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.

Danksagungen

asdf

asdf

Inhaltsverzeichnis

Abbildungsverzeichnis **I**

Codeverzeichnis **II**

Tabellenverzeichnis **III**

Definitionsverzeichnis **V**

Grammatikverzeichnis **VI**

1 Motivation **1**
1.1 RETI-Architektur . . . 2
1.2 Die Sprache PicoC . . . 4
1.3 Eigenheiten der Sprachen C und PicoC . . . 5
1.4 Gesetzte Schwerpunkte . . . 11
1.5 Über diese Arbeit . . . 12
1.5.1 Still der Schrifftlichen Ausarbeitung . . . 13
1.5.2 Aufbau der Schrifftlichen Arbeit . . . 13

2 Einführung **15**
2.1 Compiler und Interpreter . . . 15
2.1.1 T-Diagramme . . . 17
2.2 Formale Sprachen . . . 19
2.2.1 Ableitungen . . . 22
2.2.2 Präzidenz und Assoziativität . . . 25
2.3 Lexikalische Analyse . . . 26
2.4 Syntaktische Analyse . . . 29
2.5 Code Generierung . . . 36
2.5.1 Monadische Normalform . . . 37
2.5.2 A-Normalform . . . 38
2.5.3 Ausgabe des Maschinencodes . . . 40
2.6 Fehlermeldungen . . . 41

Literatur **A**

Abbildungsverzeichnis

1.1 Schritte zum Ausführen eines Programmes mit dem GCC 1
1.2 Stark vereinfachte Schritte zum Ausführen eines Programmes 1
1.3 Speicherorganisation 3
1.4 README.md im Github Repository der Bachelorarbeit 12

2.1 Horinzontale Übersetzungszwischenschritte zusammenfassen 19
2.2 Vertikale Interpretierungszwischenschritte zusammenfassen 19
2.3 Veranschaulichung von Linksassoziativität und Rechtsassoziativität 26
2.4 Veranschaulichung von Präzidenz 26
2.5 Veranschaulichung der Lexikalischen Analyse 29
2.6 Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum. 34
2.7 Veranschaulichung der Syntaktischen Analyse 35
2.8 Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten 38
2.9 Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen 40

Codeverzeichnis

1.1 Beispiel für Spiralregel 6
1.2 Ausgabe von Beispiel für Spiralregel 6
1.3 Beispiel für unterschiedliche Ausführung 7
1.4 Ausgabe des Beispiels für unterschiedliche Ausführung 7
1.5 Beispiel mit Dereferenzierungsoperator 7
1.6 Ausgabe des Beispiels mit Dereferenzierungsoperator 7
1.7 Beispiel dafür, dass Struct kopiert wird 8
1.8 Ausgabe von Beispiel, dass Struct kopiert wird 8
1.9 Beispiel dafür, dass Zeiger auf Feld übergeben wird 9
1.10 Ausgabe von Beispiel dafür, dass Zeiger auf Feld übergeben wird 9
1.11 Beispiel für Deklaration und Definition 10
1.12 Ausgabe von Beispiel für Deklaration und Definition 10
1.13 Beispiel für Sichtbarkeitsbereichs 11
1.14 Ausgabe von Beispiel für Sichtbarkeitsbereichs 11

Tabellenverzeichnis

1.1 Präzidenzregeln von PicoC 3

Definitionsverzeichnis

1.1 Imperative Programmierung . . . 5
1.2 Strukturierte Programmierung . . . 5
1.3 Prozedurale Programmierung . . . 5
1.4 Call by Value . . . 8
1.5 Call by Reference . . . 8
1.6 Funktionsprototyp . . . 9
1.7 Deklaration . . . 10
1.8 Definition . . . 10
1.9 Sichtbarkeitsbereich (bzw. engl. Scope) . . . 11
2.1 Interpreter . . . 15
2.2 Compiler . . . 15
2.3 Maschienensprache . . . 16
2.4 Immediate . . . 16
2.5 Cross-Compiler . . . 17
2.6 T-Diagram Programm . . . 17
2.7 T-Diagram Übersetzer (bzw. eng. Translator) . . . 18
2.8 T-Diagram Interpreter . . . 18
2.9 T-Diagram Maschiene . . . 18
2.10 Symbol . . . 19
2.11 Alphabet . . . 20
2.12 Wort . . . 20
2.13 Formale Sprache . . . 20
2.14 Syntax . . . 20
2.15 Semantik . . . 20
2.16 Formale Grammatik . . . 20
2.17 Chromsky Hierarchie . . . 21
2.18 Reguläre Grammatik . . . 22
2.19 Kontextfreie Grammatik . . . 22
2.20 Wortproblem . . . 22
2.21 1-Schritt-Ableitungsrelation . . . 23
2.22 Ableitungsrelation . . . 23
2.23 Links- und Rechtsableitungableitung . . . 23
2.24 Linksrekursive Grammatiken . . . 23
2.25 Formaler Ableitungsbaum . . . 24
2.26 Mehrdeutige Grammatik . . . 25
2.27 Assoziativität . . . 25
2.28 Präzidenz . . . 26
2.29 Pipe-Filter Architekturpattern . . . 26
2.30 Pattern . . . 27
2.31 Lexeme . . . 27
2.32 Lexer (bzw. Scanner oder auch Tokenizer) . . . 27
2.33 Bezeichner (bzw. Identifier) . . . 28
2.34 Literal . . . 29
2.35 Konkrette Syntax . . . 30
2.36 Konkrette Grammatik . . . 30
2.37 Ableitungsbaum (bzw. Konkretter Syntaxbaum oder auch engl. Derivation Tree) . . . 30
2.38 Parser . . . 30

2.39 Recognizer (bzw. Erkenner) . . . 31
2.40 Transformer . . . 32
2.41 Visitor . . . 33
2.42 Abstrakte Syntax . . . 33
2.43 Abstrakte Grammatik . . . 33
2.44 Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST) . . . 33
2.45 Pass . . . 36
2.46 Reiner Ausdruck (bzw. engl. pure expression) . . . 37
2.47 Unreiner Ausdruck . . . 37
2.48 Monadische Normalform (bzw. engl. monadic normal form) . . . 37
2.49 Location . . . 38
2.50 Atomarer Ausdruck . . . 39
2.51 Komplexer Ausdruck . . . 39
2.52 A-Normalform (ANF) . . . 39
2.53 Fehlermeldung . . . 41

Grammatikverzeichnis

2.1 Produktionen für Ableitungsbaum in EBNF 24

1 Motivation

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiel** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprachen L_{Python}, welche als **interpretierte** Sprache bekannt ist, wird das in der Programmiersprache L_{Python} geschriebene Programm vorher zu **Bytecode** kompiliert, bevor dieser von der **Python Virtual Machine (PVM)** interpretiert wird.

Compiler, wie der **GCC**[1] oder **Clang**[2] werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert[3] zu **Maschinencode** zu kompilieren.

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll, z.B. im Fall des **GCC** über `> gcc program.c -o machine_code`[4]. Als Ergebnis erhält man im Fall des **GCC** die mit der Option `-o` selbst benannte Datei `machine_code`, welche dann zumindest unter **Unix** über `> ./machine_code` **ausgeführt** werden kann, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

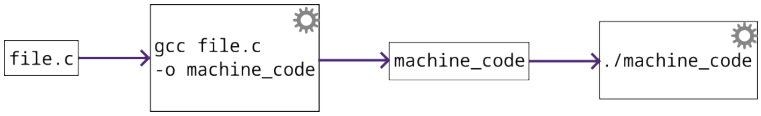


Abbildung 1.1: *Schritte zum Ausführen eines Programmes mit dem GCC*

Der ganze Kompiliervorgang kann, wie er in Abbildung 1.2 dargestellt ist zu einer Box abstrahiert werden. Der Benutzer gibt ein **Programm** in der Sprache des Compilers rein und erhält **Maschinencode**, den er dann im besten Fall in eine andere Box hineingeben kann, welche die passende **Maschine** oder den passenden **Interpreter** in Form einer **Virtuellen Maschine** repräsentiert, der bzw. die den **Maschinencode** ausführen kann.

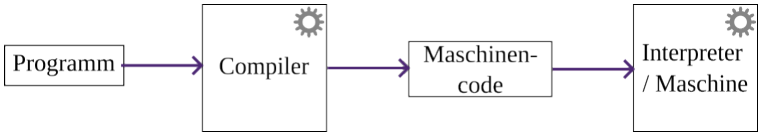


Abbildung 1.2: *Stark vereinfachte Schritte zum Ausführen eines Programmes*

[1] *GCC, the GNU Compiler Collection - GNU Project*.
[2] *clang: C++ Compiler*.
[3] Im Fall des **GCC** und **Clang** ist es die Programmiersprache L_C.
[4] Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objectcode** (Definition **??**) kompiliert wird. Das Kompilieren zu **Objectcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

Der Programmierer muss für das Vorgehen in Abbildung 1.2 **nichts** über die **Theoretischen Grundlagen des Compilerbau** wissen, noch wie der Compiler **intern** umgesetzt ist. In dieser Bachelorarbeit soll diese **Compilerbox** allerdings geöffnet werden und anhand eines eigenen im Vergleich zum **GCC** im Funktionsumfang **reduzierten Compilers** gezeigt werden, wie so ein Compiler **unter der Haube** stark vereinfacht funktionieren könnte.

