
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	V
Grammatikverzeichnis	VI
1 Motivation	1
1.1 RETI	1
1.2 PicoC	1
1.3 Aufgabenstellung	1
1.4 Eigenheiten der Sprache C	1
1.5 Richtlinien	2
2 Einführung	3
2.1 Compiler und Interpreter	3
2.1.1 T-Diagramme	6
2.2 Formale Sprachen	8
2.2.1 Mehrdeutige Grammatiken	10
2.2.2 Präzidenz und Assoziativität	11
2.3 Lexikalische Analyse	12
2.4 Syntaktische Analyse	15
2.5 Code Generierung	20
2.5.1 Monadische Normalform	21
2.5.2 A-Normalform	22
2.5.3 Ausgabe des Maschinenencodes	24
2.6 Fehlermeldungen	25
2.6.1 Kategorien von Fehlermeldungen	25
3 Implementierung	26
3.1 Lexikalische Analyse	28
3.1.1 Konkrete Syntax für die Lexikalische Analyse	28
3.1.2 Codebeispiel	30
3.2 Syntaktische Analyse	30
3.2.1 Umsetzung von Präzidenz und Assoziativität	30
3.2.2 Konkrete Syntax für die Syntaktische Analyse	35
3.2.3 Derivation Tree Generierung	37
3.2.3.1 Codebeispiel	37
3.2.3.2 Ausgabe des Derivation Tree	38
3.2.4 Derivation Tree Vereinfachung	39
3.2.4.1 Codebeispiel	40
3.2.5 Abstrakt Syntax Tree Generierung	41
3.2.5.1 PicoC-Knoten	43
3.2.5.2 RETI-Knoten	48

3.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	49
3.2.5.4	Abstrakte Syntax	51
3.2.5.5	Codebeispiel	52
3.2.5.6	Ausgabe des Abstract Syntax Tree	53
3.3	Code Generierung	53
3.3.1	Passes	55
3.3.1.1	PicoC-Shrink Pass	56
3.3.1.1.1	Aufgabe	56
3.3.1.1.2	Abstrakte Syntax	56
3.3.1.1.3	Codebeispiel	57
3.3.1.2	PicoC-Blocks Pass	59
3.3.1.2.1	Aufgabe	59
3.3.1.2.2	Abstrakte Syntax	59
3.3.1.2.3	Codebeispiel	61
3.3.1.3	PicoC-ANF Pass	62
3.3.1.3.1	Aufgabe	62
3.3.1.3.2	Abstrakte Syntax	63
3.3.1.3.3	Codebeispiel	65
3.3.1.4	RETI-Blocks Pass	66
3.3.1.4.1	Aufgabe	66
3.3.1.4.2	Abstrakte Syntax	66
3.3.1.4.3	Codebeispiel	67
3.3.1.5	RETI-Patch Pass	70
3.3.1.5.1	Aufgabe	70
3.3.1.5.2	Abstrakte Syntax	70
3.3.1.5.3	Codebeispiel	71
3.3.1.6	RETI Pass	74
3.3.1.6.1	Aufgabe	74
3.3.1.6.2	Konkrete und Abstrakte Syntax	74
3.3.1.6.3	Codebeispiel	76

Literatur

A

Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen	8
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen	8
2.3	Veranschaulichung von Linksassoziativität und Rechtsassoziativität	11
2.4	Veranschaulichung von Präzidenz	11
2.5	Veranschaulichung der Lexikalischen Analyse	15
2.6	Veranschaulichung der Syntaktischen Analyse	19
2.7	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten	22
2.8	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen	24
3.1	Ableitungsbäume zu den beiden Ableitungen	32
3.2	Derivation Tree nach Parsen eines Ausdrucks	39
3.3	Derivation Tree nach Vereinfachung	40
3.4	Abstract Syntax Tree Generierung ohne Umdrehen	42
3.5	Abstract Syntax Tree Generierung mit Umdrehen	42
3.6	Cross-Compiler Kompilervorgang ausgeschrieben	54
3.7	Cross-Compiler Kompilervorgang Kurzform	54
3.8	Architektur mit allen Passes ausgeschrieben	55

Codeverzeichnis

3.1	PicoC-Code des Codebeispiels	30
3.2	Tokens für das Codebeispiel	30
3.3	Derivation Tree nach Derivation Tree Generierung	38
3.4	Derivation Tree nach Derivation Tree Vereinfachung	41
3.5	Aus vereinfachtem Derivation Tree generierter Abstract Syntax Tree	52
3.6	PicoC Code für Codebeispiel	58
3.7	Abstract Syntax Tree für Codebeispiel	59
3.8	PicoC-Blocks Pass für Codebeispiel	62
3.9	PicoC-ANF Pass für Codebeispiel	66
3.10	RETI-Blocks Pass für Codebeispiel	70
3.11	RETI-Patch Pass für Codebeispiel	74
3.12	RETI Pass für Codebeispiel	78

Tabellenverzeichnis

3.1	Präzidenzregeln von PicoC	31
3.2	Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren	33
3.3	PicoC-Knoten Teil 1	44
3.4	PicoC-Knoten Teil 2	45
3.5	PicoC-Knoten Teil 3	46
3.6	PicoC-Knoten Teil 4	47
3.7	RETI-Knoten	49
3.8	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	50

Definitionsverzeichnis

1.1	Caller-save Register	1
1.2	Callee-save Register	1
1.3	Deklaration	1
1.4	Definition	1
1.5	Allokation	1
1.6	Initialisierung	2
1.7	Scope	2
1.8	Call by value	2
1.9	Call by reference	2
2.1	Interpreter	3
2.2	Compiler	3
2.3	Maschinensprache	4
2.4	Assemblersprache (bzw. engl. Assembly Language)	4
2.5	Assembler	5
2.6	Objectcode	5
2.7	Linker	5
2.8	Immediate	5
2.9	Transpiler (bzw. Source-to-source Compiler)	6
2.10	Cross-Compiler	6
2.11	T-Diagram Programm	6
2.12	T-Diagram Übersetzer (bzw. eng. Translator)	7
2.13	T-Diagram Interpreter	7
2.14	T-Diagram Maschine	7
2.15	Wort	9
2.16	Formale Sprache	9
2.17	Chromsky Hierarchie	9
2.18	Syntax	9
2.19	Semantik	9
2.20	Grammatik	9
2.21	Reguläre Sprachen	9
2.22	Kontextfreie Sprachen	9
2.23	Ableitung	9
2.24	Links- und Rechtsableitung	10
2.25	Linksrekursive Grammatiken	10
2.26	Ableitungsbaum	10
2.27	Mehrdeutige Grammatik	10
2.28	Wortproblem	10
2.29	LL(k)-Grammatik	10
2.30	Assoziativität	11
2.31	Präzidenz	11
2.32	Pipe-Filter Architekturpattern	12
2.33	Pattern	12
2.34	Lexeme	12
2.35	Lexer (bzw. Scanner oder auch Tokenizer)	13
2.36	Bezeichner (bzw. Identifier)	13
2.37	Literal	14
2.38	Konkrete Syntax	15

2.39	Derivation Tree (bzw. Parse Tree)	15
2.40	Parser	16
2.41	Recognizer (bzw. Erkennen)	16
2.42	Transformer	18
2.43	Visitor	18
2.44	Abstrakte Syntax	18
2.45	Abstract Syntax Tree (AST)	18
2.46	Pass	20
2.47	Reiner Ausdruck (bzw. engl. pure expression)	21
2.48	Unreiner Ausdruck	21
2.49	Monadische Normalform (bzw. engl. monadic normal form)	21
2.50	Location	22
2.51	Atomarer Ausdruck	23
2.52	Komplexer Ausdruck	23
2.53	A-Normalform (ANF)	23
2.54	Fehlermeldung	25
3.1	Metasyntax	26
3.2	Metasprache	26
3.3	Erweiterte Backus-Naur-Form (EBNF)	26
3.4	Dialekt der EBNF aus Lark	27
3.5	Abstrakte Syntax Form (ASF)	27
3.6	Earley Parser	37
3.7	Label	48
3.8	Token-Knoten	48
3.9	Container-Knoten	48
3.10	Symboltabelle	63

Grammatikverzeichnis

3.1.1 Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF	29
3.2.1 Undurchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF	31
3.2.2 Durchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF . .	32
3.2.3 Beispiel für eine unäre rechtsassoziative Produktion	33
3.2.4 Beispiel für eine unäre linksassoziative Produktion	33
3.2.5 Beispiel für eine linksassoziative Produktion	34
3.2.6 Beispiel für eine linksassoziative Produktion	34
3.2.7 Durchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF . .	35
3.2.8 Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1	36
3.2.9 Grammatik der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	37
3.2.10 Abstrakte Syntax der Sprache L_{PioC}	51
3.3.1 Abstrakte Syntax der Sprache L_{PioC_Shrink}	57
3.3.2 Abstrakte Syntax der Sprache L_{PioC_Blocks}	60
3.3.3 Abstrakte Syntax der Sprache L_{PioC_ANF}	64
3.3.4 Abstrakte Syntax der Sprache L_{RETI_Blocks}	67
3.3.5 Abstrakte Syntax der Sprache L_{RETI_Patch}	71
3.3.6 Konkrete Syntax der Sprache L_{RETI} für die Lexikalische Analyse in EBNF	75
3.3.7 Konkrete Syntax der Sprache L_{RETI} für die Syntaktische Analyse in EBNF	75
3.3.8 Abstrakte Syntax der Sprache L_{RETI}	75

1 Motivation

1.1 RETI

... basiert auf ... der Vorlesung C. Scholl, „Betriebssysteme“.

Definition 1.1: Caller-save Register

a

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

Definition 1.2: Callee-save Register

a

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

1.2 PicoC

1.3 Aufgabenstellung

1.4 Eigenheiten der Sprache C

Definition 1.3: Deklaration

a

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

Definition 1.4: Definition

a

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

Definition 1.5: Allokation

a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.6: Initialisierung*a*^aThiemann, „Einführung in die Programmierung“.**Definition 1.7: Scope***a*^aThiemann, „Einführung in die Programmierung“.**Definition 1.8: Call by value***a*^aBast, „Programmieren in C“.**Definition 1.9: Call by reference***a*^aBast, „Programmieren in C“.

1.5 Richtlinien

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

Beim **PicoC-Compiler** wurde eher darauf Wert gelegt **sauberen, strukturierten Code** zu schreiben, den die Studenten sogar selber verstehen könnten und eine **unkomplizierte Bibliothek mit guter Dokumentation**¹, nämlich das **Lark Parsing Toolkit**² für das **Parsen** zu verwenden. Vor allem, da zu erwarten ist, dass der **PicoC-Compiler** vielleicht in einigen anderen Projekten eingebunden werden könnte, ist es von **Vorteil** bei der Notwendigkeit kleiner **Erweiterungen**, diese Erweiterungen **unkompliziert** durchführen zu können.

¹[Welcome to Lark's documentation! — Lark documentation.](#)

²[Lark - a parsing toolkit for Python.](#)

2 Einführung

2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** L_{PicoC} in die **RETI-Sprache** L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**¹ und von **Tests** die **Beziehungen** in 2.46.1 zu belegen (siehe Subkapitel ??).

Definition 2.1: Interpreter

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

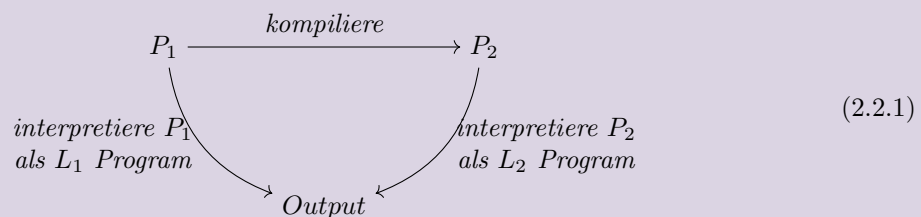
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstract Syntax Tree** (Definition 2.45) und führt je nach Komposition der **Nodes** des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.^a*

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

Definition 2.2: Compiler

***Kompiliert** ein 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 das Program P_1 so in das Program P_2 übersetzt wird, dass bei beiden Programmen, wenn sie von **Interpretern** ihrer jeweiligen Sprachen L_1 und L_2 **interpretiert** werden, der gleiche **Output** rauskommt. Also beide Programme P_1 und P_2 die gleiche **Semantik** haben und sich nur **syntaktisch** durch die Sprachen L_1 und L_2 , in denen sie geschrieben stehen unterscheiden.^a*



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

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

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C_{i.w.k.min}^{o-j}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache L_{B_i} einer Maschine M_i kompiliert. Fall die Notwendigkeit besteht die **Maschine** M_i anzugeben, zu dessen **Maschinensprache** L_{B_i} der Compiler kompiliert, wird das als C_i geschrieben. Falls die Notwendigkeit besteht die **Sprache** L_o anzugeben, in der der Compiler selbst geschrieben ist, wird das als C^o geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert ($L_{w.k}$) oder in der er selbst geschrieben ist ($L_{o.j}$) anzugeben, wird das als $C_{w.k}^{o-j}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition ??) kann man das als C_{min} schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschinenenncode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

Definition 2.3: Maschinensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauer gesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen konstanten** Anzahl von Fetch- und Execute Phasen im **Komplexeren Fall**. Die Maschinenbefehle sind meist so designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.^{a,b}*

^aViele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.8) haben.

