

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

## PicoC-Compiler

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

BACHELORARBEIT

*Abgabedatum:* 28<sup>th</sup> April 2022

*Author:*  
Jürgen Mattheis

*Gutachter:*  
Prof. Dr. Scholl

*Betreuung:*  
M.Sc. Seufert

---

Eine Bachelorarbeit am Lehrstuhl für  
Betriebssysteme

---

---

---

## **ERKLÄRUNG**

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

---

---

# Inhaltsverzeichnis

<b>1</b>	<b>Motivation</b>	<b>8</b>
1.1	PicoC und RETI	8
1.2	Aufgabenstellung	8
1.3	Eigenheiten der Sprache C	8
1.4	Richtlinien	8
<b>2</b>	<b>Einführung</b>	<b>9</b>
2.1	Compiler und Interpreter	9
2.1.1	T-Diagramme	12
2.2	Grammatiken	14
2.3	Grundlagen	14
2.3.1	Mehrdeutige Grammatiken	15
2.3.2	Präzidenz und Assoziativität	15
2.4	Lexikalische Analyse	16
2.5	Syntaktische Analyse	18
2.6	Code Generierung	24
2.7	Fehlermeldungen	24
<b>3</b>	<b>Implementierung</b>	<b>25</b>
3.1	Architektur	25
3.2	Lexikalische Analyse	26
3.2.1	Verwendung von Lark	26
3.2.2	Basic Parser	26
3.3	Syntaktische Analyse	26
3.3.1	Verwendung von Lark	26
3.3.2	Umsetzung von Präzidenz	26
3.3.3	Derivation Tree Generierung	27
3.3.4	Early Parser	27
3.3.5	Derivation Tree Vereinfachung	27
3.3.6	Abstrakt Syntax Tree Generierung	27
3.4	Code Generierung	27
3.4.1	Passes	27
3.4.2	Umsetzung von Pointern und Arrays	28
3.4.3	Umsetzung von Structs	28
3.4.4	Umsetzung von Funktionen	28
3.4.5	Umsetzung kleinerer Details	28
3.5	Fehlermeldungen	28
3.5.1	Error Handler	28
<b>4</b>	<b>Ergebnisse und Ausblick</b>	<b>29</b>
4.1	Funktionsumfang	29
4.2	Qualitätssicherung	29
4.3	Kommentierter Kompilervorgang	29
4.4	Erweiterungsideen	29
<b>A</b>	<b>Appendix</b>	<b>33</b>
A.1	Konkrete und Abstrakte Syntax	33

A.2	Bedienungsanleitungen . . . . .	33
A.2.1	PicoC-Compiler . . . . .	33
A.2.2	Showmode . . . . .	33
A.2.3	Entwicklertools . . . . .	33

---

---

# Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	14
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	14
2.3	Veranschaulichung der Lexikalischen Analyse . . . . .	18
2.4	Veranschaulichung der Syntaktischen Analyse . . . . .	23
3.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	25
3.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	26
3.3	Architektur mit allen Passes ausgeschrieben . . . . .	26
4.1	Cross-Compiler als Bootstrap Compiler . . . . .	30
4.2	Iteratives Bootstrapping . . . . .	32

---

---

# Tabellenverzeichnis

3.1 Präzidenzregeln von PicoC . . . . .	27
---	----

---

---

# Definitionen

2.1	Interpreter . . . . .	9
2.2	Compiler . . . . .	9
2.3	Maschiensprache . . . . .	10
2.4	Assemblersprache (bzw. engl. Assembly Language) . . . . .	10
2.5	Assembler . . . . .	11
2.6	Objectcode . . . . .	11
2.7	Linker . . . . .	11
2.8	Immediate . . . . .	11
2.9	Transpiler (bzw. Source-to-source Compiler) . . . . .	11
2.10	Cross-Compiler . . . . .	12
2.11	T-Diagram Programm . . . . .	12
2.12	T-Diagram Übersetzer . . . . .	13
2.13	T-Diagram Interpreter . . . . .	13
2.14	T-Diagram Maschine . . . . .	13
2.15	Sprache . . . . .	14
2.16	Chromsky Hierarchie . . . . .	14
2.17	Grammatik . . . . .	15
2.18	Reguläre Sprachen . . . . .	15
2.19	Kontextfreie Sprachen . . . . .	15
2.20	Ableitung . . . . .	15
2.21	Links- und Rechtsableitung . . . . .	15
2.22	Linksrekursive Grammatiken . . . . .	15
2.23	Ableitungsbaum . . . . .	15
2.24	Mehrdeutige Grammatik . . . . .	15
2.25	Assoziativität . . . . .	15
2.26	Präzidenz . . . . .	15
2.27	Wortproblem . . . . .	16
2.28	LL(k)-Grammatik . . . . .	16
2.29	Pattern . . . . .	16
2.30	Lexeme . . . . .	16
2.31	Lexer (bzw. Scanner) . . . . .	17
2.32	Bezeichner (bzw. Identifier) . . . . .	17
2.33	Literal . . . . .	18
2.34	Konkrete Syntax . . . . .	19
2.35	Derivation Tree (bzw. Parse Tree) . . . . .	19
2.36	Parser . . . . .	19
2.37	Recognizer (bzw. Erkennen) . . . . .	20
2.38	Transformer . . . . .	21
2.39	Visitor . . . . .	21
2.40	Abstrakte Syntax . . . . .	21
2.41	Abstrakt Syntax Tree . . . . .	22
2.42	Pass . . . . .	24
2.43	Fehlermeldung . . . . .	24
3.1	Symboltabelle . . . . .	27
4.1	Self-compiling Compiler . . . . .	29
4.2	Minimaler Compiler . . . . .	30
4.3	Bootstrap Compiler . . . . .	31

4.4	Bootstrapping . . . . .	31
-----	-------------------------	----



---

---

# 1 Motivation

1.1 PicoC und RETI

1.2 Aufgabenstellung

1.3 Eigenheiten der Sprache C

1.4 Richtlinien

# 2 Einführung

## 2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** in die **RETI-Sprache** 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 2.1 zu belegen (siehe Subkapitel 4.2).

