

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

*Abgabedatum:* 28<sup>th</sup> 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

<b>1</b>	<b>Einführung</b>	<b>10</b>
1.1	Compiler und Interpreter	10
1.1.1	T-Diagramme	13
1.2	Formale Sprachen	15
1.2.1	Mehrdeutige Grammatiken	16
1.2.2	Präzidenz und Assoziativität	17
1.3	Lexikalische Analyse	17
1.4	Syntaktische Analyse	20
1.5	Code Generierung	26
1.6	Fehlermeldungen	27
1.6.1	Kategorien von Fehlermeldungen	27
<b>2</b>	<b>Implementierung</b>	<b>28</b>
2.1	Lexikalische Analyse	28
2.1.1	Teil der Konkreten Syntax für die Lexikalische Analyse	28
2.1.2	Basic Lexer	29
2.2	Syntaktische Analyse	29
2.2.1	Teil der Konkreten Syntax für die Syntaktische Analyse	29
2.2.2	Umsetzung von Präzidenz	31
2.2.3	Derivation Tree Generierung	32
2.2.3.1	Early Parser	32
2.2.3.2	Codebeispiel	32
2.2.4	Derivation Tree Vereinfachung	33
2.2.4.1	Visitor	33
2.2.4.2	Codebeispiel	33
2.2.5	Abstrakt Syntax Tree Generierung	35
2.2.5.1	PicoC-Knoten	35
2.2.5.2	RETI-Knoten	40
2.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	41
2.2.5.4	Abstrakte Syntax	43
2.2.5.5	Transformer	45
2.2.5.6	Codebeispiel	45
2.3	Code Generierung	46
2.3.1	Übersicht	46
2.3.2	Passes	48
2.3.2.1	PicoC-Shrink Pass	49
2.3.2.1.1	Aufgabe	49
2.3.2.1.2	Codebeispiel	49
2.3.2.2	PicoC-Blocks Pass	51
2.3.2.2.1	Aufgabe	51
2.3.2.2.2	Abstrakte Syntax	51
2.3.2.2.3	Codebeispiel	51
2.3.2.3	PicoC-Mon Pass	53
2.3.2.3.1	Aufgabe	53
2.3.2.3.2	Abstrakte Syntax	53
2.3.2.3.3	Codebeispiel	53

2.3.2.4	RETI-Blocks Pass . . . . .	55
2.3.2.4.1	Aufgaben . . . . .	55
2.3.2.4.2	Abstrakte Syntax . . . . .	55
2.3.2.4.3	Codebeispiel . . . . .	55
2.3.2.5	RETI-Patch Pass . . . . .	58
2.3.2.5.1	Aufgaben . . . . .	58
2.3.2.5.2	Abstrakte Syntax . . . . .	58
2.3.2.5.3	Codebeispiel . . . . .	58
2.3.2.6	RETI Pass . . . . .	61
2.3.2.6.1	Aufgaben . . . . .	61
2.3.2.6.2	Konkrete und Abstrakte Syntax . . . . .	61
2.3.2.6.3	Codebeispiel . . . . .	62

---

---

# Abbildungsverzeichnis

1.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	15
1.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	15
1.3	Veranschaulichung der Lexikalischen Analyse . . . . .	20
1.4	Veranschaulichung der Syntaktischen Analyse . . . . .	25
2.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	47
2.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	47
2.3	Architektur mit allen Passes ausgeschrieben . . . . .	48

---

---

# Codeverzeichnis

2.1	PicoC Code für Derivation Tree Generierung . . . . .	32
2.2	Derivation Tree nach Derivation Tree Generierung . . . . .	33
2.3	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	34
2.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert . . . . .	45
2.5	PicoC Code für Codebeispiel . . . . .	50
2.6	Abstract Syntax Tree für Codebeispiel . . . . .	50
2.7	PicoC-Blocks Pass für Codebeispiel . . . . .	52
2.8	PicoC-Mon Pass für Codebeispiel . . . . .	55
2.9	RETI-Blocks Pass für Codebeispiel . . . . .	58
2.10	RETI-Patch Pass für Codebeispiel . . . . .	61
2.11	RETI Pass für Codebeispiel . . . . .	65

---

---

# Tabellenverzeichnis

2.1	Präzidenzregeln von PicoC . . . . .	32
2.2	PicoC-Knoten Teil 1 . . . . .	35
2.3	PicoC-Knoten Teil 2 . . . . .	36
2.4	PicoC-Knoten Teil 3 . . . . .	37
2.5	PicoC-Knoten Teil 4 . . . . .	38
2.6	RETI-Knoten . . . . .	40
2.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung . . . . .	42

---

---

# Definitionsverzeichnis

1.1	Interpreter . . . . .	10
1.2	Compiler . . . . .	10
1.3	Maschiensprache . . . . .	11
1.4	Assemblersprache (bzw. engl. Assembly Language) . . . . .	11
1.5	Assembler . . . . .	12
1.6	Objectcode . . . . .	12
1.7	Linker . . . . .	12
1.8	Immediate . . . . .	12
1.9	Transpiler (bzw. Source-to-source Compiler) . . . . .	13
1.10	Cross-Compiler . . . . .	13
1.11	T-Diagram Programm . . . . .	13
1.12	T-Diagram Übersetzer (bzw. eng. Translator) . . . . .	14
1.13	T-Diagram Interpreter . . . . .	14
1.14	T-Diagram Maschine . . . . .	14
1.15	Sprache . . . . .	15
1.16	Chromsky Hierarchie . . . . .	15
1.17	Grammatik . . . . .	16
1.18	Reguläre Sprachen . . . . .	16
1.19	Kontextfreie Sprachen . . . . .	16
1.20	Ableitung . . . . .	16
1.21	Links- und Rechtsableitung . . . . .	16
1.22	Linksrekursive Grammatiken . . . . .	16
1.23	Ableitungsbaum . . . . .	16
1.24	Mehrdeutige Grammatik . . . . .	17
1.25	Assoziativität . . . . .	17
1.26	Präzidenz . . . . .	17
1.27	Wortproblem . . . . .	17
1.28	LL(k)-Grammatik . . . . .	17
1.29	Pipe-Filter Architekturpattern . . . . .	18
1.30	Pattern . . . . .	18
1.31	Lexeme . . . . .	18
1.32	Lexer (bzw. Scanner oder auch Tokenizer) . . . . .	18
1.33	Bezeichner (bzw. Identifier) . . . . .	19
1.34	Literal . . . . .	20
1.35	Konkrete Syntax . . . . .	21
1.36	Derivation Tree (bzw. Parse Tree) . . . . .	21
1.37	Parser . . . . .	21
1.38	Recognizer (bzw. Erkennen) . . . . .	22
1.39	Transformer . . . . .	23
1.40	Visitor . . . . .	23
1.41	Abstrakte Syntax . . . . .	24
1.42	Abstract Syntax Tree (AST) . . . . .	24
1.43	Pass . . . . .	26
1.44	Monadische Normalform . . . . .	27
1.45	Fehlermeldung . . . . .	27
2.1	Label . . . . .	39
2.2	Location . . . . .	39



2.3	Token-Knoten . . . . .	39
2.4	Container-Knoten . . . . .	39
2.5	Symboltabelle . . . . .	53

---

---

# Grammatikverzeichnis

2.1.1 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF, Teil 1	28
2.1.2 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF, Teil 2	29
2.2.1 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 1	30
2.2.2 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 2	31
2.2.3 Abstrakte Syntax der Sprache $L_{PicoC}$	44
2.3.1 Abstrakte Syntax für $L_{PicoC\_Blocks}$	51
2.3.2 Abstrakte Syntax für $L_{PicoC\_Mon}$	53
2.3.3 Abstrakte Syntax für $L_{RETI\_Blocks}$	55
2.3.4 Abstrakte Syntax für $L_{RETI\_Patch}$	58
2.3.5 Konkrete Syntax für $L_{RETI\_Lex}$	62
2.3.6 Konkrete Syntax für $L_{RETI\_Parse}$	62
2.3.7 Abstrakte Syntax für $L_{RETI}$	62

# 1 Einführung

## 1.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 1.2) und eines **Interpreters** (Definition 1.1), 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**<sup>1</sup> und von **Tests** die **Beziehungen** in 1.43.1 zu belegen (siehe Subkapitel ??).

