

---

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	Lexikalische Analyse	9
1.1.1	Konkrete Syntax für die Lexikalische Analyse	9
1.1.2	Basic Lexer	10
1.2	Syntaktische Analyse	10
1.2.1	Konkrete Syntax für die Syntaktische Analyse	10
1.2.2	Umsetzung von Präzidenz	11
1.2.3	Derivation Tree Generierung	12
1.2.3.1	Early Parser	12
1.2.3.2	Codebeispiel	12
1.2.4	Derivation Tree Vereinfachung	13
1.2.4.1	Visitor	13
1.2.4.2	Codebeispiel	13
1.2.5	Abstrakt Syntax Tree Generierung	15
1.2.5.1	PicoC-Knoten	15
1.2.5.2	RETI-Knoten	18
1.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	18
1.2.5.4	Abstrakte Syntax	20
1.2.5.5	Transformer	22
1.2.5.6	Codebeispiel	22
1.3	Code Generierung	23
1.3.1	Übersicht	23
1.3.2	Passes	24
1.3.2.1	PicoC-Shrink Pass	24
1.3.2.1.1	Codebeispiel	24
1.3.2.2	PicoC-Blocks Pass	26
1.3.2.2.1	Abstrakte Syntax	26
1.3.2.2.2	Codebeispiel	26
1.3.2.3	PicoC-Mon Pass	28
1.3.2.3.1	Abstrakte Syntax	28
1.3.2.3.2	Codebeispiel	29
1.3.2.4	RETI-Blocks Pass	31
1.3.2.4.1	Abstrakte Syntax	31
1.3.2.4.2	Codebeispiel	31
1.3.2.5	RETI-Patch Pass	34
1.3.2.5.1	Abstrakte Syntax	34
1.3.2.5.2	Codebeispiel	34
1.3.2.6	RETI Pass	36
1.3.2.6.1	Konkrete und Abstrakte Syntax	36
1.3.2.6.2	Codebeispiel	37

---

---

# Abbildungsverzeichnis

1.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	23
1.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	23
1.3	Architektur mit allen Passes ausgeschrieben . . . . .	24

---

---

# Codeverzeichnis

1.1	PicoC Code für Derivation Tree Generierung . . . . .	12
1.2	Derivation Tree nach Derivation Tree Generierung . . . . .	13
1.3	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	14
1.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert . . . . .	22
1.5	PicoC Code für Codebeispiel . . . . .	24
1.6	Abstract Syntax Tree für Codebeispiel . . . . .	25
1.7	PicoC Shrink Pass für Codebeispiel . . . . .	26
1.8	PicoC-Blocks Pass für Codebeispiel . . . . .	28
1.9	PicoC-Mon Pass für Codebeispiel . . . . .	31
1.10	RETI-Blocks Pass für Codebeispiel . . . . .	33
1.11	RETI-Patch Pass für Codebeispiel . . . . .	36
1.12	RETI Pass für Codebeispiel . . . . .	39

---

---

# Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC . . . . .	11
1.2	PicoC-Knoten Teil 1 . . . . .	15
1.3	PicoC-Knoten Teil 2 . . . . .	16
1.4	PicoC-Knoten Teil 2 . . . . .	17
1.5	RETI-Knoten . . . . .	18
1.6	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung . . . . .	19