### Definition 2.1: Interpreter

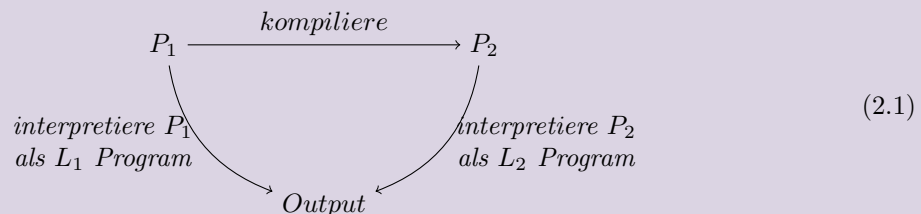
*Interpretiert die **Instructions** bzw. **Statements** eines Programmes  $P$  direkt.*

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

### Definition 2.2: Compiler

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

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



Im Folgenden wird ein voll ausgeschriebener **Compiler** als  $C_{i.w.k.min}^{o-j}$  geschrieben, wobei  $C_w$  die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache  $L_{B_i}$  einer Maschine  $M_i$  kompiliert. Fall die Notwendigkeit besteht die **Maschiene**

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

$M_i$  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 4.2) kann man das als  $C_{min}$  schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschinenncode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

### Definition 2.3: Maschinentersprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauer gesagt in der **Execute-Phase** übernehmen kann und allgemein in einer **geringen konstanten** Anzahl von Fetch- und Execute Phasen im **komplexeren Fall**. Die Maschinenbefehle sind meist so designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.<sup>a</sup>.*

<sup>a</sup>Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.8) haben.

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

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

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

<sup>b</sup>Z.B. erlaubt die Assemblersprache des **GCC** für die **X86\_64-Architektur** für manche Operanden die Syntax **n(%r)**, die einen **Speicherzugriff** mit **Offset**  $n$  zur Adresse, die im **Register** **%r** steht durchführt, wobei z.B. die Klammern () usw. nur „syntaktischer Zucker“ sind und natürlich nicht mitcodiert werden.

<sup>c</sup>Z.B. sind in **X86\_64** die Instructions in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels **jmp <label>** gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet kann keine direkte Entsprechung in einem Prozessor und Hauptspeicher haben.

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

**Definition 2.5: Assembler**

Übersetzt im allgemeinen *Assemblercode*, der in *Assemblersprache* geschrieben ist zu *Maschinencode* der in *Maschiensprache* geschrieben ist.

**Definition 2.6: Objectcode**

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

**Definition 2.7: Linker**

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

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

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht dessen mögliche interne Umsetzung<sup>2</sup>.

**Definition 2.8: Immediate**

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

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

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

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

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

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

**Definition 2.10: Cross-Compiler**

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

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

Ein **Cross-Compiler** ist entweder notwendig, wenn noch kein Compiler  $C_w$  für die **Wunschsprache**  $L_w$  existiert, der unter der **Maschinensprache**  $B_2$  einer Zielmaschine  $M_2$  läuft oder die Zielmaschine  $M_2$  nicht ausreichend **Rechenleistung** hat, um ein Programm in der Wunschsprache  $L_w$  selbst zeitnah zu kompilieren.<sup>3</sup>

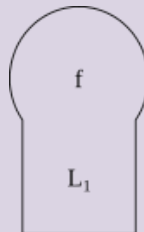
**2.1.1 T-Diagramme**

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**<sup>4</sup> besonders gut, da diese optimal darauf **zugeschnitten** sind die Eigenheiten von Compilern in ihrer Art der Darstellung unterzubringen.

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

**Definition 2.11: T-Diagram Programm**

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



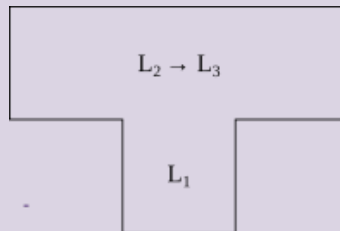
Es ist nicht notwendig beim einem entsprechenden **Platzhalter** für einen **Namen**, den **Namen der Sprache** an ein  $L$  dranzuhängen, weil hier immer eine **Sprache** steht.

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

<sup>4</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 2.12: T-Diagramm Übersetzer**

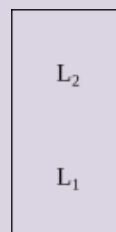
Repräsentiert einen **Übersetzer**, der in der **Sprache**  $L_1$  geschrieben ist und **Programme** von der **Sprache**  $L_2$  in die **Sprache**  $L_3$  kompiliert.



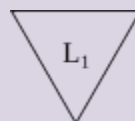
Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

**Definition 2.13: T-Diagramm Interpreter**

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

**Definition 2.14: T-Diagramm Maschine**

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



<sup>a</sup>Wenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist das auch erlaubt das so aufzuschreiben, 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 **Kompositionen** bilden, indem man sie **adjacent** zueinander platziert. Allgemein lässt sich grob sagen, dass **vertikale Adjazents** für **Interpretation** und **horizontale Adjazents** für **Übersetzung** steht.

