# Albert Ludwigs Universität Freiburg

#### TECHNISCHE FAKULTÄT

## PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 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

Bevor der Inhalt dieser Schrifftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholll im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, bin ich keine Person, die irgendwelche Dinge gerne so macht wie es üblich ist, ich schreibe meine Danksagung nicht auf eine bestimmte Weise, nur weil sich irgendwann mal etabliert hat wie eine Danksagung üblicherweise aussieht. Ich halte nicht viel von künstlichen Floskeln, wie "mein aufrichtigster Dank" oder "aus tiefstem Herzen", sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Ich halte es eher so, dass wenn mir wirklich etwas am Dank gegenüber Personen liegt, ich mir wirklich den Aufwand mache einen Text zu schreiben in dem ich diesen zum Ausdruck bringe, im anderen Fall kann man sich bei mir auf die typischen Standardfloskeln einstellen. Bei dieser Bachelorarbeit kann ich nur auf ersteres Zurückgreifen. Ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und sehr respektvoll, was ich als keine Selbverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das heraus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tuen, wie es die Anforderungen verlangen und nichts darüberhinaus tuen, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschafft heraus tuen, auch wenn es für sie keine Vorteile hat. Tobias konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe<sup>1</sup>, er hat sich bei der Korrektur dieser Schrifftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere Textkommentare zu verfassen und das trotz dessen, dass meine Bachelorarbeit recht Umfangreich zu lesen ist<sup>2</sup> und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinen Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Ich hab während meines Bachelorprojekts und meiner Bachelorarbeit wahrscheinlich einen ziemlich eigensinnigen Eindruck gemacht, bei der Weise, wie ich bestimmte Dinge umsetzen wollte. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

<sup>&</sup>lt;sup>1</sup>Wofür ich mich auch nochmal Entschuldigen will.

<sup>&</sup>lt;sup>2</sup>Wobei er sich kein einziges Mal in geringster Weise entnervt darüber gezeigt hat.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen, für dessen Implementierung Michel Giehl sich netterweise zur Verfügung gestellt hat. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler ausgeinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter Link<sup>3</sup> zu finden.

Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat, was man auch selten im Studium erlebt, dass dem Studenten freiwillig weniger Arbeit gegeben wird. Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse ziemlich viel unerwartete Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl nichts geworden. Man hat daran gemerkt, dass Prof. Dr. Scholl das Wohlergehen der Studenten wichtig ist.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht<sup>4</sup>. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist, die sonst ziemlich wilkürlich erscheinen würde, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser großartigen Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

<sup>&</sup>lt;sup>3</sup>https://github.com/michel-giehl/Reti-Emulator.

<sup>&</sup>lt;sup>4</sup>Da es recht stressig ist im Kopf zu behalten, was man schon erklärt hat und wo noch eine Verständnislücke vorliegen könnte.

# Inhaltsverzeichnis

Abbildungsverzeichnis	Ι
Codeverzeichnis	Π
Tabellenverzeichnis	Π
Definitionsverzeichnis	$\mathbf{V}$
Grammatikverzeichnis	VΙ
1.3 Lexikalische Analyse 1.4 Syntaktische Analyse 1.5 Code Generierung 1.5.1 Monadische Normalform 1.5.2 A-Normalform 1.5.3 Ausgabe des Maschinencodes	1 3 5 9 12 13 16 23 24 25 27 28
Appendix	$\mathbf{A}$
Literatur	K

# Abbildungsverzeichnis

1.1	Horinzontale Übersetzungszwischenschritte zusammenfassen.
1.2	Vertikale Interpretierungszwischenschritte zusammenfassen.
1.3	Veranschaulichung von Linksassoziativität und Rechtsassoziativität
1.4	Veranschaulichung von Präzidenz
1.5	Veranschaulichung der Lexikalischen Analyse
1.6	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.
1.7	Veranschaulichung der Syntaktischen Analyse
1.8	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten
1.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen
2.1	Datenpfade der RETI-Architektur
2.2	Cross-Compiler als Bootstrap Compiler
2.3	Iteratives Bootstrapping.

# Codeverzeichnis

# **Tabellenverzeichnis**

2.1	Load und Store Befehle	P
2.2	Compute Befehle	A
2.3	Jump Befehle	I

# Definitionsverzeichnis

1.1	Interpreter
1.2	Compiler
1.3	Maschinensprache
1.4	Immediate
1.5	Cross-Compiler
1.6	T-Diagram Programm
1.7	T-Diagram Übersetzer (bzw. eng. Translator)
1.8	T-Diagram Interpreter
1.9	T-Diagram Maschine
1.10	Symbol
	Alphabet
1.12	Wort
	Formale Sprache
	Syntax
	Semantik
1.16	Formale Grammatik
	Chromsky Hierarchie
	Reguläre Grammatik
	Kontextfreie Grammatik
	Wortproblem
	1-Schritt-Ableitungsrelation
	Ableitungsrelation
1.23	Links- und Rechtsableitungableitung
	Linksrekursive Grammatiken
	Formaler Ableitungsbaum
1.26	Mehrdeutige Grammatik
1.27	Assoziativität
1.28	Präzidenz
1.29	Pipe-Filter Architekturpattern
1.30	Pattern
1.31	Lexeme
	Lexer (bzw. Scanner oder auch Tokenizer)
	Bezeichner (bzw. Identifier)
1.34	Literal
1.35	Konkrette Syntax
	Konkrette Grammatik
1.37	Ableitungsbaum (bzw. Konkretter Syntaxbaum oder auch engl. Derivation Tree) 1
	Parser
1.39	Recognizer (bzw. Erkenner)
1.40	Transformer
	Visitor
	Abstrakte Syntax
	Abstrakte Grammatik
	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)
	Pass
1.46	Reiner Ausdruck (bzw. engl. pure expression)
1 4 7	

1.48	Monadische Normalform (bzw. engl. monadic normal form)
1.49	Location
1.50	Atomarer Ausdruck
1.51	Komplexer Ausdruck
1.52	A-Normalform (ANF)
1.53	Fehlermeldung
2.1	Assemblersprache (bzw. engl. Assembly Language)
2.2	Assembler
2.3	Objectcode
2.4	Linker
2.5	Transpiler (bzw. Source-to-source Compiler)
2.6	Rekursiver Abstieg
2.7	$\operatorname{LL}(k)\text{-}\operatorname{Grammatik}  .  .  .  .  .  .  .  D$
2.8	Earley Recognizer
2.9	Liveness Analyse
2.10	Live Variable
2.11	Graph Coloring
2.12	Interference Graph
2.13	Kontrollflussgraph
2.14	Kontrollfluss
2.15	Kontrollflussanalyse
2.16	Two-Space Copying Collector
2.17	Self-compiling Compiler
2.18	Minimaler Compiler
2.19	Boostrap Compiler
2.20	Bootstrapping

# Grammatikverzeichnis

1.1	Produktionen für	: Ableitungsbaum	in EBNF																							11
-----	------------------	------------------	---------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

# 1 Einführung

#### 1.1 Compiler und Interpreter

Die wohl wichtigsten zu klärenden Begriffe, sind die eines Compilers (Definition 1.2) und eines Interpreters (Definition 1.1), da das Schreiben eines Compilers von der PicoC-Sprache  $L_{PicoC}$  in die RETI-Sprache  $L_{RETI}$  das Thema dieser Bachelorarbeit ist und die Definition eines Interpreters genutzt wird, um zu definieren was ein Compiler ist. Des Weiteren wurde zur Qualitätsicherung ein RETI-Interpreter implementiert, um mithilfe des GCC<sup>1</sup> und von Tests die Beziehungen in 1.2.1 zu belegen (siehe Subkapitel ??).

#### Definition 1.1: Interpreter

1

Interpretiert die Befehle<sup>a</sup> oder Statements eines Programmes P direkt.

Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen Sub-Bäumen des Abstrakten Syntaxbaumes (wird später eingeführt unter Definition 1.44) und führt je nach Komposition der Knoten des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.<sup>b</sup>

<sup>a</sup>Maschinensprache kann genauso interpretiert werden, wie auch eine Programmiersprache.

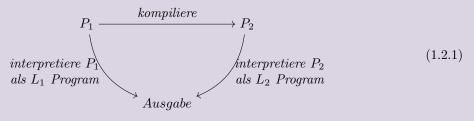
 ${}^{b}$ G. Siek, Course Webpage for Compilers (P423, P523, E313, and E513).

#### Definition 1.2: Compiler

Z

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, sie die gleiche Ausgabe haben, wie es in Diagramm 1.2.1 dargestellt ist. Also beide Programme  $P_1$  und  $P_2$  die gleiche Semantik (Definition 1.15) haben und sich nur syntaktisch (Definition 1.14) durch die Sprachen  $L_1$  und  $L_2$ , in denen sie geschrieben stehen unterscheiden.



