Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

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**[1] und von **Tests** die **Beziehungen** in 1.2.1 zu belegen (siehe Subkapitel **??**).

Definition 1.1: Interpreter

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

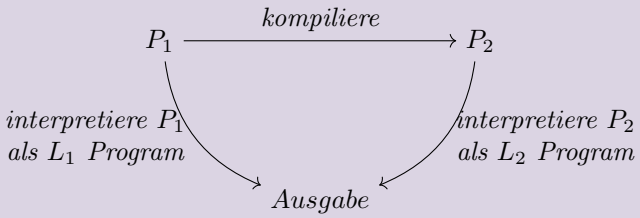
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstrakter Syntaxbaum*** *(Definition 1.42) und führt je nach Komposition der* ***Nodes*** *des Abstrakter Syntaxbaum, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.2: Compiler

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

Wobei ***Kompilieren*** *meint, dass ein beliebiges Program P_1 in der Sprache L_1 so in die Sprache L_2 zu einem Programm P_2 übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen L_1 und L_2* ***interpretiert*** *werden, die gleiche* ***Ausgabe*** *rauskommt, wie es in Diagramm 1.2.1 dargestellt ist. Also beide Programme P_1 und P_2 die gleiche* ***Semantik*** *(Definition 1.15) haben und sich nur* ***syntaktisch*** *(Definition 1.14) durch die Sprachen L_1 und L_2, in denen sie geschrieben stehen unterscheiden.*[a]



(1.2.1)

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

[1] Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **G**NU **C**ompiler **C**ollection

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C^{o\_j}_{i\_w\_k\_min}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschienensprache L_{B_i} einer Maschiene M_i kompiliert. Fall die Notwendigkeit besteht die **Maschiene** M_i anzugeben, zu dessen **Maschienensprache** 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 $C^{o\_j}_{w\_k}$ 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 **Maschienenncode**, der in **Maschienensprache** (Definition 1.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition **??**) oder **Cross-Compiler** (Definition 1.5). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition **??**) voneinander zu unterscheiden.

Definition 1.3: Maschienensprache

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

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschienenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschienenbefehle keine Operanden mit zu großen **Immediates** (Definition 1.4) haben.

[b] Scholl, „Betriebssysteme“.

Der **Maschienencode**, denn ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschienenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschiene, 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 Maschienencode, der die Maschienenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 1.4) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, 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[2].

Definition 1.4: Immediate

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

[2] Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschienencode 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 Maschienenbefehle übergibt, anstatt z.B. eine unnötig **platzverbrauchenden** UTF-8 Codierung zu verwenden, die nur in sehr vielen Schritt einen Befehl verarbeiten kann, da die Register und Speicherzellen des Hauptspeichers üblicherweise nur **32- bzw. 64-Bit Breite** haben.

[a] Ljohhuh, *What is an immediate value?*

Definition 1.5: 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 **Maschienensprachen** B_1 und B_2 haben.*[a][b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C^{Python}_{PicoC}.
[b] 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 **Maschienensprache** B_2 einer Zielmaschine M_2 läuft.[3]

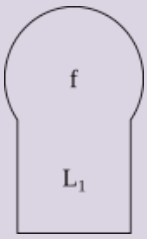
1.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus 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 Program (Definition 1.6), einen Übersetzer (Definition 1.7), einen Interpreter (Definition 1.8) und eine Maschiene (Definition 1.9) zusammen.

Definition 1.6: T-Diagram Programm

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



[a] Earley und Sturgis, „A formalism for translator interactions“.

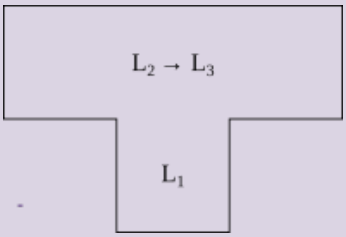
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 1.6 also reichen einfach eine 1 hinzuschreiben.

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

Definition 1.7: T-Diagram Übersetzer (bzw. eng. Translator)

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

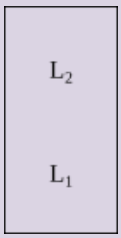
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 1.2.1.*[b]



[a] 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**.
[b] Earley und Sturgis, „A formalism for translator interactions“.

Definition 1.8: T-Diagram Interpreter

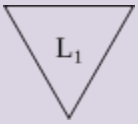
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



[a] Earley und Sturgis, „A formalism for translator interactions“.

Definition 1.9: T-Diagram Maschiene

*Repräsentiert eine **Maschiene**, welche ein **Programm** in **Maschienensprache** L_1 ausführt.*[a b]



[a] Wenn die Maschiene **Programme** in einer höheren Sprache als **Maschienensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschiene**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).
[b] Earley und Sturgis, „A formalism for translator interactions“.

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

Sowohl **horinzontale** als auch **vertikale Adjazents** lassen sich, wie man in den Abbildungen 1.1 und 1.2 erkennen kann zusammenfassen.

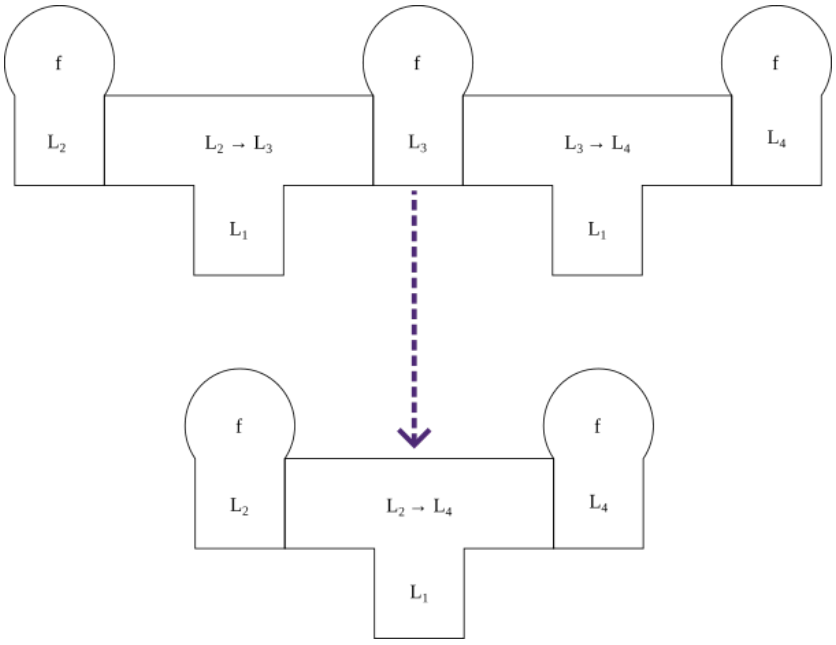


Abbildung 1.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen*

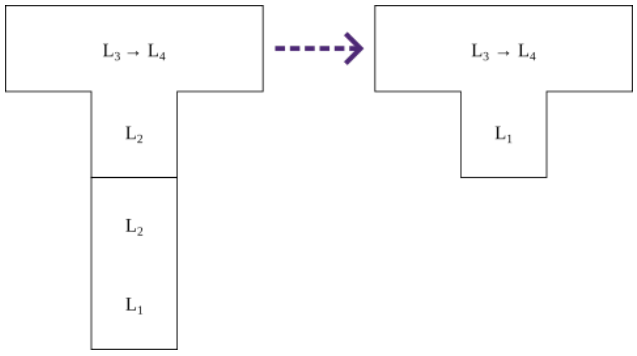


Abbildung 1.2: *Vertikale Interpretierungszwischenschritte zusammenfassen*

1.2 Formale Sprachen

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

Definition 1.10: Symbol

*„Ein Symbol ist ein **Element** eines **Alphabets** Σ.“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 1.11: Alphabet

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 1.10).“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 1.12: Wort

*„Ein Wort $w = a_1...a_n \in \Sigma^*$ ist eine **endliche Folge** von **Symbolen** aus einem **Alphabet** Σ.“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 1.13: Formale Sprache

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 1.12) über dem **Alphabet** Σ (Definition 1.11).“*[a]

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

[a] Nebel, „Theoretische Informatik“.

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

Definition 1.14: Syntax

*Die Syntax bezeichnet alles was mit dem **Aufbau** von **Formalen Sprachen** zu tuen hat. Die **Grammatik** einer Sprache, aber auch die in **Natürlicher Sprache** ausgedrückten Regeln, welche den Aufbau von Wörtern einer Formalen Sprache beschreiben werden als Syntax bezeichnet. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben*[a]*.*[b]

[a] Z.B. die **Konkrette** und **Abstrakte Syntax**, die später eingeführt werden.
[b] Thiemann, „Einführung in die Programmierung“.

Definition 1.15: Semantik

*Die Semantik bezeichnet alles was mit der **Bedeutung** von **Formalen Sprachen** zu tuen hat.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

Definition 1.16: Formale Grammatik

*„Eine Formale Grammatik beschriebt wie **Wörter** einer **Sprache** abgeleitet werden können.“*[a]

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

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

- $N \mathrel{\hat{=}}$ ***Nicht-Terminalsymbole**.*

- $\Sigma \mathrel{\hat{=}}$ ***Terminalsymbole***, *wobei* $N \cap \Sigma = \emptyset$[b][c].
- $P \mathrel{\hat{=}}$ *Menge von* ***Produktionsregeln*** $w \to v$, *wobei* $w, v \in (N \cup \Sigma)^* \wedge w \notin \Sigma^*$.[d][e]
- $S \mathrel{\hat{=}}$ ***Startsymbol***, *wobei* $S \in N$.

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

[a] Nebel, „Theoretische Informatik“.
[b] Weil mit ihnen **terminiert** wird.
[c] Kann auch als **Alphabet** bezeichnet werden.
[d] w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.
[e] Bzw. $w, v \in V^* \wedge w \notin \Sigma^*$.

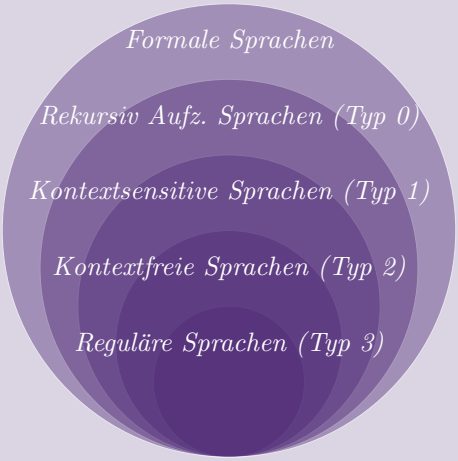
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 1.17) einteilen.

Definition 1.17: Chromsky Hierarchie

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

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

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



(1.17.1)

[a] Nebel, „Theoretische Informatik“.

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

Definition 1.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{1.18.1}$$

*haben, wobei A, B **Nicht-Terminalsymbole** sind und c ein **Terminalsymbol** ist[a b]."[c]*

[a] Diese Definition einer **Regulären Grammatik** ist **rechtsregulär**, es ist auch möglich diese Definition **linksregulär** zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.
[b] Dadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.
[c] Nebel, „Theoretische Informatik".

Definition 1.19: Kontextfreie Grammatik

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

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

*haben, wobei A ein **Nicht-Terminalsymbol** ist und v ein beliebige Folge von **Grammatiksymbolen**[a] ist."*[b]

[a] Also eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.
[b] Nebel, „Theoretische Informatik".

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

Definition 1.20: Wortproblem

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

*Das Wortproblem kann durch die folgende **Indikatorfunktion**[c] zusammengefasst werden:*

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

[a] Bzw. **„ja"** oder **„nein"** usw., es muss nicht umgedingt 1 oder 0 sein.
[b] Nebel, „Theoretische Informatik".
[c] Auch **Charakteristische Funktion** genannt.

1.2.1 Ableitungen

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

[4] Bzw. **Wort**.

Definition 1.21: 1-Schritt-Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

Definition 1.22: Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

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

Definition 1.23: Links- und Rechtsableitungableitung

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

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

[a] Nebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 1.24) ist[6].

Definition 1.24: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

[a] *Parsing Expressions · Crafting Interpreters*.

[5] Bzw. **Wort**.
[6] Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (siehe Unterkapitel **??**) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 1.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 1.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 1.36) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 1.25: Formaler Ableitungsbaum

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

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

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

[a] Z.B. **Programmcode**.
[b] Nebel, „Theoretische Informatik“.

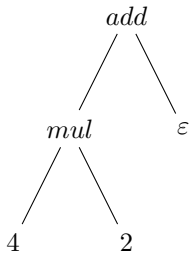
In Abbildung 1.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 1.25.1 nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition 2.3) angegebenen **Produktionen** 1.1 einer ansonsten nicht näher spezifizierten **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ ergibt.

DIG_NO_0	::=	”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”	*L_Lex*
DIG_WITH_0	::=	”0” | DIG_NO_0	
NUM	::=	”0” | DIG_NO_0 DIG_WITH_0*	
add	::=	add ”+” mul | mul	*L_Parse*
mul	::=	mul ”*” NUM | NUM	

Grammatik 1.1: *Produktionen des Ableitungsbaums*

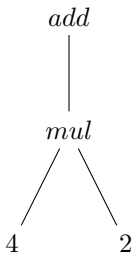
$$add \Rightarrow mul \Rightarrow mul \text{ ”*” } NUM \Rightarrow NUM \text{ ”*” } NUM \Rightarrow 4 \text{ ”*” } NUM \Rightarrow 4 \text{ ”*” } 2 \tag{1.25.1}$$

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



(1.25.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[7].



(1.25.3)

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

Definition 1.26: Mehrdeutige Grammatik

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

[a] Alternativ, wenn es für w **mehrere** unterschiedliche **Linksableitungen** gibt.
[b] Nebel, „Theoretische Informatik“.

1.2.2 Präzidenz und Assoziativität

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

Definition 1.27: Assoziativität

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

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

[a] *Parsing Expressions · Crafting Interpreters*.

