

---

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>10</b>
1.1	Lexikalische Analyse	10
1.1.1	Konkrete Syntax für die Lexikalische Analyse	10
1.1.2	Basic Lexer	11
1.2	Syntaktische Analyse	11
1.2.1	Konkrete Syntax für die Syntaktische Analyse	11
1.2.2	Umsetzung von Präzidenz	13
1.2.3	Derivation Tree Generierung	14
1.2.3.1	Early Parser	14
1.2.3.2	Codebeispiel	14
1.2.4	Derivation Tree Vereinfachung	15
1.2.4.1	Visitor	15
1.2.4.2	Codebeispiel	15
1.2.5	Abstrakt Syntax Tree Generierung	17
1.2.5.1	PicoC-Knoten	17
1.2.5.2	RETI-Knoten	22
1.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	23
1.2.5.4	Abstrakte Syntax	25
1.2.5.5	Transformer	27
1.2.5.6	Codebeispiel	27
1.3	Code Generierung	28
1.3.1	Übersicht	28
1.3.2	Passes	29
1.3.2.1	PicoC-Shrink Pass	29
1.3.2.1.1	Codebeispiel	29
1.3.2.2	PicoC-Blocks Pass	30
1.3.2.2.1	Abstrakte Syntax	30
1.3.2.2.2	Codebeispiel	31
1.3.2.3	PicoC-Mon Pass	32
1.3.2.3.1	Abstrakte Syntax	32
1.3.2.3.2	Codebeispiel	32
1.3.2.4	RETI-Blocks Pass	34
1.3.2.4.1	Abstrakte Syntax	34
1.3.2.4.2	Codebeispiel	34
1.3.2.5	RETI-Patch Pass	37
1.3.2.5.1	Abstrakte Syntax	37
1.3.2.5.2	Codebeispiel	37
1.3.2.6	RETI Pass	39
1.3.2.6.1	Konkrete und Abstrakte Syntax	39
1.3.2.6.2	Codebeispiel	40
1.3.3	Umsetzung von Pointern	43
1.3.3.1	Referenzierung	43
1.3.3.2	Dereferenzierung durch Zugriff auf Arrayindex ersetzen	45
1.3.4	Umsetzung von Arrays	46
1.3.4.1	Initialisierung von Arrays	46
1.3.4.2	Zugriff auf einen Arrayindex	51

1.3.4.3	Zuweisung an Arrayindex . . . . .	55
1.3.5	Umsetzung von Structs . . . . .	58
1.3.5.1	Deklaration und Definition von Structtypen . . . . .	58
1.3.5.2	Initialisierung von Structs . . . . .	60
1.3.5.3	Zugriff auf Structattribut . . . . .	63
1.3.5.4	Zuweisung an Structattribut . . . . .	65
1.3.6	Umsetzung der Derived Datatypes im Zusammenspiel . . . . .	67
1.3.6.1	Anfangsteil für Globale Statische Daten und Stackframe . . . . .	67
1.3.6.2	Mittelteil für die verschiedenen Derived Datatypes . . . . .	70
1.3.6.3	Schlusssteil für die verschiedenen Derived Datatypes . . . . .	72
1.3.7	Umsetzung von Funktionen . . . . .	77
1.3.7.1	Funktionen auflösen zu RETI Code . . . . .	77
1.3.7.1.1	Sprung zur Main Funktion . . . . .	79
1.3.7.2	Funktionsdeklaration und -definition und Umsetzung von Scopes . . . . .	81
1.3.7.3	Funktionsaufruf . . . . .	83
1.3.7.3.1	Ohne Rückgabewert . . . . .	83
1.3.7.3.2	Mit Rückgabewert . . . . .	85
1.3.7.3.3	Umsetzung von Call by Sharing für Arrays . . . . .	88
1.3.7.3.4	Umsetzung von Call by Value für Structs . . . . .	91
1.3.8	Umsetzung kleinerer Details . . . . .	92
1.4	Fehlermeldungen . . . . .	92
1.4.1	Error Handler . . . . .	92
1.4.2	Arten von Fehlermeldungen . . . . .	92
1.4.2.1	Syntaxfehler . . . . .	92
1.4.2.2	Laufzeitfehler . . . . .	92

---

---

# Abbildungsverzeichnis

1.1	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	28
1.2	Cross-Compiler Kompiliervorgang Kurzform . . . . .	28
1.3	Architektur mit allen Passes ausgeschrieben . . . . .	29

---

---

# Codeverzeichnis

1.1	PicoC Code für Derivation Tree Generierung . . . . .	14
1.2	Derivation Tree nach Derivation Tree Generierung . . . . .	15
1.3	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	16
1.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert . . . . .	27
1.5	PicoC Code für Codebespiel . . . . .	29
1.6	Abstract Syntax Tree für Codebespiel . . . . .	30
1.7	PicoC Shrink Pass für Codebespiel . . . . .	30
1.8	PicoC-Blocks Pass für Codebespiel . . . . .	32
1.9	PicoC-Mon Pass für Codebespiel . . . . .	34
1.10	RETI-Blocks Pass für Codebespiel . . . . .	36
1.11	RETI-Patch Pass für Codebespiel . . . . .	39
1.12	RETI Pass für Codebespiel . . . . .	42
1.13	PicoC-Code für Pointer Referenzierung . . . . .	43
1.14	Abstract Syntax Tree für Pointer Referenzierung . . . . .	43
1.15	Symboltabelle für Pointer Referenzierung . . . . .	44
1.16	PicoC-Mon Pass für Pointer Referenzierung . . . . .	44
1.17	RETI-Blocks Pass für Pointer Referenzierung . . . . .	45
1.18	PicoC-Code für Pointer Dereferenzierung . . . . .	45
1.19	Abstract Syntax Tree für Pointer Dereferenzierung . . . . .	46
1.20	PicoC-Shrink Pass für Pointer Dereferenzierung . . . . .	46
1.21	PicoC-Code für Array Initialisierung . . . . .	47
1.22	Abstract Syntax Tree für Array Initialisierung . . . . .	47
1.23	Symboltabelle für Array Initialisierung . . . . .	48
1.24	PicoC-Mon Pass für Array Initialisierung . . . . .	49
1.25	RETI-Blocks Pass für Array Initialisierung . . . . .	51
1.26	PicoC-Code für Zugriff auf einen Arrayindex . . . . .	51
1.27	Abstract Syntax Tree für Zugriff auf einen Arrayindex . . . . .	52
1.28	PicoC-Mon Pass für Zugriff auf einen Arrayindex . . . . .	53
1.29	RETI-Blocks Pass für Zugriff auf einen Arrayindex . . . . .	55
1.30	PicoC-Code für Zuweisung an Arrayindex . . . . .	55
1.31	Abstract Syntax Tree für Zuweisung an Arrayindex . . . . .	56
1.32	PicoC-Mon Pass für Zuweisung an Arrayindex . . . . .	57
1.33	RETI-Blocks Pass für Zuweisung an Arrayindex . . . . .	57
1.34	PicoC-Code für die Deklaration eines Structtyps . . . . .	58
1.35	Abstract Syntax Tree für die Deklaration eines Structtyps . . . . .	58
1.36	Symboltabelle für die Deklaration eines Structtyps . . . . .	59
1.37	PicoC-Code für Initialisierung von Structs . . . . .	60
1.38	Abstract Syntax Tree für Initialisierung von Structs . . . . .	60
1.39	Symboltabelle für Initialisierung von Structs . . . . .	62
1.40	PicoC-Mon Pass für Initialisierung von Structs . . . . .	62
1.41	RETI-Blocks Pass für Initialisierung von Structs . . . . .	63
1.42	PicoC-Code für Zugriff auf Structattribut . . . . .	63
1.43	Abstract Syntax Tree für Zugriff auf Structattribut . . . . .	64
1.44	PicoC-Mon Pass für Zugriff auf Structattribut . . . . .	64
1.45	RETI-Blocks Pass für Zugriff auf Structattribut . . . . .	65
1.46	PicoC-Code für Zuweisung an Structattribut . . . . .	65
1.47	Abstract Syntax Tree für Zuweisung an Structattribut . . . . .	66

1.48 PicoC-Mon Pass für Zuweisung an Structattribut . . . . .	66
1.49 RETI-Blocks Pass für Zuweisung an Structattribut . . . . .	67
1.50 PicoC-Code für den Anfangsteil . . . . .	67
1.51 Abstract Syntax Tree für den Anfangsteil . . . . .	68
1.52 PicoC-Mon Pass für den Anfangsteil . . . . .	69
1.53 RETI-Blocks Pass für den Anfangsteil . . . . .	69
1.54 PicoC-Code für den Mittelteil . . . . .	70
1.55 Abstract Syntax Tree für den Mittelteil . . . . .	70
1.56 PicoC-Mon Pass für den Mittelteil . . . . .	71
1.57 RETI-Blocks Pass für den Mittelteil . . . . .	72
1.58 PicoC-Code für den Schlussteil . . . . .	72
1.59 Abstract Syntax Tree für den Schlussteil . . . . .	73
1.60 PicoC-Mon Pass für den Schlussteil . . . . .	74
1.61 RETI-Blocks Pass für den Schlussteil . . . . .	76
1.62 PicoC-Code für 3 Funktionen . . . . .	77
1.63 Abstract Syntax Tree für 3 Funktionen . . . . .	77
1.64 RETI-Blocks Pass für 3 Funktionen . . . . .	78
1.65 PicoC-Mon Pass für 3 Funktionen . . . . .	78
1.66 RETI-Blocks Pass für 3 Funktionen . . . . .	79
1.67 PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	79
1.68 PicoC-Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	80
1.69 RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	80
1.70 PicoC-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist . . . . .	81
1.71 PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	82
1.72 Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss . . . . .	83
1.73 PicoC-Code für Funktionsaufruf ohne Rückgabewert . . . . .	83
1.74 PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert . . . . .	84
1.75 RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert . . . . .	84
1.76 RETI-Pass für Funktionsaufruf ohne Rückgabewert . . . . .	85
1.77 PicoC-Code für Funktionsaufruf mit Rückgabewert . . . . .	85
1.78 PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert . . . . .	86
1.79 RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert . . . . .	87
1.80 RETI-Pass für Funktionsaufruf mit Rückgabewert . . . . .	87
1.81 PicoC-Code für Call by Sharing für Arrays . . . . .	88
1.82 PicoC-Mon Pass für Call by Sharing für Arrays . . . . .	88
1.83 Symboltabelle für Call by Sharing für Arrays . . . . .	89
1.84 RETI-Block Pass für Call by Sharing für Arrays . . . . .	90
1.85 PicoC-Code für Call by Value für Structs . . . . .	91
1.86 PicoC-Mon Pass für Call by Value für Structs . . . . .	91
1.87 RETI-Block Pass für Call by Value für Structs . . . . .	92

---

---

# Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC . . . . .	13
1.2	PicoC-Knoten Teil 1 . . . . .	17
1.3	PicoC-Knoten Teil 2 . . . . .	18
1.4	PicoC-Knoten Teil 3 . . . . .	19
1.5	PicoC-Knoten Teil 4 . . . . .	20
1.6	RETI-Knoten . . . . .	22
1.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung . . . . .	24



---

---

# Definitionsverzeichnis

1.1	Token-Knoten . . . . .	21
1.2	Container-Knoten . . . . .	21
1.3	Symboltabelle . . . . .	32

---

---

# Grammatikverzeichnis

1.1.1 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 1 . . . . .	10
1.1.2 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 2 . . . . .	11
1.2.1 Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1 . . . . .	12
1.2.2 Konkrete Syntax für die Syntaktische Analyse in EBNF, Teil 2 . . . . .	13
1.2.3 Abstrakte Syntax für $L_{PiocC}$ . . . . .	26
1.3.1 Abstrakte Syntax für $L_{PicoC\_Blocks}$ . . . . .	31
1.3.2 Abstrakte Syntax für $L_{PicoC\_Mon}$ . . . . .	32
1.3.3 Abstrakte Syntax für $L_{RETI\_Blocks}$ . . . . .	34
1.3.4 Abstrakte Syntax für $L_{RETI\_Patch}$ . . . . .	37
1.3.5 Konkrete Syntax für $L_{RETI\_Lex}$ . . . . .	39
1.3.6 Konkrete Syntax für $L_{RETI\_Parse}$ . . . . .	40
1.3.7 Abstrakte Syntax für $L_{RETI}$ . . . . .	40

# 1 Implementierung

## 1.1 Lexikalische Analyse

### 1.1.1 Konkrete Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	"//"/ "[\n]*"/   "/*"/ "(. \n)*?"/ "*/"	<i>L_Comment</i>
<i>RETI_COMMENT.2</i>	::=	"//"" "?"#"/ "[\n]*/	
<i>DIG_NO_0</i>	::=	"1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"	<i>L_Arith</i>
<i>DIG_WITH_0</i>	::=	"0"   <i>DIG_NO_0</i>	
<i>NUM</i>	::=	"0"   <i>DIG_NO_0</i> <i>DIG_WITH_0</i> *	
<i>ASCII_CHAR</i>	::=	"_".." ~ "	
<i>CHAR</i>	::=	"'" <i>ASCII_CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII_CHAR</i> + ".picoc"	
<i>LETTER</i>	::=	"a".."z"   "A".."Z"	
<i>NAME</i>	::=	( <i>LETTER</i>   "_") ( <i>LETTER</i> — <i>DIG_WITH_0</i> — "_")*	
<i>name</i>	::=	<i>NAME</i>   <i>INT_NAME</i>   <i>CHAR_NAME</i>   <i>VOID_NAME</i>	
<i>NOT</i>	::=	" ~ "	
<i>REF_AND</i>	::=	"&"	
<i>un_op</i>	::=	<i>SUB_MINUS</i>   <i>LOGIC_NOT</i>   <i>NOT</i>   <i>MUL_DEREF_PNTR</i>   <i>REF_AND</i>	
<i>MUL_DEREF_PNTR</i>	::=	"*"	
<i>DIV</i>	::=	"/"	
<i>MOD</i>	::=	"%"	
<i>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>	::=	"!"	