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

<sup>&</sup>lt;sup>1</sup>Sammlung von Compilern für Linux bzw. GNU-Linux, steht für GNU Compiler Collection

#### Anmerkung Q

Im Folgenden wird ein voll ausgeschriebener Compiler als  $C_{i\_w\_k\_min}^{o\_j}$  geschrieben, wobei  $C_w$  die Sprache bezeichnet, die der Compiler als Input nimmt und zu einer nicht näher spezifizierten Maschinensprache  $L_{B_i}$  einer Maschine  $M_i$  kompiliert. Falls die Notwendigkeit besteht, die Maschine  $M_i$  anzugeben, zu dessen Maschinensprache  $L_{B_i}$  der Compiler kompiliert, wird das als  $C_i$  geschrieben. Falls die Notwendigkeit besteht die Sprache  $L_o$  anzugeben, in der der Compiler selbst geschrieben ist, wird das als  $C^o$  geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert  $(L_{w\_k})$  oder in der er selbst geschrieben ist  $(L_{o\_j})$  anzugeben, wird das als  $C_{w\_k}^{o\_j}$  geschrieben. Falls es sich um einen minimalen Compiler handelt (Definition 2.18) kann man das als  $C_{min}$  schreiben.

Üblicherweise kompiliert ein Compiler ein Program, das in einer Programmiersprache geschrieben ist zu Maschinencode, der in Maschinensprache (Definition 1.3) geschrieben ist, aber es gibt z.B. auch Transpiler (Definition 2.5) oder Cross-Compiler (Definition 1.5). Des Weiteren sind Maschinensprache und Assemblersprache (Definition 2.1) voneinander zu unterscheiden.

#### Definition 1.3: Maschinensprache

Programmiersprache, deren mögliche Programme die hardwarenaheste Repräsentation eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten Aufgabe, die die CPU im vereinfachten Fall in einem Zyklus der Fetch- und Execute-Phase, genauergesagt in der Execute-Phase übernehmen kann oder allgemein in einer geringen konstanten Anzahl von Fetch- und Execute Phasen im Komplexeren Fall. Die Maschinenbefehle sind meist so entworfen, dass sie sich innerhalb bestimmter Wortbreiten, die Zweierpotenzen sind kodieren lassen. Im einfachsten Fall innerhalb einer Speicherzelle des Hauptspeichers.

<sup>a</sup>Viele Prozessorarchitekturen erlauben es allerdings auch z.B. zwei Maschinenbefehle in eine Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen Immediates (Definition 1.4) haben.

<sup>b</sup>P. D. C. Scholl, "Betriebssysteme".

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

Der PicoC-Compiler, der den Zweck erfüllt für Studenten ein Anschauungs- und Lernwerkzeug zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in menschenlesbarer Form mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 1.4) enthält. Für den RETI-Interpreter ist es ebenfalls nicht notwendig, dass der Maschinencode, den der PicoC-Compiler generiert, in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU simulieren soll und nicht deren mögliche interne Umsetzung<sup>2</sup>.

#### Definition 1.4: Immediate

7

Konstanter Wert, der als Teil eines Maschinenbefehls gespeichert ist und dessen Wertebereich dementsprechend auch durch die Anzahl an Bits, die ihm innerhalb dieses Maschinenbefehls zur Verfügung gestellt sind beschränkt ist. Der Wertebereich ist beschränkter als bei sonstigen Werten

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

innerhalb des Hauptspeichers, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.<sup>a</sup>

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

#### Definition 1.5: Cross-Compiler

Kompiliert auf einer Maschine  $M_1$  ein Program, dass in einer Sprache  $L_w$  geschrieben ist für eine andere Maschine  $M_2$ , wobei beide Maschinen  $M_1$  und  $M_2$  unterschiedliche Maschinensprachen  $B_1$  und  $B_2$  haben. <sup>ab</sup>

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

<sup>b</sup>J. 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 Maschinensprache  $B_2$  einer Zielmaschine  $M_2$  läuft.

#### 1.1.1 T-Diagramme

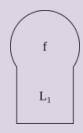
Um die Architektur von Compilern und Interpretern übersichtlich darzustellen eignen sich T-Diagramme, deren Spezifikation aus der Wissenschaftlichen Publikation J. Earley und Sturgis, "A formalism for translator interactions" entnommen ist besonders gut, da diese optimal darauf zugeschnitten sind die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

Die Notation setzt sich dabei aus den Blöcken für ein Program (Definition 1.6), einen Übersetzer (Definition 1.7), einen Interpreter (Definition 1.8) und eine Maschine (Definition 1.9) zusammen.

#### Definition 1.6: T-Diagram Programm



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



<sup>a</sup>J. Earley und Sturgis, "A formalism for translator interactions".

#### Anmerkung Q

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

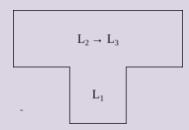
<sup>&</sup>lt;sup>3</sup>Die an vielen Universitäten und Schulen eingesetzen programmierbaren Roboter von Lego Mindstorms nutzen z.B. einen Cross-Compiler, um für den programmierbaren Microcontroller eine C-ähnliche Sprache in die Maschinensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst nicht genug Rechenleistung besitzt, um ein Programm selbst zeitnah zu kompilieren.

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

/

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

Für den Übersetzer gelten genauso, wie für einen Compiler<sup>a</sup> die Beziehungen in 1.2.1.<sup>b</sup>

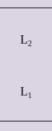


 $<sup>^</sup>a$ Zwischen den Begriffen Übersetzung und Kompilierung gibt es einen kleinen Unterschied, Übersetzung ist kleinschrittiger als Kompilierung und ist auch zwischen Passes möglich, Kompilierung beinhaltet dagegen bereits alle Passes in einem Schritt. Kompilieren ist also auch Übsersetzen, aber Übersetzen ist nicht immer auch Kompilieren.  $^b$ J. Earley und Sturgis, "A formalism for translator interactions".

#### Definition 1.8: T-Diagram Interpreter

Z

Repräsentiert einen Interpreter, der in der Sprache  $L_1$  geschrieben ist und Programme in der Sprache  $L_2$  interpretiert.



 $^a\mathrm{J}.$  Earley und Sturgis, "A formalism for translator interactions".

#### Definition 1.9: T-Diagram Maschine

Repräsentiert eine Maschine, welche ein Programm in Maschinensprache  $L_1$  ausführt. ab



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

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

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

 $<sup>^</sup>b\mathrm{J}.$  Earley und Sturgis, "A formalism for translator interactions".



Abbildung 1.1: Horinzontale Übersetzungszwischenschritte zusammenfassen.



Abbildung 1.2: Vertikale Interpretierungszwischenschritte zusammenfassen.

#### 1.2 Formale Sprachen

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



#### Definition 1.11: Alphabet

Z

"Ein Alphabet ist eine endliche, nicht-leere Menge aus Symbolen (Definition 1.10)."

<sup>a</sup>Nebel, "Theoretische Informatik".

#### Definition 1.12: Wort

Z

"Ein Wort  $w = a_1...a_n \in \Sigma^*$  ist eine endliche Folge von Symbolen aus einem Alphabet  $\Sigma$ .

Es gibt es die Konkatenation  $w_1w_2 = a_1 \dots a_n b_1 \dots b_n$  von Wörtern  $w_1 = a_1 \dots a_n$  und  $w_2 = b_1 \dots b_n$  und die Länge eines Wortes |w|.

Ein wichtiges Wort ist das leere Wort  $\varepsilon$  für das gilt:  $|\varepsilon|=0$  und  $\forall w \in \Sigma^*$ :  $\varepsilon w=w\varepsilon=w$ . Es handelt sich bei  $\varepsilon$  also um das Neutrale Element bei der Konkatenation von Wörtern.

Bei  $\Sigma^*$  handelt es sich um Kleenesche Hülle eines Alphabets  $\Sigma$ , es ist die Sprache aller Wörter, welche durch beliebige Konkatenation von Symbolen aus dem Alphabet  $\Sigma$  gebildet werden können, wobei  $\varepsilon \in \Sigma^*$ . Dies ist die größte Sprache über  $\Sigma$  und jede Sprache über  $\Sigma$  ist eine Teilmenge davon. "a

<sup>a</sup>Nebel, "Theoretische Informatik".

#### Definition 1.13: Formale Sprache



"Eine Formale Sprache ist eine Menge von Wörtern (Definition 1.12) über dem Alphabet  $\Sigma$  (Definition 1.11). "a

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

<sup>a</sup>Nebel, "Theoretische Informatik".

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

#### Definition 1.14: Syntax



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

<sup>a</sup>Z.B. die Konkrette und Abstrakte Syntax, die später eingeführt werden.

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

#### Definition 1.15: Semantik



Die Semantik bezeichnet alles was mit der Bedeutung von Formalen Sprachen zu tuen hat.<sup>a</sup>

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

#### Definition 1.16: Formale Grammatik

1

"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 = Nicht-Terminalsymbole.
- $\Sigma \triangleq Terminal symbole$ , wobei  $N \cap \Sigma = \emptyset^{bc}$ .
- $P \triangleq Menge\ von\ Produktionsregeln\ w \to v,\ wobei\ w,v \in (N \cup \Sigma)^* \land w \notin \Sigma^*.^{de}$ .
- S = Startsymbol, wobei  $S \in N$ .