^bC. Scholl, „Betriebssysteme“.

Definition 2.4: Assemblersprache (bzw. engl. Assembly Language)

*Eine sehr **hardwarenahe** Programmiersprache, deren **Instructions** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen^a haben. Viele **Instructions** haben eine ähnliche übliche Struktur **Operation** <Operanden>, mit einer **Operation**, die einem **Opcode** eines Maschinenbefehls bezeichnet und keinen oder mehreren **Operanden**, wie die späteren Maschinenbefehle, denen sie entsprechen. Allerdings gibt es oftmals noch viel „syntaktischen Zucker“ innerhalb^b der Instructions und drumherum^{c,d}.*

^aInstructions der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

^bZ.B. erlaubt die Assemblersprache des **GCC** für die **X86_64-Architektur** für manche Operanden die Syntax **n(%r)**, die einen **Speicherzugriff** mit **Offset** n zur Adresse, die im **Register** **%r** steht durchführt, wobei z.B. die Klammern () usw. nur „syntaktischer Zucker“ sind und natürlich nicht mitcodiert werden.

^cZ.B. sind im X86_64 Assembler die Instructions in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels **jmp <label>** gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.

^dP. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenenncode** bzw. **Objectcode** (Definition 2.6) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschinenenncode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 2.7) zu Maschiendencode zusammengesetzt wird ausführbar

ist.

Definition 2.5: Assembler

Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärer Repräsentation**, der in **Maschiensprache** geschrieben ist.^a

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

Definition 2.6: Objectcode

Bei Komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen wird häufig **Objectcode** erzeugt, der neben der Folge von Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.^a

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

Definition 2.7: Linker

Programm, das **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt**, sodass unter anderem kein vermeidbarer **doppelter Code** darin vorkommt.^a

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

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

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenem RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung².

Definition 2.8: Immediate

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

^aLjohhuh, *What is an immediate value?*

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

Definition 2.9: Transpiler (bzw. Source-to-source Compiler)

Kompiliert zwischen Sprachen, die ungefähr auf dem *gleichen* Level an *Abstraktion* arbeiten^{ab}

^aDie Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.

^bThiemann, „Compilerbau“.

Definition 2.10: 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 **Maschinen Sprachen** B_1 und B_2 haben.^{ab}

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

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

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

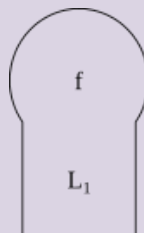
2.1.1 T-Diagramme

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

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Programm (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschine (Definition 2.14) zusammen.

Definition 2.11: T-Diagram Programm

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



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

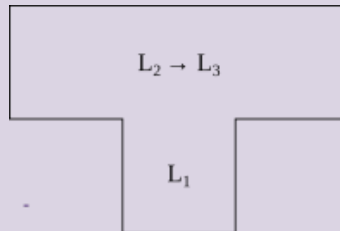
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

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

Definition 2.12: T-Diagramm Übersetzer (bzw. eng. Translator)

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

Für den **Übersetzer** gelten genauso, wie für einen **Compiler**^a die **Beziehungen** in 2.46.1.^b

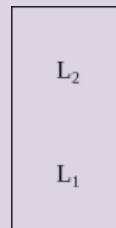


^aZwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

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

Definition 2.13: T-Diagramm Interpreter

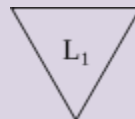
Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.^a



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

Definition 2.14: T-Diagramm Maschine

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** L_1 ausführt.^{a,b}



^aWenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).

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

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

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

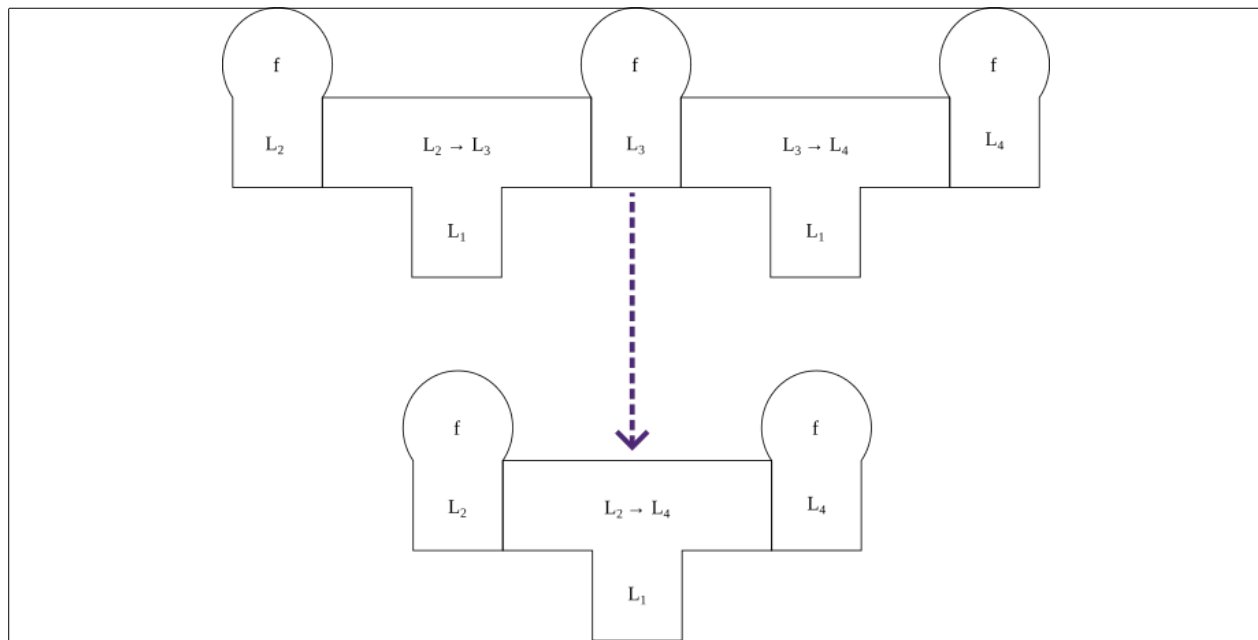


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

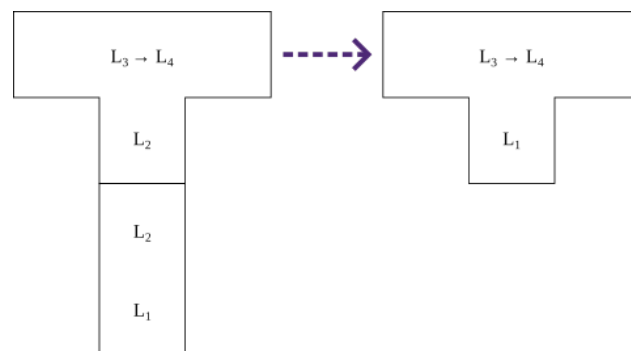


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit **Formalen Sprachen** zu tun, da bereits das Kompilieren an sich, das Übersetzen eines Programmes aus der Sprache L_1 in eine Sprache L_2 (Definition 2.16) ist, wobei die **Semantik** (Definition 2.19) **gleich** bleiben muss. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 2.20), welche diese beschreibt und können in verschiedenen Syntaxen (Definition 2.18)⁴ dargestellt werden.

Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen** vorher eingeführt zu haben, was die den Begriff eines **Wortes** (Definition 2.15), die **Chomsky Hierarchie** (Definition 2.17), **Reguläre Sprachen** (Definition 2.15), **Kontextfreie Sprachen** (Definition 2.15), den Begriff der **Ableitung** (Definition 2.15), das **Wortproblem** (Definition 2.15) usw. einschließt.

⁴Damit sind die später Erwähnung findenden **Konkreten** und **Abstrakten Syntaxen** gemeint.

Definition 2.15: Wort

„Ein Wort *text* ist **endliche Folge** von **Symbolen** aus einem **Alphabet** Σ “^a

^aNebel, „Theoretische Informatik“.

Definition 2.16: Formale Sprache

„Eine **Formale Sprache** ist eine Menge von **Wörtern** über Σ “.^a

^aNebel, „Theoretische Informatik“.

Definition 2.17: Chomsky Hierarchie

^a

^aNebel, „Theoretische Informatik“.

Definition 2.18: Syntax

Der **Aufbau** eines Programmes, der Regelsystem, welches durch eine Grammatik oder Natürliche Sprache ausgedrückt werden kann darstellt.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 2.19: Semantik

Die **Bedeutung** eines Programmes.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 2.20: Grammatik

^a

^aNebel, „Theoretische Informatik“.

Definition 2.21: Reguläre Sprachen

^a

^aNebel, „Theoretische Informatik“.

Definition 2.22: Kontextfreie Sprachen

^a

^aNebel, „Theoretische Informatik“.

Der Kompiliervorgang lässt sich in viele verschiedene Phasen unterteilen

Definition 2.23: Ableitung

^a

^aNebel, „Theoretische Informatik“.

Definition 2.24: Links- und Rechtsableitung*a*^aNebel, „Theoretische Informatik“.**Definition 2.25: Linksrekursive Grammatiken**

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

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

$$A \Rightarrow^* Aa,$$

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

^a*Parsing Expressions · Crafting Interpreters.***2.2.1 Mehrdeutige Grammatiken****Definition 2.26: Ableitungsbaum***a*^aNebel, „Theoretische Informatik“.**Definition 2.27: Mehrdeutige Grammatik**

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

^aAlternativ, wenn es für *w* *mehrere* unterschiedliche *Linksableitungen* gibt.^bNebel, „Theoretische Informatik“.**Definition 2.28: Wortproblem***a*^aNebel, „Theoretische Informatik“.**Definition 2.29: LL(k)-Grammatik**

Eine *Grammatik* ist *LL(k)* für $k \in \mathbb{N}$, falls jeder *Ableitungsschritt* eindeutig durch die nächsten *k Symbole* des *Eingabeworts* bzw. in Bezug zu *Compilerbau Token* des *Inputstrings* zu bestimmen ist^a. Dabei steht *LL* für *left-to-right* und *leftmost-derivation*, da das *Eingabewort* von *links nach rechts* geparsed und immer *Linksableitungen* genommen werden müssen^b, damit die obige Bedingung mit den *nächsten k Symbolen* gilt.^c

^aDas wird auch als *Lookahead* von *k* bezeichnet.^bWobei sich das mit den *Linksableitungen* automatisch ergibt, wenn man das Eingabewort von *links-nach-rechts* parsed und jeder der nächsten *k Ableitungsschritte* eindeutig sein soll.^cNebel, „Theoretische Informatik“.

2.2.2 Präzidenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Grammatik** für eine **Konkrete Syntax** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definition 2.30) und **Präzidenz** (Definition 2.31) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel 3.2.1 genauer erklärt.

Definition 2.30: Assoziativität

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

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

^aParsing Expressions · Crafting Interpreters.

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

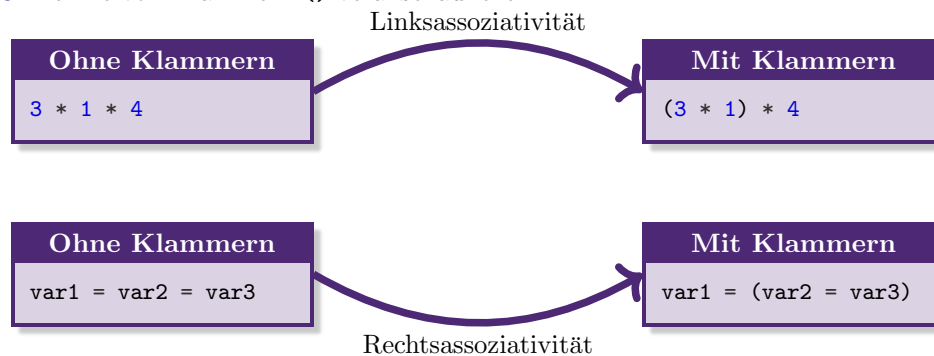


Abbildung 2.3: Veranschaulichung von Linksassoziativität und Rechtsassoziativität

Definition 2.31: Präzidenz

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

^aParsing Expressions · Crafting Interpreters.

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



Abbildung 2.4: Veranschaulichung von Präzidenz

2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.32) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.33) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 2.32: Pipe-Filter Architekturpattern

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

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

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



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

^bWestphal, „Softwaretechnik“.

Diese Folgen endlicher Symbole werden auch **Lexeme** (Definition 2.34) genannt.

Definition 2.33: Pattern

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

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

^bThiemann, „Compilerbau“.

