Albert Ludwigs Universität Freiburg

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Autor: Jürgen Mattheis

Gutachter: Prof. Dr. Scholl

Betreung: M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für

Betriebssysteme

ERKLÄRUNG

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

Danksagungen

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

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

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

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

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

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

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

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

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

²Wofür ich mich auch nochmal Entschuldigen will.

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

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

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

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

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

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

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

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

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

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

⁶Womit nicht alle Studenten so viel Glück haben.

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

Inhaltsverzeichnis

\mathbf{A}	bbild	ungsverzeichnis	Ι		
C	Codeverzeichnis				
Ta	abelle	enverzeichnis	III		
D	efinit	ionsverzeichnis	\mathbf{V}		
\mathbf{G}	ramn	natikverzeichnis	VI		
1	The	oretische Grundlagen	1		
	1.1 1.2 1.3 1.4 1.5	Compiler und Interpreter 1.1.1 T-Diagramme Formale Sprachen 1.2.1 Ableitungen 1.2.2 Präzedenz und Assoziativität Lexikalische Analyse Syntaktische Analyse Code Generierung 1.5.1 Monadische Normalform 1.5.2 A-Normalform	1 3 5 9 12 13 16 25 25 27		
	1.0	1.5.3 Ausgabe des Maschinencodes	31		
	1.6	Fehlermeldungen	31		
2		Lexikalische Analyse 2.1.1 Konkrete Grammatik für die Lexikalische Analyse 2.1.2 Codebeispiel Syntaktische Analyse 2.2.1 Umsetzung von Präzedenz und Assoziativität	33 35 35 37 38 38		
		2.2.2 Konkrete Grammatik für die Syntaktische Analyse 2.2.3 Ableitungsbaum Generierung 2.2.3.1 Codebeispiel 2.2.3.2 Ausgabe des Ableitunsgbaumes	43 45 46 47		
		2.2.4 Ableitungsbaum Vereinfachung	47 49		
		2.2.5.1 PicoC-Knoten	50 52 57 59 61 62 63		
	2.3	Code Generierung 2.3.1 Passes 2.3.1.1 PicoC-Shrink Pass	64 65 65		

	2.3.1.1.2	Codebeispiel
	2.3.1.2 PicoC-I	Blocks Pass
	2.3.1.2.1	Abstrakte Grammatik
	2.3.1.2.2	Codebeispiel
	2.3.1.3 PicoC-A	ANF Pass
	2.3.1.3.1	Abstrakte Grammatik
	2.3.1.3.2	Codebeispiel
	2.3.1.4 RETI-E	Blocks Pass
	2.3.1.4.1	Aufgabe
	2.3.1.4.2	Abstrakte Grammatik
	2.3.1.4.3	Codebeispiel
	2.3.1.5 RETI-F	Patch Pass 8
	2.3.1.5.1	Aufgabe
	2.3.1.5.2	Abstrakte Grammatik
	2.3.1.5.3	Codebeispiel
	2.3.1.6 RETI F	'ass
	2.3.1.6.1	Aufgabe
	2.3.1.6.2	Konkrete und Abstrakte Grammatik
	2.3.1.6.3	Codebeispiel
Literatur		A

Abbildungsverzeichnis

1.1	Horinzontale Übersetzungszwischenschritte zusammenfassen.
1.2	Veranschaulichung von Linksassoziativität und Rechtsassoziativität
1.3	Veranschaulichung von Präzedenz
1.4	Veranschaulichung der Lexikalischen Analyse
1.5	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum. 23
1.6	Veranschaulichung der Syntaktischen Analyse
1.7	Codebeispiel dafür Code in die Monadische Normalform zu bringen
1.8	Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke
1.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen
2.1	Ableitungsbäume zu den beiden Ableitungen
2.2	Ableitungsbaum nach Parsen eines Ausdrucks
2.3	Ableitungsbaum nach Vereinfachung
2.4	Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen
2.5	Generierung eines Abstrakten Syntaxbaumes mit Umdrehen
2.6	Kompiliervorgang Kurzform
2.7	Architektur mit allen Passes ausgeschrieben

Codeverzeichnis

2.1	PicoC-Code des Codebeispiels
2.2	Tokens für das Codebeispiel
2.3	Ableitungsbaum nach Ableitungsbaum Generierung.
2.4	Ableitungsbaum nach Ableitungsbaum Vereinfachung
2.5	Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum
2.6	PicoC Code für Codebespiel
2.7	Abstrakter Syntaxbaum für Codebespiel
	PicoC-Blocks Pass für Codebespiel
2.9	Symboltabelle für Codebespiel
2.10	PicoC-ANF Pass für Codebespiel
2.11	RETI-Blocks Pass für Codebespiel
2.12	RETI-Patch Pass für Codebespiel
2.13	RETI Pass für Codebespiel

Tabellenverzeichnis

1.1	Beispiele für Lexeme und ihre entsprechenden Tokens
2.1	Präzedenzregeln von PicoC
2.2	Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren
2.3	PicoC-Knoten Teil 1
2.4	PicoC-Knoten Teil 2
2.5	PicoC-Knoten Teil 3
2.6	PicoC-Knoten Teil 4
2.7	RETI-Knoten
2.8	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung 6

Definitionsverzeichnis

1.1	Pipe-Filter Architekturpattern
1.2	Interpreter
1.3	Compiler
1.4	Maschinensprache
1.5	Immediate
1.6	Cross-Compiler
1.7	T-Diagram Programm
1.8	T-Diagram Übersetzer (bzw. eng. Translator)
1.9	T-Diagram Interpreter
1.10	Symbol
1.11	Alphabet
1.12	Wort
1.13	Formale Sprache
	Syntax
	Semantik
1.16	Formale Grammatik
1.17	Chromsky Hierarchie
1.18	Reguläre Grammatik
1.19	Kontextfreie Grammatik
1.20	Wortproblem
1.21	1-Schritt-Ableitungsrelation
1.22	Ableitungsrelation
1.23	Links- und Rechtsableitungableitung
1.24	Linksrekursive Grammatiken
1.25	Formaler Ableitungsbaum
	Mehrdeutige Grammatik
	Assoziativität
	Präzedenz
1.29	Lexeme 1
	Token
	Lexer (bzw. Scanner oder auch Tokenizer)
	Literal
1.33	Konkrete Syntax
	Konkrete Grammatik
	Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)
	Parser
	Erkenner (bzw. engl. Recognizer)
1.38	Transformer
1.39	Visitor
1.40	Abstrakte Syntax
	Abstrakte Grammatik
	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST) $\dots \dots 2$
	Pass
	Ausdruck (bzw. engl. Expression)
	Anweisung (bzw. engl. Statement)
	Reiner Ausdruck / Reine Anweisung (bzw. engl. pure expression)
1 47	Unreiner Augdruck / Unreine Anweigung

1.48	Monadische Normalform (bzw. engl. monadic normal form)
1.49	Location
1.50	Atomarer Ausdruck
1.51	Komplexer Ausdruck
1.52	A-Normalform (ANF)
1.53	Fehlermeldung
2.1	Metasyntax
2.2	Metasprache
2.3	Erweiterte Backus-Naur-Form (EBNF)
2.4	Dialekt der Erweiterten Backus-Naur-Form aus Lark
2.5	Abstrakte Syntaxform (ASF)
2.6	Earley Parser
2.7	Entarteter Baum
2.8	Symboltabelle

Grammatikverzeichnis

1.1	Produktionen für einen Ableitungsbaum in EBNF	11
		22
1.3	Produktionen für Abstrakten Syntaxbaum in ASF	23
2.1.1	Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF	37
2.2.1	Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in	
	EBNF, die Operatorpräzidenz nicht beachtet	39
2.2.2	Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syn-	
		40
2.2.3		41
2.2.4	Beispiel für eine unäre linksassoziative Produktion in EBNF	41
2.2.5	Beispiel für eine binäre linksassoziative Produktion in EBNF	42
2.2.6	Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF	42
2.2.7	Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet	43
2.2.8	Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1	44
2.2.9	Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	45
2.2.1	OAbstrakte Grammatik der Sprache L_{PiocC} in ASF $\dots \dots \dots \dots \dots \dots \dots \dots \dots \dots$	62
2.3.1	Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF	67
2.3.2	Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF	70
		74
2.3.4	Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF	78
2.3.5	Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF \ldots	82
2.3.6	Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF	86
2.3.7	Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF \dots	86
2.3.8	Abstrakte Grammatik der Sprache L_{RETI} in ASF \ldots	86

1 Theoretische Grundlagen

In diesem Kapitel wird auf die Theoretischen Grundlagen eingegangen, die zum Verständnis der Implementierung in Kapitel 2 notwendig sind. Zuerst wird in Unterkapitel 1.1 genauer darauf eingegangen was ein Compiler und Interpreter eigentlich sind und damit in Verbindung stehende Begriffe erklärt. Danach wird in Unterkapitel 1.2 eine kleine Einführung zu einem der Grundpfeiler des Compilerbau, den Formalen Sprachen gegeben. Danach werden die einzelnen Filter des üblicherweise bei der Implementierung von Compilern genutzten Pipe-Filter-Architekturpatterns (Definition 1.1) nacheinander erklärt. Die Filter beinhalten die Lexikalische Analyse 1.3, Syntaktische Analyse 1.4 und Code Generierung 1.5. Zum Schluss wird in Unterkapitel 1.6 darauf eingegangen in welchen Situationen Fehlermeldungen auszugeben sind.

Definition 1.1: Pipe-Filter Architekturpattern

7

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

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

Eine Pipe stellt ein Bindeglied zwischen zwei Filtern dar. ab



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

1.1 Compiler und Interpreter

Unterkapitelie wohl wichtigsten zu klärenden Begriffe, sind die eines Compilers (Definition 1.3) und eines Interpreters (Definition 1.2), da das Schreiben eines Compilers von der PicoC-Sprache L_{PicoC} in die RETI-Sprache L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines Interpreters genutzt wird, um zu definieren was ein Compiler ist. Des Weiteren wurde zur Qualitätsicherung ein RETI-Interpreter implementiert, um mithilfe des GCC¹ und von Tests die Beziehungen in 1.3.1 zu belegen (siehe Unterkapitel ??).

^bWestphal, "Softwaretechnik".

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

Definition 1.2: Interpreter

Z

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

Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Teilbäumen** des **Abstrakten Syntaxbaumes** (wird später eingeführt unter Definition 1.42) und führt je nach Komposition der **Knoten** des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.^b

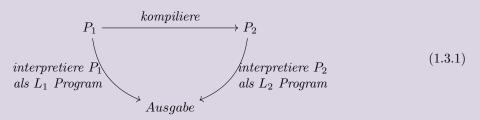
^aMaschinensprache kann genauso interpretiert werden, wie auch eine Programmiersprache.

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

Definition 1.3: Compiler

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

Wobei Kompilieren meint, dass ein beliebiges Program P_1 in der Sprache L_1 so in die Sprache L_2 zu einem Programm P_2 übersetzt wird, dass bei beiden Programmen, wenn sie von Interpretern ihrer jeweiligen Sprachen L_1 und L_2 interpretiert werden, sie die gleiche Ausgabe haben, wie es in Diagramm 1.3.1 dargestellt ist. Also beide Programme P_1 und P_2 die gleiche Semantik (Definition 1.15) haben und sich nur syntaktisch (Definition 1.14) durch die Sprachen L_1 und L_2 , in denen sie geschrieben stehen unterscheiden.^a



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

Üblicherweise kompiliert ein Compiler ein Program, das in einer Programmiersprache geschrieben ist zu Maschinencode, der in Maschinensprache (Definition 1.4) geschrieben ist, aber es gibt z.B. auch Transpiler (Definition ??) oder Cross-Compiler (Definition 1.6). Des Weiteren sind Maschinensprache und Assemblersprache (Definition ??) voneinander zu unterscheiden.

Definition 1.4: Maschinensprache

Z

Programmiersprache, deren mögliche Programme die hardwarenaheste Repräsentation eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten Aufgabe, die die CPU im vereinfachten Fall in einem Zyklus der Fetch- und Execute-Phase, genauergesagt in der Execute-Phase übernehmen kann oder allgemein in einer geringen konstanten Anzahl von Fetch- und Execute Phasen im Komplexeren Fall. Die Maschinenbefehle sind meist so entworfen, dass sie sich innerhalb bestimmter Wortbreiten, die Zweierpotenzen sind kodieren lassen. Im einfachsten Fall innerhalb einer Speicherzelle des Hauptspeichers.

^aViele Prozessorarchitekturen erlauben es allerdings auch z.B. zwei Maschinenbefehle in eine Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen Immediates (Definition 1.5)

 $[^]b$ Scholl, "Betriebssysteme".

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

Der PicoC-Compiler, der den Zweck erfüllt für Studenten ein Anschauungs- und Lernwerkzeug zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in menschenlesbarer Form mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 1.5) enthält. Für den RETI-Interpreter ist es ebenfalls nicht notwendig, dass der Maschinencode, den der PicoC-Compiler generiert, in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU simulieren soll und nicht deren mögliche interne Umsetzung².

Definition 1.5: Immediate

Z

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

Definition 1.6: Cross-Compiler



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

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

1.1.1 T-Diagramme

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

Die Notation setzt sich dabei aus den Blöcken für ein Program (Definition 1.7), einen Übersetzer (Definition 1.8), einen Interpreter (Definition 1.9) und eine Maschine (Definition ??) zusammen.

^aLjohhuh, What is an immediate value?

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

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

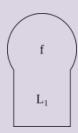
²Eine RETI-CPU zu bauen, die menschenlesbaren Maschinencode in z.B. UTF-8 Kodierung ausführen kann, wäre dagegen unnötig kompliziert und aufwändig, da Hardware binär arbeitet und man dieser daher lieber direkt die binär kodierten Maschinenbefehle übergibt, anstatt z.B. eine unnötig platzverbrauchenden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

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

Definition 1.7: T-Diagram Programm

/

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



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

Anmerkung Q

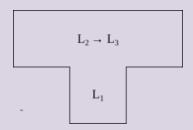
Es ist bei T-Diagrammen nicht notwendig beim entsprechenden Platzhalter, in den man die genutzte Sprache schreibt, den Namen der Sprache an ein L dranzuhängen, weil hier immer eine Sprache steht. Es würde in Definition 1.7 also reichen einfach eine 1 hinzuschreiben.

Definition 1.8: T-Diagram Übersetzer (bzw. eng. Translator)

I

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

Für den Übersetzer gelten genauso, wie für einen Compiler die Beziehungen in 1.3.1.^a



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

Anmerkung 9

Zwischen den Begriffen Übersetzung und Kompilierung gibt es einen Unterschied.

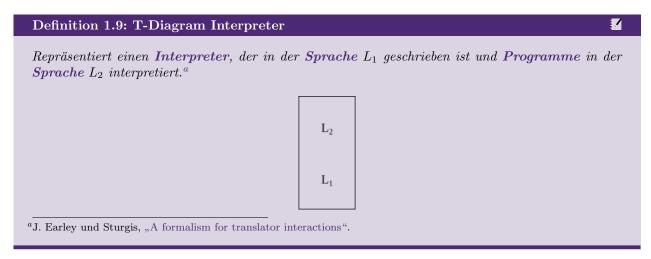
Übersetzung ist der allgemeinere Begriff und verlangt nur, dass Eingabe und Ausgabe des Übersetzers die gleiche Bedeutung^a haben müssen^b.

Kompilierung beinhaltet dagegen meist auch das Lexen und Parsen oder irgendeine Form von Umwandlung eines Programmes von der Textrepräsentation in eine compilerinterne Datenstruktur und erst dann ein oder mehrere Übersetzungsschritte.

Kompilieren ist also auch Übersetzen, aber Übersetzen ist nicht immer auch Kompilieren.

^aAuch Semantik (Definition 1.15) genannt.

^bUnd ist auch zwischen Passes (Definition 1.43) möglich.



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

Die horinzontale Adjazenz lässt sich, wie man in Abbildung 1.1 erkennen kann zusammenfassen.

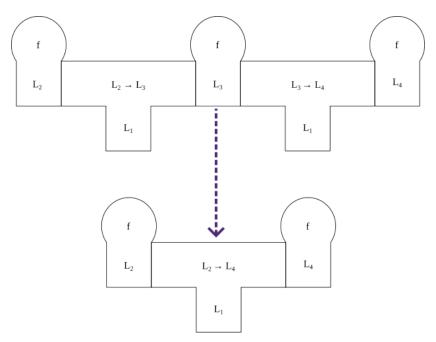


Abbildung 1.1: Horinzontale Übersetzungszwischenschritte zusammenfassen.

1.2 Formale Sprachen

Das Kompilieren eines Programmes hat viel mit dem Thema Formaler Sprachen (Definition 1.13) zu tun, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der Sprache L_1 in eine Sprache L_2 ist. Aus diesem Grund ist es wichtig die Grundlagen Formaler Sprachen, was die Begriffe Symbol (Definition 1.10), Alphabet (Definition 1.11), Wort (Definition 1.12) beinhaltet vorher eingeführt zu haben.

Definition 1.10: Symbol

Z

"Ein Symbol ist ein **Element** eines **Alphabets** Σ . "a

^aNebel, "Theoretische Informatik".

Definition 1.11: Alphabet

Z

"Ein Alphabet ist eine endliche, nicht-leere Menge aus Symbolen (Definition 1.10)."a

^aNebel, "Theoretische Informatik".

Definition 1.12: Wort

Z

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

Es gibt es die Konkatenation $w_1w_2 = a_1 \dots a_nb_1 \dots b_n$ von Wörtern $w_1 = a_1 \dots a_n$ und $w_2 = b_1 \dots b_n$ und die Länge eines Wortes |w|.

Ein wichtiges Wort ist das leere Wort ε für das gilt: $|\varepsilon|=0$ und $\forall w \in \Sigma^*$: $\varepsilon w=w\varepsilon=w$. Es handelt sich bei ε also um das Neutrale Element bei der Konkatenation von Wörtern.

Bei Σ^* handelt es sich um Kleenesche Hülle eines Alphabets Σ , es ist die Sprache aller Wörter, welche durch beliebige Konkatenation von Symbolen aus dem Alphabet Σ gebildet werden können, wobei $\varepsilon \in \Sigma^*$. Dies ist die größte Sprache über Σ und jede Sprache über Σ ist eine Teilmenge davon. "a

^aNebel, "Theoretische Informatik".

Definition 1.13: Formale Sprache



"Eine Formale Sprache ist eine Menge von Wörtern (Definition 1.12) über dem Alphabet Σ (Definition 1.11). "a

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

^aNebel, "Theoretische Informatik".

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die Semantik (Definition 1.15) gleich bleiben. Beide Sprachen L_1 und L_2 haben eine Grammatik (Definition 1.16), welche diese beschreibt und können verschiedene Syntaxen (Definition 1.14) haben.

Definition 1.14: Syntax



Bezeichnet alles was mit dem Aufbau von Wörtern einer Formalen Sprache zu tun hat. Eine Formale Grammatik, aber auch in Natürlicher Sprache ausgedrückte Regeln können die Syntax einer Sprache beschreiben. Es kann auch mehrere verschiedene Syntaxen für die gleiche Sprache geben^a.^b

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

^bThiemann, "Einführung in die Programmierung".

