

---

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>Implementierung</b>	<b>8</b>
1.1	Architektur . . . . .	8
1.2	Lexikalische Analyse . . . . .	10
1.2.1	Verwendung von Lark . . . . .	10
1.2.2	Basic Parser . . . . .	11
1.3	Syntaktische Analyse . . . . .	11
1.3.1	Verwendung von Lark . . . . .	11
1.3.2	Umsetzung von Präzidenz . . . . .	12
1.3.3	Derivation Tree Generierung . . . . .	13
1.3.4	Early Parser . . . . .	13
1.3.5	Derivation Tree Vereinfachung . . . . .	13
1.3.6	Abstrakt Syntax Tree Generierung . . . . .	13
1.3.6.1	ASTNode . . . . .	13
1.3.6.2	PicoC Nodes . . . . .	13
1.3.6.3	RETI Nodes . . . . .	13

---

---

# Abbildungsverzeichnis

1.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	8
1.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	9
1.3	Architektur mit allen Passes ausgeschrieben . . . . .	9

---

---

# Codeverzeichnis

---

---

# Tabellenverzeichnis

1.1 Präzidenzregeln von PicoC . . . . .	12
---	----

---

---

# Definitionsverzeichnis

---

---

# Grammatikverzeichnis

1.2.1 Konkrete Syntax des Lexers . . . . .	10
1.3.1 Konkrete Syntax des Parsers, Teil 1 . . . . .	11
1.3.2 Konkrete Syntax des Parsers, Teil 2 . . . . .	12
1.2.2 $\lambda$ calculus syntax . . . . .	14
1.2.3 Advanced capabilities of <code>grammar.sty</code> . . . . .	14