---

---

# Definitionsverzeichnis

1.1	Token-Knoten . . . . .	17
1.2	Container-Knoten . . . . .	18
1.3	Symboltabelle . . . . .	28

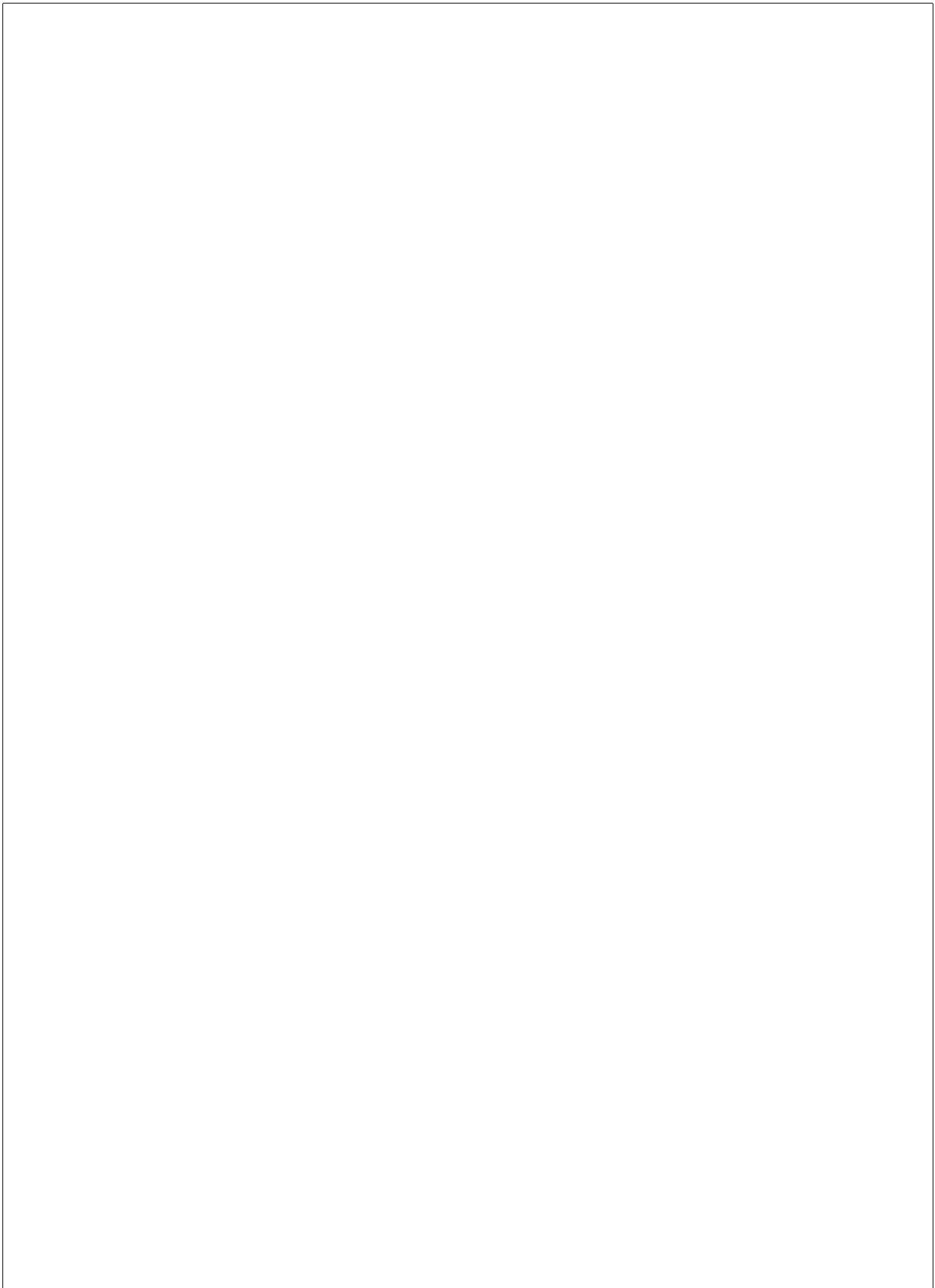
---

---

# Grammatikverzeichnis

1.1.1 Konkrete Syntax für die Lexikalische Analyse in EBNF . . . . .	9
1.2.1 Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1 . . . . .	10
1.2.2 Konkrete Syntax für die Syntaktische Analyse in EBNF, Teil 2 . . . . .	11
1.2.3 Abstrakte Syntax für $L_{PioC}$ . . . . .	21
1.3.1 Abstrakte Syntax für $L_{PicoC\_Blocks}$ . . . . .	26
1.3.2 Abstrakte Syntax für $L_{PicoC\_Mon}$ . . . . .	28
1.3.3 Abstrakte Syntax für $L_{RETI\_Blocks}$ . . . . .	31
1.3.4 Abstrakte Syntax für $L_{RETI\_Patch}$ . . . . .	34
1.3.5 Konkrete Syntax für $L_{RETI\_Lex}$ . . . . .	36
1.3.6 Konkrete Syntax für $L_{RETI\_Parse}$ . . . . .	37
1.3.7 Abstrakte Syntax für $L_{RETI}$ . . . . .	37





# 1 Implementierung

## 1.1 Lexikalische Analyse

### 1.1.1 Konkrete Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	"//"/[ $\backslash$ n]*/   "/"*//(. $\backslash$ n)*?/"*/	<i>L_Comment</i>
<i>RETI_COMMENT.2</i>	::=	"//""?"#"/[ $\backslash$ 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>prec1_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>	

### 1.1.2 Basic Lexer

## 1.2 Syntaktische Analyse

### 1.2.1 Konkrete Syntax für die Syntaktische Analyse

In 1.2.1

<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 1.2.1: Konkrete Syntax Syntaktische Analyse 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 1.2.2: Konkrete Syntax für die Syntaktische Analyse in EBNF, Teil 2

## 1.2.2 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	Rechts, dann links ←
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).

### 1.2.3 Derivation Tree Generierung

#### 1.2.3.1 Early Parser

#### 1.2.3.2 Codebeispiel

```

1 struct st {int *(*attr)[5][6];};
2
3 void main() {
4     struct st *(*var)[3][2];
5 }

```

Code 1.1: PicoC Code für Derivation Tree Generierung

```

1 file
2   ./example_dt_simple_ast_gen_array_decl_and_alloc.dt
3 decls_defs
4   decl_def
5     struct_decl
6       name st
7       struct_params
8       alloc
9       type_spec
10      prim_dt int
11      pntr_decl
12      pntr_deg *
13      array_decl
14      pntr_decl
15      pntr_deg *
16      array_decl
17      name attr
18      array_dims
19      array_dims
20      5
21      6
22  decl_def
23  fun_def
24    type_spec
25    prim_dt void
26    pntr_deg
27    name main
28    fun_params
29    decl_exec_stmts
30    decl_part
31    decl_exp_stmt
32    alloc
33    type_spec
34    struct_spec
35    name st
36    pntr_decl
37    pntr_deg *
38    array_decl
39    pntr_decl
40    pntr_deg *

```

```

41         array_decl
42         name var
43         array_dims
44     array_dims
45     3
46     2

```

Code 1.2: Derivation Tree nach Derivation Tree Generierung

## 1.2.4 Derivation Tree Vereinfachung

### 1.2.4.1 Visitor

### 1.2.4.2 Codebeispiel

Beispiel aus Subkapitel 1.2.3.2 wird fortgeführt.

```

1 file
2   ./example_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3 decls_defs
4   decl_def
5     struct_decl
6     name st
7     struct_params
8     alloc
9     ptr_decl
10    ptr_deg *
11    array_decl
12    array_dims
13    5
14    6
15    ptr_decl
16    ptr_deg *
17    array_decl
18    array_dims
19    type_spec
20    prim_dt int
21  name attr
22 decl_def
23 fun_def
24   type_spec
25   prim_dt void
26   ptr_deg
27   name main
28   fun_params
29   decl_exec_stmts
30   decl_part
31   decl_exp_stmt
32   alloc
33   ptr_decl
34   ptr_deg *
35   array_decl
36   array_dims

```

```
37         3
38         2
39     pntr_decl
40     pntr_deg *
41     array_decl
42     array_dims
43     type_spec
44     struct_spec
45     name st
46 name var
```

