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 Motivation

1.1 RETI

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

Definition 1.1: Caller-save Register

a

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

Definition 1.2: Callee-save Register

a

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

1.2 PicoC

1.3 Aufgabenstellung

1.4 Eigenheiten der Sprache C

Definition 1.3: Deklaration

a

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 1.4: Definition

a

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 1.5: Allokation

a

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

Definition 1.6: Initialisierung

a

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

Definition 1.7: Scope

a

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

Definition 1.8: Call by value

a

[a] Bast, „Programmieren in C“.

Definition 1.9: Call by reference

a

[a] Bast, „Programmieren in C“.

1.5 Richtlinien

2 Einführung

2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** L_{PicoC} in die **RETI-Sprache** L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[1] und von **Tests** die **Beziehungen** in 2.2.1 zu belegen (siehe Subkapitel 4.2).

Definition 2.1: Interpreter

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

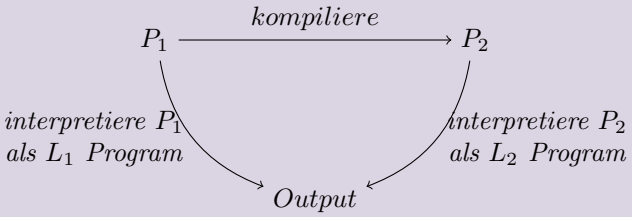
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 2.41) und führt je nach Komposition der* ***Nodes*** *des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

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

Definition 2.2: Compiler

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

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



(2.2.1)

[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 4.2) 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 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

Definition 2.3: 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 2.8) haben.

[b] C. Scholl, „Betriebssysteme“.

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

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

[a] Instructions der Assemblersprache, die mehreren Maschienenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

[b] Z.B. erlaubt die Assemblersprache des **GCC** für die X_{86_64}**-Architektur** für manche Operanden die Syntax `n(%r)`, die einen **Speicherzugriff** mit **Offset** n zur Adresse, die im **Register** `%r` steht durchführt, wobei z.B. die Klammern `()` usw. nur „syntaktischer Zucker“sind und natürlich nicht mitcodiert werden.

[c] Z.B. sind im X_{86_64} Assembler die Instructions in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels `jmp <label>` gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.

[d] P. Scholl, „Einführung in Embedded Systems“.

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

ist.

Definition 2.5: Assembler

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschienencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschienensprache** geschrieben ist.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 2.6: Objectcode

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

[a] P. Scholl, „Einführung in Embedded Systems“.

Definition 2.7: Linker

*Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschienencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt**, sodass unter anderem kein vermeidbarer **doppelter** Code darin vorkommt.*[a]

[a] P. Scholl, „Einführung in Embedded Systems“.

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 2.8) 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 2.8: 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]

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

[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.

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

*Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten*[a][b]

[a] Die Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.
[b] Thiemann, „Compilerbau“.

Definition 2.10: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L_w geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **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]

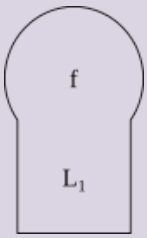
2.1.1 T-Diagramme

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

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

Definition 2.11: T-Diagram Programm

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



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

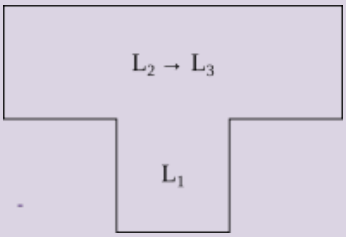
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

[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 2.12: 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.*

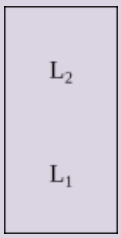
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 2.2.1.*[b]



[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 2.13: T-Diagram Interpreter

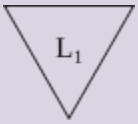
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



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

Definition 2.14: 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 2.1 und 2.2 erkennen kann zusammenfassen.

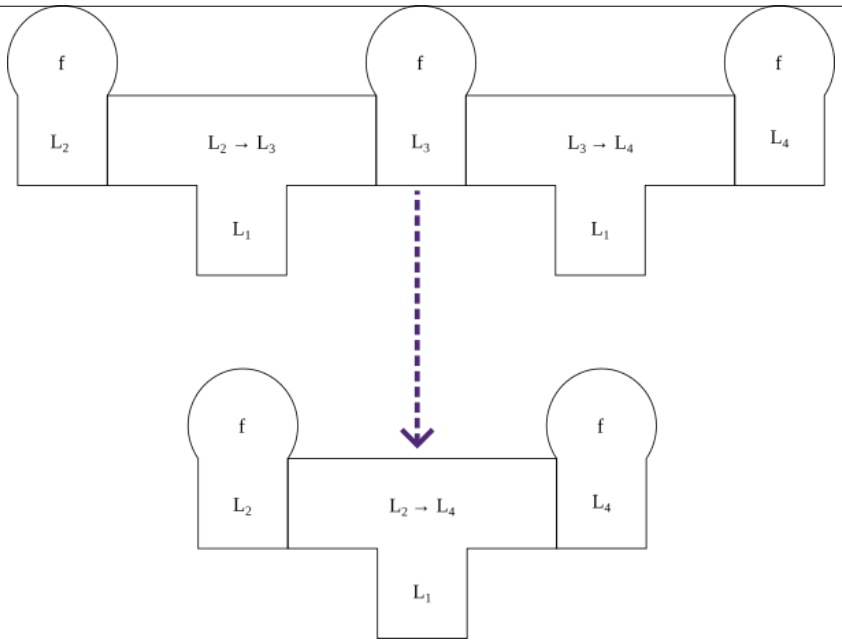


Abbildung 2.1: Horinzontale Übersetzungszwischenschritte zusammenfassen

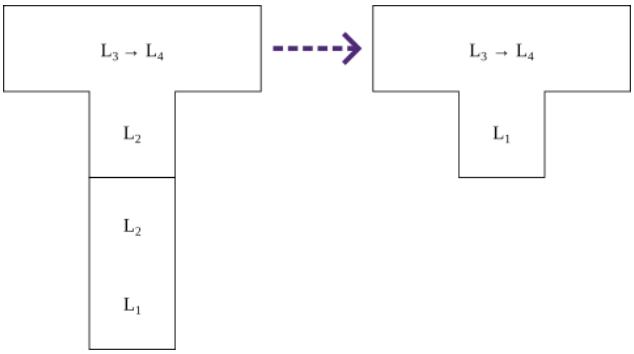


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

2.2 Formale Sprachen

Definition 2.15: Sprache

a

[a] Nebel, „Theoretische Informatik“.

Definition 2.16: Chromsky Hierarchie

a

[a] Nebel, „Theoretische Informatik“.

Definition 2.17: Grammatik

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.18: Reguläre Sprachen

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.19: Kontextfreie Sprachen

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.20: Ableitung

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.21: Links- und Rechtsableitung

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.22: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

[a] *Parsing Expressions · Crafting Interpreters*.

2.2.1 Mehrdeutige Grammatiken

Definition 2.23: Ableitungsbaum

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.24: Mehrdeutige Grammatik

[a]

[a] Nebel, „Theoretische Informatik“.

2.2.2 Präzidenz und Assoziativität

Definition 2.25: Assoziativität

[a]

[a] *Parsing Expressions · Crafting Interpreters*.

Definition 2.26: Präzidenz

[a]

[a] *Parsing Expressions · Crafting Interpreters*.

Definition 2.27: Wortproblem

[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.28: LL(k)-Grammatik

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

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

2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb der **Pipe Architektur** bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.29) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.30) genannt.

Definition 2.29: 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 2.30: 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 2.31) 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 2.31) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.31: 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 2.32) 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 3 **Symboltabelle** genannt wird.

Definition 2.32: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. **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`[4] 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`[5], 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`.

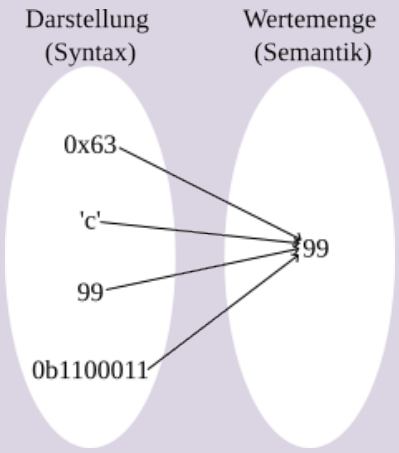
Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.33) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[6]. 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[7], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik **??** liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 2.18 erfüllt. Einzig die Produktion `CHAR ::= "'"ASCII_CHAR"'"` sieht problematisch aus, kann allerdings auch als `{CHAR ::= "'"CHAR2, CHAR2 ::= ASCII_CHAR"'"}` **regulär** ausgedrückt werden[8]. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

[4] 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.
[5] 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.
[6] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.
[7] Man nennt das auch einem **Lookahead** von 1
[8] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

Definition 2.33: 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 2.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

Inputstring

```
void main() {
  if(42) {
  }
}
```

Lexer

Tokenfolge

```
[Token('FILENAME', './example1.picoc'),
↪  Token('VOID_DT', 'void'), Token('NAME', 'main'),
↪  Token('LPAR', '('), Token('RPAR', ')'),
↪  Token('LBRACE', '{'), Token('IF', 'if'),
↪  Token('LPAR', '('), Token('NUM', '42'),
↪  Token('RPAR', ')'), Token('LBRACE', '{'),
↪  Token('RBRACE', '}'), Token('RBRACE', '}')]
```

Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

2.4 Syntaktische Analyse

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

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.34) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.36), ein **Derivation Tree** (Definition 2.35) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 2.41) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

Definition 2.34: Konkrette Syntax

__Syntax__ einer __Sprache__, die durch die __Grammatiken__ G_{Lex} und G_{Parse} zusammengenommen beschrieben wird.

Ein __Programm__ in seiner __Textrepräsentation__, wie es in einer Textdatei nach den Produktionen der __Grammatiken__ G_{Lex} und G_{Parse} abgeleitet steht, bevor man es kompiliert, ist in __Konkretter Syntax__ aufgeschrieben.[a]

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

Definition 2.35: Derivation Tree (bzw. Parse Tree)

__Compilerinterne Darstellung__ eines in __Konkretter Syntax__ geschriebenen Inputstrings als __Baumdatenstruktur__, in der __Nichtterminalsymbole__ die __Inneren Knoten__ der Baumdatenstruktur und __Terminalsymbole__ die __Blätter__ der Baumdatenstruktur bilden. Jedes zum Ableiten des Inputstrings verwendetete __Nicht-Terminalsymbol__ einer __Produktion__ der __Grammatik__ G_{Parse}, die ein Teil der __Konkrette Syntax__ ist, bildet einen eigenen __Inneren Knoten__.

Der __Derivation Tree__ wird optimalerweise immer so konstruiert bzw. die __Konkrette Syntax__ immer so definiert, dass sich möglichst einfach ein __Abstract Syntax Tree__ daraus konstruieren lässt.[a]

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

Definition 2.36: Parser

Ein __Parser__ ist ein Programm, dass aus einem Inputstring, der in __Konkretter Syntax__ geschrieben ist, eine compilerinterne Darstellung, den __Derivation Tree__ 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 2.36 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 2.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.37), denn ein Parser löst das **Wortproblem** (Definition 2.27) 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 **Derivation Tree**.

Definition 2.37: Recognizer (bzw. Erkenner)

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

[a] Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.
[b] Thiemann, „Compilerbau“.

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

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

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

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

  Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg**. Dabei wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminal** enthält.

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.22) allerdings nicht möglich, ohne die Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der **Grammatik** entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

  Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.28) 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** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkretten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.[d]
- **Chart Parser:** Es wird **Dynamische Programmierung** verwendet und partielle Zwischenergebnisse werden in einer **Tabelle** (bzw. einem **Chart**) gespeichert und können wiederverwendet werden. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist.[e]

[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 **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 2.38) und **Visitors** (Definition 2.39) generiert und ist das Endprodukt der **Syntaktischen Analyse**. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inputstring von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.40).

