
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

BACHELORARBEIT

Abgabedatum: 28th 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

1	Implementierung	9
1.1	Lexikalische Analyse	9
1.1.1	Konkrete Syntax für die Lexikalische Analyse	9
1.1.2	Basic Lexer	10
1.2	Syntaktische Analyse	10
1.2.1	Konkrete Syntax für die Syntaktische Analyse	10
1.2.2	Umsetzung von Präzidenz	12
1.2.3	Derivation Tree Generierung	13
1.2.3.1	Early Parser	13
1.2.3.2	Codebeispiel	13
1.2.4	Derivation Tree Vereinfachung	14
1.2.4.1	Visitor	14
1.2.4.2	Codebeispiel	14
1.2.5	Abstrakt Syntax Tree Generierung	16
1.2.5.1	PicoC-Knoten	16
1.2.5.2	RETI-Knoten	21
1.2.5.3	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	22
1.2.5.4	Abstrakte Syntax	24
1.2.5.5	Transformer	26
1.2.5.6	Codebeispiel	26
1.3	Code Generierung	26
1.3.1	Übersicht	26
1.3.2	Passes	29
1.3.2.1	PicoC-Shrink Pass	29
1.3.2.1.1	Zweck	29
1.3.2.1.2	Codebeispiel	29
1.3.2.2	PicoC-Blocks Pass	30
1.3.2.2.1	Zweck	30
1.3.2.2.2	Abstrakte Syntax	30
1.3.2.2.3	Codebeispiel	30
1.3.2.3	PicoC-Mon Pass	32
1.3.2.3.1	Zweck	32
1.3.2.3.2	Abstrakte Syntax	32
1.3.2.3.3	Codebeispiel	32
1.3.2.4	RETI-Blocks Pass	34
1.3.2.4.1	Zweck	34
1.3.2.4.2	Abstrakte Syntax	34
1.3.2.4.3	Codebeispiel	34
1.3.2.5	RETI-Patch Pass	36
1.3.2.5.1	Zweck	36
1.3.2.5.2	Abstrakte Syntax	36
1.3.2.5.3	Codebeispiel	37
1.3.2.6	RETI Pass	39
1.3.2.6.1	Zweck	39
1.3.2.6.2	Konkrete und Abstrakte Syntax	39
1.3.2.6.3	Codebeispiel	40

1.3.3	Umsetzung von Funktionen	43
1.3.3.1	Mehrere Funktionen	43
1.3.3.1.1	Sprung zur Main Funktion	46
1.3.3.2	Funktionsdeklaration und -definition und Umsetzung von Scopes	48
1.3.3.3	Funktionsaufruf	51
1.3.3.3.1	Ohne Rückgabewert	51
1.3.3.3.2	Mit Rückgabewert	54
1.3.3.3.3	Umsetzung von Call by Sharing für Arrays	56
1.3.3.3.4	Umsetzung von Call by Value für Structs	59
1.4	Fehlermeldungen	61
1.4.1	Error Handler	61
1.4.2	Arten von Fehlermeldungen	61
1.4.2.1	Syntaxfehler	61
1.4.2.2	Laufzeitfehler	61

Abbildungsverzeichnis

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

Codeverzeichnis

1.1	PicoC Code für Derivation Tree Generierung	13
1.2	Derivation Tree nach Derivation Tree Generierung	14
1.3	Derivation Tree nach Derivation Tree Vereinfachung	15
1.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert	26
1.5	PicoC Code für Codebeispiel	29
1.6	Abstract Syntax Tree für Codebeispiel	29
1.7	PicoC Shrink Pass für Codebeispiel	30
1.8	PicoC-Blocks Pass für Codebeispiel	32
1.9	PicoC-Mon Pass für Codebeispiel	34
1.10	RETI-Blocks Pass für Codebeispiel	36
1.11	RETI-Patch Pass für Codebeispiel	39
1.12	RETI Pass für Codebeispiel	42
1.13	PicoC-Code für 3 Funktionen	43
1.14	Abstract Syntax Tree für 3 Funktionen	44
1.15	PicoC-Blocks Pass für 3 Funktionen	45
1.16	PicoC-Mon Pass für 3 Funktionen	45
1.17	RETI-Blocks Pass für 3 Funktionen	46
1.18	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	46
1.19	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	47
1.20	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	48
1.21	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	48
1.22	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	49
1.23	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	51
1.24	PicoC-Code für Funktionsaufruf ohne Rückgabewert	51
1.25	PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert	52
1.26	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	53
1.27	RETI-Pass für Funktionsaufruf ohne Rückgabewert	54
1.28	PicoC-Code für Funktionsaufruf mit Rückgabewert	54
1.29	PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert	54
1.30	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	55
1.31	RETI-Pass für Funktionsaufruf mit Rückgabewert	56
1.32	PicoC-Code für Call by Sharing für Arrays	56
1.33	PicoC-Mon Pass für Call by Sharing für Arrays	57
1.34	Symboltabelle für Call by Sharing für Arrays	58
1.35	RETI-Block Pass für Call by Sharing für Arrays	59
1.36	PicoC-Code für Call by Value für Structs	59
1.37	PicoC-Mon Pass für Call by Value für Structs	60
1.38	RETI-Block Pass für Call by Value für Structs	61

Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC	12
1.2	PicoC-Knoten Teil 1	16
1.3	PicoC-Knoten Teil 2	17
1.4	PicoC-Knoten Teil 3	18
1.5	PicoC-Knoten Teil 4	19
1.6	RETI-Knoten	21
1.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	23

Definitionsverzeichnis

1.1	Label	20
1.2	Location	20
1.3	Token-Knoten	20
1.4	Container-Knoten	20
1.5	Symboltabelle	32
1.6	Funktionsprototyp	49
1.7	Scope (bzw. Sichtbarkeitsbereich)	50

Grammatikverzeichnis

1.1.1 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 1	9
1.1.2 Konkrete Syntax für die Lexikalische Analyse in EBNF, Teil 2	10
1.2.1 Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1	11
1.2.2 Konkrete Syntax für die Syntaktische Analyse in EBNF, Teil 2	12
1.2.3 Abstrakte Syntax für L_{PioC}	25
1.3.1 Abstrakte Syntax für L_{PicoC_Blocks}	30
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}	36
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_opun_exp</i> <i>post_exp</i>	<i>L_Struct</i> + <i>L_Fun</i>
<i>input_exp</i>	::=	"input" "(" "	<i>L_Arith</i>
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i> <i>un_exp</i>	
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i> <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i> <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i> <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i> <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i> <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_oprel_exp</i> <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i> <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> " " <i>logic_and</i> <i>logic_and</i>	
<i>type_spec</i>	::=	<i>prim_dt</i> <i>struct_spec</i>	<i>L_Assign_Alloc</i>
<i>alloc</i>	::=	<i>type_spec</i> <i>pntr_decl</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	
<i>initializer</i>	::=	<i>logic_or</i> <i>array_init</i> <i>struct_init</i>	
<i>init_stmt</i>	::=	<i>alloc</i> "=" <i>initializer</i> ";"	
<i>const_init_stmt</i>	::=	"const" <i>type_spec</i> <i>name</i> "=" <i>NUM</i> ";"	
<i>pntr_deg</i>	::=	"*" *	<i>L_Pntr</i>
<i>pntr_decl</i>	::=	<i>pntr_deg</i> <i>array_decl</i> <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]") *	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name</i> <i>array_dims</i> "(" <i>pntr_decl</i> ")" <i>array_dims</i>	
<i>array_init</i>	::=	"{" <i>initializer</i> ("," <i>initializer</i>) * "}"	
<i>array_subscr</i>	::=	<i>post_exp</i> "[" <i>logic_or</i> "]"	
<i>struct_spec</i>	::=	"struct" <i>name</i>	<i>L_Struct</i>
<i>struct_params</i>	::=	(<i>alloc</i> ";") +	
<i>struct_decl</i>	::=	"struct" <i>name</i> "{" <i>struct_params</i> "}"	
<i>struct_init</i>	::=	"{" "." <i>name</i> "=" <i>initializer</i> ("," "." <i>name</i> "=" <i>initializer</i>) * "}"	
<i>struct_attr</i>	::=	<i>post_exp</i> "." <i>name</i>	
<i>if_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_If_Else</i>
<i>if_else_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i> "else" <i>exec_part</i>	
<i>while_stmt</i>	::=	"while" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_Loop</i>
<i>do_while_stmt</i>	::=	"do" <i>exec_part</i> "while" "(" <i>logic_or</i> ")" ";"	