Code 1.3: Derivation Tree nach Derivation Tree Vereinfachung

## 1.2.5 Abstrakt Syntax Tree Generierung

### 1.2.5.1 PicoC-Knoten

PicoC-Knoten	Beschreibung
Name()	Ein <b>Bezeichner</b> , aber da es keine gute Kurzform für Identifier() (englisches Wort für Bezeichner) gibt, wurde dieser Knoten Name() genannt.
Num()	Eine <b>Zahl</b> .
Char()	Ein <b>Zeichen</b> der <b>ASCII-Zeichenkodierung</b> .
Minus(), Not(), DerefOp(), RefOp(), LogicNot()	Die <b>unären Operatoren</b> un_op: -, ~, *, &a !a.
Add(), Sub(), Mul(), Div(), Mod(), Oplus(), And(), Or(), LogicAnd(), LogicOr()	Die <b>binären Operatoren</b> bin_op: a + b, a - b, a * b, a / b, a % b, a ^ b, a & b, a   b, a && b, a    b.
Eq(), NEq(), Lt(), LtE(), Gt(), GtE()	Die <b>Relationen</b> rel: a == b, a != b, a < b, a <= b, a > b, a >= b.
Const(), Writeable()	Die <b>Type Qualifier</b> type_qual: const, was für ein <b>nicht beschreibbare Konstante</b> steht und das <b>nicht</b> Angeben von const, was für einen <b>beschreibbare</b> Variable steht.
IntType(), CharType(), VoidType()	Die <b>Type Specifier</b> für <b>Primitiven Datentypen</b> , die in der Abstrakten Syntax, um eine intuitive Bezeichnung zu haben einfach nur unter <b>Datentypen</b> eingeordnet werden: int, char, void.
Placeholder()	<b>Platzhalter</b> für einen Knoten, der diesen später <b>ersetzt</b> .
BinOp(exp, bin_op, exp)	Container für eine <b>binäre Operation</b> mit 2 Expressions.
UnOp(un_op, exp)	Container für eine <b>unäre Operation</b> mit einer Expression.
Exit(num)	Container für einen <b>Exit Code</b> , der vor der Beendigung in das ACC Register geschrieben wird und steht für die <b>Beendigung</b> des laufenden Programmes.
Atom(exp, rel, exp)	Container für eine <b>binäre Relation</b> mit 2 Expressions.
ToBool(exp)	Container für einen <b>Arithmetischen Ausdruck</b> , wie z.B. 1 + 3 oder einfach nur 3, der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis $x > 1$ auf 1 abgebildet wird.
Alloc(type_qual, datatype, name, local_var_or_param)	<b>Container</b> für eine <b>Allokation</b> mit den notwendigen Knoten type_qual, datatype und name, die alle für einen Eintrag in der <b>Symboltabelle</b> notwendigen Informationen enthalten. Zudem kann es ein <b>verstecktes Attribut</b> local_var_or_param besitzen, dass die Information trägt, ob es sich bei der <b>Variable</b> um eine <b>Lokale Variable</b> oder einen <b>Parameter</b> handelt.
Assign(lhs, exp)	Container für eine <b>Zuweisung</b> . Wobei lhs ein Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name) oder Name('var') sein kann und exp ein beliebiger <b>Logischer Ausdruck</b> sein kann.
Exp(exp, datatype, error_data)	Container für einen <b>beliebigen Ausdruck</b> , dessen Ergebnis auf den <b>Stack</b> soll. Zudem besitzt er 2 <b>versteckte Attribute</b> , wobei datatype im <b>RETI Blocks Pass</b> wichtig ist und error_data für <b>Fehlermeldungen</b> wichtig ist.
Stack(num)	Container, der für das <b>temporäre</b> Ergebnis einer Berechnung, das num Speicherzellen relativ zum <b>Stackpointer Register</b> SP steht.
Stackframe(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum <b>Begin-Aktive-Funktion Register</b> BAF steht.

Tabelle 1.2: PicoC-Knoten Teil 1



PiocC-Knoten	Beschreibung
Global(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum <b>Datensegment Register</b> DS steht.
StackMalloc(num)	Container, der für das <b>Allokieren</b> von num Speicherzellen auf dem <b>Stack</b> steht.
PntrDecl(num, datatype)	Container, der für den <b>Pointerdatatype</b> steht, wobei das <b>Attribut</b> num die <b>Anzahl zusammengefasster Pointer</b> angibt und datatype der Datentyp ist, auf den der oder die <b>Pointer</b> zeigen.
Ref(exp, datatype, error_data)	Container, der für die Anwendung des <b>Referenz-Operators</b> steht. Zudem besitzt er 2 <b>versteckte Attribute</b> , wobei datatype im <b>RETI Blocks Pass</b> wichtig ist und error_data für <b>Fehlermeldungen</b> wichtig ist.
Deref(lhs, exp)	Container für den <b>Indexzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> . Wobei exp1 eine angehängte weitere Subscr(exp1, exp2), Deref(exp1, exp2), Attr(exp, name) oder ein Name('var') sein kann und exp2 der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den <b>Arraydatatype</b> steht, wobei das <b>Attribut</b> nums eine Liste von Num('x') ist, die die <b>Dimensionen</b> des Arrays angibt und datatype der Datentyp ist, der über das Anwenden von Subscript() auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den <b>Initializer</b> eines <b>Arrays</b> , dessen Einträge exps weitere Initializer für eine <b>Array-Dimension</b> oder ein Initializer für ein <b>Struct</b> oder ein <b>Logischer Ausdruck</b> sein können.
Subscr(exp1, exp2)	Container für den <b>Indexzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> . Wobei exp1 eine angehängte weitere Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name) Operation sein kann oder ein Name('var') sein kann und exp2 der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für den <b>Structdatatype</b> .
Attr(exp, name)	Container für den <b>Attributzugriff</b> auf einen <b>Structdatatype</b> . Wobei exp1 eine angehängte weitere Subscr(exp1, exp2), Deref(exp1, exp2) oder Attr(exp, name) Operation sein kann oder ein Name('var') sein kann und name das Attribut ist, auf das zugegriffen werden soll.
Struct(assigns, datatype)	
StructDecl(name, allocs)	
If(exp, stmts_goto)	
IfElse(exp, stmts_goto1, stmts_goto2)	

Tabelle 1.3: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
While(exp, stmts_goto)	
DoWhile(exp, stmts_goto)	
Call(name, exps)	
Return(exp)	
FunDecl(datatype, name, allocs)	
FunDef(datatype, name, alocs, stmts_blocks)	
NewStackframe(fun_name, goto_after_call)	
RemoveStackframe()	
File(name, decls_defs.blocks)	
Block(name, stmts_instrs, instrs_before, num_instrs, signature_size, local_vars_size)	
GoTo(name)	
SingleLineComment(prefix, content)	
RETIComment()	

Tabelle 1.4: PicoC-Knoten Teil 2

Die ausgegrauten Attribute der PicoC-Nodes sind **versteckte Attribute**, die **nicht** direkt bei der Erstellung der **PicoC-Nodes** mit einem Wert **initialisiert** werden, sondern im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen bekommen, dass im weiteren Kompiliervorgang **Informationen** transportiert, die später im Kompiliervorgang nicht mehr so leicht zugänglich wären.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 1.1) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.2) unbesetzt ist. Das **Attribut** **position** wird später für Fehlermeldungen gebraucht.

### Definition 1.1: Token-Knoten

Ersetzt ein **Token** bei der Generierung des **Abstract Syntax Tree**, damit der Zugriff auf Knoten des Abstract Syntax Tree möglichst **simpel** ist und keine vermeidbaren Fallunterscheidungen gemacht werden müssen.

**Token-Knoten** entsprechen im Abstract Syntax Tree **Blättern**.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

**Definition 1.2: Container-Knoten**

Dient als *Container* für andere *Container-Knoten* und *Token-Knoten*. Die *Container-Knoten* werden optimalerweise immer so gewählt, dass sie *mehrere Produktionen der Konkreten Syntax* abdecken, die einen *gleichen Aufbau* haben und sich auch unter einem *Überbegriff* zusammenfassen lassen.<sup>a</sup>

*Container-Knoten* entsprechen im Abstract Syntax Tree *Inneren Knoten*.<sup>b</sup>

<sup>a</sup>Wie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. `1 % 3` und **Logischen Ausdrücke**, wie z.B. `1 && 2 < 3`, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.

<sup>b</sup>Thiemann, „Compilerbau“.

**1.2.5.2 RETI-Knoten**

RETI-Knoten	Beschreibung
	asdf

Tabelle 1.5: RETI-Knoten

**1.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung**

Hier sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax 1.2.1** enthalten sind.