Definition 2.38: Transformer

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

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

Definition 2.39: Visitor

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

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

Definition 2.40: Abstrakte Syntax

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

*Jene Produktionen, die in der **Konkretten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.*[a]

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

Definition 2.41: Abstract Syntax Tree

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

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

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

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

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.4 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

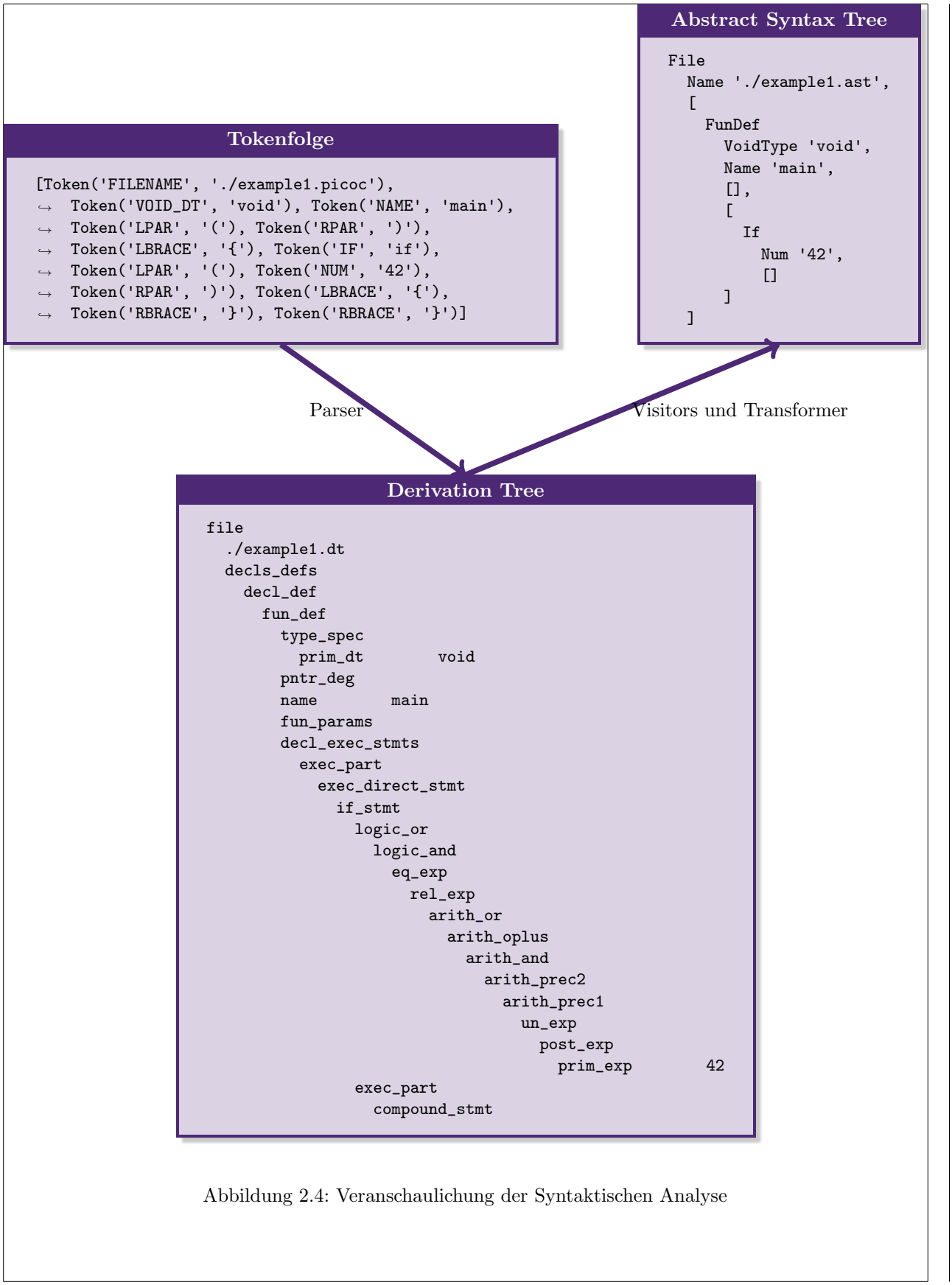


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse

2.5 Code Generierung

Definition 2.42: Pass

a

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

Definition 2.43: Monadische Normalform

a

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

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

2.6 Fehlermeldungen

Definition 2.44: Fehlermeldung

Benachrichtigung *beliebiger Form, die darüber informiert, dass:*

1. *Ein Program beim* ***Kompilieren*** *von der* ***Konkretten Syntax*** *abweicht, also der* ***Inpustring*** *sich nicht mit der Konrektten Syntax* ***ableiten*** *lässt oder auf etwas* ***zugegriffen*** *werden soll, was noch* ***nicht*** *deklariert oder definiert wurde.*

2. *Beim Ausführen eine* ***verbotene*** *Operation ausgeführt wurde.*[a]

[a] *Errors in C/C++ - GeeksforGeeks*.

2.6.1 Kategorien von Fehlermeldungen

3 Implementierung

3.1 Lexikalische Analyse

3.1.1 Konkrette Syntax für die Lexikalische Analyse

COMMENT	::=	"//" /[^\n]*/ | "/*" /(. | \n)*?/ "*/"	*L_Comment*
RETI_COMMENT.2	::=	"//""␣"?"#"/[^\n]*/	
DIG_NO_0	::=	"1" | "2" | "3" | "4" | "5"	*L_Arith*
	|	"6" | "7" | "8" | "9"	
DIG_WITH_0	::=	"0" | *DIG_NO_0*	
NUM	::=	"0" | *DIG_NO_0DIG_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*	
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*	
LOGIC_NOT	::=	"!"	

Grammar 3.1.1: Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 1

$INT\_DT.2$	::=	"*int*"	*L_Assign_Alloc*
$INT\_NAME.3$	::=	"*int*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
$CHAR\_DT.2$	::=	"*char*"	
$CHAR\_NAME.3$	::=	"*char*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
$VOID\_DT.2$	::=	"*void*"	
$VOID\_NAME.3$	::=	"*void*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
prim_dt	::=	$INT\_DT \mid CHAR\_DT \mid VOID\_DT$	

Grammar 3.1.2: Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 2

3.1.2 Basic Lexer

3.2 Syntaktische Analyse

3.2.1 Konkrette Syntax für die Syntaktische Analyse

In 3.2.1

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith +*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*L_Array +*
		| *input_exp* | *print_exp* | *prim_exp*	*L_Pntr +*
un_exp	::=	*un_opun_exp* | *post_exp*	*L_Struct + L_Fun*
input_exp	::=	"*input*""("")"	*L_Arith*
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_oprel_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*")"";"	

Grammar 3.2.1: Konkrette Syntax 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*	

Grammar 3.2.2: Konkrette Syntax für die Syntaktische Analyse in EBNF, Teil 2

3.2.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**[1]. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 3.1 aufgelistet.

Präzidenz	Operator	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 ←
13	`a,b`	Komma	Links, dann rechts →

Tabelle 3.1: Präzidenzregeln von PicoC

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

3.2.3 Derivation Tree Generierung

3.2.3.1 Early Parser

3.2.3.2 Codebeispiel

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

Code 3.1: PicoC Code für Derivation Tree Generierung

```
1  file
2    ./example_dt_simple_ast_gen_array_decl_and_alloc.dt
3    decls_defs
4      decl_def
5        struct_decl
6          name  st
7          struct_params
8            alloc
9              type_spec
10               prim_dt  int
11             pntr_decl
12               pntr_deg  *
13               array_decl
14                 pntr_decl
15                   pntr_deg  *
16                   array_decl
17                     name  attr
18                     array_dims
19                 array_dims
20                   5
21                   6
22     decl_def
23       fun_def
24         type_spec
25           prim_dt  void
26         pntr_deg
27         name  main
28         fun_params
29         decl_exec_stmts
30           decl_part
31             decl_exp_stmt
32               alloc
33                 type_spec
34                   struct_spec
35                     name  st
36                 pntr_decl
37                   pntr_deg  *
38                   array_decl
39                     pntr_decl
40                       pntr_deg  *
```

```
41                      array_decl
42                        name  var
43                        array_dims
44                    array_dims
45                      3
46                      2
```

Code 3.2: Derivation Tree nach Derivation Tree Generierung

3.2.4 Derivation Tree Vereinfachung

3.2.4.1 Visitor

3.2.4.2 Codebeispiel

Beispiel aus Subkapitel 3.2.3.2 wird fortgeführt.

```
1 file
2   ./example_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3   decls_defs
4     decl_def
5       struct_decl
6         name  st
7         struct_params
8           alloc
9             pntr_decl
10              pntr_deg  *
11              array_decl
12                array_dims
13                  5
14                  6
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                      3
38                      2
39                    pntr_decl
40                      pntr_deg  *
41                      array_decl
42                        array_dims
43                        type_spec
44                          struct_spec
45                            name  st
46                name  var
```

Code 3.3: Derivation Tree nach Derivation Tree Vereinfachung

3.2.5 Abstrakt Syntax Tree Generierung

3.2.5.1 PicoC-Knoten

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 ∧ b, a & b, a | b, a && b, a || b`.
`Eq(), NEq(), Lt(), LtE(), Gt(), GtE()`	Die **Relationen** `rel`: `a == b, a != b, a < b, a <= b, a > b, a >= b`.
`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 3.2: 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** 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-Mon 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 3.3: 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-Mon 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_goto)`	Container für ein **If Statement** `if(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`IfElse(exp, stmts_goto1, stmts_goto2)`	Container für ein **If-Else Statement** `if(<exp>) { <stmts2> } else { <stmts2> }` inklusive **Codition** `exp` und 2 **Branches** `stmts_goto1` und `stmts_goto2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Statements** oder `GoTo(Name('block.xyz'))`'s stehen können.
`While(exp, stmts_goto)`	Container für ein **While-Statement** `while(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`DoWhile(exp, stmts_goto)`	Container für ein **Do-While-Statement** `do { <stmts> } while(<exp>);` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, 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 3.4: 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**, wobei `fun_name` der Bezeichner der Funktion ist, für die ein neuer **Stackframe** erstellt werden soll und später dazu dient den **Block** dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attribute angehängt hat und `goto_after_call` ein `GoTo(Name('addr@next_instr'))` ist, welches später durch die **Adresse** der Instruction, die direkt auf die **Jump Instruction** folgt, ersetzt wird.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes**.
`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** 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 **Abstract Syntax Trees** 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 3.5: PicoC-Knoten Teil 4

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 3.1) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 3.2) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

Definition 3.1: Token-Knoten

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

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

[a] Thiemann, „Compilerbau“.

Definition 3.2: 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 Abstract Syntax Tree **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“.

3.2.5.2 RETI-Knoten

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] C. Scholl, „Betriebssysteme“

Tabelle 3.6: RETI-Knoten

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

Hier sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax** 3.2.1 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 3.7: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 3.7 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.

3.2.5.4 Abstrakte Syntax

un_op	$::=$	$Minus() \mid Not()$	*L_Arith*
bin_op	$::=$	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	$::=$	$Name(str) \mid Num(str) \mid Char(str)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), None)$	
exp_stmts	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
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*
$datatype$	$::=$	$IntType() \mid CharType() \mid VoidType()$	
lhs	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str)) \mid \langle rel\_loc\rangle$	
exp_stmts	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
$stmt$	$::=$	$Assign(\langle lhs\rangle, \langle exp\rangle)$	
	$\mid$	$Exp(\langle exp\_stmts\rangle)$	
$datatype$	$::=$	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
$deref\_loc$	$::=$	$Ref(\langle ref\_loc\rangle) \mid \langle ref\_loc\rangle$	
ref_loc	$::=$	$Name(str)$	
	$\mid$	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Attr(\langle ref\_loc\rangle, Name(str))$	
exp	$::=$	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Ref(\langle ref\_loc\rangle)$	
$datatype$	$::=$	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	$::=$	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	$::=$	$StructSpec(Name(str))$	*L_Struct*
exp	$::=$	$Attr(\langle ref\_loc\rangle, Name(str))$	
	$\mid$	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
$decl\_def$	$::=$	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
$stmt$	$::=$	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	$\mid$	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
$stmt$	$::=$	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	$\mid$	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	$::=$	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
exp_stmts	$::=$	$Call(Name(str), \langle exp\rangle*)$	
$stmt$	$::=$	$Return(\langle exp\rangle)$	
$decl\_def$	$::=$	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
$file$	$::=$	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

Grammar 3.2.3: Abstrakte Syntax für L_{PiocC}

3.2.5.5 Transformer

3.2.5.6 Codebeispiel

Beispiel welches in Subkapitel 3.2.3.2 angefangen wurde, wird hier fortgeführt.

```
File
 Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
 [
   StructDecl
     Name 'st',
     [
       Alloc
         Writeable,
         PntrDecl
           Num '1',
           ArrayDecl
             [
               Num '5',
               Num '6'
             ],
             PntrDecl
               Num '1',
               IntType 'int',
         Name 'attr'
     ],
   FunDef
     VoidType 'void',
     Name 'main',
     [],
     [
       Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
       ↪  PntrDecl(Num('1'), StructSpec(Name('st'))))), Name('var')))
     ]
 ]
```

