

---

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>9</b>
1.1	Lexikalische Analyse	9
1.1.1	Teil der Konkreten Syntax für die Lexikalische Analyse	9
1.1.2	Basic Lexer	10
1.2	Syntaktische Analyse	10
1.2.1	Teil der Konkreten Syntax für die Syntaktische Analyse	10
1.2.2	Umsetzung von Präzidenz	12
1.2.3	Derivation Tree Generierung	13
1.2.3.1	Early Parser	13
1.2.3.2	Codebeispiel	13
1.2.4	Derivation Tree Vereinfachung	14
1.2.4.1	Visitor	14
1.2.4.2	Codebeispiel	14
1.2.5	Abstrakt Syntax Tree Generierung	16
1.2.5.1	PicoC-Knoten	16
1.2.5.2	RETI-Knoten	21
1.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	22
1.2.5.4	Abstrakte Syntax	24
1.2.5.5	Transformer	26
1.2.5.6	Codebeispiel	26
1.3	Code Generierung	26
1.3.1	Übersicht	26
1.3.2	Passes	29
1.3.2.1	PicoC-Shrink Pass	29
1.3.2.1.1	Aufgabe	29
1.3.2.1.2	Abstrakte Syntax	29
1.3.2.1.3	Codebeispiel	29
1.3.2.2	PicoC-Blocks Pass	31
1.3.2.2.1	Aufgabe	31
1.3.2.2.2	Abstrakte Syntax	31
1.3.2.2.3	Codebeispiel	32
1.3.2.3	PicoC-ANF Pass	33
1.3.2.3.1	Aufgabe	33
1.3.2.3.2	Abstrakte Syntax	33
1.3.2.3.3	Codebeispiel	34
1.3.2.4	RETI-Blocks Pass	36
1.3.2.4.1	Aufgaben	36
1.3.2.4.2	Abstrakte Syntax	36
1.3.2.4.3	Codebeispiel	36
1.3.2.5	RETI-Patch Pass	39
1.3.2.5.1	Aufgaben	39
1.3.2.5.2	Abstrakte Syntax	39
1.3.2.5.3	Codebeispiel	39
1.3.2.6	RETI Pass	42
1.3.2.6.1	Aufgaben	42
1.3.2.6.2	Konkrete und Abstrakte Syntax	42

1.3.2.6.3	Codebeispiel . . . . .	43
-----------	------------------------	----

---

---

# Abbildungsverzeichnis

1.1	Cross-Compiler Kompilervorgang ausgeschrieben . . . . .	27
1.2	Cross-Compiler Kompilervorgang Kurzform . . . . .	28
1.3	Architektur mit allen Passes ausgeschrieben . . . . .	28

---

---

# Codeverzeichnis

1.1	PicoC Code für Derivation Tree Generierung . . . . .	13
1.2	Derivation Tree nach Derivation Tree Generierung . . . . .	14
1.3	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	15
1.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert . . . . .	26
1.5	PicoC Code für Codebeispiel . . . . .	30
1.6	Abstract Syntax Tree für Codebeispiel . . . . .	31
1.7	PicoC-Blocks Pass für Codebeispiel . . . . .	33
1.8	PicoC-ANF Pass für Codebeispiel . . . . .	36
1.9	RETI-Blocks Pass für Codebeispiel . . . . .	39
1.10	RETI-Patch Pass für Codebeispiel . . . . .	42
1.11	RETI Pass für Codebeispiel . . . . .	45

---

---

# Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC . . . . .	13
1.2	PicoC-Knoten Teil 1 . . . . .	16
1.3	PicoC-Knoten Teil 2 . . . . .	17
1.4	PicoC-Knoten Teil 3 . . . . .	18
1.5	PicoC-Knoten Teil 4 . . . . .	19
1.6	RETI-Knoten . . . . .	21
1.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung . . . . .	23

---

---

# Definitionsverzeichnis

1.1	Label . . . . .	20
1.2	Token-Knoten . . . . .	20
1.3	Container-Knoten . . . . .	20
1.4	Symboltabelle . . . . .	34



---

---

# Grammatikverzeichnis

1.1.1 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF, Teil 1	9
1.1.2 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF, Teil 2	10
1.2.1 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 1	11
1.2.2 Teil der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 2	12
1.2.3 Abstrakte Syntax der Sprache $L_{PicoC}$	25
1.3.1 Abstrakte Syntax für $L_{PicoC\_Blocks}$	31
1.3.2 Abstrakte Syntax für $L_{PicoC\_Mon}$	34
1.3.3 Abstrakte Syntax für $L_{RETI\_Blocks}$	36
1.3.4 Abstrakte Syntax für $L_{RETI\_Patch}$	39
1.3.5 Konkrete Syntax für $L_{RETI\_Lex}$	42
1.3.6 Konkrete Syntax für $L_{RETI\_Parse}$	43
1.3.7 Abstrakte Syntax für $L_{RETI}$	43

# 1 Implementierung

## 1.1 Lexikalische Analyse

### 1.1.1 Teil der Konkreten Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	"//"/ <i>[/[^\n]*]/</i>   <i>"/*/"/</i> <i>/(. \n)*?/</i> <i>"/*/"</i>	<i>L_Comment</i>
<i>RETI.COMMENT.2</i>	::=	<i>"/"/"</i> <i>"?"</i> <i>"#"/</i> <i>[/[^\n]*]/</i>	
<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_0DIG.WITH_0*</i>	
<i>ASCII.CHAR</i>	::=	"_".." ~ "	
<i>CHAR</i>	::=	<i>"'"ASCII.CHAR"'"</i>	
<i>FILENAME</i>	::=	<i>ASCII.CHAR</i> + <i>".picoc"</i>	
<i>LETTER</i>	::=	<i>"a".."z"</i>   <i>"A".."Z"</i>	
<i>NAME</i>	::=	<i>(LETTER   "_")</i> <i>(LETTER — DIG.WITH_0 — "_")*</i>	
<i>name</i>	::=	<i>NAME   INT.NAME   CHAR.NAME</i> <i>  VOID.NAME</i>	
<i>NOT</i>	::=	<i>" ~ "</i>	
<i>REF_AND</i>	::=	<i>"&amp;"</i>	
<i>un_op</i>	::=	<i>SUB.MINUS   LOGIC.NOT   NOT</i> <i>  MUL.DEREF.PNTR   REF_AND</i>	
<i>MUL.DEREF.PNTR</i>	::=	<i>"*"</i>	
<i>DIV</i>	::=	<i>"/"</i>	
<i>MOD</i>	::=	<i>"%"</i>	
<i>prec1_op</i>	::=	<i>MUL.DEREF.PNTR   DIV   MOD</i>	
<i>ADD</i>	::=	<i>"+"</i>	
<i>SUB.MINUS</i>	::=	<i>"_"</i>	
<i>prec2_op</i>	::=	<i>ADD   SUB.MINUS</i>	
<i>LT</i>	::=	<i>"&lt;"</i>	<i>L_Logic</i>
<i>LTE</i>	::=	<i>"&lt;="</i>	
<i>GT</i>	::=	<i>"&gt;"</i>	
<i>GTE</i>	::=	<i>"&gt;="</i>	
<i>rel_op</i>	::=	<i>LT   LTE   GT   GTE</i>	
<i>EQ</i>	::=	<i>"=="</i>	
<i>NEQ</i>	::=	<i>"!="</i>	
<i>eq_op</i>	::=	<i>EQ   NEQ</i>	
<i>LOGIC.NOT</i>	::=	<i>"!"</i>	

