{arith_and}$	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 1.2: 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)⁷ 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.

Im Folgenden werden Produktionen für alle relevanten Fälle von verschiedenen Kombinationen von Präzedenzen und Assoziativitäten erklärt. Bei z.B. der Produktion un_exp in 1.2.3 für die rechtsassoziativen unären Operatoren -a, !a ~a, *a und &a ist die Alternative un_op un_exp dafür zuständig, dass diese unären Operatoren rechtsassoziativ geschachtelt werden können (z.B. !-~42). Die Alternative post_exp ist dafür zuständig, dass die Produktion beim Ableiten 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 1.2.3: Beispiel für eine unäre rechtsassoziative Produktion in EBNF

Bei z.B. der Produktion post_exp in 1.2.4 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.

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

Grammatik 1.2.4: Beispiel für eine unäre linksassoziative Produktion in EBNF

⁷Geklammerte Ausdrücke werden nämlich von prim_exp erkannt, welches eine höhere Präzedenzstufe hat.

Bei z.B. der Produktion prec2_exp in 1.2.5 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 können⁸ (z.B. 3 + 1 - 4, wobei - und + beide Präzedenzstufe 4 haben). Die Alternative 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 1.2.5: Beispiel für eine binäre linksassoziative Produktion in EBNF

Anmerkung Q

Manche Parser^a haben allerdings ein Problem mit Linksrekursion (Definition ??), wie sie z.B. in der Produktion 1.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 1.2.5 zur Produktion 1.2.6 umschreibt.

```
arith\_prec2 ::= arith\_prec1 (prec2\_op arith\_prec1)*
```

Grammatik 1.2.6: Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF

Die von der Grammatik 1.2.6 erkannten Ausdrücke sind dieselben, wie für die Grammatik 1.2.5, allerdings ist die Grammatik 1.2.6 flach gehalten und ruft sich nicht selber auf, sondern nutzt den in der EBNF (Definition 1.3) 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 *-Quantor 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).

^aZu diesen Parsern zählt der Earley Parser, der im PicoC-Compiler verwendet wird glücklicherweise 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()). Mithilfe dieser Paradigmen lässt sich die Konkrete Grammatik 1.2.7 definieren.

⁸Bezogen auf Tabelle 1.1.

```
"*"
                                                                                                 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 "\land" arith\_and
arith\_xor
                                                 | 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_or "||" logic_and
                                                  logic\_and
logic\_or
               ::=
                     un_exp "=" logic_or";"
assign\_stmt
               ::=
                                                                                                 L_Assign
```

Grammatik 1.2.7: Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet

1.2.2 Konkrete Grammatik für die Syntaktische Analyse

Die gesamte Konkrete Grammatik 1.2.8 ergibt sich wenn man die Konkrete Grammatik 1.2.7 um die restliche Syntax der Sprache L_{PicoC} erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 1.2.1 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 $ANSI\ C\ grammar\ (Lex)$ angepasst⁹. Auf diese Weise konnte sicherer gewährleistet werden kann, dass der PicoC-Compiler sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C^{10} .

In der Konkreten Grammatik 1.2.8 für die Syntaktische Analyse werden einige der Nicht-Terminalsymbole bzw. Tokentypen aus der Konkreten Grammatik 1.1.1 für die Lexikalischen Analyse verwendet, wie z.B. NUM. Es werde aber auch Produktionen, wie name verwendet, die mehrere Tokentypen unter einem Überbegriff zusammenfassen.

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 1.2.8 für die Syntaktische Analyse zu schreiben. Der Tokentyp für diese Terminalsymbole wird in diesem Fall vom Lark Parsing Toolkit bestimmt, welches einige sehr häufig verwendete Terminalsymbole, wie z.B.; oder && bereits einen eigenen Tokentyp zugewiesen hat.

 $^{^9}$ An der für die Programmiersprache L_{PicoC} relevanten Syntax hat sich allerdings über die Jahre nichts wichtiges 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.

¹⁰Wobei z.B. die Compiler GCC (GCC, the GNU Compiler Collection - GNU Project) und Clang (clang: C++ Compiler) zu nennen wären.

Diese Terminalsymbole werden aber weiterhin vom Basic Lexer als Teil der Lexikalischen Analyse generiert.

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
$\frac{struct_attr}{if_stmt}$ 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 1.2.8: 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 1.2.9: Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2

Anmerkung Q

In der Konkreten Grammatik 1.2.8 sind alle Grammatiksymbole ausgegraut, die das Bachelorprojekt betreffen. Alle nicht ausgegrauten Grammatiksymbole wurden für die Implementierung der neuen Funktionalitäten, welche die Bachelorarbeit betreffen hinzugefügt.

1.2.3 Ableitungsbaum Generierung

Die in Unterkapitel 1.2.2 definierte Konkrete Grammatik 1.2.8 lässt sich mithilfe des Earley Parsers (Definition 1.6) von Lark dazu verwenden, Code, der in der Sprache L_{PicoC} geschrieben ist zu parsen, um einen Ableitungsbaum daraus zu generieren.

Definition 1.6: Earley Parser

Ist ein Algorithmus für das Parsen von Wörtern einer Kontextfreien Sprache. Der Earley Parser ist ein Chart Parser ist, welcher einen mittels Dynamischer Programmierung und Top-Down Ansatz arbeitenden Earley Erkenner (Defintion ?? im ??) 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 genommenen Ableitungen wieder nachvollziehen zu können und so einen Ableitungsbaum konstruieren zu können.^a

^aEarley, "An efficient context-free parsing".

1.2.3.1 Codebeispiel

Der Ableitungsbaum, der mithilfe des Earley Parsers und der Tokens der Lexikalischen Analyse in Code 1.2 generiert wurde, ist in Code 1.3 zu sehen. Das Beispiel aus Code 1.1 wird hier fortgeführt. Im Code 1.3 wurden einige Zeilen markiert, die später in Unterkapitel 1.2.4.1 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
                        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 1.3: Ableitungsbaum nach Ableitungsbaum Generierung.

1.2.3.2 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 1.2.8 den linken Seiten des ::=-Symbols¹¹ entsprechen hergenommen und die Blätter sind Terminalsymbole, genauso, wie es in der Definition ?? eines Ableitungsbaumes auch schon definiert ist. Die Konkrete Grammatik 1.2.8 des PicoC-Compilers erlaubt es 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.

1.2.4 Ableitungsbaum Vereinfachung

Der Ableitungsbaum in Code 1.3, dessen Generierung in Unterkapitel 1.2.3.1 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 der Programmiersprache L_{PicoC} korrekt bestimmen zu können die Spiralregel $Clockwise/Spiral\ Rule$ in der Implementeirung des PicoC-Compilers umgesetzt werden muss. Dies ist allerdings nicht alleinig möglich, indem man die entsprechenden Produktionen in der Konkreten Grammatik 1.2.8 auf eine spezielle Weise passend spezifiziert. Der PicoC-Compiler soll in der Lage sein den Ausdruck int (*ar[3]) [2] als "Feld der Mächtigkeit 3 von Zeigern auf Felder der Mächtigkeit 2 von Integern" erkennen zu können.

Was man erhalten will, ist ein entarteter Baum (Definition 1.7) von PicoC-Knoten, an dem man den Datentyp direkt ablesen kann, indem man sich einfach über den entarteten Baum bewegt, wie z.B. P ntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')], PntrDecl(Num('1'), StructSpec(Name('st'))))) für den Ausdruck struct st *(*var[3][2]).

Definition 1.7: Entarteter Baum

Z

 $Baum\ bei\ dem\ jeder\ Knoten\ maximal\ eine\ ausgehende\ Kante\ hat,\ also\ maximal\ Außengrad^a\ 1.$

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

Der Baum entspricht also einer verketteten Liste.c

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 1.2 zu sehen ist.

^aDer Außengrad ist die Anzahl ausgehender Kanten.

 $[^]b\mathrm{Der}$ Innengrad ist die Anzahl eingehener Kanten.

 $[^]cB\ddot{a}ume.$

¹¹ Grammar: The language of languages (BNF, EBNF, ABNF and more).



Abbildung 1.2: 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 z.B. 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 wird bei allen Felddeklaratoren array_decl(datatype, nums) immer das erste Attribut datatype mit dem zweiten Attribut nums getauscht.

Des Weiteren befindet sich in der Mitte der Spirale, die der Ableitungsbaum bildet der Name der Variable name(var) und nicht der innerste Datentyp struct st. Das liegt daran, dass der Ableitungsbaum einfach nur die kompilerinterne Darstellung, die durch das Parsen eines Ausdrucks in Konkreter Syntax (z.B. struct st *(*var[3][2])) generiert wird darstellt. Der Knoten für den Bezeichner der Variable name(var) wird daher mithilfe eines Visitors (Definition ??) mit dem Knoten für den innersten Datentyp struct st getauscht.

Eine Änderung, die eher der Ästhetik dient, ist zusätzlich den Teilbaum, der den Datentyp darstellt mit dem Knoten name(var) zu tauschen. Das hat den Grund, dass der Datentyp üblicherweise vor dem Bezeichner der Variable steht: datatype identifier¹² und im vorherigen Schritt name(var) vor den Datentyp getauscht wurde. In Abbildung 1.3 ist zu sehen, wie der Ableitungsbaum aus Abbildung 1.2 mithilfe eines Visitors vereinfacht wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des Visitors aus dem Lark Parsing Toolkit ist unter Link¹³ 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 ?? erklärt wurde.

¹²Wie z.B. bei int var.

 $^{^{13} \}texttt{https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.}$



Abbildung 1.3: Ableitungsbaum nach Vereinfachung.

1.2.4.1 Codebeispiel

In Code 1.4 ist der Ableitungsbaum aus Code 1.3 nach der Vereinfachung mithilfe eines Visitors zu sehen. Das Beispiel aus Code 1.1 wird hier fortgeführt.

```
file
     ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
     decls_defs
 4
5
       decl_def
         struct_decl
           name
                        st
           struct_params
 8
9
             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
                       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 1.4: Ableitungsbaum nach Ableitungsbaum Vereinfachung.

1.2.5 Generierung des Abstrakten Syntaxbaumes

Nachdem der Ableitungsbaum in Unterkapitel 1.2.4 vereinfacht wurde, ist der vereinfachte Ableitungsbaum in Code 1.4 nun dazu geeignet, um mit einem Transformer (Definition ??) einen Abstrakten Syntaxbaum aus ihm zu generieren. Würde man den vereinfachten Ableitungsbaum des Ausdrucks struct st *(*var[3][2]) auf die übliche Weise in einen entsprechenden Abstrakten Syntaxbaum umwandeln, so würde dabei ein Abstrakter Syntaxbaum wie in Abbildung 1.4 rauskommen.

Die Implementierung des Transformers aus dem Lark Parsing Toolkit ist unter Link¹⁴ 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 ?? erklärt wurde.

Den Teilbaum, der rechts in Abbildung 1.4 den Datentyp darstellt, würde man von oben-nach-unten¹⁵ als "Zeiger auf einen Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs st" lesen. Man würde es also bis auf die Dimensionen der Felder, bei denen es freisteht, wie man sie liest genau anders herum lesen, 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.

 $^{^{14} \}verb|https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.$

¹⁵In der Informatik wachsen Bäume von oben-nach-unten, von der Wurzel zur den Blättern, bzw. in diesem Beispiel von links-nach-rechts.



Abbildung 1.4: Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen.

Der Abstrakte Syntaxbaum rechts in Abbildung 1.4 ist für die Weiterverarbeitung also ungeeignet, denn für die Adressberechnung bei einer Aneinanderreihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute¹⁶ 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 vollständigen Datentyp steht umgedreht wird. Auf diese Weise kommt ein Abstrakter Syntaxbaum mit richtig rum gedrehtem Datentyp, wie rechts in Abbildung 1.5 zustande, der für die Weiterverarbeitung geeignet ist.

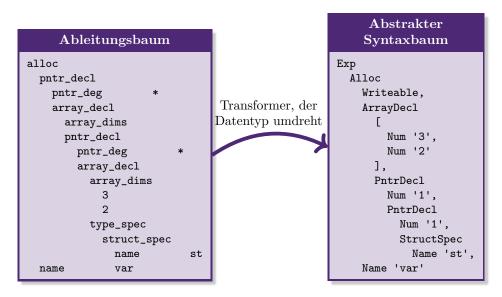


Abbildung 1.5: Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.

Die Weiterverarbeitung des Abstrakten Syntaxbaumes geschieht mithilfe von Passes, welche im Unterkapitel 1.3 genauer beschrieben werden. Da die Knoten des Abstrakten Syntaxbaumes anders als beim

¹⁶Welche in Unterkapitel 1.3.5.2 genauer erläutert wird

Ableitungsbaum nicht die gleichen Bezeichnungen haben, wie Nicht-Terminalsymbole der Konkreten Grammatik, ist es in den folgenden Unterkapiteln 1.2.5.1, 1.2.5.2 und 1.2.5.3 notwendig die Bedeutung der einzelnen PicoC-Knoten, RETI-Knoten und bestimmter Kompositionen dieser Knoten zu dokumentieren. Diese Knoten kommen später im Unterkapitel 1.3 in den unterschiedlichen von den Passes umgeformten Abstrakten Syntaxbäumen vor.

Des Weiteren gibt die Abstrakte Grammatik 1.2.10 in Unterkapitel 1.2.5.4 Aufschluss darüber welche Kompositionen von PicoC-Knoten neben den bereits in Tabelle 1.8 definierten Kompositionen mit Bedeutung insgesamt überhaupt möglich sind.

1.2.5.1 PicoC-Knoten

Bei den PicoC-Knoten handelt es sich um Knoten, die wenn man die Programmiersprache L_{PicoC} auf das herunterbricht, was wirklich entscheidend ist ein abstraktes Konstrukt darstellen, welches z.B. ein Container für andere Knoten sein kann oder einen der ursprünglichen Token darstellt. Diese Abstrakten Konstrukte sollen allerdings immer noch etwas aus der Programmiersprache 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 sollte¹⁷. Alle PicoC-Knoten, die in den von den verschiedenen Passes generierten Abstrakten Syntaxbäumen vorkommen sind in Tabelle 1.3 mit einem Beschreibungstext dokumentiert.

¹⁷Z.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(str)	Repräsentiert einen Bezeichner (z.B. my_fun, my_var usw.). Das Attribut str ist eine Zeichenkette, welche beliebige Groß- und Kleinbuchstaben (a - z, A - Z), Zahlen (0 - 9) und den Unterstrich (_) enthalten kann. An erster Stelle darf allerdings keine Zahl stehen.
Num(str)	Eine Zahl (z.B. 42, -3 usw.). Hierbei ist das Attribut str eine Zeichenkette, welche beliebige Ziffern (0 - 9) enthalten kann. Es darf nur nicht eine 0 am Anfang stehen und danach weitere Ziffern folgen. Der Wert, welcher durch die Zeichenkette dargestellt wird, darf nicht größer als $2^{32} - 1$ sein.
Char(str)	Ein Zeichen ('c', '*' usw.). Das Attribut str ist ein Zeichnen der ASCII-Zeichenkodierung.
<pre>Minus(), Not(), DerefOp(), RefOp(), LogicNot()</pre>	Die unären Operatoren un_op: -a, ~a, *a, &a !a.
Add(), Sub(), Mul(), Div(), Mod(), LShift(), RShift(), 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 \land 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 Basisdatentypen: int, char, void. Um eine intuitive Bezeichnung zu haben werden sie in der Abstrakten Grammatik einfach nur als Datentypen datatype eingeordnet werden.
Placeholder()	Platzhalter für einen Knoten, der diesen später ersetzt.
BinOp(exp, bin_op, exp)	Container für eine binäre Operation: <exp1> <bin_op> <exp2>.</exp2></bin_op></exp1>
UnOp(un_op, exp)	Container für eine unäre Operation: <un_op> <exp>.</exp></un_op>
Atom(exp, rel, exp)	Container für eine binäre Relation: <exp1> <rel> <exp2></exp2></rel></exp1>
ToBool(exp)	Container für einen Arithmetischen oder Bitweise Ausdruck, wie z.B. 1 + 3 oder einfach nur 3. Aufgrund des Kontext in dem sich dieser Ausdruck befindet, wird bei einem Ergebnis $x > 0$ auf 1 abgebildet und bei $x = 0$ auf 0.
<pre>Alloc(type_qual, datatype, name, local_var_or_param)</pre>	Container für eine Allokation <type_qual> <datatype> <name> mit den Attributen type_qual, datatype und name, die alle Informationen für einen Eintrag in der Symboltabelle 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(exp1, exp2)	Container für eine Zuweisung exp1 = exp2, wobei exp1 z.B. ein Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name) oder Name('var') sein kann und exp2 ein beliebiger Logischer Ausdruck sein kann.

Tabelle 1.3: PicoC-Knoten Teil 1.

PiocC-Knoten	Beschreibung
<pre>Exp(exp, datatype, error_data)</pre>	Container für einen beliebigen Ausdruck, dessen Ergebnis
	auf den Stack geschrieben werden soll. Zudem besitzt er 2
	versteckte Attribute, wobei datatype einen Datentyp trans-
	portiert, der für den RETI Blocks Pass wichtig ist und
~	error_data für Fehlermeldungen wichtig ist.
Stack(num)	Holt sich z.B. für eine Berechnung einen zuvor auf den Stack
	geschriebenen Wert vom Stack, der num Speicherzellen relativ
	zum SP-Register steht.
Stackframe(num)	Holt sich den Wert einer Variable, eines Verbundsattri-
	buts, eines Feldelements etc., der $num + 2$ Speicherzellen
	relativ zum BAF-Register steht.
Global(num)	Holt sich den Wert einer Variable, eines Verbundsattri-
	buts, eines Feldelements etc., der num Speicherzellen relativ
	zum DS-Register steht.
StackMalloc(num)	Steht für das Allokieren von num Speicherzellen auf dem
	Stack.
PntrDecl(num, datatype)	Container, der für den Zeigerdatentyp steht: <prim_dt></prim_dt>
•	* <var>. Hierbei gibt das Attribut num die Anzahl zusam-</var>
	mengefasster Zeiger an und datatype ist der Datentyp,
	auf den der letzte dieser Zeiger zeigt.
Ref(exp, datatype, error_data)	Steht für die Anwendung des Referenz-Operators & <var>,</var>
Nor (onp, datasyps, orrer_data,	der die Adresse einer Location (Definition ??) auf den
	Stack schreiben soll, die über exp bestimmt ist. Zudem besitzt
	er 2 versteckte Attribute, wobei datatype einen Datentyp
	transportiert, der für den RETI Blocks Pass wichtig ist
D ((1 0)	und error data für Fehlermeldungen wichtig ist.
Deref(exp1, exp2)	Container für den Indexzugriff auf einen Feld- oder Zeiger-
	datentyp: <var>[<i>]. Hierbei ist exp1 ein angehängtes weite-</i></var>
	res Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name)
	oder Name('var') und exp2 ist der Index auf den zugegriffen
, , , , , , , , , , , , , , , , , , , ,	werden soll.
ArrayDecl(nums, datatype)	Container, der für den Felddatentyp steht: <prim_dt></prim_dt>
	<pre><var>[<i>]. Hierbei ist das Attribut nums eine Liste von</i></var></pre>
	Num('x'), welche die Dimensionen des Felds angibt und
	datatype ist ein Unterdatentyp (Definition 1.9).
Array(exps, datatype)	Container für den Initialisierer eines Feldes, z.B. {{1, 2},
	{3, 4}}, dessen Attribut exps weitere Initialisierer für ein
	Feld, weitere Initialisierer für einen Verbund oder Logische
	Ausdrücke beinhalten kann. Des Weiteren besitzt es 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 Zeiger-
• • •	datentyp: <var>[<i>]. Hierbei ist exp1 ein angehängtes weite-</i></var>
	res Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name)
	oder Name('var') und exp2 ist der Index auf den zugegriffen
	werden soll.
StructSpec(name)	Container für die Spezifikation eines Verbundstyps: struct
Dor ac cobec (name)	< name >. Hierbei legt das Attribut name fest, welchen selbst
A++()	definierten Verbundstyp dieser Knoten spezifiziert.
Attr(exp, name)	Container für den Zugriff auf ein Verbundsattribut:
	<pre><var>>.<attr>>. Hierbei kann exp eine angehängte weitere</attr></var></pre>
	Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name)
	Operation oder ein Name('var') sein. Das Attribut name ist
	das Verbundsattribut auf das zugegriffen werden soll.

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initialisierer eines Verbundes, z.B {. <a attr2="" href="attr1>={1,2},.={3,4}">attr1>={1,2},.={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">attr2>={3,4}">att
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Verbundstyps, z.B. struct <var> {<datatype> <attr1>; <datat ype=""> <attr2>;};. Hierbei ist name der Bezeichner des Verbundstyps und allocs eine Liste von Bezeichnern der Verbundsattribute mit dazugehörigem Datentyp, wofür sich der Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.</attr2></datat></attr1></datatype></var>
If(exp, stmts)	Container für eine If-Anweisung if(<exp>) { <stmts> } in- klusive Bedingung exp und einem Branch stmts, indem eine Liste von Anweisungen steht.</stmts></exp>
<pre>IfElse(exp, stmts1, stmts2)</pre>	Container für eine If-Else Anweisung if (<exp>) { <stmts1> } else { <stmts2> } inklusive Bedingung exp und 2 Branches stmts1 und stmts2, die zwei Alternativen Darstellen in denen jeweils Listen von Anweisungen stehen.</stmts2></stmts1></exp>
While(exp, stmts)	Container für eine While-Anweisung while(<exp>) { <stmts> } inklusive Bedingung exp und einem Branch stmts, indem eine Liste von Anweisungen steht.</stmts></exp>
DoWhile(exp, stmts)	Container für eine Do-While-Anweisung do { <stmts> } while(<exp>); inklusive Bedingung exp und einem Branch stmts, indem eine Liste von Anweisungen steht.</exp></stmts>
Call(name, exps)	Container für einen Funktionsaufruf fun_name(exps), wobei name der Bezeichner der Funktion fun_name ist, die aufgerufen werden soll und exps eine Liste von Argumenten ist, die an diese Funktion übergeben werden soll.
Return(exp)	Container für eine Return-Anweisung return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis von der Return-Anweisung zurückgegeben wird.</exp>
FunDecl(datatype, name, allocs)	Container für eine Funktionsdeklaration datatype <param1>, datatype <param2>), wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist und allocs die Parameter der Funktion sind. Der Knoten Alloc(type_spec, datatype, name) dient dabei als Container für die Parameter in allocs.</param2></param1>

Tabelle 1.5: PicoC-Knoten Teil 3.

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs,	Container für eine Funktionsdefinition <datatype></datatype>
stmts_blocks)	<pre><fun_name>(<datatype> <param/>) {<stmts>}, wobei datatype</stmts></datatype></fun_name></pre>
20002-210012,	der Rückgabewert der Funktion ist, name der Bezeichner
	der Funktion ist, allocs die Parameter der Funktion
	•
	sind und stmts_blocks eine Liste von Statemetns bzw.
	Blöcken ist, welche diese Funktion beinhaltet. Der Knoten
	Alloc(type_spec, datatype, name) dient dabei als Container
	für die Parameter in allocs.
<pre>NewStackframe(fun_name,</pre>	Erstellt einen neuen Stackframe und speichert den Wert
<pre>goto_after_call)</pre>	des BAF-Registers der aufrufenden Funktion und die
	Rücksprungadresse nacheinander an den Anfang des neu-
	en Stackframes. Das Attribut fun_name stehte dabei für den
	Bezeichner der Funktion, für die ein neuer Stackframe er-
	stellt 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 enthält das Attribut
	goto_after_call ein GoTo(Name('addr@next_instr')), welches
	später durch die Adresse des Befehls, der direkt auf den
	-
RemoveStackframe()	Sprungbefehl folgt, ersetzt wird. Container für das Entfernen des aktuellen Stackframes,
RemoveStackIrame()	
	durch das Wiederherstellen des im noch aktuellen Stack-
	frame gespeicherten Werts des BAF-Registes der aufrufenden
	Funktion und das Setzen des SP-Registers auf den Wert des
	BAF-Registers vor der Wiederherstellung.
File(name, decls_defs_blocks)	Container der eine Datei repräsentiert, wobei name der Da-
	teiname der Datei ist und decls_defs_blocks eine Liste von
	Funktionen bzw. Blöcken ist.
<pre>Block(name, stmts_instrs, instrs_before,</pre>	Container für Anweisungen, wobei das Attribut name der
<pre>num_instrs, param_size, local_vars_size)</pre>	Bezeichner des Labels (Definition ??) des Blocks ist und
	stmts_instrs eine Liste von Anweisungen oder Befeh-
	len ist. Zudem besitzt er noch 4 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 vor-
	aussichtliche Anzahl an Speicherzellen aufaddiert, die für
	die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für einen Sprung zu einem anderen Block durch
22.20 (2101110)	Angabe des Bezeichners des Labels des Blocks, wobei das
	Attribut name der Bezeichner des Labels des Blocks ist, zu
Cincle incomment (profine	dem Gesprungen werden soll. Container für einen Kommentar (//, /* <comment> */), den</comment>
SingleLineComment(prefix, content)	
	der Compiler selbst während des Kompiliervorgangs er-
	stellt, damit die Zwischenschritte der Kompilierung und
	auch der finale RETI-Code bei Betrachtung ausgegebener
	Abstrakter Syntaxbäume der verschiedenen Passes leich-
	ter verständlich sind.
RETIComment(str)	Container für einen Kommentar im Code der Form: // #
	comment, der im RETI-Intepreter später sichtbar ist und
	zur Orientierung genutzt werden kann. So ein Kommentar
	wäre allerdings in einer tatsächlichen Implementierung einer
	RETI-CPU nicht umsetzbar und eine Umsetzung wäre auch
	nicht sinnvoll. Der Kommentar ist im Attribut str, welches
	jeder Knoten besitzt gespeichert.
	U I

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, um im weiteren Kompiliervorgang Informationen zu transportieren. Das sind Informationen, die später im Kompiliervorgang nicht mehr so leicht zugänglich sind, wie zu dem Zeitpunkt, zu dem sie zugewiesen werden.

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.

1.2.5.2 RETI-Knoten

Bei den RETI-Knoten handelt es sich um Knoten, die irgendeinen einen Bestandteil eines Befehls aus der Sprache L_{RETI} darstellen. Für die RETI-Knoten wurden aus bereits in Unterkapitel 1.2.5.1 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 1.2.5.1 mit einem Beschreibungstext dokumentiert.

RETI-Knoten	Beschreibung
Program(name, instrs)	Container der ein Programm repräsentiert: <name></name>
-	<instrs>. Hierbei ist name der Name des Programms,</instrs>
	welches ausgeführt werden soll und instrs ist eine Liste
	von Befehlen.
<pre>Instr(op, args)</pre>	Container für einen Befehl: <op> <args>. Hierbei ist op</args></op>
	eine Operation und args eine Liste von Argumenten
	für diese Operation.
Jump(rel, im_goto)	Container für einen Sprungbefehl: JUMP <rel> <im>. Hier-</im></rel>
	bei ist rel eine Relation und im_goto ist ein Immediate
	Value Im(str) für die Anzahl an Speicherzellen, um
	die relativ zum Sprungbefehl gesprungen werden soll.
	In einigen Fällen ist im ein GoTo(Name('block.xyz')), das
	später im RETI-Patch Pass durch einen passenden Im-
	mediate Value ersetzt wird.
Int(num)	Container für den Aufruf einer Interrupt-Service-
	Routine: INT <im>. Hierbei ist num die Interrruptvek-</im>
	tornummer (IVN) für die passende Adresse in der Inter-
	ruptvektortabelle, in der die Adresse der Interrupt-
	Service-Routine (ISR) steht, die man aufrufen will.
Call(name, reg)	Container für einen Prozeduraufruf: CALL <name> <reg>.</reg></name>
	Hierbei ist name der Bezeichner der Prozedur, die auf-
	gerufen werden soll und reg ist ein Register, das als
	Argument an die Prozedur dient. Diese Operation ist
	in der Betriebssysteme Vorlesung ^a nicht deklariert, son-
	dern wurde hinzugefügt, um unkompliziert ein CALL PRINT
	ACC oder CALL INPUT ACC im RETI-Interpreter simulieren
	zu können.
Name(str)	Bezeichner für eine Prozedur, z.B. print, input oder
	den Programnamen, z.B. PROGRAMNAME. Hierbei ist str
	eine Zeichenkette für welche das gleich gilt, wie für
	das str Attribut des PicoC-Knoten Name(str). Dieses
	Argument ist in der Betriebssysteme Vorlesung ^a nicht
	deklariert, sondern wurde hinzugefügt, um Bezeichner,
	wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein Register. Hierbei ist reg das Register
	auf welches zugegriffen werden soll.
Im(str)	Ein Immediate Value (z.B. 42, -3 usw.). Hierbei ist das
	Attribut str eine Zeichenkette, welche beliebige Ziffern
	(0 - 9) enthalten kann. Es darf nur nicht eine 0 am Anfang
	stehen und danach weitere Ziffern folgen. Der Wert,
	welcher durch die Zeichenkette dargestellt wird, darf nicht
	kleiner als -2^{21} oder größer als $2^{21} - 1$ sein.
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 rel: <, <=, >, >=, ==, !=, _NOP.
Always(), NOp()	
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(),	Register reg: PC, IN1, IN2, ACC, SP, BAF, CS, DS.
Cs(), Ds()	

^a Scholl, "Betriebssysteme"

1.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung

In Tabelle 1.8 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') Speicherzellen relativ zum DS-Register steht auf den Stack.
Ref(Stackframe(Num('addr')))	Speichert Adresse der Speicherzelle, die Num $('addr') + 2$ Speicherzellen relativ zum BAF-Register steht auf den Stack.
<pre>Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))), datatype)</pre>	Berechnet die nächste Adresse aus der Adresse, die an Speicherzelle Stack(Num('addr1')) steht und dem Index, der an Speicherzelle Stack(Num('addr2')) steht und speichert diese auf den Stack. Die Berechnung ist abhängig davon, ob der Datentyp im versteckten Attribut datatype ein ArrayDecl(datatype) oder PntrDecl(datatype) ist.
<pre>Ref(Attr(Stack(Num('addr1')), Name('attr')), datatype)</pre>	Berechnet die nächste Adresse aus der Adresse, die an Speicherzelle Stack(Num('addr1')) steht und dem Attributnamen Name('attr') und speichert diese auf den Stack. Zur Berechnung ist der Name Name('st') des Verbundstyps in StructSpec(Name('st')) notwendig mit dem diese Berechnung durchgeführt wird. Dieser Verbundstyp ist im versteckten Attribut datatype zu finden. Dabei muss dieser Datentyp im versteckten Attribut datatype ein StructSpec(name) sein, da diese Berechnung nur bei einem Verbund durchgeführt werden kann.
<pre>Assign(Stack(Num('size'))), Global(Num('addr')))</pre>	Schreibt Num('size') viele Speicherzellen, die Num('add r') viele Speicherzellen relativ zum DS-Register stehen, versetzt genauso auf den Stack.
<pre>Assign(Stack(Num('size')), Stackframe(Num('addr')))</pre>	Schreibt Num('size') viele Speicherzellen, die Num('addr') + 2 viele Speicherzellen relativ zum BAF-Register stehen, versetzt genauso auf den Stack.
<pre>Assign(Stack(Num('addr1')), Stack(Num('addr2')))</pre>	Speichert Inhalt der Speicherzelle, die Num('addr2') Speicherzellen relativ zum SP-Register steht an der Adresse in der Speicherzelle, die Num('addr1') Speicherzellen relativ zum SP-Register steht.
<pre>Assign(Global(Num('addr')), Stack(Num('size')))</pre>	Schreibt Num('size') viele Speicherzellen, die auf dem Stack stehen, versetzt zu den Globalen Statischen Daten ab einer Num('addr') viele Speicherzellen relativ zum DS-Register liegenden Adresse.
<pre>Assign(Stackframe(Num('addr')), Stack(Num('size')))</pre>	Schreibt Num('size') viele Speicherzellen, die auf dem Stack stehen, versetzt in den Stackframe der momentan aktiven Funktion ab einer Num('addr') viele Speicherzellen relativ zum BAF-Register liegenden Adresse.
<pre>Exp(Global(Num('addr'))</pre>	Speichert Inhalt der Speicherzelle, die Num('addr') Speicherzellen relativ zum DS-Register steht auf den Stack.
<pre>Exp(Stackframe(Num('addr'))</pre>	Speichert Inhalt der Speicherzelle, die $Num('addr') + 2$ Speicherzellen relativ zum BAF-Register steht auf den Stack.
<pre>Exp(Stack(Num('addr')))</pre>	Speichert Inhalt der Speicherzelle an der Adresse, die in der Speicherzelle gespeichert ist, die Num('addr') viele Speicherzellen relativ zum SP-Register liegt auf den Stack.
<pre>Exp(Reg(reg))</pre>	Schreibt den aktuellen Wert des Registers reg auf den Stack.
<pre>Instr(Loadi(), [Reg(reg), GoTo(Name('addr@next_instr'))])</pre>	Lädt in das reg-Register die Adresse des Befehls, der direkt nach dem Sprung zum Block einer anderen Funktion steht.

Tabelle 1.8: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung.

Anmerkung Q

Um die obige Tabelle 1.8 nicht mit unnötig viel repetetiven Inhalt zu füllen, wurden die zahlreichen Kompositionen ausgelassen, bei denen einfach nur ein $\exp i, i := "1" \mid "2" \mid \varepsilon$ durch $\operatorname{Stack}(\operatorname{Num}('x')), x \in \mathbb{N}$ ersetzt wurde^a.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen nur ein Ausdruck an ein Exp(exp) bzw. Ref(exp) drangehängt wurde^b.