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

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

#### Definition 1.17: Chromsky Hierarchie



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

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

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

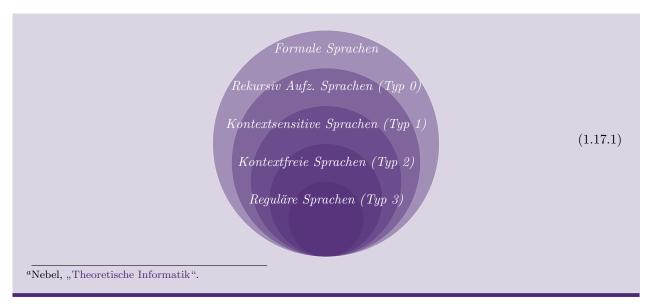
<sup>&</sup>lt;sup>a</sup>Nebel, "Theoretische Informatik".

<sup>&</sup>lt;sup>b</sup>Weil mit ihnen terminiert wird.

<sup>&</sup>lt;sup>c</sup>Kann auch als **Alphabet** bezeichnet werden.

<sup>&</sup>lt;sup>d</sup>w muss mindestens ein Nicht-Terminalsymbol enthalten.

<sup>&</sup>lt;sup>e</sup>Bzw.  $w, v \in V^* \land w \notin \Sigma^*$ .



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

#### Definition 1.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon$$
 (1.18.1)

haben, wobei A, B Nicht-Terminalsymbole sind und c ein Terminalsymbol ist<sup>ab</sup>."<sup>c</sup>

#### Definition 1.19: Kontextfreie Grammatik

1

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

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

haben,  $wobei\ A\ ein\ Nicht-Terminal symbol\ ist\ und\ v\ ein\ beliebige\ Folge\ von\ Grammatik symbolen^a$  ist."

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

<sup>&</sup>lt;sup>a</sup>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.

 $<sup>^</sup>b$ Dadurch, dass die linke Seite immer nur ein Nicht-Terminalsymbol sein darf ist jede Reguläre Grammatik auch eine Kontextfrei Grammatik.

 $<sup>^</sup>c$ Nebel, "Theoretische Informatik".

<sup>&</sup>lt;sup>a</sup>Also eine beliebige Folge von Nicht-Terminalsymbolen und Terminalsymbolen.

 $<sup>^</sup>b\mathrm{Nebel},$  "Theoretische Informatik".

#### Definition 1.20: Wortproblem

Z

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  $w \notin L$ .

Das Wortproblem kann durch die folgende Indikatorfunktion<sup>c</sup> zusammengefasst werden:

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

<sup>a</sup>Bzw. "ja" oder "nein" usw., es muss nicht umgedingt 1 oder 0 sein.

#### 1.2.1 Ableitungen

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

#### Definition 1.21: 1-Schritt-Ableitungsrelation



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

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

<sup>a</sup>Nebel, "Theoretische Informatik".

#### Definition 1.22: Ableitungsrelation



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

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

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

#### Definition 1.23: Links- und Rechtsableitungableitung



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

 $<sup>^</sup>b$ Nebel, "Theoretische Informatik".

<sup>&</sup>lt;sup>c</sup>Auch Charakteristische Funktion genannt.

<sup>&</sup>lt;sup>a</sup>Nebel, "Theoretische Informatik".

<sup>&</sup>lt;sup>4</sup>Bzw. Wort.

<sup>&</sup>lt;sup>5</sup>Bzw. Wort.

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 **Ansätz**e für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 1.24) ist<sup>6</sup>.

#### Definition 1.24: Linksrekursive Grammatiken

**I** 

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

 ${}^aParsing\ Expressions\ \cdot\ Crafting\ Interpreters.$ 

Um herauszufinden, ob eine Grammatik mehrdeutig (Definition 1.26) ist, werden Ableitungen als Formale Ableitungsbäume (Definition 1.25) dargestellt. Formale Ableitungsbäume werden im Unterkapitel 1.4 nochmal relevant, da in der Syntaktischen Analyse Ableitungsbäume (Definition 1.37) als eine compilerinterne Datenstruktur umgesetzt werden.

#### Definition 1.25: Formaler Ableitungsbaum



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

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

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

 $^a$ Z.B. Programmcode.

<sup>b</sup>Nebel, "Theoretische Informatik".

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

<sup>&</sup>lt;sup>6</sup>Für den im PicoC-Compiler verwendeten Earley Parsers stellt dies allerdings kein Problem dar.

$\overline{DIG\_NO\_0}$	::=	"1"   "2"   "3"   "4"   "5"   "6"	$L_{-}Lex$
		"7"   "8"   "9"	
		"0"   DIG_NO_0 "0"   DIG_NO_0 DIG_WITH_0*	
add	::=	add "+" mul   mul	L_Parse
mul	::=	mul "*" $NUM$   $NUM$	

Grammatik 1.1: Produktionen für Ableitungsbaum in EBNF

#### Anmerkung Q

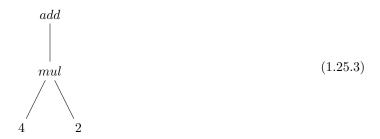
Werden die Produktionen einer Grammatik in z.B. EBNF angegeben, wie in Grammatik ??, wird die Angabe dieser Produktionen auch oft als Grammatik bezeichnet, obwohl Grammatiken eigentlich durch ein Tupel  $G = \langle N, \Sigma, P, S \rangle$  dargestellt sind.

$$add \Rightarrow mul \Rightarrow mul \ "*" \ NUM \Rightarrow NUM \ "*" \ NUM \Rightarrow 4 \ "*" \ NUM \Rightarrow 4 \ "*" \ 2$$
 (1.25.1)

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



Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen<sup>7</sup>.



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

<sup>&</sup>lt;sup>7</sup>Diese Option wurde beim **PicoC-Compiler** gewählt.

#### Definition 1.26: Mehrdeutige Grammatik

Z

"Eine Grammatik ist mehrdeutig, wenn es ein Wort  $w \in L(G)$  gibt, das mehrere Ableitungsbäume zulässt".  $^{ab}$ 

 $^a {\rm Alternativ},$ wenn es für wmehrere unterschiedliche Linksableitungen gibt.

#### 1.2.2 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 (Definition 1.27) und Präzidenz (Definition 1.28) dieser Operatoren festgelegt ist. Dieses Schema wird in Unterkapitel ?? genauer erklärt.

#### Definition 1.27: Assoziativität

Z

"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.<sup>a</sup>

 $^aParsing\ Expressions\ \cdot\ Crafting\ Interpreters.$ 

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

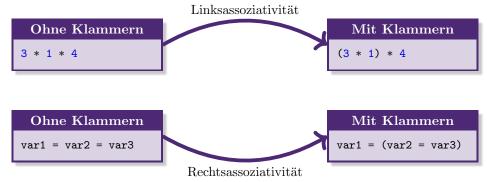


Abbildung 1.3: Veranschaulichung von Linksassoziativität und Rechtsassoziativität.

#### Definition 1.28: Präzidenz

7

"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."

<sup>a</sup>Parsing Expressions · Crafting Interpreters.

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

<sup>&</sup>lt;sup>b</sup>Nebel, "Theoretische Informatik".

Abbildung 1.4: Veranschaulichung von Präzidenz.

#### 1.3 Lexikalische Analyse

Die Lexikalische Analyse bildet üblicherweise den ersten Filter innerhalb des Pipe-Filter Architekturpatterns (Definition 1.29) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt in einem Eingabewort<sup>8</sup> endliche Folgen Symbolen<sup>9</sup> zu finden, die durch bestimmte Pattern (Definition 1.30) erkannt werden, die durch eine reguläre Grammatik spezifiziert sind. Diese Folgen endlicher Symoble werden auch Lexeme (Definition 1.31) genannt.

#### Definition 1.29: Pipe-Filter Architekturpattern

Z

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. ab



<sup>&</sup>lt;sup>a</sup>Das ein Bindeglied eine eigene Bezeichnung erhält, bedeutet allerdings nicht, dass es eine eigene wichtige Aufgabe erfüllt. Wie bei vielen Pattern, soll mit dem Namen des Pattern, in diesem Fall durch das Pipe die 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.

#### Definition 1.30: Pattern



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

#### Definition 1.31: Lexeme



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

<sup>&</sup>lt;sup>b</sup>Westphal, "Softwaretechnik".

 $<sup>^</sup>a\mathrm{Als}$  Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

 $<sup>{}^</sup>b{
m Thiemann},$  "Compilerbau".

<sup>&</sup>lt;sup>8</sup>Z.B. dem Inhalt einer Datei, welche in UTF-8 kodiert ist.

<sup>&</sup>lt;sup>9</sup>Also Teilwörter des Eingabeworts.

<sup>a</sup>Ein Lexeme könnte z.B. '0x63' sein, das entsprechende Token dazu wäre dann Token('NUM', '99'). Häufig stimmt das Lexeme aber auch mit dem Tokenwert überein, wie z.B. bei '99' und Token('NUM', '99').

<sup>b</sup>Thiemann, "Compilerbau".

Diese Lexeme werden vom Lexer (Definition 1.32) im Eingabewort identifziert und Tokens T zugeordnet. Das jeweils nächste Lexeme fängt dabei genau nach dem letzten Symbol des Lexemes an, das zuletzt vom Lexer erkannt wurde. Die Tokens (Definition 1.32) sind es, die letztendlich an die Syntaktische Analyse weitergegeben werden.

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

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

<sup>a</sup>Thiemann, "Compilerbau".