**Grammar 1.1.1:** Teil der Konkreten Syntax der Sprache  $L_{Picoc}$  für die Lexikalische Analyse in EBNF, Teil 1

<i>INT_DT.2</i>	::=	"int"	<i>L_Assign_Alloc</i>
<i>INT_NAME.3</i>	::=	"int" ( <i>LETTER</i>   <i>DIG_WITH_0</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 1.1.2:** Teil der Konkretten Syntax der Sprache  $L_{\text{PicoC}}$  für die Lexikalische Analyse in EBNF, Teil 2

### 1.1.2 Basic Lexer

## 1.2 Syntaktische Analyse

### 1.2.1 Teil der Konkretten 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>L_Array</i> +
		<i>input_exp</i>   <i>print_exp</i>   <i>prim_exp</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>pnter_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>pnter_deg</i>	::=	"*" *	<i>L_Pntr</i>
<i>pnter_decl</i>	::=	<i>pnter_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>pnter_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:** Teil der Konkreten Syntax der Sprache  $L_{\text{Picoc}}$  für die 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:** Teil der Konkretten Syntax der Sprache  $L_{PicoC}$  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.

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

Präzidenz	Operator	Beschreibung	Assoziativität
1	a()	Funktionsaufruf	Links, dann rechts →
	a[]	Indezzugriff	
	a.b	Attributzugriff	
2	-a	Unäres Minus	Rechts, dann links ←
	!a ~a	Logisches NOT und Bitweise NOT	
	*a &a	Dereferenz und Referenz, auch Adresse-von	
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 1.1: Präzidenzregeln von PicoC

## 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     pntr_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     ptr_decl
40     ptr_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

PiocC-Knoten	Beschreibung
Name(val)	Ein <b>Bezeichner</b> , z.B. <code>my_fun</code> , <code>my_var</code> usw., aber da es keine gute Kurzform für <code>Identifizier()</code> (englisches Wort für Bezeichner) gibt, wurde dieser Knoten <code>Name()</code> genannt.
Num(val)	Eine <b>Zahl</b> , z.B. 42, -3 usw.
Char(val)	Ein <b>Zeichen</b> der <b>ASCII-Zeichenkodierung</b> , z.B. <code>'c'</code> , <code>'*'</code> usw.
Minus(), Not(), DerefOp(), RefOp(), LogicNot()	Die <b>unären Operatoren</b> <code>un_op</code> : <code>-a</code> , <code>~a</code> , <code>*a</code> , <code>&amp;a !a</code> .
Add(), Sub(), Mul(), Div(), Mod(), Oplus(), And(), Or(), LogicAnd(), LogicOr()	Die <b>binären Operatoren</b> <code>bin_op</code> : <code>a + b</code> , <code>a - b</code> , <code>a * b</code> , <code>a / b</code> , <code>a % b</code> , <code>a ^ b</code> , <code>a &amp; b</code> , <code>a   b</code> , <code>a &amp;&amp; b</code> , <code>a    b</code> .
Eq(), NEq(), Lt(), LtE(), Gt(), GtE()	Die <b>Relationen</b> <code>rel</code> : <code>a == b</code> , <code>a != b</code> , <code>a &lt; b</code> , <code>a &lt;= b</code> , <code>a &gt; b</code> , <code>a &gt;= b</code> .
Const(), Writeable()	Die <b>Type Qualifier</b> <code>type_qual</code> : <code>const</code> , was für ein <b>nicht beschreibbare Konstante</b> steht und das <b>nicht</b> Angeben von <code>const</code> , 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> <code>datatype</code> eingeordnet werden: <code>int</code> , <code>char</code> , <code>void</code> .
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: <code>&lt;exp1&gt; &lt;bin_op&gt; &lt;exp2&gt;</code>
UnOp(un_op, exp)	Container für eine <b>unäre Operation</b> mit einer Expression: <code>&lt;un_op&gt; &lt;exp&gt;</code> .
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: <code>&lt;exp1&gt; &lt;rel&gt; &lt;exp2&gt;</code>
ToBool(exp)	Container für einen <b>Arithmetischen Ausdruck</b> , wie z.B. <code>1 + 3</code> oder einfach nur <code>3</code> , der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis <code>x &gt; 1</code> auf 1 abgebildet wird.
Alloc(type_qual, datatype, name, local_var_or_param)	<b>Container</b> für eine <b>Allokation</b> <code>&lt;type_qual&gt; &lt;datatype&gt; &lt;name&gt;</code> mit den notwendigen Knoten <code>type_qual</code> , <code>datatype</code> und <code>name</code> , die alle für einen Eintrag in der <b>Symboltabelle</b> notwendigen Informationen enthalten. Zudem besitzt er ein verstecktes Attribut <code>local_var_or_param</code> , 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 <code>lhs</code> ein <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> , <code>Attr(exp, name)</code> oder <code>Name('var')</code> sein kann und <code>exp</code> ein beliebiger <b>Logischer Ausdruck</b> sein kann: <code>lhs = exp</code> .