Definition 1.15: Semantik

Z

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

^aThiemann, "Einführung in die Programmierung".

Definition 1.16: Formale Grammatik

Z

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

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

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

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

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

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

Die gerade definierten Formale Sprachen lassen sich des Weiteren in Klassen der Chromsky Hierarchie (Definition 1.17) einteilen.

Definition 1.17: Chromsky Hierarchie

7

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

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

Zwischen den Sprachmengen benachbarter Klassen in Abbildung 1.17.1 besteht eine echte Teilmengenbeziehung: $L_3 \subset L_2 \subset L_1 \subset L_0$. Jede Reguläre Sprache ist auch eine Kontextfreie Sprache, aber nicht jede Kontextfreie Sprache ist auch eine Reguläre Sprache.^a

^aWeil mit ihnen terminiert wird.

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

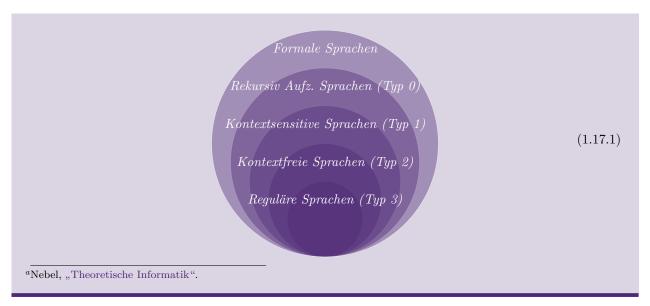
^cw muss mindestens ein Nicht-Terminalsymbol enthalten.

 $^{{}^}d \mathrm{Bzw}. \ w,v \in C^*$ und $w \not \in \Sigma^*$

^eDie Grammatik des PicoC-Compilers lässt sich in Produktionen für die Lexikalische Analyse und Syntaktische Analyse unterteilen. Die gesamte Grammatik steht allerdings vereinigt in einer Datei.

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

^gNebel, "Theoretische Informatik".



Für diese Bachelorarbeit sind allerdings nur die Spracheklassen der Chromsky-Hierarchie relevant, die von Regulären (Definition 1.18) und Kontextfreien Grammatiken (Definition 1.19) beschrieben werden.

Definition 1.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon$$
 (1.18.1)

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

Definition 1.19: Kontextfreie Grammatik

1

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

$$A \to v \tag{1.19.1}$$

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

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des Wortproblems (Definition 1.20). In einem Compiler oder Interpreter ist das Wortproblem üblicherweise immer entscheidbar. Wenn das Programm ein Wort der Sprache ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es kein Wort der Sprache, die der Compiler kompiliert, wird eine Fehlermeldung ausgegeben.

 $[^]a$ Diese Definition einer Regulären Grammatik ist rechtsregulär, es ist auch möglich diese Definition linksregulär zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.

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

^cNebel, "Theoretische Informatik".

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

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

Definition 1.20: Wortproblem

Z

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

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

$$\mathbb{1}_L: \Sigma^* \to \{0, 1\}: w \mapsto \begin{cases} 1 & falls \ w \in L \\ 0 & sonst \end{cases}$$
 (1.20.1)

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

1.2.1 Ableitungen

Um sicher zu wissen, ob ein Compiler ein **Programm**⁴ kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprach**e des Compilers abzuleiten. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 1.21) und der normalen **Ableitungsrelation** (Definition 1.22) unterschieden.

Definition 1.21: 1-Schritt-Ableitungsrelation



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

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

^aNebel, "Theoretische Informatik".

Definition 1.22: Ableitungsrelation



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

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

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**⁵ kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 1.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 1.4 relevant.

Definition 1.23: Links- und Rechtsableitungableitung



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

 $[^]b$ Nebel, "Theoretische Informatik".

^cAuch Charakteristische Funktion genannt.

^aNebel, "Theoretische Informatik".

⁴Bzw. Wort.

⁵Bzw. Wort.

die linke Seite einer Produktion.

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

^aNebel, "Theoretische Informatik".

Manche der **Ansätz**e für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 1.24) ist⁶.

Definition 1.24: Linksrekursive Grammatiken

I

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

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

$$A \Rightarrow^* Aa$$
,

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

anoauthor'parsing'nodate.

Um herauszufinden, ob eine Grammatik mehrdeutig (Definition 1.26) ist, werden Ableitungen als Formale Ableitungsbäume (Definition 1.25) dargestellt. Formale Ableitungsbäume werden im Unterkapitel 1.4 nochmal relevant, da in der Syntaktischen Analyse Ableitungsbäume (Definition 1.35) als eine compilerinterne Datenstruktur umgesetzt werden.

Definition 1.25: Formaler Ableitungsbaum



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

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

Den Knoten dieses Baumes sind Grammatiksymbole $C = N \cup \Sigma \cup \varepsilon$ (Definition 1.16) zugeordnet. Die Inneren Knoten des Baumes sind Nicht-Terminalsymbole N und die Blätter sind entweder Terminalsymbole Σ oder das leere Wort ε .

^aZ.B. **Programmcode**.

^bNebel, "Theoretische Informatik".

In Abbildung 1.25.2 ist ein Beispiel für einen Formalen Ableitungsbaum zu sehen, der sich aus der Ableitung 1.25.1 nach den im Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit (Definition 2.4) angegebenen Produktionen 1.1 einer Grammatik $G = \langle N, \Sigma, P, add \rangle$ ergibt.

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

DIG_NO_0	::=	"1" "2" "3" "4" "5" "6"	$L_{-}Lex$
		"7" "8" "9"	
DIG_WITH_0	::=	"0" DIG_NO_0	
NUM	::=	"0" DIG_NO_0 DIG_WITH_0*	
ADD_OP	::=	"+"	
MUL_OP	::=	"*"	
\overline{mul}	::=	$mul\ MUL_OP\ NUM\ \ NUM$	L_Parse
add	::=	$add\ ADD_OP\ mul\ \mid\ mul$	

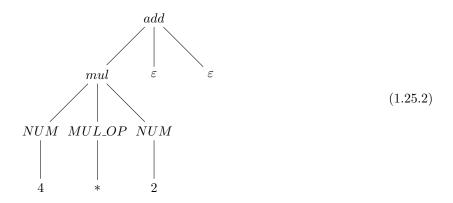
Grammatik 1.1: Produktionen für einen Ableitungsbaum in EBNF

Anmerkung Q

Werden die Produktionen einer Grammatik in z.B. EBNF angegeben, wie in Grammatik 2.1.1, wird die Angabe dieser Produktionen auch oft als Grammatik bezeichnet, obwohl Grammatiken eigentlich durch ein Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt sind.

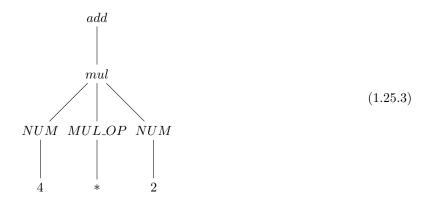
$$add \Rightarrow mul \Rightarrow mul \ MUL_OP \ NUM \Rightarrow NUM \ MUL_OP \ NUM \Rightarrow "4" "*" "2"$$
 (1.25.1)

Bei Ableitungsbäumen gibt es keine einheutliche Regelung, wie damit umgegangen wird, wenn die Alternativen einer Produktion unterschiedliche viele Nicht-Terminalsymbole enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 1.25.2 von der Maximalzahl auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der Differenz zur Maximalzahl viele Blätter mit dem leeren Wort ε hinzuzufügen.



Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.25.3 nur die vorhandenen Nicht-Terminalsymbole als Kinder hinzuzufügen⁷.

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



Für einen Compiler ist es notwendig, dass die Konkrete Grammatik keine Mehrdeutige Grammatik (Definition 1.26) ist, denn sonst können unter anderem die Präzedenzregeln der verschiedenen Operatoren nicht gewährleistet werden, wie später in Unterkapitel 2.2.1 an einem Beispiel demonstriert wird.

Definition 1.26: Mehrdeutige Grammatik

Z

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

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

^bNebel, "Theoretische Informatik".

1.2.2 Präzedenz und Assoziativität

Will man die Operatoren aus einer Programmiersprache in einer Konkreten Grammatik ausdrücken, die nicht mehrdeutig ist, so lässt sich das nach einem klaren Schema machen, wenn die Assoziativität (Definiton 1.27) und Präzedenz (Definition 1.28) dieser Operatoren festgelegt ist. Dieses Schema wird in Unterkapitel 2.2.1 genauer erklärt.

Definition 1.27: Assoziativität



"Bestimmt, welcher Operator aus einer Reihe gleicher Operatoren zuerst ausgewertet wird."

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

^anoauthor'parsing'nodate.

Bei Assoziativität ist z.B. der Multitplikationsoperator * ein Beispiel für einen linksassoziativen Operator und ein Zuweisungsoperator = ein Beispiel für einen rechtsassoziativen Operator. Dies ist in Abbildung 1.2 mithilfe von Klammern () veranschaulicht.

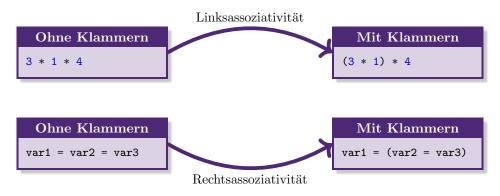
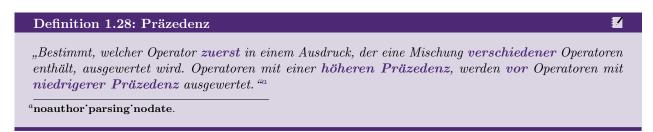


Abbildung 1.2: Veranschaulichung von Linksassoziativität und Rechtsassoziativität.



Bei Präzedenz ist die Mischung der Operatoren für Subraktion '-' und für Multiplikation * ein Beispiel für den Einfluss von Präzedenz. Dies ist in Abbildung 1.3 mithilfe der Klammern () veranschaulicht. Im Beispiel in Abbildung 1.3 ist bei den beiden Subtraktionsoperatoren '-' nacheinander und dem darauffolgenden Multitplikationsoperator * sowohl Assoziativität als auch Präzedenz im Spiel.



Abbildung 1.3: Veranschaulichung von Präzedenz.

1.3 Lexikalische Analyse

Die Lexikalische Analyse bildet üblicherweise den ersten Filter innerhalb des Pipe-Filter Architekturpatterns (Definition 1.1) bei der Implementierung von Compilern. Die Aufgabe der Lexikalischen Analyse ist vereinfacht gesagt in einem Eingabewort⁸ endliche Folgen von Symbolen⁹ zu finden, die durch eine reguläre Grammatik erkannt werden. Diese Folgen endlicher Symoble werden auch Lexeme (Definition 1.29) genannt.



Diese Lexeme werden vom Lexer (Definition 1.31) im Eingabewort identifziert und Tokens (Definition 1.30) zugeordnet. Das jeweils nächste Lexeme fängt dabei genau nach dem letzten Symbol des Lexemes an,

 $^{^8}$ Z.B. dem Inhalt einer Datei, welche in **UTF-8** kodiert ist.

⁹Also Teilwörter des Eingabeworts.

das zuletzt vom Lexer erkannt wurde. Die Tokens sind es, die letztendlich an die Syntaktische Analyse weitergegeben werden.

Definition 1.30: Token

Z

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

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

Definition 1.31: Lexer (bzw. Scanner oder auch Tokenizer)

1

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

^aThiemann, "Compilerbau".

Ein Lexer ist im Allgemeinen eine partielle Funktion, da es Zeichenfolgen geben kann, die sich unter der Grammatik G_{Lex} nicht ableiten lassen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine Fehlermeldung (Definition 1.53) ausgegeben.

Anmerkung 9

Um Verwirrung verzubeugen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

In der Syntaktischen Analyse sind die Tokentypen die Symbole.

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

^aDas ist der Grund, warum die Tabelle, in der Informationen zu Bezeichnern gespeichert werden, in Kapitel 2 Symboltabelle genannt wird.

Eine weitere Aufgabe der Lekikalischen Analyse ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen $_{-}$, Newline \n^{10} und Tabs \t aus dem Eingabewort herauszufiltern. Das geschieht mittels des Lexers, der allen für die Syntaktische Analyse unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (T \times W)^*$ ist immer der Fall bei der Kleeneschen Hülle Σ^* , wobei $\Sigma = T \times W$. Nur das, was für die Syntaktische Analyse wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die Lexeme an die Syntaktische Analyse weitergegeben werden und der Grund für die Aufteilung des Tokens in Tokentyp T und Tokenwert W, ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen und auch Zahlen beliebige Zeichenfolgen sein können. Später in der Syntaktischen Analyse in Unterkapitel 1.4 wird sich nur dafür interessiert, ob an einer bestimmten Stelle ein bestimmter Tokentyp T, z.B. eine Zahl NUM steht und der Tokenwert W ist erst wieder in der

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

Code Generierung in Unterkapitel 1.5 relevant.

Wie in Tabelle 1.1 zu sehen, gibt es für Bezeichner, wie my_fun, my_var oder my_const und verschiedenen Zahlen, wie 42, 314 oder 12 passende Tokentypen NAME¹¹ und NUM^{12 13 14}. Für Lexeme, wie if oder } sind die Tokentypen dagegen genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich IF und RBRACE.

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

Tabelle 1.1: Beispiele für Lexeme und ihre entsprechenden Tokens.

Ein Lexeme ist nicht immer das gleiche wie der Tokenwert, denn wie in Tabelle 1.1 zu sehen ist, kann z.B. im Fall von L_{PicoC} der Wert 99 durch zwei verschiedene Literale (Definition 1.32) dargestellt werden, einmal als ASCII-Zeichen 'c', das dann als Tokenwert den entsprechenden Wert aus der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99¹⁵. Der Tokenwert ist der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Anmerkung 9

Die Konkrete Grammatik G_{Lex} , die zur Beschreibung der Tokens T der Sprache L_{Lex} verwendet wird ist üblicherweise regulär, da ein typischer Lexer immer nur ein Symbol vorausschaut^a, sich nichts merkt, also unabhängig davon, was für Symbole und wie oft bestimmte Symbole davor aufgetaucht sind funktioniert. Auch für den PicoC-Compiler lässt sich aus der im Dialekt der Backus-Naur-Form des Lark Parsing Toolkit (Definition 2.4) spezifizierten Grammatik 2.1.1 schlussfolgern, dass die Sprache des PicoC-Compilers für die Lexikalische Analyse L_{PicoC_Lex} regulär ist, da alle ihre Produktionen die Definition 1.18 erfüllen.

Produktionen mit Alternative, wie z.B. $DIG_WITH_0 := "0" \mid DIG_NO_0$ sind unproblematisch, denn sie können immer auch als $\{DIG_WITH_0 := "0", DIG_WITH_0 := DIG_NO_0\}$ ausgedrückt werden und z.B. DIG_WITH_0* , (LETTER | $DIG_WITH_0 \mid "_")+$ und " $_"."\sim"$ in Grammatik 2.1.1 können alle zu Alternativen umgeschrieben werden, womit diese Alternativen wie gerade gezeigt umgeformt werden können, um ebenfalls regulär zu sein. Somit existiert mit der Grammatik 2.1.1 eine reguläre Grammatik, welche die Sprache L_{PicoC_Lex} beschreibt und damit ist die Sprache L_{PicoC_Lex} nach der Chromsky Hierarchie (Definition 1.17) regulär.

^aMan nennt das auch einem Lookahead von 1.

Definition 1.32: Literal

Z

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

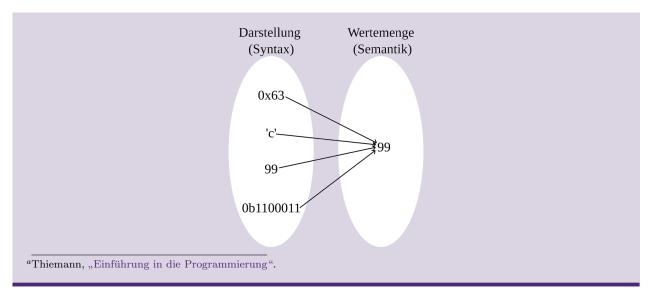
¹¹Für z.B. my_fun, my_var und my_const.

 $^{^{12}{\}rm F\ddot{u}r}$ z.B. 42, 314 und 12.

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

¹⁴Bzw. wenn man sich nicht Kurzformen sucht IDENTIFIER und NUMBER.

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



Um eine Gesamtübersicht über die Lexikalische Analyse zu geben, ist in Abbildung 1.4 die Lexikalische Analyse an einem Beispiel veranschaulicht.

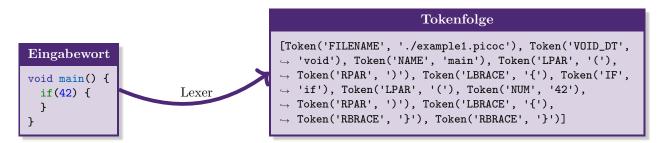


Abbildung 1.4: Veranschaulichung der Lexikalischen Analyse.

Anmerkung Q

Das Symbol \hookrightarrow zeigt im Code der Tokens in Abbildung 1.4 und in den folgenden Codes einen Zeilenumbruch an, wenn eine Zeile zu lang ist.

1.4 Syntaktische Analyse

In der Syntaktischen Analyse ist für einige Sprachen eine Kontextfreie Grammatik G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für Funktionsaufrufe fun(arg) und Codeblöcke if(1){} syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern '(' bzw. öffnende geschweifte Klammern '{'} es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden. Dies lässt sich nicht mehr mit einer Regulären Grammatik (Definition 1.18) beschreiben, sondern es braucht eine Kontextfreie Grammatik (Definition 1.19) hierfür, die es erlaubt zwischen zwei Terminalsymbolen ein Nicht-Terminalsymbol abzuleiten.

Für den PicoC-Compiler lässt sich aus der Grammatik 2.2.8 schlussfolgern, dass die Sprache des PicoC-Compilers für die Syntaktische Analyse L_{PicoC_Parse} kontextfrei, aber nicht mehr regulär ist, da alle

ihre Produktionen die Definition für Kontextfreie Grammatiken 1.19 erfüllen, aber nicht die Definition für Reguläre Grammatiken 1.18.

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

Somit existiert mit der Grammatik 2.2.8 eine Kontextfreie Grammatik und nicht Reguläre Grammatik, welche die Sprache L_{PicoC_Parse} beschreibt und damit ist die Sprache L_{PicoC_Parse} nach der Chromsky Hierarchie (Definition 1.17) kontextfrei, aber nicht regulär.

Die Syntax, in welcher ein Programm aufgeschrieben ist, wird auch als Konkrete Syntax (Definition 1.33) bezeichnet. In einem Zwischenschritt, dem Parsen wird aus diesem Programm mithilfe eines Parsers (Definition 1.36) ein Ableitungsbaum (Definition 1.35) generiert, der als Zwischenstufe hin zum einem Abstrakten Syntaxbaum (Definition 1.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des Ableitungsbaumes und dann erst des Abstrakten Syntaxbaumes.

Definition 1.33: Konkrete Syntax

I

Steht für alles, was mit dem Aufbau von nach einer Konkreten Grammatik (Definition 1.34) abgeleiteten Wörtern^a zu tun hat.