Grammar 1.2.1: Konkrete Syntax Syntaktische Analyse in EBNF, Teil 1

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

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

1.2.2 Umsetzung von Präzidenz

Die **PicoC** Programmiersprache hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache **C**¹. 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 &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<b a<=b a>b a>=b</i>	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	<i>a==b a!=b</i>	Gleichheit und Ungleichheit	
7	<i>a&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&&b</i>	Logisches UND	
11	<i>a b</i>	Logisches ODER	
12	<i>a=b</i>	Zuweisung	Rechts, dann links ←
13	<i>a,b</i>	Komma	Links, dann rechts →

Tabelle 1.1: Präzidenzregeln von PicoC

¹*C Operator Precedence - cppreference.com.*

1.2.3 Derivation Tree Generierung

1.2.3.1 Early Parser

1.2.3.2 Codebeispiel

```

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

```

Code 1.1: PicoC Code für Derivation Tree Generierung

```

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

```

```

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

```

Code 1.2: Derivation Tree nach Derivation Tree Generierung

1.2.4 Derivation Tree Vereinfachung

1.2.4.1 Visitor

1.2.4.2 Codebeispiel

Beispiel aus Subkapitel 1.2.3.2 wird fortgeführt.

```

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

```

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

Code 1.3: Derivation Tree nach Derivation Tree Vereinfachung

1.2.5 Abstrakt Syntax Tree Generierung

1.2.5.1 PicoC-Knoten

PiocC-Knoten	Beschreibung
Name(val)	Ein Bezeichner , 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 Zahl , z.B. 42, -3 usw.
Char(val)	Ein Zeichen der ASCII-Zeichenkodierung , z.B. <code>'c'</code> , <code>'*'</code> usw.
Minus(), Not(), DerefOp(), RefOp(), LogicNot()	Die unären Operatoren <code>un_op</code> : <code>-a</code> , <code>~a</code> , <code>*a</code> , <code>&a</code> !a.
Add(), Sub(), Mul(), Div(), Mod(), Oplus(), And(), Or(), LogicAnd(), LogicOr()	Die binären Operatoren <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 & b</code> , <code>a b</code> , <code>a && b</code> , <code>a b</code> .
Eq(), NEq(), Lt(), LtE(), Gt(), GtE()	Die Relationen <code>rel</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> .
Const(), Writeable()	Die Type Qualifier <code>type_qual</code> : <code>const</code> , was für ein nicht beschreibbare Konstante steht und das nicht Angeben von <code>const</code> , was für einen beschreibbare Variable steht.
IntType(), CharType(), VoidType()	Die Type Specifier für Primitiven Datentypen , die in der Abstrakten Syntax, um eine intuitive Bezeichnung zu haben einfach nur unter Datentypen <code>datatype</code> eingeordnet werden: <code>int</code> , <code>char</code> , <code>void</code> .
Placeholder()	Platzhalter für einen Knoten, der diesen später ersetzt .
BinOp(exp, bin_op, exp)	Container für eine binäre Operation mit 2 Expressions: <code><exp1> <bin_op> <exp2></code>
UnOp(un_op, exp)	Container für eine unäre Operation mit einer Expression: <code><un_op> <exp></code> .
Exit(num)	Container für einen Exit Code , der vor der Beendigung in das ACC Register geschrieben wird und steht für die Beendigung des laufenden Programmes.
Atom(exp, rel, exp)	Container für eine binäre Relation mit 2 Expressions: <code><exp1> <rel> <exp2></code>
ToBool(exp)	Container für einen Arithmetischen Ausdruck , wie z.B. <code>1 + 3</code> oder einfach nur <code>3</code> , der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis <code>x > 1</code> auf 1 abgebildet wird.
Alloc(type_qual, datatype, name, local_var_or_param)	Container für eine Allokation <code><type_qual> <datatype> <name></code> mit den notwendigen Knoten <code>type_qual</code> , <code>datatype</code> und <code>name</code> , die alle für einen Eintrag in der Symboltabelle notwendigen Informationen enthalten. Zudem besitzt er ein verstecktes Attribut <code>local_var_or_param</code> , dass die Information trägt, ob es sich bei der Variable um eine Lokale Variable oder einen Parameter handelt.
Assign(lhs, exp)	Container für eine Zuweisung , 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 Logischer Ausdruck sein kann: <code>lhs = exp</code> .