Tabelle 1.2: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen <b>beliebigen Ausdruck</b> , dessen Ergebnis auf den <b>Stack</b> soll. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Stack(num)	Container, der für das <b>temporäre</b> Ergebnis einer Berechnung, das <b>num</b> Speicherzellen relativ zum <b>Stackpointer Register SP</b> steht.
Stackframe(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Begin-Aktive-Funktion Register BAF</b> steht.
Global(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Datensegment Register DS</b> steht.
StackMalloc(num)	Container, der für das <b>Allokieren</b> von <b>num</b> Speicherzellen auf dem <b>Stack</b> steht.
PntrDecl(num, datatype)	Container, der für den <b>Pointerdatatype</b> steht: <b>&lt;prim_dt&gt; *&lt;var&gt;</b> , wobei das <b>Attribut num</b> die <b>Anzahl zusammengefasster Pointer</b> angibt und <b>datatype</b> 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 &amp;&lt;var&gt;</b> steht und die <b>Adresse</b> einer <b>Location</b> (Definition ??) auf den Stack schreiben soll, die über <b>exp</b> eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Deref(lhs, exp)	Container für den <b>Indezzugriff</b> auf einen <b>Array- oder Pointerdatatype</b> : <b>&lt;var&gt;[&lt;i&gt;]</b> , wobei <b>exp1</b> eine angehängte weitere <b>Subscr(exp1, exp2)</b> , <b>Deref(exp1, exp2)</b> , <b>Attr(exp, name)</b> oder ein <b>Name('var')</b> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den <b>Arraydatatype</b> steht: <b>&lt;prim_dt&gt; &lt;var&gt;[&lt;i&gt;]</b> , wobei das <b>Attribut nums</b> eine Liste von <b>Num('x')</b> ist, die die <b>Dimensionen</b> des Arrays angibt und <b>datatype</b> der Datentyp ist, der über das Anwenden von <b>Subscript()</b> auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den <b>Initializer</b> eines <b>Arrays</b> , dessen Einträge <b>exps</b> weitere Initializer für eine <b>Array-Dimension</b> oder ein Initializer für ein <b>Struct</b> oder ein <b>Logischer Ausdruck</b> sein können, z.B. <b>{{1, 2}, {3, 4}}</b> . Des Weiteren besitzt er ein verstecktes Attribut <b>datatype</b> , welches für den <b>PicoC-ANF Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
Subscr(exp1, exp2)	Container für den <b>Indezzugriff</b> auf einen <b>Array- oder Pointerdatatype</b> : <b>&lt;var&gt;[&lt;i&gt;]</b> , wobei <b>exp1</b> eine angehängte weitere <b>Subscr(exp1, exp2)</b> , <b>Deref(exp1, exp2)</b> oder <b>Attr(exp, name)</b> Operation sein kann oder ein <b>Name('var')</b> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten <b>Structdatatype</b> : <b>struct &lt;name&gt;</b> , wobei das <b>Attribut name</b> festlegt, welchen <b>selbst definierte</b> Structdatatype dieser Container-Knoten repräsentiert.
Attr(exp, name)	Container für den <b>Attributzugriff</b> auf einen <b>Structdatatype</b> : <b>&lt;var&gt;.&lt;attr&gt;</b> , wobei <b>exp1</b> eine angehängte weitere <b>Subscr(exp1, exp2)</b> , <b>Deref(exp1, exp2)</b> oder <b>Attr(exp, name)</b> Operation sein kann oder ein <b>Name('var')</b> sein kann und <b>name</b> das Attribut ist, auf das zugegriffen werden soll.

Tabelle 1.3: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den <b>Initializer</b> eines <b>Structs</b> , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines <b>Attributezeichners</b> , zu einem weiteren Initializer für eine <b>Array-Dimension</b> oder zu einem Initializer für ein <b>Struct</b> oder zu einem <b>Logischen Ausdruck</b> . Des Weiteren besitzt er ein verstecktes Attribut datatype, welches für den <b>PicoC-ANF Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines <b>selbstdefinierten Structdatentyps</b> , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der <b>Bezeichner</b> des Structdatentyps ist und allocs eine Liste von Bezeichnern der <b>Attribute</b> des Structdatentyps mit dazugehörigem <b>Datentyp</b> , wofür sich der <b>Container-Knoten</b> Alloc(type_qual, datatype, name) sehr gut als <b>Container</b> eignet.
If(exp, stmts)	Container für ein <b>If Statement</b> if(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts1, stmts2)	Container für ein <b>If-Else Statement</b> if(<exp>) { <stmts2> } else { <stmts2> } inklusive <b>Condition</b> exp und 2 <b>Branches</b> stmts1 und stmts2, die zwei Alternativen darstellen in denen jeweils <b>Listen von Statements</b> oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts)	Container für ein <b>While-Statement</b> while(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts)	Container für ein <b>Do-While-Statement</b> do { <stmts> } while(<exp>); inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen <b>Funktionsaufruf</b> : fun_name(exps), wobei name der <b>Bezeichner</b> der Funktion ist, die aufgerufen werden soll und exps eine <b>Liste von Argumenten</b> ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein <b>Return-Statement</b> : return <exp>, wobei das <b>Attribut</b> exp einen <b>Logischen Ausdruck</b> darstellt, dessen Ergebnis vom <b>Return-Statement</b> zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine <b>Funktionsdeklaration</b> , z.B. <datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der <b>Rückgabewert</b> der Funktion ist, name der <b>Bezeichner</b> der Funktion ist und allocs die <b>Parameter</b> der Funktion sind, wobei der <b>Container-Knoten</b> Alloc(type_spec, datatype, name) als Container für die Parameter dient.

Tabelle 1.4: PicoC-Knoten Teil 3

Piocc-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts_blocks)	Container für eine <b>Funktionsdefinition</b> , z.B. <datatype> <fun_name>(<datatype> <param>) {<stmts>}, wobei datatype der <b>Rückgabewert</b> der Funktion ist, name der <b>Bezeichner</b> der Funktion ist, allocs die <b>Parameter</b> der Funktion sind, wobei der <b>Container-Knoten</b> Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts_blocks eine Liste von <b>Statements</b> bzw. <b>Blöcken</b> ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die <b>Erstellung</b> eines neuen <b>Stackframes</b> und Speicherung des Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und der <b>Rücksprungsadresse</b> nacheinander an den <b>Anfang</b> des neuen <b>Stackframes</b> . Das Attribut fun_name steht dabei für den Bezeichner der Funktion, für die ein neuer <b>Stackframe</b> erstellt werden soll. Das Attribut fun_name dient später dazu den <b>Block</b> dieser Funktion zu finden, weil dieser für den weiteren Kompilierungsvorgang wichtige Information in seinen <b>versteckten Attributen</b> gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die <b>Adresse</b> des Befehls, der direkt auf die <b>Jump Instruction</b> folgt, ersetzt wird.
RemoveStackframe()	Container für das <b>Entfernen</b> des aktuellen <b>Stackframes</b> durch das <b>Wiederherstellen</b> des im noch <b>aktuellen Stackframe</b> gespeicherten Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und das Setzen des SP-Registers auf den Wert des BAF-Registers <b>vor</b> der Wiederherstellung.
File(name, decls_defs_blocks)	Container für alle <b>Funktionen</b> oder <b>Blöcke</b> , welche eine Datei als Ursprung haben, wobei name der <b>Dateiname</b> der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von <b>Funktionen</b> bzw. <b>Blöcken</b> ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für <b>Statements</b> , der auch als <b>Block</b> bezeichnet wird, wobei das Attribut name der Bezeichner des <b>Labels</b> (Definition 1.1) des Blocks ist und stmts_instrs eine <b>Liste von Statements</b> oder <b>Instructions</b> . Zudem besitzt er noch 3 <b>versteckte Attribute</b> , wobei instrs_before die Zahl der <b>Instructions</b> vor diesem <b>Block</b> zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>Parameter</b> der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>lokalen Variablen</b> der Funktion belegt werden müssen.
GoTo(name)	Container für ein <b>Goto</b> zu einem anderen <b>Block</b> , wobei das Attribut name der Bezeichner des <b>Labels</b> des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen <b>Kommentar</b> , den der Compiler selber während des <b>Kompilierungsvorgangs</b> erstellt, der im <b>RETI-Interpreter</b> selbst später <b>nicht</b> sichtbar sein wird, aber in den <b>Immediate-Dateien</b> , welche die <b>Abstract Syntax Trees</b> nach den verschiedenen <b>Passes</b> enthalten.
RETIComment(value)	Container für einen <b>Kommentar</b> im Code der Form: // # comment, der im <b>RETI-Interpreter</b> später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer <b>RETI-CPU nicht umsetzbar</b> ist und auch nicht sinnvoll wäre umzusetzen. Der <b>Kommentar</b> ist im Attribut <b>value</b> , welches jeder Knoten besitzt gespeichert.