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

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

Um einen kurzen Begriff für die Grammatik zu haben, welche die Konkrete Syntax einer Sprache beschreibt, wird diese im Folgenden als Konkrete Grammatik (Definition 1.34) bezeichnet.

Definition 1.34: Konkrete Grammatik



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

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

 $[^]a$ Bzw. **Programmen**.

 $[^]b$ Vor allem bei **Programmiersprachen**.

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

 $[^]d$ Vereinigungsgrammatik wie in Definition 1.16 erklärt.

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

^aVereinigungsgrammatik wie in Definition 1.16 erklärt.

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

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

Definition 1.35: Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)

Compilerinterne Datenstruktur für den Formalen Ableitungsbaum (Definition 1.25) eines in Konkreter Syntax geschriebenen Programmes.

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

^aVereinigungsgrammatik wie in Definition 1.16 erklärt.

^bNicht mehr weiter teilbar.

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

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

Definition 1.36: Parser



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

aZ.B. wiederum ein **Programm**.

^bEs gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von Konkreter Syntax in Abstrakte Syntax übersetzt. Im Folgenden wird allerdings die Definition 1.36 verwendet.

^cJSON parser - Tutorial — Lark documentation.

Anmerkung Q

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

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

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

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

 a Das würde z.B. der Rolle eines Semikolon ; in der Sprache L_{PicoC} entsprechen.

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

Die vom Lexer im Eingabewort identifizierten Tokens werden in der Syntaktischen Analyse vom Parser als Wegweiser verwendet, da je nachdem, in welcher Reihenfolge die Tokens auftauchen, dies einer anderen Ableitung in der Grammatik G_{Parse} entspricht. Dabei wird in der Grammatik L_{Parse} nach dem Tokentypen unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine Zahl steht und nicht, welchen konkreten Wert diese Zahl hat. Der Tokenwert ist erst später in der Code Generierung in 1.5 wieder relevant.

Ein Parser ist genauergesagt ein erweiterter Erkenner (Definition 1.37), denn ein Parser löst das Wortproblem (Definition 1.20) für die Sprache, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Erkennungsalgorithmus¹⁷ gesichert wurden den Ableitungsbaum.

Definition 1.37: Erkenner (bzw. engl. Recognizer)

Entspricht einem Kellerautomaten^a, in dem Wörter bestimmter Kontextfreier Sprachen erkannt werden. Der Erkenner ist ein Algorithmus, der erkennt, ob ein Eingabewort sich mit den Produktionen der Konkreten Grammatik einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der Konkreten Grammatik beschrieben wird oder nicht. Das vom Erkenner gelöste Problem ist auch als Wortproblem (Definition 1.20) bekannt.^b

 a Automat mit dem Kontextfreie Grammatiken erkannt werden.

^bThiemann, "Compilerbau".

Anmerkung Q

Für das Parsen gibt es grundsätzlich drei verschiedene Ansätze:

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

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

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

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

Mit dieser Methode ist das Parsen Linksrekursiver Grammatiken (Definition 1.24) allerdings nicht möglich, ohne die Konkrete Grammatik vorher umgeformt zu haben und jegliche Linksrekursion aus der Konkreten Grammatik entfernt zu haben, da diese zu Unendlicher Rekursion führt.

¹⁷Bzw. engl. recognition algorithm.

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

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

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

Der Abstrakte Syntaxbaum wird mithilfe von Transformern (Definition 1.38) und Visitors (Definition 1.39) generiert und ist das Endprodukt der Syntaktischen Analyse, welches an die Code Generierung weitergegeben wird. Wenn man die gesamte Syntaktische Analyse betrachtet, so übersetzt diese ein Programm von der Konkreten Syntax in die Abstrakte Syntax (Definition 1.40).

Definition 1.38: Transformer



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

Definition 1.39: Visitor



Ein Programm, das von unten-nach-oben^a, nach dem Prinzip der Breitensuche alle Knoten des Ableitungsbaumes besucht und beim Antreffen eines bestimmten Knoten des Ableitungsbaumes, diesen in-place mit anderen Knoten tauscht oder manipuliert, um den Ableitungbaum für die weitere

^a What is Top-Down Parsing?

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

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

^dWhat is Bottom-up Parsing?

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

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

^bTransformers & Visitors — Lark documentation.

Verarbeitung durch z.B. einen Transformer zu vereinfachen. bc

Definition 1.40: Abstrakte Syntax

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

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

Um einen kurzen Begriff für die Grammatik, welche die Abstrakte Syntax einer Sprache beschreibt zu haben, wird diese im Folgenden als Abstrakte Grammatik (Definition 1.41) bezeichnet.

Definition 1.41: Abstrakte Grammatik



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

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

Definition 1.42: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

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

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

Anmerkung 9

In dieser Bachelorarbeit wird häufig von der "Abstrakten Syntax", der "Abstrakten Grammatik" bzw. dem "Abstrakten Syntaxbaum" einer "Sprache" L gesprochen. Gemeint ist hier mit der Sprache L nicht die Sprache, welche durch die Abstrakte Grammatik, nach welcher der Abstrakte Syntaxbaum abgeleitet ist beschrieben wird. Es ist damit immer die Sprache gemeint, die kompiliert werden soll" und zu deren Zweck der Abstrakte Syntaxbaum überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die Abstrakte Grammatik beschrieben wird, interessiert man sich nie wirklich explizit. Diese Konvention wurde aus dem Buch G. Siek, Course Webpage for Compilers

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

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

^c Transformers & Visitors — Lark documentation.

 $[^]a$ Z.B. **Programmcode**.

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

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

(P423, P523, E313, and E513) übernommen.

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

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

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

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

Auf der linken Seite in Abbildung ?? wird das Beispiel 1.25.2 aus Unterkapitel 1.2.1 fortgeführt. Dieses Beispiel stellt den Arithmetischen Ausdruck 4 * 2 in Bezug auf die Konkrete Grammatik 1.2^{19} , welche die höhere Präzedenz der Multipikation * berücksichtigt in einem Ableitungsbaum dar. Allerdings handelt es sich bei diesem Ableitungsbaum nicht um einen Formalen Ableitungsbaum, sondern um eine compilerinterne Datenstruktur für einen solchen. Dementsprechend sind die Blätter nun Tokens, die mithilfe der Grammatik L_{Lex} generiert wurden, womit die Darstellung von Ableitungen sich auf die Grammatik L_{Parse} beschränkt.

Auf der rechten Seite in Abbildung ?? wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum abstrahiert, der nach der Abstrakten Grammatik 1.3 konstruiert ist. Die Abstrakte Grammatik ist hierbei in Abstrakter Syntaxform (Definition 2.5) angegeben. In der Abstrakten Grammatik 1.3 sind jegliche Produktionen wegabstrahiert, die in der Konkreten Grammatik 1.2 so umgesetzt sind, damit diese Präzidenz beachtet und nicht mehrdeutig ist. Aus diesem Grund gibt es nur noch einen allgemeinen Knoten für binäre Operationen $BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle)$.

DIG_NO_0	::=	"1" "2" "3" "4" "5" "6" <i>L_Lex</i>
		"7" "8" "9"
DIG_WITH_0	::=	"0" DIG_NO_0
NUM	::=	"0" DIG_NO_0 DIG_WITH_0*
ADD_OP	::=	"+"
MUL_OP	::=	"*"
\overline{mul}	::=	mul MUL_OP NUM NUM L_Parse
add	::=	add ADD_OP mul mul

Grammatik 1.2: Produktionen für Ableitungsbaum in EBNF

¹⁸Den verschiedenen Passes.

¹⁹Die Konkrette Grammatik ist hierbei im Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit (Definition 2.4) angegeben.

```
\begin{array}{cccc} bin\_op & ::= & Add() & | & Mul() \\ exp & ::= & BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle) & | & Num(\langle str\rangle) \end{array}
```

Grammatik 1.3: Produktionen für Abstrakten Syntaxbaum in ASF

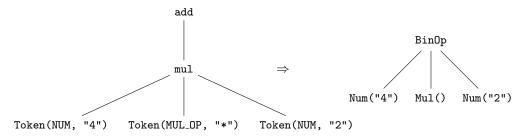


Abbildung 1.5: Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.

Die Baumdatenstruktur des Ableitungsbaumes und Abstrakten Syntaxbaumes ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Programmes ausführen muss möglichst effizient auszuführen und auf unkomplizierte Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die Syntaktische Analyse zu geben, sind in Abbildung 1.6 die einzelnen Zwischenschritte von den Tokens der Lexikalischen Analyse zum Abstrakten Syntaxbaum anhand des fortgeführten Beispiels aus Unterkapitel 1.3 veranschaulicht. In Abbildung 1.6 werden die Darstellungen des Ableitungsbaumes und des Abstrakten Syntaxbaumes verwendet, wie sie vom PicoC-Compiler ausgegeben werden. In der Darstellung des PicoC-Compilers stellen die verschiedenen Einrückungen die verschiedenen Ebenen dieser Bäume dar. Die Bäume wachsen von der Wurzel von links-nach-rechts zu den Blättern.

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

Abbildung 1.6: Veranschaulichung der Syntaktischen Analyse.

1.5 Code Generierung

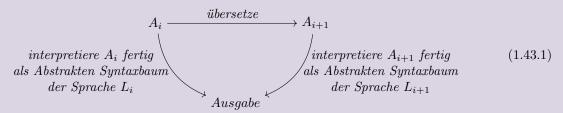
In der Code Generierung steht man nun dem Problem gegenüber einen Abstrakten Syntaxbaum einer Sprache L_1 in den Abstrakten Syntaxbaum einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man Passes (Definition 1.43) nennt. So wie es auch schon mit dem Ableitungsbaum in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum Abstrakten Syntaxbaum kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit Transformern und Visitors ein Abstrakter Syntaxbaum generiert werden.

Definition 1.43: Pass

/

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

Für jeden Pass und für einen beliebigen Abstrakten Syntaxbaum A_i gilt ähnlich, wie bei einem vollständigen Compiler in 1.43.1, dass:



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

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

1.5.1 Monadische Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 1.44) und **Anweisung** (Definition 1.45) wichtig.

Definition 1.44: Ausdruck (bzw. engl. Expression)



Code, der eine semantische Bedeutung hat und in einem bestimmten Kontext ausgewertet werden kann, um einen Wert zu liefern oder etwas zu deklarieren.

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

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

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

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

 $Aufgebaut\ sind\ Ausdr\"{u}cke\ meist\ aus\ Kostanten,\ Variablen,\ Funktionsaufrufen,\ Operatoren\ usw.^{ab}$

^aEin Ausdruck ist z.B 21 * 2;.

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

Definition 1.45: Anweisung (bzw. engl. Statement)

Code, der eine Vorschrifft darstellt, die ausgeführt werden soll und als ganzes keinen Wert liefert und nichts deklariert. Eine Anweisung kann sich jedoch aus ein oder mehreren Ausdrücken zusammensetzen, die dies tun.

Anweisungen sind zentrale Elemente Imperativer Programmiersprachen, die sich zu einem großen Teil aus Folgen von Anweisungen zusammensetzen.

In Maschinensprachen werden Anweisungen häufig als Befehle bezeichnet. ab

^aEine Anweisung ist z.B int var = 21 * 2;.

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

Hat man es mit einer Programmiersprache zu tun, deren Programme Unreine Anweisungen (Definition 1.47) besitzen, so ist es sinnvoll einen Pass einzuführen, der Unreine Ausdrücke von den Anweisungen trennt, damit diese zu Reinen Anweisungen (Definition 1.46) werden. Das wird erreicht, indem man aus jedem Unreinen Ausdruck einen vorangestellten Ausdruck macht, den man vor die jeweilige Anweisung setzt, mit welcher der Unreine Ausdruck gemischt war. Der Unreine Ausdruck muss als erstes ausgeführt werden, für den Fall, dass der Effekt, den ein Unreiner Ausdruck hat die Reine Anweisung, mit der er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 1.46: Reiner Ausdruck / Reine Anweisung (bzw. engl. pure expression)



Ein Reiner Ausdruck ist ein Ausdruck, der rein ist. Das bedeutet, dass dieser Ausdruck keine Nebeneffekte erzeugt. Ein Nebeneffekt ist eine Bedeutung, die ein Ausdruck hat, die sich nicht mit Maschinencode darstellen lässt. Sondern z.B. intern etwas am weiteren Kompiliervorgang ändert^a.

Eine Reine Anweisung ist eine Anweisung, bei der alle Ausdrücke aus denen sich die Anweisung unter anderem zusammensetzt rein sind.^b

^aZ.B. ist die Allokation von Variablen int var kein Reiner Ausdruck. Eine Allokation bestimmt den Wert einiger Immediates im finalen Maschinencode, aber entspricht keiner Folge von Maschinenbefehlen.

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

Definition 1.47: Unreiner Ausdruck / Unreine Anweisung



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

Eine Unreine Anweisung ist eine Anweisung, bei der mindestens einer der Ausdrücke aus denen sich die Anweisung unter anderem zusammensetzt unrein ist.^a

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

Auf diese Weise sind alle Anweisungen in Monadischer Normalform (Definiton 1.48).

Definition 1.48: Monadische Normalform (bzw. engl. monadic normal form)

7

Code ist in Monadischer Normalform, wenn dieser nach einer Grammatik in Monadischer Normalform abgeleitet wurde.

Eine Konkrete Grammatik ist in Monadischer Normalform, wenn alle ableitbaren Anweisungen rein sind. Oder sehr allgemein ausgedrückt, wenn Reines und Unreines klar voneinander getrennt ist.^a

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

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

Ein Beispiel für das Vorgehen, Code in die Monadische Normalform zu bringen, ist in Abbildung 1.7 zu sehen. Der Einfachheit halber wurde auf die Darstellung in Abstrakter Syntax verzichtet, welche allerdings zum großen Teil in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.7 sind daher in Konkreter Syntax²⁰ aufgeschrieben.

Links in der Abbildung 1.7 ist der Ausdruck mit dem Nebeneffekt, eine Variable zu definieren: int var, mit dem Ausdruck für eine Zuweisung exp = 5 % 4 gemischt: int var = 5 % 4. Der Unreine Definitionsausdruck int var muss daher vorangestellt werden, wie es rechts in Abbildung 1.7 dargestellt ist²¹.



Abbildung 1.7: Codebeispiel dafür Code in die Monadische Normalform zu bringen.

Die Aufgabe eines solchen Passes ist es, den Abstrakten Syntaxbaum der Syntax von Maschinenbefehlen anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in Maschinenbefehlen haben. Somit wird eine Seperation von Subbäumen, die keine Entsprechung in Maschinenbefehlen haben und denen, die eine haben bewerkstelligt wird. Eine Reine Anweisung ist Maschinenbefehlen ähnlicher als eine Unreine Anweisung. Somit sparrt man sich in der Implementierung Fallunterscheidungen, indem Reine Ausdrücke und Reine Anweisungen direkt in Maschinenbefehle übersetzt werden können und nicht unterschieden werden muss, ob darin Unreine Ausdrücke vorkommen.

1.5.2 A-Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 1.44) und **Anweisung** (Definition 1.45) wichtig.

Eine Programmiersprache L_1 soll in eine Maschinensprache L_2 kompiliert werden. Im Falle dessen, dass es sich bei einer Sprache L_1 um eine höhere Programmiersprache und bei L_2 um eine Maschinensprache handelt, ist es fast unerlässlich einen Pass einzuführen, der Komplexe Ausdrücke (Definition 1.51) in

²⁰Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

²¹Obwohl hinter int var ein ; steht, ist es immer noch ein Ausdruck. Allerdings gibt es keine einheitliche Festlegung, was eine Anweisung ist und was nicht, es wurde für diese Schrifftliche Ausarbeitung der Bachelorarbeit nur so definiert.

Anweisungen und Ausdrücken verhindert. Das wird erreicht, indem man aus den Komplexen Ausdrücken vorangestellte Ausdrücke macht, in denen die Komplexen Ausdrücke temporären Locations (Definition 1.49) zugewiesen werden und dann anstelle des Komplexen Ausdrucks auf die jeweilige temporäre Location zugegriffen wird.

Definition 1.49: Location

Z

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

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

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

Sollte der Komplexe Ausdruck, welcher einer temporären Location zugewiesen wird, Teilausdrücke enthalten, die komplex sind, muss das gleiche Vorgehen erneut für die Teilausdrücke angewandt werden, bis alle Komplexen Ausdrücke nur noch Atomare Ausdrücke (Definiton 1.50) enthalten, falls sie sich überhaupt in weitere Teilausdrücke aufteilen lassen.

Sollte es sich bei dem Komplexen Ausdruck um einen Unreinen Ausdruck handeln, welcher nur einen Nebeneffekt ausführt und sich nicht in Maschinenbefehle übersetzen lässt, so wird aus diesem ein vorangestellter Ausdruck gemacht, welcher einfach nur den Nebeneffekt dieses Unreinen Ausdrucks ausführt und keiner temporären Location zugewiesen wird.

Definition 1.50: Atomarer Ausdruck



Ein Atomarer Ausdruck ist ein Reiner Ausdruck (Definition 1.46), der keinem kompletten Maschinenbefehl entspricht, sondern nur ein Argument, wie z.B. einen Immediate in einer Folge von Maschinenbefehlen festlegt.^a

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

Bei einem üblichen Compiler, bei dem z.B. Register für temporäre Zwischenergebnisse genutzt werden und der Maschinenbefehlssatz es erlaubt zwei Register miteinander zu verechnen²² sind Atomare Ausdrücke z.B. eine Variable (z.B. var), eine Zahl (z.B. 12) oder ein ASCII-Zeichen (z.B. 'c'), da diese häufig direkt über Register zugreifbar sind, die direkt mit einem Maschinenbefehl verechnet werden können^{23–24}.

Im Fall des PicoC-Compilers ist ein Zugriff auf eine Location (z.B. stack(i)) der einzige Atomare Ausdruck, da der PicoC-Compiler so umgesetzt ist, dass er alle Zwischenergebnisse auf dem Stack speichert und dort dann auf diese zugreift, um sie in Register zu laden und miteinander zu verechnen²⁵. Aus diesem Grund braucht es mindestens einen Maschinenbefehl²⁶, um z.B. eine Zahl überhaupt für einen Maschinenbefehl zugreifbar zu machen, was der Definition 1.50 widerspricht. Daher sind z.B. Zahlen beim PicoC-Compiler keine Atomaren Ausdrücke.

 $^{^{22}{\}rm Z.B.}$ Addieren oder Subtraktion von zwei Registerinhalten.

 $^{^{23}\}mathrm{Mit}$ dem RETI-Befehlssatz wäre das durchaus möglich, durch z.B. MULT ACC IN2.

²⁴Werden allerdings keine Register für Zwischenergebnisse genutzt werden, braucht man mehrere Maschinenbefehle, um die Zwischenergebnisse auf den Stack zu speichern und ein Stackpointer Register anzupassen.

²⁵Der PicoC-Compiler nutzt, anders als es geläufig ist keine Register und Graph Coloring (Definition ??) inklusive Liveness Analysis (Definition ??) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den Hauptspeicher, wobei temporäre Zwischenergebnisse auf den Stack gespeichert werden. Beim PicoC-Compiler sollte sich an die Paradigmen aus der Vorlesung Scholl, "Betriebssysteme" gehalten werden.