```
{}^a\mathrm{Wie} z.B. bei BinOp(Stack(Num('2')), Add(), Stack(Num('1'))). {}^b\mathrm{Wie} z.B. bei Exp(Num('42')).
```

1.2.5.4 Abstrakte Grammatik

Die Abstrakte Grammatik der Sprache L_{PicoC} ist in Grammatik 1.2.10 dargestellt.

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

1.2.5.5 Codebeispiel

In Code 1.5 ist der Abstrakte Syntaxbaum zu sehen, der aus dem vereinfachten Ableitungsbaum aus Code 1.4 mithilfe eines Transformers (Definition ??) generiert wurde. Das Beispiel aus Code 1.1 wird hier fortgeführt.

```
Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
 4
       StructDecl
         Name 'st',
         Γ
            Alloc
              Writeable,
 9
              PntrDecl
10
                Num '1',
11
                ArrayDecl
12
                    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
           Exp
26
27
              Alloc
28
                Writeable,
29
                ArrayDecl
30
31
                     Num '3',
32
                    Num '2'
33
                  ],
34
                  PntrDecl
35
                     Num '1',
36
                     PntrDecl
37
                       Num '1',
38
                       StructSpec
39
                         Name 'st',
40
                Name 'var
41
         ]
42
     ]
```

Code 1.5: Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum.

1.2.5.6 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 $_repr_()^{18}$ 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

¹⁸Spezielle Methode, die immer aufgerufen wird, wenn das Objekt, dass in Besitz dieser Methode ist als Zeichenkette mittels print() oder zur Repräsentation ausgegeben werden soll.

Syntax aufgebaute Textrepräsentation seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten runden öffnenden (und schließenden) Klammern, sowie Kommas ',', Semikolons ; usw. zur Darstellung der Hierarchie und zur Abtrennung zurück. Der gesamte Abstrakte Syntaxbaum wird durchlaufen und die Magischen __repr__()-Methoden der verschiedenen Knoten aufgerufen, die immer jeweils die __repr__()-Methoden ihrer Kinder aufrufen und die zurückgegebenen Textrepräsentationen passend zusammenfügen und selbst zurückgeben.

Beim PicoC-Compiler sind Abstrakte und Konkrete Syntax miteinander gemischt. Für PicoC-Knoten wird die Abstrakte Syntax verwendet, da Passes schließlich auf Abstrakten Syntaxbäumen operieren. Bei RETI-Knoten wird die Konkrete Syntax verwendet, da Maschinenbefehle in Konkreter Syntax schließlich das Endprodukt des Kompiliervorgangs sein sollen.

Da die Konkrette Syntax von RETI-Knoten sehr 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 werden die RETI-Knoten einfach immer direkt in Konkreter Syntax ausgegeben. Auf diese Weise muss nicht beim letzten Pass daran gedacht werden am Ende die Konkrete, statt der Abstrakten Syntax für die RETI-Knoten auszugeben.

Die Ausgabe des Abstrakten Syntaxbaums ist bewusst so gewählt, dass sie sich optisch von der des Ableitungsbaums unterscheidet, indem die Bezeichner der Knoten in UpperCamelCase¹⁹ geschrieben sind. Das steht im Gegensatz zum Ableitungsbaum, dessen Innere Knoten im snake_case geschrieben sind, da sie die Nicht-Terminalsymbole auf den linken Seiten des ::=-Symbols in der Konkreten Grammatik 1.2.8 darstellen, welche in snake_case geschrieben sind.

1.3 Code Generierung

Nach der Generierung eines Abstrakten Syntaxbaums als Ergebnis der Lexikalischen und Syntaktischen Analyse in Unterkapitel 1.2.5, wird in diesem Kapitel auf Basis der verschiedenen Kompositionen von 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 1.2.10 spezifiziert ist in einen semantisch gleichen Abstrakten Syntaxbaum der Sprache L_{RETI} umzuformen, der durch die Abstrakte Grammatik 1.3.6 spezifiziert ist. In T-Diagramm-Notation (siehe Unterkapitel ??) lässt sich das darstellen, wie in Abbildung 1.6.

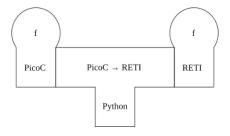


Abbildung 1.6: Kompiliervorgang Kurzform.

Das gerade angesprochene Problem lässt sich, wie in Unterkapitel ?? bereits beschrieben vereinfachen, indem man das Problem in mehrere Passes (Definition ??) aufteilt, die jeweils ein überschaubares Teilproblem lösen. Man nähert sich Schrittweise immer mehr der Syntax der Sprache L_{RETI} an.

¹⁹ Naming convention (programming).

In Abbildung 1.7 ist das T-Diagramm aus Abbildung 1.6 detailierter dargestellt. Das T-Diagramm gibt einen Überblick über alle Passes und wie diese in der Pipe-Architektur (Definition ??) 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.

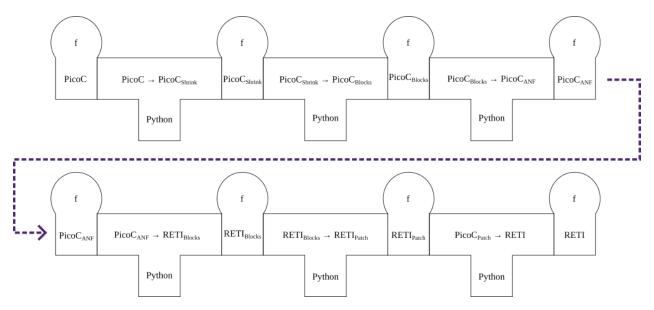


Abbildung 1.7: Architektur mit allen Passes ausgeschrieben.

Im Unterkapitel 1.3.1 werden die unterschiedlichen Passes des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln 1.3.2, 1.3.3, 1.3.4 und 1.3.6 zu Zeigern, Feldern, Verbunden und Funktionen werden einzelne Aspekte, die Thema dieser Bachelorarbeit sind genauer betrachtet und erklärt, die im Unterkapitel 1.3.1 nicht vertieft wurden. Viele der verwendenten Ansätze zur Lösung dieser Probleme basieren auf der Vorlesung 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.

1.3.1 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 Aspekte²⁰ der Bachelorarbeit wichtig.

Auf jedes Detail der einzelnen Passes wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln 1.3.2, 1.3.3, 1.3.4 und 1.3.6 zu Zeigern, Feldern, Verbunden und Funktionen im Detail erklärt sind und andererseits viele Aufgaben dieser Passes eher dem Bachelorprojekt zuzurechnen sind.

1.3.1.1 PicoC-Shrink Pass

Die Aufgabe des PicoC-Shrink Pass ist in Unterkapitel 1.3.2.2 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) in der Untermenge der Sprache L_C , welche die Sprache L_{PicoC} darstellt die gleiche Semantik hat, wie der Operator für den

²⁰In kurz: Zeiger, Felder, Verbunde und Funktionen.

Zugriff auf den Index eines Feldes ar[i]²¹.

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 den Zugriff auf einen Feldindex var[i] übernehmen.

1.3.1.1.1 Abstrakte Grammatik

Die Abstrakte Grammatik 1.3.1 der Sprache L_{PicoC_Shrink} ist fast identisch mit der Abstrakten Grammatik 1.2.10 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 (exp1, exp2) in der Abstrakten Grammatik 1.3.1 nicht mehr gibt. Das liegt daran, dass dieser Pass alle Vorkommnisse des Knoten Deref (exp1, exp2) durch den Knoten Subscr (exp1, exp2) auswechselt.

²¹Wobei *pntr und pntr[0] einander entsprechen.

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle) RETIComment()$	$L_Comment$
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
$\begin{array}{c} datatype \\ exp \end{array}$::=	$ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle) \\ Subscr(\langle exp \rangle, \langle exp \rangle) \mid Array(\langle exp \rangle+)$	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 1.3.1: Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF

Anmerkung Q

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 sind neu hinzugefügt. Das gilt genauso für alle folgenden Grammatiken.

1.3.1.1.2 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 1.6 zur Anschauung der verschiedenen Passes verwendet. Im Code 1.6 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 Scholl, "Betriebssysteme", welches diesen Algorithmus allerdings rekursiv implementiert.

Die rekursive Implementierung des Algorithmus wäre allerdings kein gutes Anschauungsbeispiel, das 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 mit der rekursiven Implementierung aus der Vorlesung nicht veranschaulicht gewesen. Daher wurde die rekursive Implementierung aus der Vorlesung zu einem iterativen Algorithmus 1.6 umgeschrieben, um unter anderem auch if- und while-Statements 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 im jeweiligen Unterkapitel beschrieben wird und werden nicht im Detail erläutert. Viele Details der Passes werden später in den Unterkapiteln 1.3.2, 1.3.3, 1.3.4 und 1.3.6 zu Zeigern, Feldern, Verbunden und Funktionen mit eigenen Codebeispielen genauer erklärt und alle sonstigen Details sind dem Bachelorprojekt zuzuordnen, dessen Aspekte in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit nicht im Detail erläutert werden.

```
based on a example program from Christoph Scholl's Operating Systems lecture
 2
   int faculty(int n){
 4
    int res = 1;
    while (1) {
       if (n == 1) {
         return res;
 8
       res = n * res:
10
         = n - 1;
11
13
14
   void main() {
    print(faculty(4));
```

Code 1.6: PicoC Code für Codebespiel.

In Code 1.7 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',
```

²²https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```
Name 'faculty',
           Alloc(Writeable(), IntType('int'), Name('n'))
         ],
10
11
           Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
12
           While
13
             Num '1',
14
15
               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
29
           Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30
31
     ]
```

Code 1.7: Abstrakter Syntaxbaum für Codebespiel.

Im PicoC-Shrink Pass ändert sich nichts im Vergleich zum Abstrakten Syntaxbaum in Code 1.7, da das Codebeispiel keine Dereferenzierung *pntr enthält. Es wurde auf ein weiteres Codebeispiel für diesen Pass verzichtet, da din diesem das gleiche zu sehen wäre, wie in Codebeispiel 1.7.

1.3.1.2 PicoC-Blocks Pass

Die Aufgabe des PicoC-Blocks Pass 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.

1.3.1.2.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 1.3.1 der Sprache L_{PicoC_Shrink} um die Knoten zu erweitern, die am Anfang dieses Unterkapitels 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(datatype, Name(str), Alloc(Writeable(), datatype, Name(str))*, (block)*) ist nun ein Container für Blöcke Block(Name(str), stmt*) und keine Anweisungen stmt mehr. Das resultiert in der Abstrakten Grammatik 1.3.2 der Sprache L_{PicoC_Blocks} .

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle) \mid RETIComment()$	$L_{-}Comment$
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	::= ::= ::=	$ \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
$\begin{array}{c} datatype \\ exp \end{array}$::= ::=	$PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)$ $Ref(\langle exp \rangle)$	L_Pntr
$\begin{array}{c} datatype \\ exp \end{array}$::=	$ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle) Subscr(\langle exp \rangle, \langle exp \rangle) \mid Array(\langle exp \rangle +)$	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	::= ::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *)$ $Return(\langle exp \rangle)$ $FunDecl(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *)$ $FunDef(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle block \rangle *)$	L_Fun
$block \\ stmt$::=	$Block(Name(\langle str \rangle), \langle stmt \rangle *) \ GoTo(Name(\langle str \rangle))$	L_Blocks
file	::=	$File(Name(\langle str \rangle), \langle decl_def \rangle *)$	L _ $File$

Grammatik 1.3.2: Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF

Anmerkung 9

Eine Abstrakte Grammatik soll im Gegensatz zu einer Konkreten Grammatik für den Programmierer, der einen darauf aufbauenden Compiler implementiert 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.

1.3.1.2.2 Codebeispiel

In Code 1.8 sieht man den aus dem Abstrakten Syntaxbaum aus Code 1.7 mithilfe des PicoC-Blocks Pass resultierenden Abstrakten Syntaxbaum. Es wird das Beispiel in Code 1.6 aus Unterkapitel 1.3.1.1 weitergeführt. Es wurden nun eigene Blöcke für die Funktion faculty und die main-Funktion erstellt, in denen die jeweils 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)- oder DoWhile(exp, stmts)-Knoten stehen. Je nachdem, ob ein If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(exp, stmts)-oder DoWhile(exp, stmts)-Knoten auftaucht, werden für die Bedingung und mögliche Branches eigene Blöcke erstellt.

```
1 File
     Name './example_faculty_it.picoc_blocks',
       FunDef
         IntType 'int',
         Name 'faculty',
           Alloc(Writeable(), IntType('int'), Name('n'))
 9
         ],
10
         [
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
24
                    GoTo(Name('while_branch.4'))
25
                 ],
26
27
                    GoTo(Name('while_after.1'))
28
                 ]
29
             ],
30
           Block
             Name 'while_branch.4',
32
33
               // If(Atom(Name('n'), Eq('=='), Num('1')), []),
34
               IfElse
35
                 Atom(Name('n'), Eq('=='), Num('1')),
36
                 [
37
                    GoTo(Name('if.3'))
38
                 ],
39
                 Ε
                    GoTo(Name('if_else_after.2'))
```

```
]
42
             ],
43
           Block
              Name 'if.3',
45
46
                Return(Name('res'))
47
             ],
48
           Block
              Name 'if_else_after.2',
49
50
51
                Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52
                Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53
                GoTo(Name('condition_check.5'))
54
             ],
55
           Block
56
              Name 'while_after.1',
57
58
         ],
59
       FunDef
60
         VoidType 'void',
61
         Name 'main',
62
         [],
63
64
           Block
65
              Name 'main.0',
66
67
                Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])))
68
              ]
69
         ]
70
     ]
```

Code 1.8: PicoC-Blocks Pass für Codebespiel.

1.3.1.3 PicoC-ANF Pass

Die Aufgabe des PicoC-ANF Pass 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 ??) und damit auch in Monadischer Normalform (Definition ??) ist. Um Redundanz zu vermeiden wird zur Erklärung der A-Normalform auf Unterkapitel ?? verwiesen und zur Erklärung der Monadischen Normalform auf Unterkapitel ?? verwiesen.

Zudem wird eine Symboltabelle (Definition 1.8) verwendet. In der Symboltabelle wird beim Anlegen eines neuen Eintrags für eine Variable zunächst eine Adresse an das value or address-Attribut dieses Eintrags 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 der momentan aktiven Funktion. Einer der beiden Counter wird nach Anlegen eines Eintrags entsprechend der Größe der angelegten Variable hochgezählt.

Kommt im Programmcode an einer späteren Stelle eine definierte Variable Name('var') vor, so wird mit der Konkatenation des Bezeichners der Variable und des Bezeichners des momentanen Sichtbarkeitsbereichts var@scope²³ als Schlüssel in der Symboltabelle der entsprechende Symboltabelleneintrag der Variable var nachgeschlagen. Anstelle des Name(str)-Knotens der Variable wird ein Global(num) bzw. Stackframe(num)-Knoten eingefügt, dessen num-Attribut die Adresse im value or address-Attribut des Symboltabelleneintrags zugewiesen bekommt.

²³Die Umsetzung von Sichtbarkeitsbereichen wird in Unterkapitel 1.3.6.2 genauer beschrieben.

Ob der Global(num)- oder der Stackframe(num)-Knoten für die Ersetzung verwendet wird, entscheidet sich anhand des Sichtbarkeitsbereichs scope. Für den Sichtbarkeitsbereich der main-Funktion wird der Global(num)-Knoten verwendet. Für den Sichtbarkeitsbereich jeder anderen Funktion wird der Stackframe(num)-Knoten verwendet.

Darüberhinaus gibt es den Sichtbarkeitsbereich global!, dessen Variablen und Konstanten überall sichtbar sind und der für globale Variablen verwendet wird. Der Sichtbarkeitsbereich global! hat die geringste Priorität aller Sichtbarkeitsbereiche. Das bedeutet, dass, wenn im Sichtbarkeitsbereich einer Funktion und im Sichtarkeitsbereich global! zweimal der gleiche Bezeichner vorkommt, dem Sichtbarkeitsbereich der Funktion Vorrang gelassen wird. Für den Sichtbarkeitsbereich global! wird der Global(num)-Knoten verwendet.

Das Symbol '!' im Suffix global! und das Symbol '@' als Trennzeichen in var@scope wurden aus einem bestimmten Grund verwendet, nämlich, weil kein Bezeichner die Symbole '@' und ! jemals selbst enthalten kann. Die Produktionen für einen Bezeichner in der Konkretten Grammatik $G_{Lex} \uplus G_{Parse}$ (siehe 1.1.1 und 1.2.10) lassen beide Symbole @ und ! nicht zu. Damit ist es ausgeschlossen, dass es zu Problemen kommt, falls ein Benutzer des PicoC-Compilers zufällig auf die Idee kommt eine Funktion auf eine unpassende Weise zu benennen²⁴.

Definition 1.8: Symboltabelle

1

Eine meist über ein Assoziatives Feld umgesetzte Datenstruktur, die notwendig ist, um das Konzept von Variablen und Konstanten 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, welche später nicht mehr so einfach zugänglich sind, wie zu dem Zeitpunkt, zu dem sie in einen Symboltabelleneintrag gespeichert werden.

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.

 $^a\mathrm{In}$ der Code Generierung werden Bezeichner als Symbole bezeichnet.

1.3.1.3.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 1.3.2 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 enthalten können. Wie es bereits im Unterkapitel ?? erklärt wurde, ist beim PicoC-Compiler der Knoten Stack(Num(str)) der einzige Atomare Knoten.

Des Weiteren werden auch Funktionen und Funktionsaufrufe aufgelöst, sodass die Blöcke Block(Name(str), stmt*) nun direkt im decls_defs_blocks-Attribut des File(Name(str), decls_defs_blocks*)-Knoten liegen usw. 25 . Die Symboltabelle ist ebenfalls als Abstrakter Syntaxbaum umgesetzt, wofür in der Abstrakten Grammatik 1.3.3 der Sprache L_{PicoC_ANF} neue Knoten eingeführt wurden. Das ganze resultiert in der Abstrakten Grammatik 1.3.3 der Sprache L_{PicoC_ANF} .

²⁴Z.B. var@fun2 oder global als Funktionsname.

 $^{^{25}\}mathrm{Im}$ Unterkapitel 1.3.6 wird das Auflösen von Funktionen genauer erklärt.

