# 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 habe<sup>2</sup>, 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

<sup>&</sup>lt;sup>1</sup>Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

<sup>&</sup>lt;sup>2</sup>Wofür 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 Link<sup>5</sup> zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt<sup>6</sup>. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht<sup>7</sup>. 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.

<sup>&</sup>lt;sup>3</sup>Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

<sup>&</sup>lt;sup>4</sup>Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes <sup>3</sup>.

 $<sup>^5</sup>$ https://github.com/michel-giehl/Reti-Emulator.

<sup>&</sup>lt;sup>6</sup>Womit nicht alle Studenten so viel Glück haben.

<sup>&</sup>lt;sup>7</sup>Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

## Inhaltsverzeichnis

Al	obild	ungsv	erzeichnis	I
Co	odeve	erzeich	nnis	II
Ta	belle	enverz	eichnis	III
De	efinit	ionsve	erzeichnis	IV
Gı	ramn	natikv	verzeichnis	$\mathbf{V}$
1	Imp	lemen	ntierung	1
	1.1	Lexika	alische Analyse	3
		1.1.1	Konkrete Grammatik für die Lexikalische Analyse	3
		1.1.2	Codebeispiel	5
	1.2	Syntal	ktische 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	15
			1.2.4.1 Codebeispiel	17
		1.2.5	Generierung des Abstrakten Syntaxbaumes	18
			1.2.5.1 PicoC-Knoten	20
			1.2.5.2 RETI-Knoten	25
			1.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung	26
			1.2.5.4 Abstrakte Grammatik	28
			1.2.5.5 Codebeispiel	29
	1.0	G 1	1.2.5.6 Ausgabe des Abstrakten Syntaxbaumes	30
	1.3		Generierung	31
		1.3.1	Passes	33
			1.3.1.1 PicoC-Shrink Pass	33
			1.3.1.1.1 Aufgabe	33
			1.3.1.1.2 Abstrakte Grammatik	$\frac{34}{35}$
			1.3.1.1.3 Codebeispiel	35 37
			1.3.1.2.1 Aufgabe	37
			1.3.1.2.1 Adagabe	37
			1.3.1.2.3 Codebeispiel	39
			1.3.1.3 PicoC-ANF Pass	40
			1.3.1.3.1 Aufgabe	40
			1.3.1.3.2 Abstrakte Grammatik	41
			1.3.1.3.3 Codebeispiel	43
			1.3.1.4 RETI-Blocks Pass	44
			1.3.1.4.1 Aufgabe	44
			1.3.1.4.2 Abstrakte Grammatik	44
			13143 Codeheispiel	45

	1.3.1.5 RETI-	Patch Pass	48
	1.3.1.5.1	Aufgabe	48
	1.3.1.5.2	Abstrakte Grammatik	49
	1.3.1.5.3	Codebeispiel	49
	1.3.1.6 RETI	Pass	52
	1.3.1.6.1	Aufgabe	52
	1.3.1.6.2	Konkrete und Abstrakte Grammatik	52
	1.3.1.6.3	Codebeispiel	54
<b>.</b>			
Literatur			$\mathbf{A}$

# Abbildungsverzeichnis

1.1	Ableitungsbäume zu den beiden Ableitungen.
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	Cross-Compiler Kompiliervorgang ausgeschrieben
1.7	Cross-Compiler Kompiliervorgang Kurzform
1.8	Architektur mit allen Passes ausgeschrieben

## 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	PicoC-ANF Pass für Codebespiel
1.10	RETI-Blocks Pass für Codebespiel
1.11	RETI-Patch Pass für Codebespiel
1.12	RETI Pass für Codebespiel

# 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 2

# Definitionsverzeichnis

1.1	Metasyntax
1.2	Metasprache
1.3	Erweiterte Backus-Naur-Form (EBNF)
1.4	Dialekt der Erweiterten Backus-Naur-Form aus Lark
1.5	Abstrakte Syntaxform (ASF)
1.6	Earley Parser
1.7	Entarteter Baum
1.8	Symboltabelle 4

# Grammatikverzeichnis

1.1.1 Konkrete Grammatik der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF	Ę
$1.2.1$ 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 $\dots$	29
1.3.1 Abstrakte Grammatik der Sprache $L_{PiocC\_Shrink}$ in ASF	35
1 Trocc Direction	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	45
1.3.5 Abstrakte Grammatik der Sprache $L_{RETI\_Patch}$ in ASF	49
$1.3.6$ Konkrete Grammatik der Sprache $L_{RETI}$ für die Lexikalische Analyse in EBNF	53
	53
1.3.8 Abstrakte Grammatik der Sprache $L_{RETI}$ in ASF	54

# 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 Parsen<sup>1</sup> des Programmes in der Konkreten Syntax der Sprache  $L_{PicoC}$  wird das Lark Parsing Toolkit<sup>2</sup> 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

Z

Eine Sprache, die dazu genutzt wird andere Sprachen zu beschreiben<sup>a</sup>.

<sup>a</sup>Das "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<sup>a</sup> 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<sup>d</sup>.

Weitere Details sind in der Spezifikation des Standards unter Link<sup>e</sup> zu finden. Allerdings werden in der Praxis, wie z.B. in Lark oft eigene abgewandelte Notationen wie in Definition 1.4 verwendet.

<sup>&</sup>lt;sup>1</sup>Wobei beim Parsen auch das Lexen inbegriffen ist.

 $<sup>^2\</sup>mathit{Lark}$  - a parsing toolkit for Python.

<sup>&</sup>lt;sup>3</sup>Shinan, 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<sup>a</sup> 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. ANTLR<sup>4</sup> 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-

<sup>&</sup>lt;sup>a</sup>Der Name kommt daher, dass es eine Erweiterung der Backus-Naur-Form ist, die hier allerdings nicht weiter erläutert wird.

<sup>&</sup>lt;sup>b</sup>Nebel, "Theoretische Informatik".

 $<sup>^</sup>c$  Grammar Reference — Lark documentation.

<sup>&</sup>lt;sup>d</sup>Also von Terminalsymbolen und Nicht-Terminalsymbolen.

ehttps://standards.iso.org/ittf/PubliclyAvailableStandards/s026153\_IS0\_IEC\_14977\_1996(E).zip.

<sup>&</sup>lt;sup>a</sup>https://lark-parser.readthedocs.io/en/latest/grammar.html.

 $<sup>^</sup>b$ Der \*-Quantor bedeutet im Gegensatz zum +-Quantor auch keinmal wiederholen.

 $<sup>^{4}</sup>ANTLR$ .

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 Link<sup>5</sup> 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 Index<sup>6</sup> 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.

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

<sup>&</sup>lt;sup>6</sup>Ein 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<sub>PicoC</sub> 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 erkannten 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="">&gt;b</b>	Bitweise Linksshift und Rechtsshift	
6	a <b a<="b&lt;/th"><th>Kleiner, Kleiner Gleich, Größer, Größer</th><th></th></b>	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<sub>PicoC</sub> 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="">&gt;b</b>
$\operatorname{arith\_and}$	a&b
arith_xor	a^b
arith_or	a b
rel_exp	a <b a="" a<="b">b a&gt;=b</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)<sup>7</sup> 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

<sup>&</sup>lt;sup>7</sup>Geklammerte 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 können<sup>8</sup> (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<sup>a</sup> 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).

<sup>a</sup>Zu 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.