²⁶Genauergesagt 4.

Definition 1.51: Komplexer Ausdruck

Z

Ein Komplexer Ausdruck ist ein Ausdruck, der nicht atomar ist.^a

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

Im Fall des PicoC-Compilers sind Komplexe Ausdrücke z.B. 5 % 4, -1, fun(12) oder int var, da diese zur Berechnung auf jeden Fall mehrere Maschinenbefehle benötigen, was Definition 1.51 widerspricht oder unrein sind. Die Teilausdrücke 4, 5, 1 müssen erst auf den Stack geschrieben werden, um dann in Register geladen zu werden, damit dann der gesamte Komplexe Ausdrück berechnet werden kann. Die Ausdrücke fun(12) und int var sind unrein und daher Komplexe Ausdrücke.

In Abbildung 1.8 ist zur besseren Vorstellung die Einteilung von Komplexen, Atomaren, Unreinen und Reinen Ausdrücken veranschaulicht. Des Weiteren sind in der Abbildung alle Ausdrücke ausgegraut, welche die Monadische Normalform nicht erfüllen. Hierbei wird vom PicoC-Compiler ausgegangen, bei dem nur ein Zugriff auf eine Location (z.B. stack(i)) einen Atomaren Ausdruck darstellt.

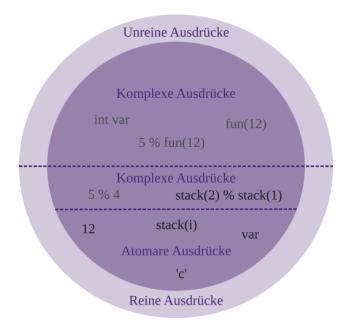


Abbildung 1.8: Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke.

Es wird in dem gerade beschriebenen Pass dafür gesorgt, dass alle Anweisungen und Ausdrücke in A-Normalform²⁷ (Definition 1.52) sind. Wenn eine Konkrete Grammatik in A-Normalform ist, ist diese per Definition 1.52 auch automatisch in Monadischer Normalform, genauso, wie ein Atomarer Ausdruck nach Definition 1.50 auch ein Reiner Ausdruck ist.

Definition 1.52: A-Normalform (ANF)

7

Code ist in A-Normalform, wenn dieser nach einer Konkreten Grammatik in A-Normalform abgeleitet ist.

Eine Konkrete Grammatik ist in A-Normalform, wenn sie in Monadischer Normalform (Definition 1.48) ist und wenn alle Komplexen Ausdrücke nur Atomare Ausdrücke enthalten,

²⁷Das 'A' kommt vermutlich von "atomar" bzw. engl. "atomic", weil alle Komplexen Ausdrücke nur noch Atomare Ausdrücke enthalten dürfen.

falls sie sich überhaupt in weitere Teilausdrücke einteilen lassen.

Eine Abstrakte Grammatik ist in A-Normalform, wenn die Konkrete Grammatik für welche sie definiert wurde in A-Normalform ist. abc

```
<sup>a</sup>A-Normalization: Why and How (with code).
```

Ein Beispiel für dieses Vorgehen, Code in die A-Normalform zu bringen, ist in Abbildung 1.9 zu sehen. Der Einfachheit halber wurde auf die Darstellung in Abstrakter Syntax verzichtet, welche allerdings zum großen Teil in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.7 sind daher in Konkreten Syntax²⁸ aufgeschrieben.

Um konsistent mit der Implementierung zu sein und später keine Verwirrung zu erzeugen, wird beim Beispiel in Abbildung 1.9 vom PicoC-Compiler ausgegangen, bei dem Variablen (z.B. var), Zahlen (z.B. 12) oder ASCII-Zeichen (z.B. 'c') Komplexe Ausdrücke darstellen.

Die Ausdrücke 4;, x;, usw. für sich sind in diesem Fall Komplexe Ausdrücke, deren Wert einer Location, in diesem Fall einer Speicherzelle des Stack zugewiesen werden. Auf das Ergebnis dieser Komplexen Ausdrücke wird mittels stack(2) und stack(1) zugegriffen, um diese in Register zu schreiben und dann z.B. im Komplexen Ausdruck stack(2) % stack(1) miteinander zu verrechnen und das Ergebnis wiederum einer Location, in diesem Fall ebenfalls einer Speicherzelle des Stack zuzuweisen. Dieses Ergebnis wird dann vom Stack stack(1) in die Globalen Statischen Daten global(0) gespeichert, wo die Variable x allokiert ist. Die Zahlen 0, 1, 2 sind dabei hierbei relative Adressen auf dem Stack und in den Globalen Statischen Daten.

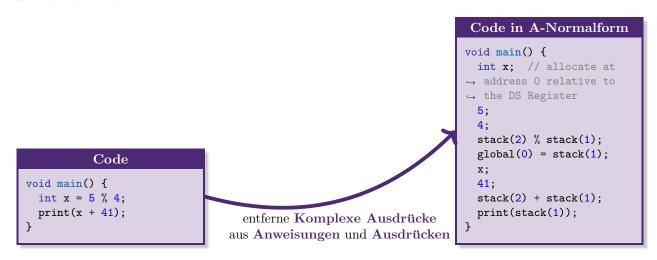


Abbildung 1.9: Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.

Ein Pass, wie er gerade beschrieben wurde hat vor allem in erster Linie die Aufgabe, den Abstrakten Syntaxbaum der Syntax von Maschinenbefehlen besonders dadurch anzunähren, dass er die Anweisungen weniger komplex macht und diese dadurch den ziemlich simplen Maschinenbefehlen syntaktisch ähnlicher sind. Des Weiteren vereinfacht dieser Pass die Implementierung der nachfolgenden Passes enorm, indem weniger Fallunterscheidungen nötig sind, da Anweisungen wie z.B. Zuweisungen nur noch die Form global(rel_addr) = stack(1) haben, welche zudem viel einfacher verarbeitet werden kann.

^bBolingbroke und Peyton Jones, "Types are calling conventions".

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

²⁸Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

Alle weiteren denkbaren Passes sind zu spezifisch auf bestimmte Anweisungen und Ausdrücke ausgelegt, als das sich zu diesen allgemein etwas mit einer Theorie dahinter sagen lässt. Alle Passes, die zur Implementierung des PicoC-Compilers geplant und ausgedacht wurden sind im Unterkapitel 2.3.1 erklärt.

1.5.3 Ausgabe des Maschinencodes

Nachdem alle Passes durchgearbeitet wurden, ist es notwendig aus dem finalen Abstrakten Syntaxbaum den eigentlichen Maschinencode in Konkreter Syntax zu generieren. In üblichen Compilern wird hier für den Maschinencode eine binäre Repräsentation gewählt²⁹. Der Weg von der Abstrakten Syntax zur Konkreten Syntax ist allerdings wesentlich einfacher, als der Weg von der Konkreten Syntax zur Abstrakten Syntax, für die eine gesamte Syntaktische Analyse, die eine Lexikalische Analyse beinhaltet durchlaufen werden muss.

Jeder Knoten des Abstrakten Syntaxbaumes erhält dazu eine Methode, welche hier to_string genannt wird, die eine Textrepräsentation seiner selbst und all seiner Knoten, mit an den richtigen Stellen passend gesetzten Semikolons; öffnenden- und schließenden runden Klammern (), öffnenden- und schließenden geschweiften Klammern {} usw. ausgibt. Dabei wird der gesamte Abstrakte Syntaxbaum durchlaufen und die Methode to_string zur Ausgabe der Textrepräsentation der verschiedenen Knoten aufgerufen, die wiederum die Methode to_string ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend zusammenfügen und selbst zurückgebeben.

1.6 Fehlermeldungen

Wenn bei einem Compiler ein unerwünschtes Verhalten der folgenden Kategorien³⁰ eintritt:

- 1. In der Lexikalischen oder Syntaktischen Analyse tritt ein Fall ein, der nicht in der Syntax der Sprache des Compilers abgedeckt ist, z.B.:
 - Der Lexer kann ein Lexeme nicht mit der Konkreten Grammatik für die Lexikalische Analyse G_{Lex} ableiten. Der Lexer ist genaugenommen ein Teil des Parsers und ist damit bereits durch den nachfolgenden Punkt "Parser" abgedeckt. Um die unterschiedlichen Ebenen, Lexikalische und Syntaktische Analyse gesondert zu betrachten wurde der Lexer an dieser Stelle ebenfalls kurz eingebracht.
 - Der Parser³¹ entscheidet das Wortproblem (Definition 1.20) für ein Eingabeprogramm³² mit 0, also das Eingabeprogramm lässt sich nicht durch die Konkrete Grammatik $G_{Lex} \uplus G_{Parse}$ des Compilers ableiten.
- 2. In den Passes tritt ein Fall ein, der nicht in der Syntax der Sprache des Compilers abgedeckt ist, z.B.:
 - Eine Variable wird verwendet, obwohl sie noch nicht deklariert ist.
 - Bei einem Funktionsaufruf werden mehr oder weniger Argumente, Argumente des falschen Datentyps oder Argumente in der falschen Reihenfolge übergeben, als sie im Funktionsprototyp angegeben sind.
- 3. Während der Laufzeit des Compilers tritt ein Ereignis ein, das nicht durch die Semantik der Sprache

²⁹Da der PicoC-Compiler vor allem zu Lernzwecken konzipiert ist, wird bei diesem der Maschinencode allerdings in einer menschenlesbaren Repräsentation ausgegeben

 $^{^{30}}Errors\ in\ C/C++$ - Geeks for Geeks.

 $^{^{31}\}mathrm{Bzw.}$ der **Erkenner** innerhalb des Parsers.

³²Bzw. ein **Wort**.

des Compilers abgedeckt ist oder welches das Betriebssystem nicht erlaubt, z.B.:

- Eine nicht erlaubte Operation, wie Division durch 0 (z.B. 42 / 0) soll ausgeführt werden.
- Segmentation Fault: Wenn auf Speicher zugegriffen wird, der vom Betriebssystem geschützt ist.

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

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

wird eine Fehlermeldung (Definition 1.53) ausgegeben.

Definition 1.53: Fehlermeldung



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

- über stdout oder stderr im einem Terminal Emulator oder richtigen Terminal.
- über eine Dialogbox in einer Graphischen Benutzeroberfläche^b oder Zeichenorientierten Benutzerschnittstelle^c.
- über ein Register oder eine spezielle Adresse des Hauptspeichers mithilfe eines Wertes.
- über eine Logdatei^d auf einem Speichermedium.

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

 $[^]b {\rm In}$ engl. Graphical User Interface, kurz GUI.

^cIn engl. Text-based User Interface, kurz TUI.

^dIn engl. log file.

2 Implementierung

In diesem Kapitel wird, nachdem im Kapitel 1 die nötigen theoretischen Grundlagen des Compilerbau vermittelt wurden, nun auf die Implementierung des PicoC-Compilers eingegangen. Aufgeteilt in die selben Kategorien Lexikalische Analyse 2.1, Syntaktische Analyse 2.2 und Code Generierung 2.3, wie in Kapitel 1, werden in den folgenden Unterkapiteln die einzelnen Zwischenschritte vom einem Programm in der Konkreten Syntax der Sprache L_{PicoC} hin zum einem Programm mit derselben Semantik in der Konkreten Syntax der Sprache L_{RETI} erklärt.

Für das Parsen¹ des Programmes in der Konkreten Syntax der Sprache L_{PicoC} wird das Lark Parsing Toolkit² verwendet. Das Lark Parsing Toolkit ist eine Bibliothek, die es ermöglicht mittels einer in einem eigenen Dialekt der Erweiterten Back-Naur-Form (Definition 2.3 bzw. für den Dialekt von Lark Definition 2.4) spezifizierten Konkreten Grammatik ein Programm in Konkreter Syntax zu parsen und daraus einen Ableitungsbaum für die kompilerintere Weiterverarbeitung zu generieren.

Definition 2.1: Metasyntax

Z

Steht für den Aufbau einer Metasprache (Definition 2.2), der durch eine Grammatik oder Natürliche Sprache beschrieben werden kann.

Definition 2.2: Metasprache

1

Eine Sprache, die dazu genutzt wird andere Sprachen zu beschreiben^a.

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

Definition 2.3: Erweiterte Backus-Naur-Form (EBNF)



Die Erweiterte Backus-Naur-Form^a ist eine Metasyntax (Definition 2.1), die dazu verwendet wird Kontextfreie Grammatiken darzustellen.

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

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

Weitere Details sind in der Spezifikation des Standards unter Link^e zu finden. Allerdings werden in der Praxis, wie z.B. in Lark oft eigene abgewandelte Notationen wie in Definition 2.4 verwendet.

¹Wobei beim Parsen auch das Lexen inbegriffen ist.

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

³Shinan, lark.

Definition 2.4: Dialekt der Erweiterten Backus-Naur-Form aus Lark

1

Das Lark Parsing Toolkit verwendet eine eigene Notation für die Erweiterte Backus-Naur-Form (Definition 2.3), die sich teilweise in einzelnen Aspekten von der Syntax aus dem Standard unterscheidet und unter Link^a dokumentiert ist.

Wichtige Unterschiede dieses Dialekts sind hierbei z.B.:

• für die Darstellung von Optionaler Wiederholung wird der aus regulären Ausdrücken bekannte *-Quantor zusammen mit optionalen runden Klammern () verwendet (z.B. ()*). Die Verwendung des *-Quantors kann wie in Umformung 2.4.1 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b*\} \quad \Rightarrow \quad \{a := b_tmp, \ b_tmp := b \ b_tmp \ \mid \ \varepsilon\} \tag{2.4.1}$$

• für die Darstellung von mindestents 1-Mal Wiederholung wird der ebenfalls aus regulären Ausdrücken bekannte +-Operator zusammen mit optionalen runden Klammern () verwendet (z.B. ()+). Die Verwendung des +-Quantors kann wie in Umformung 2.4.2 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a := b+\} \quad \Rightarrow \quad \{a := b \ b_tmp, \ b_tmp := b \ b_tmp \mid \varepsilon\} \tag{2.4.2}$$

• für alle ASCII-Symbole zwischen z.B. _ und ~ als Alternative aufgeschrieben kann auch die Abkürzung "_"..."~" verwendet werden. Die Verwendung dieser Schreibweise kann wie in Umformung 2.4.3 zu sehen ist auch wieder zu normaler Erweiterter Backus-Naur-Form umgeschrieben werden.

$$\{a ::= "ascii1" ... "ascii2"\} \Rightarrow \{a ::= "ascii1" \mid ... \mid "ascii2"\}$$
 (2.4.3)

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

Das Lark Parsing Toolkit wurde vor allem deswegen gewählt, weil es sehr einfach in der Verwendung ist. Andere derartige Tools, wie z.B. ANTLR⁴ sind Parser Generatoren, die zur Konkreten Grammatik einer Sprache einen Parser in einer vorher bestimmten Programmiersprache generieren, anstatt wie das Lark Parsing Toolkit bei Angabe einer Konkreten Grammatik direkt ein Programm in dieser Konkreten Grammatik parsen und einen Ableitungsbaum dafür generieren zu können. Lark besitzt des Weiteren eine sehr gute Dokumentation Welcome to Lark's documentation! — Lark documentation.

Neben den Konkreten Grammatiken, die aufgrund der Verwendung des Lark Parsing Toolkit in einem eigenen Dialekt der Erweiterten Back-Naur-Form spezifiziert sind, werden in den folgenden Unter-

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

^bNebel, "Theoretische Informatik".

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

^dAlso von Terminalsymbolen und Nicht-Terminalsymbolen.

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

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

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

 $^{^{4}}ANTLR$.

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

Die Notation für die Abstrakte Syntax unterscheidet sich bewusst von der Erweiterten Backus-Naur-Form, da in der Abstrakten Syntax Kompositionen von Knoten beschrieben werden, die klar auszumachen sind. Hierdurch würde die Abstrakten Grammatiken nur unnötig verkompliziert, wenn man die Erweiterte Backus-Naur-Form verwenden würde. Es gibt leider keine Standardnotation für Abstrakte Grammatiken, die sich deutlich durchgesetzt hat, daher wird für Abstrakte Grammatiken eine eigene Abstrakte Syntaxform Notation (Definition 2.5) 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 2.5: 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 2.4) 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.

2.1 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 1.3 erläutert ist.

2.1.1 Konkrete Grammatik für die Lexikalische Analyse

In der Konkreten Grammatik 2.1.1 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 2.1.1 für die Lexikalische Analyse definierten Nicht-Terminalsymbole können in der Konkreten Grammatik 2.2.8 für die Syntaktischen Analayse verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl NUM stehen darf.

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

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

Es gibt den Fall, dass ein Wort von mehreren Produktionen erkannt wird, z.B. wird das Wort int sowohl von der Produktion NAME, als auch von der Produktion INT_DT erkannt. Daher ist es notwendig für INT_DT eine Priorität INT_DT.2 zu setzen, damit das Wort int den Tokentyp INT_DT zugewiesen bekommt und nicht NAME.

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

Die Implementierung des Basic Lexer aus dem Lark Parsing Toolkit ist unter Link⁵ zu finden ist. Diese Implementierung ist allerdings zu spezifisch auf Lark zugeschnitten und ist aufgrund dessen, dass sie in der Lage ist nach einer spezifizierten Konkreten Grammatik zu lexen, zu komplex, um sie an dieser Stelle allgemein erklären zu können.

Der Basic Lexer verhält sich allerdings grundlegend so, wie es im Unterkapitel 1.3 erklärt wurde, nur berücksichtigt der Basic Lexer ebenfalls Priortiäten, sodass für den aktuellen Index⁶ im Eingabeprogramm zuerst alle Produktionen der höchsten Priorität geprüft werden. Sobald eine dieser Produktionen ein Lexeme an dem aktuellen Index im Eingabeprogramm ableiten kann, wird hieraus direkt ein Token mit dem entsprechenden Tokentyp dieser Produktion und dem abgeleiteten Tokenwert erstellt. Weitere Produktionen werden nicht mehr geprüft. Ansonsten werden alle Produktionen der nächstniedrigeren Priorität geprüft usw.