### Definition 1.1: Interpreter

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

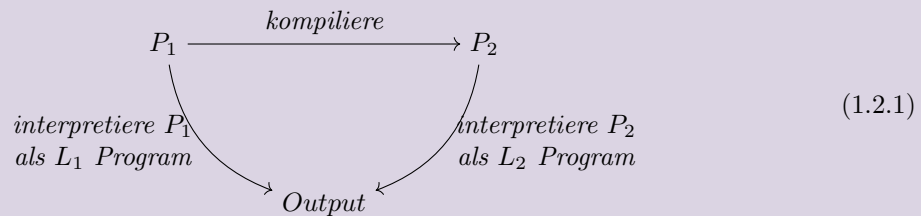
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstract Syntax Tree** (Definition 1.42) und führt je nach Komposition der **Nodes** des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

### Definition 1.2: 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.<sup>a</sup>*



<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

<sup>1</sup>Sammlung 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 1.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 1.9) oder **Cross-Compiler** (Definition 1.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 1.4) voneinander zu unterscheiden.

### Definition 1.3: 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**.<sup>a,b</sup>*

<sup>a</sup>Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 1.8) haben.

<sup>b</sup>C. Scholl, „Betriebssysteme“.

### Definition 1.4: Assemblersprache (bzw. engl. Assembly Language)

*Eine sehr **hardwarenahe** Programmiersprache, deren **Instructions** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen<sup>a</sup> 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<sup>b</sup> der Instructions und drumherum<sup>c,d</sup>.*

<sup>a</sup>Instructions der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

<sup>b</sup>Z.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.

<sup>c</sup>Z.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.

<sup>d</sup>P. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 1.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenenncode** bzw. **Objectcode** (Definition 1.6) 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 1.7) zu Maschienncode zusammengesetzt wird ausführbar

ist.

#### Definition 1.5: Assembler

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

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

#### Definition 1.6: 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.<sup>a</sup>

<sup>a</sup>P. Scholl, „Einführung in Embedded Systems“.

#### Definition 1.7: 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.<sup>a</sup>

<sup>a</sup>P. 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 1.8) 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 Umsetzung<sup>2</sup>.

#### Definition 1.8: 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.<sup>a</sup>

<sup>a</sup>Ljohhuh, *What is an immediate value?*

<sup>2</sup>Eine **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 1.9: Transpiler (bzw. Source-to-source Compiler)**

Kompiliert zwischen Sprachen, die ungefähr auf dem *gleichen* Level an *Abstraktion* arbeiten<sup>ab</sup>

<sup>a</sup>Die 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**.

<sup>b</sup>Thiemann, „Compilerbau“.

**Definition 1.10: 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.<sup>ab</sup>

<sup>a</sup>Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler**  $C_{PicoC}^{Python}$ .

<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

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

**1.1.1 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 1.11), einen Übersetzer (Definition 1.12), einen Interpreter (Definition 1.13) und eine Maschine (Definition 1.14) zusammen.

**Definition 1.11: T-Diagram Programm**

Repräsentiert ein *Programm*, dass in der *Sprache*  $L_1$  geschrieben ist und die *Funktion*  $f$  berechnet.<sup>a</sup>



<sup>a</sup>Earley 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 1.11 also reichen einfach eine 1 hinzuschreiben.

<sup>3</sup>Die 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 1.12: 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**<sup>a</sup> die **Beziehungen** in 1.43.1.<sup>b</sup>



<sup>a</sup>Zwischen 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**.

<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 1.13: T-Diagramm Interpreter**

Repräsentiert einen **Interpreter**, der in der **Sprache**  $L_1$  geschrieben ist und **Programme** in der **Sprache**  $L_2$  interpretiert.<sup>a</sup>



<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 1.14: T-Diagramm Maschine**

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache**  $L_1$  ausführt.<sup>a,b</sup>

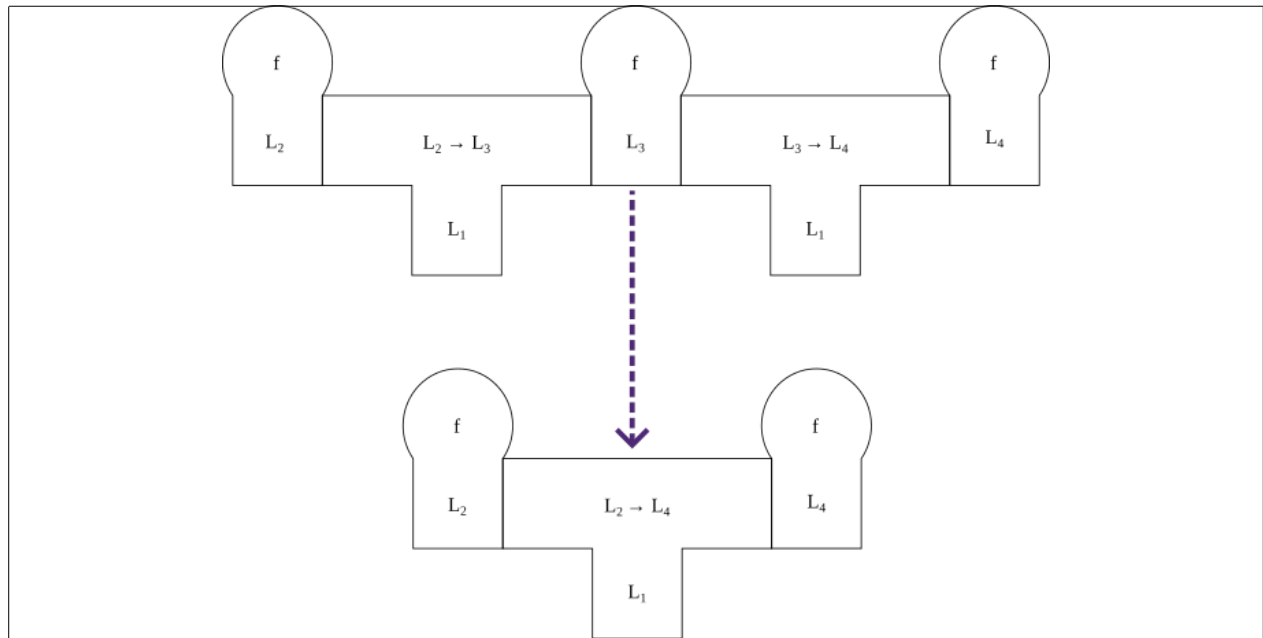


<sup>a</sup>Wenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).

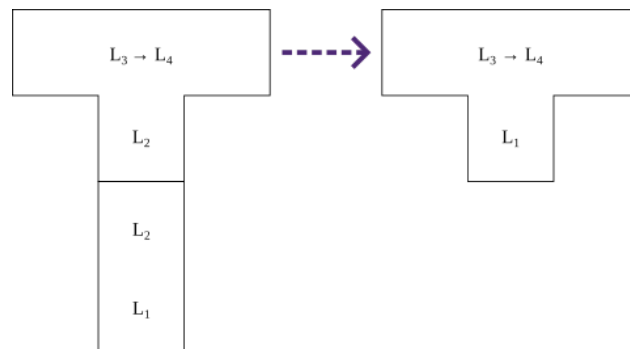
<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

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 1.1 und 1.2 erkennen kann zusammenfassen.



**Abbildung 1.1:** *Horizontale Übersetzungszwischenschritte zusammenfassen*



**Abbildung 1.2:** *Vertikale Interpretierungszwischenschritte zusammenfassen*

## 1.2 Formale Sprachen

### Definition 1.15: Sprache

*a*

<sup>a</sup>Nebel, „Theoretische Informatik“.

### Definition 1.16: Chomsky Hierarchie

*a*

<sup>a</sup>Nebel, „Theoretische Informatik“.