<sup>&</sup>lt;sup>8</sup>Bezogen 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  $ANSI\ C\ grammar\ (Lex)$  angepasst<sup>9</sup>. 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.

 $<sup>^9</sup>$ 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.

<sup>&</sup>lt;sup>10</sup>Wobei 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.<sup>a</sup>

<sup>&</sup>lt;sup>a</sup>Earley, "An efficient context-free parsing".

#### 1.2.3.1 Codebeispiel

Der Ableitungsbaum, der mithilfe des Earley Parsers und der Tokens der Lexikalischen Analyse aus dem Beispiel in Code 1.1 generiert wurde, ist in Code 1.3 zu sehen. 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
                                  int
                  prim_dt
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 ::=-Symbols<sup>11</sup> 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  $\varepsilon$  steht, weil es z.B. Produktionen, wie array\_dims ::= ("["NUM"]")\* gibt, in denen auch das leere Wort  $\varepsilon$  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.

<sup>&</sup>lt;sup>a</sup>Der Außengrad ist die Anzahl ausgehender Kanten.

 $<sup>^</sup>b\mathrm{Der}$  Innengrad ist die Anzahl eingehener Kanten.

 $<sup>^</sup>cB\ddot{a}ume.$ 

<sup>&</sup>lt;sup>11</sup> 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 Felddeclaratoren 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 Name der Variable name(var) wird daher mit dem innersten Datentyp struct st getauscht.

In Abbildung 1.3 ist zu sehen, wie der **Ableitungsbaum** aus Abbildung 1.2 mithilfe eines **Visitors** (Definition ??) vereinfacht wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des Visitors aus dem Lark Parsing Toolkit ist unter Link<sup>12</sup> 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.

 $<sup>^{12}</sup>$ 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.

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

```
decl_part
31
                decl_exp_stmt
32
                   alloc
33
                     pntr_decl
                       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 Link<sup>13</sup> 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-unten<sup>14</sup> 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.

 $<sup>^{13} \</sup>verb|https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.$ 

<sup>&</sup>lt;sup>14</sup>In 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 Aneinandereihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute<sup>15</sup> 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.

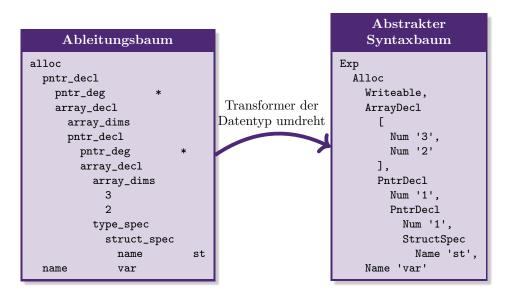


Abbildung 1.5: Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.

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

<sup>&</sup>lt;sup>15</sup>Welche in Unterkapitel ?? 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 sollte<sup>16</sup>. Alle PicoC-Knoten, die in den von den verschiedenen Passes generierten Abstrakten Syntaxbäumen vorkommen sind in Tabelle 1.3 mit einem Beschreibungstext dokumentiert.

<sup>&</sup>lt;sup>16</sup>Z.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.). Das Attribut str ist eine Zeichenkette, welche beliebige Zahlen (0 - 9) enthalten kann. Der Wert, welcher durch die Zeichenkette dargestellt wird, sollte aber nicht größer als $2^{22} - 1$ 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.
<pre>Add(), Sub(), Mul(), Div(), Mod(), LShift(), RShift(), Xor(), And(), Or(), LogicAnd(), LogicOr()</pre>	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, 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>
Exit(num)	Beendet das laufende Programm und schreibt vor der Be- endigung in das ACC Register einen Exit Code num.
Atom(exp, rel, exp) ToBool(exp)	Container für eine binäre Relation: $\langle \exp 1 \rangle \langle \exp 2 \rangle$ 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.
Alloc(type_qual, datatype, name, local_var_or_param)	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
Exp(exp, datatype, error_data)	Container für einen beliebigen Ausdruck, dessen Ergebnis
Exp(exp, datatype, error_data)	
	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>, wobei das Attribut num die Anzahl zusammen-</var>
	gefasster Zeiger angibt und datatype der Datentyp ist, auf
	den der letzte dieser Zeiger zeigt.
Ref(exp, datatype, error_data)	Steht für die Anwendung des Referenz-Operators & <var>,</var>
	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
	und error_data für Fehlermeldungen wichtig ist.
Deref(exp1, exp2)	Container für den Indexzugriff auf einen Feld- oder Zei-
berer (expr, exp2)	gerdatentyp: <var>[<i>], wobei exp1 ein angehängtes weite-</i></var>
	res Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name)
	oder ein Name('var') sein kann und exp2 der Index ist auf
	den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den Felddatentyp steht: <prim_dt></prim_dt>
Allaybeci(nums, datatype)	<pre><var>[<i>], wobei das Attribut nums eine Liste von Num('x')</i></var></pre>
	ist, welche die Dimensionen des Felds angibt und datatype
	•
A	ein Unterdatentyp (Definition ??) ist.
Array(exps, datatype)	Container für den Initializer eines Feldes, dessen Einträge
	exps weitere Initializer für eine Feld-Dimension oder ein
	Initializer für einen Verbund oder ein Logischer Ausdruck
	sein können, z.B. {{1, 2}, {3, 4}}. Des Weiteren besitzt er
	ein verstecktes Attribut datatype, welches für den PicoC-
	ANF Pass Informationen transportiert, die für Fehlermel-
	dungen wichtig sind.
Subscr(exp1, exp2)	Container für den Indexzugriff auf einen Feld- oder Zeiger-
	datentyp: <var>[<i>], wobei exp1 eine angehängte weitere</i></var>
	Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name)
	Operation sein kann oder ein Name('var') sein kann und exp2
	der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten Verbundstyp: struct
	<pre><name>, wobei das Attribut name festlegt, welchen selbst</name></pre>
	definierten Verbundstyp dieser Knoten repräsentiert.
Attr(exp, name)	Container für den Attributzugriff auf einen Verbundsda-
	tentyp: <var>.<attr>, wobei exp1 eine angehängte weitere</attr></var>
	Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name)
	Operation sein kann oder ein Name('var') sein kann und name
	das Attribut ist, auf das zugegriffen werden soll.
	, 30

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

Tabelle 1.5: PicoC-Knoten Teil 3.

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

24

Tabelle 1.6: PicoC-Knoten Teil 4.

#### 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 für alle Befehle: <name> <instrs>, wobei name</instrs></name>
	der Dateiname der Datei ist, die erstellt wird und instrs
	eine Liste von Befehlen ist.
<pre>Instr(op, args)</pre>	Container für einen Befehl: <op> <args>, wobei op eine</args></op>
	Operation ist und args eine Liste von Argumenten
	für dieser Operation.
<pre>Jump(rel, im_goto)</pre>	Container für einen Sprungbefehl: JUMP <rel> <im>, wo-</im></rel>
	bei rel eine Relation ist und im goto ein Immediate
	Value Im(val) für die Anzahl an Speicherzellen, um
	die relativ zum Sprungbefehl gesprungen werden soll
	oder ein GoTo(Name('block.xyz')), das später im RETI-
	Patch Pass durch einen passenden Immediate Value
	ersetzt wird.
Int(num)	Container für einen Interruptaufruf: INT <im>, wobei num</im>
	die Interrruptvektornummer (IVN) für die passende
	Speicherzelle in der Interruptvektortabelle ist, in der
	die Adresse der Interrupt-Service-Routine (ISR) steht.
Call(name, reg)	Container für einen Prozeduraufruf: CALL <name> <reg>,</reg></name>
	wobei name der Bezeichner der Prozedur, die aufgerufen
	werden soll ist und reg ein Register ist, das als Argu-
	ment an die Prozedur dient. Diese Operation ist in der
	Betriebssysteme Vorlesung <sup>a</sup> nicht deklariert, sondern wur-
	de dazuerfunden, um unkompliziert ein CALL PRINT ACC
	oder CALL INPUT ACC im RETI-Interpreter simulieren zu können.
Name(val)	Bezeichner für eine <b>Prozedur</b> , z.B. PRINT oder INPUT oder
Name (Val)	den Programnamen, z.B. PROGRAMNAME. Dieses Argu-
	ment ist in der Betriebssysteme Vorlesung <sup>a</sup> nicht dekla-
	riert, sondern wurde dazuerfunden, um Bezeichner, wie
	PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein Register.
Im(val)	Ein Immediate Value, z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), 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,
<pre>Xori(), Ori(), Andi()</pre>	DIVI, MODI, OPLUSI, ORI, ANDI.
Load(), Loadin(), Loadi()	Load Operationen: LOAD, LOADIN, LOADI.
Store(), Storein(), Move()	Store Operationen: STORE, STOREIN, MOVE.
Lt(), LtE(), Gt(), GtE(), Eq(), NEq(),	Relationen: <, <=, >, >=, ==, !=, _NOP.
Always(), NOp()	
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(),	Register: PC, IN1, IN2, ACC, SP, BAF, CS, DS.
Cs(), Ds()	

<sup>&</sup>lt;sup>a</sup> Scholl, "Betriebssysteme"

Tabelle 1.7: RETI-Knoten.

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

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

## Anmerkung 9

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<sup>a</sup>.

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

<sup>a</sup>Wie z.B. in Deref(exp1, exp2).

## 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 exp	::= ::=   ::=   	$\begin{array}{c cccc} Minus() &   & Not() \\ Add() &   & Sub() &   & Mul() &   & Div() &   & Mod() \\ Oplus() &   & And() &   & Or() \\ Name(\langle str \rangle) &   & Num(\langle str \rangle) &   & Char(\langle str \rangle) \\ BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle) &   & Call(Name('input'), Empty()) \\ Call(Name('print'), \langle exp \rangle) &   & Call(Name('print'), \langle exp \rangle) \end{array}$	$L\_Arith\_Bit$
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	::= ::= ::=	$Const() \mid Writeable() \\ IntType() \mid CharType() \mid VoidType() \\ Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) \\ Assign(\langle exp \rangle, \langle exp \rangle)$	$L\_Assign\_Alloc$
$\begin{array}{c} datatype \\ exp \end{array}$	::= ::=	$PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)$ $Deref(\langle exp \rangle, \langle exp \rangle) \mid Ref(\langle exp \rangle)$	$L\_Pntr$
$\begin{array}{c} datatype \\ exp \end{array}$	::=	$\begin{array}{l} ArrayDecl(Num(\langle str \rangle)+,\langle datatype \rangle) \\ Subscr(\langle exp \rangle,\langle exp \rangle) &   Array(\langle exp \rangle+) \end{array}$	$L\_Array$
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	::= ::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *) \\ Return(\langle exp \rangle) \\ FunDecl(\langle datatype \rangle, Name(\langle str \rangle), \\ Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *) \\ FunDef(\langle datatype \rangle, Name(\langle str \rangle), \\ Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle stmt \rangle *) \\$	$L\_Fun$
		$File(Name(\langle str \rangle), \langle decl\_def \rangle *)$	$L\_File$

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 generiert wurde.

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

