

---

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
1.3.3.1	Early Parser . . . . .	13
1.3.4	Derivation Tree Vereinfachung . . . . .	13
1.3.5	Abstrakt Syntax Tree Generierung . . . . .	13
1.3.5.1	PicoC Nodes . . . . .	13
1.3.5.2	RETI Nodes . . . . .	13
1.3.5.3	Kompositionen mit besonderer Bedeutung . . . . .	13
1.3.5.4	Abstrakte Syntax . . . . .	13

---

---

# Abbildungsverzeichnis

1.1	Cross-Compiler Kompilervorgang ausgeschrieben . . . . .	8
1.2	Cross-Compiler Kompilervorgang 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 in EBNF . . . . .	10
1.3.1 Konkrete Syntax des Parsers in EBNF, Teil 1 . . . . .	11
1.3.2 Konkrete Syntax des Parsers in EBNF, Teil 2 . . . . .	12
1.3.3 Abstrakte Syntax für $L_{PlocC}$ . . . . .	14
1.3.4 Abstrakte Syntax für $L_{PicoC\_Blocks}$ . . . . .	15
1.3.5 Abstrakte Syntax für $L_{PicoC\_Mon}$ . . . . .	15
1.3.6 Abstrakte Syntax für $L_{RETI\_Blocks}$ . . . . .	15
1.3.7 Abstrakte Syntax für $L_{RETI\_Patch}$ . . . . .	15
1.3.8 Abstrakte Syntax für $L_{RETI}$ . . . . .	16
1.2.2 $\lambda$ calculus syntax . . . . .	17
1.2.3 Advanced capabilities of <code>grammar.sty</code> . . . . .	17