```
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)))
type\_qual
                              Const()
                                                 Writeable()
                                                                                                                                                  L\_Assign\_Alloc
                      ::=
                              IntType() \mid CharType() \mid VoidType()
datatype
                      ::=
exp
                              Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle))
                      ::=
                              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
                      ::=
                               StructSpec(Name(\langle str \rangle))
                                                                                                                                                  L\_Struct
datatype
                      ::=
                               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
                                                                                                                                                                42
```

1.3.1.3.2 Codebeispiel

In Code 1.9 sieht man die als Abstrakter Syntaxbaum umgesetzte Symboltabelle, in der alle Variablen und Funktionen aus dem weitergeführten Beispiel in Code 1.6 aus Unterkapitel 1.3.1.1 einen Eintrag haben.

```
SymbolTable
     Ε
       Symbol
 4
         {
           type qualifier:
                                     Empty()
 6
           datatype:
                                     FunDecl(IntType('int'), Name('faculty'), [Alloc(Writeable(),

    IntType('int'), Name('n'))])

                                     Name('faculty')
           name:
 8
           value or address:
                                     Empty()
 9
           position:
                                     Pos(Num('3'), Num('4'))
10
           size:
                                     Empty()
11
         },
12
       Symbol
13
         {
14
                                     Writeable()
           type qualifier:
15
                                     IntType('int')
           datatype:
16
                                     Name('n@faculty')
17
                                     Num('0')
           value or address:
18
           position:
                                     Pos(Num('3'), Num('16'))
19
           size:
                                     Num('1')
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                     Writeable()
24
                                     IntType('int')
           datatype:
25
                                     Name('res@faculty')
           name:
26
                                     Num('1')
           value or address:
27
                                     Pos(Num('4'), Num('6'))
           position:
28
                                     Num('1')
           size:
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('14'), Num('5'))
           position:
37
                                     Empty()
           size:
38
39
    ]
```

Code 1.9: Symboltabelle für Codebespiel.

In Code 1.10 sieht man den aus dem Abstrakten Syntaxbaum aus Code 1.8 mithilfe des PicoC-ANF Pass resultierenden Abstrakten Syntaxbaum. Alle Anweisungen und Ausdrücke sind in A-Normalform, z.B. sind die IfElse(exp, stmts1, stmts2)-Knoten dadurch in A-Normalform, dass ihre Komplexen Ausdrücke im exp-Attribut in ein Exp(exp)-Knoten eingesetzt und vorgezogen werden (z.B. Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))). Die Exp(exp)-Knoten weisen die Ergebnisse der Komplexen Ausdrücke Locations zu, welche sich beim PicoC-Compiler auf dem Stack befinden. Die Ergeb-

nisse werden dann über Atomare Ausdrücke Stack(Num(str)) vom Stack gelesen: IfElse(Stack(Num(str)), stmts1, stmts2).

Funktionen sind nur noch über die Name(str)-Knoten von Blöcken zu erkennen, die den gleichen Bezeichner haben, wie die ursprüngliche Funktion und es lässt sich nur durch das Nachverfolgen bestimmter GoTo(Name(str))-Knoten nachvollziehen, was ursprünglich zur Funktion gehörte.

```
File
     Name './example_faculty_it.picoc_mon',
 4
5
       Block
         Name 'faculty.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',
16
           // IfElse(Num('1'), [], [])
17
           Exp(Num('1')),
18
           IfElse
19
             Stack
20
               Num '1',
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'))
           Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
35
36
           IfElse
37
             Stack
38
               Num '1',
39
             Γ
40
               GoTo(Name('if.3'))
41
             ],
42
43
               GoTo(Name('if_else_after.2'))
44
45
         ],
46
       Block
47
         Name 'if.3',
48
           // Return(Name('res'))
```

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

Code 1.10: Pico C-ANF Pass für Codebespiel.

1.3.1.4 RETI-Blocks Pass

Die Aufgabe des RETI-Blocks Pass ist es die PicoC-Knoten der Anweisungen in den Blöcken des Abstrakten Syntaxbaums der Sprache L_{PicoC_ANF} durch semantisch entsprechende RETI-Knoten zu ersetzen.

1.3.1.4.1 Abstrakte Grammatik

Die Abstrakte Grammatik 1.3.4 der Sprache L_{RETI_Blocks} ist verglichen mit der Abstrakten Grammatik 1.3.3 der Sprache L_{PicoC_ANF} stark verändert, denn der Großteil der PicoC-Knoten wird in diesem Pass durch semantisch 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 1.3.1.6 aufgelöst wird.

```
ACC() \mid IN1()
                                                IN2()
                                                                PC()
                                                                              SP()
                                                                                           BAF()
                                                                                                                                           L\_RETI
reg
                  CS() \mid DS()
                  Reg(\langle reg \rangle) \mid Num(\langle str \rangle)
arg
          ::=
rel
                  Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()
                  Always() \mid NOp()
                                                Sub()
                                                             Subi() \mid Mult() \mid Multi()
                  Add()
                                Addi()
op
                  Div() \quad | \quad Divi() \quad | \quad Mod() \quad | \quad Modi() \quad | \quad Oplus() \quad | \quad Oplusi()
                  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)))
instr
          ::=
                  Exp(GoTo(\langle str \rangle))
                                                                                                                                           L_{-}PicoC
block
                  Block(Name(\langle str \rangle), \langle instr \rangle *)
          ::=
                  File(Name(\langle str \rangle), \langle block \rangle *)
file
```

Grammatik 1.3.4: Abstrakte Grammatik der Sprache L_{RETI-Blocks} in ASF

1.3.1.4.2 Codebeispiel

In Code 1.11 sieht man den aus dem Abstrakten Syntaxbaum aus Code 1.10 mithilfe des RETI-Blocks Pass resultierenden Abstrakten Syntaxbaum. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. Die Anweisungen, die durch entsprechende PicoC-Knoten im Abstrakten Syntaxbaum der Sprache $L_{PicoC-ANF}$ repräsentiert waren, werden nun durch semantisch entsprechende RETI-Knoten ersetzt.

```
1 File
     Name './example_faculty_it.reti_blocks',
     Γ
       Block
         Name 'faculty.6',
           # // Assign(Name('res'), Num('1'))
 8
           # Exp(Num('1'))
 9
           SUBI SP 1;
10
           LOADI ACC 1;
11
           STOREIN SP ACC 1;
12
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
           LOADIN SP ACC 1;
14
           STOREIN BAF ACC -3;
           ADDI SP 1;
16
           # // While(Num('1'), [])
17
           # Exp(GoTo(Name('condition_check.5')))
18
           Exp(GoTo(Name('condition_check.5')))
19
         ],
20
       Block
21
         Name 'condition_check.5',
22
         Ε
23
           # // IfElse(Num('1'), [], [])
24
           # Exp(Num('1'))
25
           SUBI SP 1;
26
           LOADI ACC 1;
           STOREIN SP ACC 1;
```

```
# IfElse(Stack(Num('1')), [], [])
29
           LOADIN SP ACC 1;
30
           ADDI SP 1;
31
           JUMP== GoTo(Name('while_after.1'));
32
           Exp(GoTo(Name('while_branch.4')))
33
         ],
34
       Block
35
         Name 'while_branch.4',
36
37
           # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
38
           # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39
           # Exp(Stackframe(Num('0')))
           SUBI SP 1;
40
41
           LOADIN BAF ACC -2;
42
           STOREIN SP ACC 1;
43
           # Exp(Num('1'))
44
           SUBI SP 1;
45
           LOADI ACC 1;
46
           STOREIN SP ACC 1;
47
           LOADIN SP ACC 2;
48
           LOADIN SP IN2 1;
49
           SUB ACC IN2;
50
           JUMP== 3;
51
           LOADI ACC 0;
52
           JUMP 2;
53
           LOADI ACC 1;
54
           STOREIN SP ACC 2;
           ADDI SP 1;
56
           # IfElse(Stack(Num('1')), [], [])
57
           LOADIN SP ACC 1;
58
           ADDI SP 1;
59
           JUMP== GoTo(Name('if_else_after.2'));
60
           Exp(GoTo(Name('if.3')))
61
         ],
62
       Block
63
         Name 'if.3',
64
65
           # // Return(Name('res'))
66
           # Exp(Stackframe(Num('1')))
67
           SUBI SP 1;
68
           LOADIN BAF ACC -3;
69
           STOREIN SP ACC 1;
70
           # Return(Stack(Num('1')))
71
           LOADIN SP ACC 1;
72
           ADDI SP 1;
73
           LOADIN BAF PC -1;
74
        ],
75
       Block
76
         Name 'if_else_after.2',
77
78
           # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
           # Exp(Stackframe(Num('0')))
           SUBI SP 1;
80
81
           LOADIN BAF ACC -2;
           STOREIN SP ACC 1;
82
83
           # Exp(Stackframe(Num('1')))
           SUBI SP 1;
```

```
LOADIN BAF ACC -3;
           STOREIN SP ACC 1;
86
87
           # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88
           LOADIN SP ACC 2;
           LOADIN SP IN2 1;
90
           MULT ACC IN2:
91
           STOREIN SP ACC 2;
           ADDI SP 1;
92
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')))
           # Exp(Stackframe(Num('0')))
98
99
           SUBI SP 1;
100
           LOADIN BAF ACC -2;
01
           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;
114
           STOREIN BAF ACC -2;
115
           ADDI SP 1;
116
           # Exp(GoTo(Name('condition_check.5')))
           Exp(GoTo(Name('condition_check.5')))
117
118
         ],
L19
       Block
120
         Name 'while_after.1',
L21
122
           # Return(Empty())
123
           LOADIN BAF PC -1;
124
         ],
L25
       Block
126
         Name 'main.0',
L27
         Γ
L28
           # StackMalloc(Num('2'))
129
           SUBI SP 2;
130
           # Exp(Num('4'))
131
           SUBI SP 1;
132
           LOADI ACC 4;
133
           STOREIN SP ACC 1;
L34
           # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
L35
           MOVE BAF ACC;
136
           ADDI SP 3;
           MOVE SP BAF;
137
138
           SUBI SP 4;
L39
           STOREIN BAF ACC 0;
           LOADI ACC GoTo(Name('addr@next_instr'));
           ADD ACC CS;
```

```
STOREIN BAF ACC -1;
           # Exp(GoTo(Name('faculty.6')))
           Exp(GoTo(Name('faculty.6')))
            # RemoveStackframe()
           MOVE BAF IN1;
           LOADIN IN1 BAF O:
48
           MOVE IN1 SP;
149
           # Exp(ACC)
150
           SUBI SP 1;
L51
           STOREIN SP ACC 1;
152
           LOADIN SP ACC 1;
153
            ADDI SP 1;
154
           CALL PRINT ACC;
155
           # Return(Empty())
156
           LOADIN BAF PC -1;
157
         ]
158
     ]
```

Code 1.11: 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 1.2.5.6 vermitteltem Grund in Konkreter Syntax ausgegeben werden.

1.3.1.5 RETI-Patch Pass

Die Aufgabe des RETI-Patch Pass ist das Ausbessern (engl. to patch) des Abstrakten Syntaxbaumes der Sprache L_{RETI_Blocks} durch:

- das Einfügen eines start. <number>-Blocks, 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 vor jede Division Instr(Div(), args), die pr
 üfen, ob durch
 0 geteilt wird. Ist es der Fall, dass durch 0 geteilt wird, dann wird in das ACC-Register der Fehlercode
 1 geschrieben, der f
 ür die Fehlerart DivisionByZero steht und die Ausf
 ührung des Programmes wird
 beendet. Andernfalls l
 äuft das Programm weiter.
- das Überprüfen darauf, ob bestimmte Immediates Im(str) in Befehlen, wie z.B. Jump(rel, Im(str)), Instr(Loadin(), [reg1, reg2, Im(str)]), Instr(Loadi(), [reg, Im(str)]) usw. kleiner als -2²¹ oder größer als 2²¹ 1 sind. Im diesem Fall muss der gewünschte Zahlenwert durch Bitshiften und Anwenden von Bitweise ODER aus Zahlenwerten berechnet werden, die sich im Zahlenbereich zwischen -2²¹ und 2²¹ 1 befinden. Im Fall dessen, dass der Immediate allerdings kleiner als -(2³¹) oder größer als 2³¹ 1 ist, wird eine Fehlermeldung TooLargeLiteral ausgegeben.

1.3.1.5.1 Abstrakte Grammatik

 $^{^{26}}$ Einiges in diesem Pass fällt unter die Themenbereiche des Bachelorprojekts und wird daher nicht genauer erläutert.

Die Abstrakte Grammatik 1.3.5 der Sprache L_{RETI_Patch} ist im Vergleich zur Abstrakten Grammatik 1.3.4 der Sprache L_{RETI_Blocks} unverändert. Es sind keine neuen Knoten hinzugekommen und keine Knoten wurden abgeändert oder völlig entfernt.

```
|SP()
                 ACC() \mid IN1()
                                                          PC()
                                                                                                                                        L\_RETI
                                               IN2()
req
          ::=
                 CS() \mid DS()
                 Reg(\langle reg \rangle) \mid Num(\langle str \rangle)
arq
                 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 +) \quad | \quad Jump(\langle rel \rangle, Num(\langle str \rangle)) \quad | \quad 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))
                                                                                                                                        L_{-}PicoC
instr
          ::=
                 Block(Name(\langle str \rangle), \langle instr \rangle *)
block
                 File(Name(\langle str \rangle), \langle block \rangle *)
file
          ::=
```

Grammatik 1.3.5: Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF

1.3.1.5.2 Codebeispiel

In Code 1.12 sieht man den aus dem Abstrakten Syntaxbaum aus Code 1.11 mithilfe des RETI-Patch Pass resultierenden Abstrakten Syntaxbaum. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. 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))-Knoten entfernt, die nur einem Sprung um eine Adresse weiter entsprochen hätten²⁸.

```
2
    Name './example_faculty_it.reti_patch',
 3
     Γ
       Block
         Name 'start.7',
6
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;
17
           STOREIN SP ACC 1;
18
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
           LOADIN SP ACC 1;
19
```

 $^{^{27}\}mathrm{Dieser}$ start.

nummer>-Block wurde im Code 1.12 markiert.

²⁸Diese entfernten GoTo(Name(str))-Knoten wurden in Code 1.12 markiert.

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

```
LOADIN SP ACC 1;
78
           ADDI SP 1;
79
           LOADIN BAF PC -1;
80
         ],
81
       Block
         Name 'if_else_after.2',
82
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')))
           SUBI SP 1;
90
91
           LOADIN BAF ACC -3;
92
           STOREIN SP ACC 1;
93
           # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94
           LOADIN SP ACC 2;
95
           LOADIN SP IN2 1;
96
           MULT ACC IN2;
97
           STOREIN SP ACC 2;
98
           ADDI SP 1;
99
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
100
           LOADIN SP ACC 1;
101
           STOREIN BAF ACC -3;
102
           ADDI SP 1:
103
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104
           # Exp(Stackframe(Num('0')))
105
           SUBI SP 1;
106
           LOADIN BAF ACC -2;
107
           STOREIN SP ACC 1;
108
           # Exp(Num('1'))
109
           SUBI SP 1;
110
           LOADI ACC 1;
111
           STOREIN SP ACC 1;
112
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113
           LOADIN SP ACC 2;
114
           LOADIN SP IN2 1;
115
           SUB ACC IN2;
116
           STOREIN SP ACC 2;
117
           ADDI SP 1;
118
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
L19
           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
126
         Name 'while_after.1',
L27
128
           # Return(Empty())
129
           LOADIN BAF PC -1;
130
         ],
l31
       Block
132
         Name 'main.0',
133
```

```
# StackMalloc(Num('2'))
            SUBI SP 2;
            # Exp(Num('4'))
.36
            SUBI SP 1;
            LOADI ACC 4;
139
            STOREIN SP ACC 1:
40
            # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
L41
           MOVE BAF ACC;
142
            ADDI SP 3;
143
           MOVE SP BAF;
144
            SUBI SP 4;
45
            STOREIN BAF ACC 0;
46
            LOADI ACC GoTo(Name('addr@next_instr'));
L47
            ADD ACC CS;
.48
            STOREIN BAF ACC -1;
L49
            # Exp(GoTo(Name('faculty.6')))
L50
            Exp(GoTo(Name('faculty.6')))
151
            # RemoveStackframe()
152
            MOVE BAF IN1:
153
           LOADIN IN1 BAF 0;
L54
           MOVE IN1 SP;
155
            # Exp(ACC)
156
            SUBI SP 1;
157
            STOREIN SP ACC 1;
158
            LOADIN SP ACC 1;
159
            ADDI SP 1;
160
            CALL PRINT ACC;
l61
            # Return(Empty())
162
            LOADIN BAF PC -1;
163
         ]
164
     ]
```

Code 1.12: RETI-Patch Pass für Codebespiel.

1.3.1.6 RETI Pass

Die Aufgabe des RETI-Patch Pass ist es die letzten verbliebenen PicoC-Knoten im Abstrakten Syntaxbaum zu entfernen bzw. zu ersetzen. Dementsprechend werden die Blöcke Block(Name(str), instr*) entfernt und die Knoten in diesen Blöcken werden genauso zusammengefügt, wie die Blöcke angeordnet waren.

Des Weiteren werden 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 einen Immediate Im(str(distance_or_address)) mit passender Adresse oder Distanz oder einen Sprungbefehl mit passender Distanz Jump(Always(), Im(str(distance))) ersetzt. Die Distanz- und Adressberechnung wird in Unterkapitel 1.3.6.3 genauer erklärt.

1.3.1.6.1 Konkrete und Abstrakte Grammatik

Die Abstrakte Grammatik 1.3.6 der Sprache L_{RETI} hat im Vergleich zur Abstrakten Grammatik 1.3.5 der Sprache L_{RETI_Patch} nur noch ausschließlich RETI-Knoten, dementsprechend gibt es die Knoten Instr(Loadi(), [Reg(Acc()), GoTo(Name(str))]), Jump(Eq(), GoTo(Name(str))) nicht mehr. Des Weiteren gibt es keine Blöcke Block(Name(str), instr*) mehr, alle RETI-Knoten stehen nun in einem Program(Name(str), instr*)-Knoten.

Ausgegeben wird der finale Abstrakte Syntaxbaum in Konkreter Syntax, die durch die Konkreten Grammatiken 1.3.7 und 1.3.8 für jeweils die Lexikalische und Syntaktische Analyse $G_{RETI_Lex} \uplus G_{RETI_Parse}$ beschrieben wird.

```
SP()
                                                                                             BAF()
                                                                                                                        L\_RETI
                      ACC() \mid IN1()
                                                    IN2()
                                                                  PC()
reg
               ::=
                      CS()
                                   DS()
                                          Num(\langle str \rangle)
arg
               ::=
                      Reg(\langle reg \rangle)
                      Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()
rel
               ::=
                      Always() \mid NOp()
                                   Addi() \mid Sub() \mid Subi() \mid Mult() \mid Multi()
                      Add()
op
               ::=
                      Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus()
                                                                                             Oplusi()
                      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))])
                      Jump(Eq(), GoTo(Name(\langle str \rangle)))
                      Program(Name(\langle str \rangle), \langle instr \rangle *)
program
              ::=
                      Exp(GoTo(\langle str \rangle)) \mid Exit(Num(\langle str \rangle))
                                                                                                                        L_{-}PicoC
instr
               ::=
                      Block(Name(\langle str \rangle), \langle instr \rangle *)
block
               ::=
                      File(Name(\langle str \rangle), \langle block \rangle *)
file
               ::=
```

Grammatik 1.3.6: Abstrakte Grammatik der Sprache L_{RETI} in ASF

```
"1"
                            "2"
                                     "3"
                                             "4"
                                                               "6"
dig\_no\_0
                                                      "5"
                                                                            L_Program
             ::=
                   "7"
                                     "9"
                            "8"
                   "0"
dig\_with\_0
                            dig\_no\_0
             ::=
                   "0"
                            dig\_no\_0 dig\_with\_0*
                                                    "-"dig_no_0*
num
             ::=
                   "a"..."Z"
letter
             ::=
name
             ::=
                   letter(letter \mid dig\_with\_0 \mid \_)*
                                "IN1"
                                            "IN2"
                                                        "PC"
reg
             ::=
                   "ACC"
                   "BAF"
                                "CS"
                                           "DS"
arg
             ::=
                   reg
                              "! = "
rel
             ::=
                              "\_NOP"
```

Grammatik 1.3.7: Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF

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

Grammatik 1.3.8: Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF

1.3.1.6.2 Codebeispiel

In Code 1.13 sieht man den aus dem Abstrakten Syntaxbaum aus Code 1.12 mithilfe des PicoC-Blocks Pass resultierenden Abstrakten Syntaxbaum. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. Es gibt keine Blöcke mehr und die Knoten in diesen Blöcken wurden zusammengesetzt, wie sie in den Blöcken angeordnet waren. Das Programm ist komplett in RETI-Knoten übersetzt, die allerdings in ihrer Konkreten Syntax ausgegeben werden.

Die letzten Nicht-RETI-Befehle oder RETI-Befehle, die nicht auschließlich aus RETI-Knoten bestanden LOADI ACC GoTo(Name('addr@next_instr')), Exp(GoTo(Name('main.0'))) und JUMP== GoTo(Name('if_else_after. 2')), wurden durch RETI-Befehle ersetzt, welche in Code 1.13 markiert wurden.

Der Program(Name(str), instr)-Knoten, der alle RETI-Knoten beinhaltet, gibt alleinig die RETI-Knoten, die er beinhaltet aus und fügt ansonsten nichts zur Ausgabe hinzu, wie z.B. den Bezeichner des Programms oder Einrückung. Hierdurch erzeugt der Abstrakte Syntaxbaum, wenn er in Konkreter Syntax in eine Datei ausgegeben wird direkt RETI-Code in menschenlesbarer Repräsentation.

```
# // 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'), [])
# Exp(GoTo(Name('condition_check.5')))
14 # // not included Exp(GoTo(Name('condition_check.5')))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
```

```
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4')))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0')))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;
43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3:
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;
73 # Assign(Stackframe(Num('1')), Stack(Num('1')))
74 LOADIN SP ACC 1;
75 STOREIN BAF ACC -3;
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
```

```
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1')))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5')))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;
00 # StackMalloc(Num('2'))
01 SUBI SP 2;
02 # Exp(Num('4'))
03 SUBI SP 1;
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;
10 SUBI SP 4;
11 STOREIN BAF ACC 0;
12 LOADI ACC 80;
13 ADD ACC CS;
14 STOREIN BAF ACC -1;
# Exp(GoTo(Name('faculty.6')))
16 JUMP -78;
17 # RemoveStackframe()
18 MOVE BAF IN1;
19 LOADIN IN1 BAF 0;
20 MOVE IN1 SP;
21 # Exp(ACC)
22 SUBI SP 1;
23 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
25 ADDI SP 1;
26 CALL PRINT ACC;
27 # Return(Empty())
28 LOADIN BAF PC -1;
```

Code 1.13: RETI Pass für Codebespiel.

1.3.2 Umsetzung von Zeigern

Die Umsetzung von Zeigern ist in diesem Unterkapitel schnell erklärt, auch Dank eines kleinen Taschenspielertricks²⁹. Hierbei sind nur die Operationen für Referenzierung und Dereferenzierung in den Unterkapiteln 1.3.2.1 und 1.3.2.2 zu erläutern. Referenzierung kann dazu genutzt werden einen Zeiger zu initialisieren und Dereferenzierung kann dazu genutzt werden, um auf diesen später zuzugreifen.

1.3.2.1 Referenzierung

Die Referenzierung (z.B. &var) ist eine Operation bei der ein Zeiger auf eine Location (Definition ??) in Form der Anfangsadresse dieser Location als Ergebnis zurückgegeben wird. Die Umsetzung der Referenzierung wird im Folgenden anhand des Beispiels in Code 1.14 erklärt.

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

Code 1.14: PicoC-Code für Zeigerreferenzierung.

Der Knoten Ref(Name('var'))) repräsentiert im Abstrakten Syntaxbaum in Code 1.15 eine Referenzierung &var und der Knoten PntrDecl(Num('1'), IntType('int')) repräsentiert einen Zeiger *pntr.

```
1
  File
    Name './example_pntr_ref.ast',
2
     Γ
       {\tt FunDef}
         VoidType 'void',
6
7
8
         Name 'main',
         [],
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
         ]
    ]
```

Code 1.15: 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 1.16.