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

Bei **Assoziativität** ist z.B. der **Multitplikationsoperator** `*` ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator** `=` ein Beispiel für einen **rechtsassoziativen** Operator. Dies ist in Abbildung 1.3 mithilfe von Klammern `()` veranschaulicht.

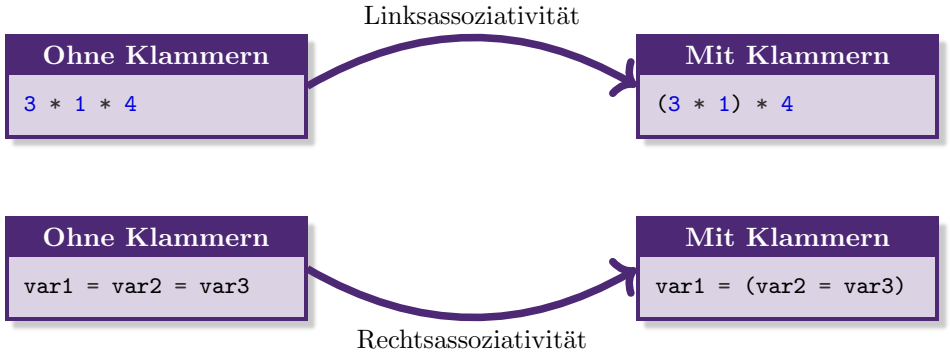


Abbildung 1.3: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität*

Definition 1.28: Präzidenz

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

[a] *Parsing Expressions · Crafting Interpreters.*

Bei **Präzidenz** ist die Mischung der Operatoren für **Subraktion** `'-'` und für **Multiplikation** `*` ein Beispiel für den Einfluss von Präzidenz. Dies ist in Abbildung 1.4 mithilfe der Klammern `()` veranschaulicht. Im Beispiel in Abbildung 1.4 ist bei den beiden **Subtraktionsoperatoren** `'-'` nacheinander und dem darauffolgenden **Multitplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzidenz** im Spiel.

Abbildung 1.4: *Veranschaulichung von Präzidenz*

1.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter 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. Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 1.31) genannt.

Definition 1.29: Pipe-Filter Architekturpattern

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

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

*Eine **Pipe** stellt ein **Bindeglied** zwischen zwei **Filtern** dar.*[a][b]

[a] 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 Anlehung an z.B. die **Pipes aus Unix**, z.B. `cat /proc/bus/input/devices | less` zum Ausdruck gebracht werden. Und so banal es klingt, sollen manche Bezeichnungen von Pattern auch einfach nur gut klingen.
[b] Westphal, „Softwaretechnik“.

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* [a]*, die für die Beschreibung eines **Tokens** T zuständig sind.*[b]

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] 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.*[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 1.32) im **Inputstring** identifziert 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 $lex : \Sigma^* \rightharpoonup (N \times W)^*$, welche ein **Wort** bzw. **Lexeme** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls dieses **Wort** sich unter der **regulären Grammatik** G_{Lex}, der **regulären Sprache** L_{Lex} ableiten lässt bzw. einem der **Pattern** der Sprache L_{Lex} entspricht.*[a]

[a] 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 verzubäugen 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[a].

[a] 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.*[a][b]

[a] Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.
[b] Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[8] und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ 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`[9], 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`[10]. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

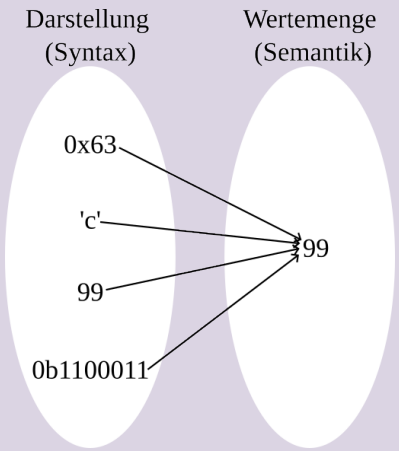
Die **Grammatik** G_{Lex}, die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[11], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 2.1.1 liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 1.18 erfüllt. Einzig die Produktion `CHAR ::= "'"ASCII_CHAR"'"` sieht problematisch aus, kann allerdings auch

[8] 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 Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.
[9] 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.
[10] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.
[11] Man nennt das auch einem **Lookahead** von 1

als `{CHAR ::= "'"CHAR2, CHAR2 ::= ASCII_CHAR"'"}` **regulär** ausgedrückt werden[12]. 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**.*[a]



[a] Thiemann, „Einführung in die Programmierung“.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 1.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

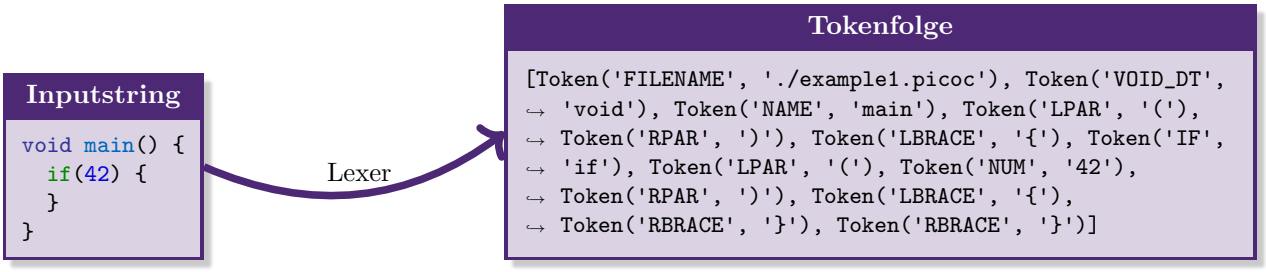


Abbildung 1.5: *Veranschaulichung der Lexikalischen Analyse*

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.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 1.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Defini-

[12] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

tion 1.37), ein **Ableitungsbaum** (Definition 1.36) generiert, der als Zwischenstufe hin zum einem **Abstrakter Syntaxbaum** (Definition 1.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaums** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 1.35: Konkrette Syntax

*Steht für alles, was mit dem **Aufbau** von **Ableitungsbäumen** zu tuen hat, also z.B. was für **Ableitungen** mit den **Grammatiken** G_{Lex} und G_{Parse} zusammengenommen möglich sind.*

*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 **Konkretter Syntax** aufgeschrieben.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.36: Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)

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

*Die **Konkrette Syntax** nach der sich der **Ableitungsbaum** richtet wird optimalerweise immer so definiert, dass sich möglichst einfach ein **Abstrakter Syntaxbaum** daraus konstruieren lässt.*[a]

[a] *JSON parser - Tutorial — Lark documentation*.

Definition 1.37: Parser

*Ein **Parser** ist ein Programm, dass aus einem Inputstring, der in **Konkretter Syntax** geschrieben ist, eine compilerinterne Darstellung, den **Ableitungsbaum** generiert, was auch als **Parsen** bezeichnet wird*[a] [b].

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 1.37 verwendet.
[b] *JSON parser - Tutorial — Lark documentation*.

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

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

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

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

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

[b] Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beinflussen 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 konkretten 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 1.20) für die **Sprache**, die durch die **Konkrette Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Ableitungsbaum**.

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** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit den Produktionen der **Konkrette Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkretten Syntax** beschrieben wird oder nicht. Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** (Definition 1.20) bekannt.*[a]

[a] Thiemann, „Compilerbau“.

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

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

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

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

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

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 1.24) 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 **??**) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass der **Inputstring** sich **nicht** mit der verwendeten Grammatik

ableiten lässt.[b]

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

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

[a] *What is Top-Down Parsing?*
[b] 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.
[c] 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.
[d] *What is Bottom-up Parsing?*
[e] Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstrakter Syntaxbaum** 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 ein Programm von der **Konkretten 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 **Ableitungsbaum** besucht und beim Antreffen eines bestimmten Knoten des **Derivation Tree** einen entsprechenden Knoten des **Abstrakter Syntaxbaum** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstrakter Syntaxbaum** konstruiert.*[a]

[a] *Transformers & Visitors — Lark documentation.*

Definition 1.40: Visitor

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

[a] Kann theoretisch auch zur Konstruktion eines **Abstrakter Syntaxbaum** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakter Syntaxbaum** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.
[b] *Transformers & Visitors — Lark documentation.*

Definition 1.41: Abstrakte Syntax

*Steht für alles, was mit dem **Aufbau** von **Abstrakten Syntaxbäumen** zu tuen hat, also z.B. was für Arten von **Kompositionen** mit den **Knoten** eines **Abstrakten Syntaxbaums** möglich sind.*

*Ein **Abstract Syntax Tree**, der zur Kompilierung eines Wortes[a] generiert wurde, ist nach einer **Abstrakter Syntax** konstruiert.*

*Jene Produktionen, die in der **Konkretten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht. Dadurch sind die **Kompositionen**, welche die Knoten im **Abstract Snytax Tree** bilden können **syntaktisch** meist näher zur Syntax von **Maschinenbefehlen**.[b]*

[a] Z.B. **Programmcode**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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

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

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

*Die **Abstrakte Syntax** nach der sich der **Abstrakte Syntaxbaum** richtet wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten[a] möglichst **einfach weiterverarbeitet** werden kann.*

[a] Die verschiedenen **Passes**.

In Abbildung 1.6 wird das Beispiel aus Unterkapitel 1.2.1 fortgeführt, welches den **Arithmetischen Ausdruck** `4 * 2` in Bezug auf die Grammatik 1.1, welche die **höhere Präzidenz** der **Multipikation** $*$ berücksichtigt in einem **Ableitungsbaum** darstellt. In Abbildung 1.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**. Das geschieht bezogen auf das Beispiel aus Unterkapitel 1.2.1, indem jegliche Knoten, die im **Ableitungsbaum** nur existieren, weil die Grammatik so umgesetzt ist, dass es nur **einen** einzigen möglichen **Ableitungsbaum** geben kann **wewgabstrahiert** werden.

Abbildung 1.6: *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

Die **Baumdatenstruktur** des **Ableitungsbaumes** und **Abstrakten Syntaxbaumes** ermöglicht es die

Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Programmes ausführen muss möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 1.7 die Syntaktische mit dem Beispiel aus Subkapitel 1.3 fortgeführt.

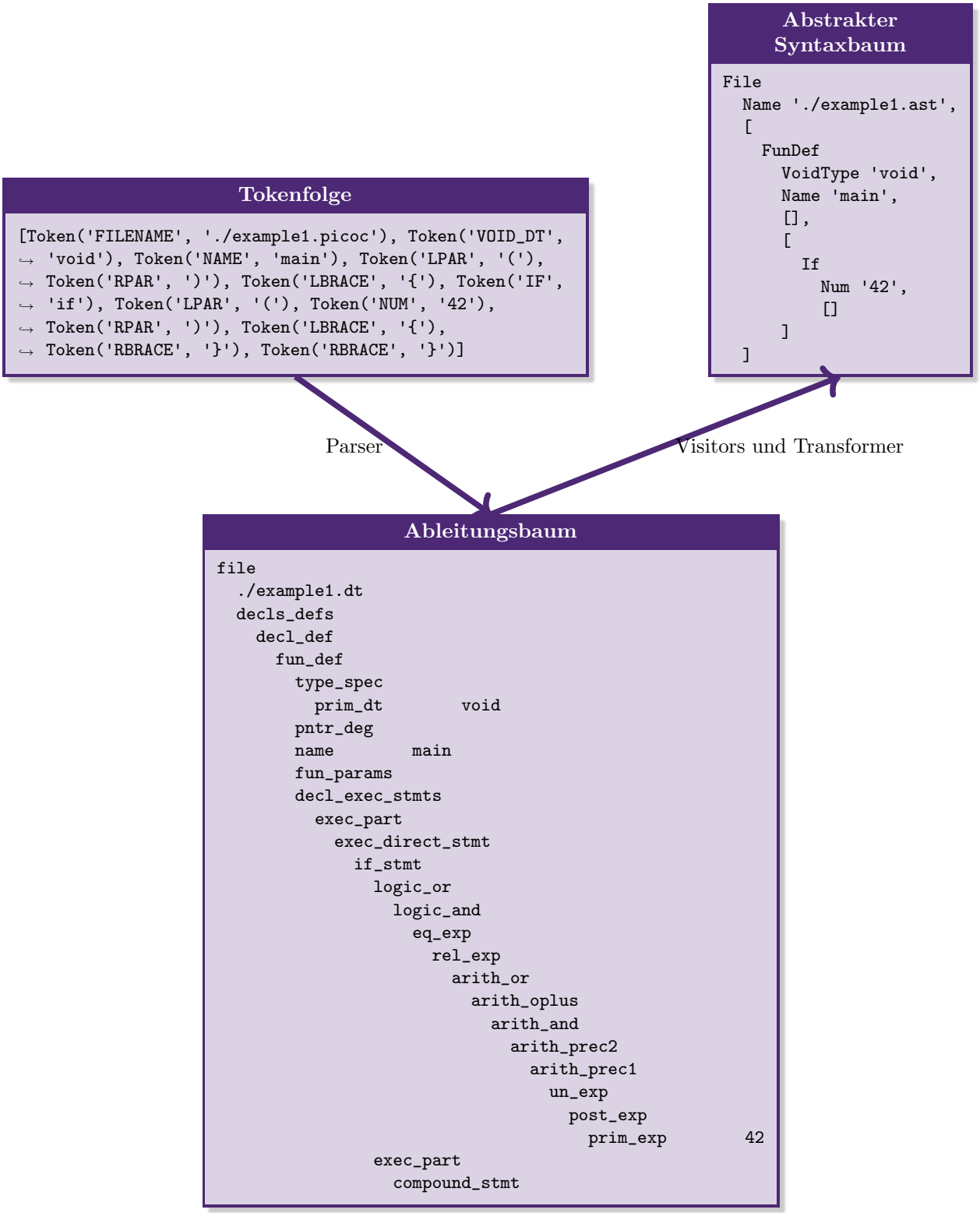


Abbildung 1.7: *Veranschaulichung der Syntaktischen Analyse*

1.5 Code Generierung

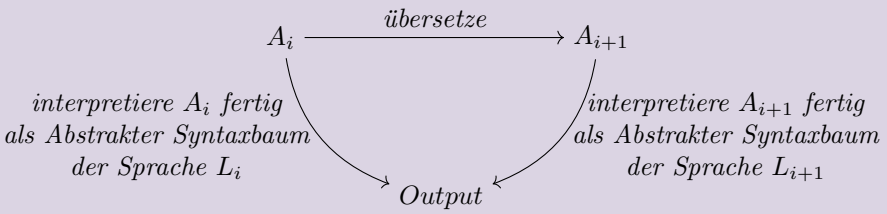
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** einer Sprache L_1 in den **Abstrakter Syntaxbaum** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition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

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

Definition 1.43: Pass

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

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



(1.43.1)

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

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

[b] Der Begriff kommt aus dem **Englischen** von „passing over“, da der gesamte **Abstrakte Syntaxbaum** in einem **Pass** durchlaufen wird.