```
StructDecl
          Name 'st',
 7
8
            Alloc
              Writeable.
 9
              PntrDecl
10
                 Num '1',
11
                 ArrayDecl
12
13
                     Num '4',
14
                     Num '5'
15
                   ],
16
                   PntrDecl
17
                     Num '1',
18
                     IntType 'int',
19
              Name 'attr'
20
          ],
21
       FunDef
22
          VoidType 'void',
23
          Name 'main',
24
          [],
25
26
            Exp
27
              Alloc
28
                 Writeable,
29
                 ArrayDecl
30
31
                     Num '3',
32
                     Num '2'
33
                   ],
34
                   PntrDecl
35
                     Num '1',
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  $\_repr_-()^{17}$  der Programmiersprache  $L_{Python}$  umgesetzt. Sobald ein PicoC-Knoten oder RETI-Knoten ausgegeben werden soll, gibt seine Magische Methode  $\_repr_-()$  eine nach der Abstrakten oder Konkreten Syntax aufgebaute Textrepräsentation seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten runden öffnenden ( und schließenden ) Klammern, sowie Kommas ',', Semikolons

<sup>&</sup>lt;sup>17</sup>Spezielle Methode, die immer aufgerufen wird, wenn das Object, dass in Besitz dieser Methode ist als String mittels print() oder zur Repräsentation ausgegeben werden soll.

; usw. zur Darstellung der Hierarchie und zur Abtrennung zurück. Dabei wird nach dem Prinzip der Tiefensuche der gesamte Abstrakte Syntaxbaum durchlaufen und die Magische \_\_repr\_\_()-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die \_\_repr\_\_()-Methode ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend zusammenfügen und selbst zurückgeben.

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

Die Ausgabe des Abstrakten Syntaxbaumes ist bewusst so gewählt, dass sie sich optisch vom Ableitungsbaum unterscheidet, indem die Bezeichner der Knoten in UpperCamelCase<sup>18</sup> geschrieben sind, im Gegensatz zum Ableitungsbaum, dessen Innere Knoten im snake\_case geschrieben sind, wie auch die Nicht-Terminalsymbole auf den linken Seiten des ::=-Symbols.

# 1.3 Code Generierung

Nach der Generierung eines Abstrakten Syntaxbaumes als Ergebnis der Lexikalischen und Syntaktischen Analyse in Unterkapitel ??, wird in diesem Kapitel auf Basis der verschiedenen Kompositionen von PicoC-Knoten und RETI-Knoten im Abstrakten Syntaxbaum das gewünschte Endprodukt des PicoC-Compilers, der RETI-Code generiert.

Man steht nun dem Problem gegenüber einen Abstrakten Syntaxbaum der Sprache  $L_{PicoC}$ , der durch die Abstrakte Grammatik 1.2.10 spezifiziert ist in einen entsprechenden Abstrakten Syntaxbaum der Sprache  $L_{RETI}$  umzuformen. Das ganze lässt sich, wie in Unterkapitel ?? bereits beschrieben vereinfachen, indem man dieses Problem in mehrere Passes (Definition ??) herunterbricht.

Beim PicoC-Compiler handelt es sich um einen Cross-Compiler (Definiton ??). Damit RETI-Code erzeugt werden kann, der auf der RETI-Architektur läuft, muss erst, wie im T-Diagram (siehe Unterkapitel ??) in Abbildung 1.6 zu sehen ist, der Python-Code des PicoC-Compilers mittels eines Compilers, der z.B. auf einer  $X_{86.64}$ -Architektur laufen könnte zu Bytecode kompiliert werden. Dieser Bytecode wird dann von der Python-Virtual-Machine (PVM) interpretiert, welche wiederum auf einer  $X_{86.64}$ -Architektur laufen könnte. Und selbst dieses T-Diagram könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die Python-Virtual-Machine geschrieben war, bevor sie zu  $X_{86.64}$ -kompiliert wurde usw.