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

#### Anmerkung Q

Um Verwirrung verzubeugen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

In der Syntaktischen Analyse sind die Tokennamen die Symbole.

In der Code Generierung sind die Bezeichner (Definition 1.33) von Variablen, Konstanten und Funktionen die Symbole<sup>a</sup>.

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

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

 $\label{lem:constant} \textbf{Tokenwert}, \ der \ eine \ Konstante, \ Variable, \ Funktion \ usw. \ innerhalb \ ihres \ Sichtbarkeitsbereichs \ eindeutig \ benennt. \ ^{ab}$ 

<sup>a</sup>Außer wenn z.B. bei Funktionen die Programmiersprache das Überladen erlaubt usw. In diesem Fall wird die Signatur der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.

 ${}^b\mathrm{Thiemann},$  "Einführung in die Programmierung".

Eine weitere Aufgabe der Lekikalischen Analyse ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen  $_{-}$ , Newline  $\n^{10}$  und Tabs  $\t$  aus dem Eingabewort herauszufiltern. Das geschieht mittels des Lexers, der allen für die Syntaktische Analyse unwichtige Zeichen das leere Wort  $\epsilon$  zuordnet.

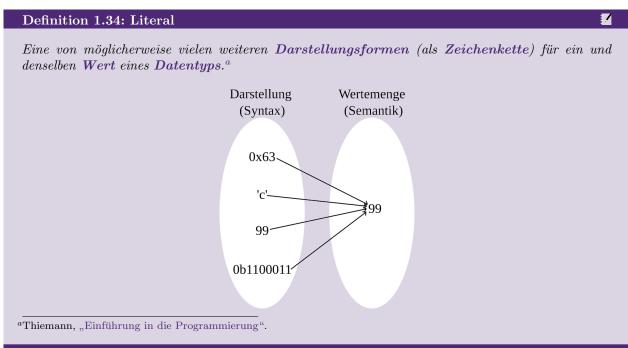
<sup>&</sup>lt;sup>10</sup>In Unix Systemen wird für Newline das ASCII Symbol line feed, in Windows hingegen die ASCII Symbole carriage return und line feed nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.

Das ist auch im Sinne der Definition, denn  $\epsilon \in (N \times W)^*$  ist immer der Fall bei der Kleeneschen Hülle  $\Sigma^*$ , wobei  $\Sigma = N \times W$ . Nur das, was für die Syntaktische Analyse wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die Lexeme an die Syntaktische Analyse weitergegeben werden und der Grund für die Aufteilung des Tokens in Tokenname und Tokenwert ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie my\_fun, my\_var oder my\_const und es auch viele verschiedenen Zahlen gibt, wie 42, 314 oder 12. Die Überbegriffe bzw. Tokennamen für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. NAME und NUM<sup>11</sup>, bzw. wenn man sich nicht Kurzformen sucht IDENTIFIER und NUMBER. Für Lexeme, wie if oder } sind die Tokennamen bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich IF und RBRACE.

Ein Lexeme ist damit aber nicht immer das gleiche, wie der Tokenwert, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene Literale (Definition 1.34) dargestellt werden, einmal als ASCII-Zeichen 'c', dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99<sup>12</sup>. 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 üblicherweise regulär, da ein typischer Lexer immer nur ein Symbol vorausschaut<sup>13</sup>, 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 1.18 erfüllt. Einzig die Produktion CHAR ::= "'"ASCII\_CHAR"'" sieht problematisch aus, kann allerdings auch als {CHAR ::= "'"CHAR2, CHAR2 ::= ASCII\_CHAR"'"} regulär ausgedrückt werden<sup>14</sup>. Somit existiert eine reguläre Grammatik, welche die Sprache  $L_{PicoC\_Lex}$  beschreibt und damit ist die Sprache  $L_{PicoC\_Lex}$  regulär.



<sup>&</sup>lt;sup>11</sup>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.

 $<sup>^{12}</sup>$ Die Programmiersprache Python erlaubt es z.B. dieser Wert auch mit den Literalen 0b1100011 und 0x63 darzustellen.

 $<sup>^{13}\</sup>mathrm{Man}$ nennt das auch einem Lookahead von 1

<sup>&</sup>lt;sup>14</sup>Eine derartige Regel würde nur Probleme bereiten, wenn sich aus ASCII\_CHAR beliebig breite Wörter ableiten liesen.

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

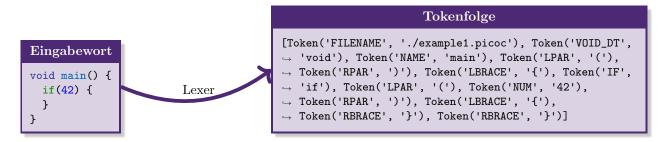


Abbildung 1.5: Veranschaulichung der Lexikalischen Analyse.

#### 1.4 Syntaktische Analyse

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

Die Syntax, in welcher ein Programm aufgeschrieben ist, wird auch als Konkrette Syntax (Definition 1.35) bezeichnet. In einem Zwischenschritt, dem Parsen wird aus diesem Programm mithilfe eines Parsers (Definition 1.38) ein Ableitungsbaum (Definition 1.37) generiert, der als Zwischenstufe hin zum einem Abstrakten Syntaxbaum (Definition 1.44) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des Ableitungsbaumes und dann erst des Abstrakten Syntaxbaumes.

#### Definition 1.35: Konkrette Syntax

Z

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.<sup>a</sup>

<sup>a</sup>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 1.36) bezeichnet.

#### Definition 1.36: Konkrette Grammatik



Grammatik, die eine Konkrette Syntax beschreibt.

## Definition 1.37: Ableitungsbaum (bzw. Konkretter Syntaxbaum oder auch engl. Derivation Tree)

Compilerinterne Datenstruktur für den Formalen Ableitungsbaum (Definition 1.25) 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.<sup>a</sup>

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

#### Definition 1.38: 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<sup>a</sup>.<sup>b</sup>

<sup>a</sup>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 1.38 verwendet.

 $^b JSON\ parser$  - Tutorial —  $Lark\ documentation$ .

#### Anmerkung Q

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 auschließlich für die Lexikalische Analyse verantwortlich und entspricht z.B., wenn man bei einem Wanderausflug verschiedenen Insekten entdeckt, dem Nachschlagen in einem Insektenlexikon und dem Aufschreiben, welchen Insekten man in welcher Reihenfolge begegnet ist. Zudem kann man bestimmte Sehenswürdigkeiten an denen man während des Ausflugs vorbeikommt ebenfalls festhalten, da es eine Rolle spielen kann in welchem örtlichen Kontext man den Insekten begegnet ist<sup>a</sup>.

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

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

 $^a\mathrm{Das}$ würde z.B. der Rolle eines Semikolon ; in der Sprache  $L_{PicoC}$ entsprechen.

Die vom Lexer im Eingabewort identifizierten Token werden in der Syntaktischen Analyse vom Parser als Wegweiser verwendet, da je nachdem, in welcher Reihenfolge die Token auftauchen, dies einer anderen Ableitung in der Grammatik  $G_{Parse}$  entspricht. Dabei wird in der Grammatik  $L_{Parse}$  nach dem Tokennamen unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine Zahl steht und nicht, welchen konkretten Wert diese Zahl hat. Der Tokenwert ist erst später in der Code Generierung in 1.5 wieder relevant.

Ein Parser ist genauergesagt ein erweiterter Recognizer (Definition 1.39), denn ein Parser löst das Wortproblem (Definition 1.20) 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

 $<sup>{}^{</sup>b}$ Z.B. gibt es bestimmte Wechselbeziehungen zwischen Insekten, Insekten beinflussen sich gegenseitig.

Recognition Algorithmus gesichert wurden den Ableitungsbaum.

#### Definition 1.39: 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 1.20) bekannt.<sup>a</sup>

#### Anmerkung Q

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.<sup>a</sup>

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 2.6).

Mit dieser Methode ist das Parsen Linksrekursiver Grammatiken (Definition 1.24) 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 2.7) 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.<sup>b</sup>

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.

- 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.<sup>d</sup>
- Chart Parsing: Es wird Dynamische Programmierung verwendet und partielle Zwischen-

<sup>&</sup>lt;sup>a</sup>Thiemann, "Compilerbau".

ergebnisse werden in einer Tabelle (bzw. einem Chart) gespeichert und können wiederverwendet werden. Das macht das Parsen Kontextfreier Grammatiken effizienter, sodass es nur noch polynomielle Zeit braucht, da Backtracking nicht mehr notwendig ist<sup>e</sup>. 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.

Der Abstrakte Syntaxbaum wird mithilfe von Transformern (Definition 1.40) und Visitors (Definition 1.41) 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 1.42).

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

#### Definition 1.41: 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.

#### Definition 1.42: 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<sup>a</sup> 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

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

<sup>&</sup>lt;sup>b</sup>Diese Form von Parsing wurde im PicoC-Compiler implementiert, als dieser noch auf dem Stand des Bachelorprojektes war, bevor er durch den nicht selbst implementierten Earley Parser von Lark (siehe Webseite Lark - a parsing toolkit for Python) ersetzt wurde.