# 1 Implementierung

## 1.1 Architektur

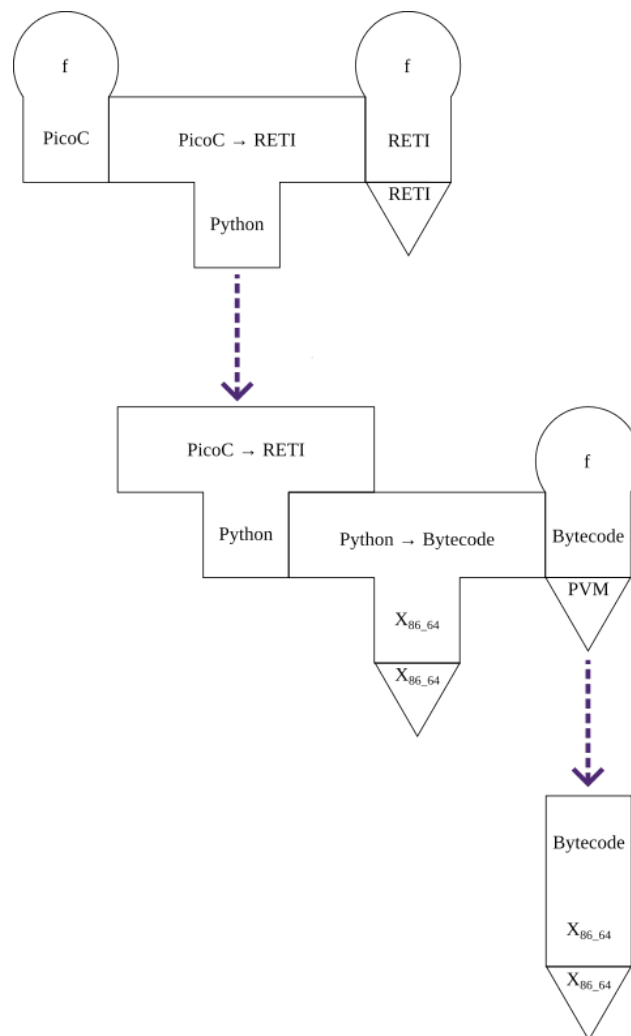


Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

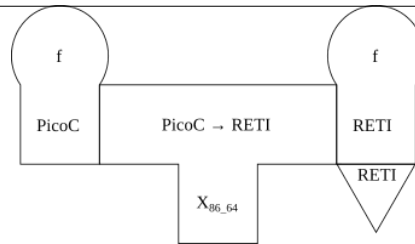


Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

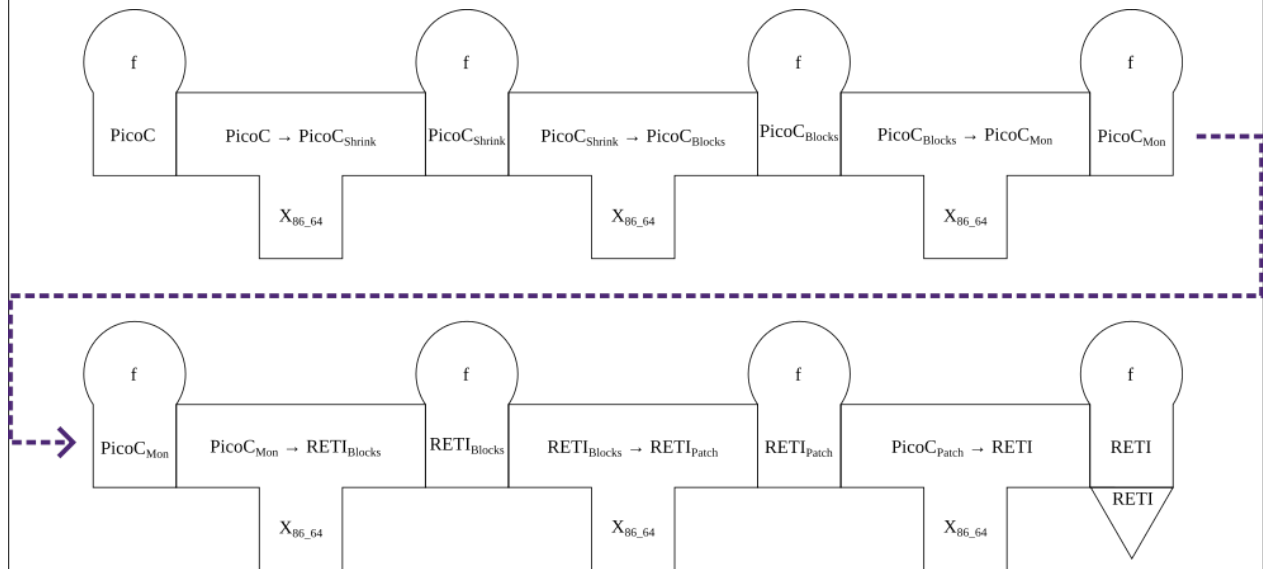


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

## 1.2 Lexikalische Analyse

### 1.2.1 Verwendung von Lark

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

Grammar 1.2.1: Konkrete Syntax des Lexers in EBNF

## 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 in EBNF, 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 in EBNF, 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	Rechts, dann links ←
11	<i>a  b</i>	Logisches ODER	
12	<i>a=b</i>	Zuweisung	
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**