Code 3.4: Abstract Syntax Tree aus vereinfachtem Derivarion Tree generiert

3.3 Code Generierung

3.3.1 Übersicht

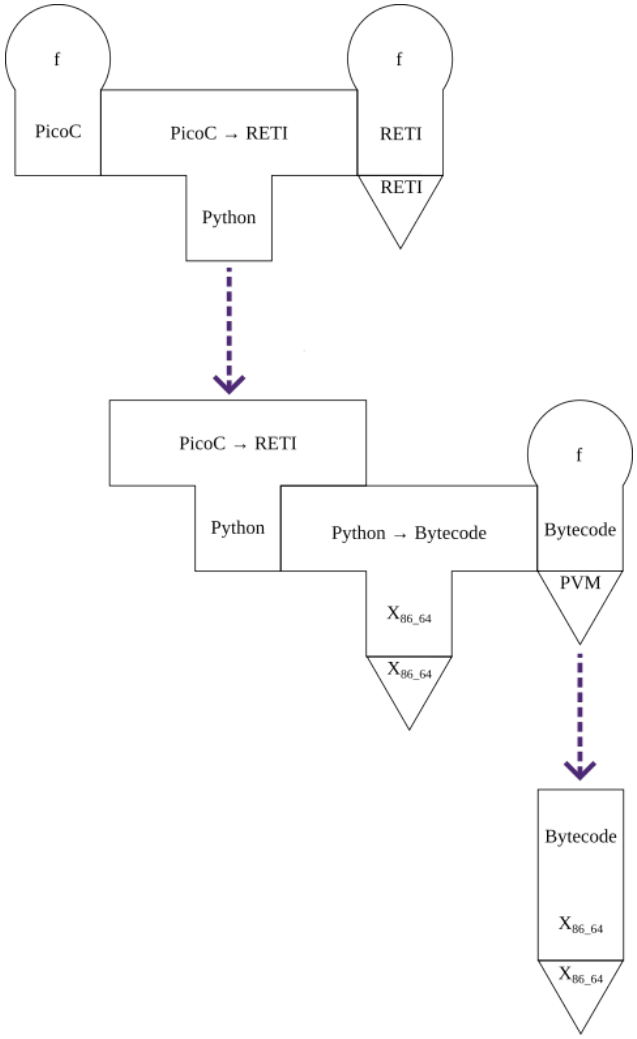


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

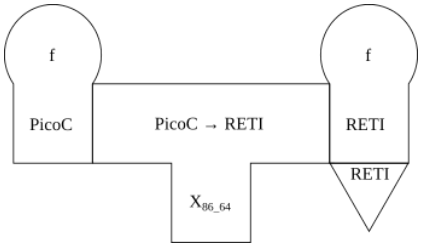


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

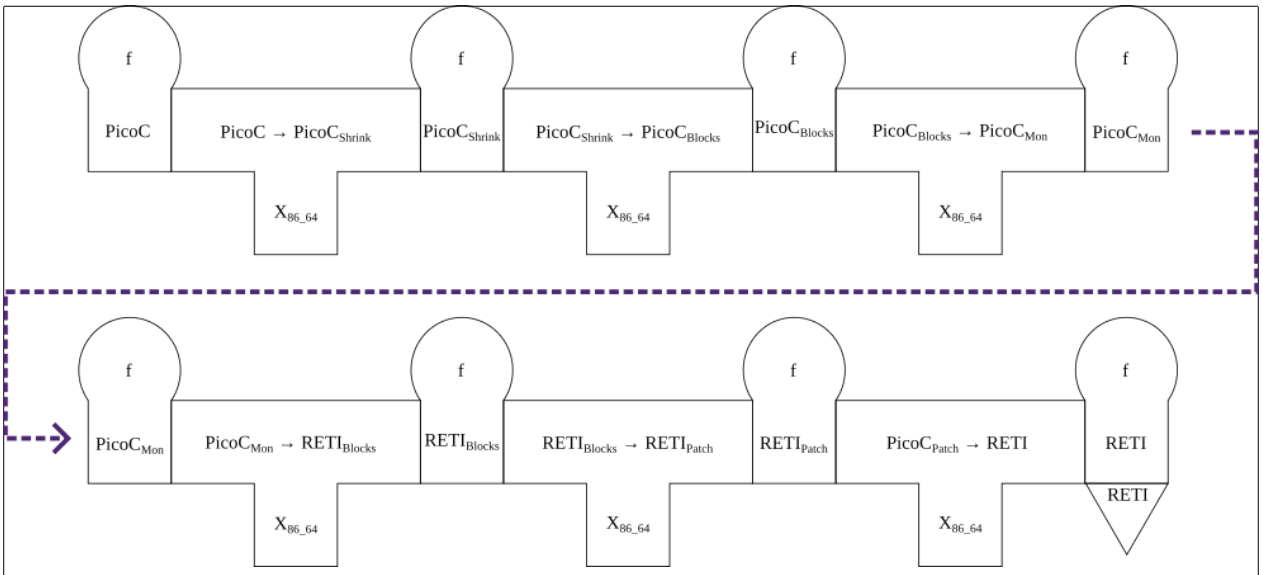


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

3.3.2 Passes

3.3.2.1 PicoC-Shrink Pass

3.3.2.1.1 Codebeispiel

```
1  // Author: Christoph Scholl, from the Operating Systems Lecture
2
3  void main() {
4    int n = 4;
5    int res = 1;
6    while (1) {
7      if (n == 1) {
8        return;
9      }
10     res = n * res;
11     n = n - 1;
12   }
13 }
```

Code 3.5: PicoC Code für Codebespiel

```
1  File
2    Name './example_faculty_it.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
```

```
9         Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
10        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
11        While
12          Num '1',
13          [
14            If
15              Atom
16                Name 'n',
17                Eq '==',
18                Num '1',
19              [
20                Return(Empty())
21              ]
22            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24          ]
25      ]
26  ]
```

Code 3.6: Abstract Syntax Tree für Codebespiel

```
1 File
2   Name './example_faculty_it.picoc_shrink',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
10        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
11        While
12          Num '1',
13          [
14            If
15              Atom
16                Name 'n',
17                Eq '==',
18                Num '1',
19              [
20                Return(Empty())
21              ]
22            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24          ]
25      ]
26  ]
```

Code 3.7: PicoC Shrink Pass für Codebespiel

3.3.2.2 PicoC-Blocks Pass

3.3.2.2.1 Abstrakte Syntax

$decl\_def$	::=	$FunDef(\langle datatype\rangle, Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle block\rangle*)$	*L_Fun*
$block$	::=	$Block(Name(str), \langle stmt\rangle*)$	*L_Blocks*
$stmt$	::=	$GoTo(Name(str)) \mid NewStackframe(Name(), GoTo(str))$	
	|	$RemoveStackframe() \mid SetScope(Name(str))$	
	|	$SingleLineComment(str, str)$	

Grammar 3.3.1: Abstrakte Syntax für L_{PicoC_Blocks}

3.3.2.2.2 Codebeispiel

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Block
          Name 'main.5',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.4'))
          ],
        Block
          Name 'condition_check.4',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.3'))
              ],
              [
                GoTo(Name('while_after.0'))
              ]
          ],
        Block
          Name 'while_branch.3',
          [
            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
            IfElse
              Atom
                Name 'n',
                Eq '==',
                Num '1',
              [
                GoTo(Name('if.2'))
              ],
              [
                GoTo(Name('if_else_after.1'))
              ]
```

```
44          ],
45        Block
46          Name 'if.2',
47          [
48            Return(Empty())
49          ],
50        Block
51          Name 'if_else_after.1',
52          [
53            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
54            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
55            GoTo(Name('condition_check.4'))
56          ],
57        Block
58          Name 'while_after.0',
59          []
60      ]
61  ]
```

Code 3.8: PicoC-Blocks Pass für Codebespiel

3.3.2.3 PicoC-Mon Pass

3.3.2.3.1 Abstrakte Syntax

ref_loc	::=	$Stack(Num(str)) \mid Global(Num(str))$	*L_Assign_Alloc*
	$\mid$	$Stackframe(Num(str))$	
$error\_data$	::=	$\langle exp \rangle \mid Pos(Num(str), Num(str))$	
exp	::=	$Stack(Num(str)) \mid Ref(\langle ref_loc \rangle, \langle datatype \rangle, \langle error_data \rangle)$	
$stmt$	::=	$Exp(\langle exp \rangle)$	
	$\mid$	$Assign(Alloc(Writeable(), StructSpec(Name(str)), Name(str)),$ $Struct(Assign(Name(str), \langle exp \rangle)+, \langle datatype \rangle))$	
	$\mid$	$Assign(Alloc(Writeable(), ArrayDecl(Num(str)+, \langle datatype \rangle)),$ $Name(str)), Array(\langle exp \rangle+, \langle datatype \rangle))$	
$symbol\_table$	::=	$SymbolTable(\langle symbol \rangle)$	*L_Symbol_Table*
$symbol$	::=	$Symbol(\langle type_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)$	
$type\_qual$	::=	$Empty()$	
$datatype$	::=	$BuiltIn() \mid SelfDefined()$	
$name$	::=	$Name(str)$	
val	::=	$Num(str) \mid Empty()$	
pos	::=	$Pos(Num(str), Num(str)) \mid Empty()$	
$size$	::=	$Num(str) \mid Empty()$	

Grammar 3.3.2: Abstrakte Syntax für L_{PicoC_Mon}

Definition 3.3: Symboltabelle

3.3.2.3.2 Codebeispiel

```
File
  Name './example_faculty_it.picoc_mon',
  [
    Block
      Name 'main.5',
      [
        // Assign(Name('n'), Num('4'))
        Exp(Num('4'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('res'), Num('1'))
        Exp(Num('1'))
        Assign(Global(Num('1')), Stack(Num('1')))
        // While(Num('1'), [])
        Exp(GoTo(Name('condition_check.4')))
      ],
    Block
      Name 'condition_check.4',
      [
        // IfElse(Num('1'), [], [])
        Exp(Num('1')),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('while_branch.3'))
          ],
          [
            GoTo(Name('while_after.0'))
          ]
      ],
    Block
      Name 'while_branch.3',
      [
        // If(Atom(Name('n'), Eq('==')), Num('1')), [])
        // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
        Exp(Global(Num('0')))
        Exp(Num('1'))
        Exp(Atom(Stack(Num('2')), Eq('==')), Stack(Num('1')))),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('if.2'))
          ],
          [
            GoTo(Name('if_else_after.1'))
          ]
      ],
    Block
      Name 'if.2',
      [
        Return(Empty())
      ],
    Block
      Name 'if_else_after.1',
      [
        // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
```

```
58        Exp(Global(Num('0')))
59        Exp(Global(Num('1')))
60        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
61        Assign(Global(Num('1')), Stack(Num('1')))
62        // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
63        Exp(Global(Num('0')))
64        Exp(Num('1'))
65        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
66        Assign(Global(Num('0')), Stack(Num('1')))
67        Exp(GoTo(Name('condition_check.4')))
68      ],
69    Block
70      Name 'while_after.0',
71      [
72        Return(Empty())
73      ]
74  ]
```

Code 3.9: PicoC-Mon Pass für Codebespiel

3.3.2.4 RETI-Blocks Pass

3.3.2.4.1 Abstrakte Syntax

program	::=	$Program(Name(str), \langle block \rangle *)$	*L_Program*
exp_stmts	::=	$GoTo(str)$	*L_Blocks*
instrs_before	::=	$Num(str)$	
num_instrs	::=	$Num(str)$	
block	::=	$Block(Name(str), \langle instr \rangle *, \langle instrs\_before \rangle, \langle num\_instrs \rangle)$	
instr	::=	$GoTo(Name(str))$	

Grammar 3.3.3: Abstrakte Syntax für L_{RETI_Blocks}

3.3.2.4.2 Codebeispiel

```
1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         # // Assign(Name('n'), Num('4'))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('res'), Num('1'))
17        # Exp(Num('1'))
```

```
18        SUBI SP 1;
19        LOADI ACC 1;
20        STOREIN SP ACC 1;
21        # Assign(Global(Num('1')), Stack(Num('1')))
22        LOADIN SP ACC 1;
23        STOREIN DS ACC 1;
24        ADDI SP 1;
25        # // While(Num('1'), [])
26        # Exp(GoTo(Name('condition_check.4')))
27        Exp(GoTo(Name('condition_check.4')))
28      ],
29    Block
30      Name 'condition_check.4',
31      [
32        # // IfElse(Num('1'), [], [])
33        # Exp(Num('1'))
34        SUBI SP 1;
35        LOADI ACC 1;
36        STOREIN SP ACC 1;
37        # IfElse(Stack(Num('1')), [], [])
38        LOADIN SP ACC 1;
39        ADDI SP 1;
40        JUMP== GoTo(Name('while_after.0'));
41        Exp(GoTo(Name('while_branch.3')))
42      ],
43    Block
44      Name 'while_branch.3',
45      [
46        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
47        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
48        # Exp(Global(Num('0')))
49        SUBI SP 1;
50        LOADIN DS ACC 0;
51        STOREIN SP ACC 1;
52        # Exp(Num('1'))
53        SUBI SP 1;
54        LOADI ACC 1;
55        STOREIN SP ACC 1;
56        # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
57        LOADIN SP ACC 2;
58        LOADIN SP IN2 1;
59        SUB ACC IN2;
60        JUMP== 3;
61        LOADI ACC 0;
62        JUMP 2;
63        LOADI ACC 1;
64        STOREIN SP ACC 2;
65        ADDI SP 1;
66        # IfElse(Stack(Num('1')), [], [])
67        LOADIN SP ACC 1;
68        ADDI SP 1;
69        JUMP== GoTo(Name('if_else_after.1'));
70        Exp(GoTo(Name('if.2')))
71      ],
72    Block
73      Name 'if.2',
74      [
```

```
75        # Return(Empty())
76        LOADIN BAF PC -1;
77      ],
78    Block
79      Name 'if_else_after.1',
80      [
81        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
82        # Exp(Global(Num('0')))
83        SUBI SP 1;
84        LOADIN DS ACC 0;
85        STOREIN SP ACC 1;
86        # Exp(Global(Num('1')))
87        SUBI SP 1;
88        LOADIN DS ACC 1;
89        STOREIN SP ACC 1;
90        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
91        LOADIN SP ACC 2;
92        LOADIN SP IN2 1;
93        MULT ACC IN2;
94        STOREIN SP ACC 2;
95        ADDI SP 1;
96        # Assign(Global(Num('1')), Stack(Num('1')))
97        LOADIN SP ACC 1;
98        STOREIN DS ACC 1;
99        ADDI SP 1;
100       # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
101       # Exp(Global(Num('0')))
102       SUBI SP 1;
103       LOADIN DS ACC 0;
104       STOREIN SP ACC 1;
105       # Exp(Num('1'))
106       SUBI SP 1;
107       LOADI ACC 1;
108       STOREIN SP ACC 1;
109       # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
110       LOADIN SP ACC 2;
111       LOADIN SP IN2 1;
112       SUB ACC IN2;
113       STOREIN SP ACC 2;
114       ADDI SP 1;
115       # Assign(Global(Num('0')), Stack(Num('1')))
116       LOADIN SP ACC 1;
117       STOREIN DS ACC 0;
118       ADDI SP 1;
119       # Exp(GoTo(Name('condition_check.4')))
120       Exp(GoTo(Name('condition_check.4')))
121     ],
122   Block
123     Name 'while_after.0',
124     [
125       # Return(Empty())
126       LOADIN BAF PC -1;
127     ]
128 ]
```

Code 3.10: RETI-Blocks Pass für Codebespiel

3.3.2.5 RETI-Patch Pass

3.3.2.5.1 Abstrakte Syntax

$stmt$	$::=$	$Exit(Num(str))$

Grammar 3.3.4: Abstrakte Syntax für L_{RETI_Patch}

3.3.2.5.2 Codebeispiel