# 1 Implementierung

## 1.1 Architektur

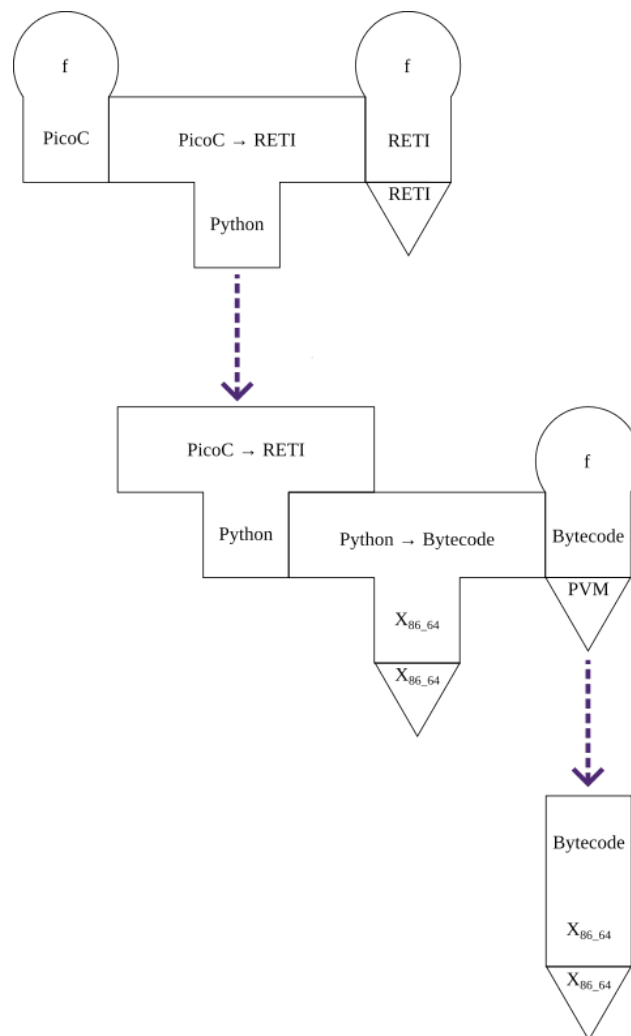


Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

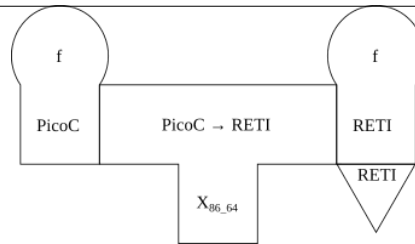


Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

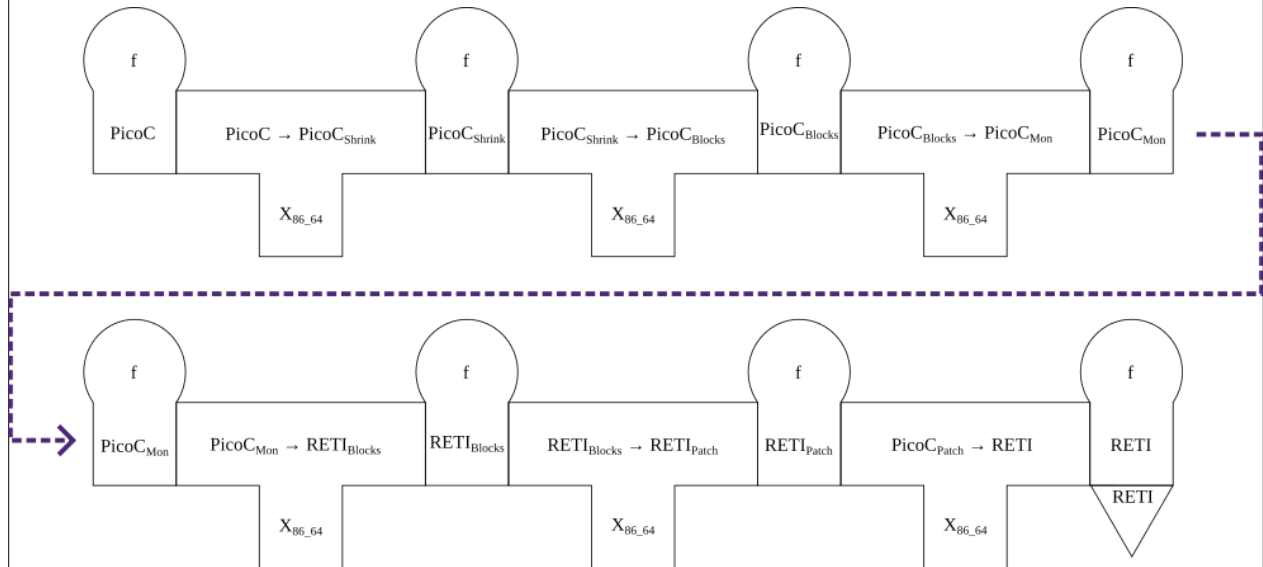


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

## 1.2 Lexikalische Analyse

### 1.2.1 Verwendung von Lark

<i>COMMENT</i>	::=	<code>"/" /[\n]*/   "/" /(.   \n)*?/ "*" /</code>	<i>L_Comment</i>
<i>RETI_COMMENT.2</i>	::=	<code>"/" " " ?" #" /[\n]*/</code>	
<i>DIG_NO_0</i>	::=	<code>"1"   "2"   "3"   "4"   "5"</code> <code>  "6"   "7"   "8"   "9"</code>	<i>L_Arith</i>
<i>DIG_WITH_0</i>	::=	<code>"0"   DIG_NO_0</code>	
<i>NUM</i>	::=	<code>"0"   DIG_NO_0DIG_WITH_0*</code>	
<i>ASCII_CHAR</i>	::=	<code>"_ " . " ~ "</code>	
<i>CHAR</i>	::=	<code>"' " ASCII_CHAR "' "</code>	
<i>FILENAME</i>	::=	<code>ASCII_CHAR + ".picoc"</code>	
<i>LETTER</i>	::=	<code>"a" . "z"   "A" . "Z"</code>	
<i>NAME</i>	::=	<code>(LETTER   "_")</code> <code>(LETTER — DIG_WITH_0 — "_")*</code>	
<i>name</i>	::=	<code>NAME   INT_NAME   CHAR_NAME</code> <code>  VOID_NAME</code>	
<i>NOT</i>	::=	<code>" ~ "</code>	
<i>REF_AND</i>	::=	<code>"&amp;"</code>	
<i>un_op</i>	::=	<code>SUB_MINUS   LOGIC_NOT   NOT</code> <code>  MUL_DEREF_PNTR   REF_AND</code>	
<i>MUL_DEREF_PNTR</i>	::=	<code>"*"</code>	
<i>DIV</i>	::=	<code>"/"</code>	
<i>MOD</i>	::=	<code>"%"</code>	
<i>pre1_op</i>	::=	<code>MUL_DEREF_PNTR   DIV   MOD</code>	
<i>ADD</i>	::=	<code>"+"</code>	
<i>SUB_MINUS</i>	::=	<code>"-"</code>	
<i>prec2_op</i>	::=	<code>ADD   SUB_MINUS</code>	
<i>LT</i>	::=	<code>"&lt;"</code>	<i>L_Logic</i>
<i>LTE</i>	::=	<code>"&lt;="</code>	
<i>GT</i>	::=	<code>"&gt;"</code>	
<i>GTE</i>	::=	<code>"&gt;="</code>	
<i>rel_op</i>	::=	<code>LT   LTE   GT   GTE</code>	
<i>EQ</i>	::=	<code>"=="</code>	
<i>NEQ</i>	::=	<code>"!="</code>	
<i>eq_op</i>	::=	<code>EQ   NEQ</code>	
<i>LOGIC_NOT</i>	::=	<code>"!"</code>	
<i>INT_DT.2</i>	::=	<code>"int"</code>	
<i>INT_NAME.3</i>	::=	<code>"int" (LETTER   DIG_WITH_0   "_")+</code>	<i>L_Assign_Alloc</i>
<i>CHAR_DT.2</i>	::=	<code>"char"</code>	
<i>CHAR_NAME.3</i>	::=	<code>"char" (LETTER   DIG_WITH_0   "_")+</code>	
<i>VOID_DT.2</i>	::=	<code>"void"</code>	
<i>VOID_NAME.3</i>	::=	<code>"void" (LETTER   DIG_WITH_0   "_")+</code>	
<i>prim_dt</i>	::=	<code>INT_DT   CHAR_DT   VOID_DT</code>	