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

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

```
/[\wedge \backslash n]*/
COMMENT
                                                  /(. | \n)*? / "*/"
                                                                           L_{-}Comment
                       ::=
                            "//""_{-}"?"#"/[\wedge \setminus n]*/
RETI\_COMMENT.2
                       ::=
                                           "3"
                                    "2"
DIG\_NO\_0
                       ::=
                            "1"
                                                                           L_Arith_Bit
                            "7"
                                    "8"
                                           "9"
DIG\_WITH\_0
                            "0"
                                    DIG\_NO\_0
                       ::=
NUM
                            "0"
                                    DIG\_NO\_0 DIG\_WITH\_0*
                       ::=
                            "_"…"∼"
CHAR
                       ::=
FILENAME
                            CHAR + ".picoc"
                       ::=
LETTER
                            "a"..."z"
                                     | "A".."Z"
                       ::=
                            (LETTER | "_")
NAME
                       ::=
                                (LETTER | DIG_WITH_0 | "_")*
                            NAME | INT_NAME | CHAR_NAME
name
                       ::=
                            VOID\_NAME
                            "!"
LOGIC_NOT
                       ::=
                            " \sim "
NOT
                       ::=
                            "&"
REF\_AND
                       ::=
                            SUB\_MINUS \mid LOGIC\_NOT \mid
                                                               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 2.1.1: Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF

2.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 2.1 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 2.1: PicoC-Code des Codebeispiels.

Die vom Basic Lexer des Lark Parsing Toolkit erkannten Tokens sind Code 2.2 zu sehen.

Code 2.2: Tokens für das Codebeispiel.

2.2 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.

2.2.1 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 2.1 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	recitis, daini miks —
		Adresse-von	
3	a*b a/b a%b	Multiplikation, Division und Modulo	
4	a+b a-b	Addition und Subtraktion	
5	a< <b a="">>b	Bitweise Linksshift und Rechtsshift	
6	a <b a<="b</td"><td>Kleiner, Kleiner Gleich, Größer, Größer</td><td></td>	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 2.1: Präzedenzregeln von PicoC.

Würde man diese Operatoren ohne Beachtung von Präzedenzreglen (Definition 1.28) und Assoziativität (Definition 1.27) in eine Konkrete Grammatik verarbeiten wollen, so könnte eine Konkrete Grammatik $G = \langle N, \Sigma, P, exp \rangle$ 2.2.1 dabei rauskommen.

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

Grammatik 2.2.1: Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet

Die Konkrete Grammatik 2.2.1 ist allerdings mehrdeutig (Definition 1.26), 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 2.5.1 als auch über die Linksableitung 2.5.2 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 2.1 zu sehen ist. Da hier nur ein Konzept vermittelt werden soll, entsprechen die beiden Ableitungsbäume in Abbildung 2.1 nicht 1-zu-1 den Ableitungen 2.5.1 und 2.5.2, sondern sind vereinfacht.



Abbildung 2.1: Ableitungsbäume zu den beiden Ableitungen.

Der linke Baum entspricht Ableitung 2.5.1 und der rechte Baum entspricht Ableitung 2.5.2. Würde man in den Ausdrücken, die von diesen Bäumen darsgestellt sind Klammern setzen, um die Präzedenz sichtbar zu machen, so würde Ableitung 2.5.1 die Klammerung (3 * 1) && 4 haben und die Ableitung 2.5.2 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 2.1 betrachtet. Für jede **Präzedenzstufe** in der Tabelle 2.1 wird eine eigene Produktion erstellt, wie es in Grammatik 2.2.2 dargestellt ist. Zudem braucht es eine **Produktion** primexp für die "höchste" **Präzedenzstufe**, welche **Literale**, wie 'c', 5 oder var und geklammerte Ausdrücke wie (3 & 14) abdeckt.

$\overline{prim_exp}$::=	 $L_Arith_Bit + L_Array$
$post_exp$::=	 + $LPntr$ $+$ $LStruct$
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 2.2.2: Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet

Einige Bezeichnungen von Nicht-Terminalsymbolen auf der linken Seite des ::=-Symbols in Grammatik 2.2.2 sind in Tabelle 2.2 ihren jeweiligen Operatoren zugeordnet, für welche sie zuständig sind.

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

Tabelle 2.2: Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren.

Als nächstes müssen die einzelnen **Produktionen** entsprechend der **Ausdrücke** für die sie zuständig sind definiert werden. Jede der **Produktionen** soll nur Ausdrücke **erkennen** können, deren **Präzedenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzedenzstufe **höher** ist. Z.B. soll un_op sowohl den Ausdruck -(3 * 14) als auch einfach nur (3 * 14)⁷ erkennen können, aber nicht 3 * 14 ohne Klammern, da dieser Ausdruck eine **geringe Präzedenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die **Operatoren** linksassoziativ oder **rechtsassoziativ**, unär, binär usw. sind.

Im Folgenden werden Produktionen für alle relevanten Fälle von verschiedenen Kombinationen von Präzedenzen und Assoziativitäten erklärt. Bei z.B. der Produktion un_exp in 2.2.3 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 2.2.3: Beispiel für eine unäre rechtsassoziative Produktion in EBNF

Bei z.B. der Produktion post_exp in 2.2.4 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 2.2.4: Beispiel für eine unäre linksassoziative Produktion in EBNF

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

Bei z.B. der Produktion prec2_exp in 2.2.5 für die binären linksassoziativen Operatoren a+b und a-b ist die Alternative arith_prec2 prec2_op arith_prec1 dafür zuständig, dass mehrere Operationen der Präzedenzstufe 4 in Folge erkannt werden können⁸ (z.B. 3 + 1 - 4, wobei - und + beide Präzedenzstufe 4 haben). Die Alternative arith_prec1 auf der rechten Seite ermöglicht es, dass zwischen den Operationen der Präzedenzstufe 4 auch Operationen der Präzedenzstufe 3 auftauchen können (z.B. 3 + 1 / 4 - 1, wobei - und + beide Präzedenzstufe 4 haben und / Präzedenzstufe 3). Mit der Alternative arith_prec1 ist es möglich, dass auschließlich ein Ausdruck höherer Präzedenz erkannt wird (z.B. 1 / 4).

```
arith\_prec2 ::= arith\_prec2 prec2\_op arith\_prec1 | arith\_prec1
```

Grammatik 2.2.5: Beispiel für eine binäre linksassoziative Produktion in EBNF

Anmerkung Q

Manche Parser^a haben allerdings ein Problem mit Linksrekursion (Definition 1.24), wie sie z.B. in der Produktion 2.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 2.2.5 zur Produktion 2.2.6 umschreibt.

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

Grammatik 2.2.6: Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF

Die von der Grammatik 2.2.6 erkannten Ausdrücke sind dieselben, wie für die Grammatik 2.2.5, allerdings ist die Grammatik 2.2.6 flach gehalten und ruft sich nicht selber auf, sondern nutzt den in der EBNF (Definition 2.3) 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).

 $^a\overline{\mathrm{Zu}}$ 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 2.2.7 definieren.

⁸Bezogen auf Tabelle 2.1.

```
"*"
                                                                                                 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 2.2.7: Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet

2.2.2 Konkrete Grammatik für die Syntaktische Analyse

Die gesamte Konkrete Grammatik 2.2.8 ergibt sich wenn man die Konkrete Grammatik 2.2.7 um die restliche Syntax der Sprache L_{PicoC} erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 2.2.1 erläutert ergibt.

Später in der Entwicklung des PicoC-Compilers wurde die Konkrete Grammatik an die aktuellste kostenlos auffindbare Version der echten Konkreten Grammatik der Sprache L_C , zusammengesetzt aus einer Grammatik für die Syntaktische Analyse $ANSI\ C\ grammar\ (Yacc)$ und Lexikalische Analyse $ANSI\ C\ grammar\ (Lex)$ angepasst⁹. Auf diese Weise konnte sicherer gewährleistet werden kann, dass der PicoC-Compiler sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C^{10} .

In der Konkreten Grammatik 2.2.8 für die Syntaktische Analyse werden einige der Nicht-Terminalsymbole bzw. Tokentypen aus der Konkreten Grammatik 2.1.1 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 2.2.8 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.

⁹An der für die Programmiersprache L_{PicoC} relevanten Syntax hat sich allerdings über die Jahre nichts wichtiges verändert, wie die Konkreten Grammatiken für die Syntaktische Analyse ANSI C grammar (Yacc) old und Lexikalische Analyse ANSI C grammar (Lex) old aus dem Jahre 1985 zeigen.

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

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

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

Grammatik 2.2.8: 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 \mid 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 2.2.9: Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2

Anmerkung Q

In der Konkreten Grammatik 2.2.8 sind alle Grammatiksymbole ausgegraut, die das Bachelorprojekt betreffen. Alle nicht ausgegrauten Grammatiksymbole wurden für die Implementierung der neuen Funktionalitäten, welche die Bachelorarbeit betreffen hinzugefügt.

2.2.3 Ableitungsbaum Generierung

Die in Unterkapitel 2.2.2 definierte Konkrete Grammatik 2.2.8 lässt sich mithilfe des Earley Parsers (Definition 2.6) von Lark dazu verwenden Code, der in der Sprache L_{PicoC} geschrieben ist zu parsen, um einen Ableitungsbaum daraus zu generieren.

Definition 2.6: Earley Parser

Ist ein Algorithmus für das Parsen von Wörtern einer Kontextfreien Sprache. Der Earley Parser ist ein Chart Parser ist, welcher einen mittels Dynamischer Programmierung und Top-Down Ansatz arbeitenden Earley Erkenner (Defintion ?? im ??) nutzt, um einen Ableitungsbaum zu konstruieren.

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

^aJay Earley, "An efficient context-free parsing".

2.2.3.1 Codebeispiel

Der Ableitungsbaum, der mithilfe des Earley Parsers und der Tokens der Lexikalischen Analyse aus dem Beispiel in Code 2.1 generiert wurde, ist in Code 2.3 zu sehen. Im Code 2.3 wurden einige Zeilen markiert, die später in Unterkapitel 2.2.4.1 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
             alloc
 9
                type_spec
10
                  prim_dt
                                  int
11
                pntr_decl
12
                  pntr_deg
13
                  array_decl
14
                    pntr_decl
15
                      pntr_deg
                      array_decl
                        name
                                     attr
18
                        array_dims
19
                    array_dims
20
                      4
                      5
22
       decl_def
23
         fun_def
24
           type_spec
25
             prim_dt
                              void
26
           pntr_deg
27
           name
                        main
28
           fun_params
29
           decl_exec_stmts
30
             decl_part
31
                decl_exp_stmt
33
                    type_spec
34
                      struct_spec
35
                        name
                                     st
36
                    pntr_decl
37
                      pntr_deg
38
                      array_decl
39
                        pntr_decl
40
                          pntr_deg
                          array_decl
                            name
                                          var
                            array_dims
44
                               3
45
                               2
                        array_dims
```

Code 2.3: Ableitungsbaum nach Ableitungsbaum Generierung.

2.2.3.2 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 2.2.8 den linken Seiten des ::=-Symbols¹¹ entsprechen hergenommen und die Blätter sind Terminalsymbole, genauso, wie es in der Definition 1.35 eines Ableitungsbaumes auch schon definiert ist. Die Konkrete Grammatik 2.2.8 des PicoC-Compilers erlaubt es auch, dass in einem Blatt garnichts ε steht, weil es z.B. Produktionen, wie array_dims ::= ("["NUM"]")* gibt, in denen auch das leere Wort ε abgeleitet werden kann.

2.2.4 Ableitungsbaum Vereinfachung

Der Ableitungsbaum in Code 2.3, dessen Generierung in Unterkapitel 2.2.3.1 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 2.2.8 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 2.7) 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 2.7: 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 2.2 zu sehen ist.

^aDer Außengrad ist die Anzahl ausgehender Kanten.

^bDer Innengrad ist die Anzahl eingehener Kanten.

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

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

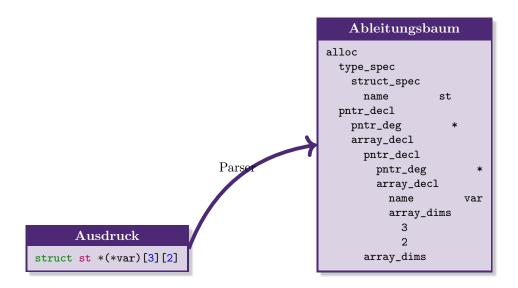


Abbildung 2.2: Ableitungsbaum nach Parsen eines Ausdrucks.

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 2.3 ist zu sehen, wie der Ableitungsbaum aus Abbildung 2.2 mithilfe eines Visitors (Definition 1.39) vereinfacht wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des Visitors aus dem Lark Parsing Toolkit ist unter Link¹² 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 1.39 erklärt wurde.

 $^{^{12}}$ https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.

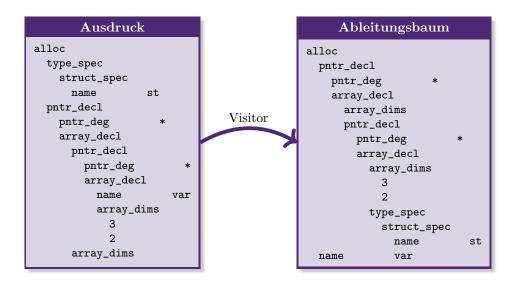


Abbildung 2.3: Ableitungsbaum nach Vereinfachung.

2.2.4.1 Codebeispiel

In Code 2.4 ist der Ableitungsbaum aus Code 2.3 nach der Vereinfachung mithilfe eines Visitors zu sehen.

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

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

Code 2.4: Ableitungsbaum nach Ableitungsbaum Vereinfachung.

2.2.5 Generierung des Abstrakten Syntaxbaumes

Nachdem der Ableitungsbaum in Unterkapitel 2.2.4 vereinfacht wurde, ist der vereinfachte Ableitungsbaum in Code 2.4 nun dazu geeignet, um mit einem Transformer (Definition 1.38) 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 2.4 rauskommen.

Die Implementierung des **Transformers** aus dem **Lark Parsing Toolkit** ist unter Link¹³ zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Transformer** verhält sich allerdings grundlegend so, wie es in Definition 1.38 erklärt wurde.

Den Teilbaum, der rechts in Abbildung 2.4 den Datentyp darstellt, würde man von oben-nach-unten¹⁴ 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.

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

¹⁴In der Informatik wachsen Bäume von oben-nach-unten, von der Wurzel zur den Blättern, bzw. in diesem Beispiel von links-nach-rechts.



Abbildung 2.4: Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen.

Der Abstrakte Syntaxbaum rechts in Abbildung 2.4 ist für die Weiterverarbeitung also ungeeignet, denn für die Adressberechnung bei einer Aneinandereihung von Zugriffen auf Zeigerelemente, Feldelemente oder Verbundsattribute¹⁵ 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 2.5 zustande, der für die Weiterverarbeitung geeignet ist.

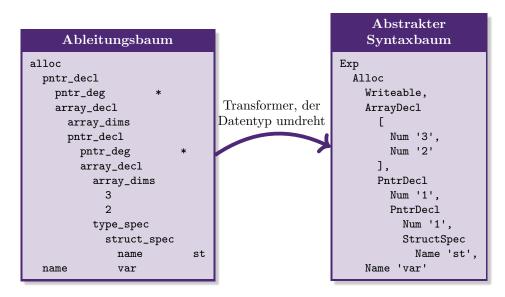


Abbildung 2.5: Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.

Die Weiterverarbeitung des Abstrakten Syntaxbaumes geschieht mithilfe von Passes, welche im Unterkapitel 2.3 genauer beschrieben werden. Da die Knoten des Abstrakten Syntaxbaumes anders als beim

¹⁵Welche in Unterkapitel ?? genauer erläutert wird

Ableitungsbaum nicht die gleichen Bezeichnungen haben, wie Nicht-Terminalsymbole der Konkreten Grammatik, ist es in den folgenden Unterkapiteln 2.2.5.1, 2.2.5.2 und 2.2.5.3 notwendig die Bedeutung der einzelnen PicoC-Knoten, RETI-Knoten und bestimmter Kompositionen dieser Knoten zu dokumentieren. Diese Knoten kommen später im Unterkapitel 2.3 in den unterschiedlichen von den Passes umgeformten Abstrakten Syntaxbäumen vor.

Des Weiteren gibt die Abstrakte Grammatik 2.2.10 in Unterkapitel 2.2.5.4 Aufschluss darüber welche Kompositionen von PicoC-Knoten neben den bereits in Tabelle 2.8 definierten Kompositionen mit Bedeutung insgesamt überhaupt möglich sind.

2.2.5.1 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 sollte¹⁶. Alle PicoC-Knoten, die in den von den verschiedenen Passes generierten Abstrakten Syntaxbäumen vorkommen sind in Tabelle 2.3 mit einem Beschreibungstext dokumentiert.

¹⁶Z.B. steht der PicoC-Knoten Name(str) für einen Bezeichner. Anstatt diesen Knoten in englisch Identifier(str) zu nennen, wurde dieser als Name(str) gewählt, da Name(str) kürzer ist und inuitiver verständlich.

PiocC-Knoten	Beschreibung
Name(str)	Repräsentiert einen Bezeichner (z.B. my_fun, my_var usw.). Das Attribut str ist eine Zeichenkette, welche beliebige Groß- und Kleinbuchstaben (a - z, A - Z), Zahlen (0 - 9) und den Unterstrich (_) enthalten kann. An erster Stelle darf allerdings keine Zahl stehen.
Num(str)	Eine Zahl (z.B. 42, -3 usw.). Hierbei ist das Attribut str eine Zeichenkette, welche beliebige Ziffern (0 - 9) enthal- ten kann. Es darf nur nicht eine 0 am Anfang stehen und danach weitere Ziffern folgen. Der Wert, welcher durch die Zeichenkette dargestellt wird, darf nicht größer als 2 ³² – 1 sein.
Char(str)	Ein Zeichen ('c', '*' usw.). Das Attribut str ist ein Zeichnen der ASCII-Zeichenkodierung.
<pre>Minus(), Not(), DerefOp(), RefOp(), LogicNot()</pre>	Die unären Operatoren un_op: -a, ~a, *a, &a !a.
Add(), Sub(), Mul(), Div(), Mod(), LShift(), RShift(), Xor(), And(), Or(), LogicAnd(), LogicOr()	Die binären Operatoren bin_op: a + b, a - b, a * b, a / b, a % b, a << b, a >> b, a \land b, a & b, a b, a && b, a b.
Eq(), NEq(), Lt(), LtE(), Gt(), GtE()	Die Relationen rel: a == b, a != b, a < b, a <= b, a > b, a >= b.
<pre>Const(), Writeable()</pre>	Die Type Qualifier type_qual: const, was für ein nicht beschreibbare Konstante steht und das nicht Angeben von const, was für einen beschreibbare Variable steht.
<pre>IntType(), CharType(), VoidType()</pre>	Die Type Specifier für Basisdatentypen: int, char, void. Um eine intuitive Bezeichnung zu haben werden sie in der Abstrakten Grammatik einfach nur als Datentypen datatype eingeordnet werden.
Placeholder()	Platzhalter für einen Knoten, der diesen später ersetzt.
BinOp(exp, bin_op, exp)	Container für eine binäre Operation: <exp1> <bin_op> <exp2>.</exp2></bin_op></exp1>
UnOp(un_op, exp)	Container für eine unäre Operation: <un_op> <exp>.</exp></un_op>
Exit(num)	Beendet das laufende Programm und schreibt vor der Be- endigung in das ACC Register einen Exit Code num.
Atom(exp, rel, exp)	Container für eine binäre Relation: <exp1> <rel> <exp2></exp2></rel></exp1>
ToBool(exp)	Container für einen Arithmetischen oder Bitweise Ausdruck, wie z.B. 1 + 3 oder einfach nur 3. Aufgrund des Kontext in dem sich dieser Ausdruck befindet, wird bei einem Ergebnis $x > 0$ auf 1 abgebildet und bei $x = 0$ auf 0.
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 2.3: 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>
V.	* <var>. Hierbei gibt das Attribut num die Anzahl zusam-</var>
	mengefasster Zeiger an und datatype ist der Datentyp,
	auf den der letzte dieser Zeiger zeigt.
Ref(exp, datatype, error_data)	Steht für die Anwendung des Referenz-Operators & <var>,</var>
nor (onp, accordate, orrespondent)	der die Adresse einer Location (Definition 1.49) auf den
	Stack schreiben soll, die über exp bestimmt ist. Zudem besitzt
	er 2 versteckte Attribute, wobei datatype einen Datentyp
	transportiert, der für den RETI Blocks Pass wichtig ist
D ((1 0)	und error data für Fehlermeldungen wichtig ist.
Deref(exp1, exp2)	Container für den Indexzugriff auf einen Feld- oder Zeiger-
	datentyp: <var>[<i>]. Hierbei ist exp1 ein angehängtes weite-</i></var>
	res Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name)
	oder Name('var') und exp2 ist der Index auf den zugegriffen
	werden soll.
ArrayDecl(nums, datatype)	Container, der für den Felddatentyp steht: <prim_dt></prim_dt>
	<pre><var>[<i>]. Hierbei ist das Attribut nums eine Liste von</i></var></pre>
	Num('x'), welche die Dimensionen des Felds angibt und
	datatype ist ein Unterdatentyp (Definition ??).
Array(exps, datatype)	Container für den Initialisierer eines Feldes, z.B. {{1, 2},
	{3, 4}}, dessen Attribut exps weitere Initialisierer für ein
	Feld, weitere Initialisierer für einen Verbund oder Logische
	Ausdrücke beinhalten kann. Des Weiteren besitzt es ein
	verstecktes Attribut datatype, welches für den PicoC-ANF
	Pass Informationen transportiert, die für Fehlermeldungen
	wichtig sind.
Subscr(exp1, exp2)	Container für den Indexzugriff auf einen Feld- oder Zeiger-
1 7 1	datentyp: <var>[<i>]. Hierbei ist exp1 ein angehängtes weite-</i></var>
	res Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name)
	oder Name ('var') und exp2 ist der Index auf den zugegriffen
	werden soll.
StructSpec(name)	Container für die Spezifikation eines Verbundstyps: struct
Dor de coppec (name)	
	<name>. Hierbei legt das Attribut name fest, welchen selbst definierten Verbundstun dieser Knoten enegifiziert</name>
A /	definierten Verbundstyp dieser Knoten spezifiziert.
Attr(exp, name)	Container für den Zugriff auf ein Verbundsattribut:
	<pre><var>>.<attr>>. Hierbei kann exp eine angehängte weitere</attr></var></pre>
	Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name)
	Operation oder ein Name('var') sein. Das Attribut name ist
	das Verbundsattribut auf das zugegriffen werden soll.

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initialisierer eines Verbundes, z.B {.

Tabelle 2.5: PicoC-Knoten Teil 3.

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs,	Container für eine Funktionsdefinition <datatype></datatype>
stmts_blocks)	<pre><fun_name>(<datatype> <param/>) {<stmts>}, wobei datatype</stmts></datatype></fun_name></pre>
20002-210012,	der Rückgabewert der Funktion ist, name der Bezeichner
	der Funktion ist, allocs die Parameter der Funktion
	•
	sind und stmts_blocks eine Liste von Statemetns bzw.
	Blöcken ist, welche diese Funktion beinhaltet. Der Knoten
	Alloc(type_spec, datatype, name) dient dabei als Container
	für die Parameter in allocs.
<pre>NewStackframe(fun_name,</pre>	Erstellt einen neuen Stackframe und speichert den Wert
goto_after_call)	des BAF-Registers der aufrufenden Funktion und die
	Rücksprungadresse nacheinander an den Anfang des neu-
	en Stackframes. Das Attribut fun_name stehte dabei für den
	Bezeichner der Funktion, für die ein neuer Stackframe er-
	stellt werden soll. Das Attribut fun_name dient später dazu den
	Block dieser Funktion zu finden, weil dieser für den weiteren
	Kompiliervorang wichtige Information in seinen versteckte
	Attributen gespeichert hat. Des Weiteren enthält das Attribut
	goto_after_call ein GoTo(Name('addr@next_instr')), welches
	später durch die Adresse des Befehls, der direkt auf den
	-
RemoveStackframe()	Sprungbefehl folgt, ersetzt wird. Container für das Entfernen des aktuellen Stackframes,
RemoveStackirame()	
	durch das Wiederherstellen des im noch aktuellen Stack-
	frame gespeicherten Werts des BAF-Registes der aufrufenden
	Funktion und das Setzen des SP-Registers auf den Wert des
	BAF-Registers vor der Wiederherstellung.
File(name, decls_defs_blocks)	Container der eine Datei repräsentiert, wobei name der Da-
	teiname der Datei ist und decls_defs_blocks eine Liste von
	Funktionen bzw. Blöcken ist.
<pre>Block(name, stmts_instrs, instrs_before,</pre>	Container für Anweisungen, wobei das Attribut name der
<pre>num_instrs, param_size, local_vars_size)</pre>	Bezeichner des Labels (Definition ??) des Blocks ist und
	stmts_instrs eine Liste von Anweisungen oder Befeh-
	len ist. Zudem besitzt er noch 4 versteckte Attribute, wobei
	instrs_before die Zahl der Befehle vor diesem Block zählt,
	num_instrs die Zahl der Befehle ohne Kommentare in
	diesem Block zählt, param_size die voraussichtliche Anzahl
	an Speicherzellen aufaddiert, die für die Parameter der
	Funktion belegt werden müssen und local_vars_size die vor-
	aussichtliche Anzahl an Speicherzellen aufaddiert, die für
	die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für einen Sprung zu einem anderen Block durch
22.20 (210110)	Angabe des Bezeichners des Labels des Blocks, wobei das
	Attribut name der Bezeichner des Labels des Blocks ist, zu
	dem Gesprungen werden soll.
Cinnal alice (Commant (constitution)	
SingleLineComment(prefix, content)	Container für einen Kommentar (//, /* <comment> */), den</comment>
	der Compiler selbst während des Kompiliervorgangs er-
	stellt, damit die Zwischenschritte der Kompilierung und
	auch der finale RETI-Code bei Betrachtung ausgegebener
	Abstrakter Syntaxbäume der verschiedenen Passes leich-
	ter verständlich sind.
RETIComment(str)	Container für einen Kommentar im Code der Form: // #
	comment, der im RETI-Intepreter später sichtbar ist und
	zur Orientierung genutzt werden kann. So ein Kommentar
	wäre allerdings in einer tatsächlichen Implementierung einer
	RETI-CPU nicht umsetzbar und eine Umsetzung wäre auch
	nicht sinnvoll. Der Kommentar ist im Attribut str, welches
	jeder Knoten besitzt gespeichert.
	, J

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.

2.2.5.2 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 2.2.5.1 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 2.2.5.1 mit einem Beschreibungstext dokumentiert.

RETI-Knoten	Beschreibung
Program(name, instrs)	Container der ein Programm repräsentiert: <name></name>
	<instrs>. Hierbei ist name der Name des Programms,</instrs>
	welches ausgeführt werden soll und instrs ist eine Liste
	von Befehlen.
<pre>Instr(op, args)</pre>	Container für einen Befehl: <op> <args>. Hierbei ist op</args></op>
• •	eine Operation und args eine Liste von Argumenten
	für diese Operation.
Jump(rel, im_goto)	Container für einen Sprungbefehl: JUMP <rel> <im>. Hier-</im></rel>
	bei ist rel eine Relation und im_goto ist ein Immediate
	Value Im(val) für die Anzahl an Speicherzellen, um
	die relativ zum Sprungbefehl gesprungen werden soll.
	In einigen Fällen ist im ein GoTo(Name('block.xyz')), das
	später im RETI-Patch Pass durch einen passenden Im-
	mediate Value ersetzt wird.
Int(num)	Container für den Aufruf einer Interrupt-Service-
	Routine: INT <im>. Hierbei ist num die Interrruptvek-</im>
	tornummer (IVN) für die passende Adresse in der Inter-
	ruptvektortabelle, in der die Adresse der Interrupt-
	Service-Routine (ISR) steht, die man aufrufen will.
Call(name, reg)	Container für einen Prozeduraufruf: CALL <name> <reg>.</reg></name>
-	Hierbei ist name der Bezeichner der Prozedur, die auf-
	gerufen werden soll und reg ist ein Register, das als
	Argument an die Prozedur dient. Diese Operation ist
	in der Betriebssysteme Vorlesung ^a nicht deklariert, son-
	dern wurde hinzugefügt, um unkompliziert ein CALL PRINT
	ACC oder CALL INPUT ACC im RETI-Interpreter simulieren
	zu können.
Name(str)	Bezeichner für eine Prozedur, z.B. PRINT, INPUT oder
	den Programnamen, z.B. PROGRAMNAME. Hierbei ist str
	eine Zeichenkette für welche das gleich gilt, wie für
	das str Attribut des PicoC-Knoten Name(str). Dieses
	Argument ist in der Betriebssysteme Vorlesung ^a nicht
	deklariert, sondern wurde hinzugefügt, um Bezeichner,
	wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein Register. Hierbei ist reg das Register
	auf welches zugegriffen werden soll.
Im(str)	Ein Immediate Value (z.B. 42, -3 usw.). Hierbei ist das
	Attribut str eine Zeichenkette, welche beliebige Ziffern
	(0 - 9) enthalten kann. Es darf nur nicht eine 0 am Anfang
	stehen und danach weitere Ziffern folgen. Der Wert,
	welcher durch die Zeichenkette dargestellt wird, darf nicht
	größer als $2^{22} - 1$ sein.
Add(), Sub(), Mult(), Div(), Mod(), Xor(),	Compute-Memory oder Compute-Register Operatio-
Or(), And()	nen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(),	Compute-Immediate Operationen: ADDI, SUBI, MULTI,
<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 rel: <, <=, >, >=, ==, !=, _NOP.
Always(), NOp()	
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(),	Register reg: PC, IN1, IN2, ACC, SP, BAF, CS, DS.
Cs(), Ds()	

^a Scholl, "Betriebssysteme"

2.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung

In Tabelle 2.8 sind jegliche Kompositionen von PicoC-Knoten und RETI-Knoten aufgelistet, die eine besondere Bedeutung haben.

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

Tabelle 2.8: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung.

Anmerkung Q

Um die obige Tabelle 2.8 nicht mit unnötig viel repetetiven Inhalt zu füllen, wurden die zahlreichen Kompositionen ausgelassen, bei denen einfach nur ein $\exp i, i := "1" \mid "2" \mid \varepsilon$ durch $\operatorname{Stack}(\operatorname{Num}('x')), x \in \mathbb{N}$ ersetzt wurde^a.

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

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

2.2.5.4 Abstrakte Grammatik

Die Abstrakte Grammatik der Sprache L_{PicoC} ist in Grammatik 2.2.10 dargestellt.