Definition 2.34: Lexeme

Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} *matched*.^a

^aThiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.35) im **Inputstring** identifiziert und **Tokens** T zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** (Definition 2.35) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.35: Lexer (bzw. Scanner oder auch Tokenizer)

Ein **Lexer** ist eine **partielle Funktion** $lex : \Sigma^* \rightarrow (N \times W)^*$, welche ein **Wort** bzw. **Lexeme** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls dieses **Wort** sich unter der **regulären Grammatik** G_{Lex} , der **regulären Sprache** L_{Lex} ableiten lässt bzw. einem der **Pattern** der Sprache L_{Lex} entspricht.^a

^aThiemann, „Compilerbau“.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die kein **Pattern** eines **Tokens** der Sprache L_{Lex} matchen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** ausgegeben.

Um Verwirrung verzubäuen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

In der Syntaktischen Analyse sind die **Tokennamen** die Symbole.

In der Code Generierung sind die **Bezeichner** (Definition 2.36) von **Variablen, Konstanten und Funktionen** die Symbole^a.

^aDas ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel 3 **Symboltabelle** genannt wird.

Definition 2.36: Bezeichner (bzw. Identifier)

Tokenwert, der eine **Konstante, Variable, Funktion** usw. innerhalb ihres **Scopes eindeutig** benennt.^{a,b}

^aAußer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterscheidungsmerkmal hinzugenommen, damit es eindeutig ist.

^bThiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`⁵ und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** $*$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie 42, 314 oder 12. Die **Überbegriffe** bzw. **Tokennamen** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`⁶, bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie

⁵In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Programmiersprache, die man zur Implementierung des Lexers nutzt wegabstrahiert.

⁶Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Nodes haben will, damit unter anderem **mehr Code** in eine Zeile passt.

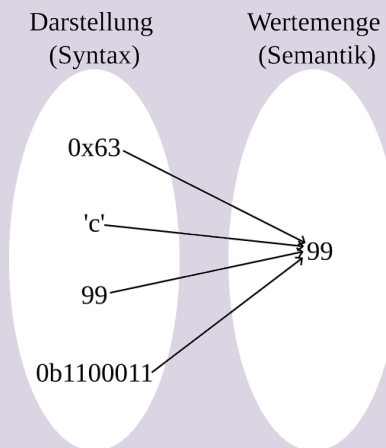
if oder } sind die **Tokennamen** bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich IF und RBRACE.

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.37) dargestellt werden, einmal als ASCII-Zeichen 'c', dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99⁷. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik** G_{Lex} , die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut⁸, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 3.1.1 liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.21 erfüllt. Einzig die Produktion $CHAR ::= "'ASCII_CHAR''$ sieht problematisch aus, kann allerdings auch als $\{CHAR ::= "'CHAR2, CHAR2 ::= ASCII_CHAR''\}$ **regulär** ausgedrückt werden⁹. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

Definition 2.37: Literal

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



^aThieman, „Einführung in die Programmierung“.

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

⁷Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen **0b1100011** und **0x63** darzustellen.

⁸Man nennt das auch einem **Lookahead** von 1

⁹Eine derartige Regel würde nur Probleme bereiten, wenn sich aus **ASCII_CHAR** **beliebig breite** Wörter ableiten lassen.

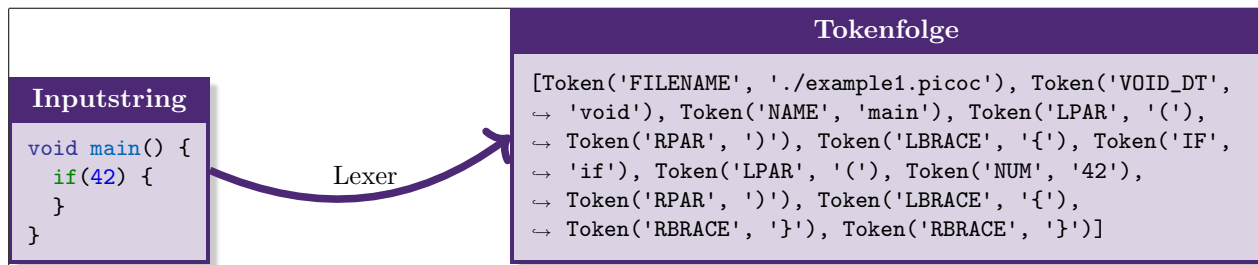


Abbildung 2.5: Veranschaulichung der Lexikalischen Analyse

2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `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.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.38) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.40), ein **Derivation Tree** (Definition 2.39) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 2.45) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

Definition 2.38: Konkrete Syntax

Syntax einer Sprache, die durch die Grammatiken G_{Lex} und G_{Parse} zusammengekommen beschrieben wird.

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

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

Definition 2.39: Derivation Tree (bzw. Parse Tree)

*Compilerinterne Darstellung eines in **Konkreter Syntax** geschriebenen Inputstrings als **Baumdatenstruktur**, in der **Nichtterminalsymbole** die **Inneren Knoten** der Baumdatenstruktur und **Terminalsymbole** die **Blätter** der Baumdatenstruktur bilden. Jedes zum Ableiten des Inputstrings verwendete **Nicht-Terminalsymbol** einer **Produktion** der **Grammatik** G_{Parse} , die ein Teil der **Konkrete Syntax** ist, bildet einen eigenen **Inneren Knoten**.*

*Der **Derivation Tree** wird optimalerweise immer so konstruiert bzw. die **Konkrete Syntax** immer so definiert, dass sich möglichst einfach ein **Abstract Syntax Tree** daraus konstruieren lässt.^a*

^aJSON parser - Tutorial — Lark documentation.

Definition 2.40: Parser

Ein **Parser** ist ein Programm, dass aus einem Inputstring, der in **Konkreter Syntax** geschrieben ist, eine compilerinterne Darstellung, den **Derivation Tree** generiert, was auch als **Parsen** bezeichnet wird^{a, b}.

^aEs gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.40 verwendet.

^b*JSON parser - Tutorial — Lark documentation.*

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

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines **Parsers**. Der **Lexer** ist ausschließlich für die **Lexikalische Analyse** verantwortlich und entspricht z.B., wenn man bei einem Wanderausflug verschiedenen Insekten entdeckt, dem Nachschlagen in einem 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 Insektenbegegnungen in einer für die **Weiterverarbeitung tauglichen Form**^b.

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

^aDas würde z.B. der Rolle eines **Semikolon** ; in der Sprache *LPicoC* entsprechen.

^bZ.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beeinflussen sich gegenseitig.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht. Dabei wird in der Grammatik L_{Parse} nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine **Zahl** steht und nicht, welchen konkreten Wert diese **Zahl** hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.41), denn ein Parser löst das **Wortproblem** (Definition 2.28) für die **Sprache**, die durch die **Konkrete Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Derivation Tree**.

Definition 2.41: Recognizer (bzw. Erkenner)

Entspricht dem Maschinenmodell eines **Automaten**. Im Bezug auf Compilerbau entspricht der **Recognizer** einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** erkennt, ob ein Inputstring bzw. **Wort** sich mit den Produktionen der **Konkrete Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Syntax** beschrieben wird oder nicht^{a, b}.

^aDas vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

^bThiemann, „Compilerbau“.

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

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

Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist, weil der **Eingabewert** bzw. der **Inputstring** 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**. Dabei wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminal** enthält.

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

Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.29) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewünschte Inputstring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass der **Inputstring** sich **nicht** mit der verwendeten Grammatik ableiten lässt.^b

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

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkreten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.^d
- **Chart Parser:** 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

^aWhat is Top-Down Parsing?

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

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

^dWhat is Bottom-up Parsing?

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

Der **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 2.42) und **Visitors** (Definition 2.43)

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 einen Inputstring von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 2.44).

Definition 2.42: Transformer

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

^aTransformers & Visitors — Lark documentation.

Definition 2.43: Visitor

Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** besucht und in Bezug zu Compilerbau, beim Antreffen eines bestimmten **Knoten** des **Derivation Tree**, diesen **in-place** mit anderen Knoten **tauscht** oder **manipuliert**, um den **Derivation Tree** für die weitere Verarbeitung durch z.B. einen **Transformer** zu vereinfachen.^{a,b}

^aKann theoretisch auch zur Konstruktion eines **Abstract Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstract Syntax Tree** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.

^bTransformers & Visitors — Lark documentation.

Definition 2.44: Abstrakte Syntax

Syntax, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstract Syntax Trees** möglich sind.

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.^a

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

Definition 2.45: Abstract Syntax Tree (AST)

Compilerinterne Darstellung eines Programs, in welcher sich anhand der Knoten auf dem **Pfad** von der **Wurzel** zu einem **Blatt** nicht mehr direkt nachvollziehen lässt, durch welche **Produktionen** dieses Blatt abgeleitet wurde.

Der **Abstract Syntax Tree** hat einmal den Zweck, dass die **Kompositionen**, die die Knoten bilden können **semantisch** näher an den **Instructions eines Assemblers** dran sind und, dass man mit einem **Abstract Syntax Tree** bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, möglichst schnell die Fragen beantworten kann, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.^a

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

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstract Syntax Tree** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.6 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

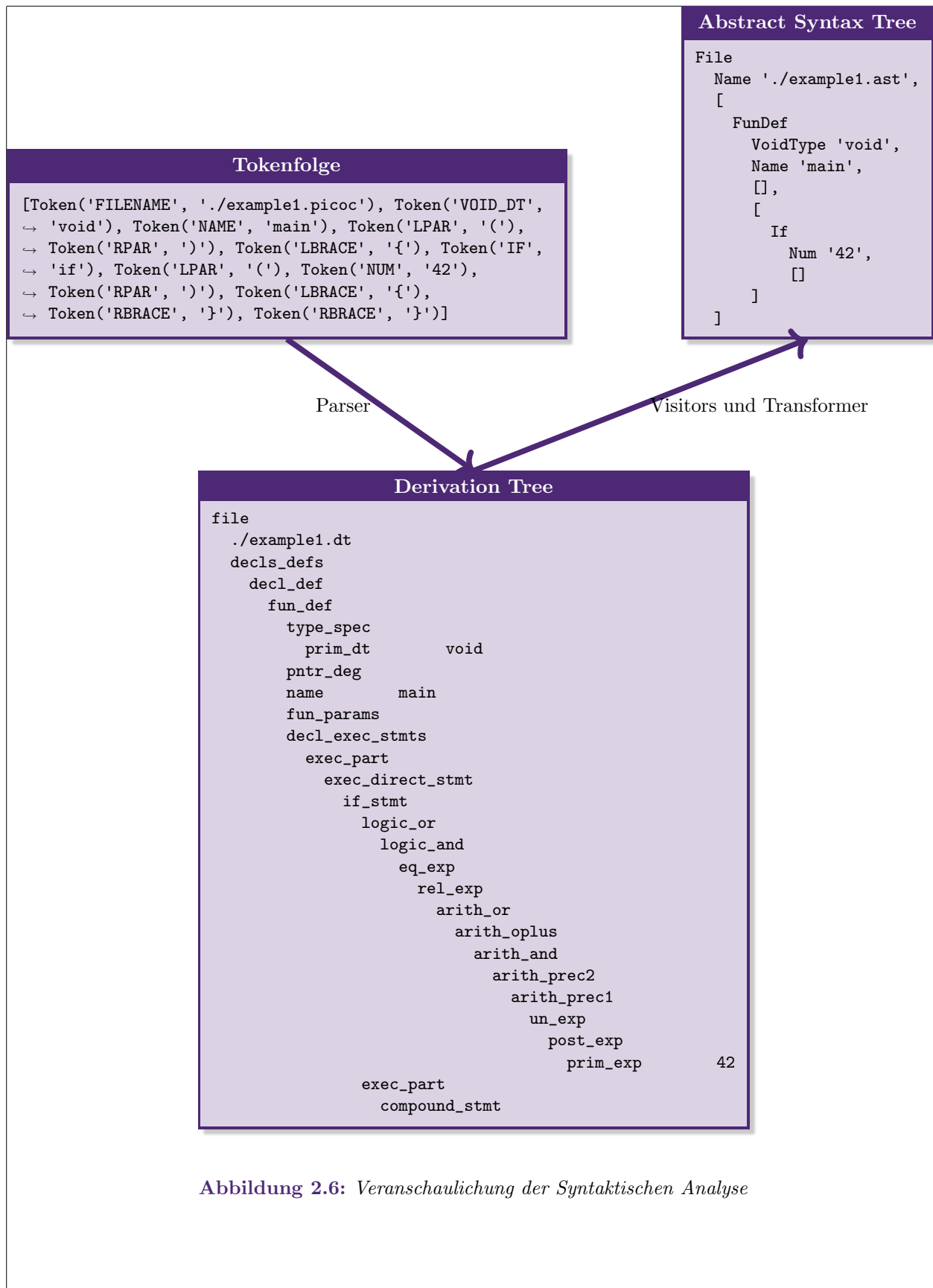


Abbildung 2.6: Veranschaulichung der Syntaktischen Analyse

2.5 Code Generierung

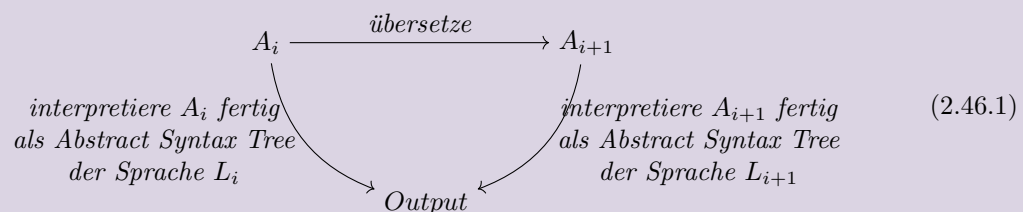
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstract Syntax Tree** einer Sprache L_1 in den **Abstract Syntax Tree** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.46) nennt. So wie es auch schon mit dem **Derivation Tree** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstract Syntax Tree** konstruiert hatte. Aus dem Derivation konnte, dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstract Syntax Tree** generiert werden.