Grammar 1.1.1: Konkrete Syntax 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: Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 2

### 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>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_op</i> <i>un_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_op</i> <i>rel_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     pntr_decl
10    pntr_deg *
11    array_decl
12    array_dims
13    5
14    6
15    pntr_decl
16    pntr_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    pntr_deg
27    name main
28    fun_params
29    decl_exec_stmts
30    decl_part
31    decl_exp_stmt
32    alloc
33    pntr_decl
34    pntr_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

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</code> !a.
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 $x > 1$ 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: <code>&lt;prim_dt&gt; *&lt;var&gt;</code> , wobei das <b>Attribut</b> <b>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</b> <code>&amp;&lt;var&gt;</code> steht und die <b>Adresse</b> einer <b>Location</b> 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>Indexzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> , <code>Attr(exp, name)</code> oder ein <code>Name('var')</code> 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: <code>&lt;prim_dt&gt; &lt;var&gt;[&lt;i&gt;]</code> , wobei das <b>Attribut</b> <b>nums</b> eine Liste von <code>Num('x')</code> ist, die die <b>Dimensionen</b> des Arrays angibt und <b>datatype</b> der Datentyp ist, der über das Anwenden von <code>Subscript()</code> 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. <code>{1, 2}</code> , <code>{3, 4}</code> . Des Weiteren besitzt er ein verstecktes Attribut <b>datatype</b> , welches für den <b>PicoC-Mon Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
Subscr(exp1, exp2)	Container für den <b>Indexzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten <b>Structdatatype</b> : <code>struct &lt;name&gt;</code> , wobei das <b>Attribut</b> <b>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> : <code>&lt;var&gt;.&lt;attr&gt;</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> 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-Mon 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_goto)	Container für ein <b>If Statement</b> if(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts_goto, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts_goto1, stmts_goto2)	Container für ein <b>If-Else Statement</b> if(<exp>) { <stmts2> } else { <stmts2> } inklusive <b>Condition</b> exp und 2 <b>Branches</b> stmts_goto1 und stmts_goto2, die zwei Alternativen darstellen in denen jeweils <b>Listen von Statements</b> oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts_goto)	Container für ein <b>While-Statement</b> while(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts_goto, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts_goto)	Container für ein <b>Do-While-Statement</b> do { <stmts> } while(<exp>); inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts_goto, 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> , wobei fun_name der Bezeichner der Funktion ist, für die ein neuer <b>Stackframe</b> erstellt werden soll und später dazu dient den <b>Block</b> dieser Funktion zu finden, weil dieser für den weiteren Kompilervorang wichtige Information in seinen versteckten Attribute angehängt hat und goto_after_call ein GoTo(Name('addr@next_instr')) ist, welches später durch die <b>Adresse</b> der Instruction, die direkt auf die <b>Jump Instruction</b> folgt, ersetzt wird.
RemoveStackframe()	Container für das <b>Entfernen</b> des aktuellen <b>Stackframes</b> .
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 Bezeichners des <b>Labels</b> des Blocks ist und stmts_instrs eine <b>Liste von Statements</b> oder <b>Instructions</b> . Zudem besitzt er noch 3 versteckte Attribute, 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>Kompilervorangs</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

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 1.1) 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
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>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(Alloc(Writeable(), PtrDecl(Num('1')), ArrayDecl([Num('3'), Num('2')],
27          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var')))
28    ]
29  ]
```

Code 1.4: Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert

## 1.3 Code Generierung

### 1.3.1 Übersicht

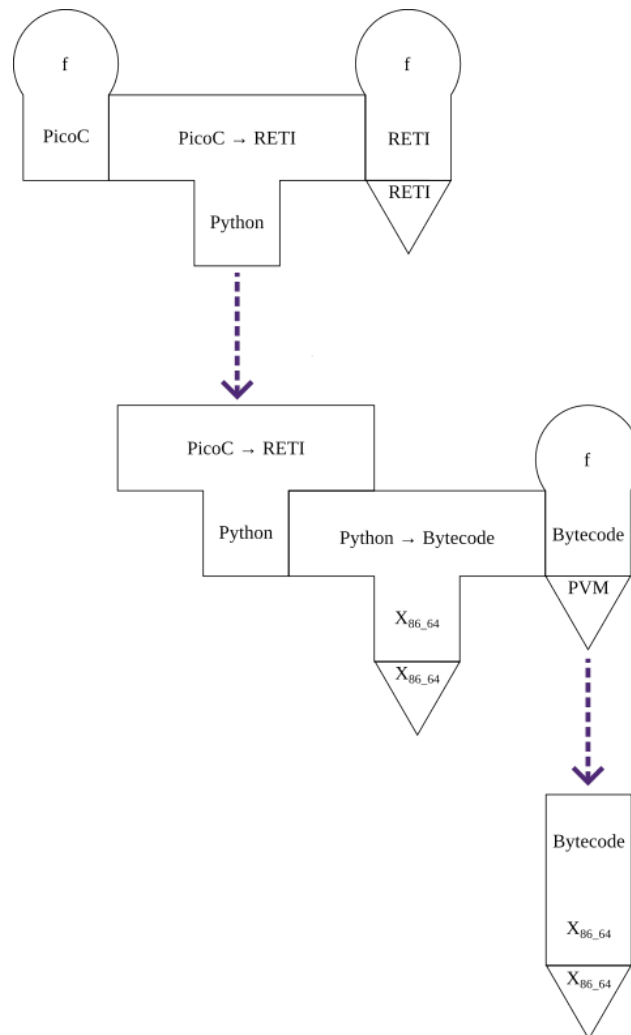


Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

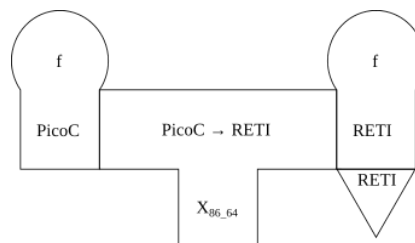


Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

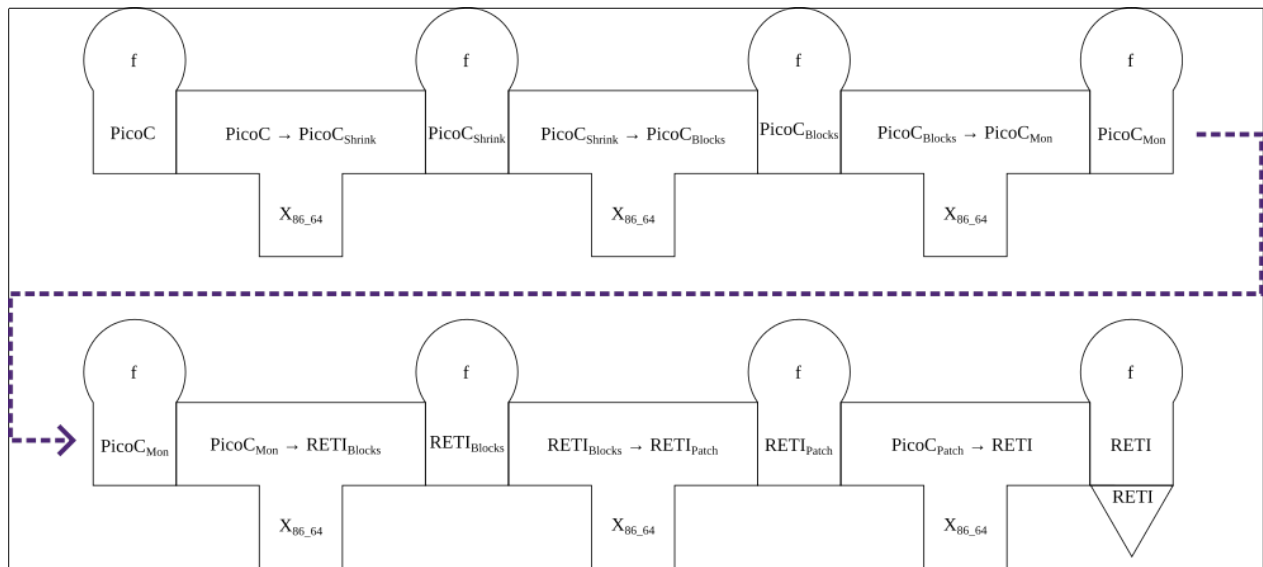


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

## 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(Alloc(Writable(), IntType('int'), Name('n')), Num('4'))
10     Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
11     While
12       Num '1',
13       [
14         If
15         Atom
16           Name 'n',
17           Eq '==',
18           Num '1',
19           [
20             Return(Empty())
21           ]
22         Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23         Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24       ]
25   ]
26 ]

```

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(Alloc(Writable(), IntType('int'), Name('n')), Num('4'))
10        Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
11        While
12          Num '1',
13          [
14            If
15            Atom
16              Name 'n',
17              Eq '==',
18              Num '1',
19              [
20                Return(Empty())
21              ]
22            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
23            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
24          ]
25        ]
26      ]

```

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>::=</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>::=</i>	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), $\langle stmt \rangle$ *)	<i>L_Blocks</i>
<i>stmt</i>	<i>::=</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 *LPicoC\_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(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
13            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
14            // While(Num('1'), [])
15            GoTo(Name('condition_check.4'))
16          ],
17          Block
18            Name 'condition_check.4',
19            [
20              IfElse
21                Num '1',
22                [
23                  GoTo(Name('while_branch.3'))
24                ],
25                [
26                  GoTo(Name('while_after.0'))
27                ]
28            ],
29            Block
30              Name 'while_branch.3',
31              [
32                // If(Atom(Name('n'), Eq('=='), Num('1')), []),
33                IfElse
34                  Atom
35                    Name 'n',
36                    Eq '==',
37                    Num '1',
38                    [
39                      GoTo(Name('if.2'))
40                    ],
41                    [
42                      GoTo(Name('if_else_after.1'))
43                    ]

```



```

44     ],
45     Block
46     Name 'if.2',
47     [
48         Return(Empty())
49     ],
50     Block
51     Name 'if_else_after.1',
52     [
53         Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
54         Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
55         GoTo(Name('condition_check.4'))
56     ],
57     Block
58     Name 'while_after.0',
59     []
60 ]
61 ]

```

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>	::= <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>Writeable</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>Writeable</i> ()), <i>ArrayDecl</i> ( <i>Num</i> ( <i>str</i> )+, $\langle datatype \rangle$ ),	
	<i>Name</i> ( <i>str</i> ), <i>Array</i> ( $\langle exp \rangle$ +, $\langle datatype \rangle$ ))	
<i>symbol_table</i>	::= <i>SymbolTable</i> ( $\langle symbol \rangle$ )	<i>L_Symbol_Table</i>
<i>symbol</i>	::= <i>Symbol</i> ( $\langle type\_qual \rangle$ , $\langle datatype \rangle$ , $\langle name \rangle$ , $\langle val \rangle$ , $\langle pos \rangle$ , $\langle size \rangle$ )	
<i>type_qual</i>	::= <i>Empty</i> ()	
<i>datatype</i>	::= <i>BuiltIn</i> ()   <i>SelfDefined</i> ()	
<i>name</i>	::= <i>Name</i> ( <i>str</i> )	
<i>val</i>	::= <i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	
<i>pos</i>	::= <i>Pos</i> ( <i>Num</i> ( <i>str</i> ), <i>Num</i> ( <i>str</i> ))   <i>Empty</i> ()	
<i>size</i>	::= <i>Num</i> ( <i>str</i> )   <i>Empty</i> ()	

Grammar 1.3.2: 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(Num('4'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('res'), Num('1'))
11        Exp(Num('1'))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        // While(Num('1'), [])
14        Exp(GoTo(Name('condition_check.4'))))
15      ],
16    Block
17      Name 'condition_check.4',
18      [
19        // IfElse(Num('1'), [], [])
20        Exp(Num('1')),
21        IfElse
22          Stack
23            Num '1',
24            [
25              GoTo(Name('while_branch.3'))
26            ],
27            [
28              GoTo(Name('while_after.0'))
29            ]
30        ],
31      Block
32        Name 'while_branch.3',
33        [
34          // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35          // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36          Exp(Global(Num('0')))
37          Exp(Num('1'))
38          Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
39          IfElse
40            Stack
41              Num '1',
42              [
43                GoTo(Name('if.2'))
44              ],
45              [
46                GoTo(Name('if_else_after.1'))
47              ]
48        ],
49      Block
50        Name 'if.2',
51        [
52          Return(Empty())
53        ],
54      Block
55        Name 'if_else_after.1',
56        [
57          // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))

```