[c] **Interpretieren** geht immer von einem Programm in **Konkretter Syntax** aus, wobei der **Abstrakte Syntaxbaum** ein **Zwischenschritt** bei der **Interpretierung** ist.

[d] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Die von den **Passes** umgeformten **Abstrakter Syntaxbaums** sollten dabei mit jedem **Pass** der **Syntax** von **Maschienenbefehlen** immer ähnlicher werden, bis es schließlich nur noch Maschienenbefehle sind.

1.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche **Unreine Ausdrücke** (Definition 1.45) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 1.44) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 1.44: Reiner Ausdruck (bzw. engl. pure expression)

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

[a] Sondern z.B. **intern** etwas am **Kompilierprozess** ändert.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.45: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definiton 1.46).

Definition 1.46: Monadische Normalform (bzw. engl. monadic normal form)

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

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

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

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 1.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[13] aufgeschrieben wurden.

In der Abbildung 1.8 ist der Ausdruck mit dem **Nebeneffekt** eine Variable zu **allokieren:** `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt, daher muss der **Unreine** Ausdruck als eigenständiges Statement **vorangestellt** werden.

[13] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

Abbildung 1.8: *Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten*

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

1.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um **Maschienensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 1.49) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Statements macht, in denen die **Komplexen Ausdrücke temporären Locations** zugewiesen werden (Definiton 1.47) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statemtent**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 1.48) enthalten.

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

Definition 1.47: 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 tuen** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 1.50). Wenn eine **Konkrette Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 1.50), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 1.48).

Definition 1.48: Atomarer Ausdruck

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

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

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.49: Komplexer Ausdruck

*Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist, wie z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`.*[a][b]

[a] `int var` ist eine **Allokation**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.50: A-Normalform (ANF)

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

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

*Eine **Abstrakte Syntax** ist in **A-Normalform**, wenn die **Konkrette Syntax** für welche sie definiert wurde in **A-Normalform** ist.*[a][b][c]

[a] *A-Normalization: Why and How (with code)*.
[b] Bolingbroke und Peyton Jones, „Types are calling conventions“.
[c] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 1.9 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[14] aufgeschrieben wurden.

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition **??**) inklusive **Liveness Analysis** (Definition **??**) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.[15]

Aus diesem Grund verwendet das Beispiel in Abbildung 1.9 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **Abstrakter Syntaxbaum** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdrück** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

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

[14] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[15] Die in diesem **Paragraph** erwähnten **Begriffe** werden nur grob erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

befehlssatz es erlaubt **zwei Register** miteinander zu verechnen[16], ist es möglich **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** *'c'* als **atomar** zu definieren, da sie mit einem **Maschinenbefehl** verarbeitet werden können[17]. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das `SP`-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** *'c'* als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verechnet werden können.

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

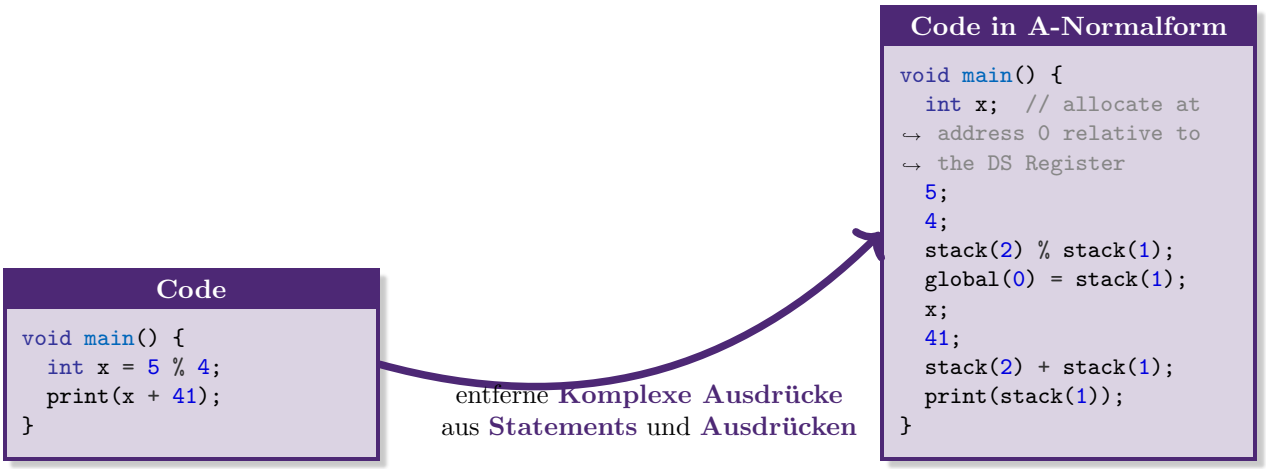


Abbildung 1.9: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen*

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

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

1.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakter Syntaxbaum** den eigentlichen **Maschinencode** in **Konkretter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt[18]. Der Weg von **Abstrakter Syntax** zu **Konkretter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkretten Syntax**

[16] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[17] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.
[18] Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschienencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben.

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

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

1.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**[19] eintritt:

1. der **Parser**[20] entscheidet das **Wortproblem** für ein **Eingabeprogramm**[21] mit `0`, also das **Eingabeprogramm** befolgt nicht die **Syntax** der **Sprache** des Compilers[22].

2. in den **Passes** tritt eine Fall ein, der **nicht** in der **Semantik** der Sprache des Compilers abgedeckt ist, z.B.:

   - eine **Variable** wird **verwendet**, obwohl sie noch **nicht deklariert** ist.

   - bei einem **Funktionsaufruf** werden **mehr** Argumente oder Argumente des **falschen Datentyps** übergeben, als in der **Funktionsdeklaration** oder **Funktionsdefinition** angegeben ist.

3. Während der **Laufzeit** des Compilers tritt ein Ereignis ein, das **nicht** durch die **Semantik** der Sprache des Compilers abgedeckt ist oder das **Betriebssystem** nicht erlaubt, z.B.:

   - eine **nicht erlaubte Operation**, wie **Division durch** 0 (z.B. `42 / 0`) soll ausgeführt werden.

   - **Segmentation Fault:** Wenn auf **Speicher zugegriffen** wird, der vom **Betriebssystem geschützt** ist.

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

- es gibt **keine** oder **mehr als eine** `main`-Funktion

- eine Funktion, die in einer **Objektdatei** (Definition **??**) benötigt wird, wird von **keiner** anderen oder **mehr als einer** Objektdatei bereitsgestellt

wird eine **Fehlermeldung** (Definition 1.51) ausgegeben.

Definition 1.51: Fehlermeldung

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

- *über* `stdout` *oder* `stderr` *im einem* ***Terminal Emulator*** *oder* ***richtigen Terminal***[b]*.*

[19] *Errors in C/C++ - GeeksforGeeks*.
[20] Bzw. der **Recognizer** im Parser.
[21] Bzw. **Wort**.
[22] Bzw. das **Eingabeprogramm** lässt sich **nicht** mit der Grammatik des Compilers **ableiten**.

- *über eine **Dialogbox** in einer **Graphischen Benutzerfläche**[c] oder **Zeichenorientierten Benutzerschnittstelle**[d].*
- *in ein **Register** oder an eine **spezielle Adresse** des **Hauptspeichers** wird ein **Wert geschrieben**.*
- ***Logdatei**[e] auf einem **Speichermedium**.*

[a] Dieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler selbst kompiliert** hat.
[b] Nur unter **Linux**, **Windows** hat sowas nicht.
[c] In engl. **G**raphical **U**ser **I**nterface, kurz **GUI**.
[d] In engl. **T**ext-based **U**ser **I**nterface, kurz **TUI**.
[e] In engl. **log file**.

2 Implementierung

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

Für das Parsen[1] des Programmes in der **Konkretten Syntax** der Sprache L_{PicoC} wird das **Lark Parsing Toolkit**[2] [3] verwendet. Das **Lark Parsing Toolkit** ist eine Bibliothek, die es ermöglicht mittels eines in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** (Definition 2.3 bzw. für den Dialekt von Lark Definition 2.4) spezifizierten Grammatik der **Konkretten Syntax** ein Programm in ebendieser **Konkretten Syntax** zu parsen und daraus einen **Ableitungsbaum** für die kompilerintere Weiterverarbeitung zu generieren.

Definition 2.1: Metasyntax

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

Definition 2.2: Metasprache

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

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

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

*Die Erweiterte Backus-Naur-Form[a] ist eine **Metasyntax** (Definition 2.1) die dazu verwendet wird **Kontextfreier Grammatiken** darzustellen.[b] [c]*

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

[a] Der Name kommt daher, dass es eine **Erweiterung** der **Backus-Naur-Form** ist, die hier allerdings **nicht** weiter erläutert wird.
[b] Nebel, „Theoretische Informatik“.
[c] *Grammar Reference — Lark documentation*.
[d] `https://standards.iso.org/ittf/PubliclyAvailableStandards/`.

[1] Wobei beim **Parsen** auch das **Lexen** inbegriffen ist.
[2] *Lark - a parsing toolkit for Python*.
[3] Shinan, *lark*.

Definition 2.4: Dialekt der EBNF aus Lark

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

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

[a] `https://lark-parser.readthedocs.io/en/latest/grammar.html`.
[b] Bzw. kann der ∗-Quantor auch **keinmal** wiederholen bedeuteten.

Das **Lark Parsing Toolkit** wurde vor allem deswegen gewählt, weil es sehr **einfach in der Verwendung** ist. Andere derartige Tools, wie z.B. **ANTLR**[4] sind **Parser Generatoren**, die zur **Konkretten Syntax** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkretten Syntax** direkt ein Programm in dieser Konkretten Syntax **parsen** und einen **Ableitungsbaum** dafür generieren zu können.

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

Neben den **Konkretten Syntaxen**[5], die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterter Back-Naur-Form** spezifiziert sind, werden in den folgenden Unterkapiteln die **Abstrakte Syntaxen**, welche spezifzieren, welche Kompositionen für die **Abstrakter Syntaxbaums** der verschiedenden **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben, die allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit** hat.

Die Notation für die **Abstrakte Syntax** unterscheidet sich bewusst von der **Erweiterten Backus-Naur-Form**, da in der Abstrakten Syntax **Kompositionen von Knoten** beschrieben werden, die **klar auszumachen** sind, wodurch es die Grammatik nur **unnötig verkomplizieren** würde, wenn man die **Erweiterte Backus-Naur-Form** verwenden würde. Es gibt leider **keine** Standardnotation für die **Abstrakte Syntax**, die sich deutlich durchgesetzt hat, daher wird für die Abstrakte Syntaxen eine eigene **Abstract Syntax Form Notation** (Definition 2.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrette** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 2.5: Abstrakte Syntax Form (ASF)

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

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

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

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

2.1 Lexikalische Analyse

Für die **Lexikalische Analyse** ist es nur notwendig eine Grammatik zu definieren, die den Teil der **Konkretten Syntax** beschreibt, der die **verschiedenen Pattern** für die verschiedenen Token der Sprache L_{PicoC} beschreibt, also den Teil der für die **Lexikalische Analyse** wichtig ist. Diese Grammatik wird dann vom **Lark Parsing Toolkit** dazu verwendet ein Programm in **Konkretter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel 1.3 erläutert ist.

2.1.1 Konkrette Syntax für die Lexikalische Analyse

In der Grammatik 2.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokennamen** oder einen **Teil der Beschreibung** eines **Tokennamen**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol `NUM` um einen **Tokennamen**, der durch die **Produktion** `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` beschrieben wird und beschreibt, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann. Das ist daran festzumachen, dass das Nicht-Terminalsymbol `NUM` in keiner anderen Produktion vorkommt, die auf der **linken Seite** des **„kann abgeleitet werden zu“-Symbols** `::=` ebenfalls ein **großgeschriebenen** Nicht-Terminalsymbol hat. Dagegen dient das **großgeschriebene** Nicht-Terminalsymbol `DIG_NO_0` aus der Produktion `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` nur zu Beschreibung von `NUM`.

Die in der Grammatik 2.1.1 definierten **Nicht-Terminalsymbole** können in der Grammatik 2.2.8 der **Konkretten Syntax** für die **Syntaktischen Analayse** verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl `NUM` stehen darf.

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

In Lark steht eine Zahl `.ZAHL`, die an ein **Nicht-Terminalsymbol** angehängt ist, dass auf der linken Seite des **„kann abgeleitet werden zu“-Symbols** `::=` einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. Es gibt den Fall, dass ein Wort von mehreren Produktionen erkannt wird, z.B. wird das Wort `int` sowohl von der Produktion `NAME`, als auch von der Produktion `INT_DT` erkannt. Daher ist es notwendig für `INT_DT` eine **Priorität** `INT_DT.2` zu setzen[6], damit das Wort `int` den **Tokennamen** `INT_DT` zugewiesen bekommt und nicht `NAME`.