<sup>&</sup>lt;sup>18</sup> Naming convention (programming).

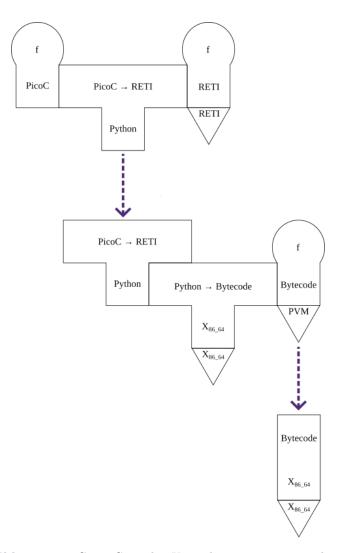


Abbildung 1.6: Cross-Compiler Kompiliervorgang ausgeschrieben.

Dieses längliche T-Diagram in Abbildung 1.6 lässt sich zusammenfassen, sodass man das T-Diagram in Abbildung 1.7 erhält, in welcher direkt angegeben ist, dass der PicoC-Compiler in  $X_{86\_64}$ -Maschinensprache geschrieben ist.

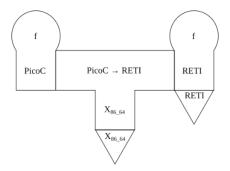


Abbildung 1.7: Cross-Compiler Kompiliervorgang Kurzform.

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

Kompilierprozess im Folgenden im horinzontalen, auf der Ebene der verschiedenen Passes genauer betrachtet. Die Abbildung 1.8 gibt einen guten 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.

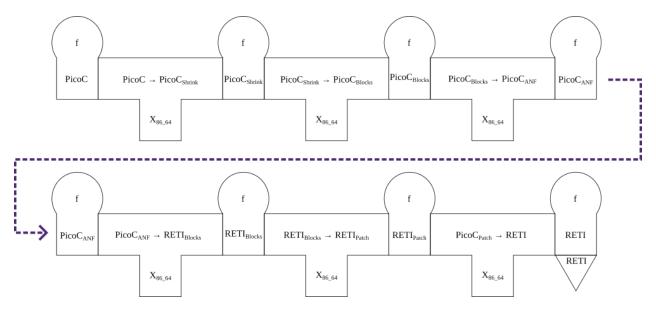


Abbildung 1.8: Architektur mit allen Passes ausgeschrieben.

Im Unterkapitel 1.3.1 werden die unterschiedlichen Passes des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln ??, ??, ?? und ?? 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 ausreichend 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 Aspekte<sup>19</sup> der Bachelorarbeit wichtig.

Auf jedes Detail der einzelnen Passes wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln ??, ??, ?? und ?? 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

#### 1.3.1.1.1 Aufgabe

Der Aufgabe des PicoC-Shrink Pass ist in Unterkapitel ?? ausführlich an einem Beispiel erklärt. Kurzgefasst hat der PicoC-Shrink Pass die Aufgabe, die Eigenheit auszunutzen, dass der Dereferenzierungoperator \*pntr und die damit einhergehende Zeigerarithmetik \*(pntr + i) sich in der Untermenge der Sprache  $L_C$ ,

<sup>&</sup>lt;sup>19</sup>In kurz: Zeiger, Felder, Verbunde und Funktionen.

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

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

#### 1.3.1.1.2 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(exp, exp) in Abstrakten Grammatik 1.3.1 nicht mehr gibt. Das liegt daran, dass dieser Pass alle Vorkommnisse des Knoten Deref(exp, exp) durch den Knoten Subscr(exp, exp) auswechselt, der ebenfalls nach der Abstrakten Grammatik der Sprache  $L_{PicoC}$  definiert ist.

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle)     RETIComment()$	$L_{-}Comment$
un_op bin_op exp	::= ::=	$\begin{array}{c cccc} Minus() &   & Not() \\ Add() &   & Sub() &   & Mul() &   & Div() &   & Mod() \\ Oplus() &   & And() &   & Or() \\ Name(\langle str \rangle) &   & Num(\langle str \rangle) &   & Char(\langle str \rangle) \\ BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle) &   & Call(Name('input'), Empty()) \\ Call(Name('print'), \langle exp \rangle) &   & Call(Name('input'), Empty()) \\ \end{array}$	$L\_Arith\_Bit$
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	::= ::= ::=	$Const() \mid Writeable() \\ IntType() \mid CharType() \mid VoidType() \\ Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) \\ Assign(\langle exp \rangle, \langle exp \rangle)$	$L\_Assign\_Alloc$
$\begin{array}{c} datatype \\ exp \end{array}$	::=	$\begin{array}{c c} PntrDecl(Num(\langle str \rangle), \langle datatype \rangle) \\ \textbf{Deref}(\langle exp \rangle, \langle exp \rangle) &   Ref(\langle exp \rangle) \end{array}$	$L\_Pntr$
$\begin{array}{c} datatype \\ exp \end{array}$	::=	$ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle) Subscr(\langle exp \rangle, \langle exp \rangle)   Array(\langle exp \rangle+)$	L_Array
datatype exp decl_def	::= ::=   ::=	$StructSpec(Name(\langle str \rangle)) \\ Attr(\langle exp \rangle, Name(\langle str \rangle)) \\ Struct(Assign(Name(\langle str \rangle), \langle exp \rangle)+) \\ StructDecl(Name(\langle str \rangle), \\ Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))+) \\$	$L\_Struct$
stmt	::=	$If(\langle exp \rangle, \langle stmt \rangle *)$ $IfElse(\langle exp \rangle, \langle stmt \rangle *, \langle stmt \rangle *)$	$L\_If\_Else$
stmt	::=	$While(\langle exp \rangle, \langle stmt \rangle *) \\ DoWhile(\langle exp \rangle, \langle stmt \rangle *)$	$L\_Loop$
exp stmt decl_def	::= ::= ::=	$Call(Name(\langle str \rangle), \langle exp \rangle *)$ $Return(\langle exp \rangle)$ $FunDecl(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *)$ $FunDef(\langle datatype \rangle, Name(\langle str \rangle),$ $Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle stmt \rangle *)$	L_Fun
		$File(Name(\langle str \rangle), \langle decl\_def \rangle *)$	$L$ _ $File$

Grammatik 1.3.1: Abstrakte Grammatik der Sprache  $L_{PiocC\_Shrink}$  in ASF

# Anmerkung Q

Ein rot markierter Knoten bedeutet, dass dieser im Vergleich zur vorherigen Abstrakten Grammatik nicht mehr da ist.

## 1.3.1.1.3 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", welcher in der Vorlesung allerdings rekursiv implementiert ist.

Dieser rekursive Algoirthmus ist allerdings kein gutes Anschaungsbeispiel, dass viele der Aufgaben der verschiedenen Passes bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der Passes, wie z.B. bei der Kompilierung von if-, if-else-, while- und do-while-Anweisungen wären im Beispiel aus der Vorlesung nicht enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem iterativen Algorithmus 1.6 umgeschrieben, um if- und while-Statemtens zu enthalten.

Beide Varianten des Algorithmus wurden zum Testen des PicoC-Compilers verwendet und sind als Tests im Ordner /tests unter Link<sup>20</sup>, unter den Testbezeichnungen example\_faculty\_rec.picoc und example\_faculty\_it.picoc zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als Anschauung des jeweiligen Passes, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln ??, ??, ?? und ?? zu Zeigern, Feldern, Verbunden und Funktionen mit eigenen Codebeispielen erklärt werden und alle sonstigen Details dem Bachelorprojekt zuzurechnen sind.