**Definition 1.17: Grammatik***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.18: Reguläre Sprachen***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.19: Kontextfreie Sprachen***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.20: Ableitung***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.21: Links- und Rechtsableitung***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.22: 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.<sup>a</sup>

<sup>a</sup>Parsing Expressions · Crafting Interpreters.**1.2.1 Mehrdeutige Grammatiken****Definition 1.23: Ableitungsbaum***a*<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 1.24: Mehrdeutige Grammatik***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**1.2.2 Präzidenz und Assoziativität****Definition 1.25: Assoziativität***a*<sup>a</sup>*Parsing Expressions · Crafting Interpreters.***Definition 1.26: Präzidenz***a*<sup>a</sup>*Parsing Expressions · Crafting Interpreters.***Definition 1.27: Wortproblem***a*<sup>a</sup>Nebel, „Theoretische Informatik“.**Definition 1.28: 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<sup>a</sup>. Dabei steht **LL** für *left-to-right* und *leftmost-derivation*, da das **Eingabewort** von **links nach rechts** geparsed und immer **Linksableitungen** genommen werden müssen<sup>b</sup>, damit die obige Bedingung mit den **nächsten**  $k$  Symbolen gilt.<sup>c</sup>

<sup>a</sup>Das wird auch als **Lookahead** von  $k$  bezeichnet.<sup>b</sup>Wobei sich das mit den **Linksableitungen** automatisch ergibt, wenn man das Eingabewort von **links-nach-rechts** parsed und jeder der nächsten  $k$  **Ableitungsschritte** eindeutig sein soll.<sup>c</sup>Nebel, „Theoretische Informatik“.**1.3 Lexikalische Analyse**

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb des **Pipe-Filter Architekturpatterns** (Definition 1.29) 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 1.30) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

**Definition 1.29: 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.<sup>a,b</sup>



<sup>a</sup>Das 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.

<sup>b</sup>Westphal, „Softwaretechnik“.

Diese Folgen endlicher Symbole werden auch **Lexeme** (Definition 1.31) genannt.

**Definition 1.30: 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<sup>a</sup>, die für die Beschreibung eines **Tokens**  $T$  zuständig sind.<sup>b</sup>

<sup>a</sup>Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

<sup>b</sup>Thiemann, „Compilerbau“.

**Definition 1.31: Lexeme**

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

<sup>a</sup>Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 1.32) 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 1.32) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

**Definition 1.32: Lexer (bzw. Scanner oder auch Tokenizer)**

Ein **Lexer** ist eine **partielle Funktion**  $lex : \Sigma^* \rightarrow (N \times W)^*$ , welche ein **Wort** bzw. **Lexeme** aus  $\Sigma^*$  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.<sup>a</sup>

<sup>a</sup>Thiemann, „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 1.33) von **Variablen, Konstanten und Funktionen** die Symbole<sup>a</sup>.

<sup>a</sup>Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel 2 **Symboltabelle** genannt wird.

### Definition 1.33: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres **Scopes eindeutig** benennt.<sup>a,b</sup>*

<sup>a</sup>Außer 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.

<sup>b</sup>Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>4</sup> und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen Zeichen das leere Wort  $\epsilon$  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`<sup>5</sup>, bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie `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 1.34) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99<sup>6</sup>. 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** vorausschaut<sup>7</sup>, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 2.1.1 liefert den Beweis.

<sup>4</sup>In 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.

<sup>5</sup>Diese **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.

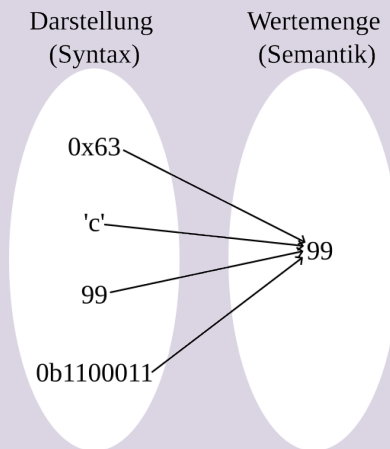
<sup>6</sup>Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

<sup>7</sup>Man nennt das auch einem **Lookahead** von 1

dass die Sprache  $L_{PicoC\_Lex}$  des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 1.18 erfüllt. Einzige die Produktion  $CHAR ::= ""ASCII\_CHAR""$  sieht problematisch aus, kann allerdings auch als  $\{CHAR ::= ""CHAR2, CHAR2 ::= ASCII\_CHAR""\}$  **regulär** ausgedrückt werden<sup>8</sup>. Somit existiert eine **reguläre Grammatik**, welche die **Sprache**  $L_{PicoC\_Lex}$  beschreibt und damit ist die **Sprache**  $L_{PicoC\_Lex}$  **regulär**.

#### Definition 1.34: Literal

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



<sup>a</sup>Thiemann, „Einführung in die Programmierung“.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 1.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

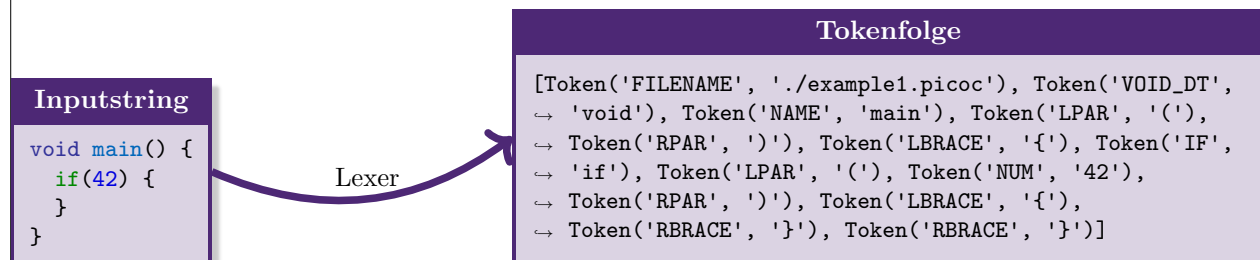


Abbildung 1.3: Veranschaulichung der Lexikalischen Analyse

## 1.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik**  $G_{Parse}$  notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern '(' bzw. öffnende geschweifte Klammern '{' es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden.

<sup>8</sup>Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten lassen.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 1.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 1.37), ein **Derivation Tree** (Definition 1.36) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 1.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

#### Definition 1.35: 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.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

#### Definition 1.36: 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.<sup>a</sup>*

<sup>a</sup>JSON parser - Tutorial — Lark documentation.

#### Definition 1.37: 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.<sup>a, b</sup>*

<sup>a</sup>Es 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 1.37 verwendet.

<sup>b</sup>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<sup>a</sup>.

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**<sup>b</sup>.

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.

<sup>a</sup>Das würde z.B. der Rolle eines **Semikolon** ; in der Sprache  $L_{PicoC}$  entsprechen.

<sup>b</sup>Z.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 1.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 1.38), denn ein Parser löst das **Wortproblem** (Definition 1.27) 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 1.38: 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<sup>a,b</sup>*

<sup>a</sup>Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

<sup>b</sup>Thiemann, „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.<sup>a</sup>

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 1.22) 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 1.28) 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.<sup>b</sup>

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.<sup>c</sup>

- **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.<sup>d</sup>
- **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.<sup>e</sup>

<sup>a</sup>What is Top-Down Parsing?

<sup>b</sup>Diese 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.

<sup>c</sup>Diese 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.

<sup>d</sup>What is Bottom-up Parsing?

<sup>e</sup>Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 1.39) und **Visitors** (Definition 1.40) generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inputstring von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 1.41).

#### Definition 1.39: 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.<sup>a</sup>

<sup>a</sup>Transformers & Visitors — Lark documentation.

#### Definition 1.40: 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.<sup>a,b</sup>

<sup>a</sup>Kann 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.

<sup>b</sup>Transformers & Visitors — Lark documentation.



**Definition 1.41: 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.<sup>a</sup>

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 1.42: 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.<sup>a</sup>

<sup>a</sup>G. 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 1.4 die Syntaktische mit dem Beispiel aus Subkapitel 1.3 fortgeführt.