```
File
  Name './example_faculty_it.reti_patch',
  [
    Block
      Name 'start.6',
      [
        # // Exp(GoTo(Name('main.5')))
        # // patched out Exp(GoTo(Name('main.5')))
      ],
    Block
      Name 'main.5',
      [
        # // Assign(Name('n'), Num('4'))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # // While(Num('1'), [])
        # Exp(GoTo(Name('condition_check.4')))
        # // patched out Exp(GoTo(Name('condition_check.4')))
      ],
    Block
      Name 'condition_check.4',
      [
        # // 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;
```

```
46        JUMP== GoTo(Name('while_after.0'));
47        # // patched out Exp(GoTo(Name('while_branch.3')))
48      ],
49    Block
50      Name 'while_branch.3',
51      [
52        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
53        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
54        # Exp(Global(Num('0')))
55        SUBI SP 1;
56        LOADIN DS ACC 0;
57        STOREIN SP ACC 1;
58        # Exp(Num('1'))
59        SUBI SP 1;
60        LOADI ACC 1;
61        STOREIN SP ACC 1;
62        # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
63        LOADIN SP ACC 2;
64        LOADIN SP IN2 1;
65        SUB ACC IN2;
66        JUMP== 3;
67        LOADI ACC 0;
68        JUMP 2;
69        LOADI ACC 1;
70        STOREIN SP ACC 2;
71        ADDI SP 1;
72        # IfElse(Stack(Num('1')), [], [])
73        LOADIN SP ACC 1;
74        ADDI SP 1;
75        JUMP== GoTo(Name('if_else_after.1'));
76        # // patched out Exp(GoTo(Name('if.2')))
77      ],
78    Block
79      Name 'if.2',
80      [
81        # Return(Empty())
82        LOADIN BAF PC -1;
83      ],
84    Block
85      Name 'if_else_after.1',
86      [
87        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
88        # Exp(Global(Num('0')))
89        SUBI SP 1;
90        LOADIN DS ACC 0;
91        STOREIN SP ACC 1;
92        # Exp(Global(Num('1')))
93        SUBI SP 1;
94        LOADIN DS ACC 1;
95        STOREIN SP ACC 1;
96        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
97        LOADIN SP ACC 2;
98        LOADIN SP IN2 1;
99        MULT ACC IN2;
100       STOREIN SP ACC 2;
101       ADDI SP 1;
102       # Assign(Global(Num('1')), Stack(Num('1')))
```

```
103         LOADIN SP ACC 1;
104         STOREIN DS ACC 1;
105         ADDI SP 1;
106         # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
107         # Exp(Global(Num('0')))
108         SUBI SP 1;
109         LOADIN DS ACC 0;
110         STOREIN SP ACC 1;
111         # Exp(Num('1'))
112         SUBI SP 1;
113         LOADI ACC 1;
114         STOREIN SP ACC 1;
115         # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
116         LOADIN SP ACC 2;
117         LOADIN SP IN2 1;
118         SUB ACC IN2;
119         STOREIN SP ACC 2;
120         ADDI SP 1;
121         # Assign(Global(Num('0')), Stack(Num('1')))
122         LOADIN SP ACC 1;
123         STOREIN DS ACC 0;
124         ADDI SP 1;
125         # Exp(GoTo(Name('condition_check.4')))
126         Exp(GoTo(Name('condition_check.4')))
127       ],
128     Block
129       Name 'while_after.0',
130       [
131         # Return(Empty())
132         LOADIN BAF PC -1;
133       ]
134   ]
```

Code 3.11: RETI-Patch Pass für Codebespiel

3.3.2.6 RETI Pass

3.3.2.6.1 Konkrette und Abstrakte Syntax

dig_no_0	::=	"1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"	*L_Program*
dig_with_0	::=	"0" | *dig_no_0*	
num	::=	"0" | *dig_no_0dig_with_0* $*$ | "$-$"*dig_with_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*"	

Grammar 3.3.5: Konkrette Syntax für L_{RETI_Lex}

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";")∗*	

Grammar 3.3.6: Konkrette Syntax für L_{RETI_Parse}

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

Grammar 3.3.7: Abstrakte Syntax für L_{RETI}

3.3.2.6.2 Codebeispiel

```
1  # // Exp(GoTo(Name('main.5')))
2  # // patched out Exp(GoTo(Name('main.5')))
3  # // Assign(Name('n'), Num('4'))
4  # Exp(Num('4'))
5  SUBI SP 1;
6  LOADI ACC 4;
7  STOREIN SP ACC 1;
8  # Assign(Global(Num('0')), Stack(Num('1')))
9  LOADIN SP ACC 1;
10 STOREIN DS ACC 0;
11 ADDI SP 1;
12 # // Assign(Name('res'), Num('1'))
13 # Exp(Num('1'))
14 SUBI SP 1;
15 LOADI ACC 1;
```

```
16 STOREIN SP ACC 1;
17 # Assign(Global(Num('1')), Stack(Num('1')))
18 LOADIN SP ACC 1;
19 STOREIN DS ACC 1;
20 ADDI SP 1;
21 # // While(Num('1'), [])
22 # Exp(GoTo(Name('condition_check.4')))
23 # // patched out Exp(GoTo(Name('condition_check.4')))
24 # // IfElse(Num('1'), [], [])
25 # Exp(Num('1'))
26 SUBI SP 1;
27 LOADI ACC 1;
28 STOREIN SP ACC 1;
29 # IfElse(Stack(Num('1')), [], [])
30 LOADIN SP ACC 1;
31 ADDI SP 1;
32 JUMP== 49;
33 # // patched out Exp(GoTo(Name('while_branch.3')))
34 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36 # Exp(Global(Num('0')))
37 SUBI SP 1;
38 LOADIN DS ACC 0;
39 STOREIN SP ACC 1;
40 # Exp(Num('1'))
41 SUBI SP 1;
42 LOADI ACC 1;
43 STOREIN SP ACC 1;
44 # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
45 LOADIN SP ACC 2;
46 LOADIN SP IN2 1;
47 SUB ACC IN2;
48 JUMP== 3;
49 LOADI ACC 0;
50 JUMP 2;
51 LOADI ACC 1;
52 STOREIN SP ACC 2;
53 ADDI SP 1;
54 # IfElse(Stack(Num('1')), [], [])
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 JUMP== 2;
58 # // patched out Exp(GoTo(Name('if.2')))
59 # Return(Empty())
60 LOADIN BAF PC -1;
61 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
62 # Exp(Global(Num('0')))
63 SUBI SP 1;
64 LOADIN DS ACC 0;
65 STOREIN SP ACC 1;
66 # Exp(Global(Num('1')))
67 SUBI SP 1;
68 LOADIN DS ACC 1;
69 STOREIN SP ACC 1;
70 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
71 LOADIN SP ACC 2;
72 LOADIN SP IN2 1;
```

```
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
# Assign(Global(Num('1')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN DS ACC 1;
ADDI SP 1;
# // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
# Exp(Global(Num('0')))
SUBI SP 1;
LOADIN DS ACC 0;
STOREIN SP ACC 1;
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
STOREIN SP ACC 1;
# Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
LOADIN SP ACC 2;
LOADIN SP IN2 1;
SUB ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
# Assign(Global(Num('0')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
# Exp(GoTo(Name('condition_check.4')))
JUMP -53;
# Return(Empty())
LOADIN BAF PC -1;
```

Code 3.12: RETI Pass für Codebespiel

3.3.3 Umsetzung von Pointern

3.3.3.1 Referenzierung

Die **Referenzierung** `&<var>` wird im Folgenden anhand des Beispiels in Code 3.13 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4 }
```

Code 3.13: PicoC Code für Pointer Referenzierung

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakt Syntax Tree** in Code 3.14 eine **Referenzierung** `&<var>`.

```
1  File
2    Name './example_pntr_ref.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10         Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
           ↪  Ref(Name('var')))
11       ]
12   ]
```

Code 3.14: Abstract Syntax Tree für Pointer Referenzierung

Im **PicoC-Mon Pass** in Code 3.15 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')), Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp))` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var')))` ist, sind noch weitere Anweisungen zwischen den Zeilen `11` und `12` nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 3.3.6.2 kümmern.

```
1  File
2    Name './example_pntr_ref.picoc_mon',
3    [
4      Block
5        Name 'main.0',
6        [
7          // Assign(Name('var'), Num('42'))
8          Exp(Num('42'))
9          Assign(Global(Num('0')), Stack(Num('1')))
10         // Assign(Name('pntr'), Ref(Name('var')))
11         Ref(Global(Num('0')))
```

```
12        Assign(Global(Num('1')), Stack(Num('1')))
13        Return(Empty())
14      ]
15  ]
```

Code 3.15: PicoC Mon Pass für Pointer Referenzierung

Im **PicoC-Blocks Pass** in Code 3.16 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_pntr_ref.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('pntr'), Ref(Name('var')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 3.16: RETI Blocks Pass für Pointer Referenzierung

3.3.3.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

Die **Dereferenzierung** `*<var>` wird im Folgenden anhand des Beispiels in Code 3.17 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4   *pntr;
5 }
```

Code 3.17: PicoC Code für Pointer Dereferenzierung

Der Knoten `Deref(Name('var'))` repräsentiert im **Abstrakt Syntax Tree** in Code 3.18 eine **Dereferenzierung** `*<var>`.

```
File
  Name './example_pntr_deref.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Deref(Name('pntr'), Num('0')))
      ]
  ]
```

Code 3.18: Abstract Syntax Tree für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 3.19 wird ein Trick angewandt, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** `*(<var> + <i>)` sich identisch zum **Operator für den Zugriff auf einen Arrayindex** `<var>[<i>]` verhält[2]. Damit sparrt man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Derefenzierung** `*(<var> + <i>)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `<var>[<i>]` übernehmen lassen.

```
File
  Name './example_pntr_deref.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Subscr(Name('pntr'), Num('0')))
      ]
  ]
```

Code 3.19: PicoC Shrink Pass für Pointer Dereferenzierung

[2] In der Sprache L_C gibt es einen Unterschied bei der Initialisierung bei z.B. `<datatype> *<var> = "string"` und `<datatype> <var>[<i>] = "string"`, der allerdings nichts mit den beiden Operatoren zu tuen hat, sondern mit der **Initialisierung**, bei der die Sprache L_C verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Arrayindex**, vor den Bezeichner schreibt: `<var>[<i>]`, obwohl es ein **Derived Datatype** ist.

3.3.4 Umsetzung von Arrays

3.3.4.1 Initialisierung von Arrays

Die **Initialisierung** eines **Arrays** (`<datatype> <var>[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 3.20 erklärt.

```
1 void main() {
2   int ar[2][1] = {{3+1}, {4}};
3 }
4
5 void fun() {
6   int ar[2][2] = {{3, 4}, {5, 6}};
7 }
```

Code 3.20: PicoC Code für Array Initialisierung

Die **Initialisierung** eines **Arrays** `<datatype> <var>[2][1] = {{3+1}, {4}}` wird im **Abstrakt Syntax Tree** in Code 3.21 mithilfe der Komposition `Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')), Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])]))` dargestellt.

```
1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
          ↪  Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('4')])]))
10      ],
11    FunDef
12      VoidType 'void',
13      Name 'fun',
14      [],
15      [
16        Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
          ↪  Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
17      ]
18  ]
```

Code 3.21: Abstract Syntax Tree für Array Initialisierung

Bei der **Initialisierung** eines **Arrays** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[3]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 3.22 dargestellt ist.

[3] Das Widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.

Bei Variablen auf dem **Stackframe** wird ein Array **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Arrays genommen. Dies macht den **Zugriff auf ein Arrayelement** in Subkapitel 3.3.4.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf ein Arrayelement** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die Entgegengesetzt Richtung der **Globalen Statischen Daten** wächst[4].

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('1')], IntType('int'))
        name:                   Name('ar@main')
        value or address:       Num('0')
        position:               Pos(Num('2'), Num('6'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun'), [])
        name:                   Name('fun')
        value or address:       Empty()
        position:               Pos(Num('5'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('2')], IntType('int'))
        name:                   Name('ar@fun')
        value or address:       Num('3')
        position:               Pos(Num('6'), Num('6'))
        size:                   Num('4')
      }
  ]
```

Code 3.22: Symboltabelle für Array Initialisierung

Im **PiocC-Mon Pass** in Code 3.23 werden zuerst die **Ausdrücke** im **Array-Initializer** `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben.

[4] Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

Im finalen Schritt muss zwischen **Globalen Statischen Daten** bei der `main`-Funktion und **Stackframe** bei der Funktion `fun` unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))`, versetzt in der selben Reihenfolge zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.