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

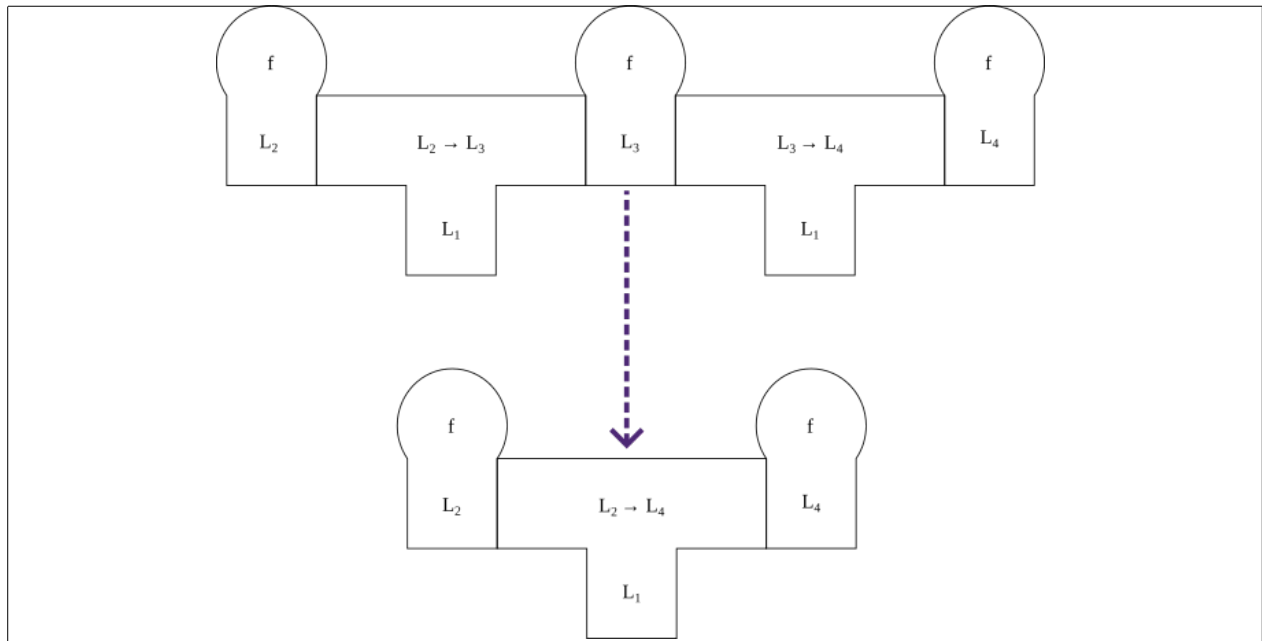


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

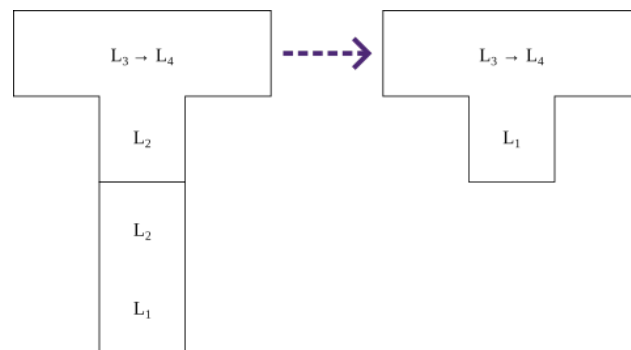


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

## 2.2 Grammatiken

## 2.3 Grundlagen

**Definition 2.15: Sprache**

**Definition 2.16: Chomsky Hierarchie**

**Definition 2.17: Grammatik****Definition 2.18: Reguläre Sprachen****Definition 2.19: Kontextfreie Sprachen****Definition 2.20: Ableitung****Definition 2.21: Links- und Rechtsableitung****Definition 2.22: Linksrekursive Grammatiken**

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

**2.3.1 Mehrdeutige Grammatiken****Definition 2.23: Ableitungsbaum****Definition 2.24: Mehrdeutige Grammatik****2.3.2 Präzidenz und Assoziativität****Definition 2.25: Assoziativität****Definition 2.26: Präzidenz**



**Definition 2.27: Wortproblem****Definition 2.28: LL(k)-Grammatik**

Eine Grammatik ist **LL(k)** für  $k \in \mathbb{N}$ , falls jeder Ableitungsschritt eindeutig durch die nächsten  $k$  **Symbole** des **Eingabeworts** bzw. in Bezug zu Compilerbau **Token** des **Inputstrings** zu bestimmen ist<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>a</sup>Das wird auch als **Lookahead** von  $k$  bezeichnet.

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

## 2.4 Lexikalische Analyse

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

**Definition 2.29: Pattern**

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

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

<sup>b</sup>What is the difference between a token and a lexeme?

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

**Definition 2.30: Lexeme**

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

<sup>a</sup>What is the difference between a token and a lexeme?

Diese **Lexeme** werden vom **Lexer** im **Inputstring** identifiziert und **Tokens**  $T$  zugeordnet (Definition 2.31). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

**Definition 2.31: Lexer (bzw. Scanner)**

Ein **Lexer** ist eine **partielle Funktion**  $lex : \Sigma^* \rightarrow (N \times W)^*$ , welche ein **Wort** aus  $\Sigma^*$  auf ein **Token**  $T$  mit einem **Tokennamen**  $N$  und einem **Tokenwert**  $W$  abbildet, falls diese Folge von Symbolen sich unter der **regulären Grammatik**  $G_{Lex}$ , der **regulären Sprache**  $L_{Lex}$  ableiten lässt.<sup>a</sup>

<sup>a</sup>lecture-notes-2021.

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

Um Verwirrung verzubäuen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

In der Syntaktischen Analyse sind die **Tokennamen** die Symbole.