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

Der **Basic Lexer** des **Lark Parsing Toolkit** funktioniert grundlegend so wie es im Unterkapitel 1.3 erklärt wurde, allerdinds berücksichtigt der **Basic Lexer** ebenfalls **Priortiäten**, sodass für den aktuellen Index im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Wort** an dem aktuellen Index im Eingabeprogramm erkennt, bekommt es direkt den

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

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

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

Grammatik 2.1.1: *Grammatik der Konkretten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF*

2.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 2.1 dazu verwendet die Konstruktion eines **Abstrakter Syntaxbaums** in seinen einzelnen **Zwischenschritten** zu erläutern.

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

Code 2.1: *PicoC-Code des Codebeispiels*

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

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

Code 2.2: *Tokens für das Codebeispiel*

2.2 Syntaktische Analyse

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

2.2.1 Umsetzung von Präzidenz und Assoziativität

Die Programmiersprache L_{PicoC} hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache L_C[7]. Die **Präzidenzregeln** der Programmiersprache L_{PicoC} sind in Tabelle 2.1 aufgelistet.

[7] *C Operator Precedence - cppreference.com*.

Präzidenzstufe	Operatoren	Beschreibung	Assoziativität
1	`a()`	Funktionsaufruf	Links, dann rechts →
	`a[]`	Indexzugriff	
	`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`	Logiches UND	
11	`a||b`	Logisches ODER	
12	`a=b`	Zuweisung	Rechts, dann links ←

Tabelle 2.1: *Präzidenzregeln von PicoC*

Würde man diese **Operatoren** ohne Beachtung von **Präzidenzreglen** (Definiton 1.28) und **Assoziativität** (Definition 1.27) in eine Grammatik verarbeiten wollen, so könnte eine Grammatik, wie Grammatik 2.2.1 dabei rauskommen.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*exp*")"	*L_Arith +*
un_op	::=	"−" | "∼" | "!" | "∗" | "&"	
un_exp	::=	*un_op exp*	
bin_op	::=	"∗" | "/" | "%" | "+" | "−" | "&" | "∧" | "|"	
		| "<" | "<=" | ">" | ">=" | "!=" | "==" | "&&" | "||"	
bin_exp	::=	*exp bin_op exp*	
exp	::=	*prim_exp* | *un_exp* | *bin_exp*	

Grammatik 2.2.1: *Undurchdachte Konkrette Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF*

Die Grammatik 2.2.1 ist allerdings **mehrdeutig**, d.h. verschiedene **Linksableitungen** in der Grammatik können zum selben **Wort** abgeleitet werden. Z.B. kann das Wort `3 * 1 && 4` sowohl über die **Linksableitung** 2.5.1 als auch über die **Linksableitung** 2.5.2 abgeleitet werden.

$$\begin{aligned} \texttt{exp} &\Rightarrow \texttt{bin\_exp} \Rightarrow \texttt{exp bin\_op exp} \Rightarrow \texttt{bin\_exp bin\_op exp} \\ &\Rightarrow \texttt{exp bin\_op exp bin\_op exp} \Rightarrow^* \texttt{3 * 1 \&\& 4} \end{aligned} \tag{2.5.1}$$

$$\begin{aligned} &\texttt{exp} \Rightarrow \texttt{bin\_exp} \Rightarrow \texttt{exp bin\_op exp} \Rightarrow \texttt{prim\_exp bin\_op exp} \Rightarrow \texttt{NUM bin\_op exp} \\ &\Rightarrow \texttt{3 bin\_op exp} \Rightarrow \texttt{3 * exp} \Rightarrow \texttt{3 * bin\_exp} \Rightarrow \texttt{3 * exp bin\_exp exp} \Rightarrow^* \texttt{3 * 1 \&\& 4} \end{aligned} \tag{2.5.2}$$

Beide **Wörter** sind **gleich**, allerdings sind die **Ableitungsbäume unterschiedlich**, wie in Abbildung 2.1 zu sehen ist.

Abbildung 2.1: *Ableitungsbäume zu den beiden Ableitungen*

Der **linke Baum** entspricht Ableitung 2.5.1 und der **rechte Baum** entspricht Ableitung 2.5.2. Würde man in den Ausdrücken, die von diesen Bäumen darsgestellt sind in **Klammern** setzen, um die **Präzidenz** sichtbar zu machen, so würde Ableitung 2.5.1 die Klammerung `(3 * 1) && 4` haben und die Ableitung 2.5.2 die Klammerung `3 * (1 && 4)` haben.

Aus diesem Grund ist es wichtig die **Präzidenzregeln** und die **Assoziativität** der Operatoren beim Erstellen der Grammatik miteinzubeziehen. Hierzu wird nun Tabelle 2.1 betrachtet. Für jede **Präzidenzstufe** in der Tabelle 2.1 wird eine eigene Regel erstellt werden, wie es in Grammatik 2.2.2 dargestellt ist. Zudem braucht es eine **Produktion** `prim_exp` für die höchste **Präzidenzstufe**, welche **Literale**, wie `'c'`, `5` oder `var` und geklammerte Ausdrücke wie `(3 && 14)` abdeckt.

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

Grammatik 2.2.2: *Durchdachte Konkrette Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF*

Einigen **Bezeichnungen** der **Produktionen** sind in Tabelle 2.2 ihren jeweiligen **Operatoren** zugeordnet für welche sie zuständig sind.

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

Tabelle 2.2: *Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren*

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

Bei z.B. der Produktion `un_exp` in 2.2.3 für die **rechtsassoziativen unären Operatoren** `-a`, `!a` `~a`, `*a` und `&a` ist die **Alternative** `un_op un_exp` dafür zuständig, dass diese unären Operatoren **rechtsassoziativ** geschachtelt werden können (z.B. `!-~42`). Die Alternative `post_exp` ist dafür zuständig, dass die Produktion auch **terminieren** kann und es auch möglich ist auschließlich einen Ausdruck **höherer Präzidenz** (z.B. `42`) zu haben.

$$un\_exp \quad ::= \quad un\_op\ un\_exp \quad | \quad post\_exp$$

Grammatik 2.2.3: *Beispiel für eine unäre rechtsassoziative Produktion*

Bei z.B. der Produktion `post_exp` in 2.2.4 für die **linksassoziativen unären Operatoren** `a()`, `a[]` und `a.b` sind die Alternativen `post_exp"["logic_or"]"` und `post_exp"."name` dafür zuständig, dass diese unären Operatoren **linksassoziativ** geschachtelt werden können (z.B. `ar[3][1].car[4]`). Die Alternative `name"("fun_args")"` ist für einen **einzelnen Funktionsaufruf** zuständig. Die Alternative `prim_exp` ist dafür zuständig, dass die Produktion nicht nur bei `name"("fun_args")"` **terminieren** kann und es auch möglich ist auschließlich einen Ausdruck der **höchsten Präzidenz** (z.B. `42`) zu haben.

$$post\_exp \quad ::= \quad post\_exp\text{"["}logic\_or\text{"]"} \quad | \quad post\_exp\text{"."}name \quad | \quad name\text{"("}fun\_args\text{")"} \quad | \quad prim\_exp$$

Grammatik 2.2.4: *Beispiel für eine unäre linksassoziative Produktion*

Bei z.B. der Produktion `prec2_exp` in 2.2.5 für die **binären linksassoziativen Operatoren** `a+b` und `a-b` ist die **Alternative** `arith_prec2 prec2_op arith_prec1` dafür zuständig, dass **mehrere** Operationen der Präzidenzstufe 4 in Folge erkannt werden können[9] (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzidenzstufe 4

[8] **Geklammerte Ausdrücke** werden nämlich von `prim_exp` erkannt, welches eine höhere **Präzidenzstufe** hat.
[9] Bezogen auf Tabelle 2.1.

haben). Das **Nicht-Terminalsymbol** `arith_prec1` auf der **rechten Seite** ermöglicht es, dass zwischen den Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzidenzstufe 4 haben und `/` Präzidenzstufe 3). Mit der Alternative `arith_prec1` ist es möglich, dass auschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B. `1 / 4`).

$$arith\_prec2 \quad ::= \quad arith\_prec2 \;\; prec2\_op \;\; arith\_prec1 \quad | \quad arith\_prec1$$

Grammatik 2.2.5: *Beispiel für eine linksassoziative Produktion*

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

$$arith\_prec2 \quad ::= \quad arith\_prec1 \;\; (prec2\_op \;\; arith\_prec1)*$$

Grammatik 2.2.6: *Beispiel für eine linksassoziative Produktion*

Die von Produktion 2.2.6 erkannten Ausdrücke sind dieselben, wie für die Produktion 2.2.5, allerdings ist die Produktion2.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 2.3) definierten `*`-Operator, um mehrere Operationen der Präzidenzstufe 4 in Folge erkennen zu können (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzidenzstufe 4 haben).

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

[a] Darunter zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird **nicht**.

Alle **Operatoren** der Sprache L_{PicoC} sind also entweder **binär** und **linksassoziativ** (z.B. `a*b`, `a-b`, `a>=b` oder `a&&b`), **unär** und **rechtsassoziativ** (z.B. `&a` oder `!a`) oder **unär** und **linksassoziativ** (z.B. `a[]` oder `a()`). Somit ergibt sich die Grammatik 2.2.7.

$prec1\_op$	::=	$"*" \mid "/" \mid "\%"$	*L_Misc*
$prec2\_op$	::=	$"+" \mid "-"$	
rel_op	::=	$"<" \mid "<=" \mid ">" \mid ">="$	
eq_op	::=	$"==" \mid "!="$	
fun_args	::=	$[logic\_or(",", logic\_or)*]$	
$prim\_exp$	::=	$name \mid NUM \mid CHAR \mid "("logic\_or")"$	*L_Arith*
$post\_exp$	::=	$post\_exp"["logic\_or"]" \mid post\_exp"."name \mid name"("fun\_args")"$	*+ L_Array*
	$\mid$	$prim\_exp$	*+ L_Pntr*
un_exp	::=	$un\_op\ un\_exp \mid post\_exp$	*+ L_Struct*
$arith\_prec1$	::=	$arith\_prec1\ prec1\_op\ un\_exp \mid un\_exp$	*+ L_Fun*
$arith\_prec2$	::=	$arith\_prec2\ prec2\_op\ arith\_prec1 \mid arith\_prec1$	
$arith\_and$	::=	$arith\_and\ "\&"\ arith\_prec2 \mid arith\_prec2$	
$arith\_oplus$	::=	$arith\_oplus\ "\wedge"\ arith\_and \mid arith\_and$	
$arith\_or$	::=	$arith\_or\ "|"\ arith\_oplus \mid arith\_oplus$	
rel_exp	::=	$rel\_exp\ rel\_op\ arith\_or \mid arith\_or$	*L_Logic*
eq_exp	::=	$eq\_exp\ eq\_op\ rel\_exp \mid rel\_exp$	
$logic\_and$	::=	$logic\_and\ "\&\&"\ eq\_exp \mid eq\_exp$	
$logic\_or$	::=	$logic\_or\ "||"\ logic\_and \mid logic\_and$	
$assign\_stmt$	::=	$un\_exp\ "="\ logic\_or";"$	*L_Assign*

Grammatik 2.2.7: *Durchdachte Konkrette Syntax für Operatorpräzidenz in EBNF*

2.2.2 Konkrette Syntax für die Syntaktische Analyse

Die gesamte Grammatik 2.2.8, welche die **Konkrette Syntax** der Sprache L_{PicoC} für die **Syntaktische Analyse** beschreibt ergibt sich wenn man die Grammatik 2.2.7 um die **restliche Syntax** der Sprache L_{PicoC} erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 2.2.7 erläutert ergibt.

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrette Syntax** an die **aktuellste konstenlos auffindbare Version** der echten Grammatik *ANSI C grammar (Yacc)* der Sprache L_C angepasst[10], damit es sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C, wobei z.B. die Compiler **GCC**[11] und **Clang**[12] zu nennen wären.

In der Grammatik 2.2.8, welche die **Konkrette Syntax** der Sprache L_{PicoC} für die **Syntaktische Analyse** beschreibt, werden einige der **Tokennamen** aus der Grammatik 2.1.1 der **Konkretten Syntax** für die **Lexikalischen Analyse** verwendet, wie z.B. `NUM` aber auch `name`, welches eine Produktion ist, die mehrere **Tokennamen** unter einem Überbegriff zusammenfasst.

Terminalsymbole, wie `;` oder `&&` gehören eigentlich zur **Lexikalischen** Analyse, jedoch erlaubt das **Lark Parsing Toolkit** um die Grammatik leichter lesbar zu machen einige **Terminalsymbole** einfach direkt in die Grammatik 2.2.8 der **Konkretten Syntax** für die **Syntaktische Analyse** zu schreiben. Der **Tokenname** für diese Terminalsymbole wird in diesem Fall vom **Lark Parsing Toolkit** bestimmt, welches einige sehr häufige verwendete **Terminalsymbole**, wie `;` oder `&&` bereits einen **Tokennamen** zugewiesen hat.