Die konkrette **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** L_{PicoC}, welche eine Untermenge der Sprache L_C ist[5] in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** L_{RETI} kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache L_{RETI} zu Grunde liegt und im Unterkapitel 1.2 wird näher auf die die Sprache L_{PicoC} eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache L_{RETI} kompilieren soll.

1.1 RETI-Architektur

Die **RETI-Architektur** ist eine zu Lernzwecken für die Vorlesungen P. D. C. Scholl, „Betriebssysteme“ und P. D. C. Scholl, „Technische Informatik“ entwickelte **32-Bit** Architektur, die sich vor allem durch ihre einfache Zugänglichkeit kennzeichnet und deren **Maschinensprache** L_{RETI} als Zielsprache des **PicoC-Compilers** hergenommen wurde. In der Vorlesung P. D. C. Scholl, „Technische Informatik“ wird die **grundlegende RETI-Architektur** erklärt und in der Vorlesung P. D. C. Scholl, „Betriebssysteme“ wird diese Architektur erweitert, sodass diese mehr darauf angepasst ist, dass auch komplexere Kontrukte, wie ein **Betriebssystem**, **Interrupts**, **Prozesse**, **Funktionen** usw. auf nicht zu komplizierte Weise implementiert werden können.

Um den den **PicoC-Compiler** zu **testen** war es notwendig einen **RETI-Interpreter** zu implementieren, der genau die Variante der **RETI-Achitektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ **simuliert**.

Anmerkung

In dieser **Bachelorarbeit** wird **im Folgenden** bei der **Maschinensprache** L_{RETI} immer von der Variante, welche durch die **RETI-Architektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ umgesetzt ist ausgegangen.

Die **Register** der **RETI-Architektur** werden in Tabelle 1.1 aufgezählt und erläutert. Die **Maschinenbefehle** und **Datenpfade** der **RETI-Architektur** sind im Appendix **??** dokumentiert, da diese **nicht explizit** zum **Verständnis** der späteren Kapitel notwendig sind, aber zum **vollständigen Verständnis** notwendig sind, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** L_{RETI} ist durch Grammatik **??** und Grammatik **??** zusammengenommen beschrieben. Für genauere **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

[5] Die der **GCC** kompilieren kann.

Register Kürzel	Register Ausgeschrieben	Aufgabe
PC	Program Counter	Zeigt auf den Maschinenbefehl, der als nächstes ausgeführt werden soll.
ACC	Accumulator	Für Operanden von Operationen oder für temporäre Werte.
IN1	Indexregister 1	Hat dieselbe Aufgabe wie das ACC-Register.
IN2	Indexregister 2	Hat dieselbe Aufgabe wie das ACC-Register.
SP	Stackpointer	Zeigt immer auf die erste freie Speicherzelle am Ende des Stacks, wo als nächstes Speicher allokiert werden kann.
BAF	Begin Aktive Funktion	Zeigt auf den Beginn des Stackframes der aktuell aktiven Funktion.
CS	Codesegment	Zeigt auf den Beginn des Codesegments. Die letzten 10 Bits werden verwendet, um 22 Bit Immediates aufzufüllen. Kann dadurch dazu verwendet werden, festzulegen welcher der 3 Peripheriegeräte[a] in der Memory Map[b] angesprochen werden soll.
DS	Datensegment	Zeigt auf den Beginn des Datensegments.

[a] EPROM, UART und SRAM.
[b] Da die Memory Map zum Verständnis der Bachelorarbeit nicht wichtig ist, wird diese nicht mehr als nötig im weiteren Verlauf erläutert.

Tabelle 1.1: *Präzidenzregeln von PicoC*

Die RETI-Architektur ermöglicht bei der Ausführung von RETI-Programmen Prozesse zu nutzen. In Abbildung 1.3 ist der Aufbau eines Prozesses im Hauptspeicher der RETI-Architektur zu sehen. Das RETI-Programm nutzt dabei den Stack für temporäre Zwischenergebnisse von Berechnungen und zum Anlegen der Stackframes von Funktionen, welche die Lokalen Variablen und Parameter einer Funktion speichern. Das SP- und BAF-Register erfüllen dabei ihre zugeteilten Aufgaben für den Stack.

Der Abschnitt für die Globalen Statischen Daten ist allgemein dazu da Daten zu beherbergen, die für den Rest der Programmausführung global zugänglich sein sollen, aber auch für die Lokalen Variablen der main-Funktion. Das DS-Register markiert den Anfang des Datensegments und damit auch den Anfang der Globalen Statischen Daten und kann als relativer Orientierungspunkt beim Zugriff und Abspeichern Globaler Statischer Daten dienen. Das CS-Register wird als relativer Orientierungspunkt genutzt, an dem die Ausführung von RETI-Programmen startet und zur Bestimmung der relativen Startadresse, an welcher der RETI-Code einer bestimmten Funktion anfängt.

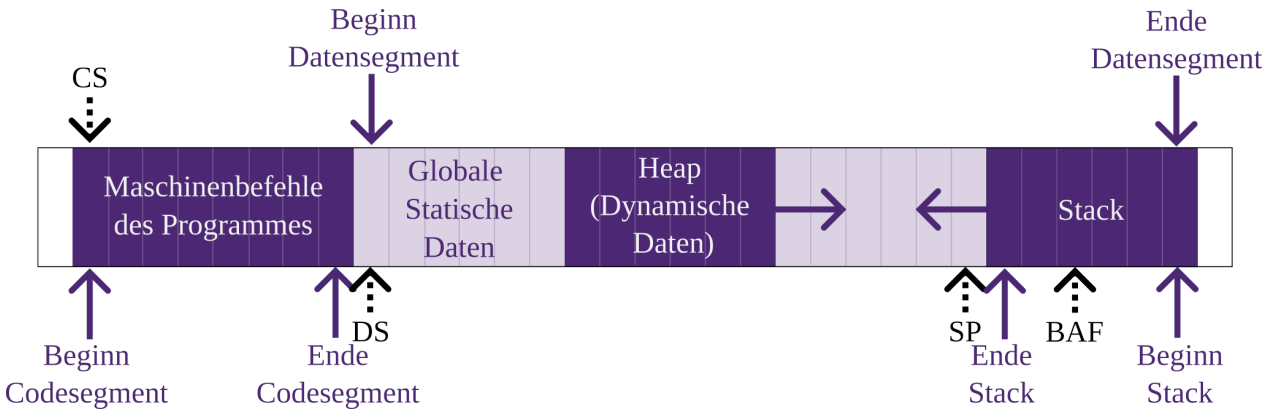


Abbildung 1.3: *Speicherorganisation*

Die RETI-Architektur nutzt 3 verschiedene **Peripheriegeräte**, **EPROM**, **UART** und **SRAM**, die über eine **Memory Map**[6] den über die **Datenpfade** der RETI-Architektur **??** ansprechbaren **Adressraum** von 2^{32} Adressen[7] unter sich aufteilen.

Die **Ausführung** eines Programmes **startet** auf die einfachste Weise, indem es von einem **Startprogramm** im **EPROM**[8] aufgerufen wird. Der **EPROM** wird beim Start einer **RETI-CPU** als **erstes** aufgerufen, da nach der **Memory Map** der erste **Adressraum** von 0 bis $2^{30} - 1$ dem **EPROM** zugeordnet ist und das `PC`-Register **initial** den Wert 0 hat, also als **erstes** das Programm ausgeführt wird, welches an **Adresse** 0 im EPROM anfängt.

Die **UART**[9] ist eine **elektronische Schaltung** mit je nach Umsetzung mehr oder weniger **Pins**, wobei es allerdings immer einen `RX`- und einen `TX`-Pin gibt, für jeweils Empfangen[10] und Versenden[11] von Daten gibt. Jeder der **Pins** wird dabei mit einer anderen **Adresse** von 2^3 verschiedenen Adressen angsprochen und jeweils 8-Bit können nach den Datenpfaden **??** auf einmal über einen Pin in ein **Register** der **UART** geschrieben werden, um **versandt** zu werden oder von einem Pin **empfangen** werden. Die **UART** kann z.B. genutzt werden, um Daten an einen sehr einfach gehaltenen **Monitor** zu senden, der diese dann anzeigt.

An letzter Stelle muss der **SRAM**[12] erwähnt werden, bei dem es sich um den **Hauptspeicher** der **RETI-CPU** handelt. Der **Zugriff auf den Hauptspeicher** ist deutlich schneller als z.B. auf ein **externes Speichermedium**, aber **langsamer** als der **Zugriff auf Register**.