In der Code Generierung sind die **Bezeichner** (Definition 2.32) 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 3 **Symboltabelle** genannt wird.

**Definition 2.32: Bezeichner (bzw. Identifier)**

**Lexeme** bzw. **Tokenwert**, das bzw. der eine Konstante, Variable, Funktion usw. **eindeutig** benennt, außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw.<sup>a</sup>

<sup>a</sup>In diesem Fall wird die **Signatur** der Funktion als weiteres Unterscheidungsmerkmal hinzugenommen, damit es eindeutig ist

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>5</sup> und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen Zeichen das leere Wort  $\epsilon$  zuordnet. Das ist auch im Sinne der Definition, denn  $\epsilon \in \Sigma^*$ . 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`, bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für Zeichenfolgen, wie `if` oder `}` sind die **Überbegriffe** genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

<sup>5</sup>In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Programmiersprache, die man zur Implementierung des Lexers nutzt wegabstrahiert.

Ein **Lexeme** ist damit aber nicht das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann z.B. der Wert 99 durch zwei verschiedene Literale dargestellt werden, einmal als ASCII-Zeichen 'c' und des Weiteren auch in Dezimalschreibweise als 99<sup>6</sup>. Der **Tokenwert** ist jedoch der letztendliche Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik**  $G_{Lex}$ , die zur Beschreibung der Token  $T$  einer regulären Sprache  $L_{Lex}$  verwendet wird, ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut<sup>7</sup>, unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

### Definition 2.33: Literal

*Eine von möglicherweise vielen weiteren **Darstellungsformen** für ein und denselben **Wert**.*

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

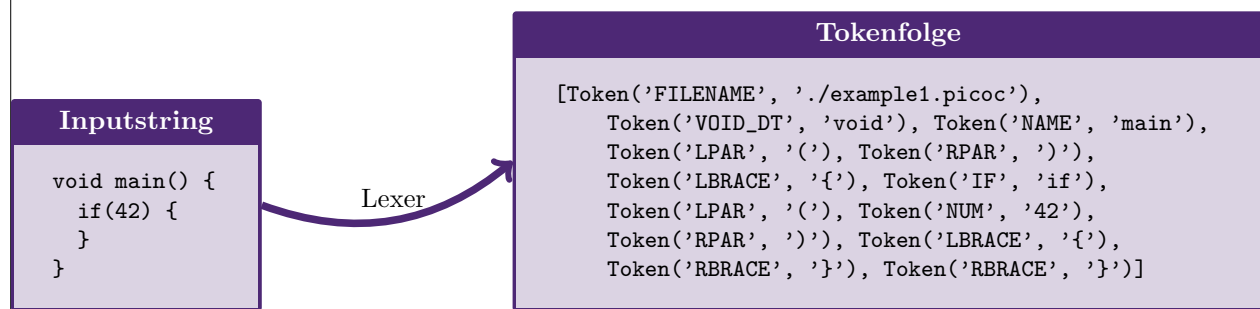


Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

## 2.5 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik**  $G_{Parse}$  notwendig, um diese Sprache 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 Klammern '(' bzw. öffnende geschweifte Klammern '{' es momentan gibt, die noch nicht durch eine entsprechende schließende Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden.

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

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

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

**Definition 2.34: Konkrete Syntax**

*Syntax einer Sprache, die durch die Grammatiken  $G_{Lex}$  und  $G_{Parse}$  zusammengekommen beschrieben wird.*

*Ein Programm in seiner Textrepräsentation, wie es in einer Textdatei nach den Produktionen der Grammatiken  $G_{Lex}$  und  $G_{Parse}$  abgeleitet steht, bevor man es kompiliert, ist in Konkreter Syntax aufgeschrieben.<sup>a</sup>*

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

**Definition 2.35: Derivation Tree (bzw. Parse Tree)**

*Compilerinterne Darstellung eines in Konkreter Syntax geschriebenen Inputstrings als Baumdatenstruktur, in der Nichtterminalsymbole die Inneren Knoten des Baumes und Terminalsymbole die Blätter des Baumes bilden. Jede Produktions der Grammatik  $G_{Parse}$ , die ein Teil der Konkrete Syntax ist, wird zu einem eigenen Knoten.*

*Der Derivation Tree wird optimalerweise immer so konstruiert bzw. die Konkrete Syntax immer so definiert, dass sich möglichst einfach ein Abstrakt Syntax Tree daraus konstruieren lässt.*

**Definition 2.36: Parser**

*Ein Programm, dass eine Eingabe in eine für die Weiterverarbeitung taugliche Form bringt.*

**2.36.1:** *In Bezug auf Compilerbau ist ein Parser ein Programm, dass einen Inputstring von Konkreter Syntax in die compilerinterne Darstellung eines Derivation Tree übersetzt, was auch als Parsen bezeichnet wird.<sup>a, b</sup>*

<sup>a</sup>Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von Konkreter Syntax in Abstrakte Syntax übersetzt. Im Folgenden wird allerdings die obige Definition 2.36.1 verwendet.

<sup>b</sup>Compiler Design - Phases of Compiler.

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung entstehen, ob ein **Lexer** nach der obigen Definition nicht auch ein **Parser** ist.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines Parsers. Der Parser vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich. Aber für sich isoliert, ohne Bezug zu Compilerbau betrachtet, ist ein Lexer nach Definition 2.36 ebenfalls ein Parser. Aber im Compilerbau hat **Parser** eine spezifischere Definition und hier überwiegt beim **Lexer** seine Funktionalität, dass er den Inputstring lexikalisch weiterverarbeitet, um ihn als Lexer zu bezeichnen, der Teil eines Parsers ist.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** (Definition 2.36) 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 nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine **Zahl** steht und nicht, welchen konkreten Wert diese **Zahl** hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.6 relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.37), denn ein Parser löst das **Wortpro-**

**blem** (Definition 2.27) für die **Sprache**, die durch die **Konkrete Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Derivation Tree**.