Tabelle 1.5: PicoC-Knoten Teil 4

**Definition 1.1: Label**

Durch einen *Bezeichner eindeutig* zuordenbares *Sprungziel* im Programmcode.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

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 Kompilierungsvorgang **Informationen** transportiert, die später im Kompilierungsvorgang nicht mehr so leicht zugänglich wären.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 1.2) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.3) unbesetzt ist. Das **Attribut** **position** wird später für Fehlermeldungen gebraucht.

**Definition 1.2: 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 1.3: 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
Program(name, instrs)	Container für alle <b>Instructions</b> : <name> <instrs>, wobei <b>name</b> der <b>Dateiname</b> der Datei ist, die erstellt wird und <b>instrs</b> eine <b>Liste von Instructions</b> ist.
Instr(op, args)	Container für eine <b>Instruction</b> : <op> <args>, wobei <b>op</b> eine <b>Operation</b> ist und <b>args</b> eine <b>Liste von Argumenten</b> für dieser Operation.
Jump(rel, im_goto)	Container für eine <b>Jump-Instruction</b> : JUMP<rel> <im>, wobei <b>rel</b> eine <b>Relation</b> ist und <b>im_goto</b> ein <b>Immediate Value</b> <b>Im(val)</b> für die <b>Anzahl an Speicherzellen</b> , um die relativ zur <b>Jump-Instruction</b> gesprungen werden soll oder ein <b>GoTo(Name('block.xyz'))</b> , das später im <b>RETI-Patch Pass</b> durch einen passenden <b>Immediate Value</b> ersetzt wird.
Int(num)	Container für einen <b>Interruptaufruf</b> : INT <im>, wobei <b>num</b> die <b>Interruptvektornummer</b> (IVN) für die passende Speicherzelle in der <b>Interruptvektortabelle</b> ist, in der die Adresse der <b>Interrupt-Service-Routine</b> (ISR) steht.
Call(name, reg)	Container für einen <b>Prozeduraufruf</b> : CALL <name> <reg>, wobei <b>name</b> der <b>Bezeichner</b> der Prozedur, die aufgerufen werden soll ist und <b>reg</b> ein <b>Register</b> ist, das als <b>Argument</b> an die Prozedur dient. Diese <b>Operation</b> ist in der Betriebssysteme Vorlesung <sup>a</sup> nicht deklariert, sondern wurde dazuerfunden, um unkompliziert ein CALL PRINT ACC oder CALL INPUT ACC im RETI-Interpreter simulieren zu können.
Name(val)	Bezeichner für eine <b>Prozedur</b> , z.B. PRINT oder INPUT oder den <b>Programmnamen</b> , z.B. PROGRAMNAME. Dieses <b>Argument</b> ist in der Betriebssysteme Vorlesung <sup>a</sup> nicht deklariert, sondern wurde dazuerfunden, um Bezeichner, wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein <b>Register</b> .
Im(val)	Ein <b>Immediate Value</b> , z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), Oplus(), Or(), And()	<b>Compute-Memory</b> oder <b>Compute-Register</b> Operationen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(), Oplusi(), Ori(), Andi()	<b>Compute-Immediate</b> Operationen: ADDI, SUBI, MULTI, DIVI, MODI, OPLUSI, ORI, ANDI.
Load(), Loadin(), Loadi()	<b>Load</b> Operationen: LOAD, LOADIN, LOADI.
Store(), Storein(), Move()	<b>Store</b> Operationen: STORE, STOREIN, MOVE.
Lt(), LtE(), Gt(), GtE(), Eq(), NEq(), Always(), NOP()	<b>Relationen</b> : <, <=, >, >=, ==, !=, _NOP.
Rti()	<b>Return-From-Interrupt</b> Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()	<b>Register</b> : PC, IN1, IN2, ACC, SP, BAF, CS, DS.

<sup>a</sup> Scholl, „Betriebssysteme“

Tabelle 1.6: 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 verstecktes Attribut 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 verstecktes Attribut 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 1.7: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung



Um die obige Tabelle 1.7 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>stmt</i>	::=	<i>RETIComment()</i>	<i>L_Comment</i>
<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;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;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 der Sprache  $L_{Pioc}$

Man spricht hier von der ”**Abstrakten Syntax der Sprache**  $L_{PicoC}$  und meint hier mit der Sprache  $L_{PicoC}$  **nicht** die Sprache, welche durch die **Abstrakte Syntax** beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck die **Abstrakt Syntax** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakt Syntax** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Redeart** wurde aus der **Quelle** G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

Das Ausgeben eines **Abstract Syntax Trees** wird in Python über die **Magische Methode** `__repr__()`<sup>2</sup> umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine Textrepräsentation seiner selbst und all seiner Knoten mit an den richtigen Stellen passenden **runden öffnenden** ( und **schließenden** ) **Klammern**, sowie **Kommas** , und **Semikolons** ; zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgeben**.