```

58     Exp(Global(Num('0')))
59     Exp(Global(Num('1')))
60     Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
61     Assign(Global(Num('1')), Stack(Num('1')))
62     // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
63     Exp(Global(Num('0')))
64     Exp(Num('1'))
65     Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
66     Assign(Global(Num('0')), Stack(Num('1')))
67     Exp(GoTo(Name('condition_check.4')))
68 ],
69 Block
70   Name 'while_after.0',
71   [
72     Return(Empty())
73   ]
74 ]

```

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(Global(Num('0')), Stack(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(Global(Num('1')), Stack(Num('1')))
22     LOADIN SP ACC 1;
23     STOREIN DS ACC 1;
24     ADDI SP 1;
25     # // While(Num('1'), [])
26     # Exp(GoTo(Name('condition_check.4')))
27     Exp(GoTo(Name('condition_check.4')))
28 ],
29 Block
30     Name 'condition_check.4',
31     [
32         # // IfElse(Num('1'), [], [])
33         # Exp(Num('1'))
34         SUBI SP 1;
35         LOADI ACC 1;
36         STOREIN SP ACC 1;
37         # IfElse(Stack(Num('1')), [], [])
38         LOADIN SP ACC 1;
39         ADDI SP 1;
40         JUMP== GoTo(Name('while_after.0'));
41         Exp(GoTo(Name('while_branch.3')))
42     ],
43 Block
44     Name 'while_branch.3',
45     [
46         # // If(Atom(Name('n'), Eq('='), Num('1')), [])
47         # // IfElse(Atom(Name('n'), Eq('='), Num('1')), [], [])
48         # Exp(Global(Num('0')))
49         SUBI SP 1;
50         LOADIN DS ACC 0;
51         STOREIN SP ACC 1;
52         # Exp(Num('1'))
53         SUBI SP 1;
54         LOADI ACC 1;
55         STOREIN SP ACC 1;
56         # Exp(Atom(Stack(Num('2')), Eq('='), Stack(Num('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(Stack(Num('1')), [], [])
67         LOADIN SP ACC 1;
68         ADDI SP 1;
69         JUMP== GoTo(Name('if_else_after.1'));
70         Exp(GoTo(Name('if.2')))
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(Global(Num('0')))
83         SUBI SP 1;
84         LOADIN DS ACC 0;
85         STOREIN SP ACC 1;
86         # Exp(Global(Num('1')))
87         SUBI SP 1;
88         LOADIN DS ACC 1;
89         STOREIN SP ACC 1;
90         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
91         LOADIN SP ACC 2;
92         LOADIN SP IN2 1;
93         MULT ACC IN2;
94         STOREIN SP ACC 2;
95         ADDI SP 1;
96         # Assign(Global(Num('1')), Stack(Num('1')))
97         LOADIN SP ACC 1;
98         STOREIN DS ACC 1;
99         ADDI SP 1;
100        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
101        # Exp(Global(Num('0')))
102        SUBI SP 1;
103        LOADIN DS ACC 0;
104        STOREIN SP ACC 1;
105        # Exp(Num('1'))
106        SUBI SP 1;
107        LOADI ACC 1;
108        STOREIN SP ACC 1;
109        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
110        LOADIN SP ACC 2;
111        LOADIN SP IN2 1;
112        SUB ACC IN2;
113        STOREIN SP ACC 2;
114        ADDI SP 1;
115        # Assign(Global(Num('0')), Stack(Num('1')))
116        LOADIN SP ACC 1;
117        STOREIN DS ACC 0;
118        ADDI SP 1;
119        # Exp(GoTo(Name('condition_check.4')))
120        Exp(GoTo(Name('condition_check.4')))
121    ],
122    Block
123        Name 'while_after.0',
124        [
125            # Return(Empty())
126            LOADIN BAF PC -1;
127        ]
128 ]

```

Code 1.10: RETI-Blocks Pass für Codebespiel

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

```

```

46     JUMP== GoTo(Name('while_after.0'));
47     # // patched out Exp(GoTo(Name('while_branch.3')))
48 ],
49 Block
50     Name 'while_branch.3',
51     [
52         # // If(Atom(Name('n'), Eq('=='), Num('1')), [], [])
53         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
54         # Exp(Global(Num('0')))
55         SUBI SP 1;
56         LOADIN DS ACC 0;
57         STOREIN SP ACC 1;
58         # Exp(Num('1'))
59         SUBI SP 1;
60         LOADI ACC 1;
61         STOREIN SP ACC 1;
62         # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
63         LOADIN SP ACC 2;
64         LOADIN SP IN2 1;
65         SUB ACC IN2;
66         JUMP== 3;
67         LOADI ACC 0;
68         JUMP 2;
69         LOADI ACC 1;
70         STOREIN SP ACC 2;
71         ADDI SP 1;
72         # IfElse(Stack(Num('1')), [], [])
73         LOADIN SP ACC 1;
74         ADDI SP 1;
75         JUMP== GoTo(Name('if_else_after.1'));
76         # // patched out Exp(GoTo(Name('if.2')))
77     ],
78 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(Global(Num('0')))
89         SUBI SP 1;
90         LOADIN DS ACC 0;
91         STOREIN SP ACC 1;
92         # Exp(Global(Num('1')))
93         SUBI SP 1;
94         LOADIN DS ACC 1;
95         STOREIN SP ACC 1;
96         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
97         LOADIN SP ACC 2;
98         LOADIN SP IN2 1;
99         MULT ACC IN2;
100        STOREIN SP ACC 2;
101        ADDI SP 1;
102        # Assign(Global(Num('1')), Stack(Num('1')))

```

```

103     LOADIN SP ACC 1;
104     STOREIN DS ACC 1;
105     ADDI SP 1;
106     # // Assign(Name('n'), BinOp(Name('n'), Sub('-', Num('1'))))
107     # Exp(Global(Num('0'))))
108     SUBI SP 1;
109     LOADIN DS ACC 0;
110     STOREIN SP ACC 1;
111     # Exp(Num('1'))
112     SUBI SP 1;
113     LOADI ACC 1;
114     STOREIN SP ACC 1;
115     # Exp(BinOp(Stack(Num('2')), Sub('-', Stack(Num('1')))))
116     LOADIN SP ACC 2;
117     LOADIN SP IN2 1;
118     SUB ACC IN2;
119     STOREIN SP ACC 2;
120     ADDI SP 1;
121     # Assign(Global(Num('0')), Stack(Num('1'))))
122     LOADIN SP ACC 1;
123     STOREIN DS ACC 0;
124     ADDI SP 1;
125     # Exp(GoTo(Name('condition_check.4'))))
126     Exp(GoTo(Name('condition_check.4'))))
127 ],
128 Block
129   Name 'while_after.0',
130   [
131     # Return(Empty())
132     LOADIN BAF PC -1;
133   ]
134 ]

```

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</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 STOREIN SP ACC 1;
17 # Assign(Global(Num('1')), Stack(Num('1')))
18 LOADIN SP ACC 1;
19 STOREIN DS ACC 1;
20 ADDI SP 1;
21 # // While(Num('1'), [])
22 # Exp(GoTo(Name('condition_check.4'))))
23 # // patched out Exp(GoTo(Name('condition_check.4'))))
24 # // IfElse(Num('1'), [], [])
25 # Exp(Num('1'))
26 SUBI SP 1;
27 LOADI ACC 1;
28 STOREIN SP ACC 1;
29 # IfElse(Stack(Num('1')), [], [])
30 LOADIN SP ACC 1;
31 ADDI SP 1;
32 JUMP== 49;
33 # // patched out Exp(GoTo(Name('while_branch.3'))))
34 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36 # Exp(Global(Num('0'))))
37 SUBI SP 1;
38 LOADIN DS ACC 0;
39 STOREIN SP ACC 1;
40 # Exp(Num('1'))
41 SUBI SP 1;
42 LOADI ACC 1;
43 STOREIN SP ACC 1;
44 # Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))
45 LOADIN SP ACC 2;
46 LOADIN SP IN2 1;
47 SUB ACC IN2;
48 JUMP== 3;
49 LOADI ACC 0;
50 JUMP 2;
51 LOADI ACC 1;
52 STOREIN SP ACC 2;
53 ADDI SP 1;
54 # IfElse(Stack(Num('1')), [], [])
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 JUMP== 2;
58 # // patched out Exp(GoTo(Name('if.2'))))
59 # Return(Empty())
60 LOADIN BAF PC -1;
61 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
62 # Exp(Global(Num('0'))))
63 SUBI SP 1;
64 LOADIN DS ACC 0;
65 STOREIN SP ACC 1;
66 # Exp(Global(Num('1'))))
67 SUBI SP 1;
68 LOADIN DS ACC 1;
69 STOREIN SP ACC 1;
70 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
71 LOADIN SP ACC 2;
72 LOADIN SP IN2 1;

```

```
73 MULT ACC IN2;
74 STOREIN SP ACC 2;
75 ADDI SP 1;
76 # Assign(Global(Num('1')), Stack(Num('1')))
77 LOADIN SP ACC 1;
78 STOREIN DS ACC 1;
79 ADDI SP 1;
80 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
81 # Exp(Global(Num('0')))
82 SUBI SP 1;
83 LOADIN DS ACC 0;
84 STOREIN SP ACC 1;
85 # Exp(Num('1'))
86 SUBI SP 1;
87 LOADI ACC 1;
88 STOREIN SP ACC 1;
89 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
90 LOADIN SP ACC 2;
91 LOADIN SP IN2 1;
92 SUB ACC IN2;
93 STOREIN SP ACC 2;
94 ADDI SP 1;
95 # Assign(Global(Num('0')), Stack(Num('1')))
96 LOADIN SP ACC 1;
97 STOREIN DS ACC 0;
98 ADDI SP 1;
99 # Exp(GoTo(Name('condition_check.4')))
100 JUMP -53;
101 # Return(Empty())
102 LOADIN BAF PC -1;
```

Code 1.12: RETI Pass für Codebespiel

### 1.3.3 Umsetzung von Pointern

#### 1.3.3.1 Referenzierung

Die **Referenzierung** (z.B. `&var`) wird im Folgenden anhand des Beispiels in Code 1.13 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4 }
```

Code 1.13: PicoC-Code für Pointer Referenzierung

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstract Syntax Tree** in Code 1.14 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Pointer `*pntr`.

```
1 File
2   Name './example_pntr_ref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
11              ↪ Ref(Name('var')))
12      ]
13  ]
```

Code 1.14: Abstract Syntax Tree für Pointer Referenzierung

Bevor man einem **Pointer** eine **Adresse** (z.B. `&var`) zuweisen kann, muss dieser erstmal **definiert** sein. Dafür braucht es einen Eintrag in der **Symboltabelle** in Code 1.15.

Die **Größe** eines Pointers (z.B. eines Pointers auf ein Array von `int`: `pntr = int *pntr[3]`), die ihm **size**-Feld der **Symboltabelle** eingetragen ist, ist dabei immer: `size(pntr) = 1`.

```
1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(VoidType('void'), Name('main'), [])
7       name:                Name('main')
8       value or address:    Empty()
9       position:            Pos(Num('1'), Num('5'))
10      size:                 Empty()
```

```

11     },
12     Symbol
13     {
14         type qualifier:      Writeable()
15         datatype:            IntType('int')
16         name:                Name('var@main')
17         value or address:    Num('0')
18         position:            Pos(Num('2'), Num('6'))
19         size:                Num('1')
20     },
21     Symbol
22     {
23         type qualifier:      Writeable()
24         datatype:            PtrDecl(Num('1'), IntType('int'))
25         name:                Name('ptr@main')
26         value or address:    Num('1')
27         position:            Pos(Num('3'), Num('7'))
28         size:                Num('1')
29     }
30 ]

```

Code 1.15: Symboltabelle für Pointer Referenzierung

Im **PicoC-Mon Pass** in Code 1.16 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')), Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp)` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var')))` ist, sind noch weitere Anweisungen zwischen den Zeilen 11 und 12 nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 1.3.6.2 kümmern.