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

Definition 2.46: Pass

*Einzelner Übersetzungsschritt in einem Kompilervorgang von einem **Abstract Syntax Tree** A_i einer Sprache L_i zu einem **Abstract Syntax Tree** 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.^{a,b}*

Für jeden **Pass** gilt ähnlich, wie bei einem **vollständigen Compiler** in 2.46.1, dass:



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

^aEin **Pass** kann mit einem **Transpiler** 2.9 (Definition 2.9) verglichen werden, da sich die zwei Sprachen L_i und L_{i+1} aufgrund der **Kleinschrittigkeit** meist auf einem ähnlichen **Abstraktionslevel** befinden. Der Unterschied ist allerdings, dass ein **Transpiler** zwei Programme, die in L_i bzw. L_{i+1} geschrieben sind kompiliert. Ein **Pass** ist dagegen immer **kleinschrittig** und operiert ausschließlich auf **Abstract Syntax Trees**, ohne Parsing usw.

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

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

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

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

2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tun, welche **Unreine Ausdrücke** (Definition 2.48) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.47) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 2.47: Reiner Ausdruck (bzw. engl. pure expression)

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

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

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

Definition 2.48: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definition 2.49).

Definition 2.49: Monadische Normalform (bzw. engl. monadic normal form)

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

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

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

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.7 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**¹⁰ aufgeschrieben wurden.

In der Abbildung 2.7 ist der Ausdruck mit dem **Nebeneffekt** eine Variable zu **allokieren**: `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt, daher muss der **Unreine** Ausdruck als eigenständiges Statement **vorangestellt** werden.

¹⁰Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

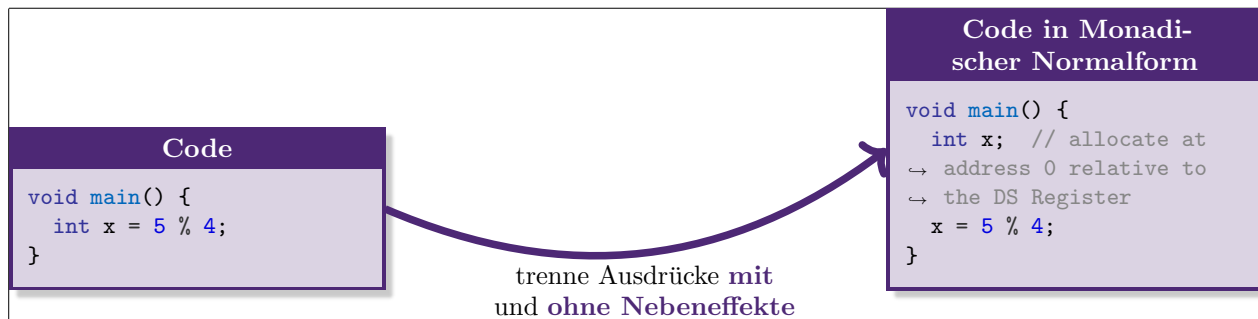


Abbildung 2.7: Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten

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

2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.52) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den **Komplexen Ausdrücken** **vorangestellte** **Statements** macht, in denen die **Komplexen Ausdrücke** **temporären Locations** zugewiesen werden (Definiton 2.50) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statement**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der **Komplexe Ausdruck** **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in **Statements** zur Zuweisung an **Locations** auftauchen, aber die **Komplexen Ausdrücke** nur **Atomare Ausdrücke** (Definiton 2.51) enthalten.

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

Definition 2.50: Location

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

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

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.53). Wenn eine **Konkrete Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.53), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.51).

Definition 2.51: Atomarer Ausdruck

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

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

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

Definition 2.52: Komplexer Ausdruck

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

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

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

Definition 2.53: A-Normalform (ANF)

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

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

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

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

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

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**¹¹ aufgeschrieben wurden.

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** inklusive **Liveness Analysis**, um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.¹²

Aus diesem Grund verwendet das Beispiel in Abbildung 2.8 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **Abstract Syntax Tree** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdruck** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl 4**, **Variable** `var` und **ASCII-Zeichen** `'c'` nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

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

¹¹Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

¹²Die in diesem **Paragraph** erwähnten **Begriffe** werden **nicht** genauer erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

befehlssatz es erlaubt **zwei Register** miteinander zu verrechnen¹³, ist es möglich **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** 'c' als **atomar** zu definieren, da sie mit einem **Maschinenbefehl** verarbeitet werden können¹⁴. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das SP-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** 'c' als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verrechnet werden können.

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

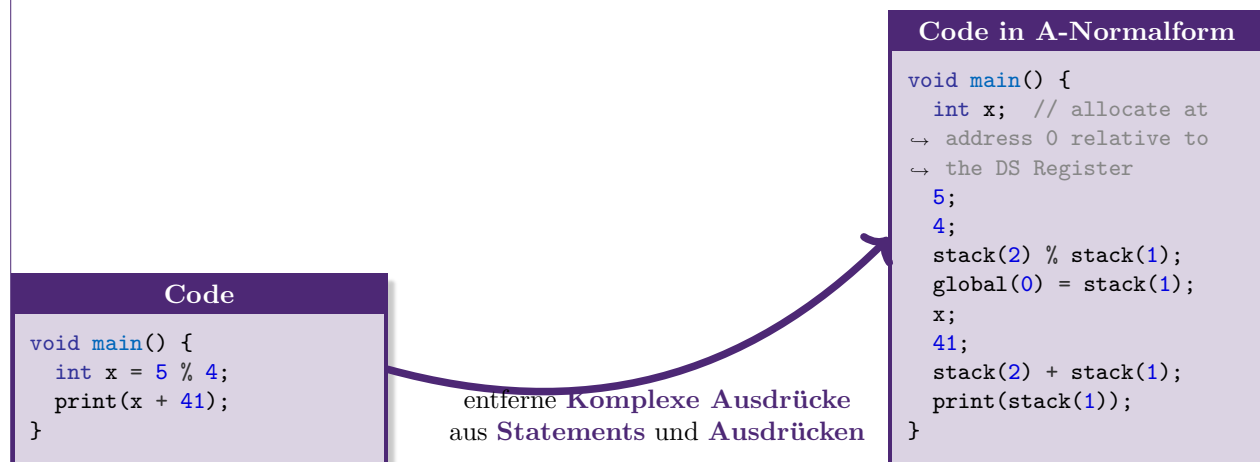


Abbildung 2.8: Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen

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

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

2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakt Syntax Tree** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt¹⁵. Der Weg von **Abstrakter Syntax** zu **Konkreter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax**

¹³Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.

¹⁴Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. **MULT ACC IN2**.

¹⁵Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenslesbaren Repräsentation** ausgegeben.

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

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

2.6 Fehlermeldungen

Definition 2.54: Fehlermeldung

Benachrichtigung beliebiger Form, die darüber informiert, dass:

1. Ein Program beim **Kompilieren** von der **Konkreten Syntax** abweicht, also der **Inputstring** sich nicht mit der Konrekten Syntax **ableiten** lässt oder auf etwas **zugegriffen** werden soll, was noch **nicht** deklariert oder definiert wurde.
2. Beim Ausführen eine **verbotene** Operation ausgeführt wurde.^a

^aErrors in C/C++ - GeeksforGeeks.

2.6.1 Kategorien von Fehlermeldungen

3 Implementierung

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

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

Definition 3.1: Metasyntax

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

Definition 3.2: Metasprache

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

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

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

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

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

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

^bNebel, „Theoretische Informatik“.

^cGrammar Reference — Lark documentation.

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

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

²Lark - a parsing toolkit for Python.

³Shinan, *lark*.

Definition 3.4: Dialekt der EBNF aus Lark

Das **Lark Parsing Toolkit** verwendet eine *eigene Notation* für die **Erweiterte Backus-Naur-Form**, die sich teilweise in einzelnen Aspekten von der Syntax aus dem **Standard** unterscheidet und unter [Link^a](https://lark-parser.readthedocs.io/en/latest/grammar.html) dokumentiert ist.

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

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

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

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

Eine möglichst **geringe Laufzeit** durch Verwenden der effizientesten Algorithmen zu erreichen war keine der Hauptzielsetzungen für den **PicoC-Compiler**, da der **PicoC-Compiler** vor allem als **Lerntool** konzipiert ist, mit dem Studenten lernen können, wie der **Kompiliervorgang** von der Programmiersprache L_{PicoC} zur Maschinensprache L_{RETI} funktioniert. Eine ausführliche Diskussion zur Priorisierung Laufzeit wurde in Unterkapitel 1.5 geführt. Lark besitzt des Weiteren eine **sehr gute Dokumentation** *Welcome to Lark's documentation! — Lark documentation*, sodass anderen Studenten, die den **PicoC-Compiler** vielleicht in ihr Projekt einbinden wollen, unkompliziert **Erweiterungen** für den **PicoC-Compiler** schreiben können.

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

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

Definition 3.5: Abstrakte Syntax Form (ASF)

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

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

⁴**ANTLR**.

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

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

3.1 Lexikalische Analyse

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

3.1.1 Konkrete Syntax für die Lexikalische Analyse

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

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

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

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

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

Der **Basic Lexer** des **Lark Parsing Toolkit** funktioniert grundlegend so wie es im Unterkapitel 2.3 erklärt wurde, allerdings berücksichtigt der **Basic Lexer** ebenfalls **Prioritäten**, sodass für den aktuellen Index im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Wort** an dem aktuellen Index im Eingabeprogramm erkennt, bekommt es direkt den

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

Tokenwert dieser Produktion zugewiesen, weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

<i>COMMENT</i>	::=	"//"/[\backslash n]*/ "/*"/(\cdot \backslash n)*?/"*/"	<i>L_Comment</i>
<i>RETI_COMMENT.2</i>	::=	"//"" \backslash "?"#[\backslash n]*/	
<i>DIG_NO_0</i>	::=	"1" "2" "3" "4" "5" "6" "7" "8" "9"	<i>L_Arith</i>
<i>DIG_WITH_0</i>	::=	"0" <i>DIG_NO_0</i>	
<i>NUM</i>	::=	"0" <i>DIG_NO_0 DIG_WITH_0*</i>	
<i>ASCII_CHAR</i>	::=	" \backslash " \cdot " \cdot " \sim "	
<i>CHAR</i>	::=	"'" <i>ASCII_CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII_CHAR</i> + ". <i>picoc</i> "	
<i>LETTER</i>	::=	"a" \cdot " \cdot "z" "A" \cdot " \cdot "Z"	
<i>NAME</i>	::=	(<i>LETTER</i> " \backslash ") (<i>LETTER</i> <i>DIG_WITH_0</i> " \backslash ")*	
<i>name</i>	::=	<i>NAME</i> <i>INT_NAME</i> <i>CHAR_NAME</i> <i>VOID_NAME</i>	
<i>LOGIC_NOT</i>	::=	"!"	
<i>NOT</i>	::=	" \sim "	
<i>REF_AND</i>	::=	"&"	
<i>un_op</i>	::=	<i>SUB_MINUS</i> <i>LOGIC_NOT</i> <i>NOT</i> <i>MUL_DEREF_PNTR</i> <i>REF_AND</i>	
<i>MUL_DEREF_PNTR</i>	::=	"*"	
<i>DIV</i>	::=	"/"	
<i>MOD</i>	::=	"%"	
<i>prec1_op</i>	::=	<i>MUL_DEREF_PNTR</i> <i>DIV</i> <i>MOD</i>	
<i>ADD</i>	::=	"+"	
<i>SUB_MINUS</i>	::=	"-"	
<i>prec2_op</i>	::=	<i>ADD</i> <i>SUB_MINUS</i>	
<i>LT</i>	::=	"<"	<i>L_Logic</i>
<i>LTE</i>	::=	"<="	
<i>GT</i>	::=	">"	
<i>GTE</i>	::=	">="	
<i>rel_op</i>	::=	<i>LT</i> <i>LTE</i> <i>GT</i> <i>GTE</i>	
<i>EQ</i>	::=	"=="	
<i>NEQ</i>	::=	"!="	
<i>eq_op</i>	::=	<i>EQ</i> <i>NEQ</i>	
<i>INT_DT.2</i>	::=	"int"	<i>L_Assign_Alloc</i>
<i>INT_NAME.3</i>	::=	"int" (<i>LETTER</i> <i>DIG_WITH_0</i> " \backslash ")+	
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" (<i>LETTER</i> <i>DIG_WITH_0</i> " \backslash ")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" (<i>LETTER</i> <i>DIG_WITH_0</i> " \backslash ")+	
<i>prim_dt</i>	::=	<i>INT_DT</i> <i>CHAR_DT</i> <i>VOID_DT</i>	

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

3.1.2 Codebeispiel

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

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

Code 3.1: PicoC-Code des Codebeispiels

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

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

Code 3.2: Tokens für das Codebeispiel

3.2 Syntaktische Analyse

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

3.2.1 Umsetzung von Präzedenz und Assoziativität

Die Programmiersprache L_{PicoC} hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache L_C ⁷. Die **Präzidenzregeln** der Programmiersprache L_{PicoC} sind in Tabelle 3.1 aufgelistet.