1.2 Die Sprache PicoC

Die Sprache L_{PicoC} ist eine Untermenge der Sprache L_C, welche

- **Einzeilige Kommentare** `//` and **Mehrzeilige Kommentare** `/*` and `*/`
- die **Primitiven Datentypen** `int`, `char` und `void`
- die **Abgeleiteten Datentypen Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`)
- `if(cond){ }`- / `else{ }`-**Statements**[13]
- `while(cond){ }`- und `do while(cond){ };`-**Statements**
- **Arithmetische Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` und **unären Operatoren** `-`, $\sim$ umgesetzt sind
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`) und **Funktionsaufrufe** (z.B. `fun(ar, st_var)`)

[6] Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, sondern nur bei der Umsetzung des **RETI-Interpreters**, wird diese **nicht näher erläutert** als notwendig.
[7] Von 0 bis $2^{32} - 1$.
[8] Kurz für **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
[9] Kurz für **U**niversal **A**synchronous **R**eceiver **T**ransmitter.
[10] Engl. **R**eceiving, daher das `R`.
[11] Engl. **T**ransmission, daher das `T`.
[12] Kurz für **S**tatic **r**andom-**a**ccess **m**emory.
[13] Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

beinhaltet. Die ausgegrauten • wurden bereits für das **Bachelorprojekt** umgesetzt und mussten für die **Bachelorarbeit** nur an die **neue Architektur** angepasst werden.

Der **Aufbau** der **Programmiersprache** L_C ist durch Grammatik **??** und Grammatik **??** zusammengenommen beschrieben.

1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache L_C, die genauso ein Teil der Programmiersprache L_{PicoC} sind, da L_{PicoC} eine Untermenge von L_C ist und welche in der Implementierung des **PicoC-Compilers** in Kapitel **??** noch eine **wichtige Rolle** spielen werden im Folgenden genauer erläutert. Im Folgenden wird immer von der Programmiersprache L_{PicoC} gesprochen, da es in dieser Bachelorarbeit um diese geht und die folgenden Beispiele für die Ausgaben alle mithilfe des **PicoC-Compilers** und **RETI-Interpreters** **kompiliert** bzw **ausgeführt** wurden, aber selbiges gilt genauso für L_C aus bereits erläutertem Grund.

Bei der Programmiersprache L_{PicoC} handelt es sich im eine **imperative** (Definition 1.1), **strukturierte** (Definition 1.2) und **prozedurale Programmiersprache** (Definition 1.3). Aufgrund dessen, dass es sich bei beiden um **Imperative Programmiersprachen** handelt ist es wichtig bei der Implementierung die **Reihenfolge** zu beachten und aufgrund dessen, dass es sich bei beiden um **Strukturierte** und **Prozedurale Programmiersprachen** handelt, ist es eine gute Methode bei der Implementierung auf **Blöcke**[14] zu setzen zwischen denen **hin und her** gesprungen werden kann und welche in den einzelnen Implementierungsschritten die **notwendige Datenstruktur** darstellen um **Auswahl** zwischen Codestücken, **Wiederholung** von Codestücken und **Sprünge** zu Blöcken mit entsprechend zu bestimmten **Bezeichnern** passenden **Labeln** umzusetzen.

Definition 1.1: Imperative Programmierung

*Wenn ein Programm aus einer **Folge von Befehlen** besteht, **deren Reihenfolge** auch bestimmt in welcher **Reihenfolge** diese **Befehle** auf einer **Maschine** ausgeführt werden.*[a]

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

Definition 1.2: Strukturierte Programmierung

*Wenn ein Programm anstelle von z.B. `goto label`-Statements **Kontrollstruturen**, wie z.B. `if(cond) { } else { }`, `while(cond) { }` usw. verwendet, welche dem Programmcode **mehr Struktur** geben, weil die **Auswahl** zwischen Statements und die **Wiederholung** von Statements eine **klare und eindeutige Struktur** hat, welche bei Umsetzung mit einem `goto label`-Statement **nicht** so **eindeutig erkennbar** wäre und auch **nicht** umbedingt immer **gleich aufgebaut** wäre.*[a]

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

Definition 1.3: Prozedurale Programmierung

*Programme werden z.B. mittels **Funktionen** in **überschaubare Unterprogramme** bzw. **Prozeduren** aufgeteilt, die **aufrufbar** sind. Dies **vermeidet** einerseits **redundanten Code**, indem Code **wiederverwendbar** gemacht wird und andererseits erlaubt es z.B. Codestücke nach ihren Aufgaben zu **abstrahieren**, den Codestücken wird eine **Aufgabe zugeteilt**, sie werden zu **Unterprogrammen** gemacht und fortan über einen **Bezeichner aufgerufen**, was den Code deutlich **überschaubarer** macht. da man die Aufgabe eines Codestücks nun nur noch mit seinem **Bezeichner assozieren** muss.*[a]

[14] Werden später im Kapitel **??** genauer erklärt.

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

In L_C ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen `[<i>]`-Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: `<remaining-datatype><var>[<i>]`, während andere Programmiersprachen die eckigen `[<i>]`-Klammern **vor** die Variable schreiben `<remaining-datatype>[<i>]<var>`.

Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, ist es **schwieriger** den **Datentyp abzulesen**, als auch ein **Programm zu implementieren** was diesen erkennt. Damit ein Programmierer den **Datentyp ablesen** kann, kann dieser die **Spiralregel** verwenden, die unter *Clockwise/Spiral Rule* nachgelesen werden kann. Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, wirken diese zum verwechseln ähnlich zum `<var>[<i>]`-Operator für den **Zugriff auf den Index eines Feldes**. Wenn ein Ausdruck geschrieben wird, wie `int ar[1] = {42}` wird, ist dieser vom Ausdruck `var[0] = 42` nur durch den **Kontext** um `var[1]` bzw. `var[0]` rum zu unterscheiden.

In Code 1.1 ist ein Beispiel zu sehen, indem die Variable `complex_var` den **Datentyp „Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Zeigern auf Felder der Mächtigkeit 2 von Verbunden vom Typ `st`“** hat. Ein Vorteil die eckigen `[<i>]`-Klammern **hinter** die Variable zu schreiben ist in der **markierten Zeile** in Code 1.1 zu sehen. Will man auf ein **Element dieses Datentyps** zugreifen `(*complex_var[0][1])[1].attr`, so ist der Ausdruck fast genau **gleich aufgebaut**, wie der Ausdruck für den **Datentyp** `struct st (*complex_var[1][2])[2]`. Die **Ausgabe** des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

```
1 struct st {int attr;};
2
3 void main() {
4   struct st st_var[2] = {{.attr=314}, {.attr=42}};
5   struct st (*complex_var[1][2])[2] = {{&st_var, &st_var}};
6   print((*complex_var[0][1])[1].attr);
7 }
```

Code 1.1: *Beispiel für Spiralregel*

```
1 42
```

Code 1.2: *Ausgabe von Beispiel für Spiralregel*

In L_C ist die **Ausführbarkeit einer Operation** oder **wie** diese Operation ausgeführt wird davon abhängig, was für einen **Datentyp** die Variable in diesem **Kontext** der Operation hat. In dem Beispiel in Code 1.3 wird in Zeile 2 ein **„Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2“** und Zeile 3 ein **„Zeiger auf Felder der Mächtigkeit 2“** erstellt. In den **markierten Zeilen** wird zweimal in Folge die **gleiche Operation** `<var>[0][1]` ausgeführt, allerdings hat die Operation aufgrund der **unterschiedlichen Datentypen** der Variablen einen **unterschiedlichen Effekt**.

In Zeile 4 wird ein **normaler Zugriff** auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt und in Zeile 5 wird allerdings erst dem **Zeiger** `int (*pntr)[2] = &ar[0];` **gefolgt** und dann ein Zugriff auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt. **Beide Operationen** haben, wie in Code 1.4 zu sehen ist die **gleiche Ausgabe**.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(ar[0][1]);
5   print(pntr[0][1]);
6 }
```

Code 1.3: *Beispiel für unterschiedliche Ausführung*

```
1 42 42
```

Code 1.4: *Ausgabe des Beispiels für unterschiedliche Ausführung*

Eine weitere interessante Eigenheit, die tätsächlich nur in der **Untermenge** von L_C, die L_{PicoC} darstellt gültig ist[15], ist dass die Operationen `<var>[0][1]` und `*(*(<var>+0)+1)` aus Code 1.3 und Code 1.5 komplett **austauschbar** sind. Die Ausgabe in Code 1.4 ist folglich **identisch** zur Ausgabe in Code 1.6.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(*(*(ar+0)+1));
5   print(*(*(pntr+0)+1));
6 }
```

Code 1.5: *Beispiel mit Dereferenzierungsoperator*

```
1 42 42
```

Code 1.6: *Ausgabe des Beispiels mit Dereferenzierungsoperator*

In der Programmiersprache L_{PicoC} werden alle **Argumente** bei einem Funktionsaufruf nach der **Call By Value**-Strategie (Definition 1.4) übergeben. Ein Beispiel hierfür ist in Code 1.7 zu sehen. Hierbei wird ein **Verbund** `struct st copyable_ar = {.ar={314, 314}};`[16] an die Funktion `fun` übergeben. Hierzu wird der **Verbund** in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopiert** und an den Parameter `fun` gebunden.

Wie an der Ausgabe in Code 1.7 zu sehen ist hat die **Zuweisung** `copyable_ar.ar[1] = 42` an den Parameter `struct st copyable_ar` in der **aufgerufenen** Funktion `fun` keinen Einfluss auf die übergebene **lokale Variable** `copyable_ar` der **aufrufenden** Funktion. Der Eintrag an Index 1 im Feld bleibt bei `314`.

[15] In der Sprache L_C gibt es einen Unterschied bei der Initialisierung bei z.B. `int *var = "string"` und z.B. `int var[1] = "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 (z.B. `var[1]`), obwohl es ein **Derived Datatype** ist.