```
Die Anzahl Speicherzellen<sup>a</sup>, die ein Zeiger<sup>b</sup> datatype *pntr belegt ist dabei immer<sup>c</sup>: size(type(pntr)) = 1 Speicherzelle. def

aDie im size-Attribut der Symboltabelle eingetragen ist.
bZ.B. ein Zeiger auf ein Feld von Integern: int (*pntr) [3].
cHier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die manchmal etwas
```

²⁹Später mehr dazu.

unpraktische Designentscheidung, die eckigen Klammern [] bei der Definition eines Feldes hinter die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche Datentyp komplett vor der Variable stehen: datatype var.

- d Eine Speicherzelle ist in der RETI-Architektur, wie in Unterkapitel $\ref{lem:speicherzelle}$ erklärt 4 Byte breit.
- ^eDie Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.
- ^fDie Funktion type ordnet einer Variable ihren Datentyp zu. Das ist notwendig, weil die Funktion size als Definitionsmenge Datentypen hat.

```
SymbolTable
     [
       Symbol
         {
           type qualifier:
                                     Empty()
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
 7
8
           name:
                                     Name('main')
           value or address:
                                     Empty()
 9
                                     Pos(Num('1'), Num('5'))
           position:
10
                                     Empty()
           size:
11
         },
12
       Symbol
13
         {
14
           type qualifier:
                                     Writeable()
15
           datatype:
                                     IntType('int')
                                     Name('var@main')
           name:
17
           value or address:
                                     Num('0')
18
           position:
                                     Pos(Num('2'), Num('6'))
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
           size:
                                     Num('1')
29
30
    ]
```

Code 1.16: Symboltabelle für Zeigerreferenzierung.

Im PicoC-ANF Pass in Code 1.17 wird der Knoten Ref(Name('var'))) durch die Knoten Ref(GlobalRead (Num('0'))) und Assign(GlobalWrite(Num('1')), Tmp(Num('1'))) ersetzt. Im Fall, dass in Ref(exp)) das exp vielleicht nicht direkt ein Name('var') enthält und exp z.B. ein Subscr(Attr(Name('var'), Name('attr')), Num('1')) ist, sind noch weitere Anweisungen zwischen den Zeilen 11 und 12 nötig. Diese weiteren Anweisungen würden sich bei z.B. Subscr(Attr(Name('var'), Name('attr')), Num('1')) um das Übersetzen von Subscr(exp) und Attr(exp,name) nach dem Schema in Unterkapitel 1.3.5.2 kümmern.³⁰

```
1 File
2 Name './example_pntr_ref.picoc_mon',
```

³⁰Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
Block
        Name 'main.0',
7
8
           // Assign(Name('var'), Num('42'))
           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 1.17: PicoC-ANF Pass für Zeigerreferenzierung.

Im RETI-Blocks Pass in Code 1.18 werden die PicoC-Knoten Ref(Global(Num('0'))) und Assign(Global (Num('1')), Stack(Num('1'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
File
     Name './example_pntr_ref.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('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
         ]
    ]
```

Code 1.18: RETI-Blocks Pass für Zeigerreferenzierung.

1.3.2.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

Die Dereferenzierung (z.B. *var) ist eine Operation bei der einem Zeiger zur Location (Definition ??) hin gefolgt wird, auf welche dieser zeigt und das Ergebnis z.B. der Inhalt der ersten Speicherzelle der referenzierten Location ist. Die Umsetzung von Dereferenzierung wird im Folgenden anhand des Beispiels in Code 1.19 erklärt.

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

Code 1.19: PicoC-Code für Zeigerdereferenzierung.

Der Knoten Deref(Name('var'), Num('0'))) repräsentiert im Abstrakten Syntaxbaum in Code 1.20 eine Dereferenzierung *var. Es gibt hierbei 3 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'))) und bei z.B. *(var - 2 - 1) zu Deref(Name('var'), UnOp(Minus(), BinOp(Num('2'), Sub(), Num('1')))). Bei einer normalen Dereferenzierung, wie z.B. *var, übersetzt sich diese zu Deref(Name('var'), Num('0'))³¹.

```
1
  File
2
    Name './example_pntr_deref.ast',
4
      FunDef
5
         VoidType 'void',
6
7
8
        Name 'main',
         [],
         Γ
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 1.20: Abstrakter Syntaxbaum für Zeigerdereferenzierung.

Im PicoC-Shrink Pass in Code 1.21 wird ein Trick angewandet, bei dem jeder Knoten Deref(exp1, exp2) einfach durch den Knoten Subscr(exp1, exp2) ersetzt wird. 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 ?? erläutert wurde. Damit spart man sich viele vermeidbare Fallunterscheidungen und doppelten Code und kann die Übersetzung der Derefenzierung (z.B. *(var + 1)) einfach von den Routinen für einen Zugriff auf einen Feldindex (z.B. var[1]) übernehmen lassen. Das Vorgehen bei der Umsetzung eines Zugriffs auf einen Feldindex (z.B. *(var + 1)) wird in Unterkapitel 1.3.3.2 erläutert.³²

³¹Das Num('0') steht dafür, dass dem Zeiger gefolgt wird, aber danach nicht noch mit einem Versatz von der Größe des Unterdatentyps (Definition 1.9) auf eine nebenliegende Location zugegriffen wird.

³²Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
Name './example_pntr_deref.picoc_shrink',
4
      FunDef
        VoidType 'void',
        Name 'main',
7
8
        [],
        Γ
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(Subscr(Name('pntr'), Num('0')))
12
13
    1
```

Code 1.21: PicoC-Shrink Pass für Zeigerdereferenzierung.

1.3.3 Umsetzung von Feldern

Bei Feldern ist in diesem Unterkapitel die Umsetzung der Innitialisierung eines Feldes 1.3.3.1, des Zugriffs auf einen Feldindex 1.3.3.2 und der Zuweisung an einen Feldindex 1.3.3.3 zu klären.

1.3.3.1 Initialisierung eines Feldes

Die Umsetzung der Initialisierung eines Feldes (z.B. int ar[2][1] = {{3+1}, {5}}) wird im Folgenden anhand des Beispiels in Code 1.22 erklärt.

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

void main() {
  int ar[2][1] = {{3+1}, {5}};
}
```

Code 1.22: PicoC-Code für die Initialisierung eines Feldes.

Die Initialisierung eines Feldes intar[2][1]={{3+1},{5}} wird im Abstrakten Syntaxbaum in Code 1.23 mithilfe der Knoten Assign(Alloc(Writeable(),ArrayDecl([Num('2'),Num('1')],IntType('int')),Name('ar')),Array([Array([BinOp(Num('3'),Add('+'),Num('1'))]),Array([Num('5')])))) dargestellt.

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

```
Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
              Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
10
         ],
11
      FunDef
12
         VoidType 'void',
13
         Name 'main',
14
         [],
15
         Γ
           Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
16
               Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
               Array([Num('5')])))
17
18
    ]
```

Code 1.23: 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 kann³³. Das Definieren der Variable ar erfolgt mittels der Symboltabelle, die in Code 1.24 dargestellt ist.

Auf dem Stackframe wird ein Feld verglichen zur Wachstumrichtung des Stacks rückwärts in den Stackframe geschrieben und die relative Adresse des ersten Elements als Adresse des Feldes in der Symboltabelle in Code 1.24 genommen. Dies ist in Tabelle 1.9 für ein Datensegment der Größe 8 und das Beispiel aus Code 1.22 dargstellt. Es wird hier so getann als würde die Funktion fun ebenfalls aufgerufen werden. Der Stack wächst zwar verglichen zu den Globalen Statischen Daten in die entgegengesetzte Richtung, aber Felder in den Globalen Statischen Daten und in einem Stackframe haben die gleiche Ausrichtung. Das macht den Zugriff auf einen Feldindex in Unterkapitel 1.3.3.2 deutlich unkomplizierter. Auf diese Weise muss beim Zugriff auf einen Feldindex nicht zwischen Stackframe und Globalen Statischen Daten unterschieden werden.

Relativ- adresse	Wert	$\operatorname{Register}$
0	4	$^{\mathrm{CS}}$
1	5	
3	3	
2	4	
1	5	
0	6	
		BAF

Tabelle 1.9: Datensegment nach der Initialisierung beider Felder.

Anmerkung 9

Die Anzahl Speicherzellen, die ein Feld^a datatype $ar[dim_1] \dots [dim_n]$ belegt^b, berechnet sich aus der Mächtigkeit der einzelnen Dimensionen des Feldes und der Anzahl Speicherzellen, die der

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

```
Datentyp, den alle Feldelemente haben belegt: size(type(\texttt{ar})) = \left(\prod_{j=1}^n \texttt{dim}_j\right) \cdot size(\texttt{datatype}).
```

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

Code 1.24: Symboltabelle für die Initialisierung eines Feldes.

Im PiocC-ANF Pass in Code 1.25 werden zuerst die Knoten für die Logischen Ausdrücke in den Blättern des Teilbaumes, dessen Wurzel der Feld-Initialisierer-Knoten Array([Array([BinOp(Num('3'), Add('+'),

 $[^]a$ Die im size-Attribut des Symboltabellene
intrags eingetragen ist.

^bHier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern [] bei der Definition eines Feldes hinter die Variable zu schreiben von $L_{\mathbb{C}}$ übernommen. Es wird so getan, als würde der restliche Datentyp komplett vor der Variable stehen: datatype var.

^cDie Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.

^dDie Funktion type ordnet einer Variable ihren Datentyp zu. Das ist notwendig, weil die Funktion size als Definitionsmenge Datentypen hat.

Num('1'))]), Array([Num('5')])]) ist ausgewertet. Die Auswertung geschieht hierbei nach dem Prinzip der Tiefensuche, von links-nach-rechts. Bei dieser Auswertung werden diese Knoten für die Logischen Ausdrücke durch Knoten erstetzt, welche das Ergebnis dieser Ausdrücke auf den Stack schreiben³⁴.

Im finalen Schritt muss zwischen den Globalen Statischen Daten der main-Funktion und dem Stackframe der Funktion fun unterschieden werden. Die auf dem Stack ausgewerteten Logischen Ausdrücke werden mittels der Knoten Assign(Global(Num('0')), Stack(Num('2'))) (für Globale Statische Daten) bzw. Assign(Stackframe(Num('3')), Stack(Num('5'))) (für Stackframe) zu den Globalen Statischen Daten bzw. auf den Stackframe geschrieben.³⁵

Zur Veranschaulichung ist in Tabelle 1.10 ein Ausschnitt des Datensegments nach der Initialisierung des Feldes der Funktion main-Funktion dargestellt. Die auf den Stack ausgewerteten Logischen Ausdrücke sind in grauer Farbe markiert. Die Kopien dieser ausgewerteten Logischen Ausdrücke in den Globalen Statischen Daten, welche die einzelnen Elemente des Feldes darstellen sind in roter Farbe markiert. In Tabelle 1.11 ist das gleiche, allerdings für die Funktion fun und den Stackframe der Funktion fun dargestellt.

Relativ- adresse	Wert	$\operatorname{Register}$
0	4	$^{\mathrm{CS}}$
1	5	
1	5	
2	4	SP

Tabelle 1.10: Ausschnitt des Datensegments nach der Initialisierung des Feldes in der main-Funktion.

Relativ- adresse	Wert	$\operatorname{Register}$
1	6	
2	5	
3	4	
4	3	SP
3	3	
2	4	
1	5	
0	6	
		BAF

Tabelle 1.11: Ausschnitt des Datensegments nach der Initialisierung des Feldes in der Funktion fun.

Der Trick ist hier, dass egal wieviele Dimensionen und was für einen grundlegenden Datentyp³⁶ das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man z.B. mit den Knoten Assign(Global(Num('0')), Stack(Num('2'))) einfach so viele Speicherzellen rüberkopiert, wie das Feld Speicherzellen belegt.

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

³⁵Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

³⁶Z.B. ein Verbund, sodass es ein "Feld von Verbunden" ist.

In die Knoten Global ('0') und Stackframe ('3') wird hierbei die Startadresse des jeweiligen Feldes geschrieben. Daher müssen nach dem PicoC-ANF Pass nie mehr Variablen in der Symboltabelle nachgesehen werden und es ist möglich direkt abzulesen, ob diese in Bezug zu den Globalen Statischen Daten oder dem Stackframe stehen.

```
1 File
    Name './example_array_init.picoc_mon',
     Γ
       Block
         Name 'fun.1',
 6
 7
           // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
           Exp(Num('3'))
 9
           Exp(Num('4'))
10
           Exp(Num('5'))
11
           Exp(Num('6'))
12
           Assign(Stackframe(Num('3')), Stack(Num('4')))
13
           Return(Empty())
14
        ],
15
       Block
16
         Name 'main.0',
17
18
           // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),

    Array([Num('5')]))))

           Exp(Num('3'))
19
20
           Exp(Num('1'))
21
           Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
22
           Exp(Num('5'))
23
           Assign(Global(Num('0')), Stack(Num('2')))
24
           Return(Empty())
25
         ]
26
    ]
```

Code 1.25: PicoC-ANF Pass für die Initialisierung eines Feldes.

Im RETI-Blocks Pass in Code 1.26 werden die PicoC-Knoten Exp(exp) und Assign(Global(Num('0')), Stack(Num('2'))) bzw. Assign(Stackframe(Num('3')), Stack(Num('5'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1 File
    Name './example_array_init.reti_blocks',
    Γ
      Block
5
        Name 'fun.1',
6
          # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           → Num('6')])))
          # Exp(Num('3'))
9
          SUBI SP 1;
10
          LOADI ACC 3;
11
          STOREIN SP ACC 1;
          # Exp(Num('4'))
```

```
SUBI SP 1;
14
           LOADI ACC 4;
15
           STOREIN SP ACC 1;
16
           # Exp(Num('5'))
17
           SUBI SP 1;
18
           LOADI ACC 5;
           STOREIN SP ACC 1;
19
20
           # Exp(Num('6'))
21
           SUBI SP 1;
22
           LOADI ACC 6;
23
           STOREIN SP ACC 1;
24
           # Assign(Stackframe(Num('3')), Stack(Num('4')))
25
           LOADIN SP ACC 1;
26
           STOREIN BAF ACC -2;
27
           LOADIN SP ACC 2;
28
           STOREIN BAF ACC -3;
29
           LOADIN SP ACC 3;
30
           STOREIN BAF ACC -4;
31
           LOADIN SP ACC 4;
32
           STOREIN BAF ACC -5;
33
           ADDI SP 4;
34
           # Return(Empty())
35
           LOADIN BAF PC -1;
36
         ],
37
       Block
38
         Name 'main.0',
39
         Γ
           # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
40
           → Array([Num('5')])))
           # Exp(Num('3'))
           SUBI SP 1;
43
           LOADI ACC 3:
44
           STOREIN SP ACC 1;
45
           # Exp(Num('1'))
46
           SUBI SP 1;
47
           LOADI ACC 1;
48
           STOREIN SP ACC 1;
49
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
50
           LOADIN SP ACC 2;
51
           LOADIN SP IN2 1;
52
           ADD ACC IN2;
53
           STOREIN SP ACC 2;
54
           ADDI SP 1;
55
           # Exp(Num('5'))
56
           SUBI SP 1;
57
           LOADI ACC 5;
58
           STOREIN SP ACC 1;
59
           # Assign(Global(Num('0')), Stack(Num('2')))
60
           LOADIN SP ACC 1;
           STOREIN DS ACC 1;
62
           LOADIN SP ACC 2;
63
           STOREIN DS ACC 0;
64
           ADDI SP 2;
65
           # Return(Empty())
           LOADIN BAF PC -1;
66
67
         ]
68
    ]
```

Code 1.26: RETI-Blocks Pass für die Initialisierung eines Feldes.

1.3.3.2 Zugriff auf einen Feldindex

Die Umsetzung des **Zugriffs auf einen Feldindex** (z.B. ar[0]) wird im Folgenden anhand des Beispiels in Code 1.27 erklärt.

Code 1.27: PicoC-Code für Zugriff auf einen Feldindex.

Der Zugriff auf einen Feldindex ar[0] wird im Abstrakten Syntaxbaum in Code 1.28 mithilfe des Knotens Subscr(Name('ar'), Num('0')) dargestellt.

```
File
    Name './example_array_access.ast',
 4
       FunDef
         VoidType 'void',
         Name 'fun',
 7
8
         [],
 9
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),

    Array([Num('42')]))

10
           Exp(Subscr(Name('ar'), Num('0')))
11
         ],
12
       FunDef
13
         VoidType 'void',
14
         Name 'main',
15
         [],
16
         Γ
17
           Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
           → Array([Num('1'), Num('2'), Num('3')]))
           Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
18
19
         ]
20
```

Code 1.28: Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.

Im PicoC-ANF Pass in Code 1.29 wird zuerst das Schreiben der Adresse einer Variable Name('ar') des Knoten Subscr(Name('ar'), Num('0')) auf den Stack dargestellt. Bei den Globalen Statischen Daten der main-Funktion wird das durch die Knoten Ref(Global(Num('0'))) dargestellt und beim Stackframe

der Funktionm fun wird das durch die Knoten Ref(Stackframe(Num('2'))) dargestellt. Diese Phase wird als Anfangsteil 1.3.5.1 bezeichnet.

Die nächste Phase wird als Mittelteil 1.3.5.2 bezeichnet. In dieser Phase 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. Ein Index eines Feldelements auf das zugegriffen werden soll kann auch durch das Ergebnis eines komplexeren Ausdrucks, wie z.B. ar[1 + var] bestimmt sein, in dem auch Variablen vorkommen. Aus diesem Grund 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, was z.B. im einfachsten Fall durch Exp(Num('0')) dargestellt wird. Danach kann die Adresse des Index berechnet werden, was durch die Knoten Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) dargestellt wird.

In Tabelle 1.12 ist das ganze veranschaulicht. In dem Auschnitt liegt die Startadresse $2^{31} + 67$ des Felds int ar[3] = {1, 2, 3} auf dem Stack und darüber wurde der Wert des Index [1+1] berechnet und auf dem Stack gespeichert (in rot markiert). Der Wert des Index wurde noch nicht auf auf die Startadresse des Felds draufaddiert.³⁷

Absolutadresse	Wert	${f Register}$
$2^{31} + 64$	1	SP
$2^{31} + 65$	2	
$2^{31} + 66$	$2^{31} + 67$	
$2^{31} + 67$	1	
$2^{31} + 68$	2	
$2^{31} + 69$	3	
•••		
• • •		BAF

Tabelle 1.12: Ausschnitt des Datensegments bei der Adressberechnung.

Zur Adressberechnung ist es notwendig auf die Dimensionen (z.B. [Num('3')]) des Feldes, auf dessen Feldelment zugegriffen werden soll, 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 zuvor, 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. Es muss auch möglich sein, z.B. einen Attributzugriff var.attr und einen Zugriff auf einen Arryindex var[1] miteinander zu kombinieren, was in Unterkapitel 1.3.5.2 allgemein erklärt wird.

Die letzte Phase wird als Schlussteil 1.3.5.3 bezeichnet. In dieser Phase wird der Inhalt des Index, dessen Adresse in den vorherigen Schritten berechnet wurde nun auf den Stack geschrieben. Hierfür wird die Adresse, die in den vorherigen Schritten auf dem Stack berechnet wurde verwendet. Beim Schreiben des Inhalts dieses Index auf den Stack, wird dieser die Adresse auf dem Stack ersetzen, die in den vorherigen Schritten berechnet wurde. Dies wird durch den Knoten Exp(Stack(Num('1'))) dargestellt. In Tabelle 1.13 ist das ganze veranschaulicht. In rot ist der Inhalt des Feldindex markiert, der auf den Stack geschrieben wurde.

³⁷Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Absolutadresse	Wert	Register
$2^{31} + 64$	1	
$2^{31} + 65$	2	SP
$2^{31} + 66$	3	
$2^{31} + 67$	1	
$2^{31} + 68$	2	
$2^{31} + 69$	3	
		BAF

Tabelle 1.13: Ausschnitt des Datensegments nach Schlussteil.

Je nachdem auf welchen Unterdatentyp (Definition 1.9) im Kontext zuletzt zugegriffen wird, abhängig davon wird der PicoC-Knoten Exp(Stack(Num('1'))) durch andere semantisch entsprechende RETI-Knoten ersetzt (siehe Unterkapitel 1.3.5.3 für genauere Erklärung). Der Unterdatentyp ist dabei über das versteckte Attribut datatype des Exp(exp, datatype)-Knoten zugänglich.

Definition 1.9: Unterdatentyp

Datentyp, der durch einen Teilbaum dargestellt wird. Dieser Teilbaum ist ein Teil eines Baumes ist, der einen gesamten Datentyp darstellt.

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. Hierbei werden, je nachdem, ob eine Variable in den Globalen Statischen Daten liegt oder sie auf dem Stackframe liegt unterschiedliche semantisch entsprechende RETI-Befehle erzeugt.

Anmerkung Q

Die Berechnung der Adresse, ab der ein Feldelement am Ende einer Aneinanderreihung von Zugriffen auf Feldelemente eines Feldes datatype $ar[dim_1] \dots [dim_n]$ abgespeichert ist^a, kann mittels der Formel 1.3.1:

$$ref(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n]) = ref(\texttt{ar}) + \left(\sum_{i=1}^n \left(\prod_{j=i+1}^n \texttt{dim}_j\right) \cdot \texttt{idx}_i\right) \cdot size(\texttt{datatype}) \tag{1.3.1}$$

aus der Betriebssysteme Vorlesung Scholl, "Betriebssysteme" berechnet werden^{bc}.

Die Knoten Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentieren dabei den Summanden für die Anfangsadresse ref(ar) in der Formel.

Der Knoten Exp(num) repräsentiert dabei einen Index beim Zugriff auf ein Feldelement (z.B. j in a[i][j][k]), der als Faktor idx_i in der Formel auftaucht.

Die Knoten Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) repräsentieren dabei einen ausmultiplizierten Summanden $\left(\prod_{j=i+1}^n \dim_{\mathbf{j}}\right) \cdot \mathrm{idx_i} \cdot size(\mathrm{datatpye})$ in der Formel.

Die Knoten Exp(Stack(Num('1'))) repräsentieren dabei das Lesen des Inhalts

```
M[ref(ar[idx_1]...[idx_n])] der Speicherzelle an der finalen Adresse ref(ar[idx_1]...[idx_n]).
```

^aHier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern [] bei der Definition eines Feldes hinter die Variable zu schreiben von L_{C} übernommen. Es wird so getan, als würde der restliche Datentyp komplett vor der Variable stehen: datatype var.

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

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

Code 1.29: PicoC-ANF Pass für Zugriff auf einen Feldindex.

Im RETI-Blocks Pass in Code 1.30 werden die PicoC-Knoten Ref(Global(Num('0'))), Ref(Subscr(Stack(Num('2'))) undStack(Num('1')))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1 File
2 Name './example_array_access.reti_blocks',
3 [
4 Block
```

^cDie Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.