### Definition 2.37: Recognizer (bzw. Erkenner)

*Entspricht dem Maschinenmodell eines **Automaten**. Im Bezug auf Compilerbau entspricht der **Recognizer** einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** erkennt, ob ein Inputstring bzw. **Wort** sich mit den Produktionen der **Konkrete Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Syntax** beschrieben wird oder nicht<sup>a</sup>.*

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

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

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

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

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

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

**Rekursiver Abstieg** kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.28) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewünschte Inputstring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind. Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.22) allerdings nicht möglich, ohne die Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der **Grammatik** entfernt zu haben, da diese zu **Unendlicher Rekursion** führt<sup>b</sup>

Wenn man eine **LL(k)** Grammatik hat, kann man auf **Backtracking verzichten** und es reicht einfach nur immer **k Symbole** im **Eingabewort** bzw. in Bezug auf Compilerbau **Token** im **Inputstring** vorzuschauen. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.<sup>c</sup>

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen**, entsprechend der **Produktionen** der **Konkreten Syntax** rückwärts anzuwenden bis man beim **Startsymbol** landet.<sup>d</sup>

- **Chart Parser:** Es wird **Dynamische Programmierung** verwendet und partielle Zwischenergebnisse werden in einer **Tabelle** (bzw. einem **Chart**) gespeichert und können wiederverwendet werden. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist.<sup>e</sup>

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

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

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

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

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

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

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstrakt Syntax Tree** ermöglicht es die Operationen, die der Compiler bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen.

#### Definition 2.38: Transformer

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

#### Definition 2.39: Visitor

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

Kann theoretisch auch zur Konstruktion eines **Abstrakt Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakt Syntax Tree** verantwortlich ist, aber dafür ist ein **Transformer** besser geeignet.

#### Definition 2.40: Abstrakte Syntax

**Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstrakt Syntax Trees** möglich sind.<sup>a</sup>

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.

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

**Definition 2.41: Abstrakt Syntax Tree**

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

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

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

Je weiter **unten**<sup>8</sup> und **links** ein Knoten im **Abstrakt Syntax Tree** liegt, desto eher wird dieser Knoten komplett abgearbeitet sein, da in der **Code Generierung** die Knoten nach dem **Depth First Search** Prinzip abgearbeitet werden.

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

<sup>8</sup>In der Informatik wachsen **Bäume** von **oben-nach-unten**. Die **Wurzel** ist also **oben**.