In die Knoten `Global('0')` und `Stackframe('3')` wurde hierbei die **Startadresse** des jeweiligen Arrays geschrieben, sodass man nach dem **PicoC-Mon Pass** nie mehr Variablen in der **Symboltabelle** nachsehen muss und gleich weiß, ob sie bei den **Globalen Statischen Daten** oder auf dem **Stackframe** liegen.

```
File
  Name './example_array_init.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
        Exp(Num('3'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Exp(Num('4'))
        Assign(Global(Num('0')), Stack(Num('2')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        Exp(Num('3'))
        Exp(Num('4'))
        Exp(Num('5'))
        Exp(Num('6'))
        Assign(Stackframe(Num('3')), Stack(Num('4')))
        Return(Empty())
      ]
  ]
```

Code 3.23: PicoC Mon Pass für Array Initialisierung

Im **PicoC-Blocks Pass** in Code 3.24 werden die **PicoC-Knoten** für die Ausdrücke `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
```

```
8         # Exp(Num('3'))
9         SUBI SP 1;
10        LOADI ACC 3;
11        STOREIN SP ACC 1;
12        # Exp(Num('1'))
13        SUBI SP 1;
14        LOADI ACC 1;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
19        ADD ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Exp(Num('4'))
23        SUBI SP 1;
24        LOADI ACC 4;
25        STOREIN SP ACC 1;
26        # Assign(Global(Num('0')), Stack(Num('2')))
27        LOADIN SP ACC 1;
28        STOREIN DS ACC 1;
29        LOADIN SP ACC 2;
30        STOREIN DS ACC 0;
31        ADDI SP 2;
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ],
35    Block
36      Name 'fun.0',
37      [
38        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
          ↪  Num('6')])]))
39        # Exp(Num('3'))
40        SUBI SP 1;
41        LOADI ACC 3;
42        STOREIN SP ACC 1;
43        # Exp(Num('4'))
44        SUBI SP 1;
45        LOADI ACC 4;
46        STOREIN SP ACC 1;
47        # Exp(Num('5'))
48        SUBI SP 1;
49        LOADI ACC 5;
50        STOREIN SP ACC 1;
51        # Exp(Num('6'))
52        SUBI SP 1;
53        LOADI ACC 6;
54        STOREIN SP ACC 1;
55        # Assign(Stackframe(Num('3')), Stack(Num('4')))
56        LOADIN SP ACC 1;
57        STOREIN BAF ACC -2;
58        LOADIN SP ACC 2;
59        STOREIN BAF ACC -3;
60        LOADIN SP ACC 3;
61        STOREIN BAF ACC -4;
62        LOADIN SP ACC 4;
63        STOREIN BAF ACC -5;
```

```
64          ADDI SP 4;
65          # Return(Empty())
66          LOADIN BAF PC -1;
67        ]
68    ]
```

Code 3.24: RETI Blocks Pass für Array Initialisierung

3.3.4.2 Zugriff auf ein Arrayelement

Der **Zugriff auf ein Arrayelement** `ar[0]` wird im Folgenden anhand des Beispiels in Code 3.25 erklärt.

```
1  void main() {
2    int ar[1] = {42};
3    ar[0];
4  }
5
6  void fun() {
7    int ar[3] = {1, 2, 3};
8    ar[1+1];
9  }
```

Code 3.25: PicoC-Code für Zugriff auf ein Arrayelement

Der **Zugriff auf ein Arrayelement** `ar[0]` wird im **Abstract Syntx Tree** in Code 3.26 mithilfe des **Container-Knotens** `Subscr(Name('ar'), Num('0'))` dargestellt.

```
1  File
2    Name './example_array_access.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
           ↪  Array([Num('42')]))
10         Exp(Subscr(Name('ar'), Num('0')))
11       ],
12     FunDef
13       VoidType 'void',
14       Name 'fun',
15       [],
16       [
17         Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
           ↪  Array([Num('1'), Num('2'), Num('3')]))
18         Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19       ]
20   ]
```

Code 3.26: Abstract Syntax Tree für Zugriff auf ein Arrayelement

Im **PicoC-Mon Pass** in Code 3.27 wird beim **Container-Knoten** `Subscr('ar'), Num('0'))` zuerst im **Einleitungsteil** die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktionm `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil** wird die Adresse des **Index**, des Arrays auf das Zugegriffen werden soll berechnet. Da der **Index** auf den Zugegriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `<ar>[1 + <var>]` bestimmt sein kann, indem auch **Variablen** vorkommen können, kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle 3.7 dokumentiert.

Der Sc

```
File
  Name './example_array_access.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Assign(Name('ar'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Exp(Subscr(Name('ar'), Num('0')))
        Ref(Global(Num('0')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
        Exp(Num('1'))
        Exp(Num('2'))
        Exp(Num('3'))
        Assign(Stackframe(Num('2')), Stack(Num('3')))
        // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
        Ref(Stackframe(Num('2')))
        Exp(Num('1'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 3.27: PicoC-Mon Pass für Zugriff auf ein Arrayelement

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Exp(Subscr(Name('ar'), Num('0')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Exp(Num('0'))
        SUBI SP 1;
        LOADI ACC 0;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        LOADIN SP IN1 2;
        LOADIN SP IN2 1;
        MULTI IN2 1;
        ADD IN1 IN2;
        ADDI SP 1;
        STOREIN SP IN1 1;
        # Exp(Stack(Num('1')))
        LOADIN SP IN1 1;
        LOADIN IN1 ACC 0;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
```

```
55          STOREIN SP ACC 1;
56          # Assign(Stackframe(Num('2')), Stack(Num('3')))
57          LOADIN SP ACC 1;
58          STOREIN BAF ACC -2;
59          LOADIN SP ACC 2;
60          STOREIN BAF ACC -3;
61          LOADIN SP ACC 3;
62          STOREIN BAF ACC -4;
63          ADDI SP 3;
64          # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65          # Ref(Stackframe(Num('2')))
66          SUBI SP 1;
67          MOVE BAF IN1;
68          SUBI IN1 4;
69          STOREIN SP IN1 1;
70          # Exp(Num('1'))
71          SUBI SP 1;
72          LOADI ACC 1;
73          STOREIN SP ACC 1;
74          # Exp(Num('1'))
75          SUBI SP 1;
76          LOADI ACC 1;
77          STOREIN SP ACC 1;
78          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79          LOADIN SP ACC 2;
80          LOADIN SP IN2 1;
81          ADD ACC IN2;
82          STOREIN SP ACC 2;
83          ADDI SP 1;
84          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85          LOADIN SP IN1 2;
86          LOADIN SP IN2 1;
87          MULTI IN2 1;
88          ADD IN1 IN2;
89          ADDI SP 1;
90          STOREIN SP IN1 1;
91          # Exp(Stack(Num('1')))
92          LOADIN SP IN1 1;
93          LOADIN IN1 ACC 0;
94          STOREIN SP ACC 1;
95          # Return(Empty())
96          LOADIN BAF PC -1;
97      ]
98  ]
```

Code 3.28: RETI-Blocks Pass für Zugriff auf ein Arrayelement

3.3.4.3 Zuweisung an Arrayindex

```
1 void main() {
2   int ar[2];
3   ar[2] = 42;
4 }
```

Code 3.29: PicoC Code für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        Assign(Subscr(Name('ar'), Num('2')), Num('42'))
      ]
  ]
```

Code 3.30: Abstract Syntax Tree für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Exp(Num('2'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 3.31: PicoC Mon Pass für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        # // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
```

```
17          STOREIN SP IN1 1;
18          # Exp(Num('2'))
19          SUBI SP 1;
20          LOADI ACC 2;
21          STOREIN SP ACC 1;
22          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23          LOADIN SP IN1 2;
24          LOADIN SP IN2 1;
25          MULTI IN2 1;
26          ADD IN1 IN2;
27          ADDI SP 1;
28          STOREIN SP IN1 1;
29          LOADIN SP IN1 1;
30          LOADIN SP ACC 2;
31          ADDI SP 2;
32          STOREIN IN1 ACC 0;
33          # Return(Empty())
34          LOADIN BAF PC -1;
35        ]
36    ]
```

Code 3.32: RETI Blocks Pass für Zuweisung an Arrayindex

3.3.5 Umsetzung von Structs

3.3.5.1 Deklaration von Structs

```
1 struct st1 {int *ar[3];};
2
3 struct st2 {struct st1 st;};
4
5 void main() {
6 }
```

Code 3.33: PicoC Code für Deklaration von Structs

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                ArrayDecl([Num('3')], PntrDecl(Num('1'), IntType('int')))
7          name:                    Name('ar@st1')
8          value or address:        Empty()
9          position:                Pos(Num('1'), Num('17'))
10         size:                    Num('3')
11       },
12     Symbol
13       {
14         type qualifier:          Empty()
```

```
15        datatype:               StructDecl(Name('st1'), [Alloc(Writeable(),
          ↪  ArrayDecl([Num('3')], PntrDecl(Num('1'), IntType('int'))), Name('ar'))])
16        name:                   Name('st1')
17        value or address:       [Name('ar@st1')]
18        position:               Pos(Num('1'), Num('7'))
19        size:                   Num('3')
20      },
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               StructSpec(Name('st1'))
25        name:                   Name('st@st2')
26        value or address:       Empty()
27        position:               Pos(Num('3'), Num('23'))
28        size:                   Num('3')
29      },
30    Symbol
31      {
32        type qualifier:         Empty()
33        datatype:               StructDecl(Name('st2'), [Alloc(Writeable(),
          ↪  StructSpec(Name('st1')), Name('st'))])
34        name:                   Name('st2')
35        value or address:       [Name('st@st2')]
36        position:               Pos(Num('3'), Num('7'))
37        size:                   Num('3')
38      },
39    Symbol
40      {
41        type qualifier:         Empty()
42        datatype:               FunDecl(VoidType('void'), Name('main'), [])
43        name:                   Name('main')
44        value or address:       Empty()
45        position:               Pos(Num('5'), Num('5'))
46        size:                   Empty()
47      }
48  ]
```

Code 3.34: Symboltabelle für Deklaration von Structs

3.3.5.2 Initialisierung von Structs

```c
struct st1 {int *pntr[1];};

struct st2 {struct st1 st;};

void main() {
  int var = 42;
  struct st1 st = {.st={.pntr={{&var}}}};
}
```

Code 3.35: PicoC Code für Initialisierung von Structs

```
File
  Name './example_struct_init.ast',
  [
    StructDecl
      Name 'st1',
      [
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '1'
            ],
            PntrDecl
              Num '1',
              IntType 'int',
          Name 'pntr'
      ],
    StructDecl
      Name 'st2',
      [
        Alloc
          Writeable,
          StructSpec
            Name 'st1',
          Name 'st'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')),
        ↪  Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
        ↪  Array([Array([Ref(Name('var'))])])])])))
      ]
  ]
```

Code 3.36: Abstract Syntax Tree für Initialisierung von Structs

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('1')], PntrDecl(Num('1'), IntType('int')))
        name:                   Name('pntr@st1')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('17'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
```

```
15        datatype:               StructDecl(Name('st1'), [Alloc(Writeable(),
          ↪  ArrayDecl([Num('1')], PntrDecl(Num('1'), IntType('int'))), Name('pntr'))])
16        name:                   Name('st1')
17        value or address:       [Name('pntr@st1')]
18        position:               Pos(Num('1'), Num('7'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               StructSpec(Name('st1'))
25        name:                   Name('st@st2')
26        value or address:       Empty()
27        position:               Pos(Num('3'), Num('23'))
28        size:                   Num('1')
29      },
30    Symbol
31      {
32        type qualifier:         Empty()
33        datatype:               StructDecl(Name('st2'), [Alloc(Writeable(),
          ↪  StructSpec(Name('st1')), Name('st'))])
34        name:                   Name('st2')
35        value or address:       [Name('st@st2')]
36        position:               Pos(Num('3'), Num('7'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Empty()
42        datatype:               FunDecl(VoidType('void'), Name('main'), [])
43        name:                   Name('main')
44        value or address:       Empty()
45        position:               Pos(Num('5'), Num('5'))
46        size:                   Empty()
47      },
48    Symbol
49      {
50        type qualifier:         Writeable()
51        datatype:               IntType('int')
52        name:                   Name('var@main')
53        value or address:       Num('0')
54        position:               Pos(Num('6'), Num('6'))
55        size:                   Num('1')
56      },
57    Symbol
58      {
59        type qualifier:         Writeable()
60        datatype:               StructSpec(Name('st1'))
61        name:                   Name('st@main')
62        value or address:       Num('1')
63        position:               Pos(Num('7'), Num('13'))
64        size:                   Num('1')
65      }
66  ]
```

Code 3.37: Symboltabelle für Initialisierung von Structs

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('st'), Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
        ↪  Array([Array([Ref(Name('var'))])]))]))]))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 3.38: PicoC Mon Pass für Initialisierung von Structs

```
File
  Name './example_struct_init.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('st'), Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
        ↪  Array([Array([Ref(Name('var'))])]))]))]))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 3.39: RETI Blocks Pass für Initialisierung von Structs

3.3.5.3 Zugriff auf Structattribut

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y;
6 }
```

Code 3.40: PicoC Code für Zugriff auf Structattribut

```
1 File
2   Name './example_struct_attr_access.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'x',
11        Alloc
12          Writeable,
13          IntType 'int',
14          Name 'y'
15      ],
16    FunDef
17      VoidType 'void',
18      Name 'main',
19      [],
20      [
21        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
          ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
22        Exp(Attr(Name('st'), Name('y')))
23      ]
24  ]
```

Code 3.41: Abstract Syntax Tree für Zugriff auf Structattribut

```
1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         Exp(Num('4'))
9         Exp(Num('2'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Exp(Attr(Name('st'), Name('y')))
```

```
12        Ref(Global(Num('0')))
13        Ref(Attr(Stack(Num('1')), Name('y')))
14        Exp(Stack(Num('1')))
15        Return(Empty())
16      ]
17  ]
```

Code 3.42: PicoC Mon Pass für Zugriff auf Structattribut