[16] **Später** wird darauf eingegangen, warum der Verbund den **Bezeichner** `copyable_ar` erhalten hat.

Definition 1.4: Call by Value

*Es wird eine **Kopie** des **Ergebnisses eines Ausdrucks**, welcher ein **Argument eines Funktionsaufrufes** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.*

*Das hat zur Folge, dass bei **Übergabe** einer Variable als **Argument** an eine Funktion, diese Variable bei **Änderungen** am entsprechenden **Parameter** der **aufgerufenen** Funktion in der **aufrufenden** Funktion **unverändert** bleibt.*[a]

[a] Bast, „Programmieren in C“.

```
1  struct st {int ar[2];};
2
3  int fun(struct st copyable_ar) {
4    copyable_ar.ar[1] = 42;
5  }
6
7  void main() {
8    struct st copyable_ar = {.ar={314, 314}};
9    print(copyable_ar.ar[1]);
10   fun(copyable_ar);
11   print(copyable_ar.ar[1]);
12 }
```

Code 1.7: *Beispiel dafür, dass Struct kopiert wird*

```
1 314 314
```

Code 1.8: *Ausgabe von Beispiel, dass Struct kopiert wird*

In der Programmiersprache L_{PicoC} gibt es **kein Call by Reference** (Definition 1.5), allerdings kann der **Effekt** von Call by Reference mittels **Zeigern simuliert** werden, wie es in Code 1.11 bei der Funktion `fun_declared_before` und dem Parameter `int *param` zu sehen ist. Genau dieser **Trick** wird bei **Feldern** verwendet, um **nicht** das gesamte Feld bei einem Funktionsaufruf in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopieren** zu müssen.

Wie im Beispiel in Code 1.9 zu sehen ist, wird in der markierten Zeile ein Feld `int ar[2] = {314, 314}` an die Funktion `fun` übergeben. Wie in der **Ausgabe** in Code 1.10 zu sehen ist, hat sich der Eintrag an Index 1 im Feld nach dem **Funktionsuaufruf** zu `42` geändert. Wird ein Feld direkt als Ausdruck `ar` ohne z.B. die eckigen `[]`-Klammern für einen **Indexzugriff** hingeschrieben wird die **Adresse** des Felds verwendet und **nicht** z.B. der **erste Eintrag** des Felds.

Eine Möglichkeit ein **Feld** als **Kopie** und **nicht** als **Referenz** zu übergeben ist es, wie in Code 1.7 das **Feld** als **Attribut** eines **Verbundes** zu übergeben, wie bei der Variable `copyable_ar`.

Definition 1.5: Call by Reference

*Es wird eine **implizite Referenz** einer Variable, welche ein **Argument eines Funktionsaufrufes** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.*

***Implizit** meint hier, dass der Benutzer einer Programmiersprache mit Call by Reference **nicht mitbekommt**, dass er das **Argument selbst verändert** und **keine lokale Kopie des Arguments**.*[a]

[a] Bast, „Programmieren in C“.

```
1  int fun(int ar[2]) {
2    ar[1] = 42;
3  }
4
5  void main() {
6    int ar[2] = {314, 314};
7    print(ar[1]);
8    fun(ar);
9    print(ar[1]);
10 }
```

Code 1.9: *Beispiel dafür, dass Zeiger auf Feld übergeben wird*

```
1 314 42
```

Code 1.10: *Ausgabe von Beispiel dafür, dass Zeiger auf Feld übergeben wird*

Ein Programm in der Programmiersprache L_{PicoC} wird von **oben-nach-unten** ausgewertet. Ein Problem tritt auf, wenn z.B. eine Funktion verwendet werden soll, die aber erst **unter** dem entsprechenden Funktionsaufruf **definiert** (Definition 1.8) wird. Es ist wichtig, dass der **Prototyp** (Definition 1.6) einer Funktion vorher durch die **Funktionsdefinition** bekannt ist, damit überprüft werden kann, ob die beim Funktionsaufruf **übergebenen Argumente** den gleichen **Datentyp** haben, wie die **Parameter** des **Prototyps** und ob die **Anzahl Argumente** mit der **Anzahl Parameter** des **Prototyps** übereinstimmt.

Allerdings lassen sich die Funktionen **nicht** immer so anordnen, dass jede in einem Funktionsaufruf referenzierte Funktion vorher definiert sein kann. Aus diesem Grund ist es möglich den **Prototyp** einer Funktion vorher zu **deklarieren** (Definition 1.7), wie es in den **markierten Zeile** im Beispiel in Code 1.11 zu sehen ist. Die **Ausgabe** des Beispiels ist in Code 1.12 zu sehen.

Definition 1.6: Funktionsprototyp

***Deklaration** einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parametereihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktiondeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.*[a] [b]

[a] Der **Funktionsprototyp** ist von der **Funktionsignatur** zu unterschieden, die in Programmiersprache wie `C++` und `Java` für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentpyen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentpyen** usw. unterschieden.

[b] *What is the difference between function prototype and function signature?*

Definition 1.7: Deklaration

*Der **Datentyp** bzw. **Prototyp** einer **Variablen** bzw. **Funktion**, sowie der **Bezeichner** dieser **Variable** bzw. **Funktion** wird dem **Compiler** mitgeteilt.*[abc]

[a] Über das **Schlüsselwort** `extern` lassen sich in der Programiersprache L_C Veriablen **deklarieren**, **ohne** sie zu **definieren**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. D. P. Scholl, „Einführung in Embedded Systems“.

Definition 1.8: Definition

*Dem **Compiler** wird mitgeteilt, dass zu einem **bestimmten Zeitpunkt** in der Programmausführung **Speicher** für eine **Variable angelegt** werden soll und **wo**[a] dieser angelegt werden soll. Eine Funktion ist definiert ihr eine **relative Anfangsadresse** im **Hauptspeicher** zugewiesen werden kann, aber welcher die **Maschinenbefehle** für diese Funktion abgespeichert werden können.*[bc]

[a] Im Fall des PicoC-Compilers in den **Globalen Statischen Daten** oder auf dem **Stack**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. D. P. Scholl, „Einführung in Embedded Systems“.

```
1  void fun_declared_before(int *param);
2
3  int fun_defined(int param) {
4    return param + 10;
5  }
6
7  void main() {
8    int res = fun_defined(22);
9    fun_declared_before(&res);
10   print(res);
11 }
12
13 void fun_declared_before(int *param) {
14   *param = *param + 10;
15 }
```

Code 1.11: *Beispiel für Deklaration und Definition*

```
1 42
```

Code 1.12: *Ausgabe von Beispiel für Deklaration und Definition*

In L_{PicoC} lässt sich eine **definierte Variable** nur innerhalb ihres **Sichtbarkeitsbereichs** (Definition 1.9) verwenden. **Lokale Variablen** und **Parameter** lassen sich nur innerhalb der **Funktion** in welcher sie deklariert bzw. definiert wurden verwenden. Der **Sichtbarkeitsbereich** von **Lokalen Variablen** und **Parametern** erstreckt sich herbei von der **öffnenden** {-Klammer bis zur **schließenden** }-Klammer der **Funktionsdefinition**, in welcher sie definiert wurden.

Verschiedene **Sichtbarkeitsbereiche** können dabei **identische** Bezeichner besitzen. Im Beispiel in Code 1.13 kommt der markierte **Bezeichner** `local_var` in 2 verschiedenen **Sichtbarkeitsbereichen** vor, doch bezeichnet er 2 **unterschiedliche Variablen**. Der **Parameter** `param` und die **Lokale Variable** `local_var` dürfen **nicht**

den **gleichen Bezeichner** haben, da sie sich im gleichen **Sichtbarkeitsbereich** der Funktion `fun_scope` befinden. Die **Ausgabe** des Beispiels in Code 1.13 ist in Code 1.14 zu sehen.

Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)

***Bereich** in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.*[a]

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

```
1  int fun_scope(int param) {
2    int local_var = 2;
3    print(param);
4    print(local_var);
5  }
6
7  void main() {
8    int local_var = 4;
9    fun_scope(local_var);
10 }
```

Code 1.13: *Beispiel für Sichtbarkeitsbereichs*

```
1 4 2
```

Code 1.14: *Ausgabe von Beispiel für Sichtbarkeitsbereichs*

1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache L_{PicoC} in die Maschinensprache L_{RETI} die **Syntax** und **Semantik** der Sprache L_C identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache L_{PicoC} im Vergleich zu z.B. dem **GCC**[17] ohne merklichen Unterschied[18] komplieren können.

In **zweiter Linie** soll dabei möglichst immer so Vorgegangen werden, wie es die **RETI-Codeschnipsel** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ vorgeben. Allerdings sollten diese bei **Inkonsistenzen** bezüglich der durch sie selbst vorgegebenen **Paradigmen** und anderen **Umstimmigkeiten** angepasst werden, da der **erstere Schwerpunkt** überwiegt.

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

[17] Da die Sprache L_{PicoC} eine **Untermenge** von L_C ist, kann der **GCC** L_{PicoC} ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinensprache L_{RETI}.

[18] Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinensprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter Link[19] zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter **„Getting Started“** ein kleines **Einführungstutorial** verlinkt. Unter **„Usage“** ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter Link[20] zu finden.