<sup>&</sup>lt;sup>c</sup>Diese Art von Parser ist im RETI-Interpreter implementiert, da die RETI-Sprache eine besonders simple LL(1) Grammatik besitzt. Diese Art von Parser wird auch als Predictive Parser oder LL(k) Recursive Descent Parser bezeichnet, wobei Recursive Descent das englische Wort für Rekursiven Abstieg ist.

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

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

<sup>&</sup>lt;sup>a</sup> Transformers  $\ensuremath{\mathscr{C}}$  Visitors — Lark documentation.

<sup>&</sup>lt;sup>a</sup>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.

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

#### $Maschinenbefehlen.^b$

<sup>a</sup>Z.B. Programmcode.

<sup>b</sup>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 1.43) bezeichnet.

#### Definition 1.43: Abstrakte Grammatik

Z

Grammatik, die eine Abstrakte Syntax beschreibt.

#### Definition 1.44: 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<sup>a</sup> möglichst einfach weiterverarbeitet werden kann.

<sup>a</sup>Den verschiedenen Passes.

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



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

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

muss.

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

#### Abstrakter Syntaxbaum File Name './example1.ast', FunDef VoidType 'void', Tokenfolge Name 'main', [], [Token('FILENAME', './example1.picoc'), Token('VOID\_DT', → 'void'), Token('NAME', 'main'), Token('LPAR', '('), Ιf → Token('RPAR', ')'), Token('LBRACE', '{'), Token('IF', Num '42', $_{\hookrightarrow}$ 'if'), Token('LPAR', '('), Token('NUM', '42'), → Token('RPAR', ')'), Token('LBRACE', '{'), ] → Token('RBRACE', '}'), Token('RBRACE', '}')] ] Parser Visitors und Transformer Ableitungsbaum file ./example1.dt decls\_defs decl\_def fun\_def type\_spec prim\_dt void pntr\_deg name main fun\_params decl\_exec\_stmts exec\_part exec\_direct\_stmt if\_stmt logic\_or logic\_and eq\_exp rel\_exp arith\_or arith\_oplus arith\_and arith\_prec2 arith\_prec1 un\_exp post\_exp 42 prim\_exp exec\_part compound\_stmt

Abbildung 1.7: Veranschaulichung der Syntaktischen Analyse.

Kapitel 1. Einführung 1.5. Code Generierung

#### 1.5 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 1.45) 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 Q

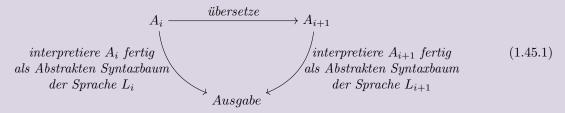
Man spricht hier von dem "Abstrakten Syntaxbaum einer Sprache  $L_1$  (bzw.  $L_2$ )" und meint hier mit der Sprache  $L_1$  (bzw.  $L_2$ ) nicht die Sprache, welche durch die Abstrakte Grammatik, nach welcher der Abstrakte Syntaxbaum abgeleitet ist beschrieben wird. Es ist damit immer die Sprache gemeint, 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 dem Buch G. Siek, Course Webpage for Compilers (P423, P523, E313, and E513) übernommen.

#### Definition 1.45: Pass

Z

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.

Für jeden Pass und für einen beliebigen Abstrakten Syntaxbaum  $A_i$  gilt ähnlich, wie bei einem vollständigen Compiler in 1.45.1, dass:



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

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

<sup>&</sup>lt;sup>a</sup>Ein Pass kann mit einem Transpiler 2.5 (Definition 2.5) 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 auschließlich auf Abstrakten Syntaxbäumen, ohne Parsing usw.

 $<sup>^</sup>b$ Der Begriff kommt aus dem Englischen von "passing over", da der gesamte Abstrakte Syntaxbaum in einem Pass durchlaufen wird.

<sup>&</sup>lt;sup>c</sup>Interpretieren geht immer von einem Programm in Konkretter Syntax aus, wobei der Abstrakte Syntaxbaum ein Zwischenschritt bei der Interpretierung ist.

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

Kapitel 1. Einführung 1.5. Code Generierung

#### 1.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche Unreine Ausdrücke (Definition 1.47) besitzt, so ist es sinnvoll einen Pass einzuführen, der Reine (Definition 1.46) 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 1.46: 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. <sup>ab</sup>

 $^a$ Sondern z.B. intern etwas am Kompilierprozess ändert.

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

#### Definition 1.47: Unreiner Ausdruck

Z

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

Auf diese Weise sind alle Statements und Ausdrücke in Monadischer Normalform (Definiton 1.48).

#### Definition 1.48: Monadische Normalform (bzw. engl. monadic normal form)

Z

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.<sup>a</sup>

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

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

Ein Beispiel für dieses Vorgehen ist in Abbildung 1.8 zu sehen, wo der Einfachheit halber auf die Darstellung in Abstrakter Syntax verzichtet wurde und die Codebeispiele in der entsprechenden Konkretten Syntax<sup>15</sup> aufgeschrieben wurden.

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

<sup>&</sup>lt;sup>15</sup>Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

Kapitel 1. Einführung 1.5. Code Generierung



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

Die Aufgabe eines solchen Passes ist es, den Abstrakten Syntaxbaum der Syntax von Maschinenbefehlen 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 Maschinenbefehlen ähnlicher als ein Ausdruck, indem ein Reiner und Unreiner Ausdruck gemischt sind. Somit sparrt man sich in der Implementierung Fallunterscheidungen, indem die Reinen Ausdrücke direkt in RETI-Code übersetzt werden können und nicht unterschieden werden muss, ob darin Unreine Ausdrücke vorkommen.

#### 1.5.2 A-Normalform

Im Falle dessen, dass es sich bei der Sprache  $L_1$  um eine höhere Programmiersprache und bei  $L_2$  um Maschinensprache handelt, ist es fast unerlässlich einen Pass einzuführen, der Komplexe Ausdrücke (Definition 1.51) 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 1.49) 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 1.50) 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 1.49: Location

Z

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.<sup>a</sup>

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

Auf diese Weise sind alle Statements und Ausdrücke in A-Normalform (Definition 1.52). Wenn eine Konkrette Grammatik in A-Normalform ist, ist diese auch automatisch in Monadischer Normalform (Definition 1.52), genauso, wie ein Atomarer Ausdruck auch ein Reiner Ausdruck ist (nach Definition 1.50).

Kapitel 1. Einführung 1.5. Code Generierung

#### Definition 1.50: 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).

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

#### Definition 1.51: Komplexer Ausdruck

Z

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

<sup>a</sup>int var ist eine Allokation.

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

#### Definition 1.52: A-Normalform (ANF)

Z

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. ab c

<sup>a</sup>A-Normalization: Why and How (with code).

<sup>b</sup>Bolingbroke und Peyton Jones, "Types are calling conventions".

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

Ein Beispiel für dieses Vorgehen ist in Abbildung 1.9 zu sehen, wo der Einfachheit halber auf die Darstellung in Abstrakter Syntax verzichtet wurde und die Codebeispiele in der entsprechenden Konkretten Syntax<sup>16</sup> aufgeschrieben wurden.

Der PicoC-Compiler nutzt, anders als es geläufig ist keine Register und Graph Coloring (Definition 2.11) inklusive Liveness Analysis (Definition 2.9) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den Hauptspeicher, wobei temporäre Zwischenergebnisse auf den Stack gespeichert werden.<sup>17</sup>

Aus diesem Grund verwendet das Beispiel in Abbildung 1.9 eine andere Definition für Komplexe und Atomare Ausdrücke, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im PicoC-ANF Pass der 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 Maschinen-

<sup>&</sup>lt;sup>16</sup>Für deren Kompilierung die Abstrakte Syntax überhaupt definiert wurde.

<sup>&</sup>lt;sup>17</sup>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.

Kapitel 1. Einführung 1.5. Code Generierung

befehlssatz es erlaubt zwei Register miteinander zu verechnen  $^{18}$ , 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  $^{19}$ . Werden allerdings keine Register für Zwischenergebnisse genutzt werden, braucht man mehrere Maschinenbefehle, um die Zwischenergebnisse vom Stack zu holen, zu verrechnen und das Ergebnis wiederum auf den Stack zu speichern und das SP-Register anzupassen. Daher werden die Ausdrücke für Zahl 4, Variable var und ASCII-Zeichen 'c' als Komplexe Ausdrücke gewertet, da sie niemals in einem Maschinenbefehl miteinander verechnet werden können.

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

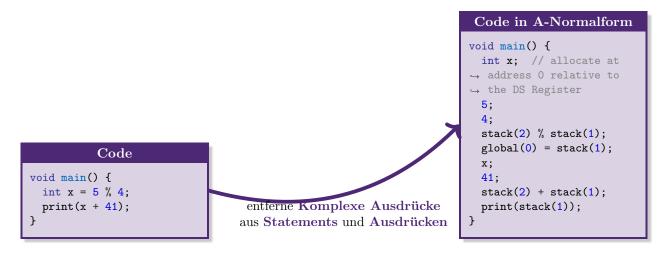


Abbildung 1.9: Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.

Ein solcher Pass hat vor allem in erster Linie die Aufgabe den 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.

#### 1.5.3 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 Maschinencode 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.