[10] An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts verändert, wie die Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Lex)* und **Lexikalische Analyse** **noauthor'ansi'nodate-2** aus dem Jahre 1985 zeigen.
[11] *GCC, the GNU Compiler Collection - GNU Project*.
[12] *clang: C++ Compiler*.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith + L_Array*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*+ L_Pntr + L_Struct*
		| *input_exp* | *print_exp* | *prim_exp*	*+ L_Fun*
un_exp	::=	*un_op un_exp* | *post_exp*	
input_exp	::=	"*input*""("")"	
print_exp	::=	"*print*""("*logic_or*")"	
*arith_prec*1	::=	*arith_prec*1 *prec*1*_op un_exp* | *un_exp*	
*arith_prec*2	::=	*arith_prec*2 *prec*2*_op arith_prec*1 | *arith_prec*1	
arith_and	::=	*arith_and* "&" *arith_prec*2 | *arith_prec*2	
arith_oplus	::=	*arith_oplus* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_oplus* | *arith_oplus*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_op rel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
type_spec	::=	*prim_dt* | *struct_spec*	*L_Assign_Alloc*
alloc	::=	*type_spec pntr_decl*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	
initializer	::=	*logic_or* | *array_init* | *struct_init*	
init_stmt	::=	*alloc* "=" *initializer*";"	
const_init_stmt	::=	"*const*" *type_spec name* "=" *NUM*";"	
pntr_deg	::=	"*"*	*L_Pntr*
pntr_decl	::=	*pntr_deg array_decl* | *array_decl*	
array_dims	::=	("["*NUM*"]")*	*L_Array*
array_decl	::=	*name array_dims* | "("*pntr_decl*")"*array_dims*	
array_init	::=	"{"*initializer*("," *initializer*) * "}"	
array_subscr	::=	*post_exp*"["*logic_or*"]"	
struct_spec	::=	"*struct*" *name*	*L_Struct*
struct_params	::=	(*alloc*";")+	
struct_decl	::=	"*struct*" *name* "{"*struct_params*"}"	
struct_init	::=	"{""."*name*"="*initializer*	
		("," "."name"="initializer)*"}"	
struct_attr	::=	*post_exp*"."*name*	
if_stmt	::=	"*if*""("*logic_or*")" *exec_part*	*L_If_Else*
if_else_stmt	::=	"*if*""("*logic_or*")" *exec_part* "*else*" *exec_part*	
while_stmt	::=	"*while*""("*logic_or*")" *exec_part*	*L_Loop*
do_while_stmt	::=	"*do*" *exec_part* "*while*""("*logic_or*")"";"	

Grammatik 2.2.8: *Grammatik der Konkretten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1*

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

Grammatik 2.2.9: *Grammatik der Konkretten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2*

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

2.2.3 Ableitungsbaum Generierung

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

Definition 2.6: Earley Parser

*Ist ein Algorithmus für das **Parsen** von **Wörtern** einer **Kontextfreien Sprache**, der ein **Chart Parser** ist, welcher einen **top-down** arbeitenden **Earley Recognizer** (Defintion 2.7) nutzt, um einen **Ableitungsbaum** zu konstruieren.*

Definition 2.7: Earley Recognizer

*Ist ein **Recognizer** der für **alle Kontextfreien Sprachen** das **Wortproblem** entscheiden kann.*

*Bevor dieser **Algorithmus** erklärt wird müssen noch einige **Symbole** und **Notationen** erklärt werden:*

- *α, β, γ stellen eine **beliebige Folge** von **Grammatiksymbolen**[a] dar*
- *A und B stellen **Nicht-Terminalsymbole** dar*

- *a stellt ein **Terminalsymbol** dar*
- ***Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine **Produktion**, in der α **bereits geparst** wurde und β **noch geparst** werden muss*

*und davor müssen noch einige **Begriffe definiert** werden:*

- ***Zustandsmenge:** Für jeden **Index** des **Eingabeworts** wird eine **Zustandsmenge** generiert. Die **Zustandsmenge** für die **Index** j im **Eingabewort** wird mit $Z(j)$ bezeichnet.*
- ***Zustand der Zustandsmenge:** Ist ein **Tupel** $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die **momentane Produktion** ist, die erkannt werden soll, die bis **Punkt** $\bullet$ geparst wurde und i der **Index** im **Eingabewort** ist, ab welchem der Versuch der Erkennung dieser **Produktion begann**.*

*Der **Ablauf** des Algorithmus ist wie folgt:*

- ***Input:** **Wort** w und **Grammatik** $G = \langle N, \Sigma, P, A \rangle$*
- ***Output:** 0 wenn $w \notin L(G)$[b] und 1 wenn $w \in L(G)$*

1. ***initialisiere** $Z(0)$ mit der **Produktion**, welches das **Startsymbol** A auf der **linken Seite** des **„kann abgeleitet werden zu"-Symbols** $::=$ hat*
2. *es werden **wiederholt** die folgenden **Operationen ausgeführt**:*
   - ***Voraussage:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet B\gamma, i)$ hat, wird für jede **Produktion** $(B ::= \beta)$ in der Grammatik, die ein B auf der **linken Seite** des **„kann abgeleitet werden zu"-Symbols** $::=$ hat ein **Zustand** $(B ::= \bullet\beta, j)$ zur **Zustandsmenge** $S(j)$ hinzugefügt*
   - ***Überprüfung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet a\gamma, i)$ hat wird der **Zustand** $(A ::= \alpha a \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j+1)$ hinzugefügt*
   - ***Vervollständigung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(B ::= \beta\bullet, i)$ hat werden alle **Zustände** in $Z(i)$ gesucht, welche die Form $(A ::= \alpha \bullet B\gamma, i)$ haben und es wird der **Zustand** $(A ::= \alpha B \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt*

   bis:

   - *der **Zustand** $(A ::= \beta\bullet, 0)$ in der **Zustandsmenge** $Z(n)$ auftaucht, wobei $A ::= \beta$ die **Produktion** ist, welche das **Startsymbol** A auf der **linken Seite** des **„kann abgeleitet werden zu"-Symbols** $::=$ hat $\Rightarrow w \in L(G)$*
   - ***keine Zustände** mehr einer **Zustandsmenge** hinzugefügt werden können $\Rightarrow w \notin L(G)$*

[a] Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[b] $L(G)$ ist die **Sprache**, welche durch die **Grammatik** G beschrieben wird.

2.2.3.1 Codebeispiel

Der **Ableitungsbaum**, der mithilfe des **Earley Parsers** und der **Token** der **Lexikalischen Analyse** aus dem Beispiel in Code 2.1 generiert wurde, ist in Code 2.3 zu sehen. Im Code 2.3 wurden einige Zeilen **markiert**, die später in Unterkapitel 2.2.4.1 zum Vergleich wichtig sind.

```
file
  ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt
  decls_defs
    decl_def
      struct_decl
        name        st
        struct_params
          alloc
            type_spec
              prim_dt        int
            pntr_decl
              pntr_deg        *
              array_decl
                pntr_decl
                  pntr_deg        *
                  array_decl
                    name        attr
                    array_dims
                array_dims
                  4
                  5
    decl_def
      fun_def
        type_spec
          prim_dt        void
        pntr_deg
        name        main
        fun_params
        decl_exec_stmts
          decl_part
            decl_exp_stmt
              alloc
                type_spec
                  struct_spec
                    name        st
                pntr_decl
                  pntr_deg        *
                  array_decl
                    pntr_decl
                      pntr_deg        *
                      array_decl
                        name        var
                        array_dims
                          3
                          2
                    array_dims
```

Code 2.3: *Ableitungsbaum nach Ableitungsbaum Generierung*

2.2.3.2 Ausgabe des Ableitunsgbaums

Die Ausgabe des **Ableitungsbaums** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Grammatik den **linken Seiten** des **„kann abgeleitet werden zu“-Symbols** ::=[13] entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition 1.36 eines **Ableitungsbaums** auch schon definiert

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

ist. Die **EBNF-Grammatik** 2.2.8 des **PicoC-Compilers** erlaubt es allerdings auch, dass in einem **Blatt** garnichts ε steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("["NUM"]")*` gibt, in denen auch das **leere Wort** ε abgeleitet werden kann.

Die Ausgabe des **Abstrakter Syntaxbaum** ist bewusst so gewählt, dass sie sich optisch vom **Ableitungsbaum** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase** geschrieben sind, im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des **„kann abgeleitet werden zu“-Symbols** `::=`.

2.2.4 Ableitungsbaum Vereinfachung

Der **Ableitungsbaum** in Code 2.3, dessen Generierung in Unterkapitel 2.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Tramsformers** ein **Abstrakter Syntaxbaum** generiert werden kann. Das Problem ist, dass um den den **Datentyp** einer Variable in der Programmiersprache L_C und somit auch die Programmiersprache L_{PicoC} korrekt bestimmen zu können, wie z.B. ein **„Array der Mächtigkeit 3 von Pointern auf Arrays der Mächtigkeit 2 von Integern“** `int (*ar[3])[2]` die **Spiralregel**[14] in der Implementeirung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Grammatik 2.2.8 der **Konkretten Syntax** auf eine spezielle Weise passend spezifiziert.

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

Es sind hierbei mehrere Probleme zu lösen. Hat man den Ausdruck `struct st *(*var[3][2])` wird dieser zu einem **Ableitungsbaum**, wie er in Abbildung 2.2 zu sehen ist.

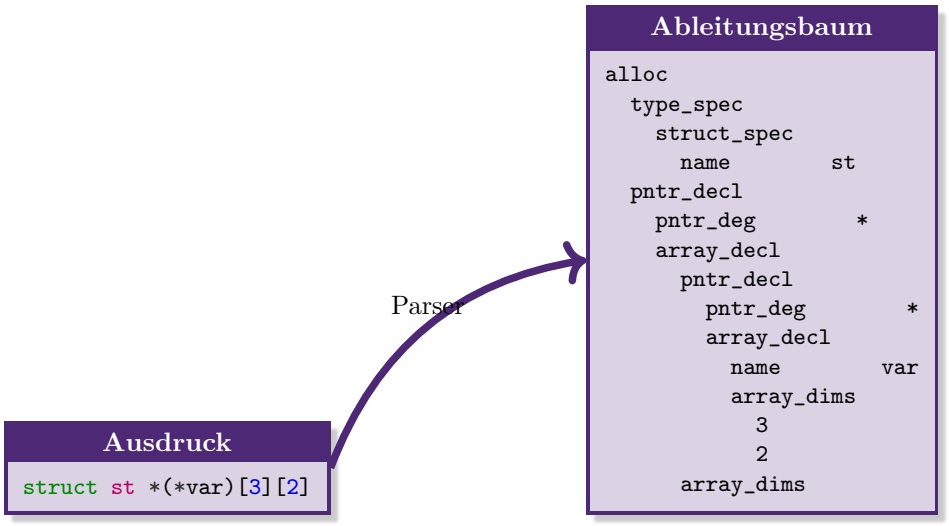


Abbildung 2.2: *Ableitungsbaum nach Parsen eines Ausdrucks*

Dieser **Ableitungsbaum** für den Ausdruck `struct st *(*var[3][2])` hat allerdings einen Aufbau welcher durch die **Syntax** der **Pointerdeklaratoren** `pntr_decl(num, datatype)` und **Arraydeklaratoren** `array_decl(datatype, nums)` bestimmt ist, die **spiralähnlich** ist. Man würde allerdings gerne einen **entarteten Baum** erhalten, bei dem der Datentyp immer im **zweiten Attribut** weitergeht, anstatt abwechselnd

[14] *Clockwise/Spiral Rule*.

im **zweiten** und **ersten**, wie beim **Pointerdeklarator** `pntr_decl(num, datatype)` und **Arraydeklarator** `array_decl(datatype, nums)`. Daher muss beim **ArrayDeclarator** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht werden.

Des Weiteren befindet sich in der **Mitte** dieser **Spirale**, die der **Ableitungsbaum** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`, da der **Ableitungsbaum** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Programms** in **Konkretter Syntax** (z.B. `struct st *(*var[3][2])`) **generiert** wird darstellt. Der **Name der Variable** `name(var)` sollte daher mit dem **innersten Datentyp** `struct st` ausgetauscht werden.

In Abbildung 2.3 ist daher zu sehen, wie der **Ableitungsbaum** aus Abbildung 2.2 mithilfe eines **Visitors** (Definition 1.40) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

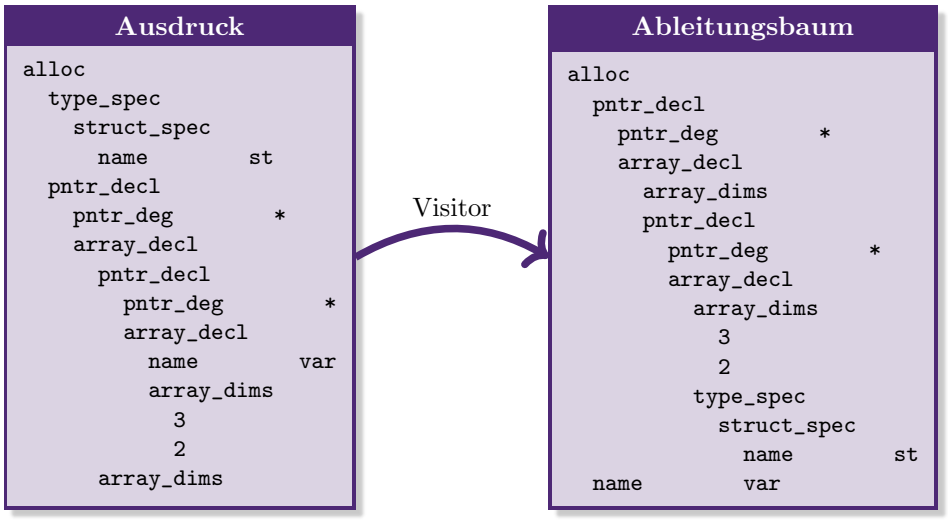


Abbildung 2.3: *Ableitungsbaum nach Vereinfachung*

2.2.4.1 Codebeispiel

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

```
1  file
2    ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3    decls_defs
4      decl_def
5        struct_decl
6          name        st
7          struct_params
8            alloc
9              pntr_decl
10               pntr_deg        *
11               array_decl
12                 array_dims
13                   4
14                   5
15                 pntr_decl
```

```
16                pntr_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        pntr_deg
27        name        main
28        fun_params
29        decl_exec_stmts
30          decl_part
31            decl_exp_stmt
32              alloc
33                pntr_decl
34                  pntr_deg        *
35                  array_decl
36                    array_dims
37                    pntr_decl
38                      pntr_deg        *
39                      array_decl
40                        array_dims
41                          3
42                          2
43                        type_spec
44                          struct_spec
45                            name          st
46                name          var
```

Code 2.4: *Ableitungsbaum nach Ableitungsbaum Vereinfachung*

2.2.5 Abstrakt Syntax Tree Generierung

Nachdem der **Derviation Tree** in Unterkapitel 2.2.4 vereinfacht wurde, ist der **vereinfachte Ableitungsbaum** in Code 2.4 nun dazu geeignet, um mit einem **Transformer** (Definition 1.39) einen **Abstrakter Syntaxbaum** aus ihm zu generieren. Würde man den **vereinfachten Ableitungsbaum** des Ausdrucks `struct st *(*var[3][2])` auf passende Weise in einen **Abstrakter Syntaxbaum** umwandeln, so würde dabei ein **Abstrakter Syntaxbaum** wie in Abbildung 2.4 rauskommen.