Abbildung 1.4: *README.md im Github Repository der Bachelorarbeit*

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schrifftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter Link[21] zu finden. Die **PDF** der Schrifftliche Ausarbeitung der Bachleorararbeit wird aus dem **Latexquellcode**, welcher unter Link[22] veröffentlicht ist automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[23] zu finden[24]. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**[25] erstellt. Falls Interesse besteht **Grafiken** aus der Schrifftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den `.svg`-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[26] zu finden.

[19] https://github.com/matthejue/PicoC-Compiler.
[20] https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971.
[21] https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.
[22] https://github.com/matthejue/Bachelorarbeit.
[23] https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.
[24] Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.
[25] Developers, *Draw Freely — Inkscape*.
[26] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

Um die verschiedenen **Aspekte** dieser Schrifftlichen Ausarbeitung der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel Motivation werden Codebeispiele zur **Anschauung** verwendet und mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters Ausgaben** erzeugt, die in dieses Dokument **eingelesen** wurden. In Kapitel **??** werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** gezeigt[27].

Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[28] finden und mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[29].

1.5.1 Still der Schrifftlichen Ausarbeitung

In dieser **Schrifftliche Ausarbeitung der Bachelorarbeit** sind die manche **Wörter** für einen besseren Lesefluss **hervorgehoben**. Es ist so gedacht, dass die **Hervorgehobenen Wörter** beim Lesen sichtbare **Ankerpunkte** darstellen an denen sich **orientiert** werden kann, aber auch damit der **Inhalt** eines vorher gelesener **Paragraphs** nochmal durch Überfliegen der Hervorgehobenen Wörter in **Erinnerung gerufen** werden kann.

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas geanu so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht[30] die Frage des **„warum“**.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**[31] verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **konsistent** sind, wie auch das in Ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernomnen, so wurde die Definition oder der entsprechende Teil in **„Anführungszeichen“** gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox**, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

1.5.2 Aufbau der Schrifftlichen Arbeit

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: Motivation, Einführung, **??** und **??**.

Im momentanen Kapitel Motivation, wurde ein kurzer **Einstieg** in das Thema **Compilerbau** gegeben und die **zentrale Aufgabenstellung** der Bachelorarbeit erläutert, sowie auf **Schwerpunkte** und kleinere **Teilprobleme**, die eines **besonderen Fokus** bedürfen eingegangen.

Im Kapitel Einführung werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische**

[27] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[28] `https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples`.
[29] Es wurde zu diesem Zweck die **Command-line Option** `-t`, `--thesis` erstellt, die bestimmte Kommentare **herausfiltert**, damit die generierten **Abstrakten Syntaxbäume** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu **überfüllt** mit Kommentaren sind.
[30] Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[31] Das ist ein **Beispiel** für eine **Fußnote**.

Vorgehen, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen in Bezug zu Compilern und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

Des Weiteren wurden für **T-Diagramme** und **Formale Sprachen** eigene Unterkapitel erstellt. Für **T-Diagramme** wurde ein eigenes Unterkapitel erstellt, da sie häufig in der Schrifftlichen Ausarbeitung verwendet werden und die **T-Diagramm Notation** nicht allgemein bekannt ist. Für **Formale Sprachen** wurde ein eigenes Unterkapitel erstellt, da für den Gutachter Prof. Dr. Scholl das Thema **Formale Sprachen** eher **fachfremd** ist, aber dieses Thema einige zentrale und wichtige Fachbegriffe besitzt, bei denen es wichtig ist die genaue **Definition** zu haben. Generell wurde im Kapitel **Einführung** versucht an Erklärungen nicht zu sparren, damit aufgrund dessen, dass das Thema eher fachfremd für Prof. Dr. Scholl ist für das Kapitel **Implementierung** keine wichtigen Aspekte unverständlich bleiben.

Im Kapitel **??** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers**, unterteilt in die verschiedenen **Phasen der Kompilierung** nach dennen das Kapitel **Einführung** ebenfalls unterteilt ist erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähliche Kapiteleinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständis eines Aspekts in Kapitel **Implemenentierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteleinteilung** die entsprechende Definition analog im Kapitel **Einleitung** gefunden werden.

Im Kapitel **??** wird ein **Überblick** über die **wichtigsten Funktionalitäten** des PicoC-Compilers gegeben, indem anhand **kleiner Anleitungen** gezeigt wird wie man diese verwendet. Des Weiteren wird darauf eingegangen, wie die **Qualitätsicherung** für den **PicoC-Compiler** umgesetzt wurde, also wie gewährleistet wird, dass der **PicoC-Compiler** funktioniert. Zum Schluss wird noch auf **weitere Erweiterungsideen** eingegangen, die auch interessant zu implementieren wären.

Im Kapitel **??** werden einige **Definitionen** und **Themen** angesprochen, die bei **Interesse** zur **weiteren Vertiefung** da sind und **unabhänging** von den anderen Kapiteln sind. Diese **Themen** und **Defintionen** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Diese **Themen** und **Definitionen** passen nicht ins Kapitel **??**, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tuen haben und auch nichts ins **Kapitel** Einführung, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in der Schrifftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition **??**) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

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 **??**).

Definition 2.1: Interpreter

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

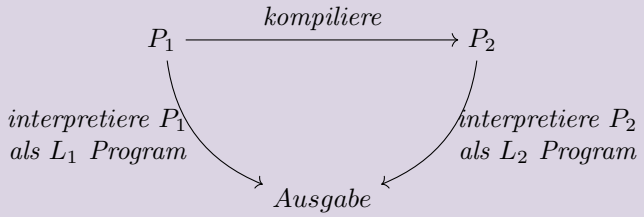
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstrakten Syntaxbaumes*** *(Definition 2.44) und führt je nach Komposition der* ***Knoten*** *des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

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

Definition 2.2: Compiler

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

Wobei ***Kompilieren*** *meint, dass ein beliebiges Program P_1 in der Sprache L_1 so in die Sprache L_2 zu einem Programm P_2 übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen L_1 und L_2* ***interpretiert*** *werden, die gleiche* ***Ausgabe*** *rauskommt, wie es in Diagramm 2.2.1 dargestellt ist. Also beide Programme P_1 und P_2 die gleiche* ***Semantik*** *(Definition 2.15) haben und sich nur* ***syntaktisch*** *(Definition 2.14) 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

Anmerkung

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

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschienenncode**, der in **Maschienensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition **??**) oder **Cross-Compiler** (Definition 2.5). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition **??**) 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 2.4) haben.

[b] P. D. C. Scholl, „Betriebssysteme“.

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

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschienencode, der die Maschienenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.4) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung[2].

Definition 2.4: Immediate

***Konstanter Wert**, der als **Teil** eines **Maschienenbefehls** gespeichert ist und dessen **Wertebereich** dementsprechend auch durch die die Anzahl an Bits, die ihm innerhalb dieses **Maschienenbefehls** zur Verfügung gestellt sind, **beschränkter** ist als bei sonstigen Werten innerhalb des Hauptspeichers,*

[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.

denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.[a]

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

Definition 2.5: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L_w geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschienensprachen** B_1 und B_2 haben.*[a b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C^{Python}_{PicoC}.
[b] Earley und Sturgis, „A formalism for translator interactions“.

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

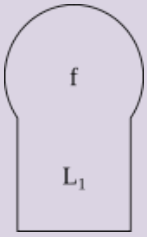
2.1.1 T-Diagramme

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 2.6), einen Übersetzer (Definition 2.7), einen Interpreter (Definition 2.8) und eine Maschiene (Definition 2.9) zusammen.

Definition 2.6: T-Diagram Programm

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



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

Anmerkung

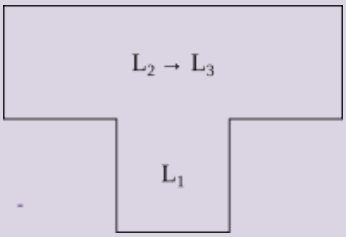
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.6 also reichen einfach eine 1 hinzuschreiben.