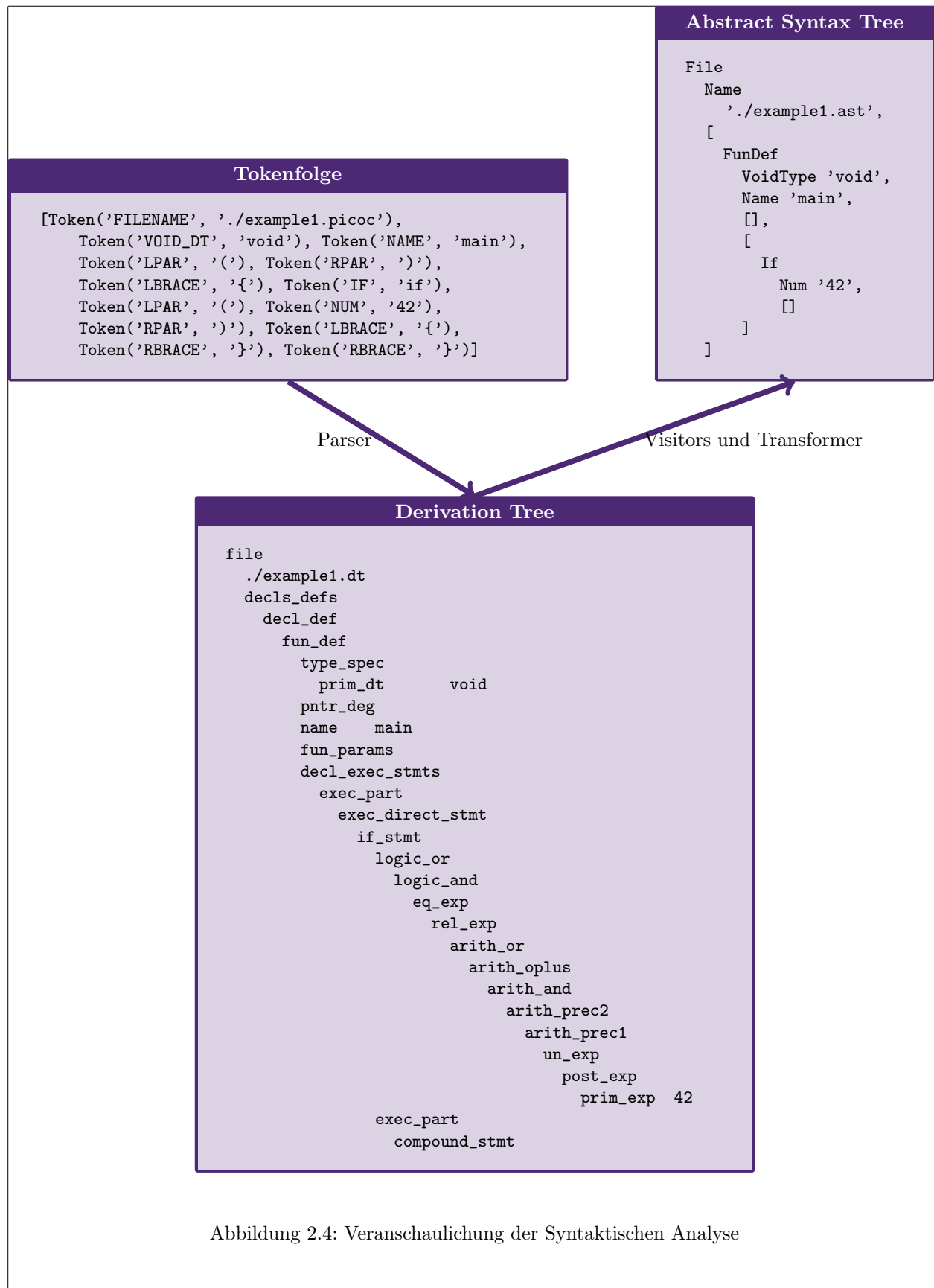


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse



## 2.6 Code Generierung

**Definition 2.42: Pass**

## 2.7 Fehlermeldungen

**Definition 2.43: Fehlermeldung**

# 3 Implementierung

## 3.1 Architektur

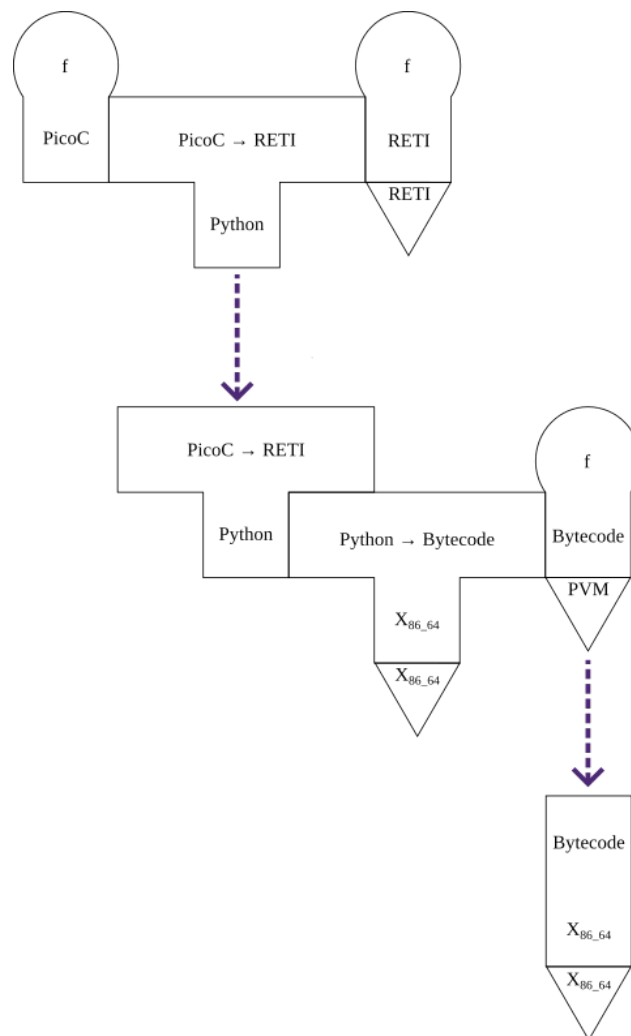


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

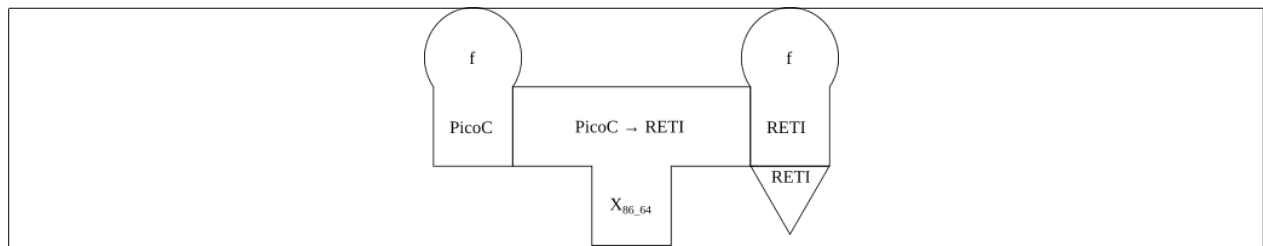


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

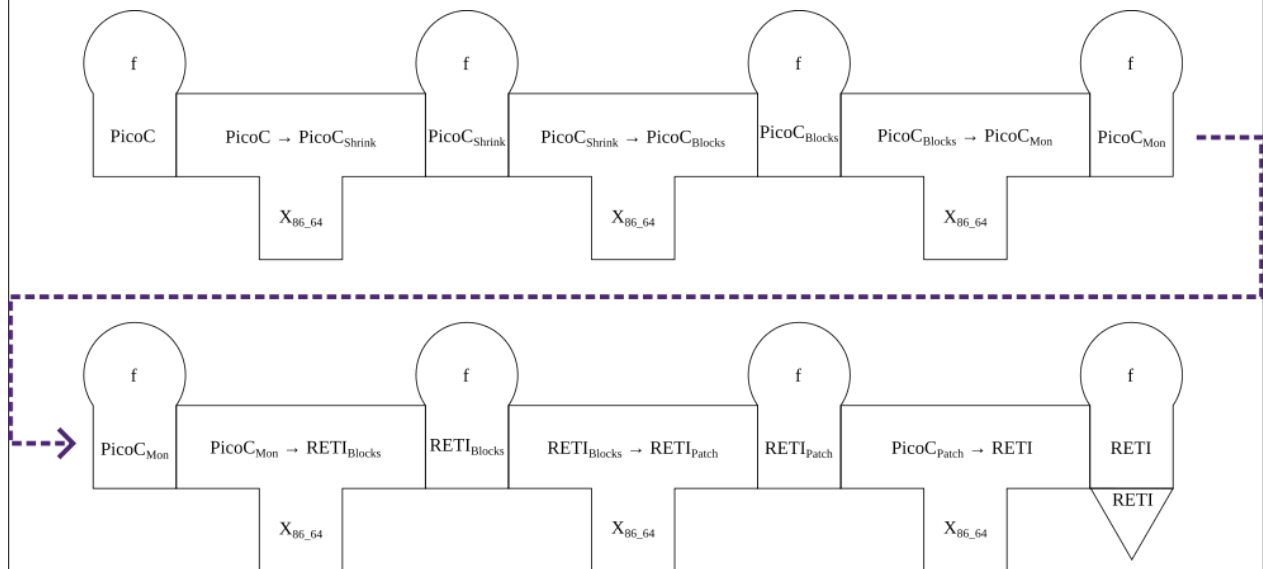


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

## 3.2 Lexikalische Analyse

### 3.2.1 Verwendung von Lark

### 3.2.2 Basic Parser

## 3.3 Syntaktische Analyse

### 3.3.1 Verwendung von Lark

### 3.3.2 Umsetzung von Präzidenz

Die **PicoC Sprache** hat dieselben **Präzidenzregeln** implementiert, wie die **Sprache C<sup>1</sup>**. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 3.3.2 aufgelistet.

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

Präzedenz	Operator	Beschreibung	Assoziativität
1	a() a[] a.b	Funktionsaufruf Indexzugriff Attributzugriff	Links, dann rechts →
2	-a !a ~a *a &a	Unäres Minus Logisches NOT und Bitweise NOT Dereferenz und Referenz, auch Adresse-von	Rechts, dann links ←
3	a*b a/b a%b	Multiplikation, Division und Modulo	Links, dann rechts →
4	a+b a-b	Addition und Subtraktion	
5	a<b a<=b a>b a>=b	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	a==b a!=b	Gleichheit und Ungleichheit	
7	a&b	Bitweise UND	
8	a^b	Bitweise XOR (exclusive or)	
9	a b	Bitweise ODER (inclusive or)	
10	a&& b	Logisches UND	
11	a   b	Logisches ODER	
12	a=b	Zuweisung	Rechts, dann links ←
13	a,b	Komma	Links, dann rechts →

Tabelle 3.1: Präzidenzregeln von PicoC