Grammar 1.2.1: Konkrete Syntax des Lexers

## 1.2.2 Basic Parser

## 1.3 Syntaktische Analyse

### 1.3.1 Verwendung von Lark

In 1.3.1

<i>prim_exp</i>	::=	<i>name</i>   <i>NUM</i>   <i>CHAR</i>   "(" <i>logic_or</i> ")"	<i>L_Arith</i> +
<i>post_exp</i>	::=	<i>array_subscr</i>   <i>struct_attr</i>   <i>fun_call</i>   <i>input_exp</i>   <i>print_exp</i>   <i>prim_exp</i>	<i>L_Array</i> + <i>L_Pntr</i> +
<i>un_exp</i>	::=	<i>un_opun_exp</i>   <i>post_exp</i>	<i>L_Struct</i> + <i>L_Fun</i>
<i>input_exp</i>	::=	"input" "(" "	<i>L_Arith</i>
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i>   <i>un_exp</i>	
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i>   <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i>   <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i>   <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i>   <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i>   <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_oprel_exp</i>   <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i>   <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> "  " <i>logic_and</i>   <i>logic_and</i>	
<i>type_spec</i>	::=	<i>prim_dt</i>   <i>struct_spec</i>	<i>L_Assign_Alloc</i>
<i>alloc</i>	::=	<i>type_spec</i> <i>pntr_decl</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	
<i>initializer</i>	::=	<i>logic_or</i>   <i>array_init</i>   <i>struct_init</i>	
<i>init_stmt</i>	::=	<i>alloc</i> "=" <i>initializer</i> ";"	
<i>const_init_stmt</i>	::=	"const" <i>type_spec</i> <i>name</i> "=" <i>NUM</i> ";"	
<i>pntr_deg</i>	::=	"*" *	<i>L_Pntr</i>
<i>pntr_decl</i>	::=	<i>pntr_deg</i> <i>array_decl</i>   <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]" ) *	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name</i> <i>array_dims</i>   "(" <i>pntr_decl</i> ")" <i>array_dims</i>	
<i>array_init</i>	::=	"{" <i>initializer</i> ("," <i>initializer</i> ) * "}"	
<i>array_subscr</i>	::=	<i>post_exp</i> "[" <i>logic_or</i> "]"	
<i>struct_spec</i>	::=	"struct" <i>name</i>	<i>L_Struct</i>
<i>struct_params</i>	::=	( <i>alloc</i> ";" ) +	
<i>struct_decl</i>	::=	"struct" <i>name</i> "{" <i>struct_params</i> "}"	
<i>struct_init</i>	::=	"{" " " <i>name</i> "=" <i>initializer</i> ("," " " <i>name</i> "=" <i>initializer</i> ) * "}"	
<i>struct_attr</i>	::=	<i>post_exp</i> "." <i>name</i>	
<i>if_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_If_Else</i>
<i>if_else_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i> "else" <i>exec_part</i>	
<i>while_stmt</i>	::=	"while" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_Loop</i>
<i>do_while_stmt</i>	::=	"do" <i>exec_part</i> "while" "(" <i>logic_or</i> ")" ";"	