```

1 File
2   Name './example_ptr_ref.picoc_mon',
3   [
4     Block
5     Name 'main.0',
6     [
7       // Assign(Name('var'), Num('42'))
8       Exp(Num('42'))
9       Assign(Global(Num('0')), Stack(Num('1')))
10      // Assign(Name('ptr'), Ref(Name('var')))
11      Ref(Global(Num('0')))
12      Assign(Global(Num('1')), Stack(Num('1')))
13      Return(Empty())
14    ]
15  ]

```

Code 1.16: PicoC-Mon Pass für Pointer Referenzierung

Im **RETI-Blocks Pass** in Code 1.17 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_pntr_ref.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('pntr'), Ref(Name('var')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29    ]

```

Code 1.17: RETI-Blocks Pass für Pointer Referenzierung

### 1.3.3.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

Die **Dereferenzierung** (z.B. `*var`) wird im Folgenden anhand des Beispiels in Code 1.18 erklärt.

```

1 void main() {
2   int var = 42;
3   int *pntr = &var;
4   *pntr;
5 }

```

Code 1.18: PicoC-Code für Pointer Dereferenzierung

Der Knoten `Deref(Name('var'))` repräsentiert im **Abstract Syntax Tree** in Code 1.19 eine **Dereferenzierung** `*var`.

```

1 File
2   Name './example_pntr_deref.ast',
3   [
4     FunDef

```

```

5      VoidType 'void',
6      Name 'main',
7      [],
8      [
9          Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10         Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11             ↪ Ref(Name('var')))
12         Exp(Deref(Name('ptr'), Num('0')))
13     ]

```

Code 1.19: Abstract Syntax Tree für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 1.20 wird ein Trick angewendet, bei dem jeder Knoten `Deref(Name('ptr'), Num('0'))` einfach durch den Knoten `Subscr(Name('ptr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Arrayindex** (z.B. `var[1]`) verhält<sup>2</sup>. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Dereferenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Arrayindex** (z.B. `var[1]`) übernehmen lassen.

```

1 File
2   Name './example_ptr_deref.picoc_shrink',
3   [
4       FunDef
5         VoidType 'void',
6         Name 'main',
7         [],
8         [
9             Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10            Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11                ↪ Ref(Name('var')))
12            Exp(Subscr(Name('ptr'), Num('0')))
13        ]
14    ]

```

Code 1.20: PicoC-Shrink Pass für Pointer Dereferenzierung

## 1.3.4 Umsetzung von Arrays

### 1.3.4.1 Initialisierung von Arrays

Die **Initialisierung** eines **Arrays** (z.B. `int ar[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 1.21 erklärt.

<sup>2</sup>In der Sprache  $L_C$  gibt es einen Unterschied bei der Initialisierung bei z.B. `int *var = "string"` und z.B. `int var[1] = "string"`, der allerdings nichts mit den beiden Operatoren zu tun hat, sondern mit der **Initialisierung**, bei der die Sprache  $L_C$  verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Arrayindex**, vor den Bezeichner schreibt (z.B. `var[1]`), obwohl es ein **Derived Datatype** ist.

```

1 void main() {
2   int ar[2][1] = {{3+1}, {4}};
3 }
4
5 void fun() {
6   int ar[2][2] = {{3, 4}, {5, 6}};
7 }

```

Code 1.21: PicoC-Code für Array Initialisierung

Die **Initialisierung** eines **Arrays** `int ar[2][1] = {{3+1}, {4}}` wird im **Abstract Syntax Tree** in Code 1.22 mithilfe der Komposition `Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')), Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')]))])` dargestellt.

```

1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
10          ↪ Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
11          ↪ Array([Num('4')]))])
12       ],
13     FunDef
14       VoidType 'void',
15       Name 'fun',
16       [],
17       [
18         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
19          ↪ Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')]))])
20       ]
21   ]

```

Code 1.22: Abstract Syntax Tree für Array Initialisierung

Bei der **Initialisierung** eines **Arrays** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann<sup>3</sup>. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 1.23 dargestellt ist.

Bei Variablen auf dem **Stackframe** wird ein Array **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Arrays genommen. Dies macht den **Zugriff auf einen Arrayindex** in Subkapitel 1.3.4.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf einen Arrayindex** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die entgegengesetzte Richtung wächst, verglichen mit den **Globalen**

<sup>3</sup>Das widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.



Statischen Daten<sup>4</sup>.

Das **Größe** des Arrays datatype ar[s<sub>1</sub>]...[s<sub>k</sub>], die ihm **size**-Feld der **Symboltabelle** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Arrays multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen Arrayelemente:  $\text{size}(\text{ar}) = (\prod_{j=1}^n s_j) \cdot \text{size}(\text{datatype})$ .

```

1 SymbolTable
2 [
3   Symbol
4   {
5     type qualifier:      Empty()
6     datatype:            FunDecl(VoidType('void'), Name('main'), [])
7     name:                Name('main')
8     value or address:    Empty()
9     position:            Pos(Num('1'), Num('5'))
10    size:                 Empty()
11  },
12  Symbol
13  {
14    type qualifier:       Writeable()
15    datatype:             ArrayDecl([Num('2'), Num('1')], IntType('int'))
16    name:                 Name('ar@main')
17    value or address:     Num('0')
18    position:             Pos(Num('2'), Num('6'))
19    size:                 Num('2')
20  },
21  Symbol
22  {
23    type qualifier:       Empty()
24    datatype:             FunDecl(VoidType('void'), Name('fun'), [])
25    name:                 Name('fun')
26    value or address:     Empty()
27    position:             Pos(Num('5'), Num('5'))
28    size:                 Empty()
29  },
30  Symbol
31  {
32    type qualifier:       Writeable()
33    datatype:             ArrayDecl([Num('2'), Num('2')], IntType('int'))
34    name:                 Name('ar@fun')
35    value or address:     Num('3')
36    position:             Pos(Num('6'), Num('6'))
37    size:                 Num('4')
38  }
39 ]

```

Code 1.23: Symboltabelle für Array Initialisierung

Im **Piocc-Mon Pass** in Code 1.24 werden zuerst die **Ausdrücke** im **Array-Initializer** `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` nach dem **Depth-First**

<sup>4</sup>Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

**Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben<sup>5</sup>.

Im finalen Schritt muss zwischen **Globalen Statischen Daten** bei der **main**-Funktion und **Stackframe** bei der Funktion **fun** unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition **Assign(Global(Num('0')), Stack(Num('2')))** bzw. **Assign(Stackframe(Num('3')), Stack(Num('4')))**, die in Tabelle 1.7 genauer beschrieben ist, versetzt in der selben Reihenfolge zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.

Der **Trick** ist hier, dass egal wieviele Dimensionen und was für einen Datentyp das **Array** hat, man letztendlich immer das gesamte Array erwischt, wenn man einfach die **Größe des Arrays** viele **Speicherzellen** mit z.B. der **Komposition** **Assign(Global(Num('0')), Stack(Num('2')))** verschiebt.

In die Knoten **Global('0')** und **Stackframe('3')** wurde hierbei die **Startadresse** des jeweiligen Arrays geschrieben, sodass man nach dem **PicoC-Mon Pass** nie mehr Variablen in der **Symboltabelle** nachsehen muss und gleich weiß, ob sie in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

```

1 File
2   Name './example_array_init.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
8         ↪   Array([Num('4')])]))
9         Exp(Num('3'))
10        Exp(Num('1'))
11        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
12        Exp(Num('4'))
13        Assign(Global(Num('0')), Stack(Num('2')))
14        Return(Empty())
15      ],
16    Block
17      Name 'fun.0',
18      [
19        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
20        ↪   Num('6')])]))
21        Exp(Num('3'))
22        Exp(Num('4'))
23        Exp(Num('5'))
24        Exp(Num('6'))
25        Assign(Stackframe(Num('3')), Stack(Num('4')))
26        Return(Empty())
27      ]
28    ]
29  ]

```

Code 1.24: PicoC-Mon Pass für Array Initialisierung

Im **RETI-Blocks Pass** in Code 1.25 werden die **PicoC-Knoten** bzw. **Kompositionen** für die Ausdrücke **Exp(exp)** und **Assign(Global(Num('0')), Stack(Num('2')))** bzw. **Assign(Stackframe(Num('3')), Stack(Num('4')))** durch ihre entsprechenden **RETI-Knoten** ersetzt.

<sup>5</sup>Da der **Zuweisungsoperator** = **rechtsassoziativ** ist und auch rein **logisch**, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

```

1 File
2   Name './example_array_init.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Array([BinOp(Num('3')), Add('+'), Num('1'))]),
8           ↪   Array([Num('4')]))])
9         # Exp(Num('3'))
10        SUBI SP 1;
11        LOADI ACC 3;
12        STOREIN SP ACC 1;
13        # Exp(Num('1'))
14        SUBI SP 1;
15        LOADI ACC 1;
16        STOREIN SP ACC 1;
17        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
18        LOADIN SP ACC 2;
19        LOADIN SP IN2 1;
20        ADD ACC IN2;
21        STOREIN SP ACC 2;
22        ADDI SP 1;
23        # Exp(Num('4'))
24        SUBI SP 1;
25        LOADI ACC 4;
26        STOREIN SP ACC 1;
27        # Assign(Global(Num('0')), Stack(Num('2')))
28        LOADIN SP ACC 1;
29        STOREIN DS ACC 1;
30        LOADIN SP ACC 2;
31        STOREIN DS ACC 0;
32        ADDI SP 2;
33        # Return(Empty())
34        LOADIN BAF PC -1;
35      ],
36    Block
37      Name 'fun.0',
38      [
39        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
40          ↪   Num('6')]))])
41        # Exp(Num('3'))
42        SUBI SP 1;
43        LOADI ACC 3;
44        STOREIN SP ACC 1;
45        # Exp(Num('4'))
46        SUBI SP 1;
47        LOADI ACC 4;
48        STOREIN SP ACC 1;
49        # Exp(Num('5'))
50        SUBI SP 1;
51        LOADI ACC 5;
52        STOREIN SP ACC 1;
53        # Exp(Num('6'))
54        SUBI SP 1;
55        LOADI ACC 6;
56        STOREIN SP ACC 1;
57        # Assign(Stackframe(Num('3')), Stack(Num('4')))

```

```

56     LOADIN SP ACC 1;
57     STOREIN BAF ACC -2;
58     LOADIN SP ACC 2;
59     STOREIN BAF ACC -3;
60     LOADIN SP ACC 3;
61     STOREIN BAF ACC -4;
62     LOADIN SP ACC 4;
63     STOREIN BAF ACC -5;
64     ADDI SP 4;
65     # Return(Empty())
66     LOADIN BAF PC -1;
67 ]
68 ]

```

Code 1.25: RETI-Blocks Pass für Array Initialisierung

### 1.3.4.2 Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 1.26 erklärt.

```

1 void main() {
2     int ar[1] = {42};
3     ar[0];
4 }
5
6 void fun() {
7     int ar[3] = {1, 2, 3};
8     ar[1+1];
9 }

```

Code 1.26: PicoC-Code für Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im **Abstract Syntax Tree** in Code 1.27 mithilfe des **Container-Knotens** `Subscr(Name('ar'), Num('0'))` dargestellt.

```

1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
10              ↪ Array([Num('42')]))
11         Exp(Subscr(Name('ar'), Num('0'))))
12       ],
13     FunDef
14       VoidType 'void',
15       Name 'fun',

```

```

15     [],
16     [
17         Assign(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
18             ↪ Array([Num('1'), Num('2'), Num('3')]))
19         Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
20     ]

```

Code 1.27: Abstract Syntax Tree für Zugriff auf einen Arrayindex

Im **PicoC-Mon Pass** in Code 1.28 wird vom **Container-Knoten** `Subscr(Name('ar'), Num('0'))` zuerst im **Anfangsteil 1.3.6.1** die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktion `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil 1.3.6.2** wird die Adresse des **Index**, des Arrays auf das Zugriffen werden soll berechnet. Da der **Index** auf den Zugriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `ar[1 + var]` bestimmt sein kann, indem auch **Variablen** vorkommen können, kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle 1.7 dokumentiert.

Je nachdem, ob mehrere `Subscr(exp, exp)` eine Komposition bilden (z.B. `Subscr(Subscr(Name('var'), Num('1')), Num('1'))`) ist es notwendig mehrere **Adressberechnungsschritte für den Index** `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` einzuleiten und es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und eine **Zugriff auf einen Arrayindex** `var[1]` miteinander zu kombinieren, was in Subkapitel 1.3.6.2 allgemein erklärt ist.

Im letzten Schritt, dem **Schlussenteil 1.3.6.3** wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde, nun auf den **Stack** geschrieben, wobei dieser die **Adresse** auf dem Stack ersetzt, die es zum Finden des **Index** brauchte. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. Je nachdem, welchen **Datentyp** die Variable `ar` hat und auf welchen **Subdatentyp**, welcher ein **verstecktes Attribut** des `Exp(Stack(Num('1')))` Knoten ist folglich im **Kontext** zuletzt zugegriffen wird, abhängig davon wird der **Schlussenteil** `Exp(Stack(Num('1')))` auf eine andere Weise verarbeitet (siehe Subkapitel 1.3.6.3).