Komposition	Beschreibung
Ref(Global(Num('addr')))	Speichert <b>Adresse</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Datensegment Register</b> DS steht auf den <b>Stack</b> .
Ref(Stackframe(Num('addr')))	Speichert <b>Adresse</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF steht auf den <b>Stack</b> .
Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))))	Berechnet die nächste <b>Adresse</b> aus der <b>Adresse</b> , die an Speicherzelle Stack(Num('addr1')) steht und dem <b>Subscript Index</b> , der an Speicherzelle Stack(Num('addr2')) steht und speichert diese auf den <b>Stack</b> . Die Berechnung ist abhängig davon ob der <b>Datentyp</b> ArrayDecl(datatype) oder PntrDecl(datatype) ist. Der <b>Datentyp</b> ist ein <b>verstecktes Attribut</b> von Ref(exp).
Ref(Attr(Stack(Num('addr1')), Name('attr')))	Berechnet die nächste <b>Adresse</b> aus der <b>Adresse</b> , die an Speicherzelle Stack(Num('addr1')) steht und dem <b>Attributnamen</b> Name('attr') und speichert diese auf den <b>Stack</b> . Zur Berechnung ist der Name des <b>Struct</b> in StructSpec(Name('st')) notwendig, dessen <b>Attribut</b> Name('attr') ist. StructSpec(Name('st')) ist ein <b>verstecktes Attribut</b> von Ref(exp).
Assign(Stack(Num('size')), Global(Num('addr')))	Schreibt Num('size') viele Speicherzellen, die ab Global(Num('addr')) relativ zum <b>Datensegment Register</b> DS stehen, versetzt genauso auf den <b>Stack</b> .
Assign(Stack(Num('size')), Stackframe(Num('addr')))	Schreibt Num('size') viele Speicherzellen, die ab Stackframe(Num('addr')) relativ zum <b>Begin-Aktive-Funktion Register</b> BAF stehen, versetzt genauso auf den <b>Stack</b> .
Exp(Global(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Datensegment Register</b> DS steht auf den <b>Stack</b> .
Exp(Stackframe(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF steht auf den <b>Stack</b> .
Exp(Stack(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht auf den <b>Stack</b> .
Assign(Stack(Num('addr1')), Stack(Num('addr2')))	Speichert <b>Inhalt</b> der Speicherzelle Stack(Num('addr2')), die Num('addr2') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht an der <b>Adresse</b> in der Speicherzelle, die Num('addr1') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht.
Assign(Global(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem <b>Stack</b> stehen, versetzt genauso auf die Speicherzellen ab Num('addr') <b>relativ</b> zum <b>Datensegment Register</b> DS.
Assign(Stackframe(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem <b>Stack</b> stehen, versetzt genauso auf die Speicherzellen ab Num('addr') <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF.
Exp(Reg(reg))	Schreibt den aktuellen Wert des <b>Registers</b> reg auf den <b>Stack</b> .
Instr(Loadi(), [Reg(Acc()), Goto(Name('addr@next_instr'))])	Lädt in das Register ACC die <b>Adresse</b> der Instruction, die in diesem Kontext direkt nach dem Sprung zum Block einer anderen Funktion steht.

Tabelle 1.6: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 1.6 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch  $\text{Stack}(\text{Num}('x')), x \in \mathbb{N}$  ersetzt wurde.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen einfach nur eine **Expression** an ein  $\text{Exp}(\text{exp})$  bzw.  $\text{Ref}(\text{exp})$  drangehängt wurde.

#### 1.2.5.4 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>lhs</i>	::=	<i>Alloc</i> ( <i>&lt;type_qual&gt;</i> , <i>&lt;datatype&gt;</i> , <i>Name(str)</i> )   <i>&lt;ref_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;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 1.2.3: Abstrakte Syntax für *L\_Piocc*

### 1.2.5.5 Transformer

### 1.2.5.6 Codebeispiel

Beispiel welches in Subkapitel 1.2.3.2 angefangen wurde, wird hier fortgeführt.

```
1 File
2   Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           PtrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '5',
14                Num '6'
15              ],
16            PtrDecl
17              Num '1',
18              IntType 'int',
19            Name 'attr'
20          ],
21      FunDef
22        VoidType 'void',
23        Name 'main',
24        [],
25        [
26          Exp
27            Alloc
28              Writeable,
29              PtrDecl
30                Num '1',
31                ArrayDecl
32                  [
33                    Num '3',
34                    Num '2'
35                  ],
36                PtrDecl
37                  Num '1',
38                  StructSpec
39                    Name 'st',
40                  Name 'var'
41                ]
42          ]
43        ]
44      ]
45    ]
46  ]
```