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

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

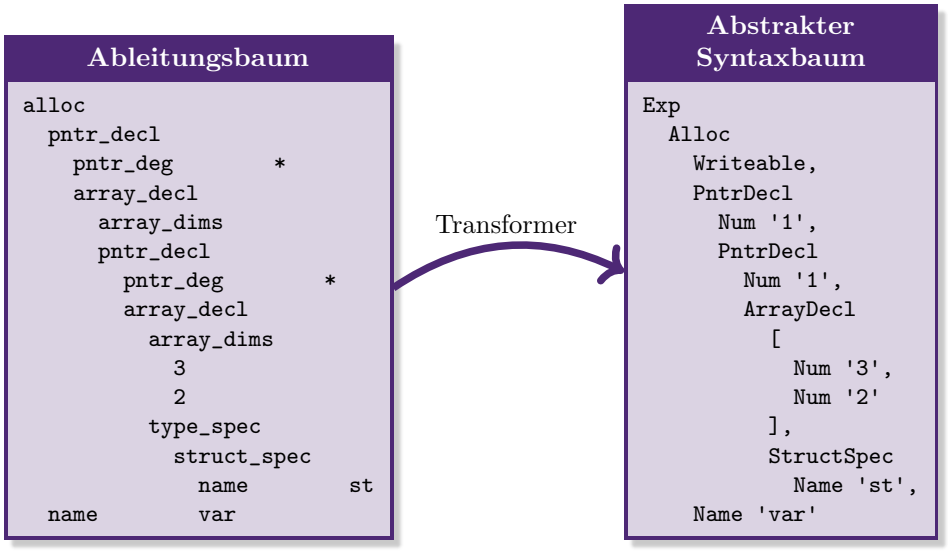


Abbildung 2.4: *Abstrakter Syntaxbaum Generierung ohne Umdrehen*

Dieser **Abstrakter Syntaxbaum** ist für die Weiterverarbeitung ungeeignet, denn für die **Adressberechnung** für eine Aneinandereihung von Zugriffen auf **Pointerelemente**, **Arrayelemente** oder **Structattribute**, welche in Unterkapitel **??** genauer erläutert wird, will man den Datentyp in **umgekehrter Reihenfolge**. Aus diesem Grund muss der **Transformer** bei der Konstruktion des **Abstrakter Syntaxbaum** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie in Abbildung 2.5 zustande, der für die Weiterverarbeitung geeignet ist.

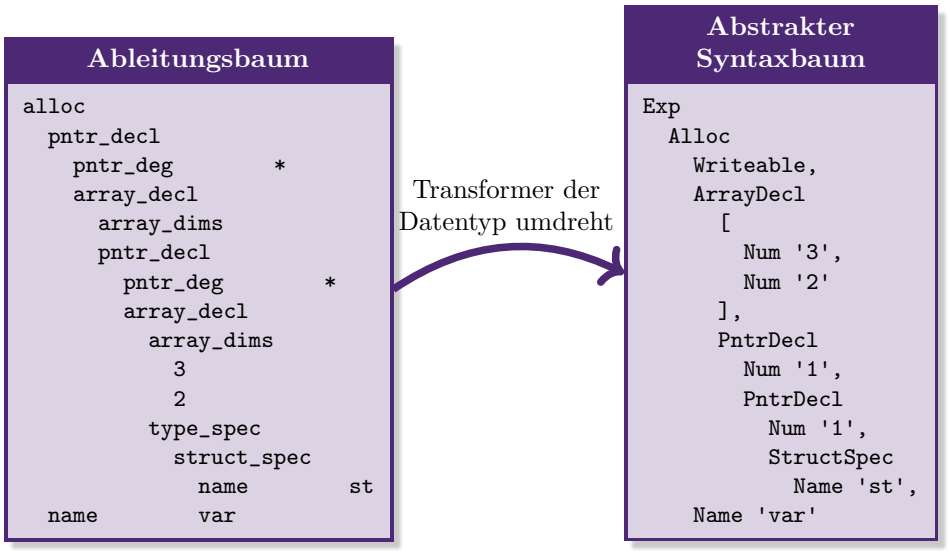


Abbildung 2.5: *Abstrakter Syntaxbaum Generierung mit Umdrehen*

Die Weiterverarbeitung des **Abstrakter Syntaxbaums** geschieht mithilfe von **Passes**, welche im Unterkapitel 1.5 genauer beschrieben werden. Da die Knoten des **Abstrakter Syntaxbaum** anders als beim **Ableitungsbaum** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Grammatik der **Kon-**

kretten Syntax ist es in den folgenden Unterkapiteln 2.2.5.1, 2.2.5.2 und 2.2.5.3 notwendig die **Bedeutung** der einzelnen **PicoC-Knoten**, **RETI-Knoten** und bestimmter **Kompositionen** dieser Knoten zu **dokumentieren**, die alle in den unterschiedlichen von den **Passes** umgeformten **Abstrakter Syntaxbaums** vorkommen.

Des Weiteren gibt die **Abstrakte Syntax** die durch die Grammatik 2.2.1 in Unterkapitel 2.2.5.4 beschrieben wird aufschluss darüber welche **Kompositionen von PicoC-Knoten**, neben den bereits in Tabelle 2.2.10 definierten Kompositionen mit Bedeutung insgesamt überhaupt **möglich** sind.

2.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache L_{PicoC} darstellen. Für die **PicoC-Knoten** wurden möglichst **kurze** und **leicht** verständliche Bezeichner gewählt, da auf diese Weise bei der Implementierung der einzelnen Passes möglichst **viel Code in eine Zeile** passt und dieser Code auch durch leicht verständliche Bezeichner von Knoten **intuitiv verständlich** sein sollte[16]. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 2.3 mit einem **Bschreibungstext** dokumentiert.

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

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

Tabelle 2.3: *PicoC-Knoten Teil 1*

PiocC-Knoten	Beschreibung
`Exp(exp, datatype, error_data)`	Container für einen **beliebigen Ausdruck**, dessen Ergebnis auf den **Stack** soll. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` im **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Stack(num)`	Container, der für das **temporäre** Ergebnis einer Berechnung, das `num` Speicherzellen relativ zum **Stackpointer Register** `SP` steht.
`Stackframe(num)`	Container, der für eine Variable steht, die `num` Speicherzellen relativ zum **Begin-Aktive-Funktion Register** `BAF` steht.
`Global(num)`	Container, der für eine Variable steht, die `num` Speicherzellen relativ zum **Datensegment Register** `DS` steht.
`StackMalloc(num)`	Container, der für das **Allokieren** von `num` Speicherzellen auf dem **Stack** steht.
`PntrDecl(num, datatype)`	Container, der für den **Pointerdatentyp** steht: `<prim_dt> *<var>`, wobei das **Attribut** `num` die **Anzahl zusammengefasster Pointer** angibt und `datatype` der Datentyp ist, auf den der oder die **Pointer** zeigen.
`Ref(exp, datatype, error_data)`	Container, der für die Anwendung des **Referenz-Operators** `&<var>` steht und die **Adresse** einer **Location** (Definition 1.47) auf den Stack schreiben soll, die über `exp` eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` im **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Deref(lhs, exp)`	Container für den **Indexzugriff** auf einen **Array-** oder **Pointerdatentyp**: `<var>[<i>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`ArrayDecl(nums, datatype)`	Container, der für den **Arraydatentyp** steht: `<prim_dt> <var>[<i>]`, wobei das **Attribut** `nums` eine Liste von `Num('x')` ist, die die **Dimensionen** des Arrays angibt und `datatype` der Datentyp ist, der über das Anwenden von `Subscript()` auf das Array zugreifbar ist.
`Array(exps, datatype)`	Container für den **Initializer** eines **Arrays**, dessen Einträge `exps` weitere Initializer für eine **Array-Dimension** oder ein Initializer für ein **Struct** oder ein **Logischer Ausdruck** sein können, z.B. `{{1, 2}, {3, 4}}`. Des Weiteren besitzt er ein verstecktes Attribut `datatype`, welches für den **PicoC-ANF Pass** Informationen transportiert, die für **Fehlermeldungen** wichtig sind.
`Subscr(exp1, exp2)`	Container für den **Indexzugriff** auf einen **Array-** oder **Pointerdatentyp**: `<var>[<i>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`StructSpec(name)`	Container für einen selbst definierten **Structdatentyp**: `struct <name>`, wobei das **Attribut** `name` festlegt, welchen **selbst definierte** Structdatentyp dieser Container-Knoten repräsentiert.
`Attr(exp, name)`	Container für den **Attributzugriff** auf einen **Structdatentyp**: `<var>.<attr>`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `name` das Attribut ist, auf das zugegriffen werden soll.

Tabelle 2.4: *PicoC-Knoten Teil 2*

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

Tabelle 2.5: *PicoC-Knoten Teil 3*

PiocC-Knoten	Beschreibung
`FunDef(datatype, name, allocs, stmts_blocks)`	Container für eine **Funktionsdefinition**, z.B. `<datatype> <fun_name>(<datatype> <param>) {<stmts>}`, wobei `datatype` der **Rückgabewert** der Funktion ist, `name` der **Bezeichner** der Funktion ist, `allocs` die **Parameter** der Funktion sind, wobei der **Container-Knoten** `Alloc(type_spec, datatype, name)` als Cotainer für die Parameter dient und `stmts_blocks` eine Liste von **Statemetns** bzw. **Blöcken** ist, welche diese Funktion beinhaltet.
`NewStackframe(fun_name, goto_after_call)`	Container für die **Erstellung** eines neuen **Stackframes** und Speicherung des Werts des `BAF`-Registers der **aufrufenden Funktion** und der **Rücksprungadresse** nacheinander an den **Anfang** des neuen **Stackframes**. Das Attribut `fun_name` stehte dabei für den Bezeichner der Funktion, für die ein neuer **Stackframe** erstellt werden soll. Das Attribut `fun_name` dient später dazu den **Block** dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attributen gespeichert hat. Des Weiteren ist das Attribut `goto_after_call` ein `GoTo(Name('addr@next_instr'))`, welches später durch die **Adresse** des Befehls, der direkt auf die **Jump Instruction** folgt, ersetzt wird.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes** durch das **Wiederherstellen** des im noch **aktuellen Stackframe** gespeicherten Werts des `BAF`-Registes der **aufrufenden Funktion** und das Setzen des `SP`-Registers auf den Wert des `BAF`-Registesr **vor** der Wiederherstellung.
`File(name, decls_defs_blocks)`	Container für alle **Funkionen** oder **Blöcke**, welche eine Datei als Ursprung haben, wobei `name` der **Dateiname** der Datei ist, die erstellt wird und `decls_defs_blocks` eine Liste von **Funktionen** bzw. **Blöcken** ist.
`Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`	Container für **Statements**, der auch als **Block** bezeichnet wird, wobei das Attribut `name` der Bezeichners des **Labels** (Definition 2.8) des Blocks ist und `stmts_instrs` eine **Liste von Statements** oder **Instructions**. Zudem besitzt er noch 3 versteckte Attribute, wobei `instrs_before` die Zahl der **Instructions** vor diesem **Block** zählt, `num_instrs` die Zahl der Instructions ohne Kommentare in diesem Block zählt, `param_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **Parameter** der Funktion belegt werden müssen und `local_vars_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **lokalen Variablen** der Funktion belegt werden müssen.
`GoTo(name)`	Container für ein `Goto` zu einem anderen **Block**, wobei das Attribut `name` der Bezeichner des **Labels** des Blocks ist zu dem Gesprungen werden soll.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar**, den der Compiler selber während des **Kompiliervorangs** erstellt, der im **RETI-Interpreter** selbst später **nicht** sichtbar sein wird, aber in den **Immediate**-Dateien, welche die **Abstrakter Syntaxbaums** nach den verschiedenen **Passes** enthalten.
`RETIComment(value)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer **RETI-CPU nicht umsetzbar** ist und auch nicht sinnvoll wäre umzusetzen. Der **Kommentar** ist im Attribut **value**, welches jeder Knoten besitzt gespeichert.

Tabelle 2.6: *PicoC-Knoten Teil 4*

Definition 2.8: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a]Thiemann, „Compilerbau“.

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

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** `value` und `position`, wobei **value** bei einem **Token-Knoten** (Definition 2.9) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 2.10) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

Definition 2.9: Token-Knoten

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

***Token-Knoten** entsprechen im Abstrakter Syntaxbaum **Blättern**.*[a]

[a]Thiemann, „Compilerbau“.

Definition 2.10: 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 **Konkretten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.*[a]

***Container-Knoten** entsprechen im Abstrakter Syntaxbaum **Inneren Knoten**.*[b]

[a]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.
[b]Thiemann, „Compilerbau“.

2.2.5.2 RETI-Knoten

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache L_{RETI} darstellen. Für die **RETI-Knoten** wurden aus bereits in Unterkapitel 2.2.5.1 erläutertem Grund, genauso wie für die **RETI-Knoten** möglichst **kurze** und **leicht** verständliche Bezeichner gewählt. Alle **RETI-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 2.2.5.1 mit einem **Beschreibungstext** dokumentiert.

