

---

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>7</b>
1.1	PicoC und RETI	7
1.2	Aufgabenstellung	7
1.3	Eigenheiten der Sprache C	7
1.4	Richtlinien	7
<b>2</b>	<b>Einführung</b>	<b>8</b>
2.1	Compiler und Interpreter	8
2.1.1	T-Diagramme	8
2.2	Grammatiken	8
2.3	Grundlagen	8
2.3.1	Mehrdeutige Grammatiken	9
2.3.2	Präzidenz und Assoziativität	9
2.4	Lexikalische Analyse	9
2.5	Syntaktische Analyse	11
2.6	Code Generierung	13
2.7	Fehlermeldungen	13
<b>3</b>	<b>Implementierung</b>	<b>14</b>
3.1	Lexikalische Analyse	14
3.1.1	Verwendung von Lark	14
3.1.2	Basic Parser	14
3.2	Syntaktische Analyse	14
3.2.1	Verwendung von Lark	14
3.2.2	Umsetzung von Präzidenz	14
3.2.3	Derivation Tree Generierung	15
3.2.4	Early Parser	15
3.2.5	Derivation Tree Vereinfachung	15
3.2.6	Abstrakt Syntax Tree Generierung	15
3.3	Code Generierung	15
3.3.1	Passes	15
3.3.2	Umsetzung von Pointern und Arrays	15
3.3.3	Umsetzung von Structs	15
3.3.4	Umsetzung von Funktionen	15
3.3.5	Umsetzung kleinerer Details	15
3.4	Fehlermeldungen	15
3.4.1	Error Handler	15
<b>4</b>	<b>Ergebnisse und Ausblick</b>	<b>16</b>
4.1	Funktionsumfang	16
4.2	Qualitätskontrolle	16
4.3	Kommentierter Kompiliervorgang	16
4.4	Erweiterungsideen	16
<b>A</b>	<b>Appendix</b>	<b>17</b>
A.1	Konkrete und Abstrakte Syntax	17
A.2	Bedienungsanleitungen	17

A.2.1	PicoC-Compiler . . . . .	17
A.2.2	Showmode . . . . .	17
A.2.3	Entwicklertools . . . . .	17

---

---

# Abbildungsverzeichnis

---

---

# Tabellenverzeichnis

3.1 Präzidenzregeln von PicoC . . . . .	14
---	----

---

---

# Definitionen

1	Compiler . . . . .	8
2	Interpreter . . . . .	8
3	T-Diagram . . . . .	8
4	Sprache . . . . .	8
5	Chromsky Hierarchie . . . . .	8
6	Grammatik . . . . .	8
7	Reguläre Sprachen . . . . .	8
8	Kontextfreie Sprachen . . . . .	9
9	Ableitungsbaum . . . . .	9
10	Mehrdeutige Grammatik . . . . .	9
11	Assoziativität . . . . .	9
12	Präzidenz . . . . .	9
13	Pattern . . . . .	9
14	Lexeme . . . . .	9
15	Lexer (bzw. Scanner) . . . . .	10
16	Literal . . . . .	11
17	Parser . . . . .	11
18	Recognizer . . . . .	12
19	Aktion . . . . .	12
20	Konkrete Syntax . . . . .	12
21	Node . . . . .	12
22	Derivation Tree (bzw. Parse Tree) . . . . .	13
23	Transformer . . . . .	13
24	Visitor . . . . .	13
25	Abstrakte Syntax . . . . .	13
26	Abstrakte Syntax Tree . . . . .	13
27	Pass . . . . .	13
28	Fehlermeldung . . . . .	13
29	Symboltabelle . . . . .	15

---

---

# 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

Definition 1: Compiler

Definition 2: Interpreter

### 2.1.1 T-Diagramme

Definition 3: T-Diagram

## 2.2 Grammatiken

## 2.3 Grundlagen

Definition 4: Sprache

Definition 5: Chomsky Hierarchie

Definition 6: Grammatik

Definition 7: Reguläre Sprachen

**Definition 8: Kontextfreie Sprachen****2.3.1 Mehrdeutige Grammatiken****Definition 9: Ableitungsbaum****Definition 10: Mehrdeutige Grammatik****2.3.2 Präzidenz und Assoziativität****Definition 11: Assoziativität****Definition 12: Präzidenz****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 13) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

**Definition 13: 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 Regeln 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 14) genannt.

**Definition 14: 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 15). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

#### Definition 15: 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.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>1</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. **Zahl** und **Bezeichner**.

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>2</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>3</sup>, unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

<sup>1</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.