Code 1.4: Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert

## 1.3 Code Generierung

### 1.3.1 Übersicht

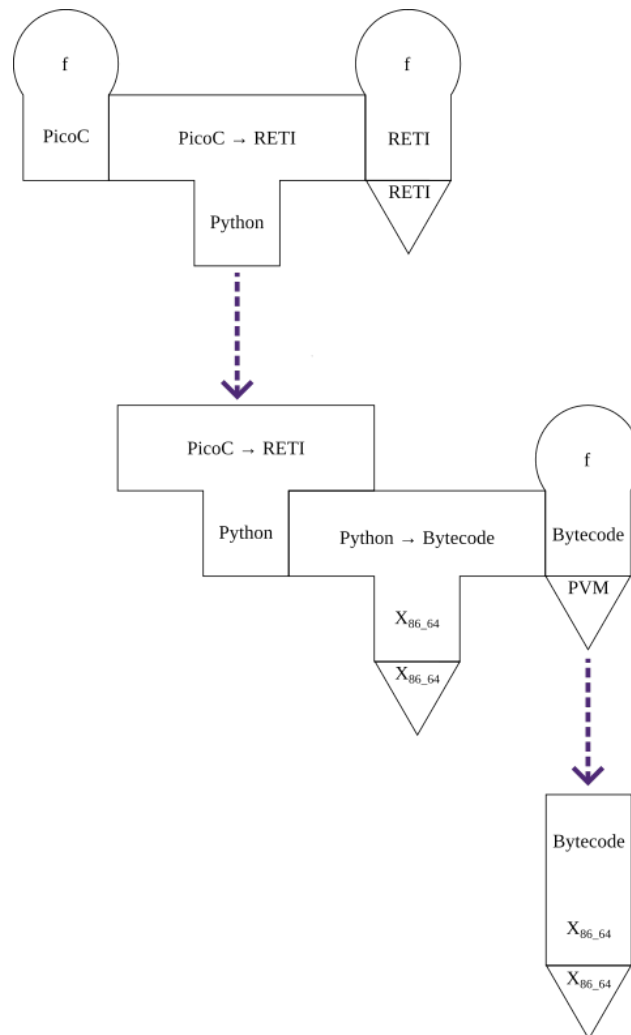


Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

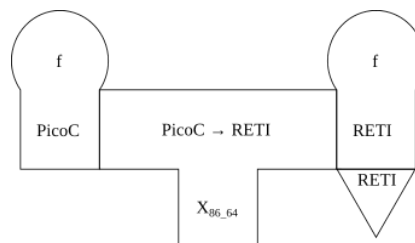


Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform



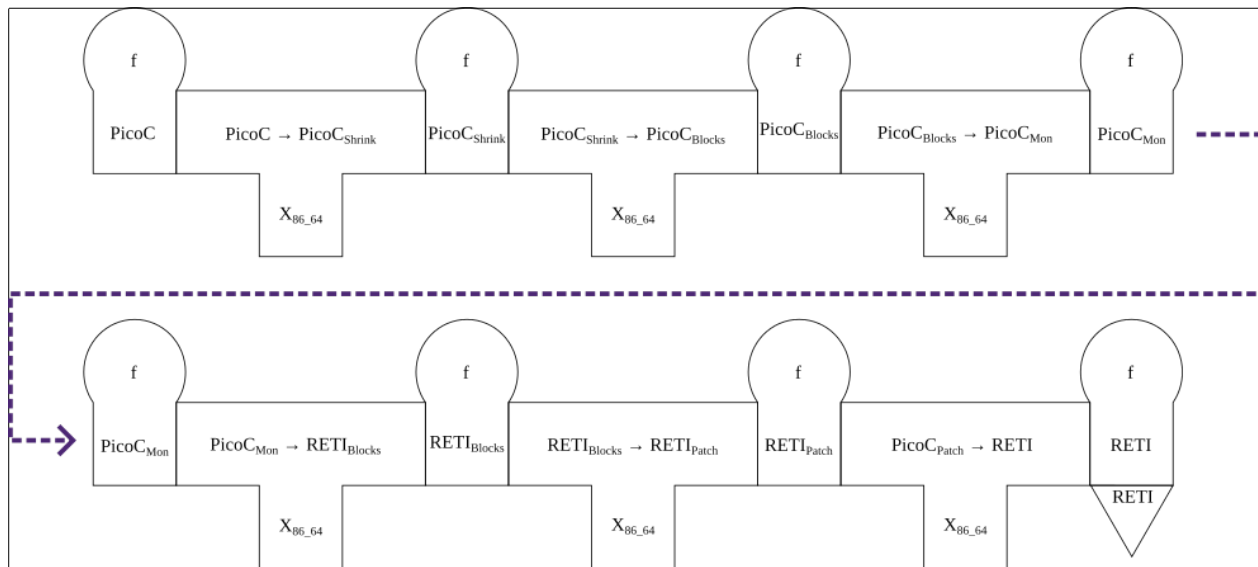


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

## 1.3.2 Passes

### 1.3.2.1 PicoC-Shrink Pass

#### 1.3.2.1.1 Codebeispiel

```

1 // Author: Christoph Scholl, from the Operating Systems Lecture
2
3 void main() {
4     int n = 4;
5     int res = 1;
6     while (1) {
7         if (n == 1) {
8             return;
9         }
10        res = n * res;
11        n = n - 1;
12    }
13 }

```

Code 1.5: PicoC Code für Codebeispiel

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [

```

```

9      Assign
10      Alloc
11      Writeable,
12      IntType 'int',
13      Name 'n',
14      Num '4',
15      Assign
16      Alloc
17      Writeable,
18      IntType 'int',
19      Name 'res',
20      Num '1',
21      While
22      Num '1',
23      [
24      If
25      Atom
26      Name 'n',
27      Eq '==',
28      Num '1',
29      [
30      Return
31      Empty
32      ],
33      Assign
34      Name 'res',
35      BinOp
36      Name 'n',
37      Mul '*',
38      Name 'res',
39      Assign
40      Name 'n',
41      BinOp
42      Name 'n',
43      Sub '-',
44      Num '1'
45      ]
46  ]
47 ]

```

Code 1.6: Abstract Syntax Tree für Codebeispiel

```

1 File
2   Name './example_faculty_it.picoc_shrink',
3   [
4     FunDef
5     VoidType 'void',
6     Name 'main',
7     [],
8     [
9       Assign
10      Alloc
11      Writeable,
12      IntType 'int',

```

```

13     Name 'n',
14     Num '4',
15   Assign
16     Alloc
17       Writeable,
18       IntType 'int',
19       Name 'res',
20     Num '1',
21   While
22     Num '1',
23   [
24     If
25       Atom
26         Name 'n',
27         Eq '==',
28         Num '1',
29       [
30         Return
31           Empty
32       ],
33     Assign
34       Name 'res',
35       BinOp
36         Name 'n',
37         Mul '*',
38         Name 'res',
39     Assign
40       Name 'n',
41       BinOp
42         Name 'n',
43         Sub '-',
44         Num '1'
45   ]
46 ]
47 ]

```

Code 1.7: PicoC Shrink Pass für Codebeispiel

### 1.3.2.2 PicoC-Blocks Pass

#### 1.3.2.2.1 Abstrakte Syntax

<i>decl_def</i>	$::=$	<i>FunDef</i> ( $\langle datatype \rangle$ , <i>Name</i> ( <i>str</i> ), <i>Alloc</i> ( <i>Writeable</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 1.3.1: Abstrakte Syntax für  $L_{PicoC\_Blocks}$ 

#### 1.3.2.2.2 Codebeispiel

```
1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.5',
11          [
12            Assign
13              Alloc
14                Writeable,
15                IntType 'int',
16                Name 'n',
17                Num '4',
18            Assign
19              Alloc
20                Writeable,
21                IntType 'int',
22                Name 'res',
23                Num '1',
24            // While(Num('1'), []),
25            GoTo
26              Name 'condition_check.4'
27          ],
28          Block
29            Name 'condition_check.4',
30            [
31              IfElse
32                Num '1',
33                GoTo
34                  Name 'while_branch.3',
35                GoTo
36                  Name 'while_after.0'
37            ],
38          Block
39            Name 'while_branch.3',
40            [
41              // If(Atom(Name('n'), Eq('=='), Num('1')), []),
42              IfElse
43                Atom
44                  Name 'n',
45                  Eq '==',
46                  Num '1',
47                GoTo
48                  Name 'if.2',
49                GoTo
50                  Name 'if_else_after.1'
51            ],
52          Block
53            Name 'if.2',
54            [
55              Return
56                Empty
57            ],
```