```
Name 'fun.1',
           # // Assign(Name('ar'), Array([Num('42')]))
           # Exp(Num('42'))
           SUBI SP 1;
10
           LOADI ACC 42;
11
           STOREIN SP ACC 1;
12
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
13
           LOADIN SP ACC 1;
14
           STOREIN BAF ACC -2;
15
           ADDI SP 1;
16
           # // Exp(Subscr(Name('ar'), Num('0')))
17
           # Ref(Stackframe(Num('0')))
18
           SUBI SP 1;
19
           MOVE BAF IN1;
20
           SUBI IN1 2;
21
           STOREIN SP IN1 1;
22
           # Exp(Num('0'))
23
           SUBI SP 1:
24
           LOADI ACC 0;
25
           STOREIN SP ACC 1;
26
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
           LOADIN SP IN1 2;
28
           LOADIN SP IN2 1;
29
           MULTI IN2 1;
30
           ADD IN1 IN2;
           ADDI SP 1;
31
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 'main.0',
42
43
           # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44
           # Exp(Num('1'))
45
           SUBI SP 1;
46
           LOADI ACC 1;
47
           STOREIN SP ACC 1;
48
           # Exp(Num('2'))
49
           SUBI SP 1;
50
           LOADI ACC 2;
51
           STOREIN SP ACC 1;
52
           # Exp(Num('3'))
53
           SUBI SP 1;
54
           LOADI ACC 3;
55
           STOREIN SP ACC 1;
56
           # Assign(Global(Num('0')), Stack(Num('3')))
57
           LOADIN SP ACC 1;
           STOREIN DS ACC 2;
58
59
           LOADIN SP ACC 2;
60
           STOREIN DS ACC 1;
           LOADIN SP ACC 3;
```

```
STOREIN DS ACC 0;
63
           ADDI SP 3;
64
           # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65
           # Ref(Global(Num('0')))
66
           SUBI SP 1;
67
           LOADI IN1 0;
68
           ADD IN1 DS;
69
           STOREIN SP IN1 1;
70
           # Exp(Num('1'))
71
           SUBI SP 1;
72
           LOADI ACC 1;
73
           STOREIN SP ACC 1;
74
           # Exp(Num('1'))
75
           SUBI SP 1;
76
           LOADI ACC 1;
77
           STOREIN SP ACC 1;
78
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79
           LOADIN SP ACC 2;
80
           LOADIN SP IN2 1;
81
           ADD ACC IN2;
82
           STOREIN SP ACC 2;
83
           ADDI SP 1;
84
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85
           LOADIN SP IN1 2;
86
           LOADIN SP IN2 1;
87
           MULTI IN2 1;
88
           ADD IN1 IN2;
89
           ADDI SP 1;
90
           STOREIN SP IN1 1;
91
           # Exp(Stack(Num('1')))
92
           LOADIN SP IN1 1;
93
           LOADIN IN1 ACC O;
94
           STOREIN SP ACC 1;
95
           # Return(Empty())
96
           LOADIN BAF PC -1;
97
         ]
     ]
98
```

Code 1.30: RETI-Blocks Pass für Zugriff auf einen Feldindex.

1.3.3.3 Zuweisung an Feldindex

Die Umsetzung einer **Zuweisung** eines Wertes an einen **Feldinde**x (z.B. ar[2] = 42;) wird im Folgenden anhand des Beispiels in Code 1.31 erläutert.

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

Code 1.31: PicoC-Code für Zuweisung an Feldindex.

Im Abstrakten Syntaxbaum in Code 1.32 wird eine Zuweisung an einen Feldindex ar[2] = 42; durch

die Knoten 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 1.32: Abstrakter Syntaxbaum für Zuweisung an Feldindex.

Im PicoC-ANF Pass in Code 1.33 wird zuerst die rechte Seite des rechtsassoziativen Zuweisungsoperators = bzw. des Knotens der diesen darstellt ausgewertet: Exp(Num('42')). Dies ist in Tabelle 1.14 für das Beispiel in Code 1.31 veranschaulicht. Der Wert 42 (in rot markiert) wurde auf den Stack geschrieben.

${f Absoluta dresse}$	Wert	${f Register}$
$2^{31} + 64$		
$2^{31} + 65$		SP
$2^{31} + 66$	42	
$2^{31} + 67$		
$2^{31} + 68$		
$2^{31} + 69$		
		BAF

Tabelle 1.14: Ausschnitt des Datensegments nach Auswerten der rechten Seite.

Danach ist das Vorgehen und die damit verbundenen Knoten, die dieses 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 Unterkapitel 1.3.3.2. Die eben genannten Knoten stellen die Berechnung der Adresse des Index, dem das Ergebnis des Logischen Ausdrucks auf der rechten Seite des Zuweisungsoperators = zugewiesen wird dar. Dies ist in Tabelle 1.15 für das Beispiel in Code 1.31 veranschaulicht. Die Adresse $2^{31} + 68$ (in rot markiert) des Index wurde auf dem Stack berechnet.

Absolutadresse	Wert	${f Register}$
$2^{31} + 64$	1	SP
$2^{31} + 65$	$2^{31} + 68$	
$2^{31} + 66$	42	
$2^{31} + 67$		
$2^{31} + 68$		
$2^{31} + 69$		
•••		BAF

Tabelle 1.15: Ausschnitt des Datensegments vor Zuweisung.

Zum Schluss stellen die Knoten Assign(Stack(Num('1')), Stack(Num('2'))) die Zuweisung stack(1) = stack(2) des Ergebnisses des Ausdrucks auf der rechten Seite der Zuweisung zum Feldindex dar. Die Adresse des Feldindex wurde im Schritt davor berechnet. Die Zuweisung des Wertes 42 an den Feldindex [1] ist in Tabelle 1.16 veranschaulicht (in rot markiert).³⁸

Absolutadresse	Wert	${f Register}$
$2^{31} + 64$	1	
$2^{31} + 65$	$2^{31} + 68$	
$2^{31} + 66$	42	SP
$2^{31} + 67$		
$2^{31} + 68$	42	
$2^{31} + 69$		
		BAF

Tabelle 1.16: Ausschnitt des Datensegments nach Zuweisung.

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

Code 1.33: PicoC-ANF Pass für Zuweisung an Feldindex.

³⁸Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Im RETI-Blocks Pass in Code 1.34 werden die PicoC-Knoten Exp(Num('42')), Ref(Global(Num('0'))), Exp(Num('1')), Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) und Assign(Stack(Num('1')), Stack(Num('2'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1 File
     Name './example_array_assignment.reti_blocks',
 4
5
       Block
         Name 'main.0',
           # // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
 8
9
           # Exp(Num('42'))
           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;
16
17
           # Exp(Num('1'))
18
           SUBI SP 1;
19
           LOADI ACC 1;
20
           STOREIN SP ACC 1;
21
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22
           LOADIN SP IN1 2;
23
           LOADIN SP IN2 1;
24
           MULTI IN2 1;
25
           ADD IN1 IN2;
26
           ADDI SP 1;
27
           STOREIN SP IN1 1;
28
           # Assign(Stack(Num('1')), Stack(Num('2')))
29
           LOADIN SP IN1 1;
30
           LOADIN SP ACC 2;
31
           ADDI SP 2;
           STOREIN IN1 ACC 0;
32
33
           # Return(Empty())
34
           LOADIN BAF PC -1;
35
         ]
36
     ]
```

Code 1.34: RETI-Blocks Pass für Zuweisung an Feldindex.

1.3.4 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 1.3.4.1 erläutert. Des Weiteren ist die Umsetzung der Innitialisierung eines Verbundes 1.3.4.2, des Zugriffs auf ein Verbundsattribut 1.3.4.3 und der Zuweisung an ein Verbundsattribut 1.3.4.4 zu klären.

1.3.4.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Umsetzung der Deklaration (Definition ??) eines neuen Verbundstyps (z.B. struct st {int len; int ar[2];}) und der Definition (Definition ??) eines Verbundes mit diesem Verbundstyp (z.B. struct st st_var;) wird im Folgenden anhand des Beispiels in Code 1.35 erläutert.

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

Code 1.35: PicoC-Code für die Deklaration eines Verbundstyps.

Bevor ein Verbund definiert werden kann, muss erstmal ein Verbundstyp deklariert werden. Im Abstrakten Syntaxbaum in Code 1.37 wird die Deklaration eines Verbundstyps struct st {int len; int ar[2];} durch die Knoten 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 Knoten 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 1.36: Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.

Für den Verbundstyp selbst und seine Verbundsattribute werden in der Symboltabelle, die in Code 1.37 dargestellt ist Symboltabelleneintrage mit den Schlüsseln st, len@st und ar@st erstellt. Die Schlüssel der

Verbundsattribute haben einen Suffix @st angehängt, welcher für die Verbundsattribute einen Verbundstyps indirekt einen Sichtbarkeitsbereich (Definition ??) über den Verbundstyp selbst erzeugt. Im Unterkapitel 1.3.6.2 wird die Funktionsweise von Sichtbarkeitsbereichen genauer erläutert. Es gilt folglich, dass innerhalb eines Verbundstyps zwei Verbundsattribute nicht gleich benannt werden können, aber dafür zwei unterschiedliche Verbundstypen ihre Verbundsattribute gleich benennen können.

Das Symbol '@' wird aus einem bestimmten Grund als Trennzeichen verwendet, welcher bereits in Unterkapitel 1.3.1.3 erläutert wurde.

Die Attribute³⁹ der Symboltabelleneinträge für die Verbundsattribute sind genauso belegt wie bei üblichen Variablen. Die Attribute des Symboltabelleneintrags für den Verbundstyp type_qualifier, datatype, name, position und size sind wie üblich belegt. In dem value_address-Attribut des Symboltabelleneintrags für den Verbundstyp sind die Verbundsattribute [Name('len@st'), Name('ar@st')] aufgelistet, sodass man über den Verbundstyp st als Schlüssel die Verbundsattribute des Verbundstyps in der Symboltabelle nachschlagen kann.

Für die Definition einer Variable st_var@main mit diesem Verbundstyp st wird ein Symboltabelleneintrag in der Symboltabelle angelegt. Das datatype-Attribut dieses Symboltabelleneintrags enthält dabei den Namen des Verbundstyps als StructSpec(Name('st')). Dadurch können jederzeit alle wichtigen Informationen zu diesem Verbundstyp⁴⁰ und seinen Verbundsattributen in der Symboltabelle nachgeschlagen werden.

Anmerkung 9

Die Anzahl Speicherzellen, die ein Verbund struct st st_var belegt^a, der mit dem Verbundstyp struct st {datatype₁ attr₁; ...; datatype_n attr_n;} definiert ist^b, berechnet sich aus der Summe der Anzahl Speicherzellen, welche die einzelnen Datentypen datatype₁ ... datatype_n der Verbundsattribute attr₁, ... attr_n des Verbundstyps belegen: $size(type(st_var)) = \sum_{i=1}^{n} size(datatype_i)$.

^aDie ihm size-Attribut des Symboltabelleneintrags eingetragen ist.

 b Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern [] bei der Definition eines Feldes hinter die Variable zu schreiben von $L_{\mathbb{C}}$ übernommen. Es wird so getan, als würde der restliche Datentyp komplett vor der Variable stehen: datatype var.

^cDie Funktion size berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.

^dDie Funktion *type* ordnet einer Variable ihren Datentyp zu. Das ist notwendig, weil die Funktion *size* als Definitionsmenge Datentypen hat.

```
SymbolTable
2
     Γ
       Symbol
4
5
6
7
8
         {
                                       Empty()
            type qualifier:
                                       IntType('int')
            datatype:
                                       Name('len@st')
            name:
                                       Empty()
            value or address:
                                       Pos(Num('1'), Num('15'))
            position:
10
            size:
                                       Num('1')
11
         },
12
       Symbol
         {
```

³⁹Die über einen Bezeichner selektierbaren Elemente eines Symboltabelleneintrags und eines Verbunds heißen bei beiden Attribute.

⁴⁰Wie z.B. vor allem die Größe bzw. Anzahl an Speicherzellen, die dieser Verbundstyp einnimmt.

```
type qualifier:
                                     Empty()
15
           datatype:
                                     ArrayDecl([Num('2')], IntType('int'))
                                     Name('ar@st')
16
           name:
17
           value or address:
                                     Empty()
18
                                     Pos(Num('1'), Num('24'))
           position:
19
                                     Num('2')
           size:
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                     Empty()
24
                                     StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
           datatype:
               Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
               Name('ar'))])
                                     Name('st')
           name:
26
           value or address:
                                     [Name('len@st'), Name('ar@st')]
27
                                     Pos(Num('1'), Num('7'))
           position:
28
                                     Num('3')
           size:
29
         },
30
       Symbol
31
         {
32
           type qualifier:
                                     Empty()
33
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
34
                                     Name('main')
           name:
35
           value or address:
                                     Empty()
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')
           name:
44
                                     Num('0')
           value or address:
45
                                     Pos(Num('4'), Num('12'))
           position:
46
                                     Num('3')
           size:
47
         }
```

Code 1.37: Symboltabelle für die Deklaration eines Verbundstyps.

1.3.4.2 Initialisierung von Verbunden

Die Umsetzung der Initialisierung eines Verbundes wird im Folgenden mithilfe des Beispiels in Code 1.38 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 1.38: PicoC-Code für Initialisierung von Verbunden.

Im Abstrakten Syntaxbaum in Code 1.39 wird die Initialisierung eines Verbundes struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}} mithilfe der Knoten Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...)) dargestellt.

```
Name './example_struct_init.ast',
 4
       StructDecl
 5
         Name 'st1',
           Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
           → Name('attr'))
         ],
 9
       StructDecl
10
         Name 'st2',
11
         Γ
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'))]))]))
         ]
22
23
    ]
```

Code 1.39: Abstrakter Syntaxbaum für Initialisierung von Verbunden.

Im PicoC-ANF Pass in Code 1.40 wird Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st1')), Struct(...)) auf fast dieselbe Weise ausgewertet, wie bei der Initialisierung eines Feldes in Unterkapitel 1.3.3.1. Für genauere Details wird an dieser Stelle daher auf Unterkapitel 1.3.3.1 verwiesen. Um das Ganze interessanter zu gestalten, wurde das Beispiel in Code 1.38 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit verschiedenen Datentypen erklären lässt.

Der Teilbaum Struct([Assign(Name('attr1'),Name('var')),Assign(Name('attr2'),Struct([Assign(Name('attr1'),Array([Array([Ref(Name('var')),Ref(Name('var'))]])]))]), der beim äußersten Verbund-Initialisierer-Knoten Struct(...) anfängt, wird auf dieselbe Weise nach dem Prinzip der Tiefensuche von links-nach-rechts ausgewertet, wie es bei der Initialisierung eines Feldes in Unterkapitel 1.3.3.1 bereits erklärt wurde. Beim Iterieren über den Teilbaum, muss bei einem Verbund-Initialisierer-Knoten Struct(...) nur beachtet werden, dass bei den Assign(exp1, exp2)-Knoten⁴¹ der Teilbaum beim rechten exp Attribut weitergeht.

Im Allgemeinen gibt es im Teilbaum beim Initialisieren eines Feldes oder Verbundes auf der rechten Seite immer nur 3 Fälle. Auf der rechten Seite hat man es entweder mit einem Verbund-Initialiser, einem

⁴¹ Über welche die Attributzuweisung (z.B. attr1=var) als z.B. Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))])))))))))

Feld-Initialiser oder einem Logischen Ausdruck zu tun. Bei einem Feld- oder Verbund-Initialiser wird über diesen nach dem Prinzip der Tiefensuche von links-nach-rechts iteriert und mithilfe von Exp(exp)-Knoten die Auswertung der Logischen Ausdrücke in den Blättern auf den Stack dargestellt. Der Fall, dass ein Logischer Ausdruck vorliegt erübrigt sich hiermit.

```
File
    Name './example_struct_init.picoc_mon',
     Γ
      Block
        Name 'main.0',
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
9
           Assign(Global(Num('0')), Stack(Num('1')))
10
           // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
           → Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),

→ Ref(Name('var'))])))))))))))
           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
        ]
    ]
```

Code 1.40: PicoC-ANF Pass für Initialisierung von Verbunden.

Im RETI-Blocks Pass in Code 1.41 werden die PicoC-Knoten Exp(Global(Num('0'))), Ref(Global(Num('0'))) und Assign(Global(Num('1')), Stack(Num('3'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
2
    Name './example_struct_init.reti_blocks',
4
      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;
           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'))]))]))))))))
           # Exp(Global(Num('0')))
18
           SUBI SP 1;
19
           LOADIN DS ACC 0;
           STOREIN SP ACC 1;
```

```
# Ref(Global(Num('0')))
22
           SUBI SP 1;
23
           LOADI IN1 0;
           ADD IN1 DS;
25
           STOREIN SP IN1 1;
26
           # Ref(Global(Num('0')))
27
           SUBI SP 1;
28
           LOADI IN1 0;
29
           ADD IN1 DS;
30
           STOREIN SP IN1 1;
31
           # Assign(Global(Num('1')), Stack(Num('3')))
32
           LOADIN SP ACC 1;
33
           STOREIN DS ACC 3;
34
           LOADIN SP ACC 2;
35
           STOREIN DS ACC 2;
36
           LOADIN SP ACC 3;
37
           STOREIN DS ACC 1;
38
           ADDI SP 3;
39
           # Return(Empty())
40
           LOADIN BAF PC -1;
41
     ]
```

Code 1.41: RETI-Blocks Pass für Initialisierung von Verbunden.

1.3.4.3 Zugriff auf Verbundsattribut

Die Umsetzung des **Zugriffs auf ein Verbundsattribut** (z.B. st.y) wird im Folgenden mithilfe des Beispiels in Code 1.42 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 1.42: PicoC-Code für Zugriff auf Verbundsattribut.

Im Abstrakten Syntaxbaum in Code 1.43 wird der Zugriff auf ein Verbundsattribut st.y mithilfe der Knoten Exp(Attr(Name('st'), Name('y'))) dargestellt.

```
1 File
2  Name './example_struct_attr_access.ast',
3  [
4   StructDecl
5   Name 'pos',
6   [
7    Alloc(Writeable(), IntType('int'), Name('x'))
8   Alloc(Writeable(), IntType('int'), Name('y'))
9  ],
```

Code 1.43: Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.

Im PicoC-ANF Pass in Code 1.44 werden die Knoten Exp(Attr(Name('st'), Name('y'))) auf eine ähnliche Weise ausgewertet, wie die Knoten Exp(Subscr(Name('ar'), Num('0'))), die in Unterkapitel 1.3.3.2 einen Zugriff auf ein Feldelement darstellen. Daher wird hier, um Redundanz zu vermeiden, nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 1.3.3.2 verwiesen.

Die Knoten Exp(Attr(Name('st'), Name('y'))) werden genauso, wie in Unterkapitel 1.3.3.2 durch Knoten ersetzt, die sich in Anfangsteil 1.3.5.1, Mittelteil 1.3.5.2 und Schlussteil 1.3.5.3 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 Unterkapitel 1.3.3.2.

Nur für den Mittelteil werden andere Knoten Ref(Attr(Stack(Num('1')), Name('y'))) gebraucht. Diese Knoten Ref(Attr(Stack(Num('1')), Name('y'))) stellen die Aufgabe dar, die Anfangsadresse des Attributs auf welches zugegriffen wird zu berechnen und auf den Stack zu legen. Hierfür wird die Anfangsadresse des Verbundes, in dem dieses Attribut liegt verwendet. Das auf den Stack-Speichern dieser Anfangsadresse wird durch Knoten des Anfangsteils dargstellt.

Beim Zugriff auf einen Feldindex muss vorher durch z.B. Exp(Num('3')) die Berechnung des Indexwerts und das auf den Stack legen des Ergebnisses dargestellt werden. Beim Zugriff auf ein Verbundsattribut steht der Bezeichner des Verbundsattributs Name('y') dagegen bereits während des Kompilierens in Ref(Attr(Stack(Num('1')), Name('y'))) zur Verfügung. Der Verbundstyp, dem dieses Attribut gehört, wird im Mittelteil aus dem versteckten Attribut datatype des Knoten Ref(exp, datatype) herausgelesen. Der Verbundstyp wird während des Kompiliervorgangs im PiocC-ANF Pass dem Knoten Ref(exp, datatype) über das versteckten Attribut datatype angehängt.

Anmerkung Q

Im Unterkapitel 1.3.5.2 wird mit der allgemeinen Formel 1.3.3 ein allgemeines Vorgehen zur Adressberechnung für alle möglichen Aneinanderreihungen von Zugriffen auf Zeigerelemente, Feldelemente und Verbundsattribute erklärt. Um die Adresse, ab der ein Verbundsattribut am Ende einer Aneinanderreihung von Zugriffen auf Verbundsattribute abgespeichert ist, zu berechnen, kann diese allgemeine Formel 1.3.3 ebenfalls genutzt werden. Im Gegensatz zu Feldern, lässt sich bei Verbunden keine vereinfachte Formel aus der allgemeinen Formel bilden, da die Verbundsattribute eines Verbunds unterschiedlich viele Speicherzellen belegen.

```
1 File
2 Name './example_struct_attr_access.picoc_mon',
3 [
```

```
Block
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
           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
        1
17
    ]
```

Code 1.44: PicoC-ANF Pass für Zugriff auf Verbundsattribut.

Im RETI-Blocks Pass in Code 1.45 werden die PicoC-Knoten Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')),Name('y'))) und Exp(Stack(Num('1'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1 File
    Name './example_struct_attr_access.reti_blocks',
     Γ
 4
       Block
         Name 'main.0',
 6
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           # Exp(Num('4'))
           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
           # // Exp(Attr(Name('st'), Name('y')))
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;
           STOREIN SP IN1 1;
```

```
# Exp(Stack(Num('1')))

LOADIN SP IN1 1;

LOADIN IN1 ACC 0;

STOREIN SP ACC 1;

Return(Empty())

LOADIN BAF PC -1;

| Coadin bar and a coadin
```

Code 1.45: RETI-Blocks Pass für Zugriff auf Verbundsattribut.

1.3.4.4 Zuweisung an Verbundsattribut

Die Umsetzung der **Zuweisung an ein Verbundsattribut** (z.B. st.y = 42) wird im Folgenden anhand des Beispiels in Code 1.46 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 1.46: PicoC-Code für Zuweisung an Verbundsattribut.

Im Abstrakten Syntaxbaum wird eine Zuweisung an ein Verbundsattribut st.y = 42 durch die Knoten Assign(Attr(Name('yt'), Name('y')), Num('42')) dargestellt.

```
1
  File
 2
    Name './example_struct_attr_assignment.ast',
     Γ
       StructDecl
         Name 'pos',
6
7
8
9
           Alloc(Writeable(), IntType('int'), Name('x'))
           Alloc(Writeable(), IntType('int'), Name('y'))
         ],
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
         ]
    ]
```

Code 1.47: Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.

Im PicoC-ANF Pass in Code 1.48 werden die Knoten Assign(Attr(Name('st'), Name('y')), Num('42')) auf eine ähnliche Weise ausgewertet, wie die Knoten Assign(Subscr(Name('ar'), Num('2')), Num('42')), die in Unterkapitel 1.3.3.3 einen Zugriff auf ein Feldelement darstellen. Daher wird hier, um Redundanz zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 1.3.3.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 1.3.3.3 muss hier zum Auswerten des linken Knoten Attr(Name('st'), Name('y')) von Assign(Attr(Name('st'), Name('y')), Num('42')) wie in Unterkapitel 1.3.4.3 vorgegangen werden.

```
File
    Name './example_struct_attr_assignment.picoc_mon',
4
        Name 'main.0',
6
           // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           Exp(Num('4'))
           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 1.48: PicoC-ANF Pass für Zuweisung an Verbundsattribut.

Im RETI-Blocks Pass in Code 1.49 werden die PicoC-Knoten Exp(Num('42')), Ref(Global(Num('0'))), Ref(Attr(Stack(Num('1')), Name('y'))) und Assign(Stack(Num('1')), Stack(Num('2'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
1
  File
2
    Name './example_struct_attr_assignment.reti_blocks',
4
      Block
5
        Name 'main.0',
6
           # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
           → Num('2'))]))
           # Exp(Num('4'))
           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')))
           LOADIN SP ACC 1;
```

```
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;
           LOADIN SP ACC 2;
38
39
           ADDI SP 2;
40
           STOREIN IN1 ACC 0;
41
           # Return(Empty())
42
           LOADIN BAF PC -1;
43
         ]
     ]
```

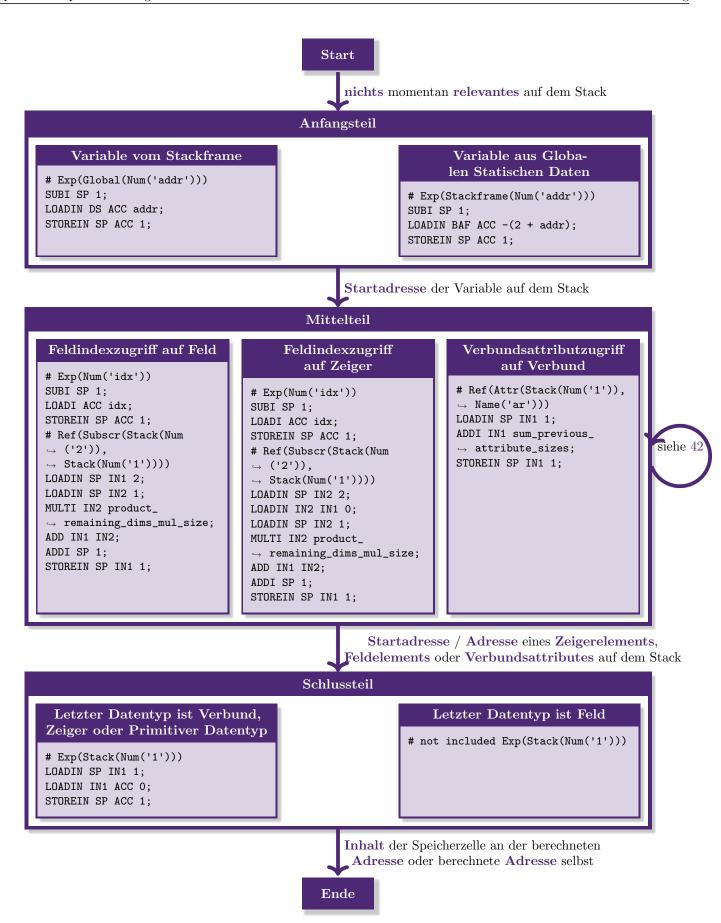
Code 1.49: RETI-Blocks Pass für Zuweisung an Verbndsattribut.

1.3.5 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

In den Unterkapiteln 1.3.2, 1.3.3 und 1.3.4 fällt auf, dass der Zugriff auf Elemente / Attribute der in diesen Kapiteln vorkommenden Datentypen (Zeiger, Feld und Verbund) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem Anfangsteil 1.3.5.1, Mittelteil 1.3.5.2 und Schlussteil 1.3.5.3 darin erkennen. In diesem allgemeinen Vorgehen lassen sich die verschiedenen Zugriffsarten für Elemente bzw. Attribute von Zeigern (z.B. *(pntr + i)), Feldern (z.B. ar[i]) und Verbunden (z.B. st.attr) miteinander kombinieren und so gemischte Ausdrücke, wie z.B. (*st_first.ar)[0] bilden. Dieses allgemeine Vorgehen ist in Abbildung 1.8 veranschaulicht.

Gemischte Ausdrücke sind 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 ein anderer RETI-Code generiert wird. Hierzu muss im exp-Attribut des Ref(exp, datatype)-Knoten die passende Zugriffsoperation Subscr(exp1, exp2) oder Attr(exp, name) vorliegen.

Der gerade erwähnte RETI-Code berechnet die Startadresse eines gewünschten Zeigerelements, Feldelements oder Verbundsattributs. Zur Berechnung wird die Startadresse des Zeigers, Feldes oder Verbundes, dessen Attribut oder Element berechnet werden soll verwendet. Die Startadresse wird in einem vorherigen Berechnungschritt oder im Anfangsteil auf den Stack geschrieben. Bei einem Zugriff auf einen Feldindex wird zudem mithilfe von entsprechendem RETI-Code dafür gesorgt, dass beim Ausführen zur Laufzeit der Wert des Index berechnet wird und nach der Startadresse auf den Stack



geschrieben wird. Dies wurde in Unterkapitel 1.3.3.2 bereits veranschaulicht.

Würde man bei einer Operation Subsc(Name('var'), Num('0')) den Datentyp der Variable Name('var') von ArrayDecl([Num('3')], IntType()) zu PointerDecl(Num('1'), IntType()) ändern, müssten beim generierten RETI-Code nur die RETI-Befehle des Mittelteils ausgetauscht werden. Die RETI-Befehle des Anfangsteils würden unverändert bleiben, da die Variable immer noch entweder in den Globalen Statischen Daten oder in einem Stackframe abgespeichert ist. Die RETI-Befehle des Schlussteils würden unverändert bleiben, da der letzte Datentyp auf den Zugegriffen wird immer noch IntType() ist.