⁷C Operator Precedence - cppreference.com.

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

Tabelle 3.1: Präzidenzregeln von PicoC

Würde man diese **Operatoren** ohne Beachtung von **Präzidenzregeln** (Definiton 2.31) und **Assoziativität** (Definition 2.30) in eine Grammatik verarbeiten wollen, so könnte eine Grammatik, wie Grammatik 3.2.1 dabei rauskommen.

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

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

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

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

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

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

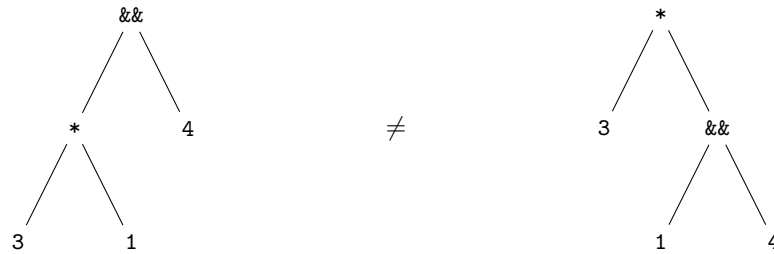


Abbildung 3.1: Ableitungsbäume zu den beiden Ableitungen

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

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

<code>prim_exp</code>	::=	...	<i>L_Arith + L_Array</i>
<code>post_exp</code>	::=	...	<i>+ L_Pntr + L_Struct</i>
<code>un_exp</code>	::=	...	<i>+ L_Fun</i>
<code>arith_prec1</code>	::=	...	
<code>arith_prec2</code>	::=	...	
<code>arith_and</code>	::=	...	
<code>arith_oplus</code>	::=	...	
<code>arith_or</code>	::=	...	
<code>rel_exp</code>	::=	...	<i>L_Logic</i>
<code>eq_exp</code>	::=	...	
<code>logic_and</code>	::=	...	
<code>logic_or</code>	::=	...	
<code>assign_stmt</code>	::=	...	<i>L_Assign</i>

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

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

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

Tabelle 3.2: Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren

Als nächstes müssen die einzelnen **Produktionen** entsprechend der **Ausdrücke** für die sie zuständig sind definiert werden. Jede der **Produktionen** soll nur Ausdrücke **erkennen** können, deren **Präzidenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzidenzstufe **höher** ist. Z.B. soll **un_op** sowohl den Ausdruck $-(3 * 14)$ als auch einfach nur $(3 * 14)$ ⁸ erkennen können, aber nicht $3 * 14$ ohne Klammern, da dieser Ausdruck eine **geringe Präzidenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

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

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

Grammar 3.2.3: Beispiel für eine unäre rechtsassoziative Produktion

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

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

Grammar 3.2.4: Beispiel für eine unäre linksassoziative Produktion

Bei z.B. der Produktion **prec2_exp** in 3.2.5 für die **binären linksassoziativen Operatoren** $a+b$ und $a-b$ ist die **Alternative** **arith_prec2 prec2_op arith_prec1** dafür zuständig, dass **mehrere** Operationen der Präzidenzstufe 4 in Folge erkannt werden können⁹ (z.B. $3 + 1 - 4$, wobei $-$ und $+$ beide Präzidenzstufe 4 haben). Das **Nicht-Terminalsymbol** **arith_prec1** auf der **rechten Seite** ermöglicht es, dass zwischen den

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

⁹Bezogen auf Tabelle 3.1.

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

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

Grammar 3.2.5: *Beispiel für eine linksassoziative Produktion*

Manche **Parser**^a haben allerdings ein Problem mit **Linksrekursion** (Definition 2.25), wie sie z.B. in der Produktion 3.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 3.2.5 zur Produktion 3.2.6 umschreibt.

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

Grammar 3.2.6: *Beispiel für eine linksassoziative Produktion*

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

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

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

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

<i>prec1_op</i>	::=	"*" "/" "%"	<i>L_Misc</i>
<i>prec2_op</i>	::=	"+" "-"	
<i>rel_op</i>	::=	"<" "<=" ">" ">="	
<i>eq_op</i>	::=	"==" "!="	
<i>fun_args</i>	::=	[<i>logic_or</i> ("," <i>logic_or</i>)*]	
<i>prim_exp</i>	::=	<i>name</i> <i>NUM</i> <i>CHAR</i> "(" <i>logic_or</i> ")"	<i>L_Arith</i>
<i>post_exp</i>	::=	<i>post_exp</i> [" <i>logic_or</i> "] <i>post_exp</i> ." <i>name</i> " <i>name</i> (" <i>fun_args</i> ")"	+ <i>L_Array</i>
		<i>prim_exp</i>	+ <i>L_Pntr</i>
<i>un_exp</i>	::=	<i>un_op</i> <i>un_exp</i> <i>post_exp</i>	+ <i>L_Struct</i>
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i> <i>un_exp</i>	+ <i>L_Fun</i>
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i> <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i> <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i> <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i> <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i> <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_op</i> <i>rel_exp</i> <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i> <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> " " <i>logic_and</i> <i>logic_and</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	<i>L_Assign</i>

Grammar 3.2.7: Durchdachte Konkrete Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF

3.2.2 Konkrete Syntax für die Syntaktische Analyse

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

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrete Syntax** an die **aktuellste konsistentlos auffindbare Version** der echten Grammatik *ANSI C grammar (Yacc)* der Sprache L_C angepasst¹⁰, damit es sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C , wobei z.B. die Compiler **GCC**¹¹ und **Clang**¹² zu nennen wären.

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

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

¹⁰An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts verändert, wie die Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Lex)* und **Lexikalische Analyse** *noauthor'ansi'nodate-2* aus dem Jahre 1985 zeigen.

¹¹*GCC, the GNU Compiler Collection - GNU Project.*

¹²*clang: C++ Compiler.*

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

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

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

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

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

3.2.3 Derivation Tree Generierung

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

Definition 3.6: Earley Parser

3.2.3.1 Codebeispiel

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

```

1 file
2   ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt
3   decls_defs
4     decl_def
5       struct_decl
6         name      st

```

```

7      struct_params
8      alloc
9      type_spec
10     prim_dt      int
11     pntr_decl
12     pntr_deg      *
13     array_decl
14     pntr_decl
15     pntr_deg      *
16     array_decl
17     name      attr
18     array_dims
19     array_dims
20     4
21     5
22 decl_def
23 fun_def
24     type_spec
25     prim_dt      void
26     pntr_deg
27     name      main
28     fun_params
29     decl_exec_stmts
30     decl_part
31     decl_exp_stmt
32     alloc
33     type_spec
34     struct_spec
35     name      st
36     pntr_decl
37     pntr_deg      *
38     array_decl
39     pntr_decl
40     pntr_deg      *
41     array_decl
42     name      var
43     array_dims
44     3
45     2
46     array_dims

```

Code 3.3: Derivation Tree nach Derivation Tree Generierung

3.2.3.2 Ausgabe des Derivation Tree

Die Ausgabe des **Derivation Tree** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Grammatik den **linken Seiten** des „kann abgeleitet werden zu“-Symbols $::=$ ¹³ entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition 2.39 eines **Derivation Tree** auch schon definiert ist. Die **EBNF-Grammatik 3.2.8** des **PicoC-Compilers** erlaubt es allerdings auch, dass in einem **Blatt** garnichts ϵ steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("["NUM"]")*` gibt, in denen auch das **leere Wort** ϵ abgeleitet werden kann.

Die Ausgabe des **Abstract Syntax Tree** ist bewusst so gewählt, dass sie sich optisch vom **Derivation**

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

Tree unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase** geschrieben sind, im Gegensatz zum **Derivation Tree**, dessen **Innere Knoten** im **snake_case** geschrieben sind, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des „kann abgeleitet werden zu“-Symbols ::=.

3.2.4 Derivation Tree Vereinfachung

Der **Derivation Tree** in Code 3.3, dessen Generierung in Unterkapitel 3.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Transformers** ein **Abstract Syntax Tree** generiert werden kann. Das Problem ist, dass um den **Datentyp** einer Variable in der Programmiersprache L_C und somit auch die Programmiersprache L_{PicoC} korrekt bestimmen zu können, wie z.B. ein „**Array der Mächtigkeit 3 von Pointern auf Arrays der Mächtigkeit 2 von Integeren**“ `int (*ar[3])[2]` die **Spiralregel**¹⁴ in der Implementierung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Grammatik 3.2.8 der **Konkreten Syntax** auf eine spezielle Weise passend spezifiziert.

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

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

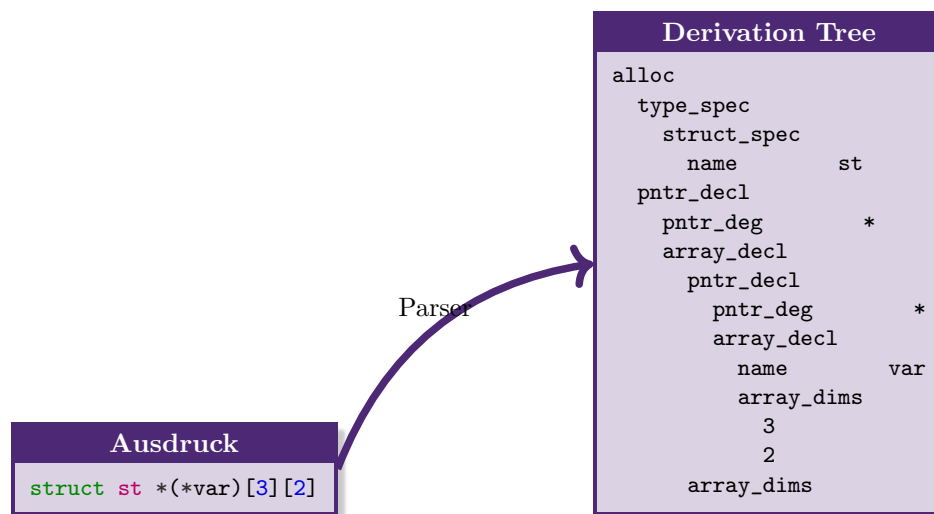


Abbildung 3.2: Derivation Tree nach Parsen eines Ausdrucks

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

Des Weiteren befindet sich in der **Mitte** dieser **Spirale**, die der **Derivation Tree** bildet der **Name der**

¹⁴Clockwise/Spiral Rule.

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

In Abbildung 3.3 ist daher zu sehen, wie der **Derivation Tree** aus Abbildung 3.2 mithilfe eines **Visitors** (Definition 2.43) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

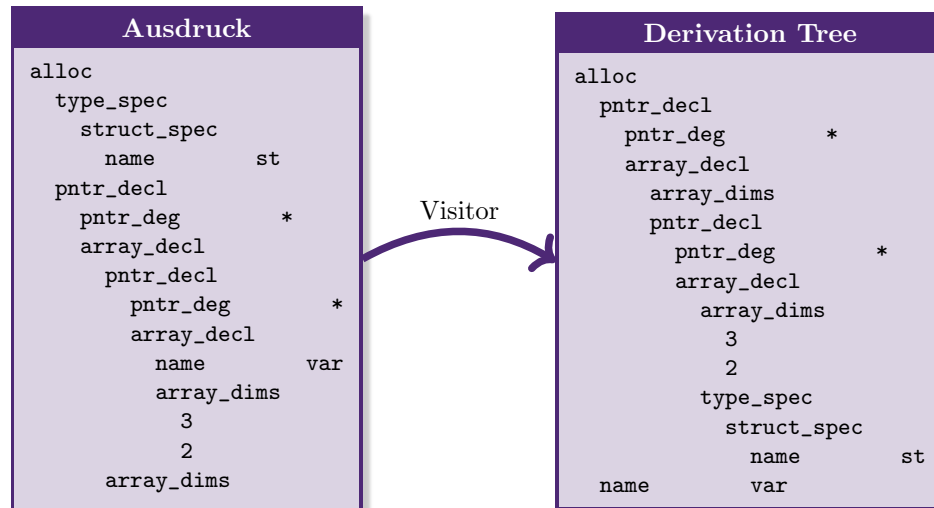


Abbildung 3.3: Derivation Tree nach Vereinfachung

3.2.4.1 Codebeispiel

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

```

1 file
2   ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3 decls_defs
4   decl_def
5     struct_decl
6       name      st
7       struct_params
8         alloc
9           ptr_decl
10            ptr_deg   *
11            array_decl
12              array_dims
13                4
14                5
15            ptr_decl
16              ptr_deg   *
17              array_decl
18                array_dims
19                type_spec
20                  prim_dt      int
21            name      attr

```

```

22 decl_def
23   fun_def
24     type_spec
25       prim_dt      void
26   ptr_deg
27     name      main
28   fun_params
29   decl_exec_stmts
30   decl_part
31   decl_exp_stmt
32     alloc
33       ptr_decl
34         ptr_deg      *
35         array_decl
36           array_dims
37             ptr_decl
38               ptr_deg      *
39               array_decl
40                 array_dims
41                   3
42                   2
43                 type_spec
44                   struct_spec
45                     name      st
46                   name      var

```

Code 3.4: Derivation Tree nach Derivation Tree Vereinfachung

3.2.5 Abstrakt Syntax Tree Generierung

Nachdem der **Derivation Tree** in Unterkapitel 3.2.4 vereinfacht wurde, ist der **vereinfachte Derivation Tree** in Code 3.4 nun dazu geeignet, um mit einem **Transformer** (Definition 2.42) einen **Abstract Syntax Tree** aus ihm zu generieren. Würde man den **vereinfachten Derivation Tree** des Ausdrucks `struct st *(*var[3][2])` auf passende Weise in einen **Abstract Syntax Tree** umwandeln, so würde dabei ein **Abstract Syntax Tree** wie in Abbildung 3.4 rauskommen.