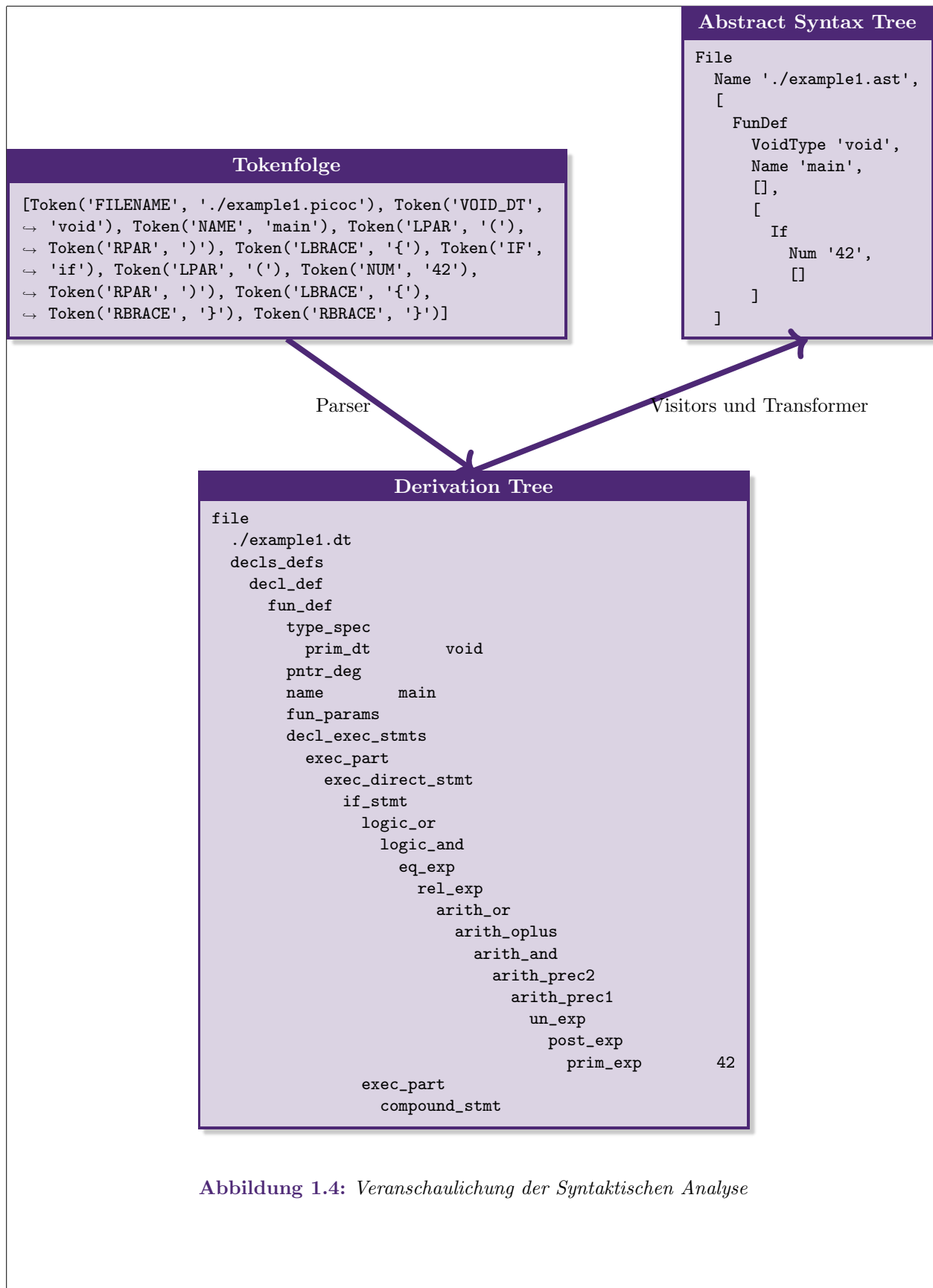


Abbildung 1.4: Veranschaulichung der Syntaktischen Analyse

## 1.5 Code Generierung

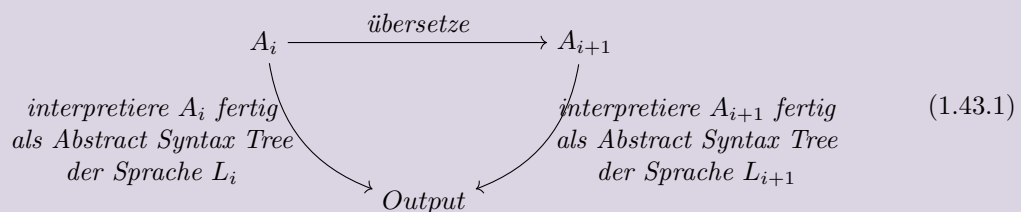
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstract Syntax Tree** einer Sprache  $L_1$  in den **Abstract Syntax Tree** einer Sprache  $L_2$  umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 1.43) nennt.

Man spricht hier von dem "**Abstrakt Syntax Tree einer Sprache  $L_1$  (bzw.  $L_2$ )**" und meint damit eine **Sprache**, die durch eine **Konkrete Syntax** beschrieben wird und nicht durch die **Abstrakte Syntax**, nach welcher der **Abstract Syntax Tree** abgeleitet ist. Hierbei handelt es sich um eine im Compilerbau geläufige **Redeart**, die vielleicht verwirrend sein kann. Man will mit der **ersten Redeart** ausdrücken, dass der **Abstrakt Syntax Tree** der zur Sprache  $L_1$  (bzw.  $L_2$ ) passend konstruierte **Abstract Syntax Tree** ist. Für die tatsächliche Sprache, die durch die **Abstrakten Syntax** beschrieben wird, interessiert man sich nicht explizit.

### Definition 1.43: 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.<sup>a,b</sup>*

Für jeden **Pass** gilt ähnlich wie beim **kompletten Compiler** in 1.43.1, dass:



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

<sup>a</sup> $L_i$  bzw.  $L_{i+1}$  sind hierbei keine **Sprachen**, die durch eine **Abstrakte Syntax** beschrieben werden, sondern Sprachen, die durch eine **Konkrete Syntax** beschrieben werden.

<sup>b</sup>Ein **Pass** kann mit einem **Transpiler** 1.9 (Definition 1.9) 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 auf **Abstract Syntax Trees**.

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

<sup>d</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Sobald das der Fall ist, kann aus dem finalen **Abstract Syntax Tree** sehr einfach die **Konkrete Syntax** generiert werden.

Die von den **Passes** generierten **Abstract Syntax Trees** werden dabei mit jedem **Pass** der **Syntax** des **RETI-Code's** immer ähnlicher werden. Jeder Pass sollte dabei möglichst eine Aufgabe übernehmen, da der Sinn von **Passes** ist, die Kompilierung in mehrere kleinschrittige Aufgaben runterzuberechnen. Wie es auch schon der Zweck des **Derivation Tree** in der Syntaktischen Analyse war, eine Zwischenstufe zum **Abstract Syntax Tree** darzustellen, aus der sich unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstract Syntax Tree** generieren lies.

**Definition 1.44: Monadische Normalform**

Eine Sprache bei der<sup>a</sup>

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein echter Compiler verwendet Graph Coloring ... Register ...

## 1.6 Fehlermeldungen

**Definition 1.45: 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 Konkreten Syntax **ableiten** lässt oder auf etwas **zugegriffen** werden soll, was noch **nicht** deklariert oder definiert wurde.
2. Beim Ausführen eine **verbotene** Operation ausgeführt wurde.<sup>a</sup>

<sup>a</sup>*Errors in C/C++ - GeeksforGeeks.*

### 1.6.1 Kategorien von Fehlermeldungen

# 2 Implementierung

## 2.1 Lexikalische Analyse

### 2.1.1 Teil der Konkreten Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	"//"/ $[\backslash n]^*/$   $"/^*/$ $/(. \backslash n)^*/$ $"/^*$	<i>L_Comment</i>
<i>RETI.COMMENT.2</i>	::=	$"/^*/$ $["'?"\#]/[\backslash n]^*/$	
<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</i> <i>DIG.WITH_0</i> *	
<i>ASCII.CHAR</i>	::=	$["'.." \sim ]$	
<i>CHAR</i>	::=	$["'ASCII.CHAR"]$	
<i>FILENAME</i>	::=	<i>ASCII.CHAR</i> + ".picoc"	
<i>LETTER</i>	::=	$"a".."z"   "A".."Z"$	
<i>NAME</i>	::=	$(LETTER   "_")$ $(LETTER \mid DIG.WITH_0 \mid "_")^*$	
<i>name</i>	::=	<i>NAME</i>   <i>INT.NAME</i>   <i>CHAR.NAME</i>   <i>VOID.NAME</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>LOGIC.NOT</i>	::=	$"!"$	

**Grammar 2.1.1:** Teil der Konkreten Syntax der Sprache  $L_{Picoc}$  für die Lexikalische Analyse in EBNF, Teil 1

<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>   "_")+	
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   "_")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   "_")+	
<i>prim_dt</i>	::=	<i>INT_DT</i>   <i>CHAR_DT</i>   <i>VOID_DT</i>	