```
SingleLineComment(\langle str \rangle, \langle str \rangle)
                                                                                      RETIComment()
                                                                                                                          L_{-}Comment
stmt
                  ::=
                                                                                                                          L_Arith_Bit
un\_op
                  ::=
                           Minus()
                                                Not()
bin\_op
                  ::=
                           Add()
                                       |Sub()|
                                                          Mul() \mid Div() \mid Mod()
                           Oplus() \mid And() \mid Or()
                                                     Num(\langle str \rangle) \mid Char(\langle str \rangle)
                           Name(\langle str \rangle)
exp
                  ::=
                           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)
                  ::=
                           LogicNot()
                                                                                                                          L\_Logic
un\_op
                  ::=
                           Eq() \mid NEq() \mid Lt() \mid LtE() \mid
rel
                  ::=
                                                                                           Gt()
                                                                                                         GtE()
                           LogicAnd() \mid LogicOr()
bin\_op
                           Atom(\langle exp \rangle, \langle rel \rangle, \langle exp \rangle)
                                                                        ToBool(\langle exp \rangle)
exp
                  ::=
                                              Writeable()
                                                                                                                          L\_Assign\_Alloc
type\_qual
                  ::=
                           Const()
datatype
                  ::=
                           IntType() \mid CharType() \mid VoidType()
                           Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle))
exp
stmt
                           Assign(\langle exp \rangle, \langle exp \rangle)
                  ::=
                           PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)
                                                                                                                          L\_Pntr
datatype
                  ::=
                           Deref(\langle exp \rangle, \langle exp \rangle)
                                                                Ref(\langle exp \rangle)
exp
                  ::=
                           ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle)
                                                                                                                          L_Array
datatype
                  ::=
exp
                  ::=
                           Subscr(\langle exp \rangle, \langle exp \rangle)
                                                                  Array(\langle exp \rangle +)
                           StructSpec(Name(\langle str \rangle))
                                                                                                                          L\_Struct
datatype
                  ::=
exp
                  ::=
                           Attr(\langle exp \rangle, Name(\langle str \rangle))
                           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))+)
                  ::=
                           If(\langle exp \rangle, \langle stmt \rangle *)
                                                                                                                          L\_If\_Else
stmt
                           IfElse(\langle exp \rangle, \langle stmt \rangle *, \langle stmt \rangle *)
stmt
                           While(\langle exp \rangle, \langle stmt \rangle *)
                                                                                                                          L_{-}Loop
                  ::=
                           DoWhile(\langle exp \rangle, \langle stmt \rangle *)
                           Call(Name(\langle str \rangle), \langle exp \rangle *)
exp
                  ::=
                                                                                                                          L_Fun
stmt
                           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 stmt \rangle *)
                           File(Name(\langle str \rangle), \langle decl\_def \rangle *)
                                                                                                                          L_File
file
```

Grammatik 2.2.10: Abstrakte Grammatik der Sprache L_{PiocC} in ASF

2.2.5.5 Codebeispiel

In Code 2.5 ist der Abstrakte Syntaxbaum zu sehen, der aus dem vereinfachten Ableitungsbaum aus Code 2.4 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',
                     IntType 'int',
18
19
              Name 'attr'
20
          ],
21
       FunDef
22
          VoidType 'void',
23
          Name 'main',
24
          [],
25
26
            Exp
27
              Alloc
28
                 Writeable,
29
                 ArrayDecl
30
31
                     Num '3',
32
                     Num '2'
33
                   ],
34
                   PntrDecl
35
                     Num '1',
                     PntrDecl
36
37
                        Num '1',
38
                        StructSpec
39
                          Name 'st',
40
                 Name 'var'
41
          ]
42
     ]
```

Code 2.5: Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum.

2.2.5.6 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

¹⁷Spezielle Methode, die immer aufgerufen wird, wenn das Objekt, dass in Besitz dieser Methode ist als Zeichenkette mittels print() oder zur Repräsentation ausgegeben werden soll.

; usw. zur Darstellung der Hierarchie und zur Abtrennung zurück. Der gesamte Abstrakte Syntaxbaum wird durchlaufen und die Magischen _repr_()-Methoden der verschiedenen Knoten aufgerufen, die immer jeweils die _repr_()-Methoden ihrer Kinder aufrufen und die zurückgegebenen Textrepräsentationen passend zusammenfügen und selbst zurückgeben.

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

Da die Konkrette Syntax von RETI-Knoten sehr simpel ist, macht es kaum einen Unterschied in der Erkennbarkeit, bis auf fehlende gescheifte Klammern () usw., ob man die RETI-Knoten in Abstrakter oder Konkreter Syntax schreibt. Daher werden die RETI-Knoten einfach immer direkt in Konkreter Syntax ausgegeben. Auf diese Weise muss nicht beim letzten Pass daran gedacht werden am Ende die Konkrete, statt der Abstrakten Syntax für die RETI-Knoten auszugeben.

Die Ausgabe des Abstrakten Syntaxbaums ist bewusst so gewählt, dass sie sich optisch von der des Ableitungsbaums unterscheidet, indem die Bezeichner der Knoten in UpperCamelCase¹⁸ geschrieben sind. Das steht im Gegensatz zum Ableitungsbaum, dessen Innere Knoten im snake_case geschrieben sind, da sie die Nicht-Terminalsymbole auf den linken Seiten des ::=-Symbols in der Konkreten Grammatik 2.2.8 darstellen, welche in snake_case geschrieben sind.

2.3 Code Generierung

Nach der Generierung eines Abstrakten Syntaxbaums als Ergebnis der Lexikalischen und Syntaktischen Analyse in Unterkapitel 2.2.5, wird in diesem Kapitel auf Basis der verschiedenen Kompositionen von Knoten im Abstrakten Syntaxbaum das gewünschte Endprodukt des PicoC-Compilers, der RETI-Code generiert.

Man steht nun dem Problem gegenüber einen Abstrakten Syntaxbaum der Sprache L_{PicoC} , der durch die Abstrakte Grammatik 2.2.10 spezifiziert ist in einen semantisch gleichen Abstrakten Syntaxbaum der Sprache L_{RETI} umzuformen, der durch die Abstrakte Grammatik 2.3.8 spezifiziert ist. In T-Diagramm-Notation (siehe Unterkapitel 1.1.1) lässt sich das darstellen, wie in Abbildung 2.6.

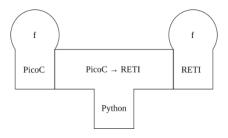


Abbildung 2.6: Kompiliervorgang Kurzform.

Das gerade angesprochene Problem lässt sich, wie in Unterkapitel 1.5 bereits beschrieben vereinfachen, indem man das Problem in mehrere Passes (Definition 1.43) aufteilt, die jeweils ein überschaubares Teilproblem lösen. Man nähert sich Schrittweise immer mehr der Syntax der Sprache L_{RETI} an.

In Abbildung 2.7 ist das **T-Diagramm** aus Abbildung 2.6 detailierter dargestellt. Das **T-Diagramm** gibt einen Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 1.1) des **PicoC-Compilers**

¹⁸ Naming convention (programming).

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.

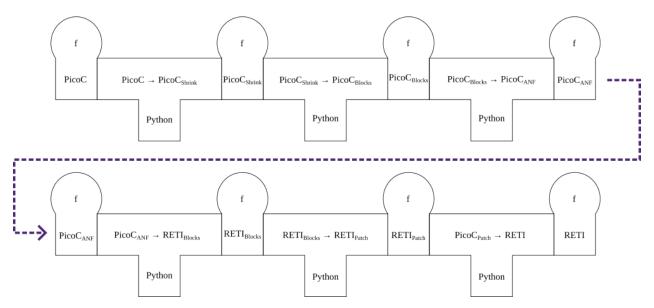


Abbildung 2.7: Architektur mit allen Passes ausgeschrieben.

Im Unterkapitel 2.3.1 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 2.3.1 nicht vertieft wurden. Viele der verwendenten Ansätze zur Lösung dieser Probleme basieren auf der Vorlesung Scholl, "Betriebssysteme" und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem PicoC-Compiler auch in der Praxis implementiert werden konnten.

2.3.1 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 Aspekte¹⁹ 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.

2.3.1.1 PicoC-Shrink Pass

Die 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) in der Untermenge der Sprache L_C , welche die Sprache L_{PicoC} darstellt die gleiche Semantik hat, wie der Operator für den Zugriff auf den Index eines Feldes ar[i]²⁰.

¹⁹In kurz: Zeiger, Felder, Verbunde und Funktionen.

²⁰Wobei *pntr und pntr[0] einander entsprechen.