#### 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(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')],
8           ↪ PtrDecl(Num('1'), IntType('int')))), Name('attr'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
16          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var'))
17      ]
18  ]

```

**Code 1.4:** *Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert*

## 1.3 Code Generierung

### 1.3.1 Übersicht

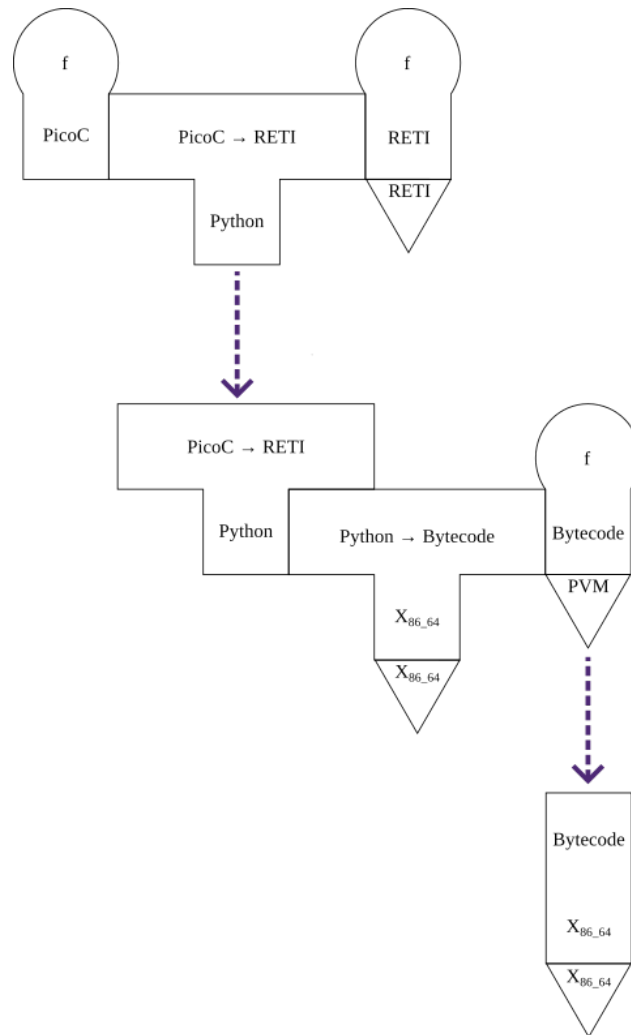
Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 1.2, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **Container-**

<sup>2</sup>Spezielle Methode, die immer aufgerufen wird, wenn das **Object**, dass in Besitz dieser Methode ist als **String** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.

**Knoten** und **Token-Knoten** im **Abstract Syntax Tree** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstract Syntax Tree** der Sprache  $L_{PicoC}$ , der durch die **Abstrakte Syntax** in Grammatik 1.2.3 spezifiziert ist in einen entsprechenden **Abstract Syntax Tree** der Sprache  $L_{RETI}$  umzuformen. Das ganze lässt sich, wie in Unterkapitel ?? bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition ??) herunterbricht.

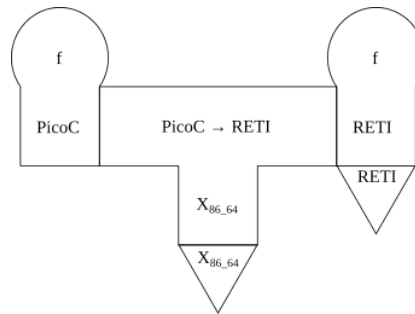
Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton ??). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel ??) in Abbildung 1.1 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer  $X_{86,64}$ -Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer  $X_{86,64}$ -Architektur laufen könnte. Und selbst dieses **T-Diagramm** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu  $X_{86,64}$  kompiliert wurde usw.



**Abbildung 1.1:** Cross-Compiler Kompiliervorgang ausgeschrieben

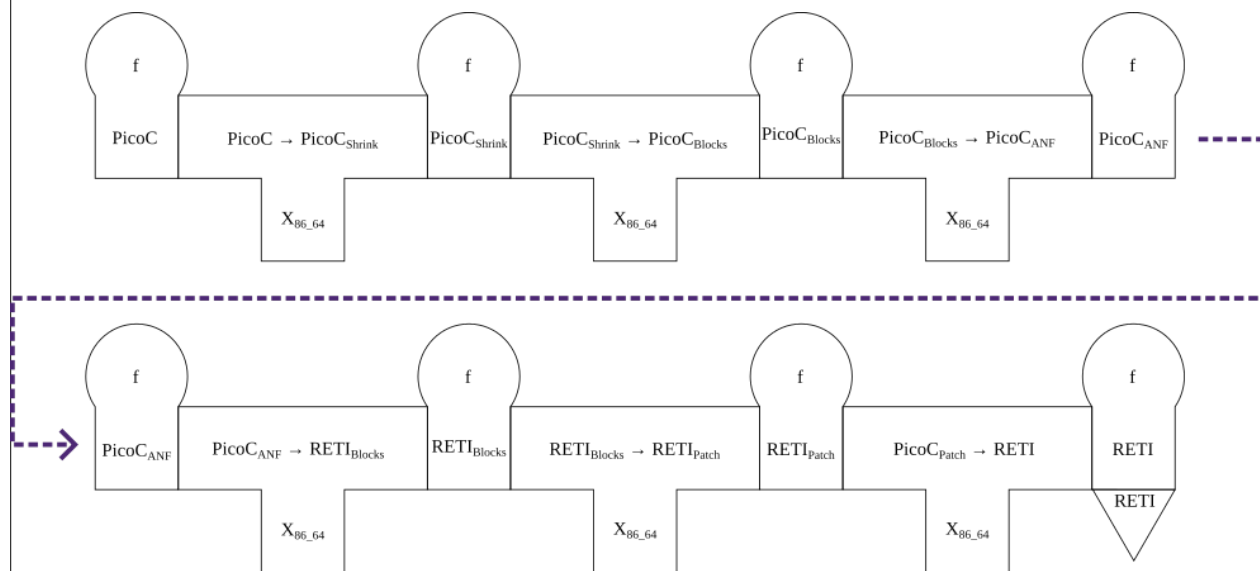
Dieses längliche **T-Diagramm** in Abbildung 1.1 lässt sich zusammenfassen, sodass man das **T-Diagramm** in

Abbildung 1.2 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in  $X_{86\_64}$ -Maschinensprache geschrieben ist.



**Abbildung 1.2:** Cross-Compiler Kompilervorgang Kurzform

Nachdem der Kompilierprozess des **PicoC-Compiler** im **vertikalen** nun genauer angesehen wurde, wird der Kompilierprozess im Folgenden im **horizontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 1.3 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition ??) des **PicoC-Compilers** aufeinanderfolgen. In der **Pipe-Architektur** nutzt der jeweils nächste **Pass** den generierten **Abstract Syntax Tree** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstract Syntax Tree** in seiner eigenen **Sprache** zu generieren.



**Abbildung 1.3:** Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 1.3.2 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 1.3.2 nicht ausreichend vertieft wurden. Viele der verwendeten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der**

**Kompilierung** gezeigt wird<sup>3</sup>. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem **Link**<sup>4</sup> finden und mithilfe der im Ordner `/code_examples` beiliegenden **Makefile** und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind<sup>5</sup>.

### 1.3.2 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der verschiedenen Aspekte<sup>6</sup> der **Bachelorarbeit** wichtig.

Auf jedes Detail der einzelnen **Passes** wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** im Detail erklärt sind und andererseits viele Aufgaben dieser **Passes** eher dem **Bachelorprojekt** zuzurechnen sind.

#### 1.3.2.1 PicoC-Shrink Pass

##### 1.3.2.1.1 Aufgabe

Der Aufgabe des **PicoC-Shrink Pass** ist in Unterkapitel ?? ausführlich an einem Beispiel erklärt. Kurzgefasst hat der **PicoC-Shrink Pass** die Aufgabe, die Eigenheit auszunutzen, dass der **Dereferenzierungoperator** `*pntnr` und die damit einhergehende **Pointer Arithmetik** `*(pntnr + i)` sich in der Untermenge der Sprache  $L_C$ , welche die Sprache  $L_{PicoC}$  darstellt genau gleich verhält, wie der **Operator** für den **Zugriff** auf den **Index** eines **Arrays** `ar[i]`.

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** `Deref(exp, i)` im jeweiligen **Abstract Syntax Tree** in **Knoten** `Subscr(exp, i)` um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung sparen lassen, denn man kann die **Dereferenzierung** `*(var + i)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `var[i]` übernehmen lassen.

##### 1.3.2.1.2 Abstrakte Syntax

Die **Abstrakte Syntax** des **PicoC-Shrink Passes** ist **identisch** mit der **Abstrakten Syntax** der Sprache  $L_{PicoC}$  aus Unterkapitel 1.2.3, nach welcher der erste **Abstract Syntax Tree** in der **Syntaktischen Analyse** generiert wurde.

##### 1.3.2.1.3 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 1.5 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 1.5 ist in der Funktion `faculty` ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen **Arguments** berechnet. Der Algorithmus basiert auf einem **Beispielprogramm** aus der Vorlesung Scholl, „Betriebssysteme“, der in der Vorlesung allerdings **rekursiv** implementiert war.

Dieser **rekursive** Algorithmus ist allerdings **kein** gutes **Anschauungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei

<sup>3</sup>Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

<sup>4</sup>[https://github.com/matthejue/Bachelorarbeit/tree/master/code\\_examples](https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples).

<sup>5</sup>Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstract Syntax Trees** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

<sup>6</sup>In kurz: **Pointer**, **Arrays**, **Structs** und **Funktionen**.

der Kompilierung von `if`-, `if-else`-, `while`- und `do-while`-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 1.5 umgeschrieben, um `if`- und `while`-Statemens zu enthalten.

Beide Varianten des Algorithmus wurden zum Testen des **PicoC-Compilers** verwendet und sind als Tests im Ordner `/tests` unter [Link<sup>7</sup>](#) unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** mit eigenen **Codebeispielen** erklärt werden und alle sonstigen Details dem **Bachelorprojekt** zuzurechnen sind.