```
1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Exp(Attr(Name('st'), Name('y')))
23        # Ref(Global(Num('0')))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        STOREIN SP IN1 1;
28        # Ref(Attr(Stack(Num('1')), Name('y')))
29        LOADIN SP IN1 1;
30        ADDI IN1 1;
31        STOREIN SP IN1 1;
32        # Exp(Stack(Num('1')))
33        LOADIN SP IN1 1;
34        LOADIN IN1 ACC 0;
35        STOREIN SP ACC 1;
36        # Return(Empty())
37        LOADIN BAF PC -1;
38      ]
39  ]
```

Code 3.43: RETI Blocks Pass für Zugriff auf Structattribut

3.3.5.4 Zuweisung an Structattribut

```
struct pos {int x; int y;};

void main() {
  struct pos st = {.x=4, .y=2};
  st.y = 42;
}
```

Code 3.44: PicoC Code für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc
          Writeable,
          IntType 'int',
          Name 'x',
        Alloc
          Writeable,
          IntType 'int',
          Name 'y'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
        ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
        Assign(Attr(Name('st'), Name('y')), Num('42'))
      ]
  ]
```

Code 3.45: Abstract Syntax Tree für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Assign(Attr(Name('st'), Name('y')), Num('42'))
```

```
12        Exp(Num('42'))
13        Ref(Global(Num('0')))
14        Ref(Attr(Stack(Num('1')), Name('y')))
15        Assign(Stack(Num('1')), Stack(Num('2')))
16        Return(Empty())
17      ]
18  ]
```

Code 3.46: PicoC Mon Pass für Zuweisung an Structattribut

```
1 File
2   Name './example_struct_attr_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Assign(Attr(Name('st'), Name('y')), Num('42'))
23        # Exp(Num('42'))
24        SUBI SP 1;
25        LOADI ACC 42;
26        STOREIN SP ACC 1;
27        # Ref(Global(Num('0')))
28        SUBI SP 1;
29        LOADI IN1 0;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Ref(Attr(Stack(Num('1')), Name('y')))
33        LOADIN SP IN1 1;
34        ADDI IN1 1;
35        STOREIN SP IN1 1;
36        LOADIN SP IN1 1;
37        LOADIN SP ACC 2;
38        ADDI SP 2;
39        STOREIN IN1 ACC 0;
40        # Return(Empty())
41        LOADIN BAF PC -1;
42      ]
43  ]
```

Code 3.47: RETI Blocks Pass für Zuweisung an Structattribut

3.3.6 Umsetzung der Derived Datatypes im Zusammenspiel

3.3.6.1 Einleitungsteil für Globale Statische Daten und Stackframe

```c
struct ar_with_len {int len; int ar[2];};

void main() {
  struct ar_with_len st_ar[3];
  int *(*pntr2)[3];
  pntr2;
}

void fun() {
  struct ar_with_len st_ar[3];
  int (*pntr1)[3];
  pntr1;
}
```

Code 3.48: PicoC Code für den Einleitungsteil

```
File
  Name './example_derived_dts_introduction_part.ast',
  [
    StructDecl
      Name 'ar_with_len',
      [
        Alloc
          Writeable,
          IntType 'int',
          Name 'len',
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '2'
            ],
            IntType 'int',
          Name 'ar'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪ Name('st_ar')))
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
        ↪ IntType('int')))), Name('pntr2')))
        Exp(Name('pntr2'))
```

```
28      ],
29    FunDef
30      VoidType 'void',
31      Name 'fun',
32      [],
33      [
34        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
35        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('pntr1')))
36        Exp(Name('pntr1'))
37      ]
38  ]
```

Code 3.49: Abstract Syntax Tree für den Einleitungsteil

```
1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
8         // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
          ↪  IntType('int')))), Name('pntr2')))
9         // Exp(Name('pntr2'))
10        Exp(Global(Num('9')))
11        Return(Empty())
12      ],
13    Block
14      Name 'fun.0',
15      [
16        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
17        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('pntr1')))
18        // Exp(Name('pntr1'))
19        Exp(Stackframe(Num('9')))
20        Return(Empty())
21      ]
22  ]
```

Code 3.50: PicoC Mon Pass für den Einleitungsteil

```
1 File
2   Name './example_derived_dts_introduction_part.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
```

```
7         # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
8         # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
          ↪  PntrDecl(Num('1'), IntType('int')))), Name('pntr2')))
9         # // Exp(Name('pntr2'))
10        # Exp(Global(Num('9')))
11        SUBI SP 1;
12        LOADIN DS ACC 9;
13        STOREIN SP ACC 1;
14        # Return(Empty())
15        LOADIN BAF PC -1;
16      ],
17    Block
18      Name 'fun.0',
19      [
20        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
21        # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
          ↪  IntType('int'))), Name('pntr1')))
22        # // Exp(Name('pntr1'))
23        # Exp(Stackframe(Num('9')))
24        SUBI SP 1;
25        LOADIN BAF ACC -11;
26        STOREIN SP ACC 1;
27        # Return(Empty())
28        LOADIN BAF PC -1;
29      ]
30  ]
```

Code 3.51: RETI Blocks Pass für den Einleitungsteil

3.3.6.2 Mittelteil für die verschiedenen Derived Datatypes

```
1 struct st1 {int (*ar)[1];};
2
3 void main() {
4   int var[1] = {42};
5   struct st1 st_first = {.ar=&var};
6   (*st_first.ar)[0];
7 }
```

Code 3.52: PicoC Code für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc
8           Writeable,
9           PntrDecl
```

```
10            Num '1',
11            ArrayDecl
12              [
13                Num '1'
14              ],
15              IntType 'int',
16          Name 'ar'
17      ],
18    FunDef
19      VoidType 'void',
20      Name 'main',
21      [],
22      [
23        Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
          ↪  Array([Num('42')]))
24        Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st_first')),
          ↪  Struct([Assign(Name('ar'), Ref(Name('var')))]))
25        Exp(Subscr(Deref(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
26      ]
27  ]
```

Code 3.53: Abstract Syntax Tree für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Array([Num('42')]))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11        Ref(Global(Num('0')))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
14        Ref(Global(Num('1')))
15        Ref(Attr(Stack(Num('1')), Name('ar')))
16        Exp(Num('0'))
17        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        Return(Empty())
22      ]
23  ]
```

Code 3.54: PicoC Mon Pass für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.reti_blocks',
```

```
[
  Block
    Name 'main.0',
    [
      # // Assign(Name('var'), Array([Num('42')]))
      # Exp(Num('42'))
      SUBI SP 1;
      LOADI ACC 42;
      STOREIN SP ACC 1;
      # Assign(Global(Num('0')), Stack(Num('1')))
      LOADIN SP ACC 1;
      STOREIN DS ACC 0;
      ADDI SP 1;
      # // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
      # Ref(Global(Num('0')))
      SUBI SP 1;
      LOADI IN1 0;
      ADD IN1 DS;
      STOREIN SP IN1 1;
      # Assign(Global(Num('1')), Stack(Num('1')))
      LOADIN SP ACC 1;
      STOREIN DS ACC 1;
      ADDI SP 1;
      # // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
      # Ref(Global(Num('1')))
      SUBI SP 1;
      LOADI IN1 1;
      ADD IN1 DS;
      STOREIN SP IN1 1;
      # Ref(Attr(Stack(Num('1')), Name('ar')))
      LOADIN SP IN1 1;
      ADDI IN1 0;
      STOREIN SP IN1 1;
      # Exp(Num('0'))
      SUBI SP 1;
      LOADI ACC 0;
      STOREIN SP ACC 1;
      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
      LOADIN SP IN2 2;
      LOADIN IN2 IN1 0;
      LOADIN SP IN2 1;
      MULTI IN2 1;
      ADD IN1 IN2;
      ADDI SP 1;
      STOREIN SP IN1 1;
      # Exp(Num('0'))
      SUBI SP 1;
      LOADI ACC 0;
      STOREIN SP ACC 1;
      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
      LOADIN SP IN1 2;
      LOADIN SP IN2 1;
      MULTI IN2 1;
      ADD IN1 IN2;
      ADDI SP 1;
      STOREIN SP IN1 1;
      # Exp(Stack(Num('1')))
```

```
60          LOADIN SP IN1 1;
61          LOADIN IN1 ACC 0;
62          STOREIN SP ACC 1;
63          # Return(Empty())
64          LOADIN BAF PC -1;
65        ]
66    ]
```

Code 3.55: RETI Blocks Pass für den Mittelteil

3.3.6.3 Schlussteil für die verschiedenen Derived Datatypes

```
1  struct st {int attr[2];};
2
3  void main() {
4    int ar1[1][2] = {{42, 314}};
5    struct st ar2[1] = {.attr={42, 314}};
6    int var = 42;
7    int *pntr1 = &var;
8    int **pntr2 = &pntr1;
9
10   ar1[0];
11   ar2[0];
12   *pntr2;
13 }
```

Code 3.56: PicoC Code für den Schlussteil

```
1  File
2    Name './example_derived_dts_final_part.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc
8            Writeable,
9            ArrayDecl
10             [
11               Num '2'
12             ],
13             IntType 'int',
14           Name 'attr'
15       ],
16     FunDef
17       VoidType 'void',
18       Name 'main',
19       [],
20       [
21         Assign(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
           ↪  Name('ar1')), Array([Array([Num('42'), Num('314')])]))
22         Assign(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
           ↪  Name('ar2')), Struct([Assign(Name('attr'), Array([Num('42'), Num('314')]))]))
```

```
23        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
24        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),
          ↪  Ref(Name('var')))
25        Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('pntr2')),
          ↪  Ref(Name('pntr1')))
26        Exp(Subscr(Name('ar1'), Num('0')))
27        Exp(Subscr(Name('ar2'), Num('0')))
28        Exp(Deref(Name('pntr2'), Num('0')))
29      ]
30    ]
```

Code 3.57: Abstract Syntax Tree für den Schlussteil

```
1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')])]))
8         Exp(Num('42'))
9         Exp(Num('314'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
          ↪  Num('314')]))]))
12        Exp(Num('42'))
13        Exp(Num('314'))
14        Assign(Global(Num('2')), Stack(Num('2')))
15        // Assign(Name('var'), Num('42'))
16        Exp(Num('42'))
17        Assign(Global(Num('4')), Stack(Num('1')))
18        // Assign(Name('pntr1'), Ref(Name('var')))
19        Ref(Global(Num('4')))
20        Assign(Global(Num('5')), Stack(Num('1')))
21        // Assign(Name('pntr2'), Ref(Name('pntr1')))
22        Ref(Global(Num('5')))
23        Assign(Global(Num('6')), Stack(Num('1')))
24        // Exp(Subscr(Name('ar1'), Num('0')))
25        Ref(Global(Num('0')))
26        Exp(Num('0'))
27        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
28        Exp(Stack(Num('1')))
29        // Exp(Subscr(Name('ar2'), Num('0')))
30        Ref(Global(Num('2')))
31        Exp(Num('0'))
32        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
33        Exp(Stack(Num('1')))
34        // Exp(Subscr(Name('pntr2'), Num('0')))
35        Ref(Global(Num('6')))
36        Exp(Num('0'))
37        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
38        Exp(Stack(Num('1')))
39        Return(Empty())
40      ]
```

```
41  ]
```