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Arrayindex** (z.B. `ar[1]`) in der `main`-Funktion oder der Funktion `fun` erfolgt, ist eigentlich nur beim **Anfangsteil**, beim Schreiben der **Adresse** der Variable `ar` auf den **Stack** zu finden, bei dem unterschiedliche **RETI-Instructions** für eine Variable, die in den **Globalen Statischen Daten** liegt und eine Variable, die auf dem **Stackframe** liegt erzeugt werden müssen.

```

1 File
2   Name './example_array_access.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [

```

```

7      // Assign(Name('ar'), Array([Num('42')]))
8      Exp(Num('42'))
9      Assign(Global(Num('0')), Stack(Num('1')))
10     // Exp(Subscr(Name('ar'), Num('0')))
11     Ref(Global(Num('0')))
12     Exp(Num('0'))
13     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
14     Exp(Stack(Num('1')))
15     Return(Empty())
16 ],
17 Block
18   Name 'fun.0',
19   [
20     // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
21     Exp(Num('1'))
22     Exp(Num('2'))
23     Exp(Num('3'))
24     Assign(Stackframe(Num('2')), Stack(Num('3')))
25     // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
26     Ref(Stackframe(Num('2')))
27     Exp(Num('1'))
28     Exp(Num('1'))
29     Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
30     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
31     Exp(Stack(Num('1')))
32     Return(Empty())
33   ]
34 ]

```

Code 1.28: PicoC-Mon Pass für Zugriff auf einen Arrayindex

Im **RETI-Blocks Pass** in Code 1.29 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Stack(Num('1'))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_array_access.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Exp(Subscr(Name('ar'), Num('0')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;

```

```

20     ADD IN1 DS;
21     STOREIN SP IN1 1;
22     # Exp(Num('0'))
23     SUBI SP 1;
24     LOADI ACC 0;
25     STOREIN SP ACC 1;
26     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27     LOADIN SP IN1 2;
28     LOADIN SP IN2 1;
29     MULTI IN2 1;
30     ADD IN1 IN2;
31     ADDI SP 1;
32     STOREIN SP IN1 1;
33     # Exp(Stack(Num('1'))))
34     LOADIN SP IN1 1;
35     LOADIN IN1 ACC 0;
36     STOREIN SP ACC 1;
37     # Return(Empty())
38     LOADIN BAF PC -1;
39 ],
40 Block
41     Name 'fun.0',
42     [
43         # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44         # Exp(Num('1'))
45         SUBI SP 1;
46         LOADI ACC 1;
47         STOREIN SP ACC 1;
48         # Exp(Num('2'))
49         SUBI SP 1;
50         LOADI ACC 2;
51         STOREIN SP ACC 1;
52         # Exp(Num('3'))
53         SUBI SP 1;
54         LOADI ACC 3;
55         STOREIN SP ACC 1;
56         # Assign(Stackframe(Num('2')), Stack(Num('3'))))
57         LOADIN SP ACC 1;
58         STOREIN BAF ACC -2;
59         LOADIN SP ACC 2;
60         STOREIN BAF ACC -3;
61         LOADIN SP ACC 3;
62         STOREIN BAF ACC -4;
63         ADDI SP 3;
64         # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65         # Ref(Stackframe(Num('2'))))
66         SUBI SP 1;
67         MOVE BAF IN1;
68         SUBI IN1 4;
69         STOREIN SP IN1 1;
70         # Exp(Num('1'))
71         SUBI SP 1;
72         LOADI ACC 1;
73         STOREIN SP ACC 1;
74         # Exp(Num('1'))
75         SUBI SP 1;
76         LOADI ACC 1;

```

```

77     STOREIN SP ACC 1;
78     # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79     LOADIN SP ACC 2;
80     LOADIN SP IN2 1;
81     ADD ACC IN2;
82     STOREIN SP ACC 2;
83     ADDI SP 1;
84     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85     LOADIN SP IN1 2;
86     LOADIN SP IN2 1;
87     MULTI IN2 1;
88     ADD IN1 IN2;
89     ADDI SP 1;
90     STOREIN SP IN1 1;
91     # Exp(Stack(Num('1')))
92     LOADIN SP IN1 1;
93     LOADIN IN1 ACC 0;
94     STOREIN SP ACC 1;
95     # Return(Empty())
96     LOADIN BAF PC -1;
97 ]
98 ]

```

Code 1.29: RETI-Blocks Pass für Zugriff auf einen Arrayindex

### 1.3.4.3 Zuweisung an Arrayindex

Die **Zuweisung** eines Wertes an einen **Arrayindex** (z.B. `ar[2] = 42;`) wird im Folgenden anhand des Beispiels in Code 1.30 erläutert.

```

1 void main() {
2     int ar[2];
3     ar[2] = 42;
4 }

```

Code 1.30: PicoC-Code für Zuweisung an Arrayindex

Im **Abstract Syntax Tree** in Code 1.31 wird eine **Zuweisung** an einen **Arrayindex** `ar[2] = 42;` durch die Komposition `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

```

1 File
2   Name './example_array_assignment.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Exp(Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
10        Assign(Subscr(Name('ar'), Num('2')), Num('42'))
11      ]

```



12 ]

## Code 1.31: Abstract Syntax Tree für Zuweisung an Arrayindex

Im **PicoC-Mon Pass** in Code 1.32 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=`, bzw. des **Container-Knotens** der diesen darstellt ausgewertet: `Exp(Num('42'))`.

Danach ist das Vorgehen, bzw. sind die Kompositionen, die dieses darauffolgende Vorgehen darstellen: `Ref(Global(Num('0')))`, `Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum **Anfangsteil** und **Mittelteil** aus dem vorherigen Subkapitel 1.3.4.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird berechnet, wie in Subkapitel 1.3.4.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`<sup>6</sup> die Zuweisung `=` des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Arrayindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays `datatype ar[s1]...[sn]` abgespeichert ist, kann mittels der Formel 1.3.1:

$$\text{ref}(\text{ar}[k_1] \dots [k_n]) = \text{ref}(\text{ar}) + \left( \sum_{i=1}^n \left( \prod_{j=i+1}^n s_j \right) \cdot k_i \right) \cdot \text{size}(\text{datatype}) \quad (1.3.1)$$

aus der Betriebssysteme Vorlesung<sup>a</sup> berechnet werden<sup>b</sup>.

Die Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('2')))` repräsentiert dabei den Summanden `ref(ar)` in der Formel.

Die Komposition `Exp(Num('2'))` repräsentiert dabei einen **Subindex** (z.B. `i` in `a[i][j][k]`) beim **Zugriff auf ein Arrayelement**, der als Faktor `ki` in der Formel auftaucht.

Der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` repräsentiert dabei einen ausmultiplizierten Summanden  $\left( \prod_{j=i+1}^n s_j \right) \cdot k_i \cdot \text{size}(\text{datatype})$  in der Formel.

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** `M[ref(ar[k1]...[kn])]` der Speicherzelle an der finalen **Adresse** `ref(ar[k1]...[kn])`.

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

<sup>b</sup>`ref(exp)` steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte

```

1 File
2   Name './example_array_assignment.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Exp(Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
8         // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
9         Exp(Num('42'))

```

<sup>6</sup>Ist in Tabelle 1.7 genauer beschrieben ist

```

10     Ref(Global(Num('0')))
11     Exp(Num('2'))
12     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
13     Assign(Stack(Num('1')), Stack(Num('2')))
14     Return(Empty())
15 ]
16 ]

```

Code 1.32: PicoC-Mon Pass für Zuweisung an Arrayindex

Im **RETI-Blocks Pass** in Code 1.33 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_array_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
8         # // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
9         # Exp(Num('42'))
10        SUBI SP 1;
11        LOADI ACC 42;
12        STOREIN SP ACC 1;
13        # Ref(Global(Num('0')))
14        SUBI SP 1;
15        LOADI IN1 0;
16        ADD IN1 DS;
17        STOREIN SP IN1 1;
18        # Exp(Num('2'))
19        SUBI SP 1;
20        LOADI ACC 2;
21        STOREIN SP ACC 1;
22        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23        LOADIN SP IN1 2;
24        LOADIN SP IN2 1;
25        MULTI IN2 1;
26        ADD IN1 IN2;
27        ADDI SP 1;
28        STOREIN SP IN1 1;
29        # Assign(Stack(Num('1')), Stack(Num('2')))
30        LOADIN SP IN1 1;
31        LOADIN SP ACC 2;
32        ADDI SP 2;
33        STOREIN IN1 ACC 0;
34        # Return(Empty())
35        LOADIN BAF PC -1;
36      ]
37    ]

```

Code 1.33: RETI-Blocks Pass für Zuweisung an Arrayindex

### 1.3.5 Umsetzung von Structs

#### 1.3.5.1 Deklaration und Definition von Structtypen

Die **Deklaration** eines neuen **Structtyps** (z.B. `struct st {int len; int ar[2];};`) und die **Definition** einer Variable mit diesem **Structtyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 1.34 erläutert.

```

1 struct st {int len; int ar[2];};
2
3 void main() {
4     struct st st_var;
5 }
```

Code 1.34: PicoC-Code für die Deklaration eines Structtyps

Bevor irgendwas definiert werden kann, muss erstmal ein **Structtyp** deklariert werden. Im **Abstract Syntax Tree** in Code 1.36 wird die **Deklaration eines Structtyps** `struct st {int len; int ar[2];};` durch die Komposition `StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` dargestellt.

Die **Definition** einer Variable mit diesem **Structtyp** `struct st st_var;` wird durch die Komposition `Alloc(Writeable(), StructSpec(Name('st')), Name('st_var'))` dargestellt.

```

1 File
2   Name './example_struct_decl_def.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), IntType('int'), Name('len'))
8         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var'))))
16      ]
17  ]
```

Code 1.35: Abstract Syntax Tree für die Deklaration eines Structtyps

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 1.36 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt, der die Felder `type_qualifier`, `datatype`, `name`, `position` und `size` wie üblich belegt, allerdings in das `value_address`-Feld seine Attribute `[Name('len@st'), Name('ar@st')]` reinschreibt. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Structtyps** zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Structtypen** ihre Attribute gleich benennen können.

Jedes der **Attribute** [Name('len@st'), Name('ar@st')] erhält auch einen eigenen Eintrag in der **Symbolta-**  
**belle**, wobei die Felder `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt  
werden. Die Felder `type_qualifier`, `datatype` und `name` werden z.B. bei Name('ar@st') mithilfe der Attribute  
von `Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))]` belegt.

```

1 SymbolTable
2   [
3     Symbol
4     {
5         type_qualifier:      Empty()
6         datatype:            IntType('int')
7         name:                 Name('len@st')
8         value or address:     Empty()
9         position:             Pos(Num('1'), Num('15'))
10        size:                 Num('1')
11    },
12    Symbol
13    {
14        type_qualifier:      Empty()
15        datatype:            ArrayDecl([Num('2')], IntType('int'))
16        name:                 Name('ar@st')
17        value or address:     Empty()
18        position:             Pos(Num('1'), Num('24'))
19        size:                 Num('2')
20    },
21    Symbol
22    {
23        type_qualifier:      Empty()
24        datatype:            StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
25        ↪ Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
26        ↪ Name('ar'))])
27        name:                 Name('st')
28        value or address:     [Name('len@st'), Name('ar@st')]
29        position:             Pos(Num('1'), Num('7'))
30        size:                 Num('3')
31    },
32    Symbol
33    {
34        type_qualifier:      Empty()
35        datatype:            FunDecl(VoidType('void'), Name('main'), [])
36        name:                 Name('main')
37        value or address:     Empty()
38        position:             Pos(Num('3'), Num('5'))
39        size:                 Empty()
40    },
41    Symbol
42    {
43        type_qualifier:      Writeable()
44        datatype:            StructSpec(Name('st'))
45        name:                 Name('st_var@main')
46        value or address:     Num('0')
47        position:             Pos(Num('4'), Num('12'))
48        size:                 Num('3')
49    }
50  ]

```

Code 1.36: Symboltabelle für die Deklaration eines Structtyps

## 1.3.5.2 Initialisierung von Structs

```

1 struct st1 {int *pntr[1];};
2
3 struct st2 {struct st1 st;};
4
5 void main() {
6     int var = 42;
7     struct st1 st = {.st={.pntr={{&var}}}};
8 }

```

Code 1.37: PicoC-Code für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc
8           Writeable,
9           ArrayDecl
10            [
11              Num '1'
12            ],
13           PtrDecl
14             Num '1',
15             IntType 'int',
16             Name 'pntr'
17         ],
18     StructDecl
19       Name 'st2',
20       [
21         Alloc
22           Writeable,
23           StructSpec
24             Name 'st1',
25             Name 'st'
26         ],
27     FunDef
28       VoidType 'void',
29       Name 'main',
30       [],
31       [
32         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
33         Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')),
34           ↳ Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
35           ↳ Array([Array([Ref(Name('var'))]))]))]))))
36       ]
37   ]

```

Code 1.38: Abstract Syntax Tree für Initialisierung von Structs

```

1 SymbolTable
2 [
3   Symbol
4   {
5     type qualifier:      Empty()
6     datatype:            ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int')))
7     name:                Name('ptr@st1')
8     value or address:    Empty()
9     position:            Pos(Num('1'), Num('17'))
10    size:                 Num('1')
11  },
12  Symbol
13  {
14    type qualifier:      Empty()
15    datatype:            StructDecl(Name('st1'), [Alloc(Writeable(),
16    ↪ ArrayDecl([Num('1')], PtrDecl(Num('1'), IntType('int'))), Name('ptr'))])
17    name:                Name('st1')
18    value or address:    [Name('ptr@st1')]
19    position:            Pos(Num('1'), Num('7'))
20    size:                 Num('1')
21  },
22  Symbol
23  {
24    type qualifier:      Empty()
25    datatype:            StructSpec(Name('st1'))
26    name:                Name('st@st2')
27    value or address:    Empty()
28    position:            Pos(Num('3'), Num('23'))
29    size:                 Num('1')
30  },
31  Symbol
32  {
33    type qualifier:      Empty()
34    datatype:            StructDecl(Name('st2'), [Alloc(Writeable(),
35    ↪ StructSpec(Name('st1')), Name('st'))])
36    name:                Name('st2')
37    value or address:    [Name('st@st2')]
38    position:            Pos(Num('3'), Num('7'))
39    size:                 Num('1')
40  },
41  Symbol
42  {
43    type qualifier:      Empty()
44    datatype:            FunDecl(VoidType('void'), Name('main'), [])
45    name:                Name('main')
46    value or address:    Empty()
47    position:            Pos(Num('5'), Num('5'))
48    size:                 Empty()
49  },
50  Symbol
51  {
52    type qualifier:      Writeable()
53    datatype:            IntType('int')
54    name:                Name('var@main')

```