<sup>2</sup>Die Programmiersprache Python erlaubt es z.B. diesen Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

<sup>3</sup>Man nennt das auch einem **Lookahead** von 1

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 von Variablen, Konstanten und Funktionen** die Symbole<sup>a</sup>.

<sup>a</sup>Das ist der Grund, warum die Tabelle, in der Informationen zu Identifiern gespeichert werden aus Kapitel 3 Symboltabelle genannt wird.

### Definition 16: Literal

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

## 2.5 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik**  $G_{Parse}$  notwendig, um die 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 entstprechende schließende Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** (Definition 17) als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung nach 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** relevant.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 20) bezeichnet. Zum Schluss der **Syntaktischen Analyse** ist der Inputstring in **Abstrakte Syntax** (Definition 25) übersetzt.

### Definition 17: Parser

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

**17.1:** *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 **Parzen** 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 17.1 verwendet.

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

In Bezug auf Compilerbau hat ein **Parser** meist die Aufgabe aus einem **Inputstring** einen **Derivation Tree** (Definition 22) zu generieren.

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 17 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.

Ein **Parser** ist aber auch ein erweiterter **Recognizer**, denn zum einen hat der **Parser** die Aufgabe eines **Recognizers** (Definition 18), nämlich zu überprüfen, ob ein Inputstring sich den Regeln der **Konkreten Syntax** ableiten lässt und des Weiteren wendet ein **Parser** auch **Aktions** an, um währenddessen einen **Derivation Tree** zu generieren.

#### Definition 18: Recognizer

#### Definition 19: Aktion

#### Definition 20: Konkrete Syntax

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

*Ein Programm in seiner Textrepräsentation, wie es in einer Textdatei nach den Regeln 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 21: Node

**Definition 22: Derivation Tree (bzw. Parse Tree)**

*Compilerinterne Darstellung* eines Programs in als Baumdatenstruktur, in der (Nichtterminalsymbole) die **Inneren Knoten** des Baumes bilden und **Terminalsymbole** die **Blätter** des Baumes bilden.

, (welches) aus einem **Inputstring**, der in **Konkreter Syntax** angegeben ist generiert wurde, welche Form in Compiler.

*Inside the compiler, we use abstract syntax trees (ASTs) to represent programs in a way that efficiently supports the operations that the compiler needs to perform*

**Definition 23: Transformer****Definition 24: Visitor****Definition 25: Abstrakte Syntax**

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

**Definition 26: Abstrakte Syntax Tree**

*Compilerinterne Darstellung* eines Programs, welches **TODO** die es erlaubt die Operationen, die der Compiler ausführen muss, möglichst effizient auszuführen.<sup>a</sup>

Im Gegensatz zum **Derivation Tree**, sind im **Abstrakt Syntax Tree** alle Nodes

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

## 2.6 Code Generierung

**Definition 27: Pass**

## 2.7 Fehlermeldungen

**Definition 28: Fehlermeldung**

# 3 Implementierung

## 3.1 Lexikalische Analyse

### 3.1.1 Verwendung von Lark

### 3.1.2 Basic Parser

## 3.2 Syntaktische Analyse

### 3.2.1 Verwendung von Lark

### 3.2.2 Umsetzung von Präzidenz

Die PicoC Sprache hat dieselben Präzidenzregeln implementiert, wie die Sprache C<sup>1</sup>. Die Präzidenzregeln von PicoC sind in Tabelle 3.2.2 aufgelistet.

Präzidenz	Operator	Beschreibung	Assoziativität
1	a() a[] a.b	Funktionsaufruf Indezzugriff 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

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

### 3.2.3 Derivation Tree Generierung

### 3.2.4 Early Parser

### 3.2.5 Derivation Tree Vereinfachung

### 3.2.6 Abstrakt Syntax Tree Generierung

ASTNode

PicoC Nodes

RETI Nodes

## 3.3 Code Generierung

### 3.3.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

Definition 29: Symboltabelle

RETI-Blocks Pass

RETI-Patch Pass

RETI Pass

### 3.3.2 Umsetzung von Pointern und Arrays

### 3.3.3 Umsetzung von Structs

### 3.3.4 Umsetzung von Funktionen

### 3.3.5 Umsetzung kleinerer Details

## 3.4 Fehlermeldungen

### 3.4.1 Error Handler



---

---

# 4 Ergebnisse und Ausblick

4.1 Funktionsumfang

4.2 Qualitätskontrolle

4.3 Kommentierter Kompiliervorgang

4.4 Erweiterungsideen

---

---

# 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).
- *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).