#### **1.3.3.1 Early Parser**

### **1.3.4 Derivation Tree Vereinfachung**

### **1.3.5 Abstrakt Syntax Tree Generierung**

#### **1.3.5.1 PicoC Nodes**

#### **1.3.5.2 RETI Nodes**

#### **1.3.5.3 Kompositionen mit besonderer Bedeutung**

#### **1.3.5.4 Abstrakte Syntax**

<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>BinOp</i> ( <i>&lt;exp&gt;</i> , <i>&lt;bin_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>UnOp</i> ( <i>&lt;un_op&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Call</i> ( <i>Name('input')</i> , <i>None</i> )	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>Call</i> ( <i>Name('print')</i> , <i>&lt;exp&gt;</i> )	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom</i> ( <i>&lt;exp&gt;</i> , <i>&lt;rel&gt;</i> , <i>&lt;exp&gt;</i> )   <i>ToBool</i> ( <i>&lt;exp&gt;</i> )	
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>	
<i>assign_lhs</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>&lt;rel_loc&gt;</i>	
<i>exp_stmts</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )	
<i>stmt</i>	::=	<i>Assign</i> ( <i>&lt;assign_lhs&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Exp</i> ( <i>&lt;exp_stmts&gt;</i> )	
<i>datatype</i>	::=	<i>PntrDecl</i> ( <i>Num(str)</i> , <i>&lt;datatype&gt;</i> )	<i>L_Pntr</i>
<i>deref_loc</i>	::=	<i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )   <i>&lt;ref_loc&gt;</i>	
<i>ref_loc</i>	::=	<i>Name(str)</i>   <i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )	
<i>exp</i>	::=	<i>Deref</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Ref</i> ( <i>&lt;ref_loc&gt;</i> )	
<i>datatype</i>	::=	<i>ArrayDecl</i> ( <i>Num(str)</i> +, <i>&lt;datatype&gt;</i> )	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> ( <i>&lt;deref_loc&gt;</i> , <i>&lt;exp&gt;</i> )   <i>Array</i> ( <i>&lt;exp&gt;</i> +)	
<i>datatype</i>	::=	<i>StructSpec</i> ( <i>Name(str)</i> )	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> ( <i>&lt;ref_loc&gt;</i> , <i>Name(str)</i> )   <i>Struct</i> ( <i>Assign</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> ) +)	
<i>decl_def</i>	::=	<i>StructDecl</i> ( <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> ) +)	
<i>stmt</i>	::=	<i>If</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>IfElse</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *, <i>&lt;stmt&gt;</i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)   <i>DoWhile</i> ( <i>&lt;exp&gt;</i> , <i>&lt;stmt&gt;</i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	<i>L_Fun</i>
<i>exp_stmts</i>	::=	<i>Call</i> ( <i>Name(str)</i> , <i>&lt;exp&gt;</i> *)	
<i>stmt</i>	::=	<i>Return</i> ( <i>&lt;exp&gt;</i> )	
<i>decl_def</i>	::=	<i>FunDecl</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*)   <i>FunDef</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc</i> ( <i>Writeable()</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )*, <i>&lt;stmt&gt;</i> *)	
<i>file</i>	::=	<i>File</i> ( <i>Name(str)</i> , <i>&lt;decl_def&gt;</i> *)	<i>L_File</i>

Grammar 1.3.3: Abstrakte Syntax für  $L_{PiocC}$

<i>decl_def</i>	::=	<i>FunDef</i> ( $\langle datatype \rangle$ , <i>Name</i> ( <i>str</i> ), <i>Alloc</i> ( <i>Writable</i> () , $\langle datatype \rangle$ , <i>Name</i> ( <i>str</i> ))* , $\langle block \rangle$ *)	<i>L_Fun</i>
<i>block</i>	::=	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), $\langle stmt \rangle$ *)	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>Goto</i> ( <i>Name</i> ( <i>str</i> ))   <i>NewStackframe</i> ( <i>Name</i> () , <i>Goto</i> ( <i>str</i> ))   <i>RemoveStackframe</i> ()   <i>SetScope</i> ( <i>Name</i> ( <i>str</i> ))   <i>SingleLineComment</i> ( <i>str</i> , <i>str</i> )	
Grammar 1.3.4: Abstrakte Syntax für <i>L<sub>PicoC_Blocks</sub></i>			
<i>ref_loc</i>	::=	<i>Tmp</i> ( <i>Num</i> ( <i>str</i> ))   <i>StackRead</i> ( <i>Num</i> ( <i>str</i> ))   <i>StackWrite</i> ( <i>Num</i> ( <i>str</i> ))   <i>GlobalRead</i> ( <i>Num</i> ( <i>str</i> ))   <i>GlobalWrite</i> ( <i>Num</i> ( <i>str</i> ))	<i>L_Assign_Alloc</i>
<i>error_data</i>	::=	$\langle exp \rangle$   <i>Pos</i> ( <i>Num</i> ( <i>str</i> ), <i>Num</i> ( <i>str</i> ))	
<i>exp</i>	::=	<i>Stack</i> ( <i>Num</i> ( <i>str</i> ))   <i>Ref</i> ( $\langle ref\_loc \rangle$ , $\langle datatype \rangle$ , $\langle error\_data \rangle$ )	
<i>stmt</i>	::=	<i>Exp</i> ( $\langle exp \rangle$ )   <i>Assign</i> ( <i>Alloc</i> ( <i>Writable</i> () , <i>StructSpec</i> ( <i>Name</i> ( <i>str</i> )), <i>Name</i> ( <i>str</i> )), <i>Struct</i> ( <i>Assign</i> ( <i>Name</i> ( <i>str</i> ), $\langle exp \rangle$ ) + , $\langle datatype \rangle$ ))   <i>Assign</i> ( <i>Alloc</i> ( <i>Writable</i> () , <i>ArrayDecl</i> ( <i>Num</i> ( <i>str</i> ) + , $\langle datatype \rangle$ ), <i>Name</i> ( <i>str</i> )), <i>Array</i> ( $\langle exp \rangle$ + , $\langle datatype \rangle$ ))	
<i>symbol_table</i>	::=	<i>SymbolTable</i> ( $\langle symbol \rangle$ )	<i>L_Symbol_Table</i>
<i>symbol</i>	::=	<i>Symbol</i> ( $\langle type\_qual \rangle$ , $\langle datatype \rangle$ , $\langle name \rangle$ , $\langle val \rangle$ , $\langle pos \rangle$ , $\langle size \rangle$ )	
<i>type_qual</i>	::=	<i>Empty</i> ()	
<i>datatype</i>	::=	<i>BuiltIn</i> ()   <i>SelfDefined</i> ()	
<i>name</i>	::=	<i>Name</i> ( <i>str</i> )	
<i>val</i>	::=	<i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	
<i>pos</i>	::=	<i>Pos</i> ( <i>Num</i> ( <i>str</i> ), <i>Num</i> ( <i>str</i> ))   <i>Empty</i> ()	
<i>size</i>	::=	<i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	
Grammar 1.3.5: Abstrakte Syntax für <i>L<sub>PicoC_Mon</sub></i>			
<i>program</i>	::=	<i>Program</i> ( <i>Name</i> ( <i>str</i> ), $\langle block \rangle$ *)	<i>L_Program</i>
<i>exp_stmts</i>	::=	<i>Goto</i> ( <i>str</i> )	<i>L_Blocks</i>
<i>instrs_before</i>	::=	<i>Num</i> ( <i>str</i> )	
<i>num_instrs</i>	::=	<i>Num</i> ( <i>str</i> )	
<i>block</i>	::=	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), $\langle instr \rangle$ *, $\langle instrs\_before \rangle$ , $\langle num\_instrs \rangle$ )	
<i>instr</i>	::=	<i>Goto</i> ( <i>Name</i> ( <i>str</i> ))	
Grammar 1.3.6: Abstrakte Syntax für <i>L<sub>RETI_Blocks</sub></i>			
<i>stmt</i>	::=	<i>Exit</i> ( <i>Num</i> ( <i>str</i> ))	
Grammar 1.3.7: Abstrakte Syntax für <i>L<sub>RETI_Patch</sub></i>			