Den Teilbaum, der den Datentyp darstellt würde man von **oben-nach-unten**¹⁵ als „**Pointer auf einen Pointer auf ein Array der Mächtigkeit 2, 3 von Structs des Typs st**“ lesen, also genau anders herum, als man den Ausdruck `struct st *(*var[3][2])` mit der **Spiralregel** lesen würde. Bei der **Spiralregel** fängt man beim Ausdruck `struct st *(*var[3][2])` bei der Variable `var` an und arbeitet sich dann auf „**Spiralbahnen**“, von **innen-nach-außen** durch den Ausdruck, um herauszufinden, dass dieser Datentyp ein „**Array der Mächtigkeit 3, 2 von Pointern auf einen Pointer auf einen Struct vom Typ st**“ ist.

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

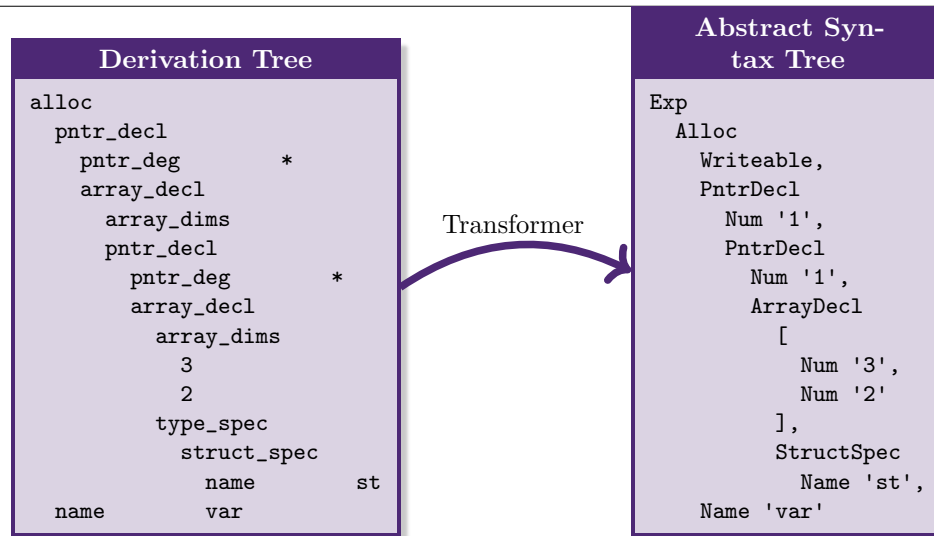


Abbildung 3.4: Abstract Syntax Tree Generierung ohne Umdrehen

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

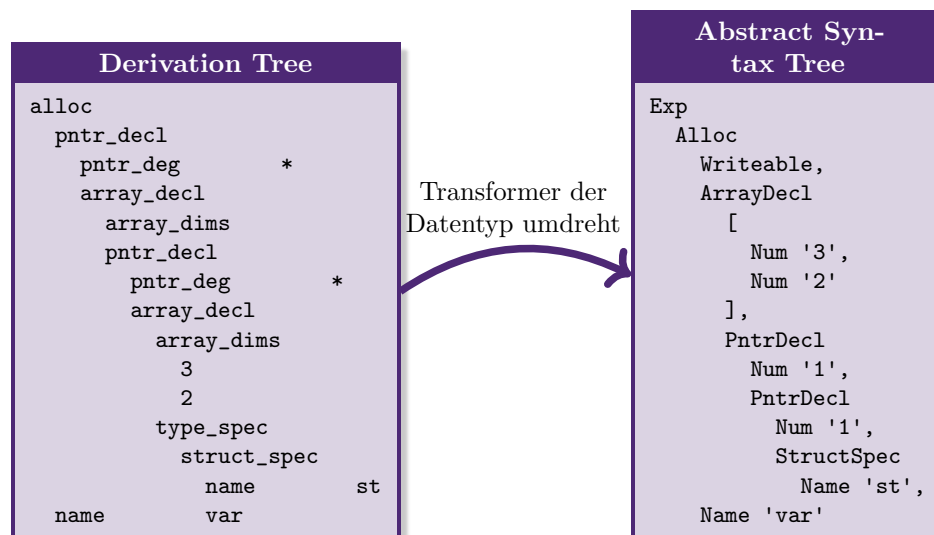


Abbildung 3.5: Abstract Syntax Tree Generierung mit Umdrehen

Die Weiterverarbeitung des **Abstract Syntax Trees** geschieht mithilfe von **Passes**, welche im Unterkapitel 2.5 genauer beschrieben werden. Da die Knoten des **Abstract Syntax Tree** anders als beim **Derivation Tree** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Grammatik der **Konkreten Syntax**

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

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

3.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache L_{PicoC} darstellen. Für die **PicoC-Knoten** wurden möglichst **kurze** und **leicht** verständliche Bezeichner gewählt, da auf diese Weise bei der Implementierung der einzelnen Passes möglichst **viel Code in eine Zeile** passt und dieser Code auch durch leicht verständliche Bezeichner von Knoten **intuitiv verständlich** sein sollte¹⁶. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstract Syntax Trees** vorkommen sind in Tabelle 3.3 mit einem **Beschreibungstext** dokumentiert.

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

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

Tabelle 3.3: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen beliebigen Ausdruck , dessen Ergebnis auf den Stack soll. Zudem besitzt er 2 versteckte Attribute, wobei datatype im RETI Blocks Pass wichtig ist und error_data für Fehlermeldungen wichtig ist.
Stack(num)	Container, der für das temporäre Ergebnis einer Berechnung, das num Speicherzellen relativ zum Stackpointer Register SP steht.
Stackframe(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Begin-Aktive-Funktion Register BAF steht.
Global(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Datensegment Register DS steht.
StackMalloc(num)	Container, der für das Allokieren von num Speicherzellen auf dem Stack steht.
PntrDecl(num, datatype)	Container, der für den Pointerdatatype steht: <prim_dt> *<var> , wobei das Attribut num die Anzahl zusammengefasster Pointer angibt und datatype der Datentyp ist, auf den der oder die Pointer zeigen.
Ref(exp, datatype, error_data)	Container, der für die Anwendung des Referenz-Operators &<var> steht und die Adresse einer Location (Definition 2.50) auf den Stack schreiben soll, die über exp eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei datatype im RETI Blocks Pass wichtig ist und error_data für Fehlermeldungen wichtig ist.
Deref(lhs, exp)	Container für den Indezzugriff auf einen Array- oder Pointerdatatype : <var>[<i>] , wobei exp1 eine angehängte weitere Subscr(exp1, exp2) , Deref(exp1, exp2) , Attr(exp, name) oder ein Name('var') sein kann und exp2 der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den Arraydatatype steht: <prim_dt> <var>[<i>] , wobei das Attribut nums eine Liste von Num('x') ist, die die Dimensionen des Arrays angibt und datatype der Datentyp ist, der über das Anwenden von Subscript() auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den Initializer eines Arrays , dessen Einträge exps weitere Initializer für eine Array-Dimension oder ein Initializer für ein Struct oder ein Logischer Ausdruck sein können, z.B. {{1, 2}, {3, 4}} . Des Weiteren besitzt er ein verstecktes Attribut datatype , welches für den PicoC-ANF Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
Subscr(exp1, exp2)	Container für den Indezzugriff auf einen Array- oder Pointerdatatype : <var>[<i>] , wobei exp1 eine angehängte weitere Subscr(exp1, exp2) , Deref(exp1, exp2) oder Attr(exp, name) Operation sein kann oder ein Name('var') sein kann und exp2 der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten Structdatatype : struct <name> , wobei das Attribut name festlegt, welchen selbst definierte Structdatatype dieser Container-Knoten repräsentiert.
Attr(exp, name)	Container für den Attributzugriff auf einen Structdatatype : <var>.<attr> , wobei exp1 eine angehängte weitere Subscr(exp1, exp2) , Deref(exp1, exp2) oder Attr(exp, name) Operation sein kann oder ein Name('var') sein kann und name das Attribut ist, auf das zugegriffen werden soll.

Tabelle 3.4: PicoC-Knoten Teil 2

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

Tabelle 3.5: PicoC-Knoten Teil 3

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts.blocks)	Container für eine Funktionsdefinition , z.B. <datatype> <fun.name>(<datatype> <param>) {<stmts>}, wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist, allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts.blocks eine Liste von Statements bzw. Blöcken ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die Erstellung eines neuen Stackframes und Speicherung des Werts des BAF-Registers der aufgerufenen Funktion und der Rücksprungsadresse nacheinander an den Anfang des neuen Stackframes . Das Attribut fun_name steht dabei für den Bezeichner der Funktion, für die ein neuer Stackframe erstellt werden soll. Das Attribut fun_name dient später dazu den Block dieser Funktion zu finden, weil dieser für den weiteren Kompilierungsvorgang wichtige Information in seinen versteckten Attributen gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die Adresse des Befehls, der direkt auf die Jump Instruction folgt, ersetzt wird.
RemoveStackframe()	Container für das Entfernen des aktuellen Stackframes durch das Wiederherstellen des im noch aktuellen Stackframe gespeicherten Werts des BAF-Registers der aufgerufenen Funktion und das Setzen des SP-Registers auf den Wert des BAF-Registers vor der Wiederherstellung.
File(name, decls_defs.blocks)	Container für alle Funktionen oder Blöcke , welche eine Datei als Ursprung haben, wobei name der Dateiname der Datei ist, die erstellt wird und decls_defs.blocks eine Liste von Funktionen bzw. Blöcken ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für Statements , der auch als Block bezeichnet wird, wobei das Attribut name der Bezeichner des Labels (Definition 3.7) des Blocks ist und stmts_instrs eine Liste von Statements oder Instructions . Zudem besitzt er noch 3 versteckte Attribute, wobei instrs_before die Zahl der Instructions vor diesem Block zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die Parameter der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für ein Goto zu einem anderen Block , wobei das Attribut name der Bezeichner des Labels des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen Kommentar , den der Compiler selber während des Kompilierungsvorgangs erstellt, der im RETI-Interpreter selbst später nicht sichtbar sein wird, aber in den Immediate-Dateien , welche die Abstract Syntax Trees nach den verschiedenen Passes enthalten.
RETIComment(value)	Container für einen Kommentar im Code der Form: // # comment, der im RETI-Interpreter später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer RETI-CPU nicht umsetzbar ist und auch nicht sinnvoll wäre umzusetzen. Der Kommentar ist im Attribut value , welches jeder Knoten besitzt gespeichert.

Tabelle 3.6: PicoC-Knoten Teil 4

Definition 3.7: Label

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

^aThiemann, „Compilerbau“.

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

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 3.8) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 3.9) unbesetzt ist. Das **Attribut position** wird später für Fehlermeldungen gebraucht.

Definition 3.8: Token-Knoten

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

Token-Knoten entsprechen im **Abstract Syntax Tree** **Blättern**.^a

^aThiemann, „Compilerbau“.

Definition 3.9: Container-Knoten

Dient als **Container** für andere **Container-Knoten** und **Token-Knoten**. Die **Container-Knoten** werden optimalerweise immer so gewählt, dass sie **mehrere Produktionen der Konkreten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.^a

Container-Knoten entsprechen im **Abstract Syntax Tree** **Inneren Knoten**.^b

^aWie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. **1 % 3** und **Logischen Ausdrücke**, wie z.B. **1 && 2 < 3**, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.