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

Definition 2.7: T-Diagram Übersetzer (bzw. eng. Translator)

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

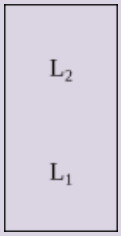
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 2.2.1.*[b]



[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 2.8: T-Diagram Interpreter

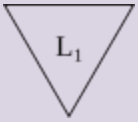
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



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

Definition 2.9: T-Diagram Maschiene

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



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

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

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

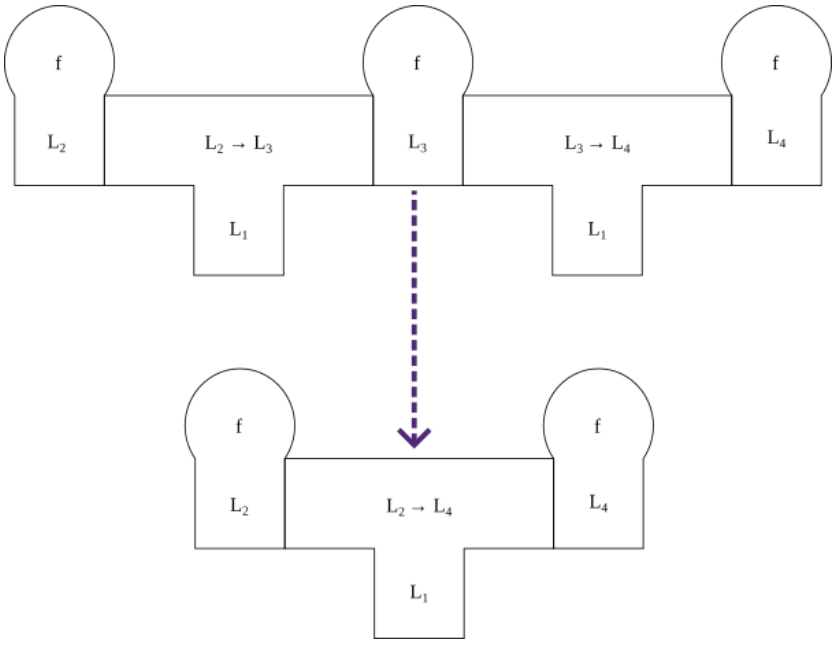


Abbildung 2.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen*

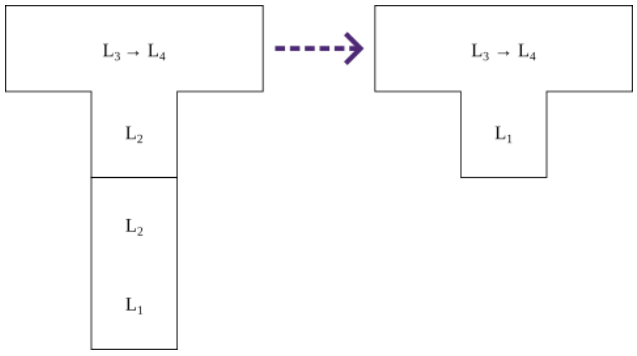


Abbildung 2.2: *Vertikale Interpretierungszwischenschritte zusammenfassen*

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.13) zu tuen, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der Sprache L_1 in eine Sprache L_2 ist. Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen**, was die Begriffe **Symbol** (Definiton 2.10), **Alphabet** (Definition 2.11), **Wort** (Definition 2.12) usw. beinhaltet vorher eingeführt zu haben.

Definition 2.10: Symbol

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.11: Alphabet

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 2.10).“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.12: Wort

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.13: Formale Sprache

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.12) über dem **Alphabet** Σ (Definition 2.11).“*[a]

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

[a] Nebel, „Theoretische Informatik“.

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die **Semantik** (Definition 2.15) **gleich** bleiben. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 2.16), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 2.14) haben.

Definition 2.14: Syntax

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

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

Definition 2.15: Semantik

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

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

Definition 2.16: Formale Grammatik

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

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

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

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

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

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

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

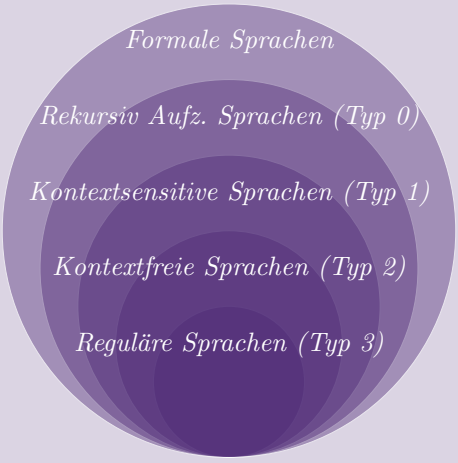
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 2.17) einteilen.

Definition 2.17: Chromsky Hierarchie

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

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

Zwischen den Sprachmengen ***benachbarter Klassen*** *in Abbildung 2.17.1 besteht eine* ***echte Teilmengenbeziehung****:* $L_3 \subset L_2 \subset L_1 \subset L_0$*. Jede* ***Reguläre Sprache*** *ist auch eine* ***Kontextfreie Sprache****, aber nicht jede* ***Kontextfreie Sprache*** *ist auch eine* ***Reguläre Sprache****.*[a]



(2.17.1)

[a] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 2.18) und **Kontextfreien Grammatiken** (Definition 2.19) beschrieben werden.

Definition 2.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{2.18.1}$$

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

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

Definition 2.19: Kontextfreie Grammatik

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

$$A \to v \tag{2.19.1}$$

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

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

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des **Wortproblems** (Definition 2.20). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise immer **entscheidbar**. Wenn das Programm ein **Wort** der **Sprache** ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es **kein Wort** der **Sprache**, die der Compiler kompiliert, wird eine **Fehlermeldung** ausgegeben.

Definition 2.20: Wortproblem

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

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

$$\mathbb{1}_L : \Sigma^* \to \{0,1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \tag{2.20.1}$$

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

2.2.1 Ableitungen

Um sicher zu wissen, ob ein Compiler ein **Programm**[4] kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprache** des Compilers **abzuleiten**. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 2.21) und der normalen **Ableitungsrelation** (Definition 2.22) unterschieden.

[4] Bzw. **Wort**.

Definition 2.21: 1-Schritt-Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.22: Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**[5] kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 relevant.

Definition 2.23: Links- und Rechtsableitungableitung

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

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

[a] Nebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 2.24) ist[6].

Definition 2.24: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

[a] *Parsing Expressions · Crafting Interpreters*.

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

Um herauszufinden, ob eine Grammatik **mehrdeutig** (siehe Unterkapitel **??**) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 2.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 2.37) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 2.25: Formaler Ableitungsbaum

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

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

*Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.16) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** N und die **Blätter** sind entweder **Terminalsymbole** Σ oder das **leere Wort** ε.*[b]

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

In Abbildung 2.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 2.25.1 nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition **??**) angegebenen **Produktionen** 2.1 einer ansonsten nicht näher spezifizierten **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ ergibt.

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

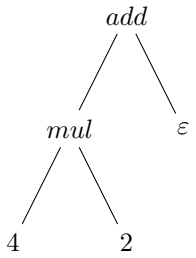
Grammatik 2.1: *Produktionen für Ableitungsbaum in EBNF*

Anmerkung

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

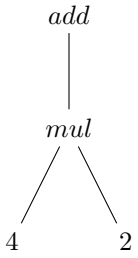
$$add \Rightarrow mul \Rightarrow mul\ ”*”\ NUM \Rightarrow NUM\ ”*”\ NUM \Rightarrow 4\ ”*”\ NUM \Rightarrow 4\ ”*”\ 2 \tag{2.25.1}$$

Bei Ableitungsbäumen gibt es **keine** einheutliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 2.25.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort** ε hinzuzufügen.



(2.25.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 2.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[7].



(2.25.3)

Für einen Compiler ist es notwendig, dass die **Konkrette Grammatik** keine **Mehrdeutige Grammatik** (Definition 2.26) ist, denn sonst können unter anderem die **Präzidenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel **??** an einem Beispiel demonstriert wird.

Definition 2.26: Mehrdeutige Grammatik

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

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

2.2.2 Präzidenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Konkretten Grammatik** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 2.27) und **Präzidenz** (Definition 2.28) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel **??** genauer erklärt.

Definition 2.27: Assoziativität

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

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

[a] *Parsing Expressions · Crafting Interpreters*.

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

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

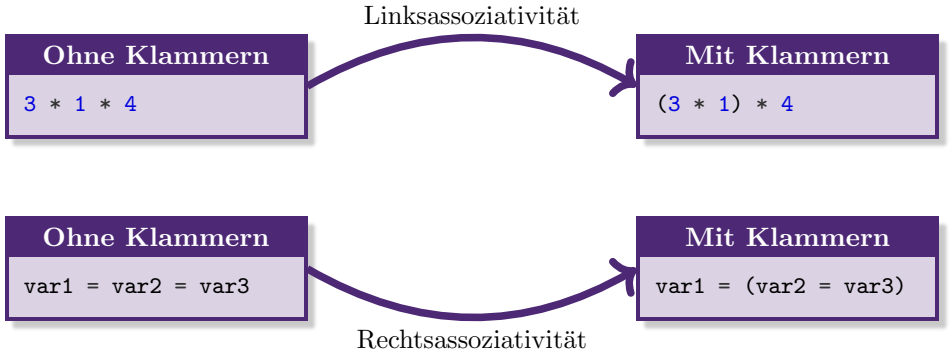


Abbildung 2.3: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität*

Definition 2.28: Präzidenz

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

[a] *Parsing Expressions · Crafting Interpreters.*

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

Abbildung 2.4: *Veranschaulichung von Präzidenz*

2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.29) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Eingabewort, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.30) matchen, die durch eine **reguläre Grammatik** spezifiziert sind. Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.31) genannt.

Definition 2.29: Pipe-Filter Architekturpattern

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

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

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

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

Definition 2.30: Pattern

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

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] Thiemann, „Compilerbau“.

Definition 2.31: Lexeme

*Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches von einem **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} erkannt wird.*[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.32) im **Eingebewort** 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 2.32) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.32: Lexer (bzw. Scanner oder auch Tokenizer)

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

[a] Thiemann, „Compilerbau“.

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

Anmerkung

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 2.33) von **Variablen, Konstanten und Funktionen** die Symbole[a].

[a] Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel ?? **Symboltabelle** genannt wird.