Im Ref(exp, datatype)-Knoten muss die Zugriffsoperation im exp-Attribut zum Datentyp im versteckten Attribut datatype passen. Im Fall, dass Operation und Datentyp nicht zusammenpassen, gibt es eine DatatypeMismatch-Fehlermeldung. Ein Zugriff auf einen Feldindex Subscr(exp1, exp2) kann dabei mit den Datentypen Feld ArrayDecl(nums, datatype) und Zeiger PntrDecl(num, datatype) kombiniert werden. Allerdings wird für beide Kombinationen unterschiedlicher RETI-Code generiert. Das liegt daran, dass in der Speicherzelle des Zeigers PntrDecl(num, datatype) eine Adresse steht und das gewünschte Element erst zu finden ist, wenn man dieser Adresse folgt. Hierfür muss ein anderer RETI-Code erzeugt werden, wie für ein Feld ArrayDecl(nums, datatype), bei dem direkt auf dessen Elemente zugegriffen werden kann. 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. In Unterkapitel 1.3.2.2 wurde bereits erklärt, dass alle Knoten Deref(exp1, exp2) im PicoC-Shrink Pass durch Subscr(exp1, exp2) ersetzt wurden. 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.

Der Anfangsteil, der durch die Knoten Ref(Name('var')) repräsentiert wird, ist dafür zuständig die Startadresse der Variablen Name('var') auf den Stack zu schreiben. Je nachdem, ob diese Variable in den Globalen Statischen Daten oder auf einem Stackframe liegt, wird ein anderer RETI-Code generiert.

Der Schlussteil wird durch die Knoten Exp(Stack(Num('1')), datatype) dargestellt. Wenn das versteckte Attribut datatype ein CharType(), IntType(), PntrDecl(num, datatype) oder StructType(name) ist, wird ein entsprechender RETI-Code generiert. Dieser RETI-Code nutzt die Adresse, die in den vorherigen Phasen auf dem Stack berechnet wurde dazu, um den Inhalt der Speicherzelle an dieser Adresse auf den Stack zu schreiben. Hierbei wird die Speicherzelle, in welcher die Adresse steht mit dem Inhalt auf den sie selbst zeigt überschrieben. 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 wird⁴⁴, 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 Datentypen⁴⁵ wird immer der Inhalt der Speicherzelle der ersten Elements bzw. Elements ausgegeben.

1.3.5.1 Anfangsteil

Die Umsetzung des Anfangsteils, bei dem die Startadresse einer Variable auf den Stack geschrieben wird (z.B. &st), wird im Folgenden mithilfe des Beispiels in Code 1.50 erklärt.

⁴²Startadresse / Adresse eines Zeigerelements, Feldelements oder Verbundsattributes auf dem Stack.

⁴³ Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

⁴⁴Und nicht auf ein Element des Feldes, welches den Datentyp CharType() oder IntType(), PntrDecl(num, datatype) oder StructType(name) hat.

⁴⁵Also CharType(), IntType(), PntrDecl(num, datatype) oder StructType(name).

```
struct ar_with_len {int len; int ar[2];};

void main() {
    struct ar_with_len st_ar[3];
    int *(*complex_var)[3];
    &complex_var;
}

void fun() {
    struct ar_with_len st_ar[3];
    int (*complex_var)[3];
    &complex_var;
}

&complex_var;
}
```

Code 1.50: PicoC-Code für den Anfangsteil.

Im Abstrakten Syntaxbaum in Code 1.51 wird die Refererenzierung &complex_var mit den Knoten Exp(Ref(Name('complex_var'))) dargestellt. Üblicherweise wird für eine Referenzierung einfach nur Ref(Name('complex_var')) geschrieben, aber da beim Erstellen des Abstrakten Syntaxbaums jeder Logische Ausdruck in ein Exp(exp) eingebettet wird, ist das Ref(Name('complex_var')) in ein Exp(exp) eingebettet. Semantisch macht es in diesem Zwischenschritt der Kompilierung keinen Unterschied, ob an einer Stelle Ref(Name('complex_var'))) steht. Man müsste an vielen Stellen eine gesonderte Fallunterschiedung aufstellen, um bei Exp(Ref(Name('complex_var'))) das Exp(exp) zu entfernen. Das Exp(exp) wird allerdings in den darauffolgenden Passes sowieso herausgefiltet. Daher wurde darauf verzichtet den Code ohne triftigen Grund komplexer zu machen.

```
File
    Name './example_derived_dts_introduction_part.ast',
      StructDecl
        Name 'ar_with_len',
7
8
9
          Alloc(Writeable(), IntType('int'), Name('len'))
          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
        ],
10
      FunDef
        VoidType 'void',
12
        Name 'main',
13
        [],
14
        Ε
15
          Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           16
          Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
           → IntType('int')))), Name('complex_var')))
          Exp(Ref(Name('complex_var')))
17
18
        ],
19
      FunDef
20
        VoidType 'void',
21
        Name 'fun',
        [],
        [
```

```
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'))),

Name('complex_var')))

Exp(Ref(Name('complex_var')))

[27]
[28]
```

Code 1.51: Abstrakter Syntaxbaum für den Anfangsteil.

Im PicoC-ANF Pass in Code 1.52 werden die Knoten Exp(Ref(Name('complex_var'))), je nachdem, ob die Variable Name('complex_var') in den Globalen Statischen Daten oder in einem Stackframe liegt durch die Knoten Ref(Global(Num('9'))) oder Ref(Stackframe(Num('9'))) ersetzt.⁴⁶

```
File
 2
    Name './example_derived_dts_introduction_part.picoc_mon',
     Γ
       Block
         Name 'main.1',
 7
8
           // Exp(Ref(Name('complex_var')))
           Ref(Global(Num('9')))
 9
           Return(Empty())
10
         ],
11
       Block
12
         Name 'fun.0',
13
14
           // Exp(Ref(Name('complex_var')))
15
           Ref(Stackframe(Num('9')))
16
           Return(Empty())
17
         ]
18
    ]
```

Code 1.52: PicoC-ANF Pass für den Anfangsteil.

Im RETI-Blocks Pass in Code 1.53 werden die PicoC-Knoten Ref(Global(Num('9'))) bzw. Ref(Stackfra me(Num('9'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

⁴⁶Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
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
         ]
    ]
```

Code 1.53: RETI-Blocks Pass für den Anfangsteil.

1.3.5.2 Mittelteil

Der Umsetzung des Mittelteils, bei dem die Startadresse bzw. Adresse des letzten Attributs oder Elements einer Aneinanderneihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute berechnet wird (z.B. (*complex_var.ar)[2-2]), wird im Folgenden mithilfe des Beispiels in Code 1.54 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 1.54: PicoC-Code für den Mittelteil.

Im Abstrakten Syntaxbaum in Code 1.55 wird die Aneinanderreihung von Zugriffen auf Zeigerelemente, Feldelemente und Verbundsattribute (*complex_var.ar)[2-2] durch die Knoten Exp(Subscr(Deref(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2')))) dargestellt.

```
FunDef
10
         VoidType 'void',
11
        Name 'main',
         [],
           Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),

    Array([Num('42')]))

           Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),
15

    Struct([Assign(Name('ar'), Ref(Name('var')))]))
           Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
16

    Sub('-'), Num('2'))))
         ]
17
18
    ]
```

Code 1.55: Abstrakter Syntaxbaum für den Mittelteil.

Im PicoC-ANF Pass in Code 1.56 werden die Knoten Exp(Subscr(Deref(Attr(Name('complex.var'), Nam e('ar')), Num('0')), BinOp(Num('2'), Sub('-'), Num('2')))) durch die Knoten Ref(Attr(Stack(Num('1')), Name('ar'))), Exp(Num('2')), Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1')))), Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))) ersetzt. Bei z.B. dem S ubscr(exp1,exp2)-Knoten wird dieser einfach dem exp-Attribut des Ref(exp)-Knoten zugewiesen und die Indexberechnung für exp2 davor gezogen. Bei Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) wird über S tack(Num('1')) auf das Ergebnis der Indexberechnung auf dem Stack zugegriffen und über Stack(Num('2')) auf das Ergebnis der Adressberechnung auf dem Stack zugegriffen. Die gerade erwähnte Indexberechnung wird in diesem Fall durch die Knoten Exp(Num(str)) und Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))) dargestellt.

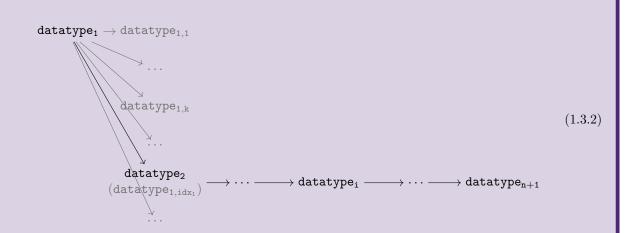
Anmerkung Q

Sei datatype_i ein Folgeglied einer Folge (datatype_i) $_{i=1,...,n+1}$, dessen erstes Folgeglied datatype_i ist. Dabei steht i für eine Ebene eines Baumes. Die Folgeglieder der Folge lassen sich Startadressen $ref(\text{datatype}_i)$ von Speicherbereichen $ref(\text{datatype}_i)$... $ref(\text{datatype}_i) + size(\text{datatype}_i)$ im Hauptspeicher zuordnen. Hierbei gilt, dass $ref(\text{datatype}_i) \le ref(\text{datatype}_{i+1}) < ref(\text{datatype}_i) + size(\text{datatype}_i)$.

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

Sei datatype_{i,idx_i} das Element / Attribut des Datentyps datatype_i für das gilt: datatype_{i,idx_i} = datatype_{i+1}. Hierbei ist idx_i der Index^c des Elements / Attributs auf welches zugegriffen wird innerhalb des Datentyps datatype_i.

In Abbildung 1.3.2 ist das ganze veranschaulicht. Die ausgegrauten Knoten stellen die verschiedenen Elemente / Attribute datatype_{i,k} des Datentyps datatype_i dar. Allerdings können nur die Knoten datatype_i Folgeglieder der Folge (datatype_i) $_{i=1,\ldots,n+1}$ darstellen.



Die Adresse, ab der ein Element / Attribut am Ende einer Folge (datatype_{i,idx_i}) $_{i=1,...,n}$ verschiedener Elemente / Attribute abgespeichert ist, kann mittels der Formel 1.3.3 berechnet werden. Diese Folge ist das Resultat einer Aneinanderreihung von Zugriffen auf Feldelemente und Verbundsattributte unterschiedlicher Datentypen datatype_i (z.B. *complex_var.attr3[2]).

$$ref(\texttt{datatype}_{\texttt{1},\texttt{idx}_1}, \ \dots, \ \texttt{datatype}_{\texttt{n},\texttt{idx}_n}) = ref(\texttt{datatype}_{\texttt{1}}) + \sum_{i=1}^n \sum_{k=1}^{idx_i-1} size(\texttt{datatype}_{\texttt{i},k}) \quad (1.3.3)$$

Die äußere Schleife iteriert nacheinander über die Folge von Attributen / Elementen (datatype $_{i,idx_i}$) $_{i=1,\dots,n}$, die aus den Zugriffen auf Feldelemente oder Verbundsattribute resultiert (z.B. *complex_var.attr3[2]). Die innere Schleife iteriert über alle Elemente oder Attribute datatype $_{i,k}$ des momentan betrachteten Datentyps datatype $_{i,idx_i}$ liegen.

Dabei darf nur das letzte Folgenglied $\mathtt{datatype_{n+1}}$ vom Datentyp Zeiger sein. Ist in einer Folge von Datentypen ein Knoten vom Datentyp Zeiger, der nicht der letzte Datentyp $\mathtt{datatype_{n+1}}$ in der Folge ist, so muss die Adressberechnung in 2 Adressberechnungen aufgeteilt werden. Dabei geht die erste Adressberechnung vom ersten Datentyp $\mathtt{datatype_{1}}$ bis direkt zum Zeiger-Datentyp $\mathtt{datatype_{pntr}}$ und die zweite Adressberechnung fängt einen Datentyp nach dem Zeiger-Datentyp $\mathtt{datatype_{pntr+1}}$ an und geht bis zum letzten Datenyp $\mathtt{datatype_{n}}$. Bei der zweiten Adressberechnung muss dabei die Adresse $ref(\mathtt{datatype_{1}})$ des Summanden aus der Formel 1.3.3 auf den Inhalt^d der Speicherzelle an der Adresse, welche in der ersten Adressberechnung^e $ref(\mathtt{datatype_{1}}, \ldots, \mathtt{datatype_{pntr}})$ berechnet wurde gesetzt werden: M [$ref(\mathtt{datatype_{1}}, \ldots, \mathtt{datatype_{pntr}}$]].

Die Formel 1.3.3 stellt dabei eine Verallgemeinerung der Formel 1.3.1 dar, die für alle möglichen Aneinanderreihungen von Zugriffen auf 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 1.3.1, da man nicht davon ausgehen kann, dass alle Elemente / Attribute den gleichen Datentyp haben^f.

Die Knoten Ref(Global(num)) bzw. Ref(Stackframe(num)) repräsentieren dabei den Summanden $ref(datatype_1)$ in der Formel.

Die Knoten Ref(Attr(Stack(Num('1')), name)) bzw. Ref(Subscr(Stack(Num('2')), Stack(Num('1')))))

```
repräsentieren dabei einen Summanden \sum_{k=1}^{idx_i-1} size(\mathtt{datatype_{i,k}}) in der Formel.
         Knoten
                       Exp(Stack(Num('1')))
                                                     repräsentieren
                                                                            dabei
                                                                                       das
                                                                                                Lesen
                                                                                                           des
                                                                                                                    Inhalts
Die
M[ref(datatype_{1.idx_1}, \ldots, datatype_{n.idx_n})]
                                                            \operatorname{der}
                                                                    Speicherzelle
                                                                                               der
                                                                                                      finalen
                                                                                                                   Adresse
                                                                                        an
ref(\text{datatype}_{1,idx_1}, \ldots, \text{datatype}_{n,idx_n})
aref(datatype) ordent dabei dem Datentyp datatype eine Startadresse zu.
{}^b\mathrm{Die}Funktion \mathtt{size} berechnet die Anzahl Speicherzellen, die ein Datentyp belegt.
^cMan fängt hier bei den Indices von 1 zu zählen an.
<sup>d</sup>Der Inhalt dieser Speicherzelle ist eine Adresse, da im momentanen Kontext ein Zeiger betrachtet wird.
<sup>e</sup>Hierbei kommt die Adresse des Zeigers selbst raus.
<sup>f</sup>Verbundsattribute haben z.B. unterschiedliche Größen.
```

```
File
    Name './example_derived_dts_main_part.picoc_mon',
      Block
        Name 'main.0',
6
7
8
9
           // 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')))
           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'))))
14
           Ref(Global(Num('1')))
15
           Ref(Attr(Stack(Num('1')), Name('ar')))
16
           Exp(Num('0'))
17
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18
           Exp(Num('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
        ]
    ]
```

Code 1.56: PicoC-ANF Pass für den Mittelteil.

Im RETI-Blocks Pass in Code 1.57 werden die PicoC-Knoten Ref(Attr(Stack(Num('1')), Name('ar'))), Exp(Num('2')), Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1')))), Ref(Subscr(Stack(Num('2')), Stack(Num('1')))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt. Bei der Generierung des RETI-Code muss auch das versteckte Attribut datatype des Ref(exp, datatpye)-Knoten berücksichtigt werden, wie es am Anfang dieses Unterkapitels 1.3.5 zusammen mit der Abbildung 1.8 bereits erklärt wurde.

```
File
Name './example_derived_dts_main_part.reti_blocks',

[
```

```
Block
         Name 'main.0',
           # // Assign(Name('var'), Array([Num('42')]))
           # Exp(Num('42'))
           SUBI SP 1:
10
           LOADI ACC 42;
           STOREIN SP ACC 1;
11
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')),
           \rightarrow BinOp(Num('2'), Sub('-'), Num('2'))))
27
           # Ref(Global(Num('1')))
28
           SUBI SP 1;
29
           LOADI IN1 1;
30
           ADD IN1 DS;
31
           STOREIN SP IN1 1;
32
           # Ref(Attr(Stack(Num('1')), Name('ar')))
33
           LOADIN SP IN1 1;
34
           ADDI IN1 0;
           STOREIN SP IN1 1;
35
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'))))
           LOADIN SP IN2 2;
41
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;
           STOREIN SP ACC 1;
52
           # Exp(Num('2'))
53
           SUBI SP 1;
54
           LOADI ACC 2;
55
           STOREIN SP ACC 1;
56
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
57
           LOADIN SP ACC 2;
58
           LOADIN SP IN2 1;
59
           SUB ACC IN2;
```

```
STOREIN SP ACC 2;
           ADDI SP 1;
61
62
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63
           LOADIN SP IN1 2;
           LOADIN SP IN2 1;
65
           MULTI IN2 1;
66
           ADD IN1 IN2;
67
           ADDI SP 1;
68
           STOREIN SP IN1 1;
69
           # Exp(Stack(Num('1')))
70
           LOADIN SP IN1 1;
71
           LOADIN IN1 ACC 0;
72
           STOREIN SP ACC 1;
73
           # Return(Empty())
74
           LOADIN BAF PC -1;
75
         ]
76
    ]
```

Code 1.57: RETI-Blocks Pass für den Mittelteil.

1.3.5.3 Schlussteil

Die Umsetzung des Schlussteils, bei dem ein Attribut oder Element, dessen Adresse im Anfangsteil 1.3.5.1 und Mittelteil 1.3.5.2 auf dem Stack berechnet wurde, auf den Stack gespeichert wird⁴⁷, wird im Folgenden mithilfe des Beispiels in Code 1.58 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 1.58: PicoC-Code für den Schlussteil.

Die Generierung des Abstrakten Syntaxbaumes in Code 1.59 verläuft wie üblich.

```
1 File
2  Name './example_derived_dts_final_part.ast',
3  [
4   StructDecl
5   Name 'st',
```

⁴⁷Und dabei die Speicherzelle der Adresse selbst überschreibt.

```
Γ
           Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
 8
         ],
 9
       FunDef
10
         VoidType 'void',
11
         Name 'main',
12
         [],
13
         Γ
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')))

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

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

    Ref(Name('pntr1')))

           Exp(Subscr(Name('complex_var1'), Num('0')))
19
20
           Exp(Subscr(Name('complex_var2'), Num('0')))
21
           Exp(Deref(Name('complex_var3'), Num('0')))
22
23
    ]
```

Code 1.59: Abstrakter Syntaxbaum für den Schlussteil.

Im PicoC-ANF Pass in Code 1.60 wird das am Anfang dieses Unterkapitels angesprochene auf den Stack speichern des Attributs oder Elements, dessen Adresse in den vorherigen Schritten auf dem Stack berechnet wurde mit den Knoten Exp(Stack(Num('1'))) dargestellt.

```
Name './example_derived_dts_final_part.picoc_mon',
       Block
         Name 'main.0',
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
 9
           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')))
           Ref(Global(Num('2')))
23
           Exp(Num('0'))
24
           Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
           Exp(Stack(Num('1')))
```

```
// Exp(Subscr(Name('complex_var3'), Num('0')))
Ref(Global(Num('6')))
Exp(Num('0'))
Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
Exp(Stack(Num('1')))
Return(Empty())

]
```

Code 1.60: PicoC-ANF Pass für den Schlussteil.

Im RETI-Blocks Pass in Code 1.61 werden die PicoC-Knoten Exp(Stack(Num('1'))) durch semantisch entsprechende RETI-Knoten ersetzt, wenn das versteckte Attribut datatype im Exp(exp,datatype)-Knoten kein Feld ArrayDecl(nums, datatype) enthält. Wenn doch, dann ist bei einem Feld die Adresse, die in vorherigen Schritten auf dem Stack berechnet wurde bereits das gewünschte Ergebnis. Genaueres wurde am Anfang dieses Unterkapitels 1.3.5 zusammen mit der Abbildung 1.8 bereits erklärt.

```
1 File
    Name './example_derived_dts_final_part.reti_blocks',
 4
       Block
         Name 'main.0',
           # // 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('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;
           ADD IN1 DS;
21
           STOREIN SP IN1 1;
22
           # Assign(Global(Num('5')), Stack(Num('1')))
           LOADIN SP ACC 1;
24
           STOREIN DS ACC 5;
25
           ADDI SP 1;
26
           # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
27
           # Ref(Global(Num('5')))
28
           SUBI SP 1;
29
           LOADI IN1 5;
30
           ADD IN1 DS;
31
           STOREIN SP IN1 1;
32
           # Assign(Global(Num('6')), Stack(Num('1')))
33
           LOADIN SP ACC 1;
34
           STOREIN DS ACC 6;
35
           ADDI SP 1;
36
           # // Exp(Subscr(Name('complex_var1'), Num('0')))
           # Ref(Global(Num('0')))
```

```
SUBI SP 1;
39
           LOADI IN1 0;
40
           ADD IN1 DS;
41
           STOREIN SP IN1 1;
           # Exp(Num('0'))
43
           SUBI SP 1;
44
           LOADI ACC 0;
45
           STOREIN SP ACC 1;
46
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47
           LOADIN SP IN1 2;
48
           LOADIN SP IN2 1;
49
           MULTI IN2 2;
50
           ADD IN1 IN2;
51
           ADDI SP 1;
52
           STOREIN SP IN1 1;
53
           # // not included Exp(Stack(Num('1')))
54
           # // Exp(Subscr(Name('complex_var2'), Num('0')))
55
           # Ref(Global(Num('2')))
56
           SUBI SP 1:
57
           LOADI IN1 2;
58
           ADD IN1 DS;
59
           STOREIN SP IN1 1;
60
           # Exp(Num('0'))
61
           SUBI SP 1;
62
           LOADI ACC 0;
63
           STOREIN SP ACC 1;
64
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65
           LOADIN SP IN1 2;
66
           LOADIN SP IN2 1;
67
           MULTI IN2 2;
68
           ADD IN1 IN2;
69
           ADDI SP 1;
70
           STOREIN SP IN1 1;
71
           # Exp(Stack(Num('1')))
72
           LOADIN SP IN1 1;
           LOADIN IN1 ACC O;
74
           STOREIN SP ACC 1;
75
           # // Exp(Subscr(Name('complex_var3'), Num('0')))
76
           # Ref(Global(Num('6')))
           SUBI SP 1;
77
78
           LOADI IN1 6;
79
           ADD IN1 DS;
80
           STOREIN SP IN1 1;
81
           # Exp(Num('0'))
82
           SUBI SP 1;
83
           LOADI ACC 0;
84
           STOREIN SP ACC 1;
85
           # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86
           LOADIN SP IN2 2;
87
           LOADIN IN2 IN1 0;
88
           LOADIN SP IN2 1;
89
           MULTI IN2 1;
90
           ADD IN1 IN2;
91
           ADDI SP 1;
92
           STOREIN SP IN1 1;
93
           # Exp(Stack(Num('1')))
           LOADIN SP IN1 1;
```

```
95 LOADIN IN1 ACC 0;
96 STOREIN SP ACC 1;
97 # Return(Empty())
98 LOADIN BAF PC -1;
99 ]
```

Code 1.61: RETI-Blocks Pass für den Schlussteil.

1.3.6 Umsetzung von Funktionen

Um die Umsetzung von Funktionen zu verstehen, ist es erstmal wichtig zu verstehen, wie Funktionen später im RETI-Code aussehen (Unterkapitel 1.3.6.1), wie Funktionen deklariert (Definition ??) und definiert (Definition ??) werden können und hierbei Sichtbarkeitsbereiche (Definition ??) umgesetzt sind (Unterkapitel 1.3.6.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines Funktionsaufrufes erklärt werden (Unterkapitel 1.3.6.3). Beim Thema Funktionsaufruf wird im speziellen darauf eingegangen werden, wie Rückgabewerte (Unterkapitel 1.3.6.3.1) umgesetzt sind und die Übergabe von Zusammengesetzten Datentypen, die mehr als eine Speicherzelle belegen, wie Verbunden (Unterkapitel 1.3.6.3.3) und Feldern (Unterkapitel 1.3.6.3.2) umgesetzt ist.

1.3.6.1 Mehrere Funktionen

Die Umsetzung mehrerer Funktionen wird im Folgenden mithilfe des Beispiels in Code 1.62 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten Passes übersetzt werden. Das Beispiel ist so gewählt, dass es möglichst isoliert von weiterem möglicherweise störendem Code ist.

```
void main() {
 2
     return;
 4
   void fun1() {
     int var = 41;
     if(1) {
 8
       var = 42;
 9
10 }
11
   int fun2() {
13
     return 1;
14 }
```

Code 1.62: PicoC-Code für 3 Funktionen.

Im Abstrakten Syntaxbaum in Code 1.63 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 1.6 erklärt.