<sup>&</sup>lt;sup>18</sup>Z.B. Addieren oder Subtraktion von zwei Registerinhalten.

<sup>&</sup>lt;sup>19</sup>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.

# 1.6 Fehlermeldungen

Wenn bei einem Compiler ein unerwünschtes Verhalten der folgenden Kategorien<sup>20</sup> eintritt:

- 1. der Parser<sup>21</sup> entscheidet das Wortproblem für ein Eingabeprogramm<sup>22</sup> mit 0, also das Eingabeprogramm befolgt nicht die Syntax der Sprache des Compilers<sup>23</sup>.
- 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 2.4) etwas nicht zusammenpasst, wie z.B.:

- es gibt keine oder mehr als eine main-Funktion.
- eine Funktion, die in einer Objektdatei (Definition 2.3) benötigt wird, wird von keiner anderen oder mehr als einer Objektdatei bereitsgestellt.

wird eine Fehlermeldung (Definition 1.53) ausgegeben.

#### Definition 1.53: Fehlermeldung



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

- über stdout oder stderr im einem Terminal Emulator oder richtigen Terminal<sup>b</sup>.
- ullet über eine Dialogbox in einer Graphischen Benutzerfläche^c oder Zeichenorientierten Benutzerschnittstelle^d.

 $<sup>^{20}</sup>Errors\ in\ C/C++$  - Geeks for Geeks.

<sup>&</sup>lt;sup>21</sup>Bzw. der **Recognizer** im Parser.

 $<sup>^{22}</sup>$ Bzw. Wort.

 $<sup>^{23}</sup>$ 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 geschrie-
- Logdatei<sup>e</sup> auf einem Speichermedium.

 $<sup>^</sup>a$ Dieses Programm kann z.B. ein Compiler sein oder ein Programm, dass dieser Compiler selbst kompiliert hat.

 $<sup>{}^</sup>b$ Nur unter Linux, Windows hat sowas nicht.

<sup>&</sup>lt;sup>c</sup>In engl. Graphical User Interface, kurz GUI.

 $<sup>^</sup>d\mathrm{In}$ engl. Text-based User Interface, kurz TUI.  $^e\mathrm{In}$ engl. log file.

# Appendix

# **RETI Architektur Details**

Typ	$\mathbf{Modus}$	Befehl	Wirkung
01	00	LOAD D i	$D := M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
01	01	LOADIN S D i	$D := M(\langle S \rangle + i), \langle PC \rangle := \langle PC \rangle + 1$
01	11	LOADI D i	$D := 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1, \text{ bei } D = PC \text{ wird der PC}$
			nicht inkrementiert
10	00	STORE S i	$M(\langle i \rangle) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	01	STOREIN D S i	$M(\langle D \rangle + i) := S, \langle PC \rangle := \langle PC \rangle + 1$
10	11	MOVE S D	$D := S, \langle PC \rangle := \langle PC \rangle + 1$ , Move: Bei $D = PC$ wird der
			PC nicht inkrementiert

Tabelle 2.1: Load und Store Befehle.

Typ	M	RO	$\mathbf{F}$	$\operatorname{Befehl}$	Wirkung
00	0	0	000	ADDI D i	$[D] := [D] + [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	001	SUBI D i	$[D] := [D] - [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	010	MULI D i	$[D] := [D] * [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	011	DIVI D i	$[D] := [D] / [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	100	MODI D i	$[D] := [D] \% [i], \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	OPLUSI D i	$[D] := [D] \oplus 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	110	ORI D i	$[D] := [D] \vee 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	0	0	101	ANDI D i	$[D] := [D] \wedge 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	000	ADD D i	$[D] := [D] + [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	001	SUB D i	$[D] := [D] - [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	010	MUL D i	$[D] := [D] * [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	011	DIV D i	$[D] := [D] / [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	100	MOD D i	$[D] := [D] \% [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	OPLUS D i	$D := D \oplus M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	110	OR D i	$D := D \vee M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	1	0	101	AND D i	$D := D \wedge M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	000	ADD D S	$[D] := [D] + [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	001	SUB D S	$[D] := [D] - [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	010	MUL D S	$[D] := [D] * [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	011	DIV D S	$[D] := [D] / [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	100	MOD D S	$[D] := [D] \% [S], \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	OPLUS D S	$D := D \oplus S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	110	OR D S	$D := D \lor S, \langle PC \rangle := \langle PC \rangle + 1$
00	*	1	101	AND D S	$D := D \land S, \langle PC \rangle := \langle PC \rangle + 1$

Tabelle 2.2: Compute Befehle.

Type	Condition	J	Befehl	Wirkung
11	000	00	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11	001	00	$\mathrm{JUMP}_{>}\mathrm{i}$	Falls $[ACC] > 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	010	00	$JUMP_{=}i$	Falls $[ACC] = 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	011	00	$JUMP_{\geq}i$	Falls $[ACC] \ge 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	100	00	$\mathrm{JUMP}_{<}\mathrm{i}$	Falls $[ACC] < 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	101	00	$\mathrm{JUMP}_{ eq}\mathrm{i}$	Falls $[ACC] \neq 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1$
11	110	00	$JUMP \le i$	Falls $[ACC] \le 0$ : $\langle PC \rangle := \langle PC \rangle + [i]$ , sonst: $\langle PC \rangle := \langle PC \rangle + 1 \langle PC \rangle := \langle PC \rangle + [i]$
11	111	00	JUMPi	$\langle PC \rangle := \langle PC \rangle + [i]$
11	*	01	INT i	$\langle PC \rangle := IVT[i]$ Interrupt Nr.i wird Ausgeführt
11	*	10	RTI	Rücksprungadresse vom Stack entfernt, in PC geladen, Wechsel in Usermodus

Tabelle 2.3: Jump Befehle.

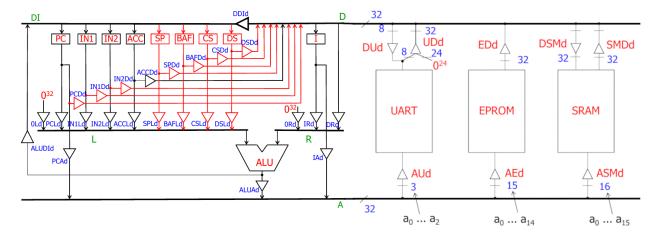


Abbildung 2.1: Datenpfade der RETI-Architektur.

# Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die zur Erklärung der Vorgehensweise zur Implementierung eines üblichen Compilers referenziert werden, aber nichts mit dem Vorgehen zur Implementierung des PicoC-Compilers zu tuen haben.

#### Definition 2.1: Assemblersprache (bzw. engl. Assembly Language)

Eine sehr hardwarenahe Programmiersprache, deren Befehle eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen haben. Viele Befehle haben eine ähnliche übliche Struktur Operation <Operanden>, mit einer Operation, die einem Opcode eines Maschinenbefehls bezeichnet und keinen oder mehreren Operanden, wie die späteren Maschinenbefehle, denen sie entsprechen. Allerdings gibt es oftmals noch viel "syntaktischen Zucker" innerhalb der Befehle und

 $drumherum^c$ . d

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

 ${}^{b}$ Z.B. erlaubt die Assemblersprache des GCC für die  $X_{86\_64}$ -Architektur für manche Operanden die Syntax  $\mathbf{n}(%\mathbf{r})$ , die einen Speicherzugriff mit Offset n zur Adresse, die im Register  $%\mathbf{r}$  steht durchführt, wobei z.B. die Klammern () usw. nur "syntaktischer Zucker" sind und natürlich nicht mitkodiert werden.

 $^{c}$ Z.B. sind im  $X_{86.64}$  Assembler die Befehle in Blöcken untergebracht, die ein Label haben und zu denen mittels jmp <label> gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.

<sup>d</sup>P. D. P. Scholl, "Einführung in Embedded Systems".

## Anmerkung Q

Ein Assembler (Definition 2.2) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt Maschinencode bzw. Objectcode (Definition 2.3) erzeugen. Ein Compiler soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur die Ausgabe liefern, welche er in den allermeisten Fällen haben will, nämlich den Maschinencode bzw. Objectcode, der direkt ausführbar ist bzw. wenn er später mit dem Linker (Definition 2.4) zu Maschienencode zusammengesetzt wird ausführbar ist.

#### Definition 2.2: Assembler

**7** 

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

<sup>a</sup>P. D. P. Scholl, "Einführung in Embedded Systems".

#### Definition 2.3: Objectcode



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

<sup>a</sup>P. D. P. Scholl, "Einführung in Embedded Systems".

#### Definition 2.4: Linker

**Z** 

Programm, dass Objektcode aus mehreren Objektdateien zu ausführbarem Maschinencode in eine ausführbare Datei oder Bibliotheksdatei linkt bzw. zusammenfügt, sodass unter anderem kein vermeidbarer doppelter Code darin vorkommt.<sup>a</sup>

<sup>a</sup>P. D. P. Scholl, "Einführung in Embedded Systems".

#### Definition 2.5: Transpiler (bzw. Source-to-source Compiler)



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

<sup>a</sup>Die Programmiersprache TypeScript will als Obermenge von JavaScript die Sprachhe Javascript erweitern und gleichzeitig die syntaktischen Mittel von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu transpilieren.