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

2.3.1.1.1 Abstrakte Grammatik

Die Abstrakte Grammatik 2.3.1 der Sprache L_{PicoC_Shrink} ist fast identisch mit der Abstrakten Grammatik 2.2.10 der Sprache L_{PicoC} , nach welcher der erste Abstrakte Syntaxbaum in der Syntaktischen Analyse generiert wurde. Der einzige Unterschied liegt darin, dass es den Knoten Deref (exp1, exp2) in der Abstrakten Grammatik 2.3.1 nicht mehr gibt. Das liegt daran, dass dieser Pass alle Vorkommnisse des Knoten Deref (exp1, exp2) durch den Knoten Subscr (exp1, exp2) auswechselt.

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

Grammatik 2.3.1: Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF

Anmerkung Q

Alles ausgegraute bedeutet, es hat sich im Vergleich zur letzten Abstrakten Grammatik nichts geändert. Alles rot markierte bedeutet, es wurde entfernt oder abgeändert. Alle normal in schwarz geschriebenen Knoten sind neu hinzugefügt. Das gilt genauso für alle folgenden Grammatiken.

2.3.1.1.2 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 2.6 zur Anschauung der verschiedenen Passes verwendet. Im Code 2.6 ist in der Funktion faculty ein iterativer Algorithmus implementiert, der die Fakultät eines übergebenen Arguments berechnet. Der Algorithmus basiert auf einem Beispielprogramm aus der Vorlesung Scholl, "Betriebssysteme", welches diesen Algorithmus allerdings rekursiv implementiert.

Die rekursive Implementierung des Algorithmus wäre allerdings kein gutes Anschauungsbeispiel, das viele der Aufgaben der verschiedenen Passes bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der Passes, wie z.B. bei der Kompilierung von if-, if-else-, while- und do-while-Anweisungen, wären mit der rekursiven Implementierung aus der Vorlesung nicht veranschaulicht gewesen. Daher wurde die rekursive Implementierung aus der Vorlesung zu einem iterativen Algorithmus 2.6 umgeschrieben, um unter anderem auch if- und while-Statements zu enthalten.

Beide Varianten des Algorithmus wurden zum Testen des PicoC-Compilers verwendet und sind als Tests im Ordner /tests unter Link²¹, unter den Testbezeichnungen example_faculty_rec.picoc und example_faculty_it.picoc zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als Anschauung des jeweiligen Passes, der im jeweiligen Unterkapitel beschrieben wird und werden nicht im Detail erläutert. Viele Details der Passes werden später in den Unterkapiteln ??, ??, ?? und ?? zu Zeigern, Feldern, Verbunden und Funktionen mit eigenen Codebeispielen genauer erklärt und alle sonstigen Details sind dem Bachelorprojekt zuzuordnen, dessen Aspekte in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit nicht im Detail erläutert werden.

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

Code 2.6: PicoC Code für Codebespiel.

In Code 2.7 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',
```

²¹https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```
Name 'faculty',
 8
           Alloc(Writeable(), IntType('int'), Name('n'))
         ],
10
11
           Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
12
           While
13
             Num '1',
14
15
               Ιf
16
                 Atom(Name('n'), Eq('=='), Num('1')),
17
18
                    Return(Name('res'))
19
20
               Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21
               Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22
23
         ],
24
       FunDef
25
         VoidType 'void',
26
         Name 'main',
27
         [],
28
29
           Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30
31
     ]
```

Code 2.7: Abstrakter Syntaxbaum für Codebespiel.

Im PicoC-Shrink Pass ändert sich nichts im Vergleich zum Abstrakten Syntaxbaum in Code 2.7, da das Codebeispiel keine Dereferenzierung *pntr enthält. Es wurde auf ein weiteres Codebeispiel für diesen Pass verzichtet, da din diesem das gleiche zu sehen wäre, wie in Codebeispiel 2.7.

2.3.1.2 PicoC-Blocks Pass

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.

2.3.1.2.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 2.3.1 der Sprache L_{PicoC_Shrink} um die Knoten zu erweitern, die am Anfang dieses Unterkapitels erwähnt wurden. Die Knoten If(exp, stmts), While(exp, stmts) und DoWhile(exp, stmts) gibt es nicht mehr, da sie durch Block(name, stmts_instrs-, GoTo(lable)- und IfElse(exp, stmts1, stmts2)-Knoten ersetzt wurden. Die Funktionsdefinition FunDef(datatype, Name(str), Alloc(Writeable(), datatype, Name(str))*, (block)*) ist nun ein Container für Blöcke Block(Name(str), stmt*) und keine Anweisungen stmt mehr. Das resultiert in der Abstrakten Grammatik 2.3.2 der Sprache L_{PicoC_Blocks} .

$bin_op \qquad ::= Add() \mid Sub() \mid Mul() \mid Div() \mid Mod() \\ \mid Oplus() \mid And() \mid Or() \\ exp \qquad ::= Name(\langle str \rangle) \mid Num(\langle str \rangle) \mid Char(\langle str \rangle) \\ \mid BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle) \\ \mid UnOp(\langle un_op \rangle, \langle exp \rangle) \mid Call(Name('input'), Empty()) \\ \mid Call(Name('print'), \langle exp \rangle) \\ stmt \qquad ::= Exp(\langle exp \rangle)$	L_Arith_Bit L_Logic
$exp \qquad ::= Name(\langle str \rangle) \mid Num(\langle str \rangle) \mid Char(\langle str \rangle) \\ \mid BinOp(\langle exp \rangle, \langle bin_op \rangle, \langle exp \rangle) \\ \mid UnOp(\langle un_op \rangle, \langle exp \rangle) \mid Call(Name('input'), Empty()) \\ \mid Call(Name('print'), \langle exp \rangle) \\ stmt \qquad ::= Exp(\langle exp \rangle) \\ un_op \qquad ::= LogicNot() \\ rel \qquad ::= Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE() \\ \end{cases}$	L_Logic
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	L_Logic
$rel \qquad ::= Eq() NEq() Lt() LtE() Gt() GtE()$	L_Logic
$exp ::= Atom(\langle exp \rangle, \langle rel \rangle, \langle exp \rangle) \mid ToBool(\langle exp \rangle)$	
$type_qual ::= Const() Writeable() $ $datatype ::= IntType() CharType() VoidType() $ $exp ::= Alloc(\langle type_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle)) $ $stmt ::= Assign(\langle exp \rangle, \langle exp \rangle) $	L_Assign_Alloc
$\begin{array}{ll} datatype & ::= & PntrDecl(Num(\langle str \rangle), \langle datatype \rangle) \\ exp & ::= & Ref(\langle exp \rangle) \end{array}$	L_Pntr
$\begin{array}{cccc} datatype & ::= & ArrayDecl(Num(\langle str \rangle) +, \langle datatype \rangle) & & \\ exp & ::= & Subscr(\langle exp \rangle, \langle exp \rangle) & & Array(\langle exp \rangle +) \end{array}$	L_Array
$exp ::= Attr(\langle exp \rangle, Name(\langle str \rangle)) Struct(Assign(Name(\langle str \rangle), \langle exp \rangle) +) decl_def ::= StructDecl(Name(\langle str \rangle),$	L_Struct
$Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))+)$ $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 *)$	$L_{-}Loop$
$\begin{array}{lll} exp & ::= & Call(Name(\langle str \rangle), \langle exp \rangle *) & I \\ stmt & ::= & Return(\langle exp \rangle) \\ decl_def & ::= & FunDecl(\langle datatype \rangle, Name(\langle str \rangle), \\ & & & Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *) \\ & & & & FunDef(\langle datatype \rangle, Name(\langle str \rangle), \\ & & & & & Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) *, \langle block \rangle *) \end{array}$	L_Fun
$block ::= Block(Name(\langle str \rangle), \langle stmt \rangle *)$ $stmt ::= GoTo(Name(\langle str \rangle))$	L_Blocks
$file ::= File(Name(\langle str \rangle), \langle decl_def \rangle *)$	L_File

Grammatik 2.3.2: Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF

Anmerkung Q

Eine Abstrakte Grammatik soll im Gegensatz zu einer Konkreten Grammatik für den Programmierer, der einen darauf aufbauenden Compiler implementiert einfach verständlich sein und stellt daher eine Obermenge aller tatsächlich möglichen Kompositionen von Knoten dar^a.

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

2.3.1.2.2 Codebeispiel

In Code 2.8 sieht man den aus dem Abstrakten Syntaxbaum aus Code 2.7 mithilfe des PicoC-Blocks Pass resultierenden Abstrakten Syntaxbaum. Es wird das Beispiel in Code 2.6 aus Unterkapitel 2.3.1.1 weitergeführt. Es wurden nun eigene Blöcke für die Funktion faculty und die main-Funktion erstellt, in denen die jeweils ersten Anweisungen der jeweiligen Funktionen bis zur letzten Anweisung oder bis zum ersten Auftauchen eines If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(exp, stmts)- oder DoWhile(exp, stmts)-Knoten stehen. Je nachdem, ob ein If(exp, stmts)-, IfElse(exp, stmts1, stmts2)-, While(exp, stmts)-oder DoWhile(exp, stmts)-Knoten auftaucht, werden für die Bedingung und mögliche Branches eigene Blöcke erstellt.

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

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

Code 2.8: PicoC-Blocks Pass für Codebespiel.

2.3.1.3 PicoC-ANF Pass

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 1.52) und damit auch in Monadischer Normalform (Definition 1.48) ist. Um Redundanz zu vermeiden wird zur Erklärung der A-Normalform auf Unterkapitel 1.5.2 verwiesen und zur Erklärung der Monadischen Normalform auf Unterkapitel 1.48 verwiesen.

Zudem wird eine Symboltabelle (Definition 2.8) verwendet. In der Symboltabelle wird beim Anlegen eines neuen Eintrags für eine Variable zunächst eine Adresse an das value or address-Attribut dieses Eintrags zugewiesen, die dem Wert einer von zwei Countern rel_global_addr und rel_stack_addr entspricht. Der Counter rel_global_addr ist für Variablen in den Globalen Statischen Daten und der Counter rel_stack_addr ist für Variablen auf dem Stackframe der momentan aktiven Funktion. Einer der beiden Counter wird nach Anlegen eines Eintrags entsprechend der Größe der angelegten Variable hochgezählt.

Kommt im Programmcode an einer späteren Stelle eine definierte Variable Name('var') vor, so wird mit der Konkatenation des Bezeichners der Variable und des Bezeichners des momentanen Sichtbarkeitsbereichts var@scope²² als Schlüssel in der Symboltabelle der entsprechende Symboltabelleneintrag der Variable var nachgeschlagen. Anstelle des Name(str)-Knotens der Variable wird ein Global(num) bzw. Stackframe(num)-Knoten eingefügt, dessen num-Attribut die Adresse im value or address-Attribut des Symboltabelleneintrags zugewiesen bekommt.

²²Die Umsetzung von Sichtbarkeitsbereichen wird in Unterkapitel?? genauer beschrieben.

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

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

Das Symbol '!' im Suffix global! und das Symbol '@' als Trennzeichen in var@scope wurden aus einem bestimmten Grund verwendet, nämlich, weil kein Bezeichner die Symbole '@' und ! jemals selbst enthalten kann. Die Produktionen für einen Bezeichner in der Konkretten Grammatik $G_{Lex} \uplus G_{Parse}$ (siehe 2.1.1 und 2.2.10) lassen beide Symbole @ und ! nicht zu. Damit ist es ausgeschlossen, dass es zu Problemen kommt, falls ein Benutzer des PicoC-Compilers zufällig auf die Idee kommt eine Funktion auf eine unpassende Weise zu benennen²³.

Definition 2.8: Symboltabelle

1

Eine meist über ein Assoziatives Feld umgesetzte Datenstruktur, die notwendig ist, um das Konzept von Variablen und Konstanten in einer Sprache umzusetzen. Diese Datenstruktur ordnet jedem Symbol^a einer Variablen, Konstanten oder Funktion aus einem Programm, Informationen, wie die Adresse, die Position im Programmcode oder den Datentyp zu, welche später nicht mehr so einfach zugänglich sind, wie zu dem Zeitpunkt, zu dem sie in einen Symboltabelleneintrag gespeichert werden.

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

^aIn der Code Generierung werden Bezeichner als Symbole bezeichnet.

2.3.1.3.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die Abstrakte Grammatik 2.3.2 der Sprache L_{PicoC_Blocks} in die A-Normalform zu bringen. Darunter fällt es unter anderem, dafür zu sorgen, dass Komplexe Knoten, wie z.B. BinOp(exp, bin_op, exp) nur Atomare Knoten enthalten können. Wie es bereits im Unterkapitel 1.5.2 erklärt wurde, ist beim PicoC-Compiler der Knoten Stack(Num(str)) der einzige Atomare Knoten.

Des Weiteren werden auch Funktionen und Funktionsaufrufe aufgelöst, sodass die Blöcke Block(Name(str), stmt*) nun direkt im decls_defs_blocks-Attribut des File(Name(str), decls_defs_blocks*)-Knoten liegen usw.²⁴. Die Symboltabelle ist ebenfalls als Abstrakter Syntaxbaum umgesetzt, wofür in der Abstrakten Grammatik 2.3.3 der Sprache L_{PicoC_ANF} neue Knoten eingeführt wurden. Das ganze resultiert in der Abstrakten Grammatik 2.3.3 der Sprache L_{PicoC_ANF} .

²³Z.B. var@fun2 oder global als Funktionsname.

²⁴Im Unterkapitel ?? wird das Auflösen von Funktionen genauer erklärt.

```
RETIComment()
                                                                                                                                                  L_{-}Comment
stmt
                               SingleLineComment(\langle str \rangle, \langle str \rangle)
                      ::=
                                                                                                                                                  L_Arith_Bit
un\_op
                      ::=
                              Minus()
                                                   Not()
bin\_op
                      ::=
                               Add()
                                          Sub()
                                                             Mul() \mid Div() \mid
                                                                                              Mod()
                                                             |Or()
                               Oplus()
                                            And()
                              Name(\langle str \rangle) \mid Num(\langle str \rangle)
                                                                                Char(\langle str \rangle)
                                                                                                         Global(Num(\langle str \rangle))
exp
                               Stackframe(Num(\langle str \rangle))
                                                                       | Stack(Num(\langle str \rangle))|
                               BinOp(Stack(Num(\langle str \rangle)), \langle bin\_op \rangle, Stack(Num(\langle str \rangle)))
                               UnOp(\langle un\_op \rangle, Stack(Num(\langle str \rangle))) \mid Call(Name('input'), Empty())
                               Call(Name('print'), \langle exp \rangle)
                               Exp(\langle exp \rangle)
                              LogicNot()
                                                                                                                                                  L\_Logic
un\_op
                      ::=
                               Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt()
rel
                                                                                                         GtE()
                      ::=
                               LogicAnd()
                                                      LogicOr()
bin\_op
                      ::=
                               Atom(Stack(Num(\langle str \rangle)), \langle rel \rangle, Stack(Num(\langle str \rangle)))
exp
                      ::=
                              ToBool(Stack(Num(\langle str \rangle)))
type\_qual
                              Const()
                                                 Writeable()
                                                                                                                                                  L\_Assign\_Alloc
                      ::=
                              IntType() \mid CharType() \mid VoidType()
datatype
                      ::=
exp
                              Alloc(\langle type\_qual \rangle, \langle datatype \rangle, Name(\langle str \rangle))
                      ::=
                              Assign(Global(Num(\langle str \rangle)), Stack(Num(\langle str \rangle)))
stmt
                      ::=
                               Assign(Stackframe(Num(\langle str \rangle)), Stack(Num(\langle str \rangle)))
                               Assign(Stack(Num(\langle str \rangle)), Global(Num(\langle str \rangle)))
                               Assign(Stack(Num(\langle str \rangle)), Stackframe(Num(\langle str \rangle)))
                               PntrDecl(Num(\langle str \rangle), \langle datatype \rangle)
                                                                                                                                                  L_{-}Pntr
datatype
                      ::=
                               Ref(Global(\langle str \rangle)) \mid Ref(Stackframe(\langle str \rangle))
                               Ref(Subscr(\langle exp \rangle, \langle exp \rangle \mid Ref(Attr(\langle exp \rangle, Name(\langle str \rangle)))))
                               ArrayDecl(Num(\langle str \rangle)+, \langle datatype \rangle)
                                                                                                                                                  L_-Array
datatupe
                      ::=
                               Subscr(\langle exp \rangle, Stack(Num(\langle str \rangle)))
                                                                                        Array(\langle exp \rangle +)
exp
                      ::=
                               StructSpec(Name(\langle str \rangle))
                                                                                                                                                  L\_Struct
datatype
                      ::=
                               Attr(\langle exp \rangle, Name(\langle str \rangle))
exp
                      ::=
                               Struct(Assign(Name(\langle str \rangle), \langle exp \rangle) +)
decl\_def
                               StructDecl(Name(\langle str \rangle),
                      ::=
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle)) + )
                               IfElse(Stack(Num(\langle str \rangle)), \langle stmt \rangle *, \langle stmt \rangle *)
                                                                                                                                                  L_If_Else
stmt
                      ::=
                              Call(Name(\langle str \rangle), \langle exp \rangle *)
                                                                                                                                                  L-Fun
                      ::=
exp
                               StackMalloc(Num(\langle str \rangle)) \mid NewStackframe(Name(\langle str \rangle), GoTo(\langle str \rangle))
stmt
                      ::=
                               Exp(GoTo(Name(\langle str \rangle))) \mid RemoveStackframe()
                               Return(Empty()) \mid Return(\langle exp \rangle)
decl\_def
                               FunDecl(\langle datatype \rangle, Name(\langle str \rangle))
                      ::=
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))*)
                               FunDef(\langle datatype \rangle, Name(\langle str \rangle),
                                     Alloc(Writeable(), \langle datatype \rangle, Name(\langle str \rangle))*, \langle block \rangle*)
block
                               Block(Name(\langle str \rangle), \langle stmt \rangle *)
                                                                                                                                                  L\_Blocks
                      ::=
stmt
                               GoTo(Name(\langle str \rangle))
                      ::=
                                                                                                                                                  L_File
file
                               File(Name(\langle str \rangle), \langle block \rangle *)
symbol\_table
                               SymbolTable(\langle symbol \rangle *)
                                                                                                                                                  L\_Symbol\_Table
                      ::=
                               Symbol(\langle type\_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)
symbol
                      ::=
                              Empty()
type\_qual
                      ::=
datatype
                      ::=
                               BuiltIn()
                                                    SelfDefined()
                               Name(\langle str \rangle)
name
                      ::=
val
                               Num(\langle str \rangle)
                                                   | Empty()
                      ::=
                               Pos(Num(\langle str \rangle), Num(\langle str \rangle))
                                                                                  Empty()
pos
                      ::=
                               Num(\langle str \rangle)
                                                     Empty()
size
                                                                                                                                                                74
```

2.3.1.3.2 Codebeispiel

In Code 2.9 sieht man die als Abstrakter Syntaxbaum umgesetzte Symboltabelle, in der alle Variablen und Funktionen aus dem weitergeführten Beispiel in Code 2.6 aus Unterkapitel 2.3.1.1 einen Eintrag haben.

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

Code 2.9: Symboltabelle für Codebespiel.