Definition 2.33: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres **Scopes eindeutig** benennt.*[a][b]

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

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[8] und Tabs `\t` aus dem Eingebewort herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** *. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie `42`, `314` oder `12`. Die **Überbegriffe** bzw. **Tokennamen** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`[9], bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie `if` oder `}` sind die **Tokennamen** bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.34) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[10]. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

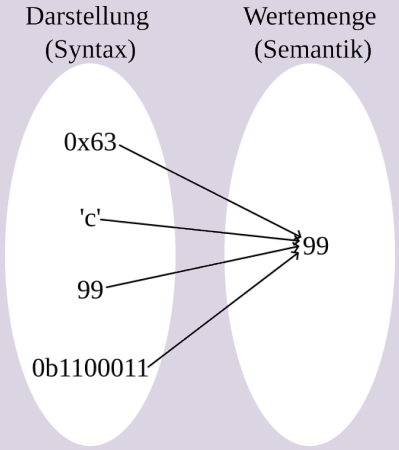
Die **Konkrette Grammatik** G_{Lex}, die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird ist

[8] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.
[9] Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Knoten haben will, damit unter anderem **mehr Code** in eine Zeile passt.
[10] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[11], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik **??** 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[12]. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

Definition 2.34: Literal

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



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

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

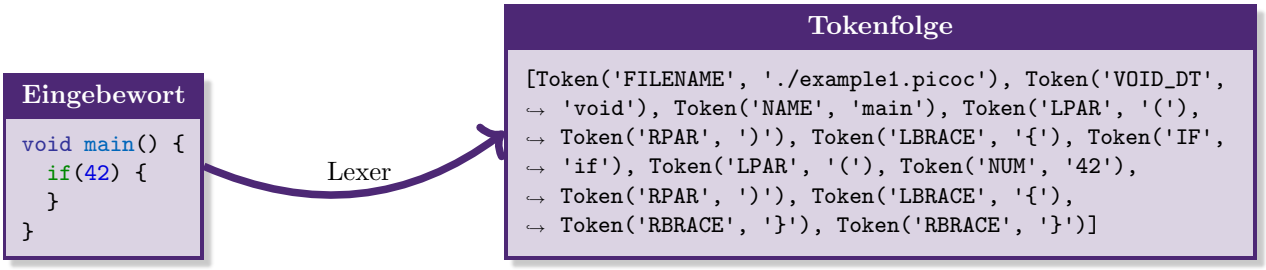


Abbildung 2.5: *Veranschaulichung der Lexikalischen Analyse*

2.4 Syntaktische Analyse

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

[11] Man nennt das auch einem **Lookahead** von 1
[12] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

eine entsprechende schließende runde Klammer `')'` bzw. schließende geschweifte Klammer `'}'` geschlossen wurden.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 2.38) ein **Ableitungsbaum** (Definition 2.37) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 2.44) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 2.35: Konkrette Syntax

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

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

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

Um einen kurzen Begriff für die **Grammatik**, welche die **Konkrette Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Konkrette Grammatik** (Definition 2.36) bezeichnet.

Definition 2.36: Konkrette Grammatik

*Grammatik, die eine **Konkrette Syntax** beschreibt.*

Definition 2.37: Ableitungsbaum (bzw. Konkretter Syntaxbaum oder auch engl. Derivation Tree)

***Compilerinterne Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 2.25) eines in **Konkretter Syntax** geschriebenen Programmes.*

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

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

Definition 2.38: Parser

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

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.38 verwendet.

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

Anmerkung

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 Eingebewort 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 2.39), denn ein Parser löst das **Wortproblem** (Definition 2.20) für die **Sprache**, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Ableitungsbaum**.

Definition 2.39: Recognizer (bzw. Erkenner)

*Entspricht einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit den Produktionen der **Konkretten Grammatik** einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkretten Grammatik** beschrieben wird oder nicht. Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** (Definition 2.20) bekannt.*[a]

[a] Thiemann, „Compilerbau“.

Anmerkung

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

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

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

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

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

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.24) allerdings nicht möglich, ohne die Konkrette Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der Konkretten Grammatik entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Konkrette Grammatiken parsen zu können, die nicht **LL(k)** (Definition **??**) sind. Dabei werden meist nach dem Prinzip der **Tiefensuche** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass das **Eingabewort** sich **nicht** mit der verwendeten Konkretten 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 Eingabewort **vorauszuschauen**. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.[c]

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

[a] *What is Top-Down Parsing?*
[b] Diese Form von Parsing wurde im **PicoC-Compiler** implementiert, als dieser noch auf dem Stand des **Bachelorprojektes** war, bevor er durch den nicht selbst implementierten **Earley Parser** von **Lark** (siehe *Lark - a parsing toolkit for Python*) ersetzt wurde.
[c] Diese Art von Parser ist im **RETI-Interpreter** implementiert, da die **RETI-Sprache** eine besonders simple **LL(1) Grammatik** besitzt. Diese Art von **Parser** wird auch als **Predictive Parser** oder **LL(k) Recursive Descent Parser** bezeichnet, wobei **Recursive Descent** das englische Wort für **Rekursiven Abstieg** ist.
[d] *What is Bottom-up Parsing?*
[e] Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstrakte Syntaxbaum** wird mithilfe von **Transformern** (Definition 2.40) und **Visitors** (Definition 2.41) generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese ein Programm von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.42).

Definition 2.40: Transformer

*Ein Programm, dass von **unten-nach-oben** nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaum** besucht und beim Antreffen eines bestimmten Knoten des **Ableitungsbaumes** je nach Kontext einen entsprechenden Knoten des **Abstrakten Syntaxbaumes** erzeugt und diesen anstelle*

*des Knotens des **Ableitungsbaumes** setzt und so Stück für Stück den **Abstrakten Syntaxbaum** konstruiert.*[a]

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

Definition 2.41: Visitor

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

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

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

Definition 2.42: Abstrakte Syntax

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

*Ein **Abstrakter Syntaxbaum**, der zur Kompilierung eines Wortes*[a] *generiert wurde ist nach einer **Abstrakten Grammatik** konstruiert.*

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

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

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

Um einen kurzen Begriff für die **Grammatik**, welche die **Abstrakte Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 2.43) bezeichnet.

Definition 2.43: Abstrakte Grammatik

*Grammatik, die eine **Abstrakte Syntax** beschreibt.*

Definition 2.44: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

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

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

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

[a] Den verschiedenen **Passes**.

In Abbildung 2.6 wird das Beispiel aus Unterkapitel 2.2.1 fortgeführt, welches den **Arithmetischen Ausdruck** `4 * 2` in Bezug auf die Konkrette Grammatik **??**, welche die **höhere Präzidenz** der **Multipikation** $*$ berücksichtigt in einem **Ableitungsbaum** darstellt. In Abbildung 2.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**. Das geschieht bezogen auf das Beispiel aus Unterkapitel 2.2.1, indem jegliche Knoten **wewgabstrahiert** werden, die im **Ableitungsbaum** nur existieren, weil die Konkrette Grammatik so umgesetzt ist, dass es nur **einen** einzigen möglichen **Ableitungsbaum** geben kann.

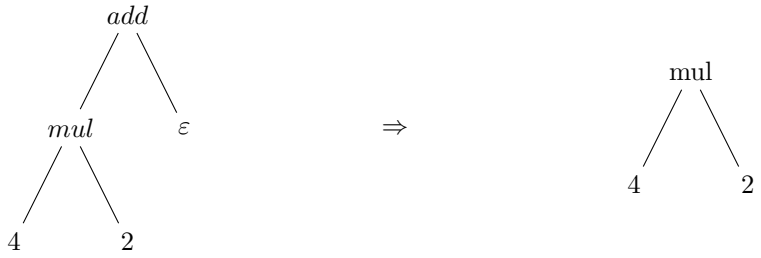


Abbildung 2.6: *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

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

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.7 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