^bThiemann, „Compilerbau“.

3.2.5.2 RETI-Knoten

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

RETI-Knoten	Beschreibung
Program(name, instrs)	Container für alle Instructions : <name> <instrs>, wobei name der Dateiname der Datei ist, die erstellt wird und instrs eine Liste von Instructions ist.
Instr(op, args)	Container für eine Instruction : <op> <args>, wobei op eine Operation ist und args eine Liste von Argumenten für dieser Operation.
Jump(rel, im_goto)	Container für eine Jump-Instruction : JUMP<rel> <im>, wobei rel eine Relation ist und im_goto ein Immediate Value Im(val) für die Anzahl an Speicherzellen , um die relativ zur Jump-Instruction gesprungen werden soll oder ein GoTo(Name('block.xyz')) , das später im RETI-Patch Pass durch einen passenden Immediate Value ersetzt wird.
Int(num)	Container für einen Interruptaufruf : INT <im>, wobei num die Interruptvektornummer (IVN) für die passende Speicherzelle in der Interruptvektortabelle ist, in der die Adresse der Interrupt-Service-Routine (ISR) steht.
Call(name, reg)	Container für einen Prozeduraufruf : CALL <name> <reg>, wobei name der Bezeichner der Prozedur, die aufgerufen werden soll ist und reg ein Register ist, das als Argument an die Prozedur dient. Diese Operation ist in der Betriebssysteme Vorlesung ^a nicht deklariert, sondern wurde dazuerfunden, um unkompliziert ein CALL PRINT ACC oder CALL INPUT ACC im RETI-Interpreter simulieren zu können.
Name(val)	Bezeichner für eine Prozedur , z.B. PRINT oder INPUT oder den Programmnamen , z.B. PROGRAMNAME. Dieses Argument ist in der Betriebssysteme Vorlesung ^a nicht deklariert, sondern wurde dazuerfunden, um Bezeichner, wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein Register .
Im(val)	Ein Immediate Value , z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), Oplus(), Or(), And()	Compute-Memory oder Compute-Register Operationen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(), Oplusi(), Ori(), Andi()	Compute-Immediate Operationen: ADDI, SUBI, MULTI, DIVI, MODI, OPLUSI, ORI, ANDI.
Load(), Loadin(), Loadi()	Load Operationen: LOAD, LOADIN, LOADI.
Store(), Storein(), Move()	Store Operationen: STORE, STOREIN, MOVE.
Lt(), LtE(), Gt(), GtE(), Eq(), NEq(), Always(), NOP()	Relationen : <, <=, >, >=, ==, !=, _NOP.
Rti()	Return-From-Interrupt Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()	Register : PC, IN1, IN2, ACC, SP, BAF, CS, DS.

^a C. Scholl, „Betriebssysteme“

Tabelle 3.7: RETI-Knoten

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

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

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

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

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

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

3.2.5.4 Abstrakte Syntax

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

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp(<exp>, <bin_op>, <exp>)</i> <i>UnOp(<un_op>, <exp>)</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i>	
<i>stmt</i>	::=	<i>Exp(<exp>)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(<exp>, <rel>, <exp>)</i> <i>ToBool(<exp>)</i>	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref(<exp>, <exp>)</i> <i>Ref(<exp>)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, <exp>)</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>StructAssign(Name(str), <exp>+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(<exp>, <stmt>*)</i> <i>IfElse(<exp>, <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(<exp>, <stmt>*)</i> <i>DoWhile(<exp>, <stmt>*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(<exp>)</i>	
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*)</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))* <stmt>*)</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>

Grammar 3.2.10: Abstrakte Syntax der Sprache L_{PicoC}

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

3.2.5.5 Codebeispiel

In Code 3.5 ist der **Abstract Syntax Tree** zu sehen, der aus dem **vereinfachten Derivation Tree** aus Code 3.4 mithilfe eines **Transformers** generiert wurde.

```

1 File
2   Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           PtrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '4',
14                Num '5'
15              ],
16            PtrDecl
17              Num '1',
18              IntType 'int',
19            Name 'attr'
20          ],
21      FunDef
22        VoidType 'void',
23        Name 'main',
24        [],
25        [
26          Exp
27            Alloc
28              Writeable,
29              ArrayDecl
30                [
31                  Num '3',
32                  Num '2'
33                ],
34              PtrDecl
35                Num '1',
36                PtrDecl
37                  Num '1',
38                  StructSpec
39                    Name 'st',
40                    Name 'var'
41                ]
42          ]

```


Code 3.5: Aus vereinfachtem Derivation Tree generierter Abstract Syntax Tree**3.2.5.6 Ausgabe des Abstract Syntax Tree**

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

Das Ausgeben eines **Abstract Syntax Trees** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`¹⁷ der Programmiersprache **Python** umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkreten Syntax** aufgebaute **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **runden öffnenden (** und **schließenden) Klammern**, sowie **Kommas** `,`, **Semikolons** `;` usw. zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach dem **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene **Textrepräsentation** passend **zusammenfügen** und selbst **zurückgeben**.

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

3.3 Code Generierung

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

Man steht nun dem Problem gegenüber einen **Abstract Syntax Tree** der Sprache L_{PicoC} , der durch die **Abstrakte Syntax** in Grammatik 3.2.10 spezifiziert ist in einen entsprechenden **Abstract Syntax Tree** der Sprache L_{RETI} umzuformen. Das ganze lässt sich, wie in Unterkapitel 2.5 bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition 2.46) herunterbricht.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton 2.10). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel 2.1.1) in Abbildung 3.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer $X_{86,64}$ -Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer $X_{86,64}$ -Architektur laufen könnte. Und selbst dieses **T-Diagramm** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu $X_{86,64}$ kompiliert wurde usw.

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

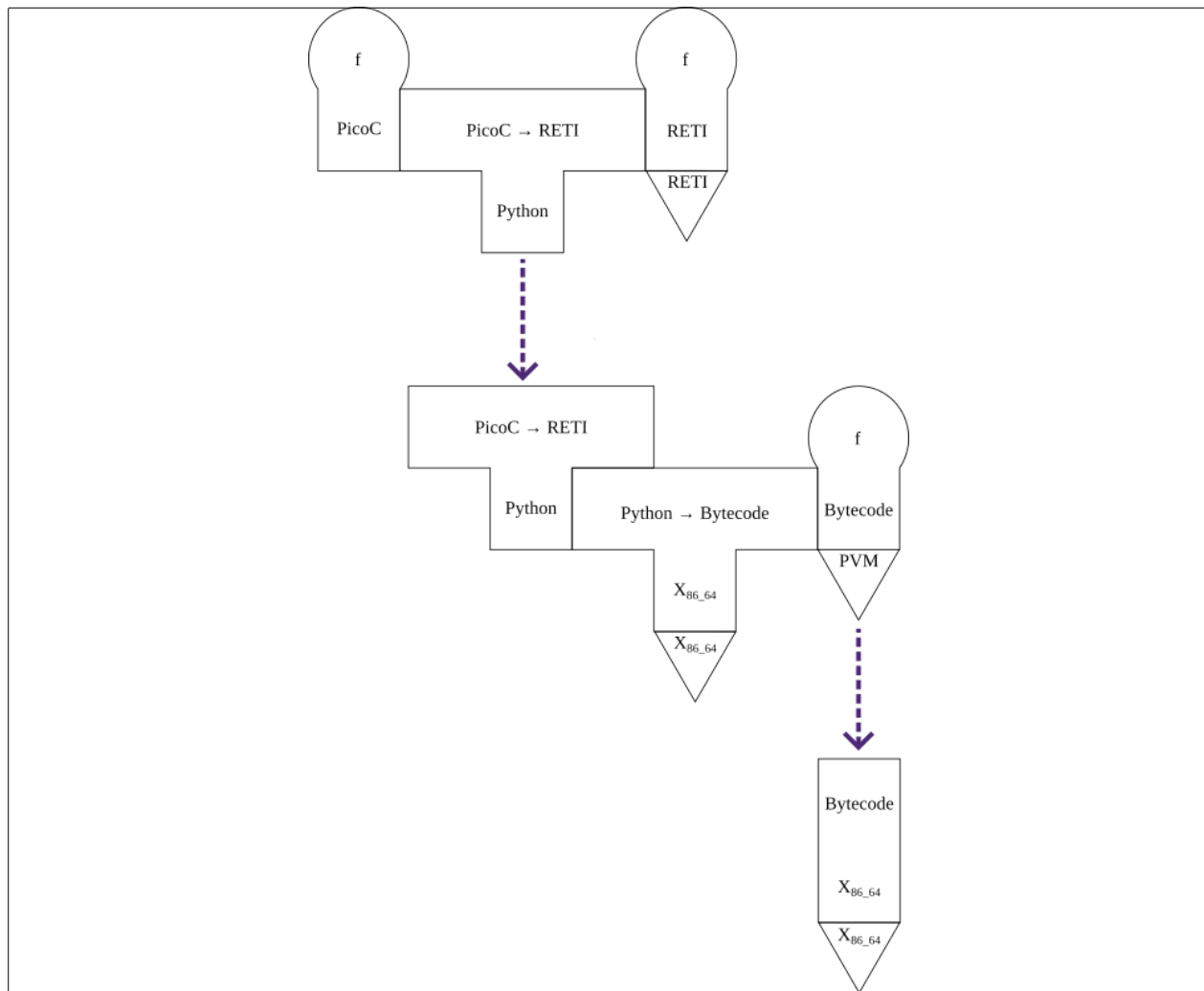


Abbildung 3.6: *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagramm** in Abbildung 3.6 lässt sich zusammenfassen, sodass man das **T-Diagramm** in Abbildung 3.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in X86_64-Maschinensprache geschrieben ist.

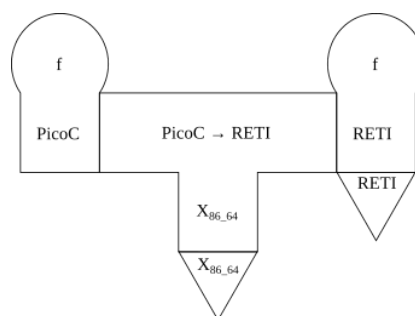


Abbildung 3.7: *Cross-Compiler Kompiliervorgang Kurzform*

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

der Kompilierprozess im Folgenden im **horizontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 3.8 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 2.32) des **PicoC-Compilers** aufeinanderfolgen. In der **Pipe-Architektur** nutzt der jeweils nächste **Pass** den generierten **Abstract Syntax Tree** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstract Syntax Tree** in seiner eigenen **Sprache** zu generieren.

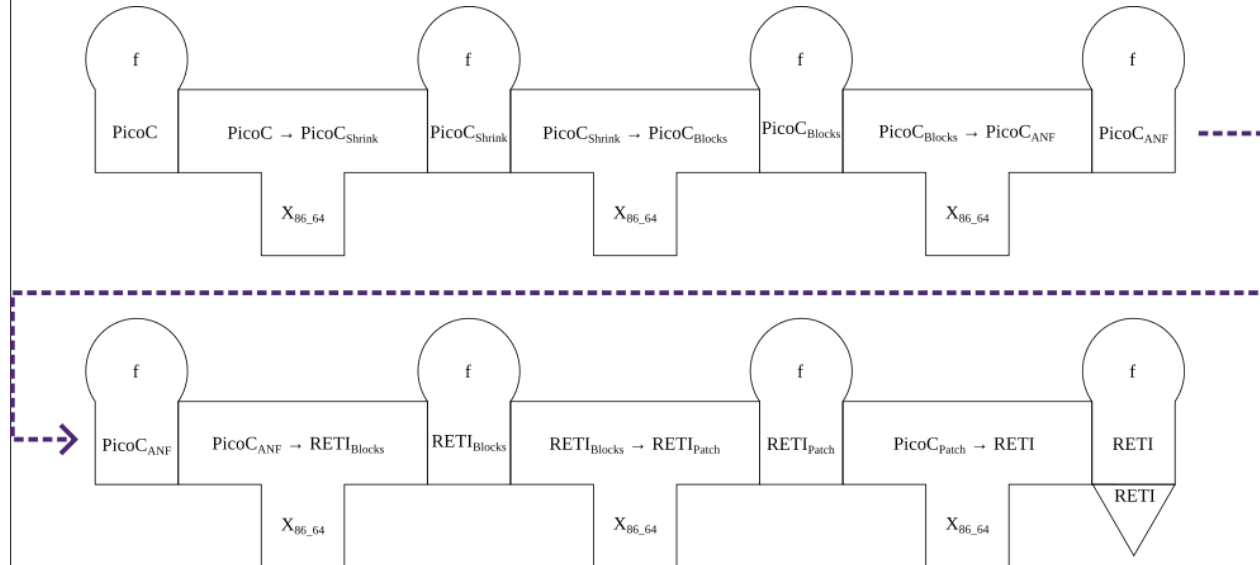


Abbildung 3.8: Architektur mit allen Passes ausgeschrieben

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

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird¹⁸. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem **Link**¹⁹ finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind²⁰.