Tabelle 1.2: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen beliebigen Ausdruck , dessen Ergebnis auf den Stack soll. Zudem besitzt er 2 versteckte Attribute, wobei datatype im RETI Blocks Pass wichtig ist und error_data für Fehlermeldungen wichtig ist.
Stack(num)	Container, der für das temporäre Ergebnis einer Berechnung, das num Speicherzellen relativ zum Stackpointer Register SP steht.
Stackframe(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Begin-Aktive-Funktion Register BAF steht.
Global(num)	Container, der für eine Variable steht, die num Speicherzellen relativ zum Datensegment Register DS steht.
StackMalloc(num)	Container, der für das Allokieren von num Speicherzellen auf dem Stack steht.
PntrDecl(num, datatype)	Container, der für den Pointerdatatype steht: <code><prim_dt> *<var></code> , wobei das Attribut num die Anzahl zusammengefasster Pointer angibt und datatype der Datentyp ist, auf den der oder die Pointer zeigen.
Ref(exp, datatype, error_data)	Container, der für die Anwendung des Referenz-Operators <code>&<var></code> steht und die Adresse einer Location (Definition 1.2) auf den Stack schreiben soll, die über exp eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei datatype im RETI Blocks Pass wichtig ist und error_data für Fehlermeldungen wichtig ist.
Deref(lhs, exp)	Container für den Indezzugriff auf einen Array- oder Pointerdatatype : <code><var>[<i>]</code> , wobei exp1 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 exp2 der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den Arraydatatype steht: <code><prim_dt> <var>[<i>]</code> , wobei das Attribut nums eine Liste von <code>Num('x')</code> ist, die die Dimensionen des Arrays angibt und datatype der Datentyp ist, der über das Anwenden von <code>Subscript()</code> auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den Initializer eines Arrays , dessen Einträge exps weitere Initializer für eine Array-Dimension oder ein Initializer für ein Struct oder ein Logischer Ausdruck sein können, z.B. <code>{{1, 2}, {3, 4}}</code> . Des Weiteren besitzt er ein verstecktes Attribut datatype , welches für den PicoC-Mon Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
Subscr(exp1, exp2)	Container für den Indezzugriff auf einen Array- oder Pointerdatatype : <code><var>[<i>]</code> , wobei exp1 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 exp2 der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten Structdatatype : <code>struct <name></code> , wobei das Attribut name festlegt, welchen selbst definierte Structdatatype dieser Container-Knoten repräsentiert.
Attr(exp, name)	Container für den Attributzugriff auf einen Structdatatype : <code><var>.<attr></code> , wobei exp1 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 name das Attribut ist, auf das zugegriffen werden soll.

Tabelle 1.3: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initializer eines Structs , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines Attributezeichners , zu einem weiteren Initializer für eine Array-Dimension oder zu einem Initializer für ein Struct oder zu einem Logischen Ausdruck . Des Weiteren besitzt er ein verstecktes Attribut datatype, welches für den PicoC-Mon Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Structdatentyps , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der Bezeichner des Structdatentyps ist und allocs eine Liste von Bezeichnern der Attribute des Structdatentyps mit dazugehörigem Datentyp , wofür sich der Container-Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.
If(exp, stmts)	Container für ein If Statement if(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts1, stmts2)	Container für ein If-Else Statement if(<exp>) { <stmts2> } else { <stmts2> } inklusive Condition exp und 2 Branches stmts1 und stmts2, die zwei Alternativen darstellen in denen jeweils Listen von Statements oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts)	Container für ein While-Statement while(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts)	Container für ein Do-While-Statement do { <stmts> } while(<exp>); inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen Funktionsaufruf : fun_name(exps), wobei name der Bezeichner der Funktion ist, die aufgerufen werden soll und exps eine Liste von Argumenten ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein Return-Statement : return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis vom Return-Statement zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine Funktionsdeklaration , z.B. <datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist und allocs die Parameter der Funktion sind, wobei der Container-Knoten 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 Funktionsdefinition , z.B. <datatype> <fun_name>(<datatype> <param>) {<stmts>}, wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist, allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts_blocks eine Liste von Statements bzw. Blöcken ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die Erstellung eines neuen Stackframes , wobei fun_name der Bezeichner der Funktion ist, für die ein neuer Stackframe erstellt werden soll und später dazu dient den Block 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 Adresse der Instruction, die direkt auf die Jump Instruction folgt, ersetzt wird.
RemoveStackframe()	Container für das Entfernen des aktuellen Stackframes .
File(name, decls_defs_blocks)	Container für alle Funktionen oder Blöcke , welche eine Datei als Ursprung haben, wobei name der Dateiname der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von Funktionen bzw. Blöcken ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für Statements , der auch als Block bezeichnet wird, wobei das Attribut name der Bezeichners des Labels (Definition 1.1) des Blocks ist und stmts_instrs eine Liste von Statements oder Instructions . Zudem besitzt er noch 3 versteckte Attribute, wobei instrs_before die Zahl der Instructions vor diesem Block zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die Parameter der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die lokalen Variablen der Funktion belegt werden müssen.
GoTo(name)	Container für ein Goto zu einem anderen Block , wobei das Attribut name der Bezeichner des Labels des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen Kommentar , den der Compiler selber während des Kompilervorangs erstellt, der im RETI-Interpreter selbst später nicht sichtbar sein wird, aber in den Immediate-Dateien , welche die Abstract Syntax Trees nach den verschiedenen Passes enthalten.
RETIComent(value)	Container für einen Kommentar im Code der Form: // # comment, der im RETI-Interpreter später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer RETI-CPU nicht umsetzbar ist und auch nicht sinnvoll wäre umzusetzen. Der Kommentar ist im Attribut value , welches jeder Knoten besitzt gespeichert.