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

Code 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',
6   Name 'faculty',
7  [
```

 $<sup>^{20}</sup>$ https://github.com/matthejue/PicoC-Compiler/tree/new\_architecture/tests.

```
Alloc(Writeable(), IntType('int'), Name('n'))
 9
         ],
10
         Γ
11
           Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
12
           While
13
             Num '1',
14
             Γ
               Ιf
16
                 Atom(Name('n'), Eq('=='), Num('1')),
17
18
                   Return(Name('res'))
19
20
               Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21
               Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22
             ]
23
         ],
24
       FunDef
25
         VoidType 'void',
26
         Name 'main',
27
         [],
28
         Γ
           Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])))
29
30
     ]
```

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 enthält.

#### 1.3.1.2 PicoC-Blocks Pass

#### 1.3.1.2.1 Aufgabe

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

#### 1.3.1.2.2 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 im Unterkapitel 1.3.1.2.1 erwähnt wurden. Die Knoten If(exp, stmts), While(exp, stmts) und DoWhile(exp, stmts) gibt es nicht mehr, da sie durch Block(name, stmts\_instrs-, GoTo(lable)-und IfElse(exp, stmts1, stmts2)-Knoten ersetzt wurden. Die Funktionsdefinition FunDef( $\langle datatype \rangle$ , Name(str), Alloc(Writeable(),  $\langle datatype \rangle$ , Name(str))\*,  $\langle block \rangle$ \*) ist nun ein Container für Blöcke Block(Name(str),  $\langle stmt \rangle$ \*) und keine Anweisungen stmt mehr. Das resultiert in der Abstrakten Grammatik 1.3.2 der Sprache  $L_{PicoC\_Blocks}$ .

stmt	::=	$SingleLineComment(\langle str \rangle, \langle str \rangle)     RETIComment()$	$L\_Comment$
un_op bin_op	::=	$Minus() \mid Not()$ $Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$ $Oplus() \mid And() \mid Or()$	$L\_Arith\_Bit$
exp	::=	$Name(\langle str \rangle) \mid Num(\langle str \rangle) \mid Char(\langle str \rangle)$ $BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle)$ $UnOp(\langle un\_op \rangle, \langle exp \rangle) \mid Call(Name('input'), Empty())$ $Call(Name('print'), \langle exp \rangle)$	
stmt	::=	$Exp(\langle exp \rangle)$	
un_op rel bin_op exp	::= ::= ::=	$ \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
$block \\ stmt$	::=	$Block(Name(\langle str \rangle), \langle stmt \rangle *)$ $GoTo(Name(\langle str \rangle))$	$L\_Blocks$

Grammatik 1.3.2: Abstrakte Grammatik der Sprache  $L_{PiocC\_Blocks}$  in ASF

# Anmerkung 9

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

Die Abstrakte Grammatik soll im Gegensatz zur Konkreten Grammatik meist nur vom Programmierer verstanden werden, der den Compiler implementiert und sollte daher vor allem einfach verständlich sein und stellt daher eine Obermenge aller tatsächlich möglichen Kompositionen von Knoten dar<sup>a</sup>.

<sup>a</sup>D.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.

#### 1.3.1.2.3 Codebeispiel

In Code 1.8 sieht man den Abstrakten Syntaxbaum des PiocC-Blocks Passes für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem nun eigene Blöcke für die Funktion faculty und die main-Funktion erstellt werden, in denen die ersten Anweisungen der jeweiligen Funktionen bis zur letzten Anweisung oder bis zum ersten Auftauchen eines If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(exp, stmts)-Knoten stehen. Je nachdem, ob ein If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(exp, stmts)- oder DoWhile(exp, stmts)- Knoten auftaucht, werden für die Bedingung und mögliche Branches eigene Blöcke erstellt.

```
Name './example_faculty_it.picoc_blocks',
 4
       FunDef
 5
         IntType 'int',
 6
         Name 'faculty',
           Alloc(Writeable(), IntType('int'), Name('n'))
 9
         ],
10
         Γ
11
           Block
12
             Name 'faculty.6',
13
14
               Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
15
               // While(Num('1'), [])
16
               GoTo(Name('condition_check.5'))
17
             ],
18
           Block
19
             Name 'condition_check.5',
20
             Γ
21
               IfElse
22
                 Num '1',
23
                 24
                    GoTo(Name('while_branch.4'))
25
                 ],
26
                 Γ
27
                    GoTo(Name('while_after.1'))
28
                 ]
29
             ],
30
           Block
31
             Name 'while_branch.4',
32
33
               // If(Atom(Name('n'), Eq('=='), Num('1')), []),
34
               IfElse
35
                 Atom(Name('n'), Eq('=='), Num('1')),
```

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

Code 1.8: PicoC-Blocks Pass für Codebespiel.

## 1.3.1.3 PicoC-ANF Pass

#### 1.3.1.3.1 Aufgabe

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

Zudem wird eine Symboltabelle (Definition 1.8) eingeführt. In der Symboltabelle wird beim Anlegen eines neuen Eintrags für eine Variable zunächst eine Adresse zugewiesen, die dem Wert einer von zwei Countern rel\_global\_addr und rel\_stack\_addr entspricht. Der Counter rel\_global\_addr ist für Variablen in den Globalen Statischen Daten und der Counter rel\_stack\_addr ist für Variablen auf dem Stackframe. Einer der beiden Counter wird entsprechend der Größe der angelegten Variable hochgezählt.

Kommt im Programmcode an einer späteren Stelle diese Variable Name('symbol') vor, so wird mit dem Symbol<sup>21</sup> als Schlüssel in der Symboltabelle nachgeschlagen und anstelle des Name(str)-Knotens die in

<sup>&</sup>lt;sup>21</sup>Bzw. der Bezeichner

der Symboltabelle nachgeschlagene Adresse in einem Global(Num('addr'))- bzw. Stackframe(Num('addr'))- Knoten eingesetzt eingefügt. Ob der Global(Num('addr'))- oder der Stackframe(Num('addr'))-Knoten zum Einsatz kommt, entscheidet sich anhand des Sichtbarkeitsbereichs (z.B. @scope), der in der Symboltabelle an den Bezeichner drangehängt ist (z.B. identifier@scope).<sup>22</sup>

#### Definition 1.8: Symboltabelle

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

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

<sup>a</sup>In einer Symboltabelle werden Bezeichner als Symbole bezeichnet.

#### 1.3.1.3.2 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, wie z.B. Stack(Num(str)) enthalten können. Des Weiteren werden auch Funktionen und Funktionsaufrufe aufgelöst, sodass u.a. die Blöcke Block(Name(str), stmt\*) nun direkt im File(Name(str), block\*)-Knoten liegen usw., was in Unterkapitel ?? genauer erklärt wird. Die Symboltabelle ist ebenfalls als Abstrakter Syntaxbaum umgesetzt, wofür in der Abstrakten Grammatik 1.3.3 der Sprache  $L_{PicoC\_ANF}$  der Sprache  $L_{PicoC\_ANF}$  neue Knoten eingeführt werden.

Das ganze resultiert in der Abstrakten Grammatik 1.3.3 der Sprache  $L_{PicoC\_ANF}$ .

<sup>&</sup>lt;sup>22</sup>Die Umsetzung von Sichtbarkeitsbereichen wird in Unterkapitel?? genauer beschrieben.