In Code 2.10 sieht man den aus dem Abstrakten Syntaxbaum aus Code 2.8 mithilfe des PicoC-ANF Pass resultierenden Abstrakten Syntaxbaum. Alle Anweisungen und Ausdrücke sind in A-Normalform. 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
     Name './example_faculty_it.picoc_mon',
       Block
 5
         Name 'faculty.6',
 6
           // 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
         Name 'condition_check.5',
14
15
16
           // IfElse(Num('1'), [], [])
17
           Exp(Num('1')),
18
           IfElse
19
             Stack
20
               Num '1',
21
             Ε
22
               GoTo(Name('while_branch.4'))
23
             ],
24
             Ε
25
               GoTo(Name('while_after.1'))
26
27
         ],
28
       Block
         Name 'while_branch.4',
29
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
             Ε
40
               GoTo(Name('if.3'))
41
             ],
42
             Ε
43
               GoTo(Name('if_else_after.2'))
44
45
         ],
46
       Block
47
         Name 'if.3',
48
         [
49
           // Return(Name('res'))
50
           Exp(Stackframe(Num('1')))
51
           Return(Stack(Num('1')))
52
         ],
```

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

Code 2.10: Pico C-ANF Pass für Codebespiel.

2.3.1.4 RETI-Blocks Pass

2.3.1.4.1 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.

2.3.1.4.2 Abstrakte Grammatik

Die Abstrakte Grammatik 2.3.4 der Sprache L_{RETI_Blocks} ist verglichen mit der Abstrakten Grammatik 2.3.3 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 2.3.8 aufgelöst wird.

```
ACC() \mid IN1() \mid IN2() \mid PC() \mid SP()
                                                                                   BAF()
                                                                                                                                      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
                 Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi()
                 Or() \mid Ori() \mid And() \mid Andi()
                 Load() \mid Loadin() \mid Loadi() \mid Store() \mid Storein() \mid Move()
                 Instr(\langle op \rangle, \langle arg \rangle +) \mid Jump(\langle rel \rangle, Num(\langle str \rangle)) \mid Int(Num(\langle str \rangle))
instr
                 RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)
                 SingleLineComment(\langle str \rangle, \langle str \rangle)
                 Instr(Loadi(), [Reg(Acc()), GoTo(Name(\langle str \rangle))]) \mid Jump(Eq(), GoTo(Name(\langle str \rangle)))
instr
          ::=
                 Exp(GoTo(\langle str \rangle))
                                                                                                                                      L_{-}PicoC
block
                 Block(Name(\langle str \rangle), \langle instr \rangle *)
                 File(Name(\langle str \rangle), \langle block \rangle *)
file
```

Grammatik 2.3.4: Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF

2.3.1.4.3 Codebeispiel

In Code 2.11 sieht man den Abstrakten Syntaxbaum des RETI-Blocks Passes für das aus Unterkapitel 2.6 weitergeführte Beispiel, indem die Anweisungen, die durch entsprechende PicoC-Knoten im Abstrakten Syntaxbaum der Sprache $L_{PicoC_ANF}^{25}$ repräsentiert waren nun durch ihre entsprechennden RETI-Knoten ersetzt werden.

```
Name './example_faculty_it.reti_blocks',
     Ε
 4
5
6
7
8
       Block
         Name 'faculty.6',
           # // 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',
22
23
           # // IfElse(Num('1'), [], [])
24
           # Exp(Num('1'))
           SUBI SP 1;
```

²⁵Beschrieben durch die Grammatik 2.3.3.

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

```
# Exp(Stackframe(Num('1')))
84
           SUBI SP 1;
85
           LOADIN BAF ACC -3;
86
           STOREIN SP ACC 1;
87
           # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88
           LOADIN SP ACC 2:
89
           LOADIN SP IN2 1;
90
           MULT ACC IN2;
91
           STOREIN SP ACC 2;
92
           ADDI SP 1;
93
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
94
           LOADIN SP ACC 1;
95
           STOREIN BAF ACC -3;
           ADDI SP 1;
96
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
97
98
           # Exp(Stackframe(Num('0')))
99
           SUBI SP 1;
100
           LOADIN BAF ACC -2;
101
           STOREIN SP ACC 1;
102
           # Exp(Num('1'))
103
           SUBI SP 1;
104
           LOADI ACC 1;
105
           STOREIN SP ACC 1;
106
           # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107
           LOADIN SP ACC 2;
108
           LOADIN SP IN2 1:
109
           SUB ACC IN2;
110
           STOREIN SP ACC 2;
111
           ADDI SP 1;
112
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
113
           LOADIN SP ACC 1;
L14
           STOREIN BAF ACC -2:
115
           ADDI SP 1;
116
           # Exp(GoTo(Name('condition_check.5')))
117
           Exp(GoTo(Name('condition_check.5')))
118
         ],
119
       Block
120
         Name 'while_after.1',
<sup>21</sup>
122
           # Return(Empty())
123
           LOADIN BAF PC -1;
124
         ],
125
       Block
126
         Name 'main.0',
L27
128
           # StackMalloc(Num('2'))
129
           SUBI SP 2;
130
           # Exp(Num('4'))
L31
           SUBI SP 1;
132
           LOADI ACC 4;
133
           STOREIN SP ACC 1;
134
           # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
135
           MOVE BAF ACC;
L36
           ADDI SP 3;
L37
           MOVE SP BAF;
           SUBI SP 4;
           STOREIN BAF ACC 0;
```

```
LOADI ACC GoTo(Name('addr@next_instr'));
            ADD ACC CS;
            STOREIN BAF ACC -1;
            # Exp(GoTo(Name('faculty.6')))
            Exp(GoTo(Name('faculty.6')))
45
            # RemoveStackframe()
L46
            MOVE BAF IN1;
L<mark>4</mark>7
            LOADIN IN1 BAF 0;
L48
            MOVE IN1 SP;
149
            # Exp(ACC)
150
            SUBI SP 1;
151
            STOREIN SP ACC 1;
152
            LOADIN SP ACC 1;
153
            ADDI SP 1;
154
            CALL PRINT ACC;
155
            # Return(Empty())
156
            LOADIN BAF PC -1;
L57
          ]
L58
     ]
```

Code 2.11: RETI-Blocks Pass für Codebespiel.

Anmerkung Q

Wenn der Abstrakte Syntaxbaum ausgegeben wird, ist die Darstellung nicht auschließlich in Abstrakter Syntax, da die RETI-Knoten aus bereits im Unterkapitel 2.2.5.6 vermitteltem Grund in Konkreter Syntax ausgeben werden.

2.3.1.5 RETI-Patch Pass

2.3.1.5.1 Aufgabe

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

- das Einfügen eines start.<nummer>-Blockes, welcher ein GoTo(Name('main')) zur main-Funktion enthält, wenn in manchen Fällen die main-Funktion nicht die erste Funktion ist und daher am Anfang zur main-Funktion gesprungen werden muss.
- das Entfernen von GoTo()'s, deren Sprung nur eine Adresse weiterspringen würde.
- das Voranstellen von RETI-Knoten, die vor jeder Division Instr(Div(), args) prüfen, ob, nicht durch 0 geteilt wird.²⁶
- das Überprüfen darauf, ob bestimmte Immediates Im(str) in Befehlen, wie z.B. Jump(rel, Im(str)), Instr(Loadin(), [reg, reg, Im(str)]), Instr(Loadi(), [reg, Im(str)]) usw. kleiner -2²¹ oder größer 2²¹ 1 sind. Im Fall dessen, dass es so ist, muss der gewünschte Zahlenwert durch Bitshiften und Anwenden von bitweise Oder berechnet werden. Im Fall, dessen, dass der Immediate allerdings kleiner -(2³¹) oder größer 2³¹ 1 ist, wird eine Fehlermeldung TooLargeLiteral ausgegeben.

2.3.1.5.2 Abstrakte Grammatik

 $[\]overline{^{26}\mathrm{Das}}$ fällt unter die Themenbereiche des Bachelorprojekts und wird daher nicht genauer erläutert.

Die Abstrakte Grammatik 2.3.5 der Sprache L_{RETI_Patch} ist im Vergleich zur Abstrakten Grammatik 2.3.4 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.

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

Grammatik 2.3.5: Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF

2.3.1.5.3 Codebeispiel

In Code 2.12 sieht man den Abstrakten Syntaxbaum des PiocC-Patch Passes für das aus Unterkapitel 2.6 weitergeführte Beispiel. Durch den RETI-Patch Pass wurde hier ein start. <nummer>-Block²⁷ eingesetzt, da die main-Funktion nicht die erste Funktion ist. Des Weiteren wurden durch diesen Pass einzelne GoTo(Name(str))-Anweisungen entfernt²⁸, die nur einen Sprung um eine Position entsprochen hätten.

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

 $^{^{27}\}mathrm{Dieser}$ Block wurde im Code 2.8 markiert.

²⁸Diese entfernten GoTo(Name(str))'s' wurden ebenfalls im Code 2.8 markiert.

```
ADDI SP 1;
           # // While(Num('1'), [])
23
           # Exp(GoTo(Name('condition_check.5')))
           # // not included Exp(GoTo(Name('condition_check.5')))
25
         ],
26
       Block
27
         Name 'condition_check.5',
28
29
           # // IfElse(Num('1'), [], [])
30
           # Exp(Num('1'))
31
           SUBI SP 1;
32
           LOADI ACC 1;
           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;
           LOADIN BAF ACC -2;
47
48
           STOREIN SP ACC 1;
49
           # Exp(Num('1'))
50
           SUBI SP 1;
51
           LOADI ACC 1;
52
           STOREIN SP ACC 1;
53
           LOADIN SP ACC 2;
54
           LOADIN SP IN2 1;
55
           SUB ACC IN2;
56
           JUMP== 3;
57
           LOADI ACC 0;
58
           JUMP 2;
59
           LOADI ACC 1;
60
           STOREIN SP ACC 2;
61
           ADDI SP 1;
62
           # IfElse(Stack(Num('1')), [], [])
63
           LOADIN SP ACC 1;
64
           ADDI SP 1;
65
           JUMP== GoTo(Name('if_else_after.2'));
66
           # // not included Exp(GoTo(Name('if.3')))
67
         ],
68
       Block
69
         Name 'if.3',
70
           # // Return(Name('res'))
72
           # Exp(Stackframe(Num('1')))
73
           SUBI SP 1;
           LOADIN BAF ACC -3;
74
           STOREIN SP ACC 1;
           # Return(Stack(Num('1')))
           LOADIN SP ACC 1;
```

```
78
           ADDI SP 1;
79
           LOADIN BAF PC -1;
80
         ],
81
       Block
82
         Name 'if_else_after.2',
83
           # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
84
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;
           # Assign(Stackframe(Num('1')), Stack(Num('1')))
99
100
           LOADIN SP ACC 1;
01
           STOREIN BAF ACC -3;
102
           ADDI SP 1;
103
           # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104
           # Exp(Stackframe(Num('0')))
105
           SUBI SP 1;
           LOADIN BAF ACC -2;
106
107
           STOREIN SP ACC 1;
108
           # Exp(Num('1'))
109
           SUBI SP 1;
110
           LOADI ACC 1;
111
           STOREIN SP ACC 1;
           # 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;
           ADDI SP 1;
117
           # Assign(Stackframe(Num('0')), Stack(Num('1')))
118
119
           LOADIN SP ACC 1;
120
           STOREIN BAF ACC -2;
121
           ADDI SP 1;
122
           # Exp(GoTo(Name('condition_check.5')))
           Exp(GoTo(Name('condition_check.5')))
123
124
         ],
L25
       Block
126
         Name 'while_after.1',
L27
L28
           # Return(Empty())
L29
           LOADIN BAF PC -1;
130
         ],
l31
       Block
132
         Name 'main.0',
133
           # StackMalloc(Num('2'))
```

```
SUBI SP 2;
.36
           # Exp(Num('4'))
.37
           SUBI SP 1;
           LOADI ACC 4;
           STOREIN SP ACC 1;
40
           # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
41
           MOVE BAF ACC;
42
           ADDI SP 3;
143
           MOVE SP BAF;
44
           SUBI SP 4;
45
           STOREIN BAF ACC 0;
46
           LOADI ACC GoTo(Name('addr@next_instr'));
47
           ADD ACC CS;
48
           STOREIN BAF ACC -1;
49
           # Exp(GoTo(Name('faculty.6')))
150
           Exp(GoTo(Name('faculty.6')))
151
           # RemoveStackframe()
152
           MOVE BAF IN1;
153
           LOADIN IN1 BAF 0;
154
           MOVE IN1 SP;
155
           # Exp(ACC)
156
           SUBI SP 1;
L57
           STOREIN SP ACC 1;
158
           LOADIN SP ACC 1;
.59
            ADDI SP 1;
160
           CALL PRINT ACC;
161
           # Return(Empty())
162
           LOADIN BAF PC -1;
163
         ]
164
     ]
```

Code 2.12: RETI-Patch Pass für Codebespiel.

2.3.1.6 RETI Pass

2.3.1.6.1 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.

2.3.1.6.2 Konkrete und Abstrakte Grammatik

Die Abstrakte Grammatik 2.3.8 der Sprache L_{RETI} hat im Vergleich zur Abstrakten Grammatik 2.3.5 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 2.3.6 und 2.3.7 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.

```
"1"
                          "2"
dig\_no\_0
                                                  "5"
                                                                      L_Program
            ::=
                 "7"
                         "8"
                                 "g"
                         dig\_no\_0
dig_-with_-0
                 "0"
            ::=
                 "0"
num
                         dig\_no\_0 dig\_with\_0* | "-"dig\_no\_0*
letter
                 "a"..."Z"
            ::=
                 letter(letter \mid dig\_with\_0 \mid \_)*
name
            ::=
                             "IN1" | "IN2" | "PC" |
                 "ACC"
reg
            ::=
                             "CS" | "DS"
                 "BAF"
arg
                 reg
                      ::=
                        num
                 "=="
                            "!=" | "<" | "<=" | ">"
rel
            ::=
                  ">="
                            "\_NOP"
```

Grammatik 2.3.6: Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF

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

Grammatik 2.3.7: Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF

```
::=
                         ACC() \mid IN1()
                                                           IN2()
                                                                           PC()
                                                                                           SP()
                                                                                                          BAF()
                                                                                                                                                              L\_RETI
reg
                         CS() \mid DS()
                         Reg(\langle reg \rangle) \mid Num(\langle str \rangle)
arq
                         Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()
rel
                         Always() \mid NOp()
                         Add()
                                        Addi()
                                                        Sub() \mid Subi() \mid Mult() \mid Multi()
op
                                       Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi()
                         Div()
                         Or() \mid Ori() \mid And() \mid Andi()
                         Load() \hspace{0.1in} | \hspace{0.1in} Loadin() \hspace{0.1in} | \hspace{0.1in} Loadi() \hspace{0.1in} | \hspace{0.1in} Store() \hspace{0.1in} | \hspace{0.1in} Storein() \hspace{0.1in} | \hspace{0.1in} Move()
                         Instr(\langle op \rangle, \langle arg \rangle +) \quad | \quad Jump(\langle rel \rangle, Num(\langle str \rangle)) \quad | \quad Int(Num(\langle str \rangle))
instr
                         RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)
                         SingleLineComment(\langle str \rangle, \langle str \rangle)
                         Instr(Loadi(), [Reg(Acc()), GoTo(Name(\langle str \rangle))]) \mid Jump(Eq(), GoTo(Name(\langle str \rangle)))
                         Program(Name(\langle str \rangle), \langle instr \rangle *)
program
instr
                         Exp(GoTo(\langle str \rangle)) \mid Exit(Num(\langle str \rangle))
                                                                                                                                                              L\_PicoC
                ::=
                         Block(Name(\langle str \rangle), \langle instr \rangle *)
block
                ::=
file
                         File(Name(\langle str \rangle), \langle block \rangle *)
                ::=
```

Grammatik 2.3.8: Abstrakte Grammatik der Sprache L_{RETI} in ASF

2.3.1.6.3 Codebeispiel

Nach dem RETI-Pass ist das Programm komplett in RETI-Knoten übersetzt, die allerdings in ihrer Konkreten Syntax ausgegeben werden, wie in Code 2.13 zu sehen ist. Es gibt keine Blöcke mehr und die RETI-Befehle in diesen Blöcken wurden zusammengesetzt, wie sie in den Blöcken angeordnet waren. Die letzten Nicht-RETI-Befehle oder RETI-Befehle, die nicht auschließlich aus RETI-Ausdrücken bestehen²⁹, die sich in den Blöcken befunden haben, wurden durch RETI-Befehle ersetzt.

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

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

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

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

```
00 # StackMalloc(Num('2'))
01 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
10 SUBI SP 4;
11 STOREIN BAF ACC 0;
112 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;
120 MOVE IN1 SP;
21 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
26 CALL PRINT ACC;
27 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 2.13: RETI Pass für Codebespiel.

Literatur

Online

- A-Normalization: Why and How (with code). URL: https://matt.might.net/articles/a-normalization/(besucht am 23.07.2022).
- ANSI C grammar (Lex). URL: https://www.quut.com/c/ANSI-C-grammar-1-2011.html (besucht am 15.08.2022).
- ANSI C grammar (Lex) old. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (besucht am 15.08.2022).
- ANSI C grammar (Yacc). URL: https://www.quut.com/c/ANSI-C-grammar-y.html (besucht am 15.08.2022).
- ANSI C grammar (Yacc) old. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html (besucht am 15.08.2022).
- ANTLR. URL: https://www.antlr.org/ (besucht am 31.07.2022).
- Bäume. URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17.07.2022).
- C Operator Precedence cppreference.com. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27.04.2022).
- clang: C++ Compiler. URL: http://clang.org/ (besucht am 29.07.2022).
- $Clockwise/Spiral\ Rule$. URL: https://c-faq.com/decl/spiral.anderson.html (besucht am 29.07.2022).
- Errors in C/C++ GeeksforGeeks. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10.05.2022).
- GCC, the GNU Compiler Collection GNU Project. URL: https://gcc.gnu.org/ (besucht am 13.07.2022).
- Grammar Reference Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/grammar.html (besucht am 31.07.2022).
- Grammar: The language of languages (BNF, EBNF, ABNF and more). URL: https://matt.might.net/articles/grammars-bnf-ebnf/ (besucht am 30.07.2022).
- JSON parser Tutorial Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09.07.2022).
- Ljohhuh. What is an immediate value? 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13.04.2022).

- Transformers & Visitors Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09.07.2022).
- Welcome to Lark's documentation! Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/ (besucht am 31.07.2022).
- What is Bottom-up Parsing? URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22.06.2022).
- What is Top-Down Parsing? URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22.06.2022).

Bücher

- G. Siek, Jeremy. Course Webpage for Compilers (P423, P523, E313, and E513). 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28.01.2022).
- Nystrom, Robert. Parsing Expressions · Crafting Interpreters. Genever Benning, 2021. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09.07.2022).

Artikel

- Earley, J. und Howard E. Sturgis. "A formalism for translator interactions". In: *CACM* (1970). DOI: 10.1145/355598.362740.
- Earley, Jay. "An efficient context-free parsing". In: 13 (1968). URL: https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf (besucht am 10.08.2022).

Vorlesungen

- 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).
- Thiemann, Peter. "Compilerbau". Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09.07.2022).
- — "Einführung in die Programmierung". Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09.07.2022).
- Westphal, Dr. Bernd. "Softwaretechnik". Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19.07.2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. "Types are calling conventions". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23.07.2022).
- 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).