Tabelle 1.5: PicoC-Knoten Teil 4

Definition 1.1: Label

Durch einen *Bezeichner eindeutig* zuordenbares *Sprungziel* im Programmcode.^a

^a`tab:picoc`knoten`teil`4.`

Definition 1.2: Location

Kollektiver Begriff für *Variablen*, *Attribute* bzw. *Elemente* von Variablen bestimmter Datentypen, *Speicherbereiche auf dem Stack*, die *temporäre Zwischenergebnisse* speichern und *Register*.

Im Grunde genommen alles, was mit einem *Programm zu tun* hat und irgendwo *gespeichert* ist.^a

^aG. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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 1.3) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.4) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

Definition 1.3: Token-Knoten

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

Token-Knoten entsprechen im Abstract Syntax Tree *Blättern*.^a

^aThiemann, „Compilerbau“.

Definition 1.4: 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.^a

Container-Knoten entsprechen im Abstract Syntax Tree *Inneren Knoten*.^b

^aWie 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.

^bThiemann, „Compilerbau“.

1.2.5.2 RETI-Knoten

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

^a 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 Adresse der Speicherzelle, die Num('addr') Speicherzellen relativ zum Datensegment Register DS steht auf den Stack .
Ref(Stackframe(Num('addr')))	Speichert Adresse der Speicherzelle, die Num('addr') Speicherzellen relativ zum Begin-Aktive-Funktion Register BAF steht auf den Stack .
Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))))	Berechnet die nächste Adresse aus der Adresse , die an Speicherzelle Stack(Num('addr1')) steht und dem Subscript Index , der an Speicherzelle Stack(Num('addr2')) steht und speichert diese auf den Stack . Die Berechnung ist abhängig davon ob der Datentyp ArrayDecl(datatype) oder PntrDecl(datatype) ist. Der Datentyp ist ein verstecktes Attribut von Ref(exp).
Ref(Attr(Stack(Num('addr1')), Name('attr')))	Berechnet die nächste Adresse aus der Adresse , die an Speicherzelle Stack(Num('addr1')) steht und dem Attributnamen Name('attr') und speichert diese auf den Stack . Zur Berechnung ist der Name des Struct in StructSpec(Name('st')) notwendig, dessen Attribut 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 Datensegment Register DS stehen, versetzt genauso auf den Stack .
Assign(Stack(Num('size')), Stackframe(Num('addr')))	Schreibt Num('size') viele Speicherzellen, die ab Stackframe(Num('addr')) relativ zum Begin-Aktive-Funktion Register BAF stehen, versetzt genauso auf den Stack .
Exp(Global(Num('addr')))	Speichert Inhalt der Speicherzelle, die Num('addr') Speicherzellen relativ zum Datensegment Register DS steht auf den Stack .
Exp(Stackframe(Num('addr')))	Speichert Inhalt der Speicherzelle, die Num('addr') Speicherzellen relativ zum Begin-Aktive-Funktion Register BAF steht auf den Stack .
Exp(Stack(Num('addr')))	Speichert Inhalt der Speicherzelle, die Num('addr') Speicherzellen relativ zum Stackpointer Register SP steht auf den Stack .
Assign(Stack(Num('addr1')), Stack(Num('addr2')))	Speichert Inhalt der Speicherzelle Stack(Num('addr2')), die Num('addr2') Speicherzellen relativ zum Stackpointer Register SP steht an der Adresse in der Speicherzelle, die Num('addr1') Speicherzellen relativ zum Stackpointer Register SP steht.
Assign(Global(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem Stack stehen, versetzt genauso auf die Speicherzellen ab Num('addr') relativ zum Datensegment Register DS .
Assign(Stackframe(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem Stack stehen, versetzt genauso auf die Speicherzellen ab Num('addr') relativ zum Begin-Aktive-Funktion Register BAF .
Exp(Reg(reg))	Schreibt den aktuellen Wert des Registers reg auf den Stack .