```

1 // based on a example program from Christoph Scholl's Operating Systems lecture
2
3 int faculty(int n){
4     int res = 1;
5     while (1) {
6         if (n == 1) {
7             return res;
8         }
9         res = n * res;
10        n = n - 1;
11    }
12 }
13
14 void main() {
15     print(faculty(4));
16 }

```

**Code 1.5:** *PicoC Code für Codebespiel*

In Code 1.6 sieht man den **Abstract Syntax Tree**, der in der **Syntaktischen Analyse** generiert wurde.

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
10      [
11        Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
12        While
13          Num '1',
14          [

```

<sup>7</sup>[https://github.com/matthejue/PicoC-Compiler/tree/new\\_architecture/tests](https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests).

```

15      If
16      Atom(Name('n'), Eq('=='), Num('1')),
17      [
18          Return(Name('res'))
19      ]
20      Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21      Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22  ]
23 ],
24 FunDef
25 VoidType 'void',
26 Name 'main',
27 [],
28 [
29     Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30 ]
31 ]

```

Code 1.6: Abstract Syntax Tree für Codebeispiel

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstract Syntax Tre** in Code 1.6, da das Codebeispiel keine **Dereferenzierung** enthält.

### 1.3.2.2 PicoC-Blocks Pass

#### 1.3.2.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** ist die Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs-`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)-`Knoten umzusetzen. Der `IfElse(exp, stmts1, stmts2)-`Knoten wird zur Umsetzung der **Bedingung** verwendet und es wird, je nachdem, ob die Bedingung **wahr** oder **falsch** ist mithilfe der `GoTo(label)-`Knoten in einen von zwei **alternativen Branches** gesprungen oder ein **Branch** erneut aufgerufen usw.

#### 1.3.2.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax 1.2.3** um die Knoten zu erweitern, die im Unterkapitel 1.3.2.2.1 erwähnt wurden. Des Weiteren wird für die **Kommentare**, die in vielen Codebeispielen zur leichteren Verständlichkeit eingefügt wurden ein `SingleLineComment(prefix, content)-`Knoten benötigt. Die **Funktionsdefinition** `FunDef(<datatype>, Name(str), Alloc(Writeable(), <datatype>, Name(str))* , <block>*)` ist nun ein Container für **Blöcke** `Block(Name(str), <stmt>*)` und keine Statements `stmt` mehr.

<i>decl_def</i>	$::=$	<i>FunDef</i> ( <i>&lt;datatype&gt;</i> , <i>Name(str)</i> , <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))* , &lt;block&gt;*)</i>	<i>L_Fun</i>
<i>block</i>	$::=$	<i>Block</i> ( <i>Name(str)</i> , <i>&lt;stmt&gt;*)</i>	<i>L_Blocks</i>
<i>stmt</i>	$::=$	<i>GoTo</i> ( <i>Name(str)</i> )   <i>SingleLineComment</i> ( <i>str, str</i> )	

Grammar 1.3.1: Abstrakte Syntax für  $L_{PicoC\_Blocks}$



Die **Abstrakte Syntax** in Tabelle 1.3.1 ist als Erweiterung bzw.  
Theoretisch gibt es

### 1.3.2.2.3 Codebeispiel

In Code 1.7 sieht man den **Abstract-Syntax-Tree** des **Piocc-Blocks Passes** für das aus Unterkapitel 1.5 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion `faculty` und die `main`-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement oder bis zum ersten **Auftauchen** eines `If(exp, stmts)-`, `IfElse(exp, stmts1, stmts2)-`, `While(exp, stmts)-` oder `DoWhile(exp, stmts)-`Knoten stehen. Je nachdem, ob ein `If(exp, stmts)-`, `IfElse(exp, stmts1, stmts2)-`, `While(exp, stmts)-` oder `DoWhile(exp, stmts)-`Knoten auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```

1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
10      [
11        Block
12          Name 'faculty.6',
13          [
14            Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1'))
15            // While(Num('1'), [])
16            GoTo(Name('condition_check.5'))
17          ],
18        Block
19          Name 'condition_check.5',
20          [
21            IfElse
22              Num '1',
23              [
24                GoTo(Name('while_branch.4'))
25              ],
26              [
27                GoTo(Name('while_after.1'))
28              ]
29          ],
30        Block
31          Name 'while_branch.4',
32          [
33            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
34            IfElse
35              Atom(Name('n'), Eq('=='), Num('1')),
36              [
37                GoTo(Name('if.3'))
38              ],
39              [
40                GoTo(Name('if_else_after.2'))

```

```
41     ]
42   ],
43   Block
44     Name 'if.3',
45     [
46       Return(Name('res'))
47     ],
48   Block
49     Name 'if_else_after.2',
50     [
51       Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52       Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53       GoTo(Name('condition_check.5'))
54     ],
55   Block
56     Name 'while_after.1',
57     []
58 ],
59 FunDef
60   VoidType 'void',
61   Name 'main',
62   [],
63   [
64     Block
65       Name 'main.0',
66       [
67         Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
68       ]
69   ]
70 ]
```

**Code 1.7:** *PicoC-Blocks Pass für Codebeispiel*

### 1.3.2.3 PicoC-ANF Pass

#### 1.3.2.3.1 Aufgabe

#### 1.3.2.3.2 Abstrakte Syntax

<i>ref_loc</i>	$::=$	<i>Stack</i> ( <i>Num</i> ( <i>str</i> ))   <i>Global</i> ( <i>Num</i> ( <i>str</i> ))	<i>L_Assign_Alloc</i>
		<i>Stackframe</i> ( <i>Num</i> ( <i>str</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>NewStackframe</i> ( <i>Name</i> ( <i>str</i> ), <i>GoTo</i> ( <i>str</i> ))	
		<i>RemoveStackframe</i> ()	
<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 1.3.2:** Abstrakte Syntax für *L<sub>PicoC-Mon</sub>***Definition 1.4: Symboltabelle****1.3.2.3.3 Codebeispiel**

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         // Assign(Name('res'), Num('1'))
8         Exp(Num('1'))
9         Assign(Stackframe(Num('1')), Stack(Num('1')))
10        // While(Num('1'), [])
11        Exp(GoTo(Name('condition_check.5'))))
12      ],
13    Block
14      Name 'condition_check.5',
15      [
16        // IfElse(Num('1'), [], [])
17        Exp(Num('1')),
18        IfElse
19          Stack
20            Num '1',
21            [
22              GoTo(Name('while_branch.4'))
23            ],
24            [
25              GoTo(Name('while_after.1'))