**Grammar 2.1.2:** Teil der Konkretten Syntax der Sprache  $L_{\text{PicoC}}$  für die Lexikalische Analyse in EBNF, Teil 2

### 2.1.2 Basic Lexer

## 2.2 Syntaktische Analyse

### 2.2.1 Teil der Konkretten Syntax für die Syntaktische Analyse

In 2.2.1

<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>array_subscr</i>   <i>struct_attr</i>   <i>fun_call</i>	<i>L_Array</i> +
		<i>input_exp</i>   <i>print_exp</i>   <i>prim_exp</i>	<i>L_Pntr</i> +
<i>un_exp</i>	::=	<i>un_opun_exp</i>   <i>post_exp</i>	<i>L_Struct</i> + <i>L_Fun</i>
<i>input_exp</i>	::=	"input" "(" ")"	<i>L_Arith</i>
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i>   <i>un_exp</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_oprel_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</i> <i>pnter_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</i> <i>name</i> "=" <i>NUM</i> ";"	
<i>pnter_deg</i>	::=	"*" "*" *	<i>L_Pntr</i>
<i>pnter_decl</i>	::=	<i>pnter_deg</i> <i>array_decl</i>   <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]" ) *	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name</i> <i>array_dims</i>   "(" <i>pnter_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 2.2.1:** Teil der Konkreten Syntax der Sprache  $L_{\text{Picoc}}$  für die Syntaktische Analyse in EBNF, Teil 1

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

**Grammar 2.2.2:** Teil der Konkretten Syntax der Sprache  $L_{PicoC}$  für die Syntaktische Analyse in EBNF, Teil 2

## 2.2.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**<sup>1</sup>. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 2.1 aufgelistet.

<sup>1</sup>[C Operator Precedence - cppreference.com.](http://c.operatorprecedence.com/)



Präzidenz	Operator	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	
12	a=b	Zuweisung	Rechts, dann links ←
13	a,b	Komma	Links, dann rechts →

Tabelle 2.1: Präzidenzregeln von PicoC

## 2.2.3 Derivation Tree Generierung

### 2.2.3.1 Early Parser

### 2.2.3.2 Codebeispiel

```

1 struct st {int *(*attr)[5][6];};
2
3 void main() {
4     struct st *(*var)[3][2];
5 }

```

Code 2.1: PicoC Code für Derivation Tree Generierung

```

1 file
2   ./example_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         5
21         6
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     array_dims
45     3
46     2

```

**Code 2.2:** *Derivation Tree nach Derivation Tree Generierung*

## 2.2.4 Derivation Tree Vereinfachung

### 2.2.4.1 Visitor

### 2.2.4.2 Codebeispiel

Beispiel aus Subkapitel 2.2.3.2 wird fortgeführt.

```

1 file
2 ./example_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     pntr_decl

```

```

10         ptr_deg *
11         array_decl
12         array_dims
13         5
14         6
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     3
38     2
39     ptr_decl
40     ptr_deg *
41     array_decl
42     array_dims
43     type_spec
44     struct_spec
45         name st
46     name var

```

**Code 2.3:** *Derivation Tree nach Derivation Tree Vereinfachung*

## 2.2.5 Abstrakt Syntax Tree Generierung

### 2.2.5.1 PicoC-Knoten

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

Tabelle 2.2: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen <b>beliebigen Ausdruck</b> , dessen Ergebnis auf den <b>Stack</b> soll. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Stack(num)	Container, der für das <b>temporäre</b> Ergebnis einer Berechnung, das <b>num</b> Speicherzellen relativ zum <b>Stackpointer Register SP</b> steht.
Stackframe(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Begin-Aktive-Funktion Register BAF</b> steht.
Global(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Datensegment Register DS</b> steht.
StackMalloc(num)	Container, der für das <b>Allokieren</b> von <b>num</b> Speicherzellen auf dem <b>Stack</b> steht.
PntrDecl(num, datatype)	Container, der für den <b>Pointerdatatype</b> steht: <code>&lt;prim_dt&gt; *&lt;var&gt;</code> , wobei das <b>Attribut</b> <b>num</b> die <b>Anzahl zusammengefasster Pointer</b> angibt und <b>datatype</b> der Datentyp ist, auf den der oder die <b>Pointer</b> zeigen.
Ref(exp, datatype, error_data)	Container, der für die Anwendung des <b>Referenz-Operators</b> <code>&amp;&lt;var&gt;</code> steht und die <b>Adresse</b> einer <b>Location</b> (Definition 2.2) auf den Stack schreiben soll, die über <b>exp</b> eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Deref(lhs, exp)	Container für den <b>Indezzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> , <code>Attr(exp, name)</code> oder ein <code>Name('var')</code> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den <b>Arraydatatype</b> steht: <code>&lt;prim_dt&gt; &lt;var&gt;[&lt;i&gt;]</code> , wobei das <b>Attribut</b> <b>nums</b> eine Liste von <code>Num('x')</code> ist, die die <b>Dimensionen</b> des Arrays angibt und <b>datatype</b> der Datentyp ist, der über das Anwenden von <code>Subscript()</code> auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den <b>Initializer</b> eines <b>Arrays</b> , dessen Einträge <b>exps</b> weitere Initializer für eine <b>Array-Dimension</b> oder ein Initializer für ein <b>Struct</b> oder ein <b>Logischer Ausdruck</b> sein können, z.B. <code>{1, 2}</code> , <code>{3, 4}</code> . Des Weiteren besitzt er ein verstecktes Attribut <b>datatype</b> , welches für den <b>PicoC-Mon Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
Subscr(exp1, exp2)	Container für den <b>Indezzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten <b>Structdatatype</b> : <code>struct &lt;name&gt;</code> , wobei das <b>Attribut</b> <b>name</b> festlegt, welchen <b>selbst definierte</b> Structdatatype dieser Container-Knoten repräsentiert.
Attr(exp, name)	Container für den <b>Attributzugriff</b> auf einen <b>Structdatatype</b> : <code>&lt;var&gt;.&lt;attr&gt;</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> sein kann und <b>name</b> das Attribut ist, auf das zugegriffen werden soll.

Tabelle 2.3: PicoC-Knoten Teil 2

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

Tabelle 2.4: PicoC-Knoten Teil 3

Piocc-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts_blocks)	Container für eine <b>Funktionsdefinition</b> , z.B. <datatype> <fun_name>(<datatype> <param>) {<stmts>}, wobei datatype der <b>Rückgabewert</b> der Funktion ist, name der <b>Bezeichner</b> der Funktion ist, allocs die <b>Parameter</b> der Funktion sind, wobei der <b>Container-Knoten</b> Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts_blocks eine Liste von <b>Statements</b> bzw. <b>Blöcken</b> ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die <b>Erstellung</b> eines neuen <b>Stackframes</b> und Speicherung des Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und der <b>Rücksprungsadresse</b> nacheinander an den <b>Anfang</b> des neuen <b>Stackframes</b> . Das Attribut fun_name steht dabei für den Bezeichner der Funktion, für die ein neuer <b>Stackframe</b> erstellt werden soll. Das Attribut fun_name dient später dazu den <b>Block</b> dieser Funktion zu finden, weil dieser für den weiteren Kompilierungsvorgang wichtige Information in seinen <b>versteckten Attributen</b> gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die <b>Adresse</b> des Befehls, der direkt auf die <b>Jump Instruction</b> folgt, ersetzt wird.
RemoveStackframe()	Container für das <b>Entfernen</b> des aktuellen <b>Stackframes</b> durch das <b>Wiederherstellen</b> des im noch <b>aktuellen Stackframe</b> gespeicherten Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und das Setzen des SP-Registers auf den Wert des BAF-Registers <b>vor</b> der Wiederherstellung.
File(name, decls_defs_blocks)	Container für alle <b>Funktionen</b> oder <b>Blöcke</b> , welche eine Datei als Ursprung haben, wobei name der <b>Dateiname</b> der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von <b>Funktionen</b> bzw. <b>Blöcken</b> ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für <b>Statements</b> , der auch als <b>Block</b> bezeichnet wird, wobei das Attribut name der Bezeichner des <b>Labels</b> (Definition 2.1) des Blocks ist und stmts_instrs eine <b>Liste von Statements</b> oder <b>Instructions</b> . Zudem besitzt er noch 3 <b>versteckte Attribute</b> , wobei instrs_before die Zahl der <b>Instructions</b> vor diesem <b>Block</b> zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>Parameter</b> der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>lokalen Variablen</b> der Funktion belegt werden müssen.
GoTo(name)	Container für ein <b>Goto</b> zu einem anderen <b>Block</b> , wobei das Attribut name der Bezeichner des <b>Labels</b> des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen <b>Kommentar</b> , den der Compiler selber während des <b>Kompilierungsvorgangs</b> erstellt, der im <b>RETI-Interpreter</b> selbst später <b>nicht</b> sichtbar sein wird, aber in den <b>Immediate-Dateien</b> , welche die <b>Abstract Syntax Trees</b> nach den verschiedenen <b>Passes</b> enthalten.
RETIComment(value)	Container für einen <b>Kommentar</b> im Code der Form: // # comment, der im <b>RETI-Interpreter</b> später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer <b>RETI-CPU nicht umsetzbar</b> ist und auch nicht sinnvoll wäre umzusetzen. Der <b>Kommentar</b> ist im Attribut <b>value</b> , welches jeder Knoten besitzt gespeichert.