```
RETIComment()
                                                                                                                                                  L_{-}Comment
stmt
                               SingleLineComment(\langle str \rangle, \langle str \rangle)
                      ::=
                                                                                                                                                  L_Arith_Bit
un\_op
                      ::=
                              Minus()
                                                   Not()
bin\_op
                      ::=
                               Add()
                                          Sub()
                                                             Mul() \mid Div() \mid
                                                                                              Mod()
                                                             |Or()
                               Oplus()
                                            And()
                              Name(\langle str \rangle) \mid Num(\langle str \rangle)
                                                                                Char(\langle str \rangle)
                                                                                                         Global(Num(\langle str \rangle))
exp
                               Stackframe(Num(\langle str \rangle))
                                                                       | Stack(Num(\langle str \rangle))|
                               BinOp(Stack(Num(\langle str \rangle)), \langle bin\_op \rangle, Stack(Num(\langle str \rangle)))
                               UnOp(\langle un\_op \rangle, Stack(Num(\langle str \rangle))) \mid Call(Name('input'), Empty())
                               Call(Name('print'), \langle exp \rangle)
                               Exp(\langle exp \rangle)
                              LogicNot()
                                                                                                                                                  L\_Logic
un\_op
                      ::=
                               Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt()
rel
                                                                                                         GtE()
                      ::=
                               LogicAnd()
                                                      LogicOr()
bin\_op
                      ::=
                               Atom(Stack(Num(\langle str \rangle)), \langle rel \rangle, Stack(Num(\langle str \rangle)))
exp
                      ::=
                              ToBool(Stack(Num(\langle str \rangle)))
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
```

#### 1.3.1.3.3 Codebeispiel

In Code 1.9 sieht man den Abstrakten Syntaxbaum des PiocC-ANF Passes für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem alle Anweisungen und Ausdrücke in A-Normalform sind. Die IfElse(exp, stmts, stmts)-Knoten sind hier in A-Normalform gebracht worden, indem ihre Komplexe Bedingung vorgezogen wurde und das Ergebnis der Komplexen Bedingung einer Location zugewiesen ist und sie selbst das Ergebnis über den Atomaren Ausdruck Stack(Num(str)) vom Stack lesen: IfElse(Stack(Num(str)), stmts, stmts). Funktionen sind nur noch über die Labels von Blöcken zu erkennen, die den gleichen Bezeichner haben, wie die ursprüngliche Funktion und es lässt sich nur durch das Nachverfolgen der GoTo(Name('label'))-Knoten nachvollziehen, was ursprünglich zur Funktion gehörte.

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

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

Code 1.9: Pico C-ANF Pass für Codebespiel.

#### 1.3.1.4 RETI-Blocks Pass

#### 1.3.1.4.1 Aufgabe

Die Aufgabe des RETI-Blocks Passes ist es die Anweisungen in den Blöcken, die durch PicoC-Knoten im Abstrakten Syntaxbaum der Sprache  $L_{PicoC\_ANF}$  dargestellt sind durch ihren entsprechenden RETI-Knoten zu ersetzen.

#### 1.3.1.4.2 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 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 1.3.8 aufgelöst wird.

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

Grammatik 1.3.4: Abstrakte Grammatik der Sprache  $L_{RETI\_Blocks}$  in ASF

#### 1.3.1.4.3 Codebeispiel

In Code 1.10 sieht man den Abstrakten Syntaxbaum des RETI-Blocks Passes für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem die Anweisungen, die durch entsprechende PicoC-Knoten im Abstrakten Syntaxbaum der Sprache  $L_{PicoC\_ANF}^{23}$  repräsentiert waren nun durch ihre entsprechennden RETI-Knoten ersetzt werden.

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

<sup>&</sup>lt;sup>23</sup>Beschrieben durch die Grammatik 1.3.3.

```
# // IfElse(Num('1'), [], [])
24
           # Exp(Num('1'))
25
           SUBI SP 1;
26
           LOADI ACC 1;
27
           STOREIN SP ACC 1;
28
           # IfElse(Stack(Num('1')), [], [])
29
           LOADIN SP ACC 1;
30
           ADDI SP 1;
31
           JUMP== GoTo(Name('while_after.1'));
32
           Exp(GoTo(Name('while_branch.4')))
33
         ],
34
       Block
         Name 'while_branch.4',
36
         Γ
37
           # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
38
           # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39
           # Exp(Stackframe(Num('0')))
40
           SUBI SP 1;
41
           LOADIN BAF ACC -2;
           STOREIN SP ACC 1;
42
43
           # Exp(Num('1'))
44
           SUBI SP 1;
45
           LOADI ACC 1;
46
           STOREIN SP ACC 1;
47
           LOADIN SP ACC 2;
48
           LOADIN SP IN2 1;
49
           SUB ACC IN2;
50
           JUMP== 3;
51
           LOADI ACC 0;
52
           JUMP 2;
53
           LOADI ACC 1;
54
           STOREIN SP ACC 2;
55
           ADDI SP 1;
56
           # IfElse(Stack(Num('1')), [], [])
57
           LOADIN SP ACC 1;
58
           ADDI SP 1;
59
           JUMP== GoTo(Name('if_else_after.2'));
60
           Exp(GoTo(Name('if.3')))
61
         ],
62
       Block
63
         Name 'if.3',
64
         Ε
65
           # // Return(Name('res'))
66
           # Exp(Stackframe(Num('1')))
67
           SUBI SP 1;
68
           LOADIN BAF ACC -3;
69
           STOREIN SP ACC 1;
70
           # Return(Stack(Num('1')))
71
           LOADIN SP ACC 1;
72
           ADDI SP 1;
73
           LOADIN BAF PC -1;
74
        ],
75
       Block
76
         Name 'if_else_after.2',
           # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
           # Exp(Stackframe(Num('0')))
```

```
SUBI SP 1;
81
           LOADIN BAF ACC -2;
82
           STOREIN SP ACC 1;
83
           # Exp(Stackframe(Num('1')))
           SUBI SP 1;
85
           LOADIN BAF ACC -3;
86
           STOREIN SP ACC 1;
87
           # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88
           LOADIN SP ACC 2;
89
           LOADIN SP IN2 1;
90
           MULT ACC IN2;
91
           STOREIN SP ACC 2;
92
           ADDI SP 1;
93
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
94
           LOADIN SP ACC 1;
95
           STOREIN BAF ACC -3;
96
           ADDI SP 1;
97
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98
           # Exp(Stackframe(Num('0')))
99
           SUBI SP 1;
00
           LOADIN BAF ACC -2;
101
           STOREIN SP ACC 1;
102
           # Exp(Num('1'))
103
           SUBI SP 1;
104
           LOADI ACC 1;
105
           STOREIN SP ACC 1;
106
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107
           LOADIN SP ACC 2;
108
           LOADIN SP IN2 1;
109
           SUB ACC IN2;
110
           STOREIN SP ACC 2;
111
           ADDI SP 1;
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
113
           LOADIN SP ACC 1;
           STOREIN BAF ACC -2;
114
115
           ADDI SP 1;
116
           # Exp(GoTo(Name('condition_check.5')))
117
           Exp(GoTo(Name('condition_check.5')))
118
         ],
119
       Block
120
         Name 'while_after.1',
121
         Ε
           # Return(Empty())
123
           LOADIN BAF PC -1;
124
         ],
L25
       Block
126
         Name 'main.0',
L27
128
           # StackMalloc(Num('2'))
129
           SUBI SP 2;
130
           # Exp(Num('4'))
131
           SUBI SP 1;
132
           LOADI ACC 4;
133
           STOREIN SP ACC 1;
134
           # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
           MOVE BAF ACC;
           ADDI SP 3;
```