### 3.3.3 Derivation Tree Generierung

### 3.3.4 Early Parser

### 3.3.5 Derivation Tree Vereinfachung

### 3.3.6 Abstrakt Syntax Tree Generierung

ASTNode

PicoC Nodes

RETI Nodes

## 3.4 Code Generierung

### 3.4.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

**Definition 3.1: Symboltabelle**

**RETI-Blocks Pass**

**RETI-Patch Pass**

**RETI Pass**

**3.4.2 Umsetzung von Pointern und Arrays**

**3.4.3 Umsetzung von Structs**

**3.4.4 Umsetzung von Funktionen**

**3.4.5 Umsetzung kleinerer Details**

**3.5 Fehlermeldungen**

**3.5.1 Error Handler**

# 4 Ergebnisse und Ausblick

## 4.1 Funktionsumfang

## 4.2 Qualitätssicherung

## 4.3 Kommentierter Kompiliervorgang

## 4.4 Erweiterungsideen

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

### Definition 4.1: Self-compiling Compiler

*Compiler  $C_w^w$ , der in der Sprache  $L_w$  **geschrieben** ist, die er **selbst** kompiliert. Also ein Compiler, der sich **selbst** kompilieren kann.*

Will man nun für eine Maschine  $M_{RETI}$ , auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 4.4) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$  implementieren und hat bereits den gesamten **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 4.3) 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

<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.

**RETI-Maschiendencode** dann einfach von der Maschine  $M_{other}$  auf die Maschine  $M_{RETI}$  kopiert.<sup>2</sup>

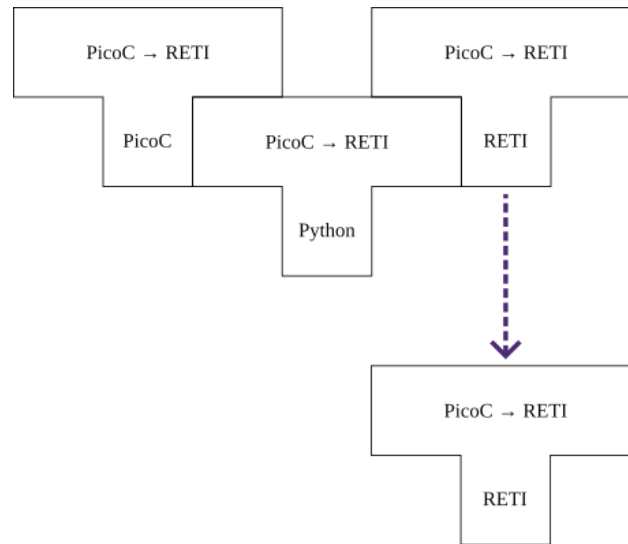


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

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 allerersten 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.

<sup>a</sup>In diesem Fall, dem **Cross-Compiler**  $C_{PicoC}^{Python}$ .