**Definition 2.1: Label**

Durch einen *Bezeichner* eindeutig zuordenbares *Sprungziel* im Programmcode.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

**Definition 2.2: 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.<sup>a</sup>

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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 2.3) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 2.4) unbesetzt ist. Das **Attribut** **position** wird später für Fehlermeldungen gebraucht.

**Definition 2.3: 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*.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

**Definition 2.4: 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.<sup>a</sup>

*Container-Knoten* entsprechen im *Abstract Syntax Tree* *Inneren Knoten*.<sup>b</sup>

<sup>a</sup>Wie 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.

<sup>b</sup>Thiemann, „Compilerbau“.



## 2.2.5.2 RETI-Knoten

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

<sup>a</sup> C. Scholl, „Betriebssysteme“

Tabelle 2.6: RETI-Knoten

**2.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung**

Hier sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax 2.2.1** enthalten sind.

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

Tabelle 2.7: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 2.7 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch  $\text{Stack}(\text{Num}('x')), x \in \mathbb{N}$  ersetzt wurde.

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

#### 2.2.5.4 Abstrakte Syntax

<i>stmt</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</i> ( <i>&lt;exp&gt;</i> , <i>&lt;bin_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>UnOp</i> ( <i>&lt;un_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Call</i> ( <i>Name('input')</i> , <i>None</i> )	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>Call</i> ( <i>Name('print')</i> , <i>&lt;exp&gt;</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</i> ( <i>&lt;exp&gt;</i> , <i>&lt;rel&gt;</i> , <i>&lt;exp&gt;</i> )   <i>ToBool</i> ( <i>&lt;exp&gt;</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>lhs</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>&lt;rel_loc&gt;</i>	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )	
<i>stmt</i>	::=	<i>Assign</i> ( <i>&lt;lhs&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Exp</i> ( <i>&lt;exp_stmts&gt;</i> )	
<i>datatype</i>	::=	<i>PntrDecl</i> ( <i>Num(str)</i> , <i>&lt;datatype&gt;</i> )	<i>L_Pntr</i>
<i>deref_loc</i>	::=	<i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )   <i>&lt;ref_loc&gt;</i>	
<i>ref_loc</i>	::=	<i>Name(str)</i>   <i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )	
<i>exp</i>	::=	<i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )	
<i>datatype</i>	::=	<i>ArrayDecl</i> ( <i>Num(str)</i> +, <i>&lt;datatype&gt;</i> )	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Array</i> ( <i>&lt;exp&gt;</i> +)	
<i>datatype</i>	::=	<i>StructSpec</i> ( <i>Name(str)</i> )	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )   <i>Struct</i> ( <i>Assign</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> ) +)	
<i>decl_def</i>	::=	<i>StructDecl</i> ( <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> ) +)	
<i>stmt</i>	::=	<i>If</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>IfElse</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *, <i>&lt;stmt&gt;</i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>DoWhile</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	<i>L_Fun</i>
<i>exp_stmts</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	
<i>stmt</i>	::=	<i>Return</i> ( <i>&lt;exp&gt;</i> )	
<i>decl_def</i>	::=	<i>FunDecl</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*)   <i>FunDef</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*, <i>&lt;stmt&gt;</i> *)	
<i>file</i>	::=	<i>File</i> ( <i>Name(str)</i> , <i>&lt;decl_def&gt;</i> *)	<i>L_File</i>

Grammar 2.2.3: Abstrakte Syntax der Sprache  $L_{Pioc}$

Man spricht hier von der ”**Abstrakten Syntax der Sprache  $L_{PicoC}$** ”, aber man hat bei der **Konkreten Syntax 2.1.1** auch schon von der ”**Konkreten Syntax der Sprache  $L_{PicoC}$** ” gesprochen. Hierbei handelt es sich um eine im Compilerbau geläufige **Redeart**, die vielleicht verwirrend sein kann, da die **Konkrete Syntax** und die **Abstrakte Syntax** zwei **verschiedene Grammatiken** sind, die für sich genommen zwei **unterschiedliche Sprachen** beschreiben. Die Sprache  $L_{PicoC}$  wird von ihrer **Konkreten Syntax** komplett beschrieben. Allerdings will man mit der **ersten Redeart** ausdrücken, dass die **Abstrakte Syntax** die zur Sprache  $L_{PicoC}$  passend definierte Grammatik ist, die beschreibt, was für **Kompositionen** mit den **PicoC-Knoten** möglich sind, die für die Kompilierung dieser Sprache  $L_{PicoC}$  definiert wurden. Für die tatsächliche Sprache, die durch die **Abstrakten Syntax** beschrieben wird, interessiert man sich nicht explizit.

Das Ausgeben eines **Abstract Syntax Trees** wird in Python über die **Magische Methode `__repr__()`**<sup>2</sup> umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine Textrepräsentation seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **runden öffnenden** ( und **schließenden** ) **Klammern**, sowie **Kommas** , und **Semikolons** ; zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach **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**.

#### 2.2.5.5 Transformer

#### 2.2.5.6 Codebeispiel

Beispiel welches in Subkapitel 2.2.3.2 angefangen wurde, wird hier fortgeführt.

```

1 File
2   Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')]),
8           ↪ PtrDecl(Num('1'), IntType('int')))), Name('attr'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')]),
16          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var'))
17      ]
18  ]

```

**Code 2.4:** *Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert*

<sup>2</sup>Spezielle Methode, die immer aufgerufen wird, wenn das **Object**, dass in Besitz dieser Methode ist als **String** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.