```

```

26     ]
27 ],
28 Block
29   Name 'while_branch.4',
30   [
31     // If(Atom(Name('n'), Eq('=='), Num('1')), [], [])
32     // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
33     Exp(Stackframe(Num('0')))
34     Exp(Num('1'))
35     Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
36     IfElse
37       Stack
38         Num '1',
39         [
40           GoTo(Name('if.3'))
41         ],
42         [
43           GoTo(Name('if_else_after.2'))
44         ]
45     ],
46 Block
47   Name 'if.3',
48   [
49     // Return(Name('res'))
50     Exp(Stackframe(Num('1')))
51     Return(Stack(Num('1')))
52   ],
53 Block
54   Name 'if_else_after.2',
55   [
56     // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
57     Exp(Stackframe(Num('0')))
58     Exp(Stackframe(Num('1')))
59     Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
60     Assign(Stackframe(Num('1')), Stack(Num('1')))
61     // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
62     Exp(Stackframe(Num('0')))
63     Exp(Num('1'))
64     Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
65     Assign(Stackframe(Num('0')), Stack(Num('1')))
66     Exp(GoTo(Name('condition_check.5')))
67   ],
68 Block
69   Name 'while_after.1',
70   [
71     Return(Empty())
72   ],
73 Block
74   Name 'main.0',
75   [
76     StackMalloc(Num('2'))
77     Exp(Num('4'))
78     NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
79     Exp(GoTo(Name('faculty.6')))
80     RemoveStackframe()
81     Exp(ACC)
82     Exp(Call(Name('print'), [Stack(Num('1'))]))

```

```

83     Return(Empty())
84   ]
85 ]

```

**Code 1.8:** *PicoC-ANF Pass für Codebeispiel*

#### 1.3.2.4 RETI-Blocks Pass

##### 1.3.2.4.1 Aufgaben

##### 1.3.2.4.2 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  $L_{RETI\_Blocks}$*

##### 1.3.2.4.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         # // Assign(Name('res'), Num('1'))
8         # Exp(Num('1'))
9         SUBI SP 1;
10        LOADI ACC 1;
11        STOREIN SP ACC 1;
12        # Assign(Stackframe(Num('1')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN BAF ACC -3;
15        ADDI SP 1;
16        # // While(Num('1'), [])
17        # Exp(GoTo(Name('condition_check.5')))
18        Exp(GoTo(Name('condition_check.5')))
19      ],
20    Block
21      Name 'condition_check.5',
22      [
23        # // IfElse(Num('1'), [], [])
24        # Exp(Num('1'))
25        SUBI SP 1;
26        LOADI ACC 1;
27        STOREIN SP ACC 1;
28        # IfElse(Stack(Num('1')), [], [])
29        LOADIN SP ACC 1;

```

```

30     ADDI SP 1;
31     JUMP== GoTo(Name('while_after.1'));
32     Exp(GoTo(Name('while_branch.4')))
33 ],
34 Block
35     Name 'while_branch.4',
36     [
37         # // If(Atom(Name('n'), Eq('=='), Num('1')), [], [])
38         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39         # Exp(Stackframe(Num('0')))
40         SUBI SP 1;
41         LOADIN BAF ACC -2;
42         STOREIN SP ACC 1;
43         # Exp(Num('1'))
44         SUBI SP 1;
45         LOADI ACC 1;
46         STOREIN SP ACC 1;
47         LOADIN SP ACC 2;
48         LOADIN SP IN2 1;
49         SUB ACC IN2;
50         JUMP== 3;
51         LOADI ACC 0;
52         JUMP 2;
53         LOADI ACC 1;
54         STOREIN SP ACC 2;
55         ADDI SP 1;
56         # IfElse(Stack(Num('1')), [], [])
57         LOADIN SP ACC 1;
58         ADDI SP 1;
59         JUMP== GoTo(Name('if_else_after.2'));
60         Exp(GoTo(Name('if.3')))
61     ],
62 Block
63     Name 'if.3',
64     [
65         # // Return(Name('res'))
66         # Exp(Stackframe(Num('1')))
67         SUBI SP 1;
68         LOADIN BAF ACC -3;
69         STOREIN SP ACC 1;
70         # Return(Stack(Num('1')))
71         LOADIN SP ACC 1;
72         ADDI SP 1;
73         LOADIN BAF PC -1;
74     ],
75 Block
76     Name 'if_else_after.2',
77     [
78         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
79         # Exp(Stackframe(Num('0')))
80         SUBI SP 1;
81         LOADIN BAF ACC -2;
82         STOREIN SP ACC 1;
83         # Exp(Stackframe(Num('1')))
84         SUBI SP 1;
85         LOADIN BAF ACC -3;
86         STOREIN SP ACC 1;

```

```

87      # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88      LOADIN SP ACC 2;
89      LOADIN SP IN2 1;
90      MULT ACC IN2;
91      STOREIN SP ACC 2;
92      ADDI SP 1;
93      # Assign(Stackframe(Num('1')), Stack(Num('1')))
94      LOADIN SP ACC 1;
95      STOREIN BAF ACC -3;
96      ADDI SP 1;
97      # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98      # Exp(Stackframe(Num('0')))
99      SUBI SP 1;
100     LOADIN BAF ACC -2;
101     STOREIN SP ACC 1;
102     # Exp(Num('1'))
103     SUBI SP 1;
104     LOADI ACC 1;
105     STOREIN SP ACC 1;
106     # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107     LOADIN SP ACC 2;
108     LOADIN SP IN2 1;
109     SUB ACC IN2;
110     STOREIN SP ACC 2;
111     ADDI SP 1;
112     # Assign(Stackframe(Num('0')), Stack(Num('1')))
113     LOADIN SP ACC 1;
114     STOREIN BAF ACC -2;
115     ADDI SP 1;
116     # Exp(GoTo(Name('condition_check.5')))
117     Exp(GoTo(Name('condition_check.5')))
118 ],
119 Block
120   Name 'while_after.1',
121   [
122     # Return(Empty())
123     LOADIN BAF PC -1;
124   ],
125 Block
126   Name 'main.0',
127   [
128     # StackMalloc(Num('2'))
129     SUBI SP 2;
130     # Exp(Num('4'))
131     SUBI SP 1;
132     LOADI ACC 4;
133     STOREIN SP ACC 1;
134     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
135     MOVE BAF ACC;
136     ADDI SP 3;
137     MOVE SP BAF;
138     SUBI SP 4;
139     STOREIN BAF ACC 0;
140     LOADI ACC GoTo(Name('addr@next_instr'));
141     ADD ACC CS;
142     STOREIN BAF ACC -1;
143     # Exp(GoTo(Name('faculty.6')))

```