Grammar 1.3.1: Konkrete Syntax des Parsers, Teil 1

<i>decl_exp_stmt</i>	::=	<i>alloc</i> ";"	<i>L_Stmt</i>
<i>decl_direct_stmt</i>	::=	<i>assign_stmt</i>   <i>init_stmt</i>   <i>const_init_stmt</i>	
<i>decl_part</i>	::=	<i>decl_exp_stmt</i>   <i>decl_direct_stmt</i>   <i>RETI_COMMENT</i>	
<i>compound_stmt</i>	::=	"{" <i>exec_part</i> * "}"	
<i>exec_exp_stmt</i>	::=	<i>logic_or</i> ";"	
<i>exec_direct_stmt</i>	::=	<i>if_stmt</i>   <i>if_else_stmt</i>   <i>while_stmt</i>   <i>do_while_stmt</i>   <i>assign_stmt</i>   <i>fun_return_stmt</i>	
<i>exec_part</i>	::=	<i>compound_stmt</i>   <i>exec_exp_stmt</i>   <i>exec_direct_stmt</i>   <i>RETI_COMMENT</i>	
<i>decl_exec_stmts</i>	::=	<i>decl_part</i> * <i>exec_part</i> *	
<i>fun_args</i>	::=	[ <i>logic_or</i> ("," <i>logic_or</i> )*]	<i>L_Fun</i>
<i>fun_call</i>	::=	<i>name</i> (" <i>fun_args</i> ")	
<i>fun_return_stmt</i>	::=	"return" [ <i>logic_or</i> ];	
<i>fun_params</i>	::=	[ <i>alloc</i> ("," <i>alloc</i> )*]	
<i>fun_decl</i>	::=	<i>type_spec</i> <i>pntr_deg</i> <i>name</i> (" <i>fun_params</i> ")	
<i>fun_def</i>	::=	<i>type_spec</i> <i>pntr_deg</i> <i>name</i> (" <i>fun_params</i> ") " {" <i>decl_exec_stmts</i> } "	
<i>decl_def</i>	::=	( <i>struct_decl</i>   <i>fun_decl</i> );   <i>fun_def</i>	<i>L_File</i>
<i>decls_defs</i>	::=	<i>decl_def</i> *	
<i>file</i>	::=	<i>FILENAME</i> <i>decls_defs</i>	

Grammar 1.3.2: Konkrete Syntax des Parsers, Teil 2

### 1.3.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**<sup>1</sup>. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 1.1 aufgelistet.

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

Tabelle 1.1: Präzidenzregeln von PicoC

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

### **1.3.3 Derivation Tree Generierung**

### **1.3.4 Early Parser**

### **1.3.5 Derivation Tree Vereinfachung**

### **1.3.6 Abstrakt Syntax Tree Generierung**

#### **1.3.6.1 ASTNode**

#### **1.3.6.2 PicoC Nodes**

#### **1.3.6.3 RETI Nodes**

$$\begin{array}{lll}
 T & ::= & \mathcal{V} \quad \textit{Variable} \\
 & | & (\mathcal{T} \mathcal{T}) \quad \textit{Application} \\
 & | & \lambda \mathcal{V} \cdot \mathcal{T} \quad \textit{Abstraction} \\
 V & ::= & x, y, \dots \quad \textit{Variables}
 \end{array}$$
Grammar 1.2.2:  $\lambda$  calculus syntax
$$\begin{array}{lll}
 A & ::= & \mathcal{T} \mid \mathcal{V} \quad \textit{Multiple option on a single line} \\
 & | & \mathcal{A} \quad \textit{Highlighted form} \\
 & | & \mathcal{B} \mid \mathcal{C} \quad \textit{Downplayed form} \\
 & | & \textcolor{red}{\mathcal{A}} \mid \mathcal{B} \quad \textit{Emphasize part of the line}
 \end{array}$$
Grammar 1.2.3: Advanced capabilities of `grammar.sty`

---

---

# 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).