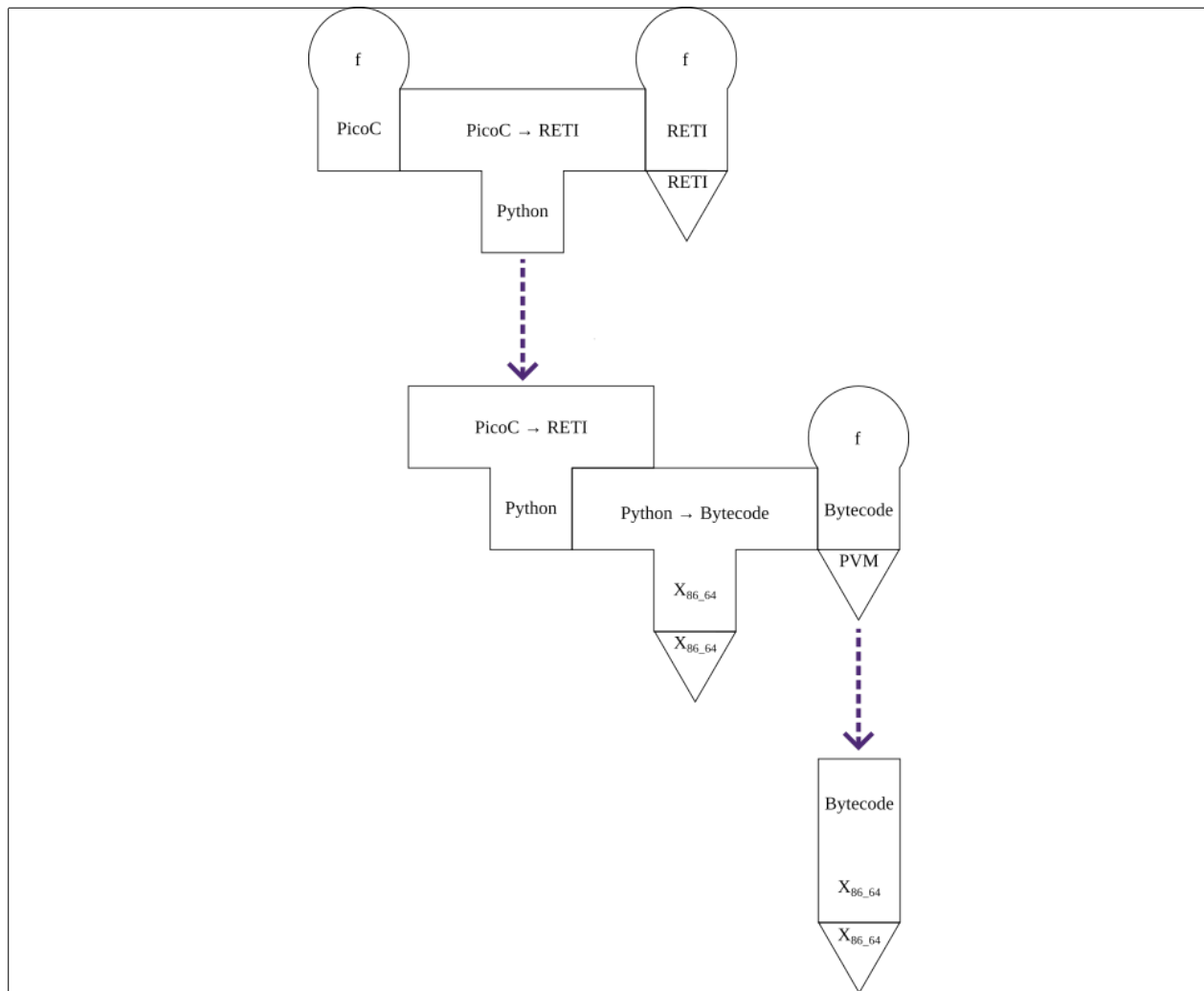
## 2.3 Code Generierung

### 2.3.1 Übersicht

Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 2.2, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **Container-Knoten** und **Token-Knoten** im **Abstract Syntax Tree** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

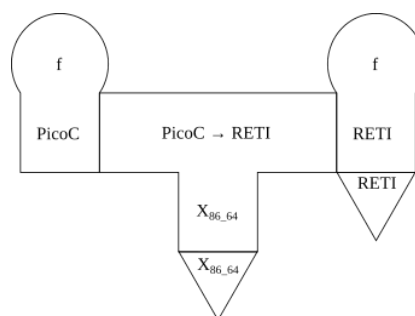
Man steht nun dem Problem gegenüber einen **Abstract Syntax Tree** der Sprache  $L_{PicoC}$ , der durch die **Abstrakte Syntax** in Grammatik 2.2.3 spezifiziert ist in einen entsprechenden **Abstract Syntax Tree** der Sprache  $L_{RETI}$  umzuformen. Das ganze lässt sich, wie in Unterkapitel 1.5 bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition 1.43) herunterbricht.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton 1.10). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel 1.1.1) in Abbildung 2.1 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.



**Abbildung 2.1:** *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagramm** in Abbildung 2.1 lässt sich zusammenfassen, sodass man das **T-Diagramm** in Abbildung 2.2 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in X<sub>86\_64</sub>-Maschinensprache geschrieben ist.



**Abbildung 2.2:** *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 2.3 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition 1.29) 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.

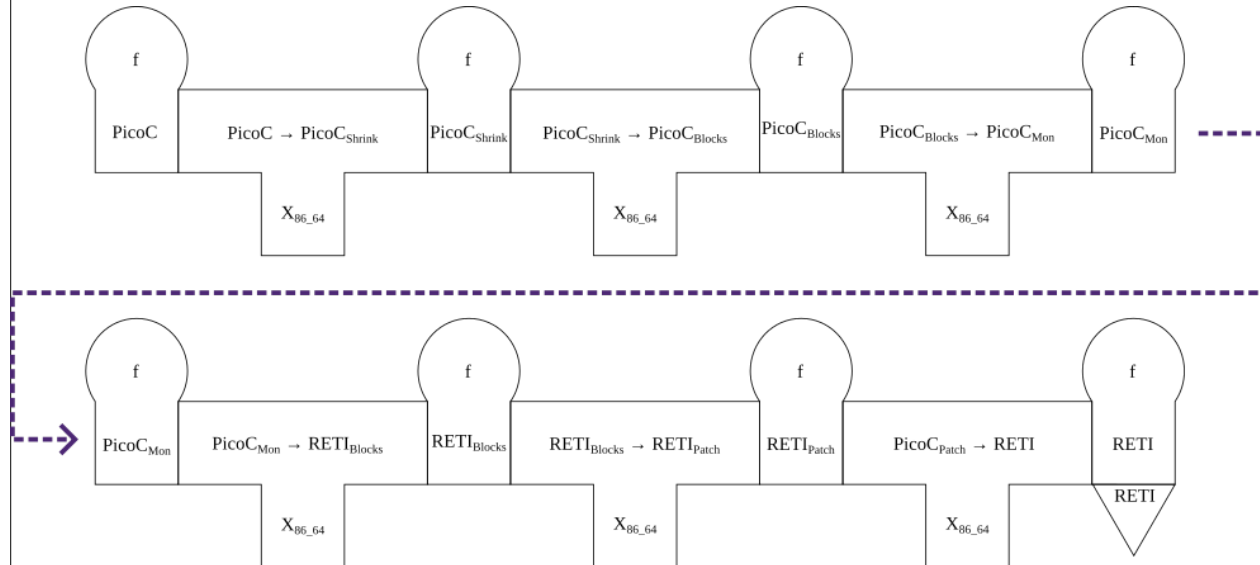


Abbildung 2.3: Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 2.3.2 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 2.3.2 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 wird<sup>3</sup>. 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**<sup>4</sup> finden und mithilfe der im Ordner `/code_examples` beiliegenden **Makefile** und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind<sup>5</sup>.

### 2.3.2 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 Aspekte<sup>6</sup> der

<sup>3</sup>Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

<sup>4</sup>[https://github.com/matthejue/Bachelorarbeit/tree/master/code\\_examples](https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples)

<sup>5</sup>Es 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.

<sup>6</sup>In 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.

### 2.3.2.1 PicoC-Shrink Pass

#### 2.3.2.1.1 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 sparen lassen, denn man kann die **Dereferenzierung** `*(var + i)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `var[i]` übernehmen lassen.

#### 2.3.2.1.2 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 2.5 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 2.5 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“, der in der Vorlesung allerdings **rekursiv** implementiert war.

Dieser **rekursive** Algorithmus ist allerdings **kein** gutes **Anschaungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von `if`-, `if-else`-, `while`- und `do-while`-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 2.5 umgeschrieben, um `if`- und `while`-Statemtnens zu enthalten.