```

144     Exp(GoTo(Name('faculty.6'))))
145     # RemoveStackframe()
146     MOVE BAF IN1;
147     LOADIN IN1 BAF 0;
148     MOVE IN1 SP;
149     # Exp(ACC)
150     SUBI SP 1;
151     STOREIN SP ACC 1;
152     LOADIN SP ACC 1;
153     ADDI SP 1;
154     CALL PRINT ACC;
155     # Return(Empty())
156     LOADIN BAF PC -1;
157 ]
158 ]

```

Code 1.9: RETI-Blocks Pass für Codebeispiel

### 1.3.2.5 RETI-Patch Pass

#### 1.3.2.5.1 Aufgaben

#### 1.3.2.5.2 Abstrakte Syntax

---


$$stmt ::= Exit(Num(str))$$


---

Grammar 1.3.4: Abstrakte Syntax für  $L_{RETI-Patch}$ 

#### 1.3.2.5.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.7',
6       [
7         # // Exp(GoTo(Name('main.0'))))
8         Exp(GoTo(Name('main.0'))))
9       ],
10    Block
11      Name 'faculty.6',
12      [
13        # // Assign(Name('res'), Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Assign(Stackframe(Num('1')), Stack(Num('1'))))
19        LOADIN SP ACC 1;
20        STOREIN BAF ACC -3;
21        ADDI SP 1;
22        # // While(Num('1'), [])
23        # Exp(GoTo(Name('condition_check.5'))))

```



```

24     # // not included Exp(GoTo(Name('condition_check.5')))
25 ],
26 Block
27     Name 'condition_check.5',
28     [
29         # // IfElse(Num('1'), [], [])
30         # Exp(Num('1'))
31         SUBI SP 1;
32         LOADI ACC 1;
33         STOREIN SP ACC 1;
34         # IfElse(Stack(Num('1')), [], [])
35         LOADIN SP ACC 1;
36         ADDI SP 1;
37         JUMP== GoTo(Name('while_after.1'));
38         # // not included Exp(GoTo(Name('while_branch.4')))
39     ],
40 Block
41     Name 'while_branch.4',
42     [
43         # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
44         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
45         # Exp(Stackframe(Num('0')))
46         SUBI SP 1;
47         LOADIN BAF ACC -2;
48         STOREIN SP ACC 1;
49         # Exp(Num('1'))
50         SUBI SP 1;
51         LOADI ACC 1;
52         STOREIN SP ACC 1;
53         LOADIN SP ACC 2;
54         LOADIN SP IN2 1;
55         SUB ACC IN2;
56         JUMP== 3;
57         LOADI ACC 0;
58         JUMP 2;
59         LOADI ACC 1;
60         STOREIN SP ACC 2;
61         ADDI SP 1;
62         # IfElse(Stack(Num('1')), [], [])
63         LOADIN SP ACC 1;
64         ADDI SP 1;
65         JUMP== GoTo(Name('if_else_after.2'));
66         # // not included Exp(GoTo(Name('if.3')))
67     ],
68 Block
69     Name 'if.3',
70     [
71         # // Return(Name('res'))
72         # Exp(Stackframe(Num('1')))
73         SUBI SP 1;
74         LOADIN BAF ACC -3;
75         STOREIN SP ACC 1;
76         # Return(Stack(Num('1')))
77         LOADIN SP ACC 1;
78         ADDI SP 1;
79         LOADIN BAF PC -1;
80     ],

```

```

81  Block
82      Name 'if_else_after.2',
83      [
84          # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85          # Exp(Stackframe(Num('0')))
86          SUBI SP 1;
87          LOADIN BAF ACC -2;
88          STOREIN SP ACC 1;
89          # Exp(Stackframe(Num('1')))
90          SUBI SP 1;
91          LOADIN BAF ACC -3;
92          STOREIN SP ACC 1;
93          # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94          LOADIN SP ACC 2;
95          LOADIN SP IN2 1;
96          MULT ACC IN2;
97          STOREIN SP ACC 2;
98          ADDI SP 1;
99          # Assign(Stackframe(Num('1')), Stack(Num('1')))
100         LOADIN SP ACC 1;
101         STOREIN BAF ACC -3;
102         ADDI SP 1;
103         # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104         # Exp(Stackframe(Num('0')))
105         SUBI SP 1;
106         LOADIN BAF ACC -2;
107         STOREIN SP ACC 1;
108         # Exp(Num('1'))
109         SUBI SP 1;
110         LOADI ACC 1;
111         STOREIN SP ACC 1;
112         # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113         LOADIN SP ACC 2;
114         LOADIN SP IN2 1;
115         SUB ACC IN2;
116         STOREIN SP ACC 2;
117         ADDI SP 1;
118         # Assign(Stackframe(Num('0')), Stack(Num('1')))
119         LOADIN SP ACC 1;
120         STOREIN BAF ACC -2;
121         ADDI SP 1;
122         # Exp(GoTo(Name('condition_check.5')))
123         Exp(GoTo(Name('condition_check.5')))
124     ],
125  Block
126      Name 'while_after.1',
127      [
128          # Return(Empty())
129          LOADIN BAF PC -1;
130      ],
131  Block
132      Name 'main.0',
133      [
134          # StackMalloc(Num('2'))
135          SUBI SP 2;
136          # Exp(Num('4'))
137          SUBI SP 1;

```

```

138     LOADI ACC 4;
139     STOREIN SP ACC 1;
140     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
141     MOVE BAF ACC;
142     ADDI SP 3;
143     MOVE SP BAF;
144     SUBI SP 4;
145     STOREIN BAF ACC 0;
146     LOADI ACC GoTo(Name('addr@next_instr'));
147     ADD ACC CS;
148     STOREIN BAF ACC -1;
149     # Exp(GoTo(Name('faculty.6')))
150     Exp(GoTo(Name('faculty.6')))
151     # RemoveStackframe()
152     MOVE BAF IN1;
153     LOADIN IN1 BAF 0;
154     MOVE IN1 SP;
155     # Exp(ACC)
156     SUBI SP 1;
157     STOREIN SP ACC 1;
158     LOADIN SP ACC 1;
159     ADDI SP 1;
160     CALL PRINT ACC;
161     # Return(Empty())
162     LOADIN BAF PC -1;
163 ]
164 ]

```

Code 1.10: RETI-Patch Pass für Codebespiel

### 1.3.2.6 RETI Pass

#### 1.3.2.6.1 Aufgaben

#### 1.3.2.6.2 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</i> <i>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>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  $L_{RETI\_Lex}$



```

16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4')))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0')))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;
43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3;
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;

```

```
73 # Assign(Stackframe(Num('1')), Stack(Num('1'))))
74 LOADIN SP ACC 1;
75 STOREIN BAF ACC -3;
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1'))))
78 # Exp(Stackframe(Num('0'))))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1'))))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5'))))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;
100 # StackMalloc(Num('2'))
101 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr'))))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6'))))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

**Code 1.11:** *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).

## Bücher

- G. Siek, Jeremy. *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).

## Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [https://abs.informatik.uni-freiburg.de/src/teach\\_main.php?id=157](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157) (besucht am 09.07.2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).