```

53     value or address:    Num('0')
54     position:           Pos(Num('6'), Num('6'))
55     size:               Num('1')
56   },
57   Symbol
58   {
59     type qualifier:      Writeable()
60     datatype:           StructSpec(Name('st1'))
61     name:               Name('st@main')
62     value or address:    Num('1')
63     position:           Pos(Num('7'), Num('13'))
64     size:               Num('1')
65   }
66 ]

```

Code 1.39: Symboltabelle für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('st'), Struct([Assign(Name('pntr'),
11        ↪   Array([Array([Ref(Name('var'))]))]))))
12        Ref(Global(Num('0')))
13        Assign(Global(Num('1')), Stack(Num('1')))
14        Return(Empty())
15      ]
16    ]

```

Code 1.40: PicoC-Mon Pass für Initialisierung von Structs

```

1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;

```

```

16      # // Assign(Name('st'), Struct([Assign(Name('st'), Struct([Assign(Name('pntr'),
17      ↪ Array([Array([Ref(Name('var'))]))]))]))))
18      # Ref(Global(Num('0')))
19      SUBI SP 1;
20      LOADI IN1 0;
21      ADD IN1 DS;
22      STOREIN SP IN1 1;
23      # Assign(Global(Num('1')), Stack(Num('1')))
24      LOADIN SP ACC 1;
25      STOREIN DS ACC 1;
26      ADDI SP 1;
27      # Return(Empty())
28      LOADIN BAF PC -1;
29  ]

```

Code 1.41: RETI-Blocks Pass für Initialisierung von Structs

### 1.3.5.3 Zugriff auf Structattribut

```

1 struct pos {int x; int y;};
2
3 void main() {
4     struct pos st = {.x=4, .y=2};
5     st.y;
6 }

```

Code 1.42: PicoC-Code für Zugriff auf Structattribut

```

1 File
2   Name './example_struct_attr_access.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'x',
11        Alloc
12          Writeable,
13          IntType 'int',
14          Name 'y'
15      ],
16    FunDef
17      VoidType 'void',
18      Name 'main',
19      [],
20      [
21        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
22        ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
23        Exp(Attr(Name('st'), Name('y')))

```



```

23   ]
24 ]

```

Code 1.43: Abstract Syntax Tree für Zugriff auf Structattribut

```

1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8         ↪ Num('2'))]))
9         Exp(Num('4'))
10        Exp(Num('2'))
11        Assign(Global(Num('0')), Stack(Num('2')))
12        // Exp(Attr(Name('st'), Name('y')))
13        Ref(Global(Num('0')))
14        Ref(Attr(Stack(Num('1')), Name('y')))
15        Exp(Stack(Num('1')))
16        Return(Empty())
17      ]
18    ]

```

Code 1.44: PicoC-Mon Pass für Zugriff auf Structattribut

```

1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8         ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # Assign(Global(Num('0')), Stack(Num('2')))
18        LOADIN SP ACC 1;
19        STOREIN DS ACC 1;
20        LOADIN SP ACC 2;
21        STOREIN DS ACC 0;
22        ADDI SP 2;
23        # // Exp(Attr(Name('st'), Name('y')))
24        # Ref(Global(Num('0')))
25        SUBI SP 1;

```

```

25     LOADI IN1 0;
26     ADD IN1 DS;
27     STOREIN SP IN1 1;
28     # Ref(Attr(Stack(Num('1')), Name('y')))
29     LOADIN SP IN1 1;
30     ADDI IN1 1;
31     STOREIN SP IN1 1;
32     # Exp(Stack(Num('1')))
33     LOADIN SP IN1 1;
34     LOADIN IN1 ACC 0;
35     STOREIN SP ACC 1;
36     # Return(Empty())
37     LOADIN BAF PC -1;
38 ]
39 ]

```

Code 1.45: RETI-Blocks Pass für Zugriff auf Structattribut

#### 1.3.5.4 Zuweisung an Structattribut

```

1 struct pos {int x; int y;};
2
3 void main() {
4     struct pos st = {.x=4, .y=2};
5     st.y = 42;
6 }

```

Code 1.46: PicoC-Code für Zuweisung an Structattribut

```

1 File
2   Name './example_struct_attr_assignment.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc
8           Writeable,
9           IntType 'int',
10          Name 'x',
11         Alloc
12           Writeable,
13           IntType 'int',
14           Name 'y'
15       ],
16     FunDef
17       VoidType 'void',
18       Name 'main',
19       [],
20       [
21         Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
22           ↪ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
23         Assign(Attr(Name('st'), Name('y')), Num('42'))

```

```

23   ]
24 ]

```

Code 1.47: Abstract Syntax Tree für Zuweisung an Structattribut

```

1 File
2   Name './example_struct_attr_assignment.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8         ↪ Num('2'))]))
9         Exp(Num('4'))
10        Exp(Num('2'))
11        Assign(Global(Num('0')), Stack(Num('2')))
12        // Assign(Attr(Name('st'), Name('y')), Num('42'))
13        Exp(Num('42'))
14        Ref(Global(Num('0')))
15        Ref(Attr(Stack(Num('1')), Name('y')))
16        Assign(Stack(Num('1')), Stack(Num('2')))
17        Return(Empty())
18      ]
19    ]

```

Code 1.48: PicoC-Mon Pass für Zuweisung an Structattribut

```

1 File
2   Name './example_struct_attr_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8         ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # Assign(Global(Num('0')), Stack(Num('2')))
18        LOADIN SP ACC 1;
19        STOREIN DS ACC 1;
20        LOADIN SP ACC 2;
21        STOREIN DS ACC 0;
22        ADDI SP 2;
23        # // Assign(Attr(Name('st'), Name('y')), Num('42'))
24        # Exp(Num('42'))

```

```

24     SUBI SP 1;
25     LOADI ACC 42;
26     STOREIN SP ACC 1;
27     # Ref(Global(Num('0')))
28     SUBI SP 1;
29     LOADI IN1 0;
30     ADD IN1 DS;
31     STOREIN SP IN1 1;
32     # Ref(Attr(Stack(Num('1')), Name('y')))
33     LOADIN SP IN1 1;
34     ADDI IN1 1;
35     STOREIN SP IN1 1;
36     # Assign(Stack(Num('1')), Stack(Num('2')))
37     LOADIN SP IN1 1;
38     LOADIN SP ACC 2;
39     ADDI SP 2;
40     STOREIN IN1 ACC 0;
41     # Return(Empty())
42     LOADIN BAF PC -1;
43 ]
44 ]

```

Code 1.49: RETI-Blocks Pass für Zuweisung an Structattribut

### 1.3.6 Umsetzung der Derived Datatypes im Zusammenspiel

#### 1.3.6.1 Anfangsteil für Globale Statische Daten und Stackframe

```

1 struct ar_with_len {int len; int ar[2];};
2
3 void main() {
4     struct ar_with_len st_ar[3];
5     int (*pntr2)[3];
6     pntr2;
7 }
8
9 void fun() {
10    struct ar_with_len st_ar[3];
11    int (*pntr1)[3];
12    pntr1;
13 }

```

Code 1.50: PicoC-Code für den Anfangsteil

```

1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     StructDecl
5       Name 'ar_with_len',
6       [
7         Alloc

```

```

8      Writeable,
9      IntType 'int',
10     Name 'len',
11   Alloc
12     Writeable,
13     ArrayDecl
14       [
15         Num '2'
16       ],
17     IntType 'int',
18     Name 'ar'
19   ],
20   FunDef
21     VoidType 'void',
22     Name 'main',
23     [],
24     [
25       Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
26         ↪ Name('st_ar')))
27       Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
28         ↪ IntType('int')))), Name('ptr2')))
29       Exp(Name('ptr2'))
30     ],
31   FunDef
32     VoidType 'void',
33     Name 'fun',
34     [],
35     [
36       Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
37         ↪ Name('st_ar')))
38       Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
39         ↪ Name('ptr1')))
40       Exp(Name('ptr1'))
41     ]
42 ]

```

Code 1.51: Abstract Syntax Tree für den Anfangsteil

```

1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
8         ↪ Name('st_ar')))
9         // Exp(Alloc(Writeable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
10        ↪ IntType('int')))), Name('ptr2')))
11        // Exp(Name('ptr2'))
12        Exp(Global(Num('9')))
13        Return(Empty())
14      ],
15    Block
16      Name 'fun.0',

```

```

15  [
16    // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
    ↪ Name('st_ar')))
17    // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
    ↪ Name('pntr1')))
18    // Exp(Name('pntr1'))
19    Exp(Stackframe(Num('9')))
20    Return(Empty())
21  ]
22 ]

```

Code 1.52: PicoC-Mon Pass für den Anfangsteil

```

1 File
2   Name './example_derived_dts_introduction_part.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
            ↪ Name('st_ar')))
8         # // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')],
            ↪ PtrDecl(Num('1'), IntType('int')))), Name('pntr2')))
9         # // Exp(Name('pntr2'))
10        # Exp(Global(Num('9')))
11        SUBI SP 1;
12        LOADIN DS ACC 9;
13        STOREIN SP ACC 1;
14        # Return(Empty())
15        LOADIN BAF PC -1;
16      ],
17    Block
18      Name 'fun.0',
19      [
20        # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
            ↪ Name('st_ar')))
21        # // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')],
            ↪ IntType('int'))), Name('pntr1')))
22        # // Exp(Name('pntr1'))
23        # Exp(Stackframe(Num('9')))
24        SUBI SP 1;
25        LOADIN BAF ACC -11;
26        STOREIN SP ACC 1;
27        # Return(Empty())
28        LOADIN BAF PC -1;
29      ]
30    ]

```

Code 1.53: RETI-Blocks Pass für den Anfangsteil

## 1.3.6.2 Mittelteil für die verschiedenen Derived Datatypes

```

1 struct st1 {int (*ar)[1];};
2
3 void main() {
4     int var[1] = {42};
5     struct st1 st_first = {.ar=&var};
6     (*st_first.ar)[0];
7 }

```

Code 1.54: PicoC-Code für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc
8           Writeable,
9           PntrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '1'
14              ],
15            IntType 'int',
16            Name 'ar'
17          ],
18      FunDef
19        VoidType 'void',
20        Name 'main',
21        [],
22        [
23          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
24            ↳ Array([Num('42')]))
25          Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st_first')),
26            ↳ Struct([Assign(Name('ar'), Ref(Name('var')))]))
27          Exp(Subscr(Deref(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
28        ]
29      ]
30    ]

```

Code 1.55: Abstract Syntax Tree für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Array([Num('42')]))

```

```

8      Exp(Num('42'))
9      Assign(Global(Num('0')), Stack(Num('1')))
10     // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11     Ref(Global(Num('0')))
12     Assign(Global(Num('1')), Stack(Num('1')))
13     // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
14     Ref(Global(Num('1')))
15     Ref(Attr(Stack(Num('1')), Name('ar')))
16     Exp(Num('0'))
17     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18     Exp(Num('0'))
19     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20     Exp(Stack(Num('1')))
21     Return(Empty())
22 ]
23 ]

```

Code 1.56: PicoC-Mon Pass für den Mittelteil

```

1 File
2   Name './example_derived_dts_main_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
27        # Ref(Global(Num('1')))
28        SUBI SP 1;
29        LOADI IN1 1;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Ref(Attr(Stack(Num('1')), Name('ar')))
33        LOADIN SP IN1 1;
34        ADDI IN1 0;
35        STOREIN SP IN1 1;

```



```

36      # Exp(Num('0'))
37      SUBI SP 1;
38      LOADI ACC 0;
39      STOREIN SP ACC 1;
40      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41      LOADIN SP IN2 2;
42      LOADIN IN2 IN1 0;
43      LOADIN SP IN2 1;
44      MULTI IN2 1;
45      ADD IN1 IN2;
46      ADDI SP 1;
47      STOREIN SP IN1 1;
48      # Exp(Num('0'))
49      SUBI SP 1;
50      LOADI ACC 0;
51      STOREIN SP ACC 1;
52      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
53      LOADIN SP IN1 2;
54      LOADIN SP IN2 1;
55      MULTI IN2 1;
56      ADD IN1 IN2;
57      ADDI SP 1;
58      STOREIN SP IN1 1;
59      # Exp(Stack(Num('1')))
60      LOADIN SP IN1 1;
61      LOADIN IN1 ACC 0;
62      STOREIN SP ACC 1;
63      # Return(Empty())
64      LOADIN BAF PC -1;
65  ]
66 ]

```

Code 1.57: RETI-Blocks Pass für den Mittelteil

### 1.3.6.3 Schlussteil für die verschiedenen Derived Datatypes

```

1 struct st {int attr[2];};
2
3 void main() {
4     int ar1[1][2] = {{42, 314}};
5     struct st ar2[1] = {.attr={42, 314}};
6     int var = 42;
7     int *pntr1 = &var;
8     int **pntr2 = &pntr1;
9
10    ar1[0];
11    ar2[0];
12    *pntr2;
13 }