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

[a] Scholl, „Betriebssysteme“

Tabelle 2.7: *RETI-Knoten*

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

In Tabelle 2.8 sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax** 2.2.8 enthalten sind.

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

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

Um die obige Tabelle 2.8 nicht mit unnötig viel repetetiven Inhalt zu füllen, wurden die zahlreichen Kompostionen **ausgelassen**, bei denen einfach nur `exp` durch $\texttt{Stack}(\texttt{Num}('\texttt{x}'))$, $\texttt{x} \in \mathbb{N}$ ersetzt wurde.

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

2.2.5.4 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache L_{PicoC} wird durch die Grammatik 2.2.10 beschrieben.

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

Grammatik 2.2.10: *Abstrakte Syntax der Sprache* L_{PiocC}

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

2.2.5.5 Codebeispiel

In Code 2.5 ist der **Abstrakter Syntaxbaum** zu sehen, der aus dem **vereinfachten Ableitungsbaum** aus Code 2.4 mithilfe eines **Transformers** generiert wurde.

```
File
  Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc
          Writeable,
          PntrDecl
            Num '1',
            ArrayDecl
              [
                Num '4',
                Num '5'
              ],
              PntrDecl
                Num '1',
                IntType 'int',
          Name 'attr'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '3',
                Num '2'
              ],
              PntrDecl
                Num '1',
                PntrDecl
                  Num '1',
                  StructSpec
                    Name 'st',
            Name 'var'
      ]
  ]
```

Code 2.5: *Aus vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum*

2.2.5.6 Ausgabe des Abstrakter Syntaxbaum

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

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

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

2.3 Code Generierung

Nach der Generierung eines **Abstrakter Syntaxbaum** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 1.4, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** im **Abstrakter Syntaxbaum** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** der Sprache L_{PicoC}, der durch die **Abstrakte Syntax** in Grammatik 2.2.10 spezifiziert ist in einen entsprechenden **Abstrakter Syntaxbaum** 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 1.5). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagram** (siehe Unterkapitel 1.1.1) in Abbildung 2.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte. Und selbst dieses **T-Diagram** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu $\texttt{X}_{86\_64}$ kompiliert wurde usw.

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

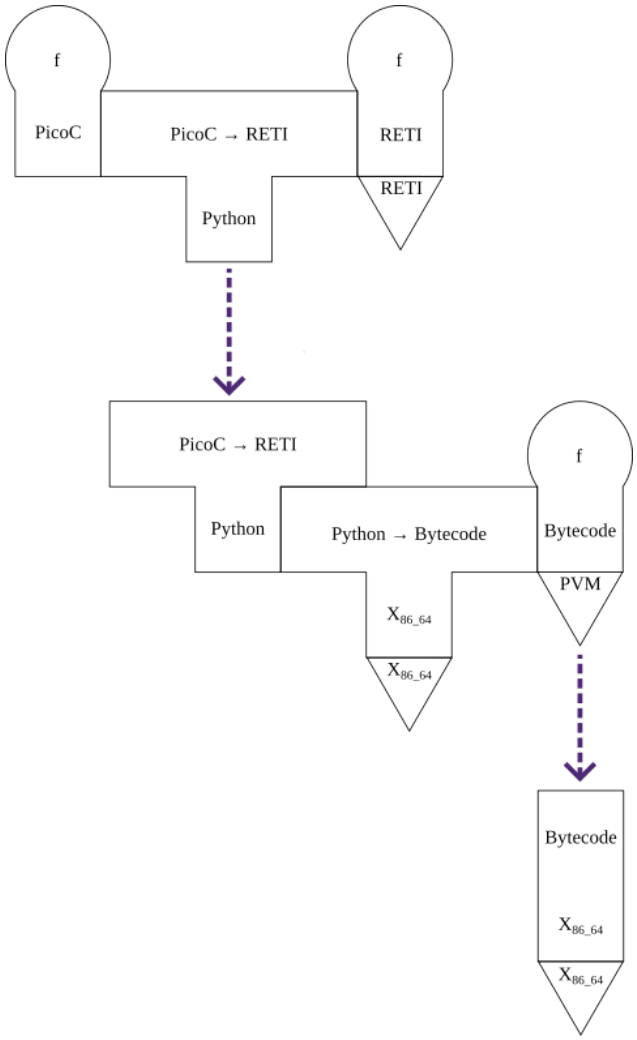


Abbildung 2.6: *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagram** in Abbildung 2.6 lässt sich zusammenfassen, sodass man das **T-Diagram** in Abbildung 2.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in $\mathtt{X}_{86\_64}$-Maschienensprache geschrieben ist.

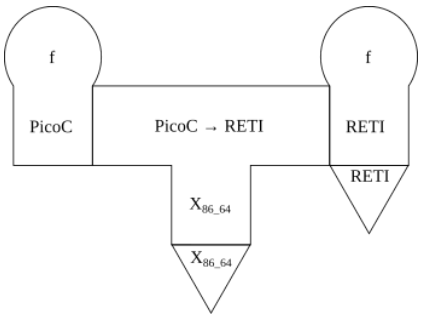


Abbildung 2.7: *Cross-Compiler Kompiliervorgang Kurzform*

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

der Kompilierprozess im Folgenden im **horinzontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 2.8 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 **Abstrakter Syntaxbaum** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstrakter Syntaxbaum** in seiner eigenen **Sprache** zu generieren.

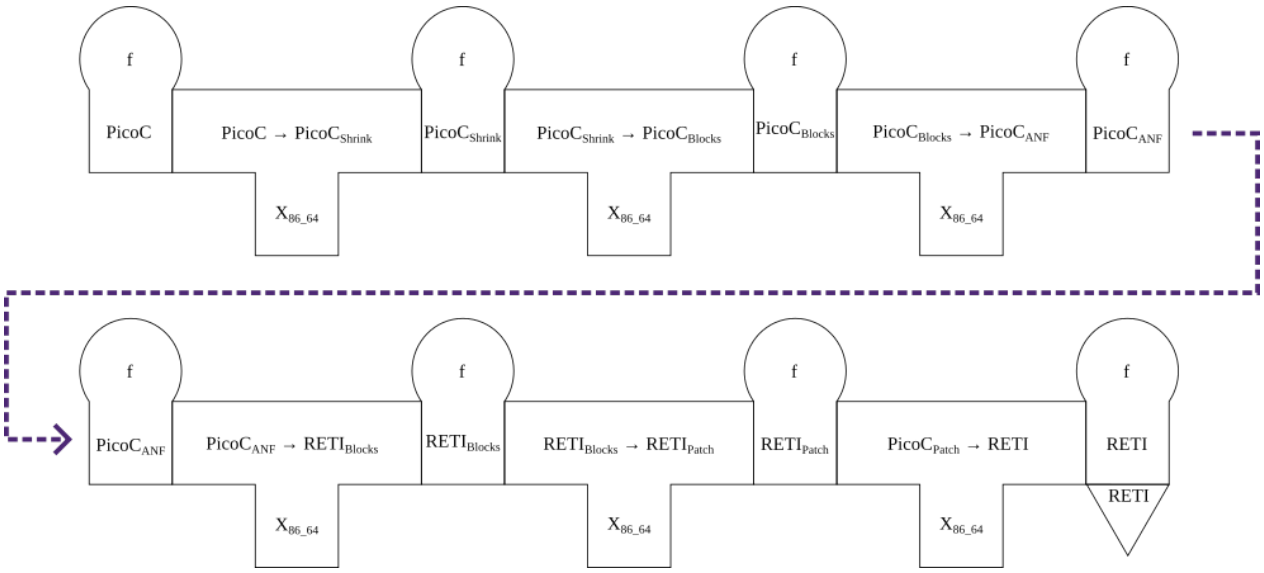


Abbildung 2.8: *Architektur mit allen Passes ausgeschrieben*

Im Unterkapitel 2.3.1 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln **??**, **??**, **??** und **??** zu **Pointern**, **Arrays**, **Structs** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 2.3.1 nicht ausreichend vertieft wurden. Viele der verwendenten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

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[18]. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[19] finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[20].

2.3.1 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der veschiedenen Aspekte[21] der

[18] Also die verschiedenen in den **Passes** generierten **Abstrakter Syntaxbaums**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[19] `https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples`.
[20] Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstrakter Syntaxbaums** in den verscchiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.
[21] In kurz: **Pointer**, **Arrays**, **Strcuts** 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.

2.3.1.1 PicoC-Shrink Pass

2.3.1.1.1 Aufgabe

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

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

2.3.1.1.2 Abstrakte Syntax

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

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

Grammatik 2.3.1: *Abstrakte Syntax der Sprache* L_{PiocC_Shrink}

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

2.3.1.1.3 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 2.6 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 2.6 ist in der Funktion `faculty` ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen **Arguments** berechnet. Der Algorithmus basiert auf einem **Beispielprogramm**

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

Dieser **rekursive** Algoirthmus ist allerdings **kein** gutes **Anschaungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von `if-`, `if-else-`, `while-` und `do-while`-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 2.6 umgeschrieben, um `if-` und `while`-Statemtens zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter Link[22], unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

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

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

Code 2.6: *PicoC Code für Codebespiel*

In Code 2.7 sieht man den **Abstrakter Syntaxbaum**, der in der **Syntaktischen Analyse** generiert wurde.

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

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

```
10      [
11        Assign(Alloc(Writeable(), 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 2.7: *Abstrakter Syntaxbaum für Codebespiel*

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstrakter Syntaxbaum** in Code 2.7, da das Codebeispiel keine **Dereferenzierung** enthält.

2.3.1.2 PicoC-Blocks Pass

2.3.1.2.1 Aufgabe

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

2.3.1.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache L_{PicoC_Shrink} in Tabelle 2.3.1 um die Knoten zu erweitern, die im Unterkapitel 2.3.1.2.1 erwähnt wurden. Die Knoten `If(exp, stmts)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` gibt es nicht mehr, da sie durch `Block(name, stmts_instrs-`, `GoTo(lable)-` und `IfElse(exp, stmts1, stmts2)`-Knoten ersetzt wurden. Die **Funktionsdefinition** `FunDef(⟨datatype⟩, Name(str), Alloc(Writeable(), ⟨datatype⟩, Name(str))*, ⟨block⟩*)` ist nun ein Container für **Blöcke** `Block(Name(str), ⟨stmt⟩*)` und keine Statements `stmt` mehr. Das resultiert in der **Abstrakten Syntax** der Sprache L_{PicoC_Blocks} in Tabelle 2.3.2.

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

Grammatik 2.3.2: *Abstrakte Syntax der Sprache* L_{PiocC_Blocks}

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

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

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

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

[a] D.h. auch wenn dort `exp` als **Attribut** steht, kann dort **nicht** jeder Knoten, der sich aus der **Produktion** `exp` ergibt auch wirklich eingesetzt werden.

2.3.1.2.3 Codebeispiel

In Code 2.8 sieht man den **Abstract-Syntax-Tree** des **PiocC-Blocks Passes** für das aus Unterkapitel 2.6 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion `faculty` und die `main`-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement oder bis zum ersten **Auftauchen** eines `If(exp, stmts)`-, `IfElse(exp, stmts1, stmts2)`-, `While(exp, stmts)`- oder `DoWhile(exp, stmts)`-Knoten stehen. Je nachdem, ob ein `If(exp, stmts)`-, `IfElse(exp, stmts1, stmts2)`-, `While(exp, stmts)`- oder `DoWhile(exp, stmts)`-Knoten auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      IntType 'int',
      Name 'faculty',
      [
        Alloc(Writeable(), IntType('int'), Name('n'))
      ],
      [
        Block
          Name 'faculty.6',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.5'))
          ],
        Block
          Name 'condition_check.5',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.4'))
              ],
              [
                GoTo(Name('while_after.1'))
              ]
          ],
        Block
          Name 'while_branch.4',
```

```
32          [
33            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
34            IfElse
35              Atom(Name('n'), Eq('=='), Num('1')),
36              [
37                GoTo(Name('if.3'))
38              ],
39              [
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 2.8: *PicoC-Blocks Pass für Codebespiel*

2.3.1.3 PicoC-ANF Pass

2.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** ist es den **Abstrakter Syntaxbaum** der Sprache L_{PicoC_Blocks} in die **Abstrakte Syntax** der Sprache L_{PicoC_ANF} umzuformen, welche in **A-Normalform** (Definition 1.50) und damit auch in **Monadischer Normalform** (Definition 1.46) ist. Um Wiederholung zu vermeiden wird zur Erklärung der **A-Normalform** auf Unterkapitel 1.5.2 verwiesen.

Zudem wird eine **Symboltabelle** (Definition 2.11) eingeführt. In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** zugewiesen, die dem Wert einer von zwei **Countern** `rel_global_addr` und `rel_stack_addr` entspricht. Der Counter `rel_global_addr` ist für Variablen in den **Globalen Statischen Daten** und der **Counter** `rel_stack_addr` ist für Variablen auf dem **Stackframe**. Einer der beiden **Counter** wird entsprechend der **Größe** der angelegten Variable **hochgezählt**.

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

Definition 2.11: Symboltabelle

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

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

[a] In einer **Symboltabelle** werden **Bezeichner** als **Symbole** bezeichnet.

2.3.1.3.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache L_{PicoC_Blocks} in Tabelle 2.3.2 in die **A-Normalform** zu bringen. Darunter fällt es unter anderem, dafür zu sorgen, dass **Komplexe Knoten**, wie z.B. `BinOp(exp, bin_op, exp)` nur **Atomare Knoten**, wie z.B. `Stack(Num(str))` enthalten können. Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass u.a. die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `File(Name(str), block*)`-Knoten liegen usw., was in Unterkapitel **??** genauer erklärt wird. Die **Symboltabelle** ist ebenfalls als **Abstrakter Syntaxbaum** umgesetzt, wofür in der **Abstrakten Syntax** der Sprache L_{PicoC_ANF} in Grammatik 2.3.3 neue Knoten eingeführt werden.

Das ganze resultiert in der **Abstrakten Syntax** der Sprache L_{PicoC_ANF} in Grammatik 2.3.3.

[23] Bzw. der **Bezeichner**

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

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

Grammatik 2.3.3: *Abstrakte Syntax der Sprache* L_{PiocC_ANF}

2.3.1.3.3 Codebeispiel

In Code 2.9 sieht man den **Abstract-Syntax-Tree** des **PiocC-ANF Passes** für das aus Unterkapitel 2.6 weitergeführte Beispiel, indem alls Statements und Ausdrücke in **A-Normalform** sind. Die `IfElse(exp, stmts, stmts)`-Knoten sind hier in **A-Normalform** gebracht worden, indem ihre **Komplexe Bedingung** vorgezogen wurde und das Ergebnis der **Komplexen Bedingung** einer **Location** zugewiesen ist und sie selbst das Ergebnis über den **Atomaren Ausdruck** `Stack(Num(str))` vom Stack lesen: `IfElse(Stack(Num(str)), stmts, stmts)`. **Funktionen** sind nur noch über die **Labels** von Blöcken zu erkennen, die den gleichen Bezeichner haben, wie die ursprüngliche Funktion und es lässt sich nur durch das **Nachverfolgen** der `GoTo(Name('label'))`-Knoten nachvollziehen, was ursprünglich zur Funktion gehörte.