```

58     Block
59     Name 'if_else_after.1',
60     [
61         Assign
62         Name 'res',
63         BinOp
64         Name 'n',
65         Mul '*',
66         Name 'res',
67         Assign
68         Name 'n',
69         BinOp
70         Name 'n',
71         Sub '-',
72         Num '1',
73         GoTo
74         Name 'condition_check.4'
75     ],
76     Block
77     Name 'while_after.0',
78     []
79 ]
80 ]

```

Code 1.8: PicoC-Blocks Pass für Codebeispiel

### 1.3.2.3 PicoC-Mon Pass

#### 1.3.2.3.1 Abstrakte Syntax

<i>ref_loc</i>	$::=$	$Stack(Num(str)) \mid Global(Num(str))$	$L\_Assign\_Alloc$
		$\mid Stackframe(Num(str))$	
<i>error_data</i>	$::=$	$\langle exp \rangle \mid Pos(Num(str), Num(str))$	
<i>exp</i>	$::=$	$Stack(Num(str)) \mid Ref(\langle ref\_loc \rangle, \langle datatype \rangle, \langle error\_data \rangle)$	
<i>stmt</i>	$::=$	$Exp(\langle exp \rangle)$	
		$\mid Assign(Alloc(Writable(), StructSpec(Name(str)), Name(str)),$	
		$Struct(Assign(Name(str), \langle exp \rangle +, \langle datatype \rangle))$	
		$\mid Assign(Alloc(Writable(), ArrayDecl(Num(str) +, \langle datatype \rangle),$	
		$Name(str), Array(\langle exp \rangle +, \langle datatype \rangle))$	
<i>symbol_table</i>	$::=$	$SymbolTable(\langle symbol \rangle)$	$L\_Symbol\_Table$
<i>symbol</i>	$::=$	$Symbol(\langle type\_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)$	
<i>type_qual</i>	$::=$	$Empty()$	
<i>datatype</i>	$::=$	$BuiltIn() \mid SelfDefined()$	
<i>name</i>	$::=$	$Name(str)$	
<i>val</i>	$::=$	$Num(str) \mid Empty()$	
<i>pos</i>	$::=$	$Pos(Num(str), Num(str)) \mid Empty()$	
<i>size</i>	$::=$	$Num(str) \mid Empty()$	

Grammar 1.3.2: Abstrakte Syntax für  $L_{PicoC\_Mon}$ 

#### Definition 1.3: Symboltabelle

**1.3.2.3.2 Codebeispiel**

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'main.5',
6       [
7         // Assign(Name('n'), Num('4')),
8         Exp
9           Num '4',
10        Assign
11          GlobalWrite
12            Num '0',
13          Tmp
14            Num '1',
15          // Assign(Name('res'), Num('1')),
16        Exp
17          Num '1',
18        Assign
19          GlobalWrite
20            Num '1',
21          Tmp
22            Num '1',
23          // While(Num('1'), []),
24        Exp
25          GoTo
26            Name 'condition_check.4'
27      ],
28    Block
29      Name 'condition_check.4',
30      [
31        // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
32        Exp
33          Num '1',
34        IfElse
35          Tmp
36            Num '1',
37          GoTo
38            Name 'while_branch.3',
39          GoTo
40            Name 'while_after.0'
41      ],
42    Block
43      Name 'while_branch.3',
44      [
45        // If(Atom(Name('n'), Eq('=='), Num('1')), []),
46        // IfElse(Atom(Name('n'), Eq('=='), Num('1')), GoTo(Name('if.2')),
47        ↪ GoTo(Name('if_else_after.1'))),
48        Exp
49          GlobalRead
50            Num '0',
51        Exp
52          Num '1',
53        Exp
54          Atom
55            Tmp

```

```
55         Num '2',
56         Eq '==',
57         Tmp
58         Num '1',
59     IfElse
60     Tmp
61     Num '1',
62     GoTo
63     Name 'if.2',
64     GoTo
65     Name 'if_else_after.1'
66 ],
67 Block
68     Name 'if.2',
69     [
70         Return
71         Empty
72     ],
73 Block
74     Name 'if_else_after.1',
75     [
76         // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res'))),
77         Exp
78         GlobalRead
79         Num '0',
80         Exp
81         GlobalRead
82         Num '1',
83         Exp
84         BinOp
85         Tmp
86         Num '2',
87         Mul '*',
88         Tmp
89         Num '1',
90         Assign
91         GlobalWrite
92         Num '1',
93         Tmp
94         Num '1',
95         // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1'))),
96         Exp
97         GlobalRead
98         Num '0',
99         Exp
100        Num '1',
101        Exp
102        BinOp
103        Tmp
104        Num '2',
105        Sub '-',
106        Tmp
107        Num '1',
108        Assign
109        GlobalWrite
110        Num '0',
111        Tmp
```

```

112     Num '1',
113     Exp
114     GoTo
115     Name 'condition_check.4'
116 ],
117 Block
118   Name 'while_after.0',
119   [
120     Return
121     Empty
122   ]
123 ]

```

Code 1.9: PicoC-Mon Pass für Codebeispiel

### 1.3.2.4 RETI-Blocks Pass

#### 1.3.2.4.1 Abstrakte Syntax

<i>program</i>	$::=$	$Program(Name(str), \langle block \rangle^*)$	$L\_Program$
<i>exp_stmts</i>	$::=$	$Goto(str)$	$L\_Blocks$
<i>instrs_before</i>	$::=$	$Num(str)$	
<i>num_instrs</i>	$::=$	$Num(str)$	
<i>block</i>	$::=$	$Block(Name(str), \langle instr \rangle^*, \langle instrs\_before \rangle, \langle num\_instrs \rangle)$	
<i>instr</i>	$::=$	$Goto(Name(str))$	

Grammar 1.3.3: Abstrakte Syntax für  $L_{RETI\_Blocks}$ 

#### 1.3.2.4.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         # // Assign(Name('n'), Num('4')),
8         # Exp(Num('4')),
9         SUBI SP 1,
10        LOADI ACC 4,
11        STOREIN SP ACC 1,
12        # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
13        LOADIN SP ACC 1,
14        STOREIN DS ACC 0,
15        ADDI SP 1,
16        # // Assign(Name('res'), Num('1')),
17        # Exp(Num('1')),
18        SUBI SP 1,
19        LOADI ACC 1,
20        STOREIN SP ACC 1,
21        # Assign(GlobalWrite(Num('1')), Tmp(Num('1'))),
22        LOADIN SP ACC 1,

```