#### Definition 4.2: Minimaler Compiler

Compiler  $C_{w.min}$ , der nur die **notwendigsten Funktionalitäten** einer Wunschsprache  $L_w$ , wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers**  $C_w^o$  oder einer **ersten Version**  $C_{w_i}^w$  des Self-compiling Compilers  $C_w^w$  wichtig sind.<sup>a</sup>

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

<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 mithilfe des **Cross-Compilers**  $C_{PicoC}^{Python}$  als **externe Entität** und der Programmiersprache  $L_{Python}$  auf der Maschine  $M_{RETI}$  selbst kompiliert werden.

**Definition 4.3: Bootstrap Compiler**

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

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

Aufbauend auf dem **Self-compiling Compiler**  $C_{RETI\_PicoC}^{PicoC}$ , der einen **minimalen Compiler** (Definition 4.2) für eine Teilmenge der **Programmiersprache**  $C$  bzw.  $L_C$  darstellt, könnte man auch noch weitere Teile der Programmiersprache  $C$  bzw.  $L_C$  für die 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 4.4.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler**  $C_C$  weiterentwickelt.

**Definition 4.4: Bootstrapping**

Wenn man einen **Self-compiling Compiler**  $C_w^w$  einer Wunschsprache  $L_w$  auf einer **Zielmaschine**  $M$  zum laufen bringt<sup>a,b,c,d</sup>. Dabei ist die Art von **Bootstrapping** in 4.4.1 nochmal gesondert hervorzuheben:

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

<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 **Lügend Geschichten 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  $M$  weiterentwickeln, 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.

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

<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  $L_{Python}$ .



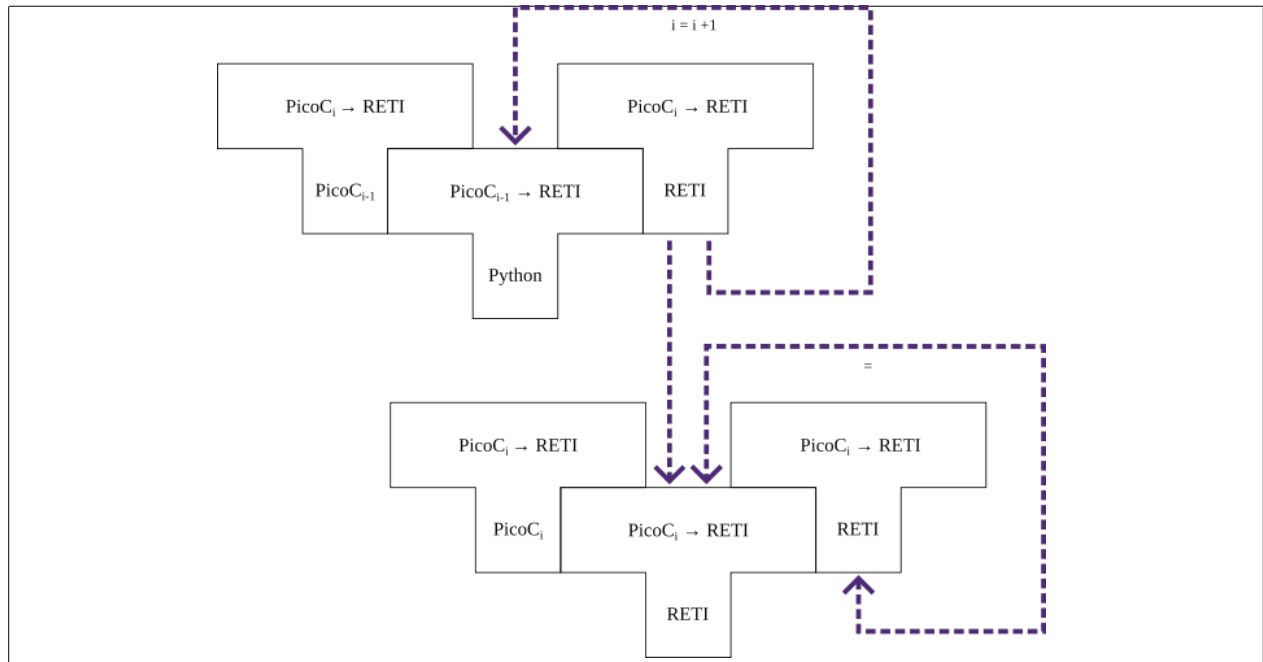


Abbildung 4.2: Iteratives Bootstrapping

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

---

---

# A Appendix

## A.1 Konkrete und Abstrakte Syntax

## A.2 Bedienungsanleitungen

### A.2.1 PicoC-Compiler

### A.2.2 Showmode

### A.2.3 Entwicklertools

---

---

# Literatur

## Online

- *C Operator Precedence* - *cppreference.com*. URL: [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence) (besucht am 27.04.2022).
- *Compiler Design - Phases of Compiler*. URL: [https://www.tutorialspoint.com/compiler\\_design/compiler\\_design\\_phases\\_of\\_compiler.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm) (besucht am 19.06.2022).
- *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).
- *lecture-notes-2021*. 20. Jan. 2022. URL: <https://github.com/Compiler-Construction-Uni-Freiburg/lecture-notes-2021/blob/56300e6649e32f0594bbbd046a2e19351c57dd0c/material/lexical-analysis.pdf> (besucht am 28.04.2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: <https://reverseengineering.stackexchange.com/q/17671> (besucht am 13.04.2022).
- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *What is the difference between a token and a lexeme?* NewbeDEV. URL: <http://newbedev.com/what-is-the-difference-between-a-token-and-a-lexeme> (besucht am 17.06.2022).
- *What is Top-Down Parsing?* URL: <https://www.tutorialspoint.com/what-is-top-down-parsing> (besucht am 22.06.2022).