```

Code 1.58: PicoC-Code für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           ArrayDecl
10            [
11              Num '2'
12            ],
13            IntType 'int',
14            Name 'attr'
15        ],
16      FunDef
17        VoidType 'void',
18        Name 'main',
19        [],
20        [
21          Assign(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
22            ↪ Name('ar1')), Array([Array([Num('42'), Num('314')])]))
23          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
24            ↪ Name('ar2')), Struct([Assign(Name('attr'), Array([Num('42'), Num('314')])]))))
25          Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
26          Assign(Alloc(Writeable(), PtrDecl(Num('1'), IntType('int')), Name('pntr1')),
27            ↪ Ref(Name('var')))
28          Assign(Alloc(Writeable(), PtrDecl(Num('2'), IntType('int')), Name('pntr2')),
29            ↪ Ref(Name('pntr1')))
30          Exp(Subscr(Name('ar1'), Num('0')))
31          Exp(Subscr(Name('ar2'), Num('0')))
32          Exp(Deref(Name('pntr2'), Num('0')))
33        ]
34      ]
35    ]

```

Code 1.59: Abstract Syntax Tree für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.O',
6       [
7         // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')])]))
8         Exp(Num('42'))
9         Exp(Num('314'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
12          ↪ Num('314')])]))))
13        Exp(Num('42'))
14        Exp(Num('314'))
15        Assign(Global(Num('2')), Stack(Num('2')))
16        // Assign(Name('var'), Num('42'))
17        Exp(Num('42'))

```

```

17     Assign(Global(Num('4')), Stack(Num('1')))
18     // Assign(Name('pntr1'), Ref(Name('var')))
19     Ref(Global(Num('4')))
20     Assign(Global(Num('5')), Stack(Num('1')))
21     // Assign(Name('pntr2'), Ref(Name('pntr1')))
22     Ref(Global(Num('5')))
23     Assign(Global(Num('6')), Stack(Num('1')))
24     // Exp(Subscr(Name('ar1'), Num('0')))
25     Ref(Global(Num('0')))
26     Exp(Num('0'))
27     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
28     Exp(Stack(Num('1')))
29     // Exp(Subscr(Name('ar2'), Num('0')))
30     Ref(Global(Num('2')))
31     Exp(Num('0'))
32     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
33     Exp(Stack(Num('1')))
34     // Exp(Subscr(Name('pntr2'), Num('0')))
35     Ref(Global(Num('6')))
36     Exp(Num('0'))
37     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
38     Exp(Stack(Num('1')))
39     Return(Empty())
40 ]
41 ]

```

Code 1.60: PicoC-Mon Pass für den Schlussteil

```

1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('ar1'), Array([Array([Num('42'), Num('314')]))))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Exp(Num('314'))
13        SUBI SP 1;
14        LOADI ACC 314;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Assign(Name('ar2'), Struct([Assign(Name('attr'), Array([Num('42'),
23        ↪ Num('314')]))]))
24        # Exp(Num('42'))
25        SUBI SP 1;
26        LOADI ACC 42;

```

```
26     STOREIN SP ACC 1;
27     # Exp(Num('314'))
28     SUBI SP 1;
29     LOADI ACC 314;
30     STOREIN SP ACC 1;
31     # Assign(Global(Num('2')), Stack(Num('2')))
32     LOADIN SP ACC 1;
33     STOREIN DS ACC 3;
34     LOADIN SP ACC 2;
35     STOREIN DS ACC 2;
36     ADDI SP 2;
37     # // Assign(Name('var'), Num('42'))
38     # Exp(Num('42'))
39     SUBI SP 1;
40     LOADI ACC 42;
41     STOREIN SP ACC 1;
42     # Assign(Global(Num('4')), Stack(Num('1')))
43     LOADIN SP ACC 1;
44     STOREIN DS ACC 4;
45     ADDI SP 1;
46     # // Assign(Name('pntr1'), Ref(Name('var')))
47     # Ref(Global(Num('4')))
48     SUBI SP 1;
49     LOADI IN1 4;
50     ADD IN1 DS;
51     STOREIN SP IN1 1;
52     # Assign(Global(Num('5')), Stack(Num('1')))
53     LOADIN SP ACC 1;
54     STOREIN DS ACC 5;
55     ADDI SP 1;
56     # // Assign(Name('pntr2'), Ref(Name('pntr1')))
57     # Ref(Global(Num('5')))
58     SUBI SP 1;
59     LOADI IN1 5;
60     ADD IN1 DS;
61     STOREIN SP IN1 1;
62     # Assign(Global(Num('6')), Stack(Num('1')))
63     LOADIN SP ACC 1;
64     STOREIN DS ACC 6;
65     ADDI SP 1;
66     # // Exp(Subscr(Name('ar1'), Num('0')))
67     # Ref(Global(Num('0')))
68     SUBI SP 1;
69     LOADI IN1 0;
70     ADD IN1 DS;
71     STOREIN SP IN1 1;
72     # Exp(Num('0'))
73     SUBI SP 1;
74     LOADI ACC 0;
75     STOREIN SP ACC 1;
76     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
77     LOADIN SP IN1 2;
78     LOADIN SP IN2 1;
79     MULTI IN2 2;
80     ADD IN1 IN2;
81     ADDI SP 1;
82     STOREIN SP IN1 1;
```

```

83      # Exp(Stack(Num('1')))
84      # // Exp(Subscr(Name('ar2'), Num('0')))
85      # Ref(Global(Num('2')))
86      SUBI SP 1;
87      LOADI IN1 2;
88      ADD IN1 DS;
89      STOREIN SP IN1 1;
90      # Exp(Num('0'))
91      SUBI SP 1;
92      LOADI ACC 0;
93      STOREIN SP ACC 1;
94      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
95      LOADIN SP IN1 2;
96      LOADIN SP IN2 1;
97      MULTI IN2 2;
98      ADD IN1 IN2;
99      ADDI SP 1;
100     STOREIN SP IN1 1;
101     # Exp(Stack(Num('1')))
102     LOADIN SP IN1 1;
103     LOADIN IN1 ACC 0;
104     STOREIN SP ACC 1;
105     # // Exp(Subscr(Name('pntr2'), Num('0')))
106     # Ref(Global(Num('6')))
107     SUBI SP 1;
108     LOADI IN1 6;
109     ADD IN1 DS;
110     STOREIN SP IN1 1;
111     # Exp(Num('0'))
112     SUBI SP 1;
113     LOADI ACC 0;
114     STOREIN SP ACC 1;
115     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
116     LOADIN SP IN2 2;
117     LOADIN IN2 IN1 0;
118     LOADIN SP IN2 1;
119     MULTI IN2 1;
120     ADD IN1 IN2;
121     ADDI SP 1;
122     STOREIN SP IN1 1;
123     # Exp(Stack(Num('1')))
124     # Return(Empty())
125     LOADIN BAF PC -1;
126 ]
127 ]

```

Code 1.61: RETI-Blocks Pass für den Schlussteil

## 1.3.7 Umsetzung von Funktionen

### 1.3.7.1 Funktionen auflösen zu RETI Code

```

1 void main() {
2     return;
3 }
4
5 void fun1() {
6 }
7
8 int fun2() {
9     return 1;
10 }

```

Code 1.62: PicoC-Code für 3 Funktionen

```

1 File
2   Name './example_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return(Empty())
10      ],
11     FunDef
12       VoidType 'void',
13       Name 'fun1',
14       [],
15       [],
16     FunDef
17       IntType 'int',
18       Name 'fun2',
19       [],
20       [
21         Return(Num('1'))
22      ]
23   ]

```

Code 1.63: Abstract Syntax Tree für 3 Funktionen

```

1 File
2   Name './example_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],

```

```

8      [
9      Block
10         Name 'main.2',
11         [
12             Return(Empty())
13         ]
14     ],
15     FunDef
16     VoidType 'void',
17     Name 'fun1',
18     [],
19     [
20         Block
21             Name 'fun1.1',
22             []
23     ],
24     FunDef
25     IntType 'int',
26     Name 'fun2',
27     [],
28     [
29         Block
30             Name 'fun2.0',
31             [
32                 Return(Num('1'))
33             ]
34     ]
35 ]

```

Code 1.64: RETI-Blocks Pass für 3 Funktionen

```

1 File
2 Name './example_3_funs.picoc_mon',
3 [
4     Block
5         Name 'main.2',
6         [
7             Return(Empty())
8         ],
9     Block
10        Name 'fun1.1',
11        [
12            Return(Empty())
13        ],
14    Block
15        Name 'fun2.0',
16        [
17            // Return(Num('1'))
18            Exp(Num('1'))
19            Return(Stack(Num('1')))
20        ]
21 ]

```

Code 1.65: PicoC-Mon Pass für 3 Funktionen

```

1 File
2   Name './example_3_funs.reti_blocks',
3   [
4     Block
5       Name 'main.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun1.1',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun2.0',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ]
29  ]

```

Code 1.66: RETI-Blocks Pass für 3 Funktionen

### 1.3.7.1.1 Sprung zur Main Funktion

```

1 void fun1() {
2 }
3
4 int fun2() {
5     return 1;
6 }
7
8 void main() {
9     return;
10 }

```

Code 1.67: PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```

1 File
2   Name './example_3_funs_main.picoc_mon',

```



```

3  [
4    Block
5      Name 'fun1.2',
6      [
7        Return(Empty())
8      ],
9    Block
10     Name 'fun2.1',
11     [
12       // Return(Num('1'))
13       Exp(Num('1'))
14       Return(Stack(Num('1')))
15     ],
16    Block
17     Name 'main.0',
18     [
19       Return(Empty())
20     ]
21 ]

```

Code 1.68: PicoC-Mon Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```

1 File
2   Name './example_3_funs_main.reti_blocks',
3   [
4     Block
5       Name 'fun1.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'fun2.1',
12      [
13        # // Return(Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Return(Stack(Num('1')))
19        LOADIN SP ACC 1;
20        ADDI SP 1;
21        LOADIN BAF PC -1;
22      ],
23    Block
24      Name 'main.0',
25      [
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29 ]

```

Code 1.69: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```

1 File
2   Name './example_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun2.1',
18      [
19        # // Return(Num('1'))
20        # Exp(Num('1'))
21        SUBI SP 1;
22        LOADI ACC 1;
23        STOREIN SP ACC 1;
24        # Return(Stack(Num('1')))
25        LOADIN SP ACC 1;
26        ADDI SP 1;
27        LOADIN BAF PC -1;
28      ],
29    Block
30      Name 'main.0',
31      [
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ]
35  ]

```

Code 1.70: PicoC-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

### 1.3.7.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

```

1 int fun2(int var);
2
3 void fun1() {
4 }
5
6 void main() {
7   int var = fun2(42);
8   return;
9 }
10
11 int fun2(int var) {
12   return var;
13 }

```

Code 1.71: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Bei mehreren Funktionen werden die **Scopes** der unterschiedlichen **Funktionen** mittels eines **Suffix** "<fun\_name>@" umgesetzt, der an den **Variablen**namen <var> drangehängt wird: <var>@<fun\_name>. Dieser **Suffix** wird geändert sobald beim **Top-Down**<sup>7</sup> Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** nach dem **Depth-First-Search** Schema über den

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(),
7         ↪ IntType('int'), Name('var'))])
8       name:                Name('fun2')
9       value or address:    Empty()
10      position:            Pos(Num('1'), Num('4'))
11      size:                Empty()
12    },
13    Symbol
14    {
15      type qualifier:      Empty()
16      datatype:            FunDecl(VoidType('void'), Name('fun1'), [])
17      name:                Name('fun1')
18      value or address:    Empty()
19      position:            Pos(Num('3'), Num('5'))
20      size:                Empty()
21    },
22    Symbol
23    {
24      type qualifier:      Empty()
25      datatype:            FunDecl(VoidType('void'), Name('main'), [])
26      name:                Name('main')
27      value or address:    Empty()
28      position:            Pos(Num('6'), Num('5'))
29      size:                Empty()
30    },
31    Symbol
32    {
33      type qualifier:      Writable()
34      datatype:            IntType('int')
35      name:                Name('var@main')
36      value or address:    Num('0')
37      position:            Pos(Num('7'), Num('6'))
38      size:                Num('1')
39    },
40    Symbol
41    {
42      type qualifier:      Writable()
43      datatype:            IntType('int')
44      name:                Name('var@fun2')
45      value or address:    Num('0')
46      position:            Pos(Num('11'), Num('13'))

```

<sup>7</sup>D.h. von der Wurzel zu den Blättern eines Baumes

```

46     size:          Num('1')
47   }
48 ]

```

Code 1.72: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

### 1.3.7.3 Funktionsaufruf

#### 1.3.7.3.1 Ohne Rückgabewert

```

1 struct st {int attr1; int attr2[2];};
2
3 void stack_fun(struct st param[2][3]);
4
5 void main() {
6     struct st local_var[2][3];
7     stack_fun(local_var);
8     return;
9 }
10
11 void stack_fun(struct st param[2][3]) {
12     int local_var;
13 }