<i>reg</i>	$::=$	<i>ACC()</i>   <i>IN1()</i>   <i>IN2()</i>   <i>PC()</i>   <i>SP()</i>   <i>BAF()</i>	<i>L_Program</i>
		<i>CS()</i>   <i>DS()</i>	
<i>arg</i>	$::=$	<i>Reg</i> ( <i>&lt;reg&gt;</i> )   <i>Num</i> ( <i>str</i> )	
<i>rel</i>	$::=$	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
		<i>Always()</i>   <i>NOp()</i>	
<i>op</i>	$::=$	<i>Add()</i>   <i>Addi()</i>   <i>Sub()</i>   <i>Subi()</i>   <i>Mult()</i>	
		<i>Multi()</i>   <i>Div()</i>   <i>Divi()</i>	
		<i>Mod()</i>   <i>Modi()</i>   <i>Oplus()</i>   <i>Oplusi()</i>   <i>Or()</i>	
		<i>Ori()</i>   <i>And()</i>   <i>Andi()</i>	
		<i>Load()</i>   <i>Loadin()</i>   <i>Loadi()</i>	
		<i>Store()</i>   <i>Storein()</i>   <i>Move()</i>	
<i>instr</i>	$::=$	<i>Instr</i> ( <i>&lt;op&gt;</i> , <i>&lt;arg&gt;</i> +)   <i>Jump</i> ( <i>&lt;rel&gt;</i> , <i>Num</i> ( <i>str</i> ))   <i>Int</i> ( <i>Num</i> ( <i>str</i> ))	
		<i>RTI()</i>   <i>Call</i> ( <i>Name</i> ('print'), <i>&lt;reg&gt;</i> )   <i>Call</i> ( <i>Name</i> ('input'), <i>&lt;reg&gt;</i> )	
		<i>SingleLineComment</i> ( <i>str</i> , <i>str</i> )	
<i>program</i>	$::=$	<i>Program</i> ( <i>Name</i> ( <i>str</i> ), <i>&lt;instr&gt;</i> *)	

Grammar 1.3.8: Abstrakte Syntax für *L<sub>RETI</sub>*

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