```
1 File
2  Name './verbose_3_funs.ast',
3  [
4  FunDef
5  VoidType 'void',
6  Name 'main',
7  [],
8  [
9  Return
10  Empty
11  ],
12  FunDef
```

```
VoidType 'void',
14
          Name 'fun1',
15
          [],
16
          Γ
17
            Assign
18
              Alloc
19
                 Writeable,
20
                 IntType 'int',
21
                 Name 'var',
22
              Num '41',
23
            Ιf
24
              Num '1',
25
               Γ
26
                 Assign
27
                   Name 'var',
28
                   Num '42'
29
30
          ],
31
       FunDef
          IntType 'int',
32
33
          Name 'fun2',
34
          [],
35
36
            Return
37
              Num '1'
38
          ]
39
     ]
```

Code 1.63: Abstrakter Syntaxbaum für 3 Funktionen.

Im PicoC-Blocks Pass in Code 1.64 werden die Anweisungen der Funktion in Blöcke Block(name, stmts_instrs) aufgeteilt. Hierbei 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) 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 <number> angehängt <name>.<number> 49.50

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>.<number> ist dabei bis auf die angehängte Nummer <number> identisch zu dem der Funktion. Diese Zuordnung ist nötig, da Blöcke eine Nummer an ihren Bezeichner <name>.<number> angehängt haben, die auf anderem Wege nicht ohne großen Aufwand herausgefunden werden kann.

```
1 File
2 Name './verbose_3_funs.picoc_blocks',
3 [
4 FunDef
5 VoidType 'void',
```

⁴⁸Eine Erklärung dazu ist in Unterkapitel 1.3.1.2 zu finden.

 $^{^{49}\}mathrm{Der}$ Grund dafür kann im Unterkapitel1.3.1.2nachgelesen werden.

⁵⁰Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

```
Name 'main',
         [],
         Ε
           Block
10
             Name 'main.4',
11
12
                Return(Empty())
13
14
         ],
15
       {\tt FunDef}
16
         VoidType 'void',
17
         Name 'fun1',
18
         [],
19
         Ε
20
           Block
              Name 'fun1.3',
22
23
                Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
24
                // If(Num('1'), []),
               IfElse
25
26
                  Num '1',
27
                  Γ
28
                    GoTo
29
                      Name 'if.2'
30
                  ],
31
                  [
32
                    GoTo
33
                      Name 'if_else_after.1'
34
                  ]
35
             ],
36
           Block
37
             Name 'if.2',
38
39
                Assign(Name('var'), Num('42'))
40
                GoTo(Name('if_else_after.1'))
41
             ],
42
43
              Name 'if_else_after.1',
44
              []
45
         ],
46
       FunDef
47
         IntType 'int',
48
         Name 'fun2',
49
         [],
50
51
           Block
52
             Name 'fun2.0',
53
54
                Return(Num('1'))
56
         ]
     ]
```

Code 1.64: PicoC-Blocks Pass für 3 Funktionen.

Im PicoC-ANF Pass in Code 1.65 werden die FunDef(datatype, name, allocs, stmts)-Knoten komplett

aufgelöst, sodass sich im File(name, decls_defs_blocks)-Knoten nur noch Blöcke befinden.

```
1 File
    Name './verbose_3_funs.picoc_mon',
 4
       Block
         Name 'main.4',
 6
           Return(Empty())
         ],
 9
       Block
10
         Name 'fun1.3',
           // Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
13
           // Assign(Name('var'), Num('41'))
14
           Exp(Num('41'))
15
           Assign(Stackframe(Num('0')), Stack(Num('1')))
16
           // If(Num('1'), [])
           // IfElse(Num('1'), [], [])
18
           Exp(Num('1')),
19
           IfElse
20
             Stack
               Num '1',
22
             23
               GoTo
24
                 Name 'if.2'
25
             ],
26
             [
27
               GoTo
28
                 Name 'if_else_after.1'
29
             ]
30
         ],
       Block
32
         Name 'if.2',
33
34
           // Assign(Name('var'), Num('42'))
           Exp(Num('42'))
36
           Assign(Stackframe(Num('0')), Stack(Num('1')))
37
           Exp(GoTo(Name('if_else_after.1')))
38
         ],
39
       Block
40
         Name 'if_else_after.1',
41
         Γ
42
           Return(Empty())
43
         ],
44
       Block
45
         Name 'fun2.0',
46
         Γ
47
           // Return(Num('1'))
48
           Exp(Num('1'))
49
           Return(Stack(Num('1')))
50
         ]
    ]
```

Code 1.65: PicoC-ANF Pass für 3 Funktionen.

Nach dem RETI Pass in Code 1.66 gibt es nur noch RETI-Befehle, die Blöcke wurden entfernt. Die RETI-Befehle in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die Kommentare könnte man die RETI-Befehle nicht mehr direkt Funktionen zuordnen. Die Kommentare enthalten die Bezeichner <name>.<number> 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 Adressen gemacht werden. Die Sprünge werden zu den Adressen gemacht, wo die RETI-Befehle anfangen, die aus den Anweisungen einer Funktion kompiliert wurden.

```
# // Block(Name('start.5'), [])
 2 # // Exp(GoTo(Name('main.4')))
3 # // not included Exp(GoTo(Name('main.4')))
 4 # // Block(Name('main.4'), [])
 5 # Return(Emptv())
 6 LOADIN BAF PC -1;
 7 # // Block(Name('fun1.3'), [])
 8 # // Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
 9 # // Assign(Name('var'), Num('41'))
10 # Exp(Num('41'))
11 SUBI SP 1;
12 LOADI ACC 41:
13 STOREIN SP ACC 1:
14 # Assign(Stackframe(Num('0')), Stack(Num('1')))
15 LOADIN SP ACC 1;
16 STOREIN BAF ACC -2;
17 ADDI SP 1;
18 # // If(Num('1'), [])
19 # // IfElse(Num('1'), [], [])
20 # Exp(Num('1'))
21 SUBI SP 1;
22 LOADI ACC 1;
23 STOREIN SP ACC 1;
24 # IfElse(Stack(Num('1')), [], [])
25 LOADIN SP ACC 1;
26 ADDI SP 1:
27 # JUMP== GoTo(Name('if_else_after.1'));
28 JUMP== 7;
29 # GoTo(Name('if.2'))
30 # // not included Exp(GoTo(Name('if.2')))
31 # // Block(Name('if.2'), [])
32 # // Assign(Name('var'), Num('42'))
33 # Exp(Num('42'))
34 SUBI SP 1;
35 LOADI ACC 42;
36 STOREIN SP ACC 1;
37 # Assign(Stackframe(Num('0')), Stack(Num('1')))
38 LOADIN SP ACC 1;
39 STOREIN BAF ACC -2;
40 ADDI SP 1;
41 # Exp(GoTo(Name('if_else_after.1')))
42 # // not included Exp(GoTo(Name('if_else_after.1')))
43 # // Block(Name('if_else_after.1'), [])
44 # Return(Empty())
```

```
45 LOADIN BAF PC -1;
46 # // Block(Name('fun2.0'), [])
47 # // Return(Num('1'))
48 # Exp(Num('1'))
49 SUBI SP 1;
50 LOADI ACC 1;
51 STOREIN SP ACC 1;
52 # Return(Stack(Num('1')))
53 LOADIN SP ACC 1;
54 ADDI SP 1;
55 LOADIN BAF PC -1;
```

Code 1.66: RETI-Blocks Pass für 3 Funktionen.

1.3.6.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 1.62 war die main-Funktion die erste Funktion, die im Code vorkam. Dadurch konnte die main-Funktion direkt betreten werden, da die Ausführung eines 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 1.67 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 1.67: PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

Im RETI-Blocks Pass in Code 1.68 sind die Funktionen nur noch durch Blöcke umgesetzt.

```
1 File
2  Name './verbose_3_funs_main.reti_blocks',
3  [
4   Block
5   Name 'fun1.2',
6   [
7   # Return(Empty())
8   LOADIN BAF PC -1;
9  ],
10  Block
11  Name 'fun2.1',
```

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

Code 1.68: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

Eine simple Möglichkeit die Ausführung durch die main-Funktion zu starten, 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 den ersten Index 0 geschoben werden.

Die Möglichkeit für die sich in der Implementierung des PicoC-Compilers allerdings entschieden wurde, ist es, wenn die main-Funktion nicht die erste auftauchende Funktion ist, einen start.<number>-Block als ersten Block einzufügen. Dieser start.<number>-Block enthält einen GoTo(Name('main.<number>'))-Knoten, der im RETI Pass 1.70 in einen Sprung zur main-Funktion übersetzt wird.⁵¹

In der Implementierung des PicoC-Compilers wurde sich für diese Möglichkeit entschieden, da es für Verwender⁵² 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. <number>-Blockes erfolgt im RETI-Patch Pass in Code 1.69. Der RETI-Patch Pass ist der Pass, der für das Ausbessern⁵³ von Befehlen und Anweisungen zuständig ist, wenn z.B. in manchen Fällen die main-Funktion nicht die erste Funktion ist.

```
1 File
2  Name './verbose_3_funs_main.reti_patch',
3  [
4  Block
5  Name 'start.3',
6  [
7  # // Exp(GoTo(Name('main.0')))
8  Exp(GoTo(Name('main.0')))
9  ],
10  Block
```

⁵¹Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

 $^{^{52}}$ Also die kommenden Studentengenerationen.

⁵³In engl. to patch.

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

Code 1.69: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

Im RETI Pass in Code 1.70 wird das Exp(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 1.70: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

1.3.6.2 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 Fehlermeldungen ausgegeben werden können, wenn der Prototyp (Definition ??) 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 1.71 markierten Zeile 1 oder zusammen mit der Funktionsdefinition (z.B. void fun1(){}), wie in den markierten Zeilen 3-4.

In dem Beispiel in Code 1.71 erfolgt ein Funktionsaufruf der Funktion fun2, die allerdings erst nach der main-Funktion definiert ist. Daher ist eine Funktionsdekleration, wie in der markierten Zeile 1 notwendig. Beim Funktionsaufruf der Funktion fun1 ist das nicht notwendig, da die Funktion vorher definiert wurde, wie in den markierten Zeilen 3-4 zu sehen ist.

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

Code 1.71: Pico C-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.

Die Deklaration einer Funktion erfolgt mithilfe der Symboltabelle, die in Code 1.72 für das Beispiel in Code 1.71 dargestellt ist. Für z.B. die Funktion int fun2(int var) werden die Attribute des Symbols Symbols(type_qual, datatype, name, val_addr, pos, size) wie üblich gesetzt. Dem datatype-Attribut wird dabei einfach die komplette 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??). 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-Down⁵⁴-Iterieren über den Abstrakten Syntaxbaum des aktuellen Passes ein neuer FunDef (datatype, name, allocs, stmts_blocks)-Knoten betreten wird und über dessen Anweisungen im stmts-Attribut iteriert wird. Beim Iterieren über die Anweisungen eines Funktionsknotens wird beim Erstellen neuer Symboltabelleneinträge an die Schlüssel ein Suffix angehängt, der aus dem name-Attribut des Funktionsknotens FunDef (name, datatype, params, stmts_blocks) 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. Des Weiteren lässt sich

⁵⁴D.h. von der Wurzel zu den Blättern eines Baumes.

aus dem Symboltabelleneintrag einer Variable direkt ihr Sichtbarkeitsbereich, in dem sie definiert wurde ablesen. Der Suffix ist ebenfalls im Name(str)-Knoten des name-Attribubtes eines Symboltabelleneintrags der Symboltabelle angehängt. Dies ist in Code 1.72 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 sein müssen⁵⁵, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie definiert wurde.

Das Symbol '@' wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel 1.3.1.3 erläutert wurde.

```
SymbolTable
 2
     Ε
       Symbol
         {
           type qualifier:
                                     Empty()
 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
           type qualifier:
                                     Empty()
15
                                     FunDecl(VoidType('void'), Name('fun1'), [])
           datatype:
16
                                     Name('fun1')
           name:
17
                                     Empty()
           value or address:
                                     Pos(Num('3'), Num('5'))
18
           position:
19
                                     Empty()
           size:
20
         },
21
       Symbol
22
         {
23
           type qualifier:
                                     Empty()
24
                                     FunDecl(VoidType('void'), Name('main'), [])
           datatype:
25
                                     Name('main')
           name:
26
                                     Empty()
           value or address:
27
                                     Pos(Num('6'), Num('5'))
           position:
28
                                     Empty()
           size:
29
         },
30
       Symbol
31
32
                                     Writeable()
           type qualifier:
33
           datatype:
                                     IntType('int')
34
           name:
                                     Name('var@main')
35
                                     Num('0')
           value or address:
36
           position:
                                     Pos(Num('7'), Num('6'))
37
                                     Num('1')
           size:
         },
```

⁵⁵Sonst gibt es eine Fehlermeldung, wie ReDeclarationOrDefinition.

```
Symbol
40
         {
41
           type qualifier:
                                     Writeable()
           datatype:
                                     IntType('int')
43
                                     Name('var@fun2')
           name:
44
           value or address:
                                     Num('0')
45
                                     Pos(Num('12'), Num('13'))
           position:
46
                                     Num('1')
           size:
47
         }
```

Code 1.72: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.

1.3.6.3 Funktionsaufruf

Ein Funktionsaufruf (z.B. stack_fun(local_var)) wird im Folgenden mithilfe des Beispiels in Code 1.73 erklärt. Das Beispiel ist so gewählt, dass alleinig der Funktionsaufruf im Vordergrund steht und das Beispiel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines Rückgabewertes überladen ist. Der Aspekt der Umsetzung eines Rückgabewertes wird erst im nächsten Unterkapitel 1.3.6.3.1 erklärt. 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.

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

Code 1.73: PicoC-Code für Funktionsaufruf ohne Rückgabewert.

Im Abstrakten Syntaxbaum in Code 1.74 wird ein Funktionsaufruf stack_fun(1+1) durch die Knoten Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))])) dargestellt.

```
1 File
2  Name './example_fun_call_no_return_value.ast',
3  [
4   StructDecl
5   Name 'st',
6   [
7    Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8   ],
9  FunDecl
10  VoidType 'void',
```

```
Name 'stack_fun',
12
13
           Alloc
14
             Writeable,
             IntType 'int',
15
16
             Name 'param'
17
         ],
18
       FunDef
19
         VoidType 'void',
20
         Name 'main',
21
         [],
22
23
           Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
           → Name('local_var')))
           Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))
25
           Return(Empty())
26
         ],
27
       FunDef
28
         VoidType 'void',
29
         Name 'stack_fun',
30
           Alloc(Writeable(), IntType('int'), Name('param'))
31
32
33
           Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
34
               Name('local_var')))
35
         ]
36
    ]
```

Code 1.74: Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.

Alle Funktionen außer der main-Funktion besitzen einen Stackframe (Definition 1.10). Bei der main-Funktion werden Lokale Variablen einfach zu den Globalen Statischen Daten geschrieben.

In Tabelle 1.17 ist für das Beispiel in Code 1.73 das Datensegment inklusive Stackframe der Funktion stack_fun mit allen allokierten Variablen dargestellt. Mithilfe der Spalte Relativadresse in der Tabelle 1.17 erklären sich auch die Relativadressen der Variablen local_var@main, local_var@stack_fun, param@stack_fun in den value or address-Attributen der markierten Symboltabelleneinträge in der Symboltabelle in Code 1.75. Bei Stackframes fangen die Relativadressen erst 2 Speicherzellen relativ zum BAF-Register an, da die Rücksprungadresse und die Startadresse des Vorgängerframes Platz brauchen.

Relativ- adresse	Inhalt	${f Register}$
0	$\langle local_var@main \rangle$	CS
1		
2		
3		
		SP
4	$\langle local_var@stack_fun \rangle$	
3		
2		
1		
0	$\langle param_var@stack_fun \rangle$	
	Rücksprungadresse	
• • •	Startadresse Vorgängerframe	BAF

Tabelle 1.17: Datensegment mit Stackframe.

Definition 1.10: Stackframe

Z

Eine Datenstruktur, die dazu dient während der Laufzeit eines Programmes den Zustand einer Funktion "konservieren" zu können, um die Ausführung dieser Funktion später im selben Zustand fortsetzen zu können. Stackframes werden dabei in einem Stack übereinander gestappelt und in die entgegengesetzte Richtung wieder abgebaut, wenn sie nicht mehr benötigt werden. Der Aufbau eines Stackframes ist in Tabelle 1.18 dargestellt.^a

 $\begin{array}{ccc} & & \leftarrow \text{SP} \\ \hline \text{Tempor\"{a}re Berechnungen} & & \\ & \text{Lokale Variablen} & & \\ & \text{Parameter} & \\ & \text{R\"{u}cksprungadresse} \\ \hline \text{Startadresse Vorg\"{a}ngerframe} & \leftarrow \text{BAF} \\ \hline \end{array}$

Tabelle 1.18: Aufbau Stackframe

Üblicherweise steht als erstes^b in einem Stackframe die Startadresse des Vorgängerframes. Diese ist notwendig, damit beim Rücksprung aus einer aufgerufenen Funktion, zurück zur aufrufenden Funktion das BAF-Register wieder so gesetzt werden kann, dass es auf den Stackframe der aufrufenden Funktion zeigt.

Als zweites steht in einem Stackframe üblicherweise die Rücksprungadresse. Die Rücksprungadresse ist die Adresse relativ zum Anfang des Codesegments, an welcher die Ausführung der aufrufenden Funktion nach einem Funktionsaufruf fortgesetzt wird.

Die Startadresse des Vorgängerframes und die Rücksprungadresse stehen beide im Stackframe der aufgerufenen Funktion, da andere Stackframes bei einem Funktionsaufruf nicht mehr so einfach zugänglich sind und diese beiden Informationen für den Rücksprung benötigt werden. Alles weitere in Tabelle 1.18 ist selbsterklärend.

^aWenn von "auf den Stack schreiben" gesprochen wird, dann wird damit gemeint, dass der Bereich für Temporäre Berechnungen (nach Tabelle 1.18) vergrößert wird und ein Wert hinein geschrieben wird.

 $[^]b$ Die Tabelle 1.18 ist von unten-nach-oben zu lesen, da im PicoC-Compiler Stackframes in einem Stack untergebracht

werden, der von unten-nach-oben wächst. Alles soll konsistent dazu gehalten werden, wie es im PicoC-Compiler umgesetzt ist.

^cScholl, "Betriebssysteme".

```
SymbolTable
     Γ
       Symbol
 4
5
           type qualifier:
                                     Empty()
 6
                                     ArrayDecl([Num('2')], IntType('int'))
           datatype:
                                     Name('attr@st')
           name:
                                     Empty()
           value or address:
                                     Pos(Num('1'), Num('15'))
           position:
10
                                     Num('2')
           size:
11
         },
12
       Symbol
13
         {
14
           type qualifier:
                                     Empty()
15
                                     StructDecl(Name('st'), [Alloc(Writeable(),
           datatype:
           → ArrayDecl([Num('2')], IntType('int')), Name('attr'))])
16
           name:
                                     Name('st')
17
                                     [Name('attr@st')]
           value or address:
18
           position:
                                     Pos(Num('1'), Num('7'))
19
           size:
                                     Num('2')
20
         },
21
       Symbol
22
         {
23
           type qualifier:
24
           datatype:
                                     FunDecl(VoidType('void'), Name('stack_fun'),
               [Alloc(Writeable(), IntType('int'), Name('param'))])
                                     Name('stack_fun')
25
           name:
26
           value or address:
                                     Empty()
27
           position:
                                     Pos(Num('3'), Num('5'))
28
           size:
                                     Empty()
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
           position:
                                     Pos(Num('5'), Num('5'))
37
           size:
                                     Empty()
38
         },
39
       Symbol
40
         {
41
           type qualifier:
                                     Writeable()
                                     ArrayDecl([Num('2')], StructSpec(Name('st')))
42
           datatype:
43
                                     Name('local_var@main')
44
           value or address:
                                     Num('0')
45
           position:
                                     Pos(Num('6'), Num('12'))
46
                                     Num('4')
           size:
47
         },
48
       Symbol
49
         {
           type qualifier:
                                     Writeable()
```

```
IntType('int')
           datatype:
52
           name:
                                     Name('param@stack_fun')
53
           value or address:
                                     Num('0')
           position:
                                     Pos(Num('11'), Num('19'))
55
           size:
                                     Num('1')
56
         },
57
       Symbol
58
59
                                     Writeable()
            type qualifier:
60
                                     ArrayDecl([Num('2')], StructSpec(Name('st')))
           datatype:
61
                                     Name('local_var@stack_fun')
           name:
62
           value or address:
                                     Num('4')
63
           position:
                                     Pos(Num('12'), Num('12'))
64
                                     Num('4')
           size:
65
         }
66
     ]
```

Code 1.75: Symboltabelle für Funktionsaufruf ohne Rückgabewert.

Im PicoC-ANF Pass in Code 1.76 werden die Knoten Exp(Call(Name('stack_fun'), [Name('local_var')])) durch die Knoten 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. Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Der Knoten StackMalloc(Num('2')) ist notwendig, weil auf dem Stackframe für den Wert des BAF-Registers der aufrufenden Funktion und die Rücksprungadresse am Anfang des Stackframes 2 Speicherzellen Platz gelassen werden müssen. 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 allokiert wird. 56

```
Name './example_fun_call_no_return_value.picoc_mon',
       Block
         Name 'main.1',
           StackMalloc(Num('2'))
           Exp(Num('1'))
 9
           Exp(Num('1'))
10
           Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
11
           NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
12
           Exp(GoTo(Name('stack_fun.0')))
13
           RemoveStackframe()
14
           Return(Empty())
15
         ],
16
       Block
17
         Name 'stack_fun.0',
18
19
           Return(Empty())
20
```

⁵⁶Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Code 1.76: PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.

Im RETI-Blocks Pass in Code 1.77 werden die PicoC-Knoten 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 semantisch entsprechenden RETI-Knoten ersetzt.

Die Knoten LOADI ACC GoTo(Name('addr@next_instr')) und Exp(GoTo(Name('stack_fun.0'))) sind noch keine RETI-Knoten und werden erst später in dem für sie vorgesehenen RETI-Pass passend ergänzt bzw. ersetzt.

Der Bezeichner des Blocks stack_fun.0 in Exp(GoTo(Name('stack_fun.0'))) wird im Assoziativen Feld fun_name_to_block_name⁵⁷ mit dem Schlüssel stack_fun⁵⁸, der im Knoten NewStackframe(Name('stack_fun')) gespeichert ist nachgeschlagen.

```
File
     Name './example_fun_call_no_return_value.reti_blocks',
       Block
         Name 'main.1',
 6
7
8
           # StackMalloc(Num('2'))
           SUBI SP 2;
 9
           # Exp(Num('1'))
10
           SUBI SP 1;
11
           LOADI ACC 1;
12
           STOREIN SP ACC 1;
13
           # Exp(Num('1'))
14
           SUBI SP 1;
15
           LOADI ACC 1;
16
           STOREIN SP ACC 1;
17
           # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
18
           LOADIN SP ACC 2;
19
           LOADIN SP IN2 1;
20
           ADD ACC IN2;
           STOREIN SP ACC 2;
22
           ADDI SP 1;
23
           # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24
           MOVE BAF ACC;
25
           ADDI SP 3;
26
           MOVE SP BAF;
27
           SUBI SP 7;
28
           STOREIN BAF ACC 0;
29
           LOADI ACC GoTo(Name('addr@next_instr'));
30
           ADD ACC CS;
31
           STOREIN BAF ACC -1;
32
           # Exp(GoTo(Name('stack_fun.0')))
33
           Exp(GoTo(Name('stack_fun.0')))
34
           # RemoveStackframe()
           MOVE BAF IN1;
35
```

 $^{^{57} \}mathrm{Dieses}$ Assoziative Feld wurde in Unterkapitel 1.3.6.1 eingeführt.

⁵⁸Dem Bezeichner der Funktion.