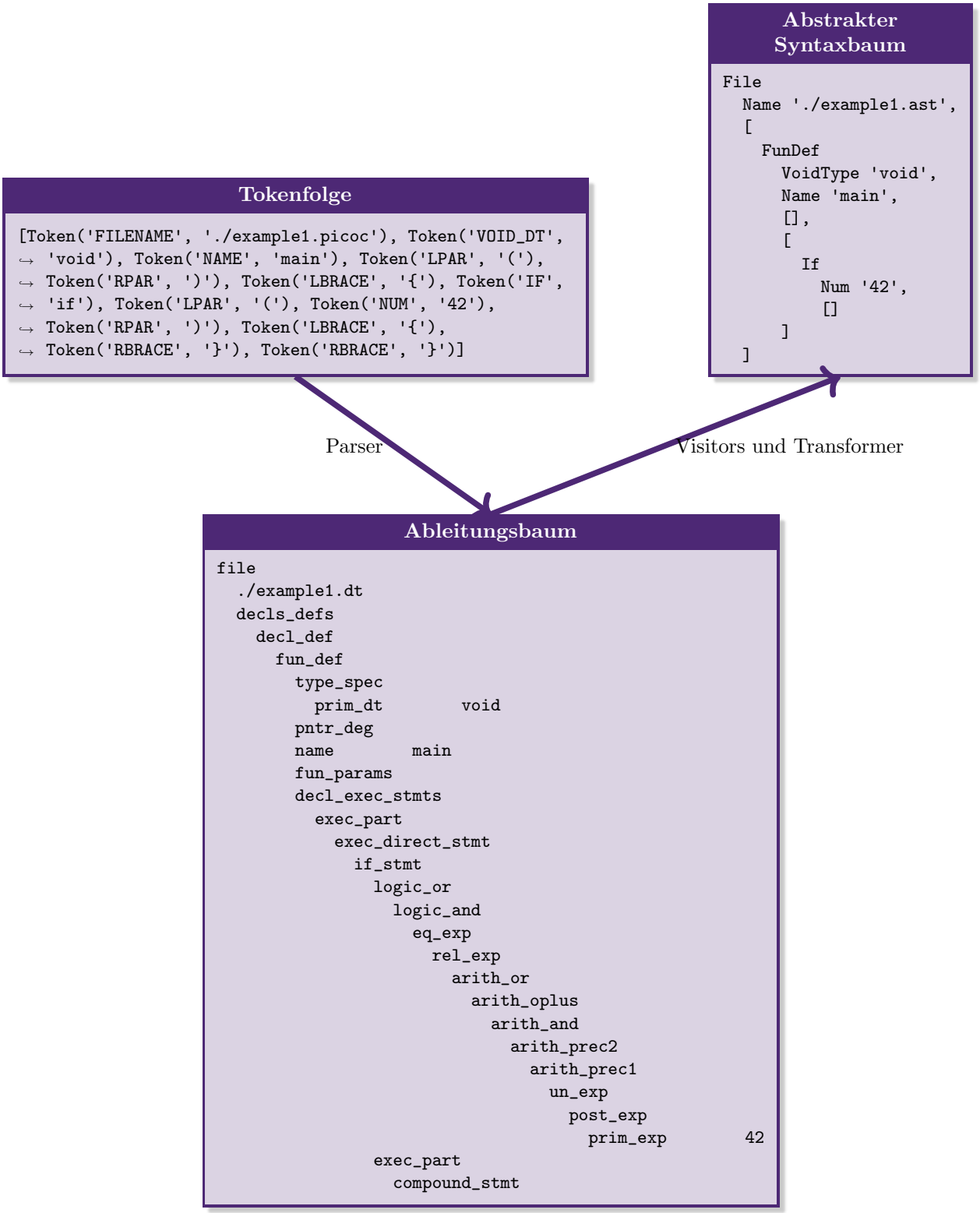


Abbildung 2.7: *Veranschaulichung der Syntaktischen Analyse*

2.5 Code Generierung

In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** einer Sprache L_1 in den **Abstrakten Syntaxbaum** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.45) nennt. So wie es auch schon mit dem **Ableitungsbaum** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakten Syntaxbaum** kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

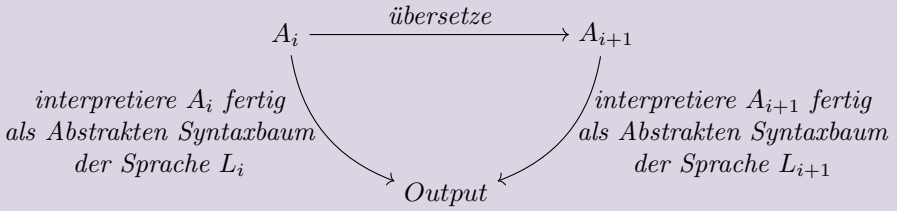
Anmerkung

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

Definition 2.45: Pass

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

Für ***jeden Pass*** *und für einen* ***beliebigen*** *Abstrakten Syntaxbaum* A_i *gilt ähnlich, wie bei einem* ***vollständigen Compiler*** *in 2.45.1, dass:*



(2.45.1)

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

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

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

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

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

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

2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche **Unreine Ausdrücke** (Definition 2.47) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.46) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 2.46: Reiner Ausdruck (bzw. engl. pure expression)

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

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

Definition 2.47: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definiton 2.48).

Definition 2.48: Monadische Normalform (bzw. engl. monadic normal form)

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

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

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

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[13] aufgeschrieben wurden.

In der Abbildung 2.8 ist der Ausdruck mit dem **Nebeneffekt** eine Variable zu **allokieren:** `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt, daher muss der **Unreine** Ausdruck als eigenständiges Statement **vorangestellt** werden.

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

Abbildung 2.8: *Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten*

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

2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um **Maschienensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.51) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Statements macht, in denen die **Komplexen Ausdrücke temporären Locations** zugewiesen werden (Definiton 2.49) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statemtent**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 2.50) enthalten.

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

Definition 2.49: Location

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

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

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.52). Wenn eine **Konkrette Grammatik** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.52), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.50).

Definition 2.50: Atomarer Ausdruck

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

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

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

Definition 2.51: Komplexer Ausdruck

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

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

Definition 2.52: A-Normalform (ANF)

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

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

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

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.9 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[14] aufgeschrieben wurden.

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

Aus diesem Grund verwendet das Beispiel in Abbildung 2.9 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **Abstrakte Syntaxbaum** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdrück** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

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

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

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

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

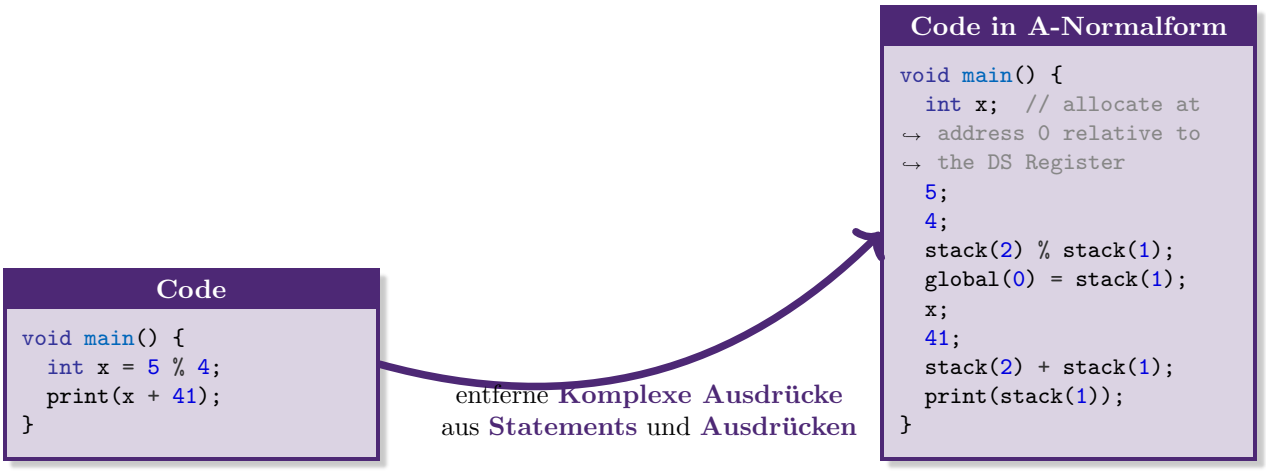


Abbildung 2.9: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen*

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

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

2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden ist es notwendig aus dem finalen **Abstrakten Syntaxbaum** den eigentlichen **Maschinencode** in **Konkretter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt. Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschienencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben. Der Weg von der **Abstrakten Syntax** zur **Konkretten Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkretten Syntax** zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden musste.

[16] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[17] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.

Jeder **Knoten** des **Abstrakten Syntaxbaumes** erhält dazu eine Methode, welche hier `to_string` genannt wird, die eine **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **Semikolons** ; usw. ausgibt. Dabei wird nach dem Prinzip der **Tiefensuche** der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die immer wiederum die Methode `to_string` ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgebeben**.

2.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**[18] eintritt:

1. der **Parser**[19] entscheidet das **Wortproblem** für ein **Eingabeprogramm**[20] mit `0`, also das **Eingabeprogramm** befolgt nicht die **Syntax** der **Sprache** des Compilers[21].

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

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

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

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

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

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

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

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

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

wird eine **Fehlermeldung** (Definition 2.53) ausgegeben.

Definition 2.53: Fehlermeldung

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

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

- *über eine* ***Dialogbox*** *in einer* ***Graphischen Benutzerfläche***[c] *oder* ***Zeichenorientierten Benutzerschnittstelle***[d]*.*

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

- *in ein **Register** oder an eine **spezielle Adresse** des **Hauptspeichers** wird ein **Wert geschrieben**.*
- ***Logdatei**[e] auf einem **Speichermedium**.*

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

Literatur

Online

- *A-Normalization: Why and How (with code)*. URL: https://matt.might.net/articles/a-normalization/ (besucht am 23. 07. 2022).
- *clang: C++ Compiler*. URL: http://clang.org/ (besucht am 29. 07. 2022).
- *Clockwise/Spiral Rule*. URL: https://c-faq.com/decl/spiral.anderson.html (besucht am 29. 07. 2022).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: https://inkscape.org/ (besucht am 03. 08. 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).
- *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*. URL: https://www.coder-welten.de/einstieg/variablen-in-c-3.html (besucht am 11. 08. 2022).
- *What is Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/ (besucht am 18. 07. 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).
- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand*. 1. Aufl. Wiley, 20. Nov. 2012.

Artikel

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

Vorlesungen

- Bast, Prof. Dr. 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, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03. 08. 2022).
- Scholl, Prof. Dr. 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, Prof. Dr. Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19. 07. 2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23. 07. 2022).

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).
- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03. 08. 2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: https://github.com/tueda/makefile4latex (besucht am 03. 08. 2022).