Beide Varianten des Algorithmus wurden zum Testen des **PicoC-Compilers** verwendet und sind als Tests im Ordner `/tests` unter [Link<sup>7</sup>](#) unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln ??, ??, ?? und ?? zu **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) {
```

<sup>7</sup>[https://github.com/matthejue/PicoC-Compiler/tree/new\\_architecture/tests](https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests)

```

7     return res;
8 }
9     res = n * res;
10    n = n - 1;
11 }
12 }
13
14 void main() {
15     print(faculty(4));
16 }

```

Code 2.5: PicoC Code für Codebeispiel

In Code 2.6 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       ],
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 2.6: Abstract Syntax Tree für Codebeispiel

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstract Syntax Tree** in Code 2.6, da das Codebeispiel keine **Dereferenzierung** enthält.

### 2.3.2.2 PicoC-Blocks Pass

#### 2.3.2.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** ist 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.

#### 2.3.2.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax 2.2.3** um die Knoten zu erweitern, die im Unterkapitel 2.3.2.2.1 erwähnt wurden. Des Weiteren wird für die **Kommentare**, die in vielen Codebeispielen zur leichteren Verständlichkeit eingefügt wurden ein `SingleLineComment(prefix, content)`-Knoten benötigt. Die **Funktionsdefinition** `FunDef(<datatype>, Name(str), L Fun Alloc(Writeable(), <datatype>, Name(str))* , <block>*)` ist nun ein Container für **Blöcke** `Block(Name(str), <stmt>*)` und keine Statements `stmt` mehr.

<i>decl_def</i>	$::=$	<i>FunDef</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))* , &lt;block&gt;*)</i>	<i>L_Fun</i>
<i>block</i>	$::=$	<i>Block</i> ( <i>Name(str)</i> , <i>&lt;stmt&gt;*)</i>	<i>L_Blocks</i>
<i>stmt</i>	$::=$	<i>GoTo</i> ( <i>Name(str)</i> )   <i>SingleLineComment</i> ( <i>str, str</i> )	

**Grammar 2.3.1:** Abstrakte Syntax für  $L_{PicoC\_Blocks}$

#### 2.3.2.2.3 Codebeispiel

In Code 2.7 sieht man den **Abstract-Syntax-Tree** des **Piocc-Blocks Passes** für das aus Unterkapitel 2.5 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 zu **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(Writeable(), IntType('int'), Name('n'))
9       ],
10      [
11        Block
12          Name 'faculty.6',
13          [
14            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
15            // While(Num('1'), [])
16            GoTo(Name('condition_check.5'))

```

```

17     ],
18     Block
19     Name 'condition_check.5',
20     [
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 2.7: PicoC-Blocks Pass für Codebeispiel

### 2.3.2.3 PicoC-Mon Pass

#### 2.3.2.3.1 Aufgabe

Die Aufgabe des **PicoC-Mon Pass** ist es den **Abstract Syntax Tree** "in eine **eingeschränkte Form** zu bringen, in der die **Argumente** von **Operationen** **Atomare Ausdrücke** sind"<sup>8</sup>

#### 2.3.2.3.2 Abstrakte Syntax

<i>ref_loc</i>	::=	<i>Stack</i> ( <i>Num</i> ( <i>str</i> ))   <i>Global</i> ( <i>Num</i> ( <i>str</i> ))	<i>L_Assign_Alloc</i>
		<i>Stackframe</i> ( <i>Num</i> ( <i>str</i> ))	
<i>error_data</i>	::=	<i>&lt;exp&gt;</i>   <i>Pos</i> ( <i>Num</i> ( <i>str</i> ), <i>Num</i> ( <i>str</i> ))	
<i>exp</i>	::=	<i>Stack</i> ( <i>Num</i> ( <i>str</i> ))   <i>Ref</i> ( <i>&lt;ref_loc&gt;</i> , <i>&lt;datatype&gt;</i> , <i>&lt;error_data&gt;</i> )	
<i>stmt</i>	::=	<i>Exp</i> ( <i>&lt;exp&gt;</i> )	
		<i>Assign</i> ( <i>Alloc</i> ( <i>Writable</i> ()), <i>StructSpec</i> ( <i>Name</i> ( <i>str</i> )), <i>Name</i> ( <i>str</i> )),	
		<i>Struct</i> ( <i>Assign</i> ( <i>Name</i> ( <i>str</i> ), <i>&lt;exp&gt;</i> )+, <i>&lt;datatype&gt;</i> ) )	
		<i>Assign</i> ( <i>Alloc</i> ( <i>Writable</i> ()), <i>ArrayDecl</i> ( <i>Num</i> ( <i>str</i> )+, <i>&lt;datatype&gt;</i> ),	
		<i>Name</i> ( <i>str</i> )), <i>Array</i> ( <i>&lt;exp&gt;</i> +, <i>&lt;datatype&gt;</i> ))	
		<i>NewStackframe</i> ( <i>Name</i> ()), <i>GoTo</i> ( <i>str</i> ))	
		<i>RemoveStackframe</i> ()	
<i>symbol_table</i>	::=	<i>SymbolTable</i> ( <i>&lt;symbol&gt;</i> )	<i>L_Symbol_Table</i>
<i>symbol</i>	::=	<i>Symbol</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>&lt;name&gt;</i> , <i>&lt;val&gt;</i> , <i>&lt;pos&gt;</i> , <i>&lt;size&gt;</i> )	
<i>type_qual</i>	::=	<i>Empty</i> ()	
<i>datatype</i>	::=	<i>BuiltIn</i> ()   <i>SelfDefined</i> ()	
<i>name</i>	::=	<i>Name</i> ( <i>str</i> )	
<i>val</i>	::=	<i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	
<i>pos</i>	::=	<i>Pos</i> ( <i>Num</i> ( <i>str</i> ), <i>Num</i> ( <i>str</i> ))   <i>Empty</i> ()	
<i>size</i>	::=	<i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	

**Grammar 2.3.2:** Abstrakte Syntax für *L<sub>PicoC\_Mon</sub>*

#### Definition 2.5: Symboltabelle

#### 2.3.2.3.3 Codebeispiel

```

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',

```

<sup>8</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

```

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 2.8: PicoC-Mon Pass für Codebeispiel

### 2.3.2.4 RETI-Blocks Pass

#### 2.3.2.4.1 Aufgaben

#### 2.3.2.4.2 Abstrakte Syntax

<i>program</i>	$::=$	<i>Program</i> ( <i>Name</i> ( <i>str</i> ), $\langle$ <i>block</i> $\rangle^*$ )	<i>L_Program</i>
<i>exp_stmts</i>	$::=$	<i>GoTo</i> ( <i>str</i> )	<i>L_Blocks</i>
<i>instrs_before</i>	$::=$	<i>Num</i> ( <i>str</i> )	
<i>num_instrs</i>	$::=$	<i>Num</i> ( <i>str</i> )	
<i>block</i>	$::=$	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), $\langle$ <i>instr</i> $\rangle^*$ , $\langle$ <i>instrs_before</i> $\rangle$ , $\langle$ <i>num_instrs</i> $\rangle$ )	
<i>instr</i>	$::=$	<i>GoTo</i> ( <i>Name</i> ( <i>str</i> ))	

Grammar 2.3.3: Abstrakte Syntax für  $L_{RETI\_Blocks}$ 

#### 2.3.2.4.3 Codebeispiel

```

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     LOADIN 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 2.9: RETI-Blocks Pass für Codebeispiel

### 2.3.2.5 RETI-Patch Pass

#### 2.3.2.5.1 Aufgaben

#### 2.3.2.5.2 Abstrakte Syntax

$$\text{stmt} ::= \text{Exit}(\text{Num}(\text{str}))$$
Grammar 2.3.4: Abstrakte Syntax für  $L_{\text{RETI\_Patch}}$ 

#### 2.3.2.5.3 Codebeispiel

```

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;
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 2.10: RETI-Patch Pass für Codebespiel

### 2.3.2.6 RETI Pass

#### 2.3.2.6.1 Aufgaben

#### 2.3.2.6.2 Konkrete und Abstrakte Syntax

<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_with_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 2.3.5:** Konkrete Syntax für *L<sub>RETI\_Lex</sub>*

<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>arg</i> <i>num</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 2.3.6:** Konkrete Syntax für *L<sub>RETI\_Parse</sub>*

<i>reg</i>	::=	<i>ACC</i> ()   <i>IN1</i> ()   <i>IN2</i> ()   <i>PC</i> ()   <i>SP</i> ()   <i>BAF</i> ()	<i>L_Program</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>program</i>	::=	<i>Program</i> ( <i>Name</i> ( <i>str</i> ), ⟨ <i>instr</i> ⟩*)	

**Grammar 2.3.7:** Abstrakte Syntax für *L<sub>RETI</sub>***2.3.2.6.3 Codebeispiel**

```

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;
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 2.11:** *RETI Pass für Codebespiel*

---

---

# Literatur

## Online

- *C Operator Precedence* - *cppreference.com*. URL: [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence) (besucht am 27.04.2022).
- *Errors in C/C++* - *GeeksforGeeks*. URL: <https://www.geeksforgeeks.org/errors-in-cc/> (besucht am 10.05.2022).
- *JSON parser - Tutorial* — *Lark documentation*. URL: [https://lark-parser.readthedocs.io/en/latest/json\\_tutorial.html](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).
- *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

- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index\\_de.html](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](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/swtvl> (besucht am 19.07.2022).

## Sonstige Quellen

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).