```

23     STOREIN DS ACC 1,
24     ADDI SP 1,
25     # // While(Num('1'), []),
26     Exp
27         GoTo
28             Name 'condition_check.4'
29 ],
30 Block
31     Name 'condition_check.4',
32     [
33         # // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
34         # Exp(Num('1')),
35         SUBI SP 1,
36         LOADI ACC 1,
37         STOREIN SP ACC 1,
38         # IfElse(Tmp(Num('1')), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
39         LOADIN SP ACC 1,
40         ADDI SP 1,
41         JUMP== GoTo
42             Name 'while_after.0';,
43     Exp
44         GoTo
45             Name 'while_branch.3'
46 ],
47 Block
48     Name 'while_branch.3',
49     [
50         # // If(Atom(Name('n'), Eq('=='), Num('1')), []),
51         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), GoTo(Name('if.2')),
52         ↪ GoTo(Name('if_else_after.1'))),
53         # Exp(GlobalRead(Num('0'))),
54         SUBI SP 1,
55         LOADIN DS ACC 0,
56         STOREIN SP ACC 1,
57         # Exp(Num('1')),
58         SUBI SP 1,
59         LOADI ACC 1,
60         STOREIN SP ACC 1,
61         LOADIN SP ACC 2,
62         LOADIN SP IN2 1,
63         SUB ACC IN2,
64         JUMP== 3;,
65         LOADI ACC 0,
66         JUMP 2;,
67         LOADI ACC 1,
68         STOREIN SP ACC 2,
69         ADDI SP 1,
70         # IfElse(Tmp(Num('1')), GoTo(Name('if.2')), GoTo(Name('if_else_after.1'))),
71         LOADIN SP ACC 1,
72         ADDI SP 1,
73         JUMP== GoTo
74             Name 'if_else_after.1';,
75     Exp
76         GoTo
77             Name 'if.2'
78 ],
79 Block

```

```

79     Name 'if.2',
80     [
81         # Return(Empty()),
82         LOADIN BAF PC -1
83     ],
84     Block
85     Name 'if_else_after.1',
86     [
87         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res'))),
88         # Exp(GlobalRead(Num('0'))),
89         SUBI SP 1,
90         LOADIN DS ACC 0,
91         STOREIN SP ACC 1,
92         # Exp(GlobalRead(Num('1'))),
93         SUBI SP 1,
94         LOADIN DS ACC 1,
95         STOREIN SP ACC 1,
96         LOADIN SP ACC 2,
97         LOADIN SP IN2 1,
98         MULT ACC IN2,
99         STOREIN SP ACC 2,
100        ADDI SP 1,
101        # Assign(GlobalWrite(Num('1')), Tmp(Num('1'))),
102        LOADIN SP ACC 1,
103        STOREIN DS ACC 1,
104        ADDI SP 1,
105        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1'))),
106        # Exp(GlobalRead(Num('0'))),
107        SUBI SP 1,
108        LOADIN DS ACC 0,
109        STOREIN SP ACC 1,
110        # Exp(Num('1')),
111        SUBI SP 1,
112        LOADI ACC 1,
113        STOREIN SP ACC 1,
114        LOADIN SP ACC 2,
115        LOADIN SP IN2 1,
116        SUB ACC IN2,
117        STOREIN SP ACC 2,
118        ADDI SP 1,
119        # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
120        LOADIN SP ACC 1,
121        STOREIN DS ACC 0,
122        ADDI SP 1,
123        Exp
124        GoTo
125        Name 'condition_check.4'
126    ],
127    Block
128    Name 'while_after.0',
129    [
130        # Return(Empty()),
131        LOADIN BAF PC -1
132    ]
133 ]

```

Code 1.10: RETI-Blocks Pass für Codebeispiel

### 1.3.2.5 RETI-Patch Pass

#### 1.3.2.5.1 Abstrakte Syntax

$$\text{stmt} ::= \text{Exit}(\text{Num}(\text{str}))$$

Grammar 1.3.4: Abstrakte Syntax für  $L_{RETI\_Patch}$

#### 1.3.2.5.2 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.6',
6       [],
7     Block
8       Name 'main.5',
9       [
10        # // Assign(Name('n'), Num('4')),
11        # Exp(Num('4')),
12        SUBI SP 1,
13        LOADI ACC 4,
14        STOREIN SP ACC 1,
15        # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
16        LOADIN SP ACC 1,
17        STOREIN DS ACC 0,
18        ADDI SP 1,
19        # // Assign(Name('res'), Num('1')),
20        # Exp(Num('1')),
21        SUBI SP 1,
22        LOADI ACC 1,
23        STOREIN SP ACC 1,
24        # Assign(GlobalWrite(Num('1')), Tmp(Num('1'))),
25        LOADIN SP ACC 1,
26        STOREIN DS ACC 1,
27        ADDI SP 1,
28        # // While(Num('1'), [])
29      ],
30    Block
31      Name 'condition_check.4',
32      [
33        # // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
34        # Exp(Num('1')),
35        SUBI SP 1,
36        LOADI ACC 1,
37        STOREIN SP ACC 1,
38        # IfElse(Tmp(Num('1')), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
39        LOADIN SP ACC 1,
40        ADDI SP 1,
41        JUMP== GoTo
42          Name 'while_after.0';
43      ],
44    Block
45      Name 'while_branch.3',

```

```

46     [
47         # // If(Atom(Name('n'), Eq('=='), Num('1')), []),
48         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), GoTo(Name('if.2')),
49         ↪ GoTo(Name('if_else_after.1'))),
49         # Exp(GlobalRead(Num('0'))),
50         SUBI SP 1,
51         LOADIN DS ACC 0,
52         STOREIN SP ACC 1,
53         # Exp(Num('1')),
54         SUBI SP 1,
55         LOADI ACC 1,
56         STOREIN SP ACC 1,
57         LOADIN SP ACC 2,
58         LOADIN SP IN2 1,
59         SUB ACC IN2,
60         JUMP== 3;,
61         LOADI ACC 0,
62         JUMP 2;,
63         LOADI ACC 1,
64         STOREIN SP ACC 2,
65         ADDI SP 1,
66         # IfElse(Tmp(Num('1')), GoTo(Name('if.2')), GoTo(Name('if_else_after.1'))),
67         LOADIN SP ACC 1,
68         ADDI SP 1,
69         JUMP== GoTo
70             Name 'if_else_after.1';
71     ],
72     Block
73     Name 'if.2',
74     [
75         # Return(Empty()),
76         LOADIN BAF PC -1
77     ],
78     Block
79     Name 'if_else_after.1',
80     [
81         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res'))),
82         # Exp(GlobalRead(Num('0'))),
83         SUBI SP 1,
84         LOADIN DS ACC 0,
85         STOREIN SP ACC 1,
86         # Exp(GlobalRead(Num('1'))),
87         SUBI SP 1,
88         LOADIN DS ACC 1,
89         STOREIN SP ACC 1,
90         LOADIN SP ACC 2,
91         LOADIN SP IN2 1,
92         MULT ACC IN2,
93         STOREIN SP ACC 2,
94         ADDI SP 1,
95         # Assign(GlobalWrite(Num('1')), Tmp(Num('1'))),
96         LOADIN SP ACC 1,
97         STOREIN DS ACC 1,
98         ADDI SP 1,
99         # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1'))),
100        # Exp(GlobalRead(Num('0'))),
101        SUBI SP 1,