```
MOVE SP BAF;
           SUBI SP 4;
.39
           STOREIN BAF ACC 0;
           LOADI ACC GoTo(Name('addr@next_instr'));
           ADD ACC CS;
           STOREIN BAF ACC -1;
43
           # Exp(GoTo(Name('faculty.6')))
L44
           Exp(GoTo(Name('faculty.6')))
L45
           # RemoveStackframe()
L46
           MOVE BAF IN1;
L47
           LOADIN IN1 BAF 0;
48
           MOVE IN1 SP;
49
           # Exp(ACC)
150
           SUBI SP 1;
151
           STOREIN SP ACC 1;
152
           LOADIN SP ACC 1;
153
           ADDI SP 1;
154
           CALL PRINT ACC;
L55
            # Return(Empty())
156
           LOADIN BAF PC -1;
L57
158
```

Code 1.10: RETI-Blocks Pass für Codebespiel.

## Anmerkung 9

Wenn der Abstrakte Syntaxbaum ausgegeben wird, ist die Darstellung nicht auschließlich in Abstrakter Syntax, da die RETI-Knoten aus bereits im Unterkapitel 1.2.5.6 vermitteltem Grund in Konkreter Syntax ausgeben werden.

#### 1.3.1.5 RETI-Patch Pass

#### 1.3.1.5.1 Aufgabe

Die Aufgabe des RETI-Patch Passes ist das Ausbessern (engl. to patch) des Abstrakten Syntaxbaumes, durch:

- das Einfügen eines start.<nummer>-Blockes, welcher ein GoTo(Name('main')) zur main-Funktion enthält, wenn in manchen Fällen die main-Funktion nicht die erste Funktion ist und daher am Anfang zur main-Funktion gesprungen werden muss.
- das Entfernen von GoTo()'s, deren Sprung nur eine Adresse weiterspringen würde.
- das Voranstellen von RETI-Knoten, die vor jeder Division Instr(Div(), args) prüfen, ob, nicht durch 0 geteilt wird.<sup>24</sup>
- das Überprüfen darauf, ob bestimmte Immediates Im(str) in Befehlen, wie z.B. Jump(rel, Im(str)), Instr(Loadin(), [reg, reg, Im(str)]), Instr(Loadi(), [reg, Im(str)]) usw. kleiner -2<sup>21</sup> oder größer 2<sup>21</sup> 1 sind. Im Fall dessen, dass es so ist, muss der gewünschte Zahlenwert durch Bitshiften und Anwenden von bitweise Oder berechnet werden. Im Fall, dessen, dass der Immediate allerdings kleiner -(2<sup>31</sup>) oder größer 2<sup>31</sup> 1 ist, wird eine Fehlermeldung TooLargeLiteral ausgegeben.

<sup>&</sup>lt;sup>24</sup>Das fällt unter die Themenbereiche des Bachelorprojekts und wird daher nicht genauer erläutert.

#### 1.3.1.5.2 Abstrakte Grammatik

Die Abstrakte Grammatik 1.3.5 der Sprache  $L_{RETI\_Patch}$  ist im Vergleich zur Abstrakten Grammatik 1.3.4 der Sprache  $L_{RETI\_Blocks}$  kaum verändert. Es muss nur ein Knoten Exit() hinzugefügt werden, der im Falle einer Division durch 0 die Ausführung des Programs beendet.

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

Grammatik 1.3.5: Abstrakte Grammatik der Sprache L<sub>RETI Patch</sub> in ASF

#### 1.3.1.5.3 Codebeispiel

In Code 1.11 sieht man den Abstrakten Syntaxbaum des PiocC-Patch Passes für das aus Unterkapitel 1.6 weitergeführte Beispiel. Durch den RETI-Patch Pass wurde hier ein start. <nummer>-Block<sup>25</sup> eingesetzt, da die main-Funktion nicht die erste Funktion ist. Des Weiteren wurden durch diesen Pass einzelne GoTo(Name(str))-Anweisungen entfernt<sup>26</sup>, die nur einen Sprung um eine Position entsprochen hätten.

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

 $<sup>^{25}\</sup>mathrm{Dieser}$ Block wurde im Code 1.8 markiert.

<sup>&</sup>lt;sup>26</sup>Diese entfernten GoTo(Name(str))'s' wurden ebenfalls im Code 1.8 markiert.

```
LOADIN SP ACC 1;
20
           STOREIN BAF ACC -3;
21
           ADDI SP 1;
           # // While(Num('1'), [])
           # Exp(GoTo(Name('condition_check.5')))
24
           # // not included Exp(GoTo(Name('condition_check.5')))
25
         ],
26
       Block
27
         Name 'condition_check.5',
28
29
           # // IfElse(Num('1'), [], [])
30
           # Exp(Num('1'))
           SUBI SP 1;
32
           LOADI ACC 1;
33
           STOREIN SP ACC 1;
34
           # IfElse(Stack(Num('1')), [], [])
35
           LOADIN SP ACC 1;
36
           ADDI SP 1;
37
           JUMP== GoTo(Name('while_after.1'));
38
           # // not included Exp(GoTo(Name('while_branch.4')))
39
         ],
40
       Block
41
         Name 'while_branch.4',
42
43
           # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
44
           # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
45
           # Exp(Stackframe(Num('0')))
46
           SUBI SP 1;
47
           LOADIN BAF ACC -2;
48
           STOREIN SP ACC 1;
           # Exp(Num('1'))
50
           SUBI SP 1;
51
           LOADI ACC 1;
52
           STOREIN SP ACC 1;
53
           LOADIN SP ACC 2;
54
           LOADIN SP IN2 1;
55
           SUB ACC IN2;
56
           JUMP== 3;
57
           LOADI ACC 0;
58
           JUMP 2;
59
           LOADI ACC 1;
60
           STOREIN SP ACC 2;
61
           ADDI SP 1;
62
           # IfElse(Stack(Num('1')), [], [])
63
           LOADIN SP ACC 1;
64
           ADDI SP 1;
65
           JUMP== GoTo(Name('if_else_after.2'));
66
           # // not included Exp(GoTo(Name('if.3')))
67
         ],
68
       Block
69
         Name 'if.3',
70
71
           # // Return(Name('res'))
           # Exp(Stackframe(Num('1')))
           SUBI SP 1;
           LOADIN BAF ACC -3;
           STOREIN SP ACC 1;
```