3.3.1 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der verschiedenen Aspekte²¹ der

¹⁸Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

¹⁹https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.

²⁰Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstract Syntax Trees** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

²¹In kurz: **Pointer**, **Arrays**, **Structs** und **Funktionen**.

Bachelorarbeit wichtig.

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

3.3.1.1 PicoC-Shrink Pass

3.3.1.1.1 Aufgabe

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

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

3.3.1.1.2 Abstrakte Syntax

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

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp(<exp>, <bin_op>, <exp>)</i> <i>UnOp(<un_op>, <exp>)</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i>	
<i>stmt</i>	::=	<i>Exp(<exp>)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(<exp>, <rel>, <exp>)</i> <i>ToBool(<exp>)</i>	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref(<exp>, <exp>)</i> <i>Ref(<exp>)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, <exp>)</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>StructAssign(Name(str), <exp>)+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(<exp>, <stmt>*)</i> <i>IfElse(<exp>, <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(<exp>, <stmt>*)</i> <i>DoWhile(<exp>, <stmt>*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(<exp>)</i>	
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*)</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))**, <stmt>*)</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>

Grammar 3.3.1: Abstrakte Syntax der Sprache *L_{PiocShrink}*

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

3.3.1.1.3 Codebeispiel

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

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

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

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter [Link²²](#), unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

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

```

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

```

Code 3.6: PicoC Code für Codebeispiel

In Code 3.7 sieht man den **Abstract Syntax Tree**, der in der **Syntaktischen Analyse** generiert wurde.

```

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

```

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

```

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

```

Code 3.7: Abstract Syntax Tree für Codebeispiel

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

3.3.1.2 PicoC-Blocks Pass

3.3.1.2.1 Aufgabe

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

3.3.1.2.2 Abstrakte Syntax

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

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp(<exp>, <bin_op>, <exp>)</i> <i>UnOp(<un_op>, <exp>)</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i>	
<i>stmt</i>	::=	<i>Exp(<exp>)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(<exp>, <rel>, <exp>)</i> <i>ToBool(<exp>)</i>	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Ref(<exp>)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, <exp>)</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>StructAssign(Name(str), <exp>)+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(<exp>, <stmt>*)</i> <i>IfElse(<exp>, <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(<exp>, <stmt>*)</i> <i>DoWhile(<exp>, <stmt>*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(<exp>)</i>	
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))*)</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))**, <block>*)</i>	
<i>block</i>	::=	<i>Block(Name(str), <stmt>*)</i>	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo(Name(str))</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>

Grammar 3.3.2: Abstrakte Syntax der Sprache *L_Pioc_Blocks*

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

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

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

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

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

3.3.1.2.3 Codebeispiel

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

```

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

```

```

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

```

Code 3.8: PicoC-Blocks Pass für Codebeispiel

3.3.1.3 PicoC-ANF Pass

3.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** ist es den **Abstract Syntax Tree** der Sprache L_{PicoC_Blocks} in die **Abstrakte Syntax** der Sprache L_{PicoC_ANF} umzuformen, welche in **A-Normalform** (Definition 2.53) und damit auch in **Monadischer Normalform** (Definition 2.49) ist. Um Wiederholung zu vermeiden wird zur Erklärung der **A-Normalform** auf Unterkapitel 2.5.2 verwiesen.

Zudem wird eine **Symboltabelle** (Definition 3.10) eingeführt. In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** zugewiesen, die dem Wert einer von zwei **Countern** `rel_global_addr` und `rel_stack_addr` entspricht. Der Counter `rel_global_addr` ist für Variablen in den **Globalen Statischen Daten** und der Counter `rel_stack_addr` ist für Variablen auf dem **Stackframe**. Einer der beiden **Counter** wird entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle diese Variable `Name('symbol')` vor, so wird mit dem **Symbol**²³ als Schlüssel in der **Symboltabelle** nachgeschlagen und anstelle des `Name(str)`-Knotens die in der **Symboltabelle** nachgeschlagene Adresse in einem `Global(Num('addr'))`- bzw. `Stackframe(Num('addr'))`-Knoten eingesetzt eingefügt. Ob der `Global(Num('addr'))`- oder der `Stackframe(Num('addr'))`-Knoten zum Einsatz kommt, entscheidet sich anhand des **Scopes** (z.B. `@scope`), der in der **Symboltabelle** an den **Bezeichner** drangehängt ist (z.B. `identifizier@scope`).²⁴

Definition 3.10: Symboltabelle

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

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

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

3.3.1.3.2 Abstrakte Syntax

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

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

²³Bzw. der **Bezeichner**

²⁴Die Umsetzung von **Scopes** wird in Unterkapitel ?? genauer beschrieben.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i> <i>RETIComment()</i>	<i>L_Comment</i>	
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>	
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>		
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>Global(Num(str))</i> <i>Stackframe(Num(str))</i> <i>Stack(Num(str))</i> <i>BinOp(Stack(Num(str)), <bin_op>, Stack(Num(str)))</i> <i>UnOp(<un_op>, Stack(Num(str)))</i> <i>Call(Name('input'), Empty())</i> <i>Call(Name('print'), <exp>)</i> <i>Exp(<exp>)</i>		
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>	
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>		
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>		
<i>exp</i>	::=	<i>Atom(Stack(Num(str)), <rel>, Stack(Num(str)))</i> <i>ToBool(Stack(Num(str)))</i>		
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>	
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>		
<i>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>		
<i>stmt</i>	::=	<i>Assign(Global(Num(str)), Stack(Num(str)))</i> <i>Assign(Stackframe(Num(str)), Stack(Num(str)))</i> <i>Assign(Stack(Num(str)), Global(Num(str)))</i> <i>Assign(Stack(Num(str)), Stackframe(Num(str)))</i>		
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i> <i>Ref(Global(str))</i> <i>Ref(Stackframe(str))</i> <i>Ref(Subscr(<exp>, <exp> Ref(Attr(<exp>, Name(str))))</i>	<i>L_Pntr</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>	
<i>exp</i>	::=	<i>Subscr(<exp>, Stack(Num(str)))</i> <i>Array(<exp>+)</i>		
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>	
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>Struct(Assign(Name(str), <exp>)+)</i>		
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>		
<i>stmt</i>	::=	<i>IfElse(Stack(Num(str)), <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>	
<i>exp</i>	::=	<i>Call(Name(str), <exp>*)</i>	<i>L_Fun</i>	
<i>stmt</i>	::=	<i>StackMalloc(Num(str))</i> <i>NewStackframe(Name(str), GoTo(str))</i> <i>Exp(GoTo(Name(str)))</i> <i>RemoveStackframe()</i> <i>Return(Empty())</i> <i>Return(<exp>)</i>		
<i>decl_def</i>	::=	<i>FunDecl(<datatype>, Name(str))</i> <i>Alloc(Writeable(), <datatype>, Name(str))*</i> <i>FunDef(<datatype>, Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))* <block>*)</i>		
<i>block</i>	::=	<i>Block(Name(str), <stmt>*)</i>	<i>L_Blocks</i>	
<i>stmt</i>	::=	<i>GoTo(Name(str))</i>		
<i>file</i>	::=	<i>File(Name(str), <block>*)</i>	<i>L_File</i>	
<i>symbol_table</i>	::=	<i>SymbolTable(<symbol>*)</i>	<i>L_Symbol_Table</i>	
<i>symbol</i>	::=	<i>Symbol(<type_qual>, <datatype>, <name>, <val>, <pos>, <size>)</i>		
<i>type_qual</i>	::=	<i>Empty()</i>		
<i>datatype</i>	::=	<i>BuiltIn()</i> <i>SelfDefined()</i>		
<i>name</i>	::=	<i>Name(str)</i>		
<i>val</i>	::=	<i>Num(str)</i> <i>Empty()</i>		
<i>pos</i>	::=	<i>Pos(Num(str), Num(str))</i> <i>Empty()</i>		
<i>size</i>	::=	<i>Num(str)</i> <i>Empty()</i>		

3.3.1.3.3 Codebeispiel

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

```

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

```

```

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

```

Code 3.9: PicoC-ANF Pass für Codebespiel

3.3.1.4 RETI-Blocks Pass

3.3.1.4.1 Aufgabe

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

3.3.1.4.2 Abstrakte Syntax

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

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

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

Grammar 3.3.4: Abstrakte Syntax der Sprache L_{RETI_Blocks}

3.3.1.4.3 Codebeispiel

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

```

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

```

```

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

```

```

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

```

```

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

```

Code 3.10: RETI-Blocks Pass für Codebeispiel

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

3.3.1.5 RETI-Patch Pass

3.3.1.5.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist das **Ausbessern** (engl. to patch) des **Abstract Syntax Trees**, 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.²⁵
- das Überprüfen darauf, ob bestimmte **Immediates Im(str)** in Befehlen, wie z.B. **Jump(rel, Im(str)), Instr(Loadin(), [reg, reg, Im(str)]), Instr(Loadi(), [reg, Im(str)])** usw. **kleiner** -2^{21} oder **größer** $2^{21} - 1$ sind. Im Fall dessen, dass es so ist, muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **bitweise Oder** berechnet werden. Im Fall, dessen, dass der **Immediate** allerdings **kleiner** $-(2^{31})$ oder **größer** $2^{31} - 1$ ist, wird eine Fehlermeldung **TooLargeLiteral** ausgegeben.

3.3.1.5.2 Abstrakte Syntax

²⁵Das fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

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

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

Grammar 3.3.5: Abstrakte Syntax der Sprache L_{RETI_Patch}

3.3.1.5.3 Codebeispiel

In Code 3.11 sieht man den **Abstract-Syntax-Tree** des **Pioco-Patch Passes** für das aus Unterkapitel 3.6 weitergeführte Beispiel. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>-Block`²⁶ eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-**Statements** entfernt²⁷, die nur einen Sprung um **eine** Position entsprachen hätten.

```

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

```

²⁶Dieser **Block** wurde im Code 3.8 markiert.

²⁷Diese **entfernten GoTo(Name(str))**'s wurden ebenfalls im Code 3.8 markiert.

```

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

```

```

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

```

```

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

```

Code 3.11: RETI-Patch Pass für Codebeispiel

3.3.1.6 RETI Pass

3.3.1.6.1 Aufgabe

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

3.3.1.6.2 Konkrete und Abstrakte Syntax

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

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

<i>dig_no_0</i>	::=	"1" "2" "3" "4" "5" "6"	<i>L_Program</i>
		"7" "8" "9"	
<i>dig_with_0</i>	::=	"0" <i>dig_no_0</i>	
<i>num</i>	::=	"0" <i>dig_no_0</i> <i>dig_with_0</i> * "-" <i>dig_no_0</i> *	
<i>letter</i>	::=	"a"..."Z"	
<i>name</i>	::=	<i>letter</i> (<i>letter</i> <i>dig_with_0</i> _)*	
<i>reg</i>	::=	"ACC" "IN1" "IN2" "PC" "SP"	
		"BAF" "CS" "DS"	
<i>arg</i>	::=	<i>reg</i> <i>num</i>	
<i>rel</i>	::=	"==" "!=" "<" "<=" ">"	
		">=" "_NOP"	

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

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

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

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

Grammar 3.3.8: Abstrakte Syntax der Sprache L_{RETI}

3.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkreten Syntax** ausgegeben werden, wie in Code 3.12 zu sehen ist. Es gibt **keine Blöcke** mehr und die **RETI-Befehle** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** ausschließlich aus **RETI-Ausdrücken** bestehen²⁸, 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 **Abstract Syntax Tree**, wenn er in eine Datei ausgegeben wird, direkt **RETI-Code** in **menschenlesbarer Repräsentation** erzeugt.

```

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

```

²⁸Wie 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;
```

```
100 # StackMalloc(Num('2'))
101 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;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6'))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 3.12: *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.lysator.liu.se/c/ANSI-C-grammar-1.html> (besucht am 29.07.2022).
- *ANSI C grammar (Yacc)*. URL: <http://www.quut.com/c/ANSI-C-grammar-y.html> (besucht am 29.07.2022).
- *ANTLR*. URL: <https://www.antlr.org/> (besucht am 31.07.2022).
- *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).
- *Parsing Expressions · Crafting Interpreters*. URL: <https://www.craftinginterpreters.com/parsing-expressions.html> (besucht am 09.07.2022).
- *Transformers & Visitors — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/visitors.html> (besucht am 09.07.2022).
- *Welcome to Lark's documentation! — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/> (besucht am 31.07.2022).

- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *What is 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).

Artikel

- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: [10.1145/355598.362740](https://doi.org/10.1145/355598.362740).

Vorlesungen

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09.07.2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- 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/swtv1> (besucht am 19.07.2022).

Sonstige Quellen

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

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).
- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: <https://github.com/lark-parser/lark> (besucht am 31.07.2022).
- *Syntax*. In: *Wiktionary*. Page Version ID: 9196998. 7. Juni 2022. URL: <https://de.wiktionary.org/w/index.php?title=Syntax&oldid=9196998> (besucht am 31.07.2022).