Code 3.58: PicoC Mon Pass für den Schlussteil

```
1  File
2    Name './example_derived_dts_final_part.reti_blocks',
3    [
4      Block
5        Name 'main.0',
6        [
7          # // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')])]))
8          # Exp(Num('42'))
9          SUBI SP 1;
10         LOADI ACC 42;
11         STOREIN SP ACC 1;
12         # Exp(Num('314'))
13         SUBI SP 1;
14         LOADI ACC 314;
15         STOREIN SP ACC 1;
16         # Assign(Global(Num('0')), Stack(Num('2')))
17         LOADIN SP ACC 1;
18         STOREIN DS ACC 1;
19         LOADIN SP ACC 2;
20         STOREIN DS ACC 0;
21         ADDI SP 2;
22         # // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
           ↪  Num('314')]))]))
23         # Exp(Num('42'))
24         SUBI SP 1;
25         LOADI ACC 42;
26         STOREIN SP ACC 1;
27         # Exp(Num('314'))
28         SUBI SP 1;
29         LOADI ACC 314;
30         STOREIN SP ACC 1;
31         # Assign(Global(Num('2')), Stack(Num('2')))
32         LOADIN SP ACC 1;
33         STOREIN DS ACC 3;
34         LOADIN SP ACC 2;
35         STOREIN DS ACC 2;
36         ADDI SP 2;
37         # // Assign(Name('var'), Num('42'))
38         # Exp(Num('42'))
39         SUBI SP 1;
40         LOADI ACC 42;
41         STOREIN SP ACC 1;
42         # Assign(Global(Num('4')), Stack(Num('1')))
43         LOADIN SP ACC 1;
44         STOREIN DS ACC 4;
45         ADDI SP 1;
46         # // Assign(Name('pntr1'), Ref(Name('var')))
47         # Ref(Global(Num('4')))
48         SUBI SP 1;
49         LOADI IN1 4;
```

```
50          ADD IN1 DS;
51          STOREIN SP IN1 1;
52          # Assign(Global(Num('5')), Stack(Num('1')))
53          LOADIN SP ACC 1;
54          STOREIN DS ACC 5;
55          ADDI SP 1;
56          # // Assign(Name('pntr2'), Ref(Name('pntr1')))
57          # Ref(Global(Num('5')))
58          SUBI SP 1;
59          LOADI IN1 5;
60          ADD IN1 DS;
61          STOREIN SP IN1 1;
62          # Assign(Global(Num('6')), Stack(Num('1')))
63          LOADIN SP ACC 1;
64          STOREIN DS ACC 6;
65          ADDI SP 1;
66          # // Exp(Subscr(Name('ar1'), Num('0')))
67          # Ref(Global(Num('0')))
68          SUBI SP 1;
69          LOADI IN1 0;
70          ADD IN1 DS;
71          STOREIN SP IN1 1;
72          # Exp(Num('0'))
73          SUBI SP 1;
74          LOADI ACC 0;
75          STOREIN SP ACC 1;
76          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
77          LOADIN SP IN1 2;
78          LOADIN SP IN2 1;
79          MULTI IN2 2;
80          ADD IN1 IN2;
81          ADDI SP 1;
82          STOREIN SP IN1 1;
83          # Exp(Stack(Num('1')))
84          # // Exp(Subscr(Name('ar2'), Num('0')))
85          # Ref(Global(Num('2')))
86          SUBI SP 1;
87          LOADI IN1 2;
88          ADD IN1 DS;
89          STOREIN SP IN1 1;
90          # Exp(Num('0'))
91          SUBI SP 1;
92          LOADI ACC 0;
93          STOREIN SP ACC 1;
94          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
95          LOADIN SP IN1 2;
96          LOADIN SP IN2 1;
97          MULTI IN2 2;
98          ADD IN1 IN2;
99          ADDI SP 1;
100         STOREIN SP IN1 1;
101         # Exp(Stack(Num('1')))
102         LOADIN SP IN1 1;
103         LOADIN IN1 ACC 0;
104         STOREIN SP ACC 1;
105         # // Exp(Subscr(Name('pntr2'), Num('0')))
106         # Ref(Global(Num('6')))
```

```
107         SUBI SP 1;
108         LOADI IN1 6;
109         ADD IN1 DS;
110         STOREIN SP IN1 1;
111         # Exp(Num('0'))
112         SUBI SP 1;
113         LOADI ACC 0;
114         STOREIN SP ACC 1;
115         # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
116         LOADIN SP IN2 2;
117         LOADIN IN2 IN1 0;
118         LOADIN SP IN2 1;
119         MULTI IN2 1;
120         ADD IN1 IN2;
121         ADDI SP 1;
122         STOREIN SP IN1 1;
123         # Exp(Stack(Num('1')))
124         # Return(Empty())
125         LOADIN BAF PC -1;
126       ]
127   ]
```

Code 3.59: RETI Blocks Pass für den Schlussteil

3.3.7 Umsetzung von Funktionen

3.3.7.1 Funktionen auflösen zu RETI Code

```
1  void main() {
2    return;
3  }
4
5  void fun1() {
6  }
7
8  int fun2() {
9    return 1;
10 }
```

Code 3.60: PicoC Code für 3 Funktionen

```
1  File
2    Name './example_3_funs.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Return(Empty())
10       ],
```

```
    FunDef
      VoidType 'void',
      Name 'fun1',
      [],
      [],
    FunDef
      IntType 'int',
      Name 'fun2',
      [],
      [
        Return(Num('1'))
      ]
  ]
```

Code 3.61: Abstract Syntax Tree für 3 Funktionen

```
File
  Name './example_3_funs.picoc_blocks',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Block
          Name 'main.2',
          [
            Return(Empty())
          ]
      ],
    FunDef
      VoidType 'void',
      Name 'fun1',
      [],
      [
        Block
          Name 'fun1.1',
          []
      ],
    FunDef
      IntType 'int',
      Name 'fun2',
      [],
      [
        Block
          Name 'fun2.0',
          [
            Return(Num('1'))
          ]
      ]
  ]
```

Code 3.62: PicoC Blocks Pass für 3 Funktionen

```
File
  Name './example_3_funs.picoc_mon',
  [
    Block
      Name 'main.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 3.63: PicoC Mon Pass für 3 Funktionen

```
File
  Name './example_3_funs.reti_blocks',
  [
    Block
      Name 'main.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun1.1',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.0',
      [
        # // Return(Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ]
  ]
```

Code 3.64: RETI Blocks Pass für 3 Funktionen

3.3.7.1.1 Sprung zur Main Funktion

```
1  void fun1() {
2  }
3
4  int fun2() {
5    return 1;
6  }
7
8  void main() {
9    return;
10 }
```

Code 3.65: PicoC Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
1  File
2    Name './example_3_funs_main.picoc_mon',
3    [
4      Block
5        Name 'fun1.2',
6        [
7          Return(Empty())
8        ],
9      Block
10       Name 'fun2.1',
11       [
12         // Return(Num('1'))
13         Exp(Num('1'))
14         Return(Stack(Num('1')))
15       ],
16     Block
17       Name 'main.0',
18       [
19         Return(Empty())
20       ]
21   ]
```

Code 3.66: PicoC Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
1  File
2    Name './example_3_funs_main.reti_blocks',
3    [
4      Block
5        Name 'fun1.2',
6        [
7          # Return(Empty())
```

```
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun2.1',
12      [
13        # // Return(Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Return(Stack(Num('1')))
19        LOADIN SP ACC 1;
20        ADDI SP 1;
21        LOADIN BAF PC -1;
22      ],
23    Block
24      Name 'main.0',
25      [
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 3.67: PicoC Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
1 File
2   Name './example_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun2.1',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ],
29    Block
```

```
30      Name 'main.0',
31      [
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ]
35  ]
```

Code 3.68: PicoC Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

3.3.7.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

```
1  int fun2(int var);
2
3  void fun1() {
4  }
5
6  void main() {
7    int var = fun2(42);
8    return;
9  }
10
11 int fun2(int var) {
12   return var;
13 }
```

Code 3.69: PicoC Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Bei mehreren Funktionen werden die **Scopes** der unterschiedlichen **Funktionen** mittels eines **Suffix** `"<fun_name>@"` umgesetzt, der an den **Variablennamen** `<var>` drangehängt wird: `<var>@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**[5] Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** nach dem **Depth-First-Search** Schema über den

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
           ↪  IntType('int'), Name('var'))])
7          name:                   Name('fun2')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('4'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Empty()
15         datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
16         name:                   Name('fun1')
17         value or address:       Empty()
```

[5] D.h. von der Wurzel zu den Blättern eines Baumes

```
18        position:               Pos(Num('3'), Num('5'))
19        size:                   Empty()
20      },
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               FunDecl(VoidType('void'), Name('main'), [])
25        name:                   Name('main')
26        value or address:       Empty()
27        position:               Pos(Num('6'), Num('5'))
28        size:                   Empty()
29      },
30    Symbol
31      {
32        type qualifier:         Writeable()
33        datatype:               IntType('int')
34        name:                   Name('var@main')
35        value or address:       Num('0')
36        position:               Pos(Num('7'), Num('6'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               IntType('int')
43        name:                   Name('var@fun2')
44        value or address:       Num('0')
45        position:               Pos(Num('11'), Num('13'))
46        size:                   Num('1')
47      }
48  ]
```

Code 3.70: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

3.3.7.3 Funktionsaufruf

3.3.7.3.1 Ohne Rückgabewert

```
1  struct st {int attr1; int attr2[2];};
2
3  void stack_fun(struct st param[2][3]);
4
5  void main() {
6    struct st local_var[2][3];
7    stack_fun(local_var);
8    return;
9  }
10
11 void stack_fun(struct st param[2][3]) {
12   int local_var;
13 }
```

Code 3.71: PicoC Code für Funktionsaufruf ohne Rückgabewert

```
File
  Name './example_fun_call_no_return_value.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        // Exp(Call(Name('stack_fun'), [Name('local_var')]))
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.0')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'stack_fun.0',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
        ↪  Name('param')))
        // Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
        Return(Empty())
      ]
  ]
```

Code 3.72: PicoC Mon Pass für Funktionsaufruf ohne Rückgabewert

```
File
  Name './example_fun_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 4;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
```

```
25        # Exp(GoTo(Name('stack_fun.0')))
26        Exp(GoTo(Name('stack_fun.0')))
27        # RemoveStackframe()
28        MOVE BAF IN1;
29        LOADIN IN1 BAF 0;
30        MOVE IN1 SP;
31        # Return(Empty())
32        LOADIN BAF PC -1;
33      ],
34    Block
35      Name 'stack_fun.0',
36      [
37        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
          ↪  Name('param')))
38        # // Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
39        # Return(Empty())
40        LOADIN BAF PC -1;
41      ]
42  ]
```

Code 3.73: RETI Blocks Pass für Funktionsaufruf ohne Rückgabewert

```
1  # // Exp(GoTo(Name('main.1')))
2  # // patched out Exp(GoTo(Name('main.1')))
3  # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
   ↪  Name('local_var')))
4  # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
5  # StackMalloc(Num('2'))
6  SUBI SP 2;
7  # Ref(Global(Num('0')))
8  SUBI SP 1;
9  LOADI IN1 0;
10 ADD IN1 DS;
11 STOREIN SP IN1 1;
12 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
13 MOVE BAF ACC;
14 ADDI SP 3;
15 MOVE SP BAF;
16 SUBI SP 4;
17 STOREIN BAF ACC 0;
18 LOADI ACC 14;
19 ADD ACC CS;
20 STOREIN BAF ACC -1;
21 # Exp(GoTo(Name('stack_fun.0')))
22 JUMP 5;
23 # RemoveStackframe()
24 MOVE BAF IN1;
25 LOADIN IN1 BAF 0;
26 MOVE IN1 SP;
27 # Return(Empty())
28 LOADIN BAF PC -1;
29 # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('st'))), Name('param')))
30 # // Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
31 # Return(Empty())
```

```
32 LOADIN BAF PC -1;
```

Code 3.74: RETI Pass für Funktionsaufruf ohne Rückgabewert

3.3.7.3.2 Mit Rückgabewert

```
1 void stack_fun() {
2   return 42;
3 }
4
5 void main() {
6   int var = stack_fun();
7 }
```

Code 3.75: PicoC Code für Funktionsaufruf mit Rückgabewert

```
1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         // Return(Num('42'))
8         Exp(Num('42'))
9         Return(Stack(Num('1')))
10      ],
11    Block
12      Name 'main.0',
13      [
14        // Assign(Name('var'), Call(Name('stack_fun'), []))
15        StackMalloc(Num('2'))
16        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
17        Exp(GoTo(Name('stack_fun.1')))
18        RemoveStackframe()
19        Assign(Global(Num('0')), Stack(Num('1')))
20        Return(Empty())
21      ]
22  ]
```

Code 3.76: PicoC Mon Pass für Funktionsaufruf mit Rückgabewert

```
1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
```

```
        # // Return(Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Call(Name('stack_fun'), []))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 2;
        MOVE SP BAF;
        SUBI SP 2;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('stack_fun.1')))
        Exp(GoTo(Name('stack_fun.1')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 3.77: RETI Blocks Pass für Funktionsaufruf mit Rückgabewert

```
# // Exp(GoTo(Name('main.0')))
JUMP 7;
# // Return(Num('42'))
# Exp(Num('42'))
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
# Return(Stack(Num('1')))
LOADIN SP ACC 1;
ADDI SP 1;
LOADIN BAF PC -1;
# // Assign(Name('var'), Call(Name('stack_fun'), []))
```

```
# StackMalloc(Num('2'))
SUBI SP 2;
# NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
MOVE BAF ACC;
ADDI SP 2;
MOVE SP BAF;
SUBI SP 2;
STOREIN BAF ACC 0;
LOADI ACC 17;
ADD ACC CS;
STOREIN BAF ACC -1;
# Exp(GoTo(Name('stack_fun.1')))
JUMP -15;
# RemoveStackframe()
MOVE BAF IN1;
LOADIN IN1 BAF 0;
MOVE IN1 SP;
# Assign(Global(Num('0')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
# Return(Empty())
LOADIN BAF PC -1;
```

Code 3.78: RETI Pass für Funktionsaufruf mit Rückgabewert

3.3.7.3.3 Umsetzung von Call by Sharing für Arrays

```
void stack_fun(int (*param1)[3], int param2[2][3]) {
}

void main() {
  int local_var1[2][3];
  int local_var2[2][3];
  stack_fun(local_var1, local_var2);
}
```

Code 3.79: PicoC Code für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_sharing_array.picoc_mon',
  [
    Block
      Name 'stack_fun.1',
      [
        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('param1')))
        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
        Return(Empty())
      ],
    Block
```

```
12      Name 'main.0',
13      [
14        // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
          ↪  Name('local_var1')))
15        // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
          ↪  Name('local_var2')))
16        // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
17        StackMalloc(Num('2'))
18        Ref(Global(Num('0')))
19        Ref(Global(Num('6')))
20        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
21        Exp(GoTo(Name('stack_fun.1')))
22        RemoveStackframe()
23        Return(Empty())
24      ]
25  ]
```