```

```

102     LOADIN DS ACC 0,
103     STOREIN SP ACC 1,
104     # Exp(Num('1')),
105     SUBI SP 1,
106     LOADI ACC 1,
107     STOREIN SP ACC 1,
108     LOADIN SP ACC 2,
109     LOADIN SP IN2 1,
110     SUB ACC IN2,
111     STOREIN SP ACC 2,
112     ADDI SP 1,
113     # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
114     LOADIN SP ACC 1,
115     STOREIN DS ACC 0,
116     ADDI SP 1,
117     Exp
118     GoTo
119     Name 'condition_check.4'
120 ],
121 Block
122     Name 'while_after.0',
123     [
124         # Return(Empty()),
125         LOADIN BAF PC -1
126     ]
127 ]

```

Code 1.11: RETI-Patch Pass für Codebeispiel

### 1.3.2.6 RETI Pass

#### 1.3.2.6.1 Konkrete und Abstrakte Syntax

<i>dig_no_0</i>	::=	"1"   "2"   "3"   "4"   "5"   "6"	<i>L_Program</i>
		"7"   "8"   "9"	
<i>dig_with_0</i>	::=	"0"   <i>dig_no_0</i>	
<i>num</i>	::=	"0"   <i>dig_no_0 dig_with_0</i> *   "-" <i>dig_with_0</i> *	
<i>letter</i>	::=	"a" ... "Z"	
<i>name</i>	::=	<i>letter</i> ( <i>letter</i>   <i>dig_with_0</i>   <i>_</i> )*	
<i>reg</i>	::=	"ACC"   "IN1"   "IN2"   "PC"   "SP"	
		"BAF"   "CS"   "DS"	
<i>arg</i>	::=	<i>reg</i>   <i>num</i>	
<i>rel</i>	::=	"=="   "!="   "<"   "<="   ">"	
		">="   "_NOP"	

Grammar 1.3.5: Konkrete Syntax für  $L_{RETI\_Lex}$

---

```

instr      ::=  "ADD" reg arg | "ADDI" reg num | "SUB" reg arg           L_Program
              | "SUBI" reg num | "MULT" reg arg | "MULTI" reg num
              | "DIV" reg arg | "DIVI" reg num | "MOD" reg arg
              | "MODI" reg num | "OPLUS" reg arg | "OPLUSI" reg num
              | "OR" reg arg | "ORI" reg num
              | "AND" reg arg | "ANDI" reg num
              | "LOAD" reg num | "LOADIN" arg arg num
              | "LOADI" reg num
              | "STORE" reg num | "STOREIN" arg argnum
              | "MOVE" reg reg
              | "JUMP" rel num | INT num | RTI
              | "CALL" "INPUT" reg | "CALL" "PRINT" reg
program    ::=  name (instr";")*

```

---

Grammar 1.3.6: Konkrete Syntax für  $L_{RETI\_Parse}$ 


---

```

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

```

---

Grammar 1.3.7: Abstrakte Syntax für  $L_{RETI}$ 

### 1.3.2.6.2 Codebeispiel

```

1 # // Assign(Name('n'), Num('4'))
2 # Exp(Num('4'))
3 SUBI SP 1;
4 LOADI ACC 4;
5 STOREIN SP ACC 1;
6 # Assign(GlobalWrite(Num('0')), Tmp(Num('1')))
7 LOADIN SP ACC 1;
8 STOREIN DS ACC 0;
9 ADDI SP 1;
10 # // Assign(Name('res'), Num('1'))
11 # Exp(Num('1'))
12 SUBI SP 1;
13 LOADI ACC 1;
14 STOREIN SP ACC 1;
15 # Assign(GlobalWrite(Num('1')), Tmp(Num('1')))

```

```

16 LOADIN SP ACC 1;
17 STOREIN DS ACC 1;
18 ADDI SP 1;
19 # // While(Num('1'), [])
20 # // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0')))
21 # Exp(Num('1'))
22 SUBI SP 1;
23 LOADI ACC 1;
24 STOREIN SP ACC 1;
25 # IfElse(Tmp(Num('1')), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0')))
26 LOADIN SP ACC 1;
27 ADDI SP 1;
28 JUMP== 49;
29 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
30 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), GoTo(Name('if.2')),
    ↪ GoTo(Name('if_else_after.1')))
31 # Exp(GlobalRead(Num('0')))
32 SUBI SP 1;
33 LOADIN DS ACC 0;
34 STOREIN SP ACC 1;
35 # Exp(Num('1'))
36 SUBI SP 1;
37 LOADI ACC 1;
38 STOREIN SP ACC 1;
39 LOADIN SP ACC 2;
40 LOADIN SP IN2 1;
41 SUB ACC IN2;
42 JUMP== 3;
43 LOADI ACC 0;
44 JUMP 2;
45 LOADI ACC 1;
46 STOREIN SP ACC 2;
47 ADDI SP 1;
48 # IfElse(Tmp(Num('1')), GoTo(Name('if.2')), GoTo(Name('if_else_after.1')))
49 LOADIN SP ACC 1;
50 ADDI SP 1;
51 JUMP== 2;
52 # Return(Empty())
53 LOADIN BAF PC -1;
54 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
55 # Exp(GlobalRead(Num('0')))
56 SUBI SP 1;
57 LOADIN DS ACC 0;
58 STOREIN SP ACC 1;
59 # Exp(GlobalRead(Num('1')))
60 SUBI SP 1;
61 LOADIN DS ACC 1;
62 STOREIN SP ACC 1;
63 LOADIN SP ACC 2;
64 LOADIN SP IN2 1;
65 MULT ACC IN2;
66 STOREIN SP ACC 2;
67 ADDI SP 1;
68 # Assign(GlobalWrite(Num('1')), Tmp(Num('1')))
69 LOADIN SP ACC 1;
70 STOREIN DS ACC 1;
71 ADDI SP 1;

```

```
72 # // Assign(Name('n'), BinOp(Name('n'), Sub('-', Num('1'))))
73 # Exp(GlobalRead(Num('0'))))
74 SUBI SP 1;
75 LOADIN DS ACC 0;
76 STOREIN SP ACC 1;
77 # Exp(Num('1'))
78 SUBI SP 1;
79 LOADI ACC 1;
80 STOREIN SP ACC 1;
81 LOADIN SP ACC 2;
82 LOADIN SP IN2 1;
83 SUB ACC IN2;
84 STOREIN SP ACC 2;
85 ADDI SP 1;
86 # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))))
87 LOADIN SP ACC 1;
88 STOREIN DS ACC 0;
89 ADDI SP 1;
90 JUMP -53;
91 # Return(Empty())
92 LOADIN BAF PC -1;
```

Code 1.12: RETI Pass für Codebeispiel



---

---

# 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).

## Vorlesungen

- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).