```
File
 Name './example_faculty_it.picoc_mon',
 [
   Block
     Name 'faculty.6',
     [
       // Assign(Name('res'), Num('1'))
       Exp(Num('1'))
       Assign(Stackframe(Num('1')), Stack(Num('1')))
       // While(Num('1'), [])
       Exp(GoTo(Name('condition_check.5')))
     ],
   Block
     Name 'condition_check.5',
     [
       // IfElse(Num('1'), [], [])
       Exp(Num('1')),
       IfElse
         Stack
           Num '1',
         [
           GoTo(Name('while_branch.4'))
         ],
         [
           GoTo(Name('while_after.1'))
         ]
     ],
   Block
     Name 'while_branch.4',
     [
       // If(Atom(Name('n'), Eq('=='), Num('1')), [])
       // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
       Exp(Stackframe(Num('0')))
       Exp(Num('1'))
       Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
       IfElse
         Stack
           Num '1',
         [
           GoTo(Name('if.3'))
         ],
         [
           GoTo(Name('if_else_after.2'))
         ]
     ],
```

```
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 2.9: *PicoC-ANF Pass für Codebespiel*

2.3.1.4 RETI-Blocks Pass

2.3.1.4.1 Aufgabe

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

2.3.1.4.2 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache L_{RETI_Blocks} in Grammatik 2.3.4 ist verglichen mit der **Abstrakten Syntax** der Sprache L_{PicoC_ANF} in Grammatik 2.3.3 stark verändert, denn der Großteil der **PicoC-Knoten** wird in diesem Pass durch entsprechende **RETI-Knoten** ersetzt. Die einzigen verbleibenden **PicoC-Knoten**

sind `Exp(GoTo(str))`, `Block(Name(str), ⟨instr⟩*)` und `File(Name(str), ⟨block⟩*)`, da das gesamte Konzept mit den **Blöcken** erst im **RETI-Pass** in Unterkapitel 2.3.8 aufgelöst wird.

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

Grammatik 2.3.4: *Abstrakte Syntax der Sprache* L_{RETI_Blocks}

2.3.1.4.3 Codebeispiel

In Code 2.10 sieht man den **Abstract-Syntax-Tree** des **RETI-Blocks Passes** für das aus Unterkapitel 2.6 weitergeführte Beispiel, indem die **Statements**, die durch entsprechende **PicoC-Knoten** im **Abstrakt Syntax Tree** der Sprache L_{PicoC_ANF} in Grammatik 2.3.3 repräsentiert waren nun durch ihre entsprechennden **RETI-Knoten** ersetzt werden.

```
File
  Name './example_faculty_it.reti_blocks',
  [
    Block
      Name 'faculty.6',
      [
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN BAF ACC -3;
        ADDI SP 1;
        # // While(Num('1'), [])
        # Exp(GoTo(Name('condition_check.5')))
        Exp(GoTo(Name('condition_check.5')))
      ],
    Block
      Name 'condition_check.5',
      [
        # // IfElse(Num('1'), [], [])
        # Exp(Num('1'))
```

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

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

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

Code 2.10: *RETI-Blocks Pass für Codebespiel*

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

2.3.1.5 RETI-Patch Pass

2.3.1.5.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist das **Ausbessern** (engl. to patch) des **Abstrakter Syntaxbaums**, durch:

- das **Einfügen** eines `start.<nummer>`-Blockes, welcher ein `GoTo(Name('main'))` zur `main`-Funktion enthält, wenn in manchen Fällen die `main`-Funktion **nicht** die erste Funktion ist und daher am Anfang zur `main`-Funktion gesprungen werden muss.
- das **Entfernen** von `GoTo()`'s, deren Sprung nur **eine** Adresse weiterspringen würde.
- das **Voranstellen** von **RETI-Knoten**, die vor jeder **Division** `Instr(Div(), args)` prüfen, ob, nicht durch `0` geteilt wird.[25]
- das Überprüfen darauf, ob bestimmte **Immediates** `Im(str)` in Befehlen, wie z.B. `Jump(rel, Im(str))`, `Instr(Loadin(), [reg, reg, Im(str)])`, `Instr(Loadi(), [reg, Im(str)])` usw. **kleiner** -2^{21} oder **größer** $2^{21} - 1$ sind. Im Fall dessen, dass es so ist, muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **bitweise Oder** berechnet werden. Im Fall, dessen, dass der **Immediate** allerdings **kleiner** $-(2^{31})$ oder **größer** $2^{31} - 1$ ist, wird eine Fehlermeldung `TooLargeLiteral` ausgegeben.

2.3.1.5.2 Abstrakte Syntax

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

Die **Abstrakte Syntax** der Sprache L_{RETI_Patch} in Grammatik 2.3.5 ist im Vergleich zur **Abstrakten Syntax** der Sprache L_{RETI_Blocks} in Grammatik 2.3.4 kaum verändert. Es muss nur ein Knoten `Exit()` hinzugefügt werden, der im Falle einer **Division durch** 0 die Ausführung des Programs beendet.

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

Grammatik 2.3.5: *Abstrakte Syntax der Sprache* L_{RETI_Patch}

2.3.1.5.3 Codebeispiel

In Code 2.11 sieht man den **Abstract-Syntax-Tree** des **PiocC-Patch Passes** für das aus Unterkapitel 2.6 weitergeführte Beispiel. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>`-Block[26] eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-**Statements** entfernt[27], die nur einen Sprung um **eine** Position entsprochen hätten.

```
File
  Name './example_faculty_it.reti_patch',
  [
    Block
      Name 'start.7',
      [
        # // Exp(GoTo(Name('main.0')))
        Exp(GoTo(Name('main.0')))
      ],
    Block
      Name 'faculty.6',
      [
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN BAF ACC -3;
```

[26] Dieser **Block** wurde im Code 2.8 markiert.
[27] Diese **entfernten** `GoTo(Name(str))`'s' wurden ebenfalls im Code 2.8 markiert.

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

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

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

Code 2.11: *RETI-Patch Pass für Codebespiel*

2.3.1.6 RETI Pass

2.3.1.6.1 Aufgabe

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

2.3.1.6.2 Konkrette und Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache L_{RETI} in Grammatik 2.3.8 hat im Vergleich zur **Abstrakten Syntax** der Sprache L_{RETI_Patch} in Grammatik 2.3.5 nur noch ausschließlich **RETI-Knoten**. Alle **RETI-Knoten** stehen nun einem `Program(Name(str), instr)`-Knoten.

Ausgegeben wird der finale **Maschinencode** allerdings in **Konkretter Syntax**, die sich aus den Grammatiken 2.3.6 und 2.3.7 für jeweils die **Lexikalische** und **Syntaktische Analyse** zusammensetzt. Der Grund, warum die **Konkrette Syntax** der Sprache L_{RETI} auch nochmal in einen Teil für die **Lexikalische** und **Syntaktische Analyse** unterteilt ist, hat den Grund, dass für die Bachelorarbeit zum **Testen** des **PicoC-Compilers** ein **RETI-Interpreter** implementiert wurde, der den RETI-Code **lexen** und **parsen** muss, um ihn später **interpretieren** zu können.

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

Grammatik 2.3.6: *Konkrette Syntax der Sprache L_{RETI} für die Lexikalische Analyse in EBNF*

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

Grammatik 2.3.7: *Konkrette Syntax der Sprache L_{RETI} für die Syntaktische Analyse in EBNF*

reg	::=	*ACC*() | *IN*1() | *IN*2() | *PC*() | *SP*() | *BAF*()	*L_RETI*
		| *CS*() | *DS*()	
arg	::=	*Reg*(⟨*reg*⟩) | *Num*(*str*)	
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
		| *Always*() | *NOp*()	
op	::=	*Add*() | *Addi*() | *Sub*() | *Subi*() | *Mult*() | *Multi*()	
		| *Div*() | *Divi*() | *Mod*() | *Modi*() | *Oplus*() | *Oplusi*()	
		| *Or*() | *Ori*() | *And*() | *Andi*()	
		| *Load*() | *Loadin*() | *Loadi*() | *Store*() | *Storein*() | *Move*()	
instr	::=	*Instr*(⟨*op*⟩, ⟨*arg*⟩+) | *Jump*(⟨*rel*⟩, *Num*(*str*)) | *Int*(*Num*(*str*))	
		| *RTI*() | *Call*(*Name*('*print*'), ⟨*reg*⟩) | *Call*(*Name*('*input*'), ⟨*reg*⟩)	
		| *SingleLineComment*(*str*, *str*)	
		| *Instr*(*Loadi*(), [*Reg*(*Acc*()), *GoTo*(*Name*(*str*))]) | *Jump*(*Eq*(), *GoTo*(*Name*(*str*)))	
program	::=	*Program*(*Name*(*str*), ⟨*instr*⟩$*$)	
instr	::=	*Exp*(*GoTo*(*str*)) | *Exit*(*Num*(*str*))	*L_PicoC*
block	::=	*Block*(*Name*(*str*), ⟨*instr*⟩$*$)	
file	::=	*File*(*Name*(*str*), ⟨*block*⟩$*$)	

Grammatik 2.3.8: *Abstrakte Syntax der Sprache L_{RETI}*

2.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkretten Syntax** ausgegeben werden, wie in Code 2.12 zu sehen ist. Es gibt **keine Blöcke** mehr und die **RETI-Befehle** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** auschließlich aus **RETI-Ausdrücken** bestehen[28], die sich in den **Blöcken** befunden haben, wurden durch **RETI-Befehle** ersetzt.

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

```
# // Exp(GoTo(Name('main.0')))
JUMP 67;
# // Assign(Name('res'), Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Assign(Stackframe(Num('1')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN BAF ACC -3;
ADDI SP 1;
# // While(Num('1'), [])
# Exp(GoTo(Name('condition_check.5')))
# // not included Exp(GoTo(Name('condition_check.5')))
# // IfElse(Num('1'), [], [])
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# IfElse(Stack(Num('1')), [], [])
LOADIN SP ACC 1;
ADDI SP 1;
JUMP== 54;
# // not included Exp(GoTo(Name('while_branch.4')))
# // If(Atom(Name('n'), Eq('=='), Num('1')), [])
# // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
# Exp(Stackframe(Num('0')))
SUBI SP 1;
LOADIN BAF ACC -2;
STOREIN SP ACC 1;
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
SUB ACC IN2;
JUMP== 3;
LOADI ACC 0;
JUMP 2;
LOADI ACC 1;
STOREIN SP ACC 2;
```

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

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

```
100 # StackMalloc(Num('2'))
101 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6')))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 2.12: *RETI Pass für Codebespiel*

Literatur

Online

- *A-Normalization: Why and How (with code)*. URL: https://matt.might.net/articles/a-normalization/ (besucht am 23. 07. 2022).
- *ANSI C grammar (Lex)*. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (besucht am 29. 07. 2022).
- *ANSI C grammar (Yacc)*. URL: http://www.quut.com/c/ANSI-C-grammar-y.html (besucht am 29. 07. 2022).
- *ANTLR*. URL: https://www.antlr.org/ (besucht am 31. 07. 2022).
- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *clang: C++ Compiler*. URL: http://clang.org/ (besucht am 29. 07. 2022).
- *Clockwise/Spiral Rule*. URL: https://c-faq.com/decl/spiral.anderson.html (besucht am 29. 07. 2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10. 05. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 07. 2022).
- *Grammar Reference — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/grammar.html (besucht am 31. 07. 2022).
- *Grammar: The language of languages (BNF, EBNF, ABNF and more)*. URL: https://matt.might.net/articles/grammars-bnf-ebnf/ (besucht am 30. 07. 2022).
- *JSON parser - Tutorial — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09. 07. 2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13. 04. 2022).
- *Parsing Expressions · Crafting Interpreters*. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09. 07. 2022).
- *Transformers & Visitors — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09. 07. 2022).
- *Welcome to Lark's documentation! — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/ (besucht am 31. 07. 2022).

- *What is Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 2022).
- *What is Top-Down Parsing?* URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22. 06. 2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).

Artikel

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

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 (besucht am 09. 07. 2022).
- Scholl, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Thiemann, Prof. Dr. Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19. 07. 2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23. 07. 2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).
- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: https://github.com/lark-parser/lark (besucht am 31. 07. 2022).

- *Syntax*. In: *Wiktionary*. Page Version ID: 9196998. 7. Juni 2022. URL: https://de.wiktionary.org/w/index.php?title=Syntax&oldid=9196998 (besucht am 31. 07. 2022).