<sup>b</sup>Thiemann, "Compilerbau".

#### Definition 2.6: Rekursiver Abstieg

Z

Es wird jedem Nicht-Terminalsymbol eine Prozedur zugeordnet, welche die Produktionen dieses Nicht-Terminalsymbols umsetzt. Prozeduren rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes Nicht-Terminalsymbol enthält.

#### Anmerkung Q

Bei manchen Ansätzen für das Parsen eines Programmes, ist es notwendig eine LL(k)-Grammatik (Definition 2.7) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des Rekursiven Abstiegs (Definition 2.6) verwenden lässt sich eine bessere minimale Laufzeit garantieren, da aufgrund der LL(k)-Eigenschafft ausgeschlossen werden kann, dass Backtracking notwendig ist<sup>a</sup>.

 $^a\mathrm{Mehr}$  Erklärung hierzu findet sich im Unterkapitel 1.4.

#### Definition 2.7: LL(k)-Grammatik

Z

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

<sup>c</sup>Nebel, "Theoretische Informatik".

#### Definition 2.8: Earley Recognizer

Z

Ist ein Recognizer, der für alle Kontextfreien Sprachen das Wortproblem entscheiden kann und dies mittels Dynamischer Programmierung mit dem Top-Down Ansatz umsetzt. a b c

Eingabe und Ausgabe des Algorithmus sind:

- Eingabe: Eingabewort w und Konkrette Grammatik  $G_{Parse} = \langle N, \Sigma, P, S \rangle$ .
- Ausgabe: 0 wenn  $w \notin L(G_{Parse})^d$  und 1 wenn  $w \in L(G_{Parse})$ .

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

- $\alpha$ ,  $\beta$ ,  $\gamma$  stellen eine beliebige Folge von Grammatiksymbolen<sup>e</sup> dar.
- A und B stellen Nicht-Terminalsymbole dar.
- a stellt ein Terminalsymbol dar.
- Earley's Punktnotation:  $A := \alpha \bullet \beta$  stellt eine Produktion, in der  $\alpha$  bereits geparst wurde und  $\beta$  noch geparst werden muss.
- Die Indexierung ist informell ausgedrückt so umgesetzt, dass die Indices zwischen Tokennamen liegen, also Index 0 vor dem ersten Tokennamen verortet ist, Index 1 nach dem ersten Tokennamen verortet ist und Index n nach dem letzten Tokennamen verortet ist.

und davor müssen noch einige Begriffe definiert werden:

 $<sup>^</sup>a$ Das wird auch als **Lookahead** von k bezeichnet.

 $<sup>^</sup>b$ Wobei sich das mit den Linksableitungen automatisch ergibt, wenn man das Eingabewort von links-nach-rechts parsed und jeder der nächsten k Ableitungsschritte eindeutig sein soll.

- Zustandsmenge: Für jeden der n + 1 Indices j wird eine Zustandsmenge Z(j) generiert.
- Zustand einer Zustandsmenge: Ist ein Tupel (A ::= α β, i), wobei A ::= α β die aktuelle Produktion ist, die bis Punkt • geparst wurde und i der Index ist, ab welchem der Versuch der Erkennung eines Teilworts des Eingabeworts mithilfe dieser Produktion begann.

Der Ablauf des Algorithmus ist wie folgt:

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

bis:

- der Zustand  $(A := \beta \bullet, 0)$  in der Zustandsmenge Z(n) auftaucht, wobei A das Startsymbol S ist  $\Rightarrow w \in L(G_{Parse})$ .
- keine Zustände mehr hinzugefügt werden können  $\Rightarrow w \notin L(G_{Parse})$ .

#### Definition 2.9: Liveness Analyse

1

Findet heraus, welche Variablen in welchen Regionen eines Programmes verwendet werden. a

#### Definition 2.10: Live Variable

1

Eine Location, deren momentaner Wert später im Programmablauf noch verwendet wird. Man sagt auch die Location ist live. ab

<sup>&</sup>lt;sup>a</sup>Jay Earley, "An efficient context-free parsing".

 $<sup>{}^</sup>b\mathbf{Erkl\"{a}rweise}$ wurde von der Webseite  $\mathit{Earley\ parser}$ übernommen.

<sup>&</sup>lt;sup>c</sup>Earley Parser.

 $<sup>^{</sup>d}L(G_{Parse})$  ist die Sprache, welche durch die Konkrette Grammatik  $G_{Parse}$  beschrieben wird.

 $<sup>^</sup>e {\rm Also}$ eine Folge von Terminalsymbolen und Nicht-Terminalsymbolen.

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

 $<sup>^</sup>a\mathrm{Es}$  gibt leider kein allgemein verwendetes deutsches Wort für Live Variable.

<sup>&</sup>lt;sup>b</sup>G. Siek, Course Webpage for Compilers (P423, P523, E313, and E513).

#### Definition 2.11: Graph Coloring

Z

Problem bei dem den Knoten eines Graphen<sup>a</sup> Zahlen<sup>b</sup> zugewiesen werden sollen, sodass keine zwei adjazente Knoten die gleiche Zahl haben und möglichst wenige unterschiedliche Zahlen gebraucht werden.<sup>cd</sup>

<sup>a</sup>In Bezug zu Compilerbau ein Ungerichteter Graph.

 $^b$ Bzw. Farben.

<sup>c</sup>Es gibt leider kein allgemein verwendetes deutsches Wort für Graph Coloring.

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

#### Definition 2.12: Interference Graph

1

Ein ungerichteter Graph mit Locations als Knoten, der eine Kante zwischen zwei Locations hat, wenn es sich bei beiden Locations zu dem Zeitpunkt um Live Locations handelt. In Bezug auf Graph Coloring bedeutet eine Kante, dass diese zwei Locations nicht die gleiche Zahl<sup>a</sup> zugewiesen bekommen dürfen.<sup>b</sup>

<sup>a</sup>Bzw. Farbe.

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

#### Definition 2.13: Kontrollflussgraph



Gerichteter Graph, der den Kontrollfluss eines Programmes beschreibt.<sup>a</sup>

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

#### Definition 2.14: Kontrollfluss



Die Reihenfolge in der z.B. Statements, Funktionsaufrufe usw. eines Programmes ausgewertet werden<sup>a</sup>.

<sup>a</sup>Man geht hier von einem **imperativen** Programm aus.

#### Definition 2.15: Kontrollflussanalyse



Analyse des Kontrollflusses (Defintion 2.14) eines Programmes, um herauszufinden zwischen welchen Teilen des Programms Daten ausgetauscht werden und welche Abhängigkeiten sich daraus ergeben.

Der simpelste Ansatz ist es in einen Kontrollflussgraph iterativ einen Algorithmus^a anzuwenden, bis sich an den Werten der Knoten nichts mehr  $\ddot{a}ndert^b$ .

 $^a\mathrm{Im}$ Bezug zu Compilerbau die Linveness Analayse.

<sup>b</sup>Bis diese sich stabilisiert haben

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

#### Definition 2.16: Two-Space Copying Collector



Ein Garbabe Collector bei dem der Heap in FromSpace und ToSpace unterteilt wird und bei nicht ausreichendem Speicherplatz auf dem Heap alle Variablen, die in Zukunft noch verwendet werden vom FromSpace zum ToSpace kopiert werden. Der aktuelle ToSpace wird danach zum neuen FromSpace und der aktuelle FromSpace wird danach zum neuen ToSpace.<sup>a</sup>

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

# **Bootstrapping**

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler**  $C_{RETI-PicoC}^{PicoC}$  (Defintion 2.17) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache  $L_{Python}$ , in der der momentane Compiler  $C_{PicoC}$  für  $L_{PicoC}$  implementiert ist und die Unabhängigkeit von einer **anderen Maschine**, die bisher immer für das Cross-Compiling notwendig war erreicht werden.

#### Definition 2.17: Self-compiling Compiler

Compiler  $C_w^w$ , der in der Sprache  $L_w$  geschrieben ist, die er selbst kompiliert. Also ein Compiler, der sich selbst kompilieren kann.<sup>a</sup>

<sup>a</sup>J. Earley und Sturgis, "A formalism for translator interactions".

Will man nun für eine Maschine  $M_{RETI}$ , auf der bisher keine anderen Programmiersprachen mittels Bootstrapping (Definition 2.20) zum laufen gebracht wurden, den gerade beschriebenen Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  implementieren und hat bereits den gesamtem Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  in der Sprache  $L_{PicoC}$  geschrieben, so stösst man auf ein Problem, dass auf das Henne-Ei-Problem<sup>1</sup> reduziert werden kann. Man bräuchte, um den Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  auf der Maschine  $M_{RETI}$  zu kompilieren bereits einen kompilierten Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$ , der mit der Maschinensprache  $B_{RETI}$  läuft. Es liegt eine zirkulare Abhängigkeit vor, die man nur auflösen kann, indem eine externe Entität zur Hilfe nimmt.