```

Code 1.73: PicoC-Code für Funktionsaufruf ohne Rückgabewert

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
8         //   ↳ Name('local_var')))
9         // Exp(Call(Name('stack_fun'), [Name('local_var')]))
10        StackMalloc(Num('2'))
11        Ref(Global(Num('0')))
12        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
13        Exp(GoTo(Name('stack_fun.0')))
14        RemoveStackframe()
15        Return(Empty())
16      ],
17    Block
18      Name 'stack_fun.0',
19      [
20        // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
21        //   ↳ Name('param')))
22        // Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
23        Return(Empty())
24      ]
25    ]
26  ]

```

Code 1.74: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

```

1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
8         ↪   Name('local_var'))
9         # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
10        # StackMalloc(Num('2'))
11        SUBI SP 2;
12        # Ref(Global(Num('0')))
13        SUBI SP 1;
14        LOADI IN1 0;
15        ADD IN1 DS;
16        STOREIN SP IN1 1;
17        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
18        MOVE BAF ACC;
19        ADDI SP 3;
20        MOVE SP BAF;
21        SUBI SP 4;
22        STOREIN BAF ACC 0;
23        LOADI ACC GoTo(Name('addr@next_instr'));
24        ADD ACC CS;
25        STOREIN BAF ACC -1;
26        # Exp(GoTo(Name('stack_fun.0')))
27        Exp(GoTo(Name('stack_fun.0')))
28        # RemoveStackframe()
29        MOVE BAF IN1;
30        LOADIN IN1 BAF 0;
31        MOVE IN1 SP;
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ],
35    Block
36      Name 'stack_fun.0',
37      [
38        # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
39        ↪   Name('param'))
40        # // Exp(Alloc(Writable(), IntType('int'), Name('local_var')))
41        # Return(Empty())
42        LOADIN BAF PC -1;
43      ]
44    ]
45  ]

```

Code 1.75: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

```

1 # // Exp(GoTo(Name('main.1')))
2 # // patched out Exp(GoTo(Name('main.1')))

```

```

3 # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
  ↳ Name('local_var'))))
4 # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
5 # StackMalloc(Num('2'))
6 SUBI SP 2;
7 # Ref(Global(Num('0'))))
8 SUBI SP 1;
9 LOADI IN1 0;
10 ADD IN1 DS;
11 STOREIN SP IN1 1;
12 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
13 MOVE BAF ACC;
14 ADDI SP 3;
15 MOVE SP BAF;
16 SUBI SP 4;
17 STOREIN BAF ACC 0;
18 LOADI ACC 14;
19 ADD ACC CS;
20 STOREIN BAF ACC -1;
21 # Exp(GoTo(Name('stack_fun.0'))))
22 JUMP 5;
23 # RemoveStackframe()
24 MOVE BAF IN1;
25 LOADIN IN1 BAF 0;
26 MOVE IN1 SP;
27 # Return(Empty())
28 LOADIN BAF PC -1;
29 # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))), Name('param'))))
30 # // Exp(Alloc(Writable(), IntType('int'), Name('local_var'))))
31 # Return(Empty())
32 LOADIN BAF PC -1;

```

Code 1.76: RETI-Pass für Funktionsaufruf ohne Rückgabewert

### 1.3.7.3.2 Mit Rückgabewert

```

1 void stack_fun() {
2     return 42;
3 }
4
5 void main() {
6     int var = stack_fun();
7 }

```

Code 1.77: PicoC-Code für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',

```

```

6      [
7          // Return(Num('42'))
8          Exp(Num('42'))
9          Return(Stack(Num('1')))
10     ],
11     Block
12     Name 'main.0',
13     [
14         // Assign(Name('var'), Call(Name('stack_fun'), []))
15         StackMalloc(Num('2'))
16         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
17         Exp(GoTo(Name('stack_fun.1')))
18         RemoveStackframe()
19         Assign(Global(Num('0')), Stack(Num('1')))
20         Return(Empty())
21     ]
22 ]

```

Code 1.78: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4       Block
5       Name 'stack_fun.1',
6       [
7           # // Return(Num('42'))
8           # Exp(Num('42'))
9           SUBI SP 1;
10          LOADI ACC 42;
11          STOREIN SP ACC 1;
12          # Return(Stack(Num('1')))
13          LOADIN SP ACC 1;
14          ADDI SP 1;
15          LOADIN BAF PC -1;
16      ],
17      Block
18      Name 'main.0',
19      [
20          # // Assign(Name('var'), Call(Name('stack_fun'), []))
21          # StackMalloc(Num('2'))
22          SUBI SP 2;
23          # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24          MOVE BAF ACC;
25          ADDI SP 2;
26          MOVE SP BAF;
27          SUBI SP 2;
28          STOREIN BAF ACC 0;
29          LOADI ACC GoTo(Name('addr@next_instr'));
30          ADD ACC CS;
31          STOREIN BAF ACC -1;
32          # Exp(GoTo(Name('stack_fun.1')))
33          Exp(GoTo(Name('stack_fun.1')))
34          # RemoveStackframe()

```

```

35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Assign(Global(Num('0')), Stack(Num('1')))
39     LOADIN SP ACC 1;
40     STOREIN DS ACC 0;
41     ADDI SP 1;
42     # Return(Empty())
43     LOADIN BAF PC -1;
44 ]
45 ]

```

Code 1.79: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1 # // Exp(GoTo(Name('main.0'))))
2 JUMP 7;
3 # // Return(Num('42'))
4 # Exp(Num('42'))
5 SUBI SP 1;
6 LOADI ACC 42;
7 STOREIN SP ACC 1;
8 # Return(Stack(Num('1')))
9 LOADIN SP ACC 1;
10 ADDI SP 1;
11 LOADIN BAF PC -1;
12 # // Assign(Name('var'), Call(Name('stack_fun'), []))
13 # StackMalloc(Num('2'))
14 SUBI SP 2;
15 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
16 MOVE BAF ACC;
17 ADDI SP 2;
18 MOVE SP BAF;
19 SUBI SP 2;
20 STOREIN BAF ACC 0;
21 LOADI ACC 17;
22 ADD ACC CS;
23 STOREIN BAF ACC -1;
24 # Exp(GoTo(Name('stack_fun.1'))))
25 JUMP -15;
26 # RemoveStackframe()
27 MOVE BAF IN1;
28 LOADIN IN1 BAF 0;
29 MOVE IN1 SP;
30 # Assign(Global(Num('0')), Stack(Num('1')))
31 LOADIN SP ACC 1;
32 STOREIN DS ACC 0;
33 ADDI SP 1;
34 # Return(Empty())
35 LOADIN BAF PC -1;

```

Code 1.80: RETI-Pass für Funktionsaufruf mit Rückgabewert



**1.3.7.3.3 Umsetzung von Call by Sharing für Arrays**

```

1 void stack_fun(int (*param1)[3], int param2[2][3]) {
2 }
3
4 void main() {
5     int local_var1[2][3];
6     int local_var2[2][3];
7     stack_fun(local_var1, local_var2);
8 }

```

Code 1.81: PicoC-Code für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param1'))
9         // Exp(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
10        Return(Empty())
11      ],
12    Block
13      Name 'main.0',
14      [
15        // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
16        ↪ Name('local_var1'))
17        // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
18        ↪ Name('local_var2'))
19        // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
20        StackMalloc(Num('2'))
21        Ref(Global(Num('0')))
22        Ref(Global(Num('6')))
23        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
24        Exp(GoTo(Name('stack_fun.1')))
25        RemoveStackframe()
26        Return(Empty())
27      ]
28    ]
29  ]

```

Code 1.82: PicoC-Mon Pass für Call by Sharing für Arrays

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()

```

```

6      datatype:      FunDecl(VoidType('void'), Name('stack_fun'),
   ↪ [Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
   ↪ Name('param1')), Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')),
   ↪ Name('param2'))])
7      name:          Name('stack_fun')
8      value or address: Empty()
9      position:      Pos(Num('1'), Num('5'))
10     size:          Empty()
11 },
12 Symbol
13 {
14     type qualifier:  Writable()
15     datatype:      PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16     name:          Name('param1@stack_fun')
17     value or address: Num('0')
18     position:      Pos(Num('1'), Num('21'))
19     size:          Num('1')
20 },
21 Symbol
22 {
23     type qualifier:  Writable()
24     datatype:      PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
25     name:          Name('param2@stack_fun')
26     value or address: Num('1')
27     position:      Pos(Num('1'), Num('37'))
28     size:          Num('1')
29 },
30 Symbol
31 {
32     type qualifier:  Empty()
33     datatype:      FunDecl(VoidType('void'), Name('main'), [])
34     name:          Name('main')
35     value or address: Empty()
36     position:      Pos(Num('4'), Num('5'))
37     size:          Empty()
38 },
39 Symbol
40 {
41     type qualifier:  Writable()
42     datatype:      ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:          Name('local_var1@main')
44     value or address: Num('0')
45     position:      Pos(Num('5'), Num('6'))
46     size:          Num('6')
47 },
48 Symbol
49 {
50     type qualifier:  Writable()
51     datatype:      ArrayDecl([Num('2'), Num('3')], IntType('int'))
52     name:          Name('local_var2@main')
53     value or address: Num('6')
54     position:      Pos(Num('6'), Num('6'))
55     size:          Num('6')
56 }
57 ]

```

Code 1.83: Symboltabelle für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # // Exp(Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
8           ↪ IntType('int'))), Name('param1')))
9         # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
10        # Return(Empty())
11        LOADIN BAF PC -1;
12      ],
13    Block
14      Name 'main.0',
15      [
16        # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
17          ↪ Name('local_var1')))
18        # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
19          ↪ Name('local_var2')))
20        # // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
21        # StackMalloc(Num('2'))
22        SUBI SP 2;
23        # Ref(Global(Num('0')))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        STOREIN SP IN1 1;
28        # Ref(Global(Num('6')))
29        SUBI SP 1;
30        LOADI IN1 6;
31        ADD IN1 DS;
32        STOREIN SP IN1 1;
33        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
34        MOVE BAF ACC;
35        ADDI SP 4;
36        MOVE SP BAF;
37        SUBI SP 4;
38        STOREIN BAF ACC 0;
39        LOADI ACC GoTo(Name('addr@next_instr'));
40        ADD ACC CS;
41        STOREIN BAF ACC -1;
42        # Exp(GoTo(Name('stack_fun.1')))
43        Exp(GoTo(Name('stack_fun.1')))
44        # RemoveStackframe()
45        MOVE BAF IN1;
46        LOADIN IN1 BAF 0;
47        MOVE IN1 SP;
48        # Return(Empty())
49        LOADIN BAF PC -1;
50      ]
51    ]
52  ]

```

Code 1.84: RETI-Block Pass für Call by Sharing für Arrays

## 1.3.7.3.4 Umsetzung von Call by Value für Structs

```

1 struct st {int attr1; int attr2[2];};
2
3 void stack_fun(struct st param) {
4 }
5
6 void main() {
7     struct st local_var;
8     stack_fun(local_var);
9 }

```

Code 1.85: PicoC-Code für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('param')))
8         Return(Empty())
9       ],
10    Block
11      Name 'main.0',
12      [
13        // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('local_var')))
14        // Exp(Call(Name('stack_fun'), [Name('local_var')]))
15        StackMalloc(Num('2'))
16        Assign(Stack(Num('3')), Global(Num('0')))
17        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
18        Exp(GoTo(Name('stack_fun.1')))
19        RemoveStackframe()
20        Return(Empty())
21      ]
22  ]

```

Code 1.86: PicoC-Mon Pass für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('param')))
8         # Return(Empty())
9         LOADIN BAF PC -1;
10      ],
11    Block

```

```

12     Name 'main.0',
13     [
14         # // Exp(Alloc(Writable(), StructSpec(Name('st')), Name('local_var')))
15         # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
16         # StackMalloc(Num('2'))
17         SUBI SP 2;
18         # Assign(Stack(Num('3')), Global(Num('0')))
19         SUBI SP 3;
20         LOADIN DS ACC 0;
21         STOREIN SP ACC 1;
22         LOADIN DS ACC 1;
23         STOREIN SP ACC 2;
24         LOADIN DS ACC 2;
25         STOREIN SP ACC 3;
26         # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
27         MOVE BAF ACC;
28         ADDI SP 5;
29         MOVE SP BAF;
30         SUBI SP 5;
31         STOREIN BAF ACC 0;
32         LOADI ACC GoTo(Name('addr@next_instr'));
33         ADD ACC CS;
34         STOREIN BAF ACC -1;
35         # Exp(GoTo(Name('stack_fun.1')))
36         Exp(GoTo(Name('stack_fun.1')))
37         # RemoveStackframe()
38         MOVE BAF IN1;
39         LOADIN IN1 BAF 0;
40         MOVE IN1 SP;
41         # Return(Empty())
42         LOADIN BAF PC -1;
43     ]
44 ]

```

Code 1.87: RETI-Block Pass für Call by Value für Structs

### 1.3.8 Umsetzung kleinerer Details

## 1.4 Fehlermeldungen

### 1.4.1 Error Handler

### 1.4.2 Arten von Fehlermeldungen

#### 1.4.2.1 Syntaxfehler

#### 1.4.2.2 Laufzeitfehler

---

---

# 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).
- *GCC, the GNU Compiler Collection* - *GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).

## 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).