Code 3.80: PicoC Mon Pass für Call by Sharing für Arrays

```
1 SymbolTable
2   [
3     Symbol
4       {
5         type qualifier:         Empty()
6         datatype:               FunDecl(VoidType('void'), Name('stack_fun'),
          ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('param1')), Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')),
          ↪  Name('param2'))])
7         name:                   Name('stack_fun')
8         value or address:       Empty()
9         position:               Pos(Num('1'), Num('5'))
10        size:                   Empty()
11      },
12    Symbol
13      {
14        type qualifier:         Writeable()
15        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16        name:                   Name('param1@stack_fun')
17        value or address:       Num('0')
18        position:               Pos(Num('1'), Num('21'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         Writeable()
24        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
25        name:                   Name('param2@stack_fun')
26        value or address:       Num('1')
27        position:               Pos(Num('1'), Num('37'))
28        size:                   Num('1')
29      },
30    Symbol
31      {
32        type qualifier:         Empty()
```

```
      datatype:               FunDecl(VoidType('void'), Name('main'), [])
      name:                   Name('main')
      value or address:       Empty()
      position:               Pos(Num('4'), Num('5'))
      size:                   Empty()
    },
  Symbol
    {
      type qualifier:         Writeable()
      datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
      name:                   Name('local_var1@main')
      value or address:       Num('0')
      position:               Pos(Num('5'), Num('6'))
      size:                   Num('6')
    },
  Symbol
    {
      type qualifier:         Writeable()
      datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
      name:                   Name('local_var2@main')
      value or address:       Num('6')
      position:               Pos(Num('6'), Num('6'))
      size:                   Num('6')
    }
]
```

Code 3.81: Symboltabelle für Call by Sharing für Arrays

```
File
 Name './example_fun_call_by_sharing_array.reti_blocks',
 [
   Block
     Name 'stack_fun.1',
     [
       # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
       ↪  IntType('int'))), Name('param1')))
       # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
       # Return(Empty())
       LOADIN BAF PC -1;
     ],
   Block
     Name 'main.0',
     [
       # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
       ↪  Name('local_var1')))
       # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
       ↪  Name('local_var2')))
       # // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
       # StackMalloc(Num('2'))
       SUBI SP 2;
       # Ref(Global(Num('0')))
       SUBI SP 1;
       LOADI IN1 0;
       ADD IN1 DS;
```

```
24        STOREIN SP IN1 1;
25        # Ref(Global(Num('6')))
26        SUBI SP 1;
27        LOADI IN1 6;
28        ADD IN1 DS;
29        STOREIN SP IN1 1;
30        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
31        MOVE BAF ACC;
32        ADDI SP 4;
33        MOVE SP BAF;
34        SUBI SP 4;
35        STOREIN BAF ACC 0;
36        LOADI ACC GoTo(Name('addr@next_instr'));
37        ADD ACC CS;
38        STOREIN BAF ACC -1;
39        # Exp(GoTo(Name('stack_fun.1')))
40        Exp(GoTo(Name('stack_fun.1')))
41        # RemoveStackframe()
42        MOVE BAF IN1;
43        LOADIN IN1 BAF 0;
44        MOVE IN1 SP;
45        # Return(Empty())
46        LOADIN BAF PC -1;
47      ]
48  ]
```

Code 3.82: RETI Block Pass für Call by Sharing für Arrays

3.3.7.3.4 Umsetzung von Call by Value für Structs

```
1 struct st {int attr1; int attr2[2];};
2
3 void stack_fun(struct st param) {
4 }
5
6 void main() {
7   struct st local_var;
8   stack_fun(local_var);
9 }
```

Code 3.83: PicoC Code für Call by Value für Structs

```
1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         // Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('param')))
8         Return(Empty())
9       ],
```

```
    Block
      Name 'main.0',
      [
        // Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('local_var')))
        // Exp(Call(Name('stack_fun'), [Name('local_var')]))
        StackMalloc(Num('2'))
        Assign(Stack(Num('3')), Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 3.84: PicoC Mon Pass für Call by Value für Structs

```
File
  Name './example_fun_call_by_value_struct.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
        # // Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('param')))
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('local_var')))
        # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Assign(Stack(Num('3')), Global(Num('0')))
        SUBI SP 3;
        LOADIN DS ACC 0;
        STOREIN SP ACC 1;
        LOADIN DS ACC 1;
        STOREIN SP ACC 2;
        LOADIN DS ACC 2;
        STOREIN SP ACC 3;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 5;
        MOVE SP BAF;
        SUBI SP 5;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('stack_fun.1')))
        Exp(GoTo(Name('stack_fun.1')))
        # RemoveStackframe()
        MOVE BAF IN1;
```

```
39          LOADIN IN1 BAF 0;
40          MOVE IN1 SP;
41          # Return(Empty())
42          LOADIN BAF PC -1;
43        ]
44    ]
```

Code 3.85: RETI Block Pass für Call by Value für Structs

3.3.8 Umsetzung kleinerer Details

3.4 Fehlermeldungen

3.4.1 Error Handler

3.4.2 Arten von Fehlermeldungen

3.4.2.1 Syntaxfehler

3.4.2.2 Laufzeitfehler

4 Ergebnisse und Ausblick

4.1 Compiler

4.1.1 Überblick über Funktionen

4.1.2 Vergleich mit GCC

4.1.3 Showmode

4.2 Qualitätssicherung

4.3 Erweiterungsideen

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ (Defintion 4.1) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache L_Python, in der der momentane Compiler C_{PicoC} für L_{PicoC} implementiert ist und die Unabhängigkeit von einer **anderen Maschiene**, die bisher immer für das Cross-Compiling notwendig war erreicht werden.

Definition 4.1: Self-compiling Compiler

*Compiler C^w_w, der in der Sprache L_w **geschrieben** ist, die er **selbst** kompiliert. Also ein Compiler, der sich **selbst** kompilieren kann.*[a]

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

Will man nun für eine Maschiene M_{RETI}, auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 4.4) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**[1] reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auf der **Maschiene** M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der mit der Maschienensprache B_{RETI} läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ nicht selbst komplett in der Maschienensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C^{Python}_{PicoC}, den man bereits in der Programmiersprache L_{Python} implementiert hat, der in diesem Fall einen **Bootstrapping Compiler**

[1] Beschreibt die Situation, wenn ein System sich selbst als **Abhängigkeit** hat, damit es überhaupt einen **Anfang** für dieses System geben kann. Dafür steht das Problem mit der **Henne** und dem **Ei** sinnbildlich, da hier die Frage ist, wie das ganze seinen Anfang genommen hat, da beides **zirkular** voneinander abhängt.

(Definition 4.3) darstellt, auf einer anderen Maschiene M_{other} dafür nutzt, damit dieser den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ für die Maschiene M_{RETI} kompiliert bzw. **bootstraped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschiene M_{other} auf die Maschiene M_{RETI} kopiert.[2]

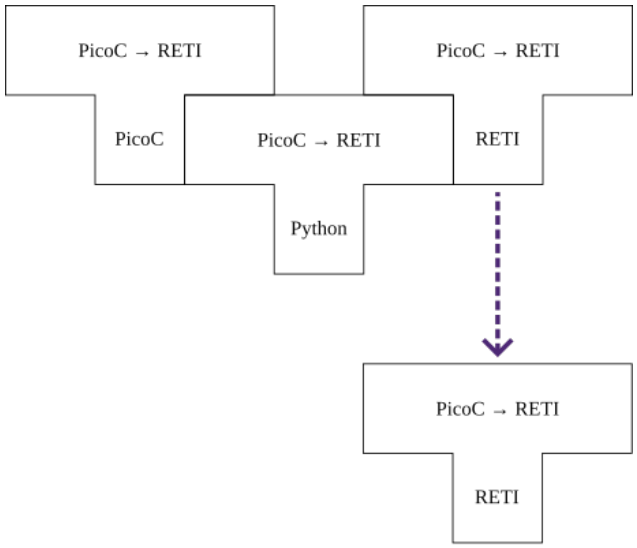


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

Einen ersten **minimalen Compiler** $C_{2\_w\_min}$ für eine Maschiene M_2 und Wunschsprache L_w kann man entweder mittels eines **externen Bootstrap Compilers** C^o_w kompilieren[a] oder man schreibt ihn direkt in der **Maschienensprache** B_2 bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** A_2.

Die letzte Option wäre allerdings nur beim allerersten Compiler C_{first} für eine allererste **abstraktere Programmiersprache** L_{first} mit Schleifen, Verzweigungen usw. notwendig gewesen. Ansonsten hätte man immer eine Kette, die beim allersten Compiler C_{first} anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

[a] In diesem Fall, dem **Cross-Compiler** C^{Python}_{PicoC}.

Definition 4.2: Minimaler Compiler

*Compiler C_{w_min}, der nur die **notwendigsten Funktionalitäten** einer Wunschsprache L_w, wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** C^w_w oder einer **ersten Version** $C^{w_i}_{w_i}$ des Self-compiling Compilers C^w_w wichtig sind.*[a b]

[a] Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.
[b] Thiemann, „Compilerbau“.

[2] Im Fall, dass auf der Maschiene M_{RETI} die Programmiersprache L_{Python} bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auch mithife des **Cross-Compilers** C^{Python}_{PicoC} als **externe Entität** und der Programmiersprache L_{Python} auf der Maschiene M_{RETI} selbst kompiliert werden.

Definition 4.3: Boostrap Compiler

*Compiler C^o_w, der es ermöglicht einen **Self-compiling Compiler** C^w_w zu **boostrapen**, indem der Self-compiling Compiler C^w_w mit dem **Bootstrap Compiler** C^o_w **kompiliert** wird[a]. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** C^w_w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.[b]*

[a] Dabei kann es sich um einen **lokal** auf der Maschiene selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.
[b] Thiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der einen **minimalen Compiler** (Definition 4.2) für eine Teilmenge der **Programmiersprache** C bzw. L_C darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw. L_C für die Maschiene M_{RETI} mittels **Bootstrapping** implementieren.[3]

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem **minimalen Compiler** $C^{PicoC}_{RETI\_PicoC}$, wie in Subdefinition 4.4.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Definition 4.4: Bootstrapping

*Wenn man einen **Self-compiling Compiler** C^w_w einer Wunschsprache L_w auf einer **Zielmaschine** M zum laufen bringt[abcd]. Dabei ist die Art von **Bootstrapping** in 4.4.1 nochmal gesondert hervorzuheben:*

4.4.1: *Wenn man die **aktuelle Version** eines **Self-compiling Compilers** $C^{w_i}_{w_i}$ der Wunschsprache L_{w_i} mithilfe von **früheren Versionen** seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ kompiliert wird und schafft es so **iterativ** immer umfangreichere Compiler zu bauen.[efg]*

[a] Z.B. mithilfe eines **Bootstrap Compilers**.
[b] Der Begriff hat seinen Ursprung in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügengeschichten des Freiherrn von Münchhausen** bekannten Redewendung „sich am eigenen Schopf aus dem Sumpf ziehen“entspricht.
[c] Hat man einmal einen solchen **Self-compiling Compiler** C^w_w auf der Maschiene M zum laufen gebracht, so kann man den Compiler auf der Maschiene M weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache L_o, in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.
[d] Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute **Probe aufs Exempel** darstellen, dass der Compiler auch wirklich funktioniert.
[e] Es ist hierbei theoretisch nicht notwendig den **letzten** Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ für das Kompilieren des **neuen** Self-compiling Compilers $C^{w_i}_{w_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C^{w_{i-3}}_{w_{i-3}}$ auch bereits alle Funktionalitäten, die beim Schreiben des **Self-compiling Compilers** C^w_w verwendet werden kompilieren kann.
[f] Der Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, welches bei Bedarf dann **Systemsoftware**, Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber. und **Anwendungssoftware**, Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt, lädt.
[g] Earley und Sturgis, „A formalism for translator interactions“.

[3] Natürlich könnte man aber auch einfach den **Cross-Compiler** C^{Python}_{PicoC} um weitere Funktionalitäten von L_C erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache L_{Python}.

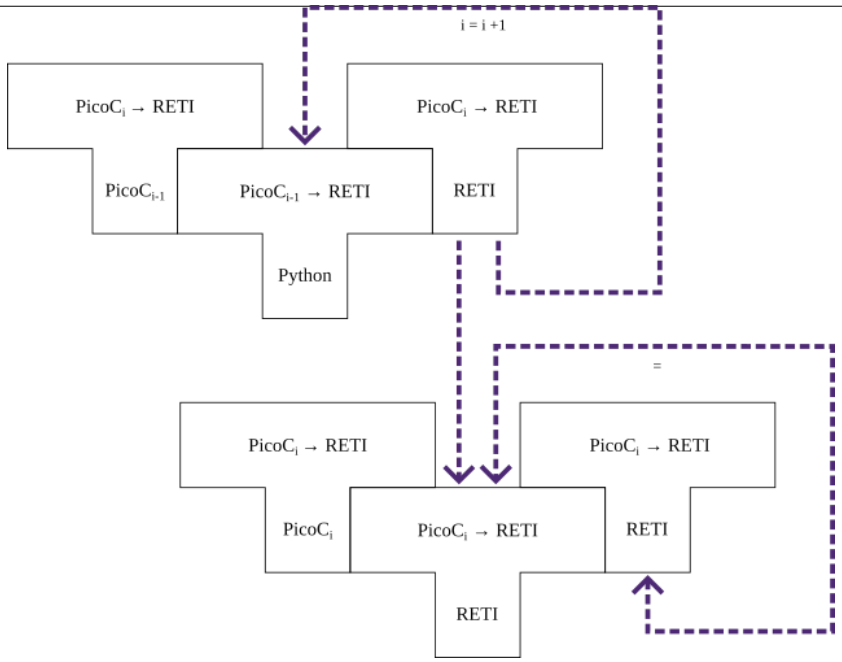


Abbildung 4.2: Iteratives Bootstrapping

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 4.4.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 4.4.1 eine frühere Version $C_{w_{i-1}}^{w_{i-1}}$ nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

A Appendix

A.1 Konkrette und Abstrakte Syntax

A.2 Bedienungsanleitungen

A.2.1 PicoC-Compiler

A.2.2 Showmode

A.2.3 Entwicklertools

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10. 05. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 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).
- *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

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09. 07. 2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).

Sonstige Quellen

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).