```
LOADIN IN1 BAF 0;
37
           MOVE IN1 SP;
38
           # Return(Empty())
39
           LOADIN BAF PC -1;
40
         ],
41
       Block
42
         Name 'stack_fun.0',
43
44
           # Return(Empty())
45
           LOADIN BAF PC -1;
46
```

Code 1.77: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.

Im RETI Pass in Code 1.77 wird nun der finale RETI-Code generiert. Die RETI-Befehle aus den Blöcken sind nun zusammengefügt und es gibt keine Blöcke mehr. Des Weiteren wird das GoTo(Name('addr@next_instr')) in LOADI ACC GoTo(Name('addr@next_instr')) durch die Adresse des nächsten Befehls direkt nach dem Befehl JUMP 5^{59 60} ersetzt: LOADI ACC 14. 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 versteckten instrs_before-Attributs des Zielblocks Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)⁶¹ und des aktuellen Blocks, in dem der RETI-Knoten JUMP 5 selbst liegt berechnet.

Die relative Adresse 14 des Befehls LOADI ACC 14 wird ebenfalls mithilfe des versteckten instrs_before-Attributs des aktuellen Blocks Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size) berechnet. Es handelt sich bei 14 um eine relative Adresse, die relativ zum CS-Register⁶² berechnet wird.

Anmerkung Q

Die Berechnung der Adresse adr_{danach} bzw. <addr@next_instr> des Befehls nach dem Sprung JUMP <distanz> für den Befehl LOADI ACC <addr@next_instr> erfolgt mithilfe der folgenden Formel 1.3.4:

$$adr_{danach} = \#Bef_{vor\ akt.\ Bl.} + idx + 4 \tag{1.3.4}$$

wobei:

- es sich bei adr_{danach} um eine relative Adresse handelt, die relativ zum CS-Register berechnet wird.
- #Bef_{vor akt. Bl.} Anzahl Befehle vor dem aktuellen 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. Das liegt darin begründet, dass im RETI-Patch Pass Goto()'s entfernt werden, deren Sprung nur eine Adresse weiterspringen würde. Die finale Anzahl an Befehlen kann sich in

⁵⁹Der für den Sprung zur gewünschten Funktion verantwortlich ist.

⁶⁰Also der Befehl, der bisher durch die Komposition Exp(GoTo(Name('stack_fun.0'))) dargestellt wurde.

⁶¹Welcher den ersten Befehl der gewünschten Funktion enthält.

⁶²Welches im RETI-Interpreter von einem Startprogramm im EPROM immer so gesetzt wird, dass es die Adresse enthält, an der das Codesegment anfängt.

diesem Pass also noch ändern und muss daher im letzten Schritt dieses Pass berechnet werden.

- idx = relativer Index des Befehls LOADI ACC <addr@next_instr> selbst im aktuellen Block.
- 4 \(\hat{=}\) Distanz, die zwischen den in Code 1.78 markierten Befehlen LOADI ACC <im> und JUMP <im> liegt und noch eins mehr, weil man ja zum n\(\text{a}\)chsten Befehl will.

Die Berechnug der Distanz^a $Dist_{Zielbl.}$ bzw. <distance> zum ersten Befehl eines im vorhergehenden Pass existenten Blockes^b für den Sprungbefehl JUMP <distance> erfolgt nach der folgenden Formel 1.3.5:

$$Dist_{Zielbl.} = \begin{cases} #Bef_{vor\ Zielbl.} - #Bef_{vor\ akt.\ Bl.} - idx & #Bef_{vor\ Zielbl.}! = #Bef_{vor\ akt.\ Bl.} \\ -idx & #Bef_{vor\ Zielbl.} = #Bef_{vor\ akt.\ Bl.} \end{cases}$$
(1.3.5)

wobei:

- #Bef_{vor Zielbl.} Anzahl Befehle vor dem Zielblock 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 1.3.4.
- idx = relativer Index des Befehls JUMP distance> selbst im aktuellen Block.

In Abbildung 1.9 sind alle 3 möglichen Konstellationen $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$, $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$ und $\#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.}$ veranschaulicht, welche durch die Formel 1.3.5 abgedeckt sind c .

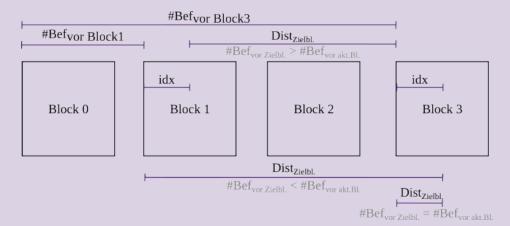


Abbildung 1.9: Veranschaulichung der Dinstanzberechnung

```
# // Exp(GoTo(Name('main.1')))
# // not included Exp(GoTo(Name('main.1')))
# StackMalloc(Num('2'))
# SUBI SP 2;
# Exp(Num('1'))
```

^aDiese **Distanz** kann auch negativ werden.

 $^{{}^}b\mathrm{Im}$ RETI-Pass gibt es keine Blöcke mehr.

^cDie Konstellationen $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$, $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$ sind im Fall $\#Bef_{vor\ Zielbl.}! = \#Bef_{vor\ akt.\ Bl.}$ zusammengefasst, da sie auf die gleiche Weise berechnet werden.

```
6 SUBI SP 1;
 7 LOADI ACC 1;
 8 STOREIN SP ACC 1;
 9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
14 LOADIN SP ACC 2;
15 LOADIN SP IN2 1;
16 ADD ACC IN2;
17 STOREIN SP ACC 2;
18 ADDI SP 1;
19 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
20 MOVE BAF ACC;
21 ADDI SP 3;
22 MOVE SP BAF;
23 SUBI SP 7;
24 STOREIN BAF ACC 0;
25 LOADI ACC 21;
26 ADD ACC CS;
27 STOREIN BAF ACC -1;
28 # Exp(GoTo(Name('stack_fun.0')))
29 JUMP 5;
30 # RemoveStackframe()
31 MOVE BAF IN1;
32 LOADIN IN1 BAF 0;
33 MOVE IN1 SP;
34 # Return(Empty())
35 LOADIN BAF PC -1;
36 # Return(Empty())
37 LOADIN BAF PC -1;
```

Code 1.78: RETI-Pass für Funktionsaufruf ohne Rückgabewert.

1.3.6.3.1 Rückgabewert

Die Umsetzung eines Funktionsaufrufs inklusive Zuweisung eines Rückgabewertes (z.B. int var = fun _with_return_value()) wird im Folgenden mithilfe des Beispiels in Code 1.79 erklärt.

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

```
int fun_with_return_value() {
   return 21 * 2;
3 }

void fun_no_return_value() {
   return;
}

void main() {
```

```
int var = fun_with_return_value();
fun_no_return_value();
}
```

Code 1.79: PicoC-Code für Funktionsaufruf mit Rückgabewert.

Im Abstrakten Syntaxbaum in Code 1.80 wird eine Return-Anweisung mit Rückgabewert return 21 * 2 mit den Knoten Return(BinOp(Num('21'), Mul('*'), Num('2'))) dargestellt, eine Return-Anweisung ohne Rückgabewert return mit den Knoten Return(Empty()) und ein Funktionsaufruf inklusive Zuweisung des Rückgabewertes int var = fun_with_return_value() mit den Knoten Assign(Alloc(Writeable(),IntTy pe('int'),Name('var')),Call(Name('fun_with_return_value'),[])).

```
2
    Name './example_fun_call_with_return_value.ast',
      FunDef
        IntType 'int',
        Name 'fun_with_return_value',
8
9
        Γ
          Return(BinOp(Num('21'), Mul('*'), Num('2')))
10
        ],
11
      FunDef
12
        VoidType 'void',
13
        Name 'fun_no_return_value',
14
        [],
15
        Γ
16
          Return(Empty())
17
        ],
18
      FunDef
19
        VoidType 'void',
20
        Name 'main',
21
        [],
22
        Γ
23
          Assign(Alloc(Writeable(), IntType('int'), Name('var')),
          Exp(Call(Name('fun_no_return_value'), []))
24
25
    ]
```

Code 1.80: Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.

Im PicoC-ANF Pass in Code 1.81 werden bei den Knoten Return(BinOp(Num('21'), Mul('*'), Num('2'))) erst die Knoten BinOp(Num('21'), Mul('*'), Num('2')) ausgewertet. Die hierfür erstellten Knoten 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 den Knoten Return(Stack(Num('1'))) vom Stack gelesen und in das Register ACC geschrieben. Des Weiteren wird vom Return(Stack(Num('1')))-Knoten die Rücksprungadresse in das PC-Register geladen⁶³, um wieder zur aufrufenden Funktion zurückzuspringen.

⁶³Die Rücksprungadresse wurde zuvor durch den NewStackframe()-Knoten (siehe Unterkapitel 1.3.6.3 für Zusammenhang) eine Speicherzelle nach der Speicherzelle auf die das BAF-Register zeigt im Stackframe gespeichert.

Ein wichtiges Detail bei der Funktion int fun_with_return_value() { return 21*2; } ist, dass der Funktionsaufruf Call(Name('fun_with_return_value'), [])) anders übersetzt wird⁶⁴, da diese Funktion einen Rückgabewert vom Datentyp IntType() und nicht VoidType() hat. Bei dieser Übersetzung wird durch die Knoten Exp(ACC) der Rückgabewert der aufgerufenen Funktion für die aufrufende Funktion, deren Stackframe nun wieder der aktuelle ist auf den Stack geschrieben. Der Rückgabewert wurde zuvor in der aufgerufenen Funktion durch die Knoten Return(BinOp(Num('21'), Mul('*), Num('2'))) in das ACC-Register geschrieben.

Dieser Trick mit dem Speichern des Rückgabewerts im ACC-Register ist notwendidg, da der Rückgabewert nicht einfach auf den Stack gespeichert werden kann. Nach dem Entfernen des Stackframes der aufgerufenen Funktion zeigt das SP-Register nicht mehr an die gleiche Stelle. Daher sind alle temporären Werte, die in der aufgerufenen Funktion auf den Stack geschrieben wurden unzugänglich. Man kann nicht wissen, um wieviel die Adresse im SP-Register verglichen zu vorher verschoben ist, weil der Speicherplatz, den Parameter und Lokale Variablen im Stackframe einnehmen bei unterschiedlichen aufgerufenen Funktionen unterschiedlich groß sein kann.

Die Knoten Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[])) vereinen mehrere Aufgaben. Mittels Alloc(Writeable(), IntType('int'), Name('var')) wird die Variable Name('var') allokiert. Die Knoten Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call (Name('fun_with_return_value'),[])) werden durch die Knoten Assign(Global(Num('0')),Stack(Num('1'))) ersetzt, welche den Rückgabewert der Funktion 'fun_with_return_value' nun vom Stack in die Speicherzelle der Variable Name('var') in den Globalen Statischen Daten speichern. Hierzu muss die Adresse der Variable Name('var') in der Symboltabelle nachgeschlagen werden. Der Rückgabewert der Funktion 'fun_with_return_value' wurde zuvor durch die Knoten Exp(Acc) aus dem ACC-Register auf den Stack geschrieben.

Der Umgang mit einer Funktion ohne Rückgabewert wurde am Anfang dieses Unterkapitels 1.3.6.3 bereits besprochen. Für ein return ohne Rückgabewert bleiben die Knoten Return(Empty()) in diesem Pass unverändert, sie stellen nur das Laden der Rücksprungsadresse in das PC-Register dar.

Des Weiteren kann anhand der main-Funktion beobachtet werden, dass wenn bei einer Funktion mit dem Rückgabedatentyp void keine return-Anweisung explizit ans Ende geschrieben wird, im PicoC-ANF Pass eine in Form der Knoten Return(Empty()) hinzufügt wird. Bei Nicht-Angeben wird im Falle eines Rückgabedatentyps, der nicht void ist allerdings eine MissingReturn-Fehlermeldung ausgelöst.

```
File
 2
3
     Name './example_fun_call_with_return_value.picoc_mon',
     Γ
 4
5
6
7
8
       Block
         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
16
           Return(Empty())
17
         ],
```

⁶⁴Als in Unterkapitel 1.3.6.3 bisher erklärt wurde.

```
18
       Block
         Name 'main.0',
19
20
21
           // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
22
           StackMalloc(Num('2'))
           NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
23
24
           Exp(GoTo(Name('fun_with_return_value.2')))
25
           RemoveStackframe()
26
           Exp(ACC)
27
           Assign(Global(Num('0')), Stack(Num('1')))
28
           StackMalloc(Num('2'))
29
           NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
30
           Exp(GoTo(Name('fun_no_return_value.1')))
31
           RemoveStackframe()
32
           Return(Empty())
33
         ]
34
    ]
```

Code 1.81: PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.

Im RETI-Blocks Pass in Code 1.82 werden die PicoC-Knoten 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 semantisch entsprechenden RETI-Knoten ersetzt.

```
1
  File
     Name './example_fun_call_with_return_value.reti_blocks',
 4
       Block
         Name 'fun_with_return_value.2',
 7
8
           # // 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'))))
17
           LOADIN SP ACC 2;
18
           LOADIN SP IN2 1;
19
           MULT ACC IN2;
20
           STOREIN SP ACC 2;
           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
           # Return(Empty())
```

```
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;
           STOREIN SP ACC 1;
56
57
           # Assign(Global(Num('0')), Stack(Num('1')))
58
           LOADIN SP ACC 1;
59
           STOREIN DS ACC 0;
60
           ADDI SP 1;
61
           # StackMalloc(Num('2'))
62
           SUBI SP 2;
63
           # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
64
           MOVE BAF ACC;
65
           ADDI SP 2;
66
           MOVE SP BAF;
67
           SUBI SP 2;
68
           STOREIN BAF ACC 0;
69
           LOADI ACC GoTo(Name('addr@next_instr'));
70
           ADD ACC CS;
71
           STOREIN BAF ACC -1;
72
           # Exp(GoTo(Name('fun_no_return_value.1')))
73
           Exp(GoTo(Name('fun_no_return_value.1')))
74
           # RemoveStackframe()
75
           MOVE BAF IN1;
76
           LOADIN IN1 BAF O;
77
           MOVE IN1 SP;
78
           # Return(Empty())
79
           LOADIN BAF PC -1;
80
    ]
```

Code 1.82: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.

1.3.6.3.2 Umsetzung der Übergabe eines Feldes

Die Eigenheit, dass bei der Übergabe eines Felds an eine andere Funktion, dieses als Zeiger übergeben wird, wurde bereits im Unterkapitel ?? erläutert. Die Umsetzung der Übergabe eines Feldes an eine andere Funktion wird im Folgenden mithilfe des Beispiels in Code 1.83 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);

fun_array_from_global_data(local_var);
}
```

Code 1.83: PicoC-Code für die Übergabe eines Feldes.

Später im PicoC-ANF Pass muss im Fall dessen, dass der Datentyp, der an eine Funktion übergeben wird ein Feld ArrayDecl(nums, datatype) ist, auf spezielle Weise vorgegangen werden. Der oberste Knoten des Teilbaums, der den Feld-Datentyp ArrayDecl(nums, datatype) darstellt, muss zu einem Zeiger PntrDecl(num, datatype) umgewandelt werden und der Rest des Teilbaumes, der am datatype-Attribut hängt, muss an das datatype-Attribut des Zeigers PntrDecl(num, datatype) gehängt werden. Bei einem Mehrdimensionalen Feld fällt eine Dimension an den Zeiger weg und der Rest des Felds wird an das datatype-Attribut des Zeigers PntrDecl(num, datatype) gehängt.

Diese Umwandlung eines Felds zu einem Zeiger kann in der Symboltabelle in Code 1.84 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 werden sie an Parameter 'param@fun_array_from_global_data' und 'param@fun_array_from_stackframe' mit dem Datentyp PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))) gebunden. Die Größe dieser Parameter beträgt dabei Num('1'), da ein Zeiger nur eine Speicherzelle einnimmt.

```
SymbolTable
     Ε
      Symbol
         {
           type qualifier:
                                    Empty()
                                    FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
           datatype:
               [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
               Name('param'))])
                                    Name('fun_array_from_stackframe')
8
           value or address:
                                    Empty()
                                    Pos(Num('1'), Num('5'))
          position:
10
           size:
                                    Empty()
11
        },
12
      Symbol
13
         {
14
           type qualifier:
                                    Writeable()
                                    PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
           datatype:
           name:
                                    Name('param@fun_array_from_stackframe')
```

```
value or address:
                                  Num('0')
18
          position:
                                  Pos(Num('1'), Num('37'))
19
                                  Num('1')
          size:
20
        },
21
      Symbol
22
        {
23
          type qualifier:
                                  Empty()
24
                                  FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
          datatype:
           25
                                  Name('fun_array_from_global_data')
26
          value or address:
                                  Empty()
27
                                  Pos(Num('4'), Num('5'))
          position:
28
                                  Empty()
          size:
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
          datatype:
                                  ArrayDecl([Num('2'), Num('3')], IntType('int'))
43
                                  Name('local_var@fun_array_from_global_data')
          name:
44
          value or address:
                                  Num('6')
45
                                  Pos(Num('5'), Num('6'))
          position:
46
          size:
                                  Num('6')
47
        },
48
      Symbol
49
        {
50
          type qualifier:
                                  Empty()
51
                                  FunDecl(VoidType('void'), Name('main'), [])
          datatype:
52
          name:
                                  Name('main')
53
          value or address:
                                  Empty()
54
          position:
                                  Pos(Num('9'), Num('5'))
55
          size:
                                  Empty()
56
        },
57
      Symbol
58
        {
59
                                  Writeable()
          type qualifier:
60
                                  ArrayDecl([Num('2'), Num('3')], IntType('int'))
          datatype:
61
                                  Name('local_var@main')
          name:
62
                                  Num('0')
          value or address:
63
                                  Pos(Num('10'), Num('6'))
          position:
64
                                  Num('6')
          size:
65
        }
66
    ]
```

Code 1.84: Symboltabelle für die Übergabe eines Feldes.

Im PicoC-ANF Pass in Code 1.85 ist zu sehen, dass zur Übergabe der beiden Felder local_var@main und local_var@fun_array_from_global_data die Adressen der Felder mithilfe der Knoten Ref(Global(Num('0')))

und Ref (Stackframe (Num('6'))) auf den Stack geschrieben werden. Die Knoten Ref (Global (Num('0'))) sind für die Variable local_var aus der main-Funktion, da diese in den Globalen Statischen Daten liegt und die Knoten Ref (Stackframe (Num('6'))) sind für die Variable local_var aus der Funktion fun_array_from_global_data, da diese auf dem Stackframe dieser Funktion liegt.

Die Knoten Ref(Global(Num('0'))) und Ref(Stackframe(Num('6'))) werden später im RETI-Pass durch unterschiedliche RETI-Befehle ersetzt. Hierbei stellen die Zahlen '0' bzw. '6' in den Knoten Global(num) bzw. Stackframe(num), die aus der Symboltabelle entnommen sind die relative Adressen relativ zum DS-Register bzw. SP-Register dar. Die Zahl '6' ergibt sich dadurch, dass das Feld local_var die Dimensionen 2×3 hat und ein Feld von Integern ist, also $size(type(local_var)) = \left(\prod_{j=1}^n \dim_j\right) \cdot size(int) = 2 \cdot 3 \cdot 1 = 6$ Speicherzellen.

```
File
 2
    Name './example_fun_call_by_sharing_array.picoc_mon',
 4
 5
         Name 'fun_array_from_stackframe.2',
           Return(Empty())
 8
         ],
 9
       Block
10
         Name 'fun_array_from_global_data.1',
11
12
           StackMalloc(Num('2'))
13
           Ref(Stackframe(Num('6')))
14
           NewStackframe(Name('fun_array_from_stackframe'), GoTo(Name('addr@next_instr')))
15
           Exp(GoTo(Name('fun_array_from_stackframe.2')))
16
           RemoveStackframe()
17
           Return(Empty())
18
         ],
19
       Block
20
         Name 'main.0',
21
22
           StackMalloc(Num('2'))
23
           Ref(Global(Num('0')))
24
           NewStackframe(Name('fun_array_from_global_data'), GoTo(Name('addr@next_instr')))
25
           Exp(GoTo(Name('fun_array_from_global_data.1')))
26
           RemoveStackframe()
27
           Return(Empty())
28
         ٦
29
    ]
```

Code 1.85: PicoC-ANF Pass für die Übergabe eines Feldes.

Im RETI-Blocks Pass in Code 1.86 werden PicoC-Knoten Ref(Global(Num('0'))) und Ref(Stackframe(Num('6'))) durch ihre entsprechenden RETI-Knoten ersetzt.

```
File
Name './example_fun_call_by_sharing_array.reti_blocks',

| Block
Name 'fun_array_from_stackframe.2',
```

```
Γ
           # 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;
           # 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()
           MOVE BAF IN1:
32
           LOADIN IN1 BAF 0;
33
34
           MOVE IN1 SP;
35
           # Return(Empty())
36
           LOADIN BAF PC -1;
37
        ],
38
       Block
39
         Name 'main.0',
40
41
           # StackMalloc(Num('2'))
42
           SUBI SP 2;
43
           # Ref(Global(Num('0')))
44
           SUBI SP 1;
           LOADI IN1 0;
45
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()
           MOVE BAF IN1;
60
           LOADIN IN1 BAF 0;
61
           MOVE IN1 SP;
```

```
63  # Return(Empty())
64  LOADIN BAF PC -1;
65  ]
66 ]
```

Code 1.86: RETI-Block Pass für die Übergabe eines Feldes.

1.3.6.3.3 Umsetzung der 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 ?? erläutert. Die Umsetzung der Übergabe eines Verbundes wird im Folgenden mithilfe des Beispiels in Code 1.87 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 1.87: PicoC-Code für die Übergabe eines Verbundes.

Im PicoC-ANF Pass in Code 1.88 werden zur Übergabe der beiden Verbunde local_var@main und param@fun_array_from_global_data, die beiden Verbunde mittels der Knoten Assign(Stack(Num('3')), Global(Num('0'))) bzw. Assign(Stack(Num('3')), Stackframe(Num('2'))) jeweils auf den Stack kopiert.

Bei der Übergabe an eine Funktion wird der Zugriff auf einen gesamten Verbund anders gehandhabt als bei einem Feld⁶⁵. Beim einem Feld wurde bei der Übergabe an eine Funktion die Adresse des ersten Feldelements auf den Stack geschrieben. Bei einem Verbund wird bei der Übergabe an eine Funktion 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 übersetzt wird und wieder auf false gesetzt, wenn die Übersetzung des Funktionsaufrufs abgeschlossen ist. Solange die Variable argmode_on auf true gesetzt ist, werden immer die Knoten Assign(Stack(Num('3')), Global(Num('0'))) bzw. Assign(Stack(Num('3')), Stackframe(Num('2'))) für die Ersetzung verwendet. Ist die Variable argmode_on auf false werden die Knoten Ref(Global(num)) bzw. Ref(Stackframe(num)) für die Ersetzung verwendet.

 $^{^{65}\}mathrm{Wie}$ es in Unterkapitel 1.3.6.3.2 erklärt wurde

⁶⁶Die Bedeutung aller hier erwähnten Knoten und Kompositionen von Knoten wird in den Tabellen der Kapitel PicoC-Knoten, RETI-Knoten und Kompositionen von Knoten mit besonderer Bedeutung erläutert.

Die Knoten Assign(Stack(Num('3')), Global(Num('0'))) werden verwendet, da die Verbundsvariable local_var der main-Funktion in den Globalen Statischen Daten liegt und die Knoten Assign(Stack(Num('3')), Stackframe(Num('2'))) werden verwendet, da die Verbundsvariable local_var der Funktion fun_struct_from_global_data im Stackframe der Funktion fun_struct_from_global_data liegt.

```
1 File
    Name './example_fun_call_by_value_struct.picoc_mon',
     Ε
      Block
         Name 'fun_struct_from_stackframe.2',
6
           Return(Empty())
8
        ],
9
      Block
10
        Name 'fun_struct_from_global_data.1',
11
12
           StackMalloc(Num('2'))
13
           Assign(Stack(Num('3')), Stackframe(Num('2')))
14
           NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
15
           Exp(GoTo(Name('fun_struct_from_stackframe.2')))
16
           RemoveStackframe()
17
           Return(Empty())
18
        ],
19
      Block
20
        Name 'main.0',
21
22
           StackMalloc(Num('2'))
23
           Assign(Stack(Num('3')), Global(Num('0')))
24
           NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
25
           Exp(GoTo(Name('fun_struct_from_global_data.1')))
26
           RemoveStackframe()
27
           Return(Empty())
28
        ]
    ]
```

Code 1.88: PicoC-ANF Pass für die Übergabe eines Verbundes.

Im RETI-Blocks Pass in Code 1.89 werden die PicoC-Knoten Assign(Stack(Num('3')), Stackframe(Num('2'))) und Assign(Stack(Num('3')), Global(Num('0'))) durch ihre semantisch entsprechenden RETI-Knoten ersetzt.

```
# StackMalloc(Num('2'))
           SUBI SP 2;
14
15
           # Assign(Stack(Num('3')), Stackframe(Num('2')))
16
17
           LOADIN BAF ACC -4;
18
           STOREIN SP ACC 1;
19
           LOADIN BAF ACC -3;
20
           STOREIN SP ACC 2;
21
           LOADIN BAF ACC -2;
22
           STOREIN SP ACC 3;
23
           # NewStackframe(Name('fun_struct_from_stackframe'), GoTo(Name('addr@next_instr')))
24
           MOVE BAF ACC;
25
           ADDI SP 5;
26
           MOVE SP BAF;
27
           SUBI SP 5;
28
           STOREIN BAF ACC 0;
29
           LOADI ACC GoTo(Name('addr@next_instr'));
30
           ADD ACC CS;
31
           STOREIN BAF ACC -1:
32
           # Exp(GoTo(Name('fun_struct_from_stackframe.2')))
33
           Exp(GoTo(Name('fun_struct_from_stackframe.2')))
34
           # RemoveStackframe()
35
           MOVE BAF IN1;
36
           LOADIN IN1 BAF 0;
37
           MOVE IN1 SP;
38
           # Return(Empty())
39
           LOADIN BAF PC -1;
40
        ],
41
       Block
42
         Name 'main.0',
           # StackMalloc(Num('2'))
45
           SUBI SP 2;
46
           # Assign(Stack(Num('3')), Global(Num('0')))
47
           SUBI SP 3;
48
           LOADIN DS ACC 0;
49
           STOREIN SP ACC 1;
50
           LOADIN DS ACC 1;
51
           STOREIN SP ACC 2;
52
           LOADIN DS ACC 2;
53
           STOREIN SP ACC 3;
54
           # NewStackframe(Name('fun_struct_from_global_data'), GoTo(Name('addr@next_instr')))
55
           MOVE BAF ACC;
56
           ADDI SP 5;
57
           MOVE SP BAF;
58
           SUBI SP 5;
59
           STOREIN BAF ACC 0;
60
           LOADI ACC GoTo(Name('addr@next_instr'));
61
           ADD ACC CS;
62
           STOREIN BAF ACC -1;
63
           # Exp(GoTo(Name('fun_struct_from_global_data.1')))
64
           Exp(GoTo(Name('fun_struct_from_global_data.1')))
65
           # RemoveStackframe()
66
           MOVE BAF IN1;
67
           LOADIN IN1 BAF 0;
           MOVE IN1 SP;
           # Return(Empty())
```

```
70 LOADIN BAF PC -1;
71 ]
72 ]
```

Code 1.89: RETI-Block Pass für die Übergabe eines Verbundes.

Literatur

Online

- 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).
- 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).
- Welcome to Lark's documentation! Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/ (besucht am 31.07.2022).

Bücher

• Nystrom, Robert. Parsing Expressions · Crafting Interpreters. Genever Benning, 2021. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09.07.2022).

Artikel

• 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

- 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).

Sonstige Quellen

- 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).
- Shinan, Erez. lark: a modern parsing library. Version 1.1.2. URL: https://github.com/lark-parser/lark (besucht am 31.07.2022).