```
76
           # Return(Stack(Num('1')))
77
           LOADIN SP ACC 1;
78
           ADDI SP 1;
79
           LOADIN BAF PC -1;
80
         ],
81
       Block
82
         Name 'if_else_after.2',
83
84
           # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85
           # Exp(Stackframe(Num('0')))
86
           SUBI SP 1;
87
           LOADIN BAF ACC -2;
88
           STOREIN SP ACC 1;
89
           # Exp(Stackframe(Num('1')))
90
           SUBI SP 1;
91
           LOADIN BAF ACC -3;
92
           STOREIN SP ACC 1;
93
           # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94
           LOADIN SP ACC 2:
95
           LOADIN SP IN2 1;
96
           MULT ACC IN2;
97
           STOREIN SP ACC 2;
98
           ADDI SP 1;
99
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
100
           LOADIN SP ACC 1;
01
           STOREIN BAF ACC -3:
           ADDI SP 1:
102
103
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104
           # Exp(Stackframe(Num('0')))
           SUBI SP 1;
106
           LOADIN BAF ACC -2;
L07
           STOREIN SP ACC 1;
108
           # Exp(Num('1'))
109
           SUBI SP 1;
L10
           LOADI ACC 1;
111
           STOREIN SP ACC 1;
12
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113
           LOADIN SP ACC 2;
114
           LOADIN SP IN2 1;
115
           SUB ACC IN2;
116
           STOREIN SP ACC 2;
17
           ADDI SP 1;
18
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
119
           LOADIN SP ACC 1;
120
           STOREIN BAF ACC -2;
l21
           ADDI SP 1;
122
           # Exp(GoTo(Name('condition_check.5')))
123
           Exp(GoTo(Name('condition_check.5')))
124
         ],
125
       Block
L26
         Name 'while_after.1',
L27
128
           # Return(Empty())
L29
           LOADIN BAF PC -1;
130
         ],
l31
       Block
132
         Name 'main.0',
```

```
Γ
            # StackMalloc(Num('2'))
            SUBI SP 2;
            # Exp(Num('4'))
            SUBI SP 1;
137
138
            LOADI ACC 4;
139
            STOREIN SP ACC 1;
L40
            # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
L41
           MOVE BAF ACC;
142
            ADDI SP 3;
143
           MOVE SP BAF;
44
            SUBI SP 4;
45
            STOREIN BAF ACC 0;
146
           LOADI ACC GoTo(Name('addr@next_instr'));
47
            ADD ACC CS;
148
            STOREIN BAF ACC -1;
149
            # Exp(GoTo(Name('faculty.6')))
150
            Exp(GoTo(Name('faculty.6')))
L51
            # RemoveStackframe()
152
            MOVE BAF IN1;
153
           LOADIN IN1 BAF O;
154
           MOVE IN1 SP;
155
            # Exp(ACC)
156
            SUBI SP 1;
157
            STOREIN SP ACC 1;
158
            LOADIN SP ACC 1;
159
            ADDI SP 1;
160
            CALL PRINT ACC;
161
            # Return(Empty())
62
            LOADIN BAF PC -1;
163
         ]
164
     ]
```

Code 1.11: RETI-Patch Pass für Codebespiel.

#### 1.3.1.6 **RETI Pass**

#### 1.3.1.6.1 Aufgabe

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

#### 1.3.1.6.2 Konkrete und Abstrakte Grammatik

Die Abstrakte Grammatik 1.3.8 der Sprache  $L_{RETI}$  hat im Vergleich zur Abstrakten Grammatik 1.3.5 der Sprache  $L_{RETI\_Patch}$  nur noch auschließlich **RETI-Knoten**. Alle **RETI-Knoten** stehen nun in einem Program(Name(str), instr)-Knoten.

Ausgegeben wird der finale Maschinencode allerdings in Konkreter Syntax, die durch die Konkreten Grammatiken 1.3.6 und 1.3.7 für jeweils die Lexikalische und Syntaktische Analyse beschrieben wird. Der Grund, warum die Konkrete Grammatik der Sprache  $L_{RETI}$  auch nochmal in einen Teil für die Lexikalische und Syntaktische Analyse unterteilt ist, hat den Grund, dass für die Bachelorarbeit zum

Testen des PicoC-Compilers ein RETI-Interpreter implementiert wurde, der den RETI-Code lexen und parsen muss, um ihn später interpretieren zu können.

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

Grammatik 1.3.6: Konkrete Grammatik der Sprache L<sub>RETI</sub> für die Lexikalische Analyse in EBNF

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

Grammatik 1.3.7: Konkrete Grammatik der Sprache L<sub>BETI</sub> für die Syntaktische Analyse in EBNF

```
ACC() \mid IN1()
                                                                                                                                           L_{-}RETI
                                                   IN2()
                                                                  PC()
                                                                                SP()
                                                                                             BAF()
              ::=
reg
                      CS()
                                  DS()
                      Reg(\langle reg \rangle) \mid Num(\langle str \rangle)
arg
rel
                                  NEq() \mid Lt() \mid LtE() \mid
                                                                            Gt() \mid GtE()
                      Always() \mid NOp()
                                                   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()
                                 | Loadin() | Loadi() | Store() | Storein() | Move()
                      Load()
                      Instr(\langle op \rangle, \langle arg \rangle +) \mid Jump(\langle rel \rangle, Num(\langle str \rangle)) \mid Int(Num(\langle str \rangle))
instr
                      RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)
                      SingleLineComment(\langle str \rangle, \langle str \rangle)
                      Instr(Loadi(), [Reg(Acc()), GoTo(Name(\langle str \rangle))]) \mid Jump(Eq(), GoTo(Name(\langle str \rangle)))
                      Program(Name(\langle str \rangle), \langle instr \rangle *)
program
              ::=
                      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 1.3.8: Abstrakte Grammatik der Sprache  $L_{RETI}$  in ASF

#### 1.3.1.6.3 Codebeispiel

Nach dem RETI-Pass ist das Programm komplett in RETI-Knoten übersetzt, die allerdings in ihrer Konkreten Syntax ausgegeben werden, wie in Code 1.12 zu sehen ist. Es gibt keine Blöcke mehr und die RETI-Befehle in diesen Blöcken wurden zusammengesetzt, wie sie in den Blöcken angeordnet waren. Die letzten Nicht-RETI-Befehle oder RETI-Befehle, die nicht auschließlich aus RETI-Ausdrücken bestehen<sup>27</sup>, die sich in den Blöcken befunden haben, wurden durch RETI-Befehle ersetzt.

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

```
# // Exp(GoTo(Name('main.0')))
 2 JUMP 67;
 3 # // Assign(Name('res'), Num('1'))
 4 # Exp(Num('1'))
 5 SUBI SP 1;
 6 LOADI ACC 1;
 7 STOREIN SP ACC 1;
 8 # Assign(Stackframe(Num('1')), Stack(Num('1')))
 9 LOADIN SP ACC 1;
10 STOREIN BAF ACC -3;
11 ADDI SP 1;
12 # // While(Num('1'), [])
13 # Exp(GoTo(Name('condition_check.5')))
14 # // not included Exp(GoTo(Name('condition_check.5')))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
```

<sup>&</sup>lt;sup>27</sup>Wie z.B. LOADI ACC GoTo(Name('addr@next\_instr')), Exp(GoTo(Name('main.0'))) und JUMP== GoTo(Name('if\_else\_after.2')).

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

```
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1')))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5')))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;
00 # StackMalloc(Num('2'))
01 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
05 STOREIN SP ACC 1;
06 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
107 MOVE BAF ACC;
08 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
11 STOREIN BAF ACC 0;
12 LOADI ACC 80;
13 ADD ACC CS;
14 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6')))
116 JUMP -78;
17 # RemoveStackframe()
18 MOVE BAF IN1;
19 LOADIN IN1 BAF 0;
20 MOVE IN1 SP;
21 # Exp(ACC)
22 SUBI SP 1;
23 STOREIN SP ACC 1;
24 LOADIN SP ACC 1;
125 ADDI SP 1;
26 CALL PRINT ACC;
27 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 1.12: RETI Pass für Codebespiel.

# 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).