Da man den gesamten Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  nicht selbst komplett in der Maschinensprache  $B_{RETI}$  schreiben will, wäre eine Möglichkeit, dass man den Cross-Compiler  $C_{PicoC}^{Python}$ , den man bereits in der Programmiersprache  $L_{Python}$  implementiert hat, der in diesem Fall einen Bootstrapping Compiler (Definition 2.19) darstellt, auf einer anderen Maschine  $M_{other}$  dafür nutzt, damit dieser den Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  für die Maschine  $M_{RETI}$  kompiliert bzw. bootstraped und man den kompilierten RETI-Maschiendencode dann einfach von der Maschine  $M_{other}$  auf die Maschine  $M_{RETI}$  kopiert.<sup>2</sup>

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

 $<sup>^2</sup>$ Im Fall, dass auf der Maschine  $M_{RETI}$  die Programmiersprache  $L_{Python}$  bereits mittels Bootstrapping zum Laufen gebracht wurde, könnte der Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$  auch mithife des Cross-Compilers  $C_{PicoC}^{Python}$  als externe Entität und der Programmiersprache  $L_{Python}$  auf der Maschine  $M_{RETI}$  selbst kompiliert werden.

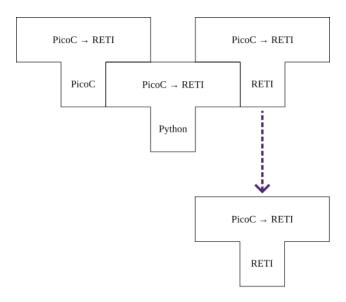


Abbildung 2.2: Cross-Compiler als Bootstrap Compiler.

### Anmerkung 9

Einen ersten minimalen Compiler  $C_{2\_w\_min}$  für eine Maschine  $M_2$  und Wunschsprache  $L_w$  kann man entweder mittels eines externen Bootstrap Compilers  $C_w^o$  kompilieren<sup>a</sup> oder man schreibt ihn direkt in der Maschinensprache  $B_2$  bzw. wenn ein Assembler vorhanden ist, in der Assemblesprache  $A_2$ .

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

 ${}^{a}$ In diesem Fall, dem Cross-Compiler  $C_{PicoC}^{Python}$ 

#### Definition 2.18: Minimaler Compiler

**I** 

Compiler  $C_{w\_min}$ , der nur die notwendigsten Funktionalitäten einer Wunschsprache  $L_w$ , wie Schleifen, Verzweigungen kompiliert, die für die Implementierung eines Self-compiling Compilers  $C_w^w$  oder einer ersten Version  $C_{w_i}^{w_i}$  des Self-compiling Compilers  $C_w^w$  wichtig sind.  $a^b$ 

<sup>a</sup>Den PicoC-Compiler könnte man auch als einen minimalen Compiler ansehen.

<sup>b</sup>Thiemann, "Compilerbau".

#### Definition 2.19: Boostrap Compiler

1

Compiler  $C_w^o$ , der es ermöglicht einen Self-compiling Compiler  $C_w^w$  zu boostrapen, indem der Self-compiling Compiler  $C_w^o$  mit dem Bootstrap Compiler  $C_w^o$  kompiliert wird. Der Bootstrapping Compiler stellt die externe Entität dar, die es ermöglicht die zirkulare Abhängikeit, dass initial ein Self-compiling Compiler  $C_w^o$  bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.

<sup>a</sup>Dabei kann es sich um einen lokal auf der Maschine selbst laufenden Compiler oder auch um einen Cross-Compiler

handeln.

<sup>b</sup>Thiemann, "Compilerbau".

Aufbauend auf dem Self-compiling Compiler  $C_{RETI\_PicoC}^{PicoC}$ , der einen minimalen Compiler (Definition 2.18) für eine Teilmenge der Programmiersprache C bzw.  $L_C$  darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw.  $L_C$  für die Maschine  $M_{RETI}$  mittels Bootstrapping implementieren.<sup>3</sup>

Das bewerkstelligt man, indem man iterativ auf der Zielmaschine  $M_{RETI}$  selbst, aufbauend auf diesem minimalen Compiler  $C_{RETI\_PicoC}^{PicoC}$ , wie in Subdefinition 2.20.1 den minimalen Compiler schrittweise zu einem immer vollständigeren C-Compiler  $C_C$  weiterentwickelt.

#### Definition 2.20: Bootstrapping

Z

Wenn man einen Self-compiling Compiler  $C_w^w$  einer Wunschsprache  $L_w$  auf einer Zielmaschine M zum laufen bringt<sup>abcd</sup>. Dabei ist die Art von Bootstrapping in 2.20.1 nochmal gesondert hervorzuheben:

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

<sup>a</sup>Z.B. mithilfe eines Bootstrap Compilers.

<sup>b</sup>Der Begriff hat seinen Ursprung in der englischen Redewendung "pulling yourself up by your own bootstraps", was im deutschen ungefähr der aus den Lügengeschichten des Freiherrn von Münchhausen bekannten Redewendung "sich am eigenen Schopf aus dem Sumpf ziehen"entspricht.

<sup>c</sup>Hat man einmal einen solchen Self-compiling Compiler  $C_w^w$  auf der Maschine M zum laufen gebracht, so kann man den Compiler auf der Maschine M weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache  $L_o$ , in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.

 $^d$ Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute Probe aufs Exempel darstellen, dass der Compiler auch wirklich funktioniert.

<sup>e</sup>Es ist hierbei theoretisch nicht notwendig den letzten Self-compiling Compiler  $C_{w_{i-1}}^{w_{i-1}}$  für das Kompilieren des neuen Self-compiling Compilers  $C_{w_{i}}^{w_{i}}$  zu verwenden, wenn z.B. der Self-compiling Compiler  $C_{w_{i-3}}^{w_{i-3}}$  auch bereits alle Funktionalitäten, die beim Schreiben des Self-compiling Compilers  $C_{w}^{w}$  verwendet werden kompilieren kann.

<sup>f</sup>Der Begriff ist sinnverwandt mit dem Booten eines Computers, wo die wichtigste Software, der Kernel zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, welches bei Bedarf dann Systemsoftware, Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber. und Anwendungssoftware, Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt, lädt.

<sup>g</sup>J. Earley und Sturgis, "A formalism for translator interactions".

<sup>&</sup>lt;sup>3</sup>Natürlich könnte man aber auch einfach den Cross-Compiler  $C_{PicoC}^{Python}$  um weitere Funktionalitäten von  $L_C$  erweitern, hat dann aber weiterhin eine Abhängigkeit von der Programmiersprache  $L_{Python}$ .

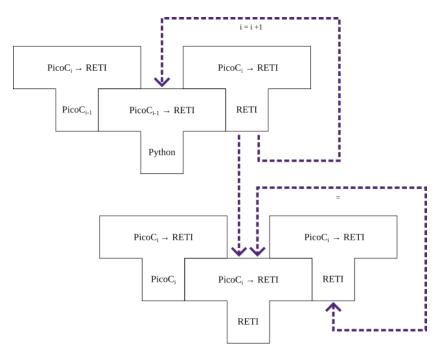


Abbildung 2.3: Iteratives Bootstrapping.

### Anmerkung Q

Auch wenn ein Self-compiling Compiler  $C_{w_i}^{w_i}$  in der Subdefinition 2.20.1 selbst in einer früheren Version  $L_{w_{i-1}}$  der Programmiersprache  $L_{w_i}$  geschrieben wird, wird dieser nicht mit  $C_{w_i}^{w_{i-1}}$  bezeichnet, sondern mit  $C_{w_i}^{w_i}$ , da es bei Self-compiling Compilern darum geht, dass diese zwar in der Subdefinition 2.20.1 eine frühere Version  $C_{w_{i-1}}^{w_{i-1}}$  nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

# Literatur

#### Online

- A-Normalization: Why and How (with code). URL: https://matt.might.net/articles/a-normalization/(besucht am 23.07.2022).
- Earley Parser. URL: https://rahul.gopinath.org/post/2021/02/06/earley-parsing/ (besucht am 20.06.2022).
- Errors in C/C++ GeeksforGeeks. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10.05.2022).
- JSON parser Tutorial Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/json\_tutorial.html (besucht am 09.07.2022).
- Ljohhuh. What is an immediate value? 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13.04.2022).
- Parsing Expressions · Crafting Interpreters. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09.07.2022).
- Transformers & Visitors Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09.07.2022).
- What is Bottom-up Parsing? URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22.06.2022).
- What is Top-Down Parsing? URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22.06.2022).

#### Bücher

• G. Siek, Jeremy. Course Webpage for Compilers (P423, P523, E313, and E513). 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28.01.2022).

#### Artikel

- Earley, J. und Howard E. Sturgis. "A formalism for translator interactions". In: *CACM* (1970). DOI: 10.1145/355598.362740.
- Earley, Jay. "An efficient context-free parsing". In: 13 (1968). URL: https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf (besucht am 10.08.2022).

# Vorlesungen

- Nebel, Prof. Dr. Bernhard. "Theoretische Informatik". Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index\_de.html (besucht am 09.07.2022).
- Scholl, Prof. Dr. Christoph. "Betriebssysteme". Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach\_main.php?id=157 (besucht am 09.07.2022).
- 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).
- Earley parser. In: Wikipedia. Page Version ID: 1090848932. 31. Mai 2022. URL: https://en.wikipedia.org/w/index.php?title=Earley\_parser&oldid=1090848932 (besucht am 15.08.2022).
- Lark a parsing toolkit for Python. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28.04.2022).