Instr(Loadi(), [Reg(Acc()), GoTo(Name('addr@next_instr'))])	Lädt in das Register ACC die Adresse 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 **Stack(Num('x'))**, $x \in \mathbb{N}$ ersetzt wurde.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen einfach nur eine **Expression** an ein **Exp(exp)** bzw. **Ref(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><exp></i> , <i><bin_op></i> , <i><exp></i>) <i>UnOp</i> (<i><un_op></i> , <i><exp></i>) <i>Call</i> (<i>Name('input')</i> , <i>None</i>)	
<i>exp_stmts</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>) <i>Call</i> (<i>Name('print')</i> , <i><exp></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><exp></i> , <i><rel></i> , <i><exp></i>) <i>ToBool</i> (<i><exp></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><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>) <i><ref_loc></i>	
<i>exp_stmts</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>)	
<i>stmt</i>	::=	<i>Assign</i> (<i><lhs></i> , <i><exp></i>) <i>Exp</i> (<i><exp_stmts></i>)	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num(str)</i> , <i><datatype></i>)	<i>L_Pntr</i>
<i>deref_loc</i>	::=	<i>Ref</i> (<i><ref_loc></i>) <i><ref_loc></i>	
<i>ref_loc</i>	::=	<i>Name(str)</i> <i>Deref</i> (<i><deref_loc></i> , <i><exp></i>) <i>Subscr</i> (<i><deref_loc></i> , <i><exp></i>) <i>Attr</i> (<i><ref_loc></i> , <i>Name(str)</i>)	
<i>exp</i>	::=	<i>Deref</i> (<i><deref_loc></i> , <i><exp></i>) <i>Ref</i> (<i><ref_loc></i>)	
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num(str)</i> +, <i><datatype></i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i><deref_loc></i> , <i><exp></i>) <i>Array</i> (<i><exp></i> +)	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name(str)</i>)	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i><ref_loc></i> , <i>Name(str)</i>) <i>Struct</i> (<i>Assign</i> (<i>Name(str)</i> , <i><exp></i>) +)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>) +)	
<i>stmt</i>	::=	<i>If</i> (<i><exp></i> , <i><stmt></i> *) <i>IfElse</i> (<i><exp></i> , <i><stmt></i> *, <i><stmt></i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> (<i><exp></i> , <i><stmt></i> *) <i>DoWhile</i> (<i><exp></i> , <i><stmt></i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> (<i>Name(str)</i> , <i><exp></i> *)	<i>L_Fun</i>
<i>exp_stmts</i>	::=	<i>Call</i> (<i>Name(str)</i> , <i><exp></i> *)	
<i>stmt</i>	::=	<i>Return</i> (<i><exp></i>)	
<i>decl_def</i>	::=	<i>FunDecl</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*) <i>FunDef</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*, <i><stmt></i> *)	
<i>file</i>	::=	<i>File</i> (<i>Name(str)</i> , <i><decl_def></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(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')],
8           ↪ PtrDecl(Num('1'), IntType('int')))), Name('attr'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
16          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var'))
17      ]
18  ]

```

Code 1.4: Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert

1.3 Code Generierung

1.3.1 Übersicht

Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse**, wird in diesem Kapitel aus den verschiedenen **Kompositionen** von **Container-Knoten** und **Token-Knoten** im Abstract Syntax Tree das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definition ??). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel ??) in Abbildung 1.1 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer **X_{86_64}**-Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer **X_{86_64}**-Architektur laufen könnte. Und selbst dieses **T-Diagramm** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu **X_{86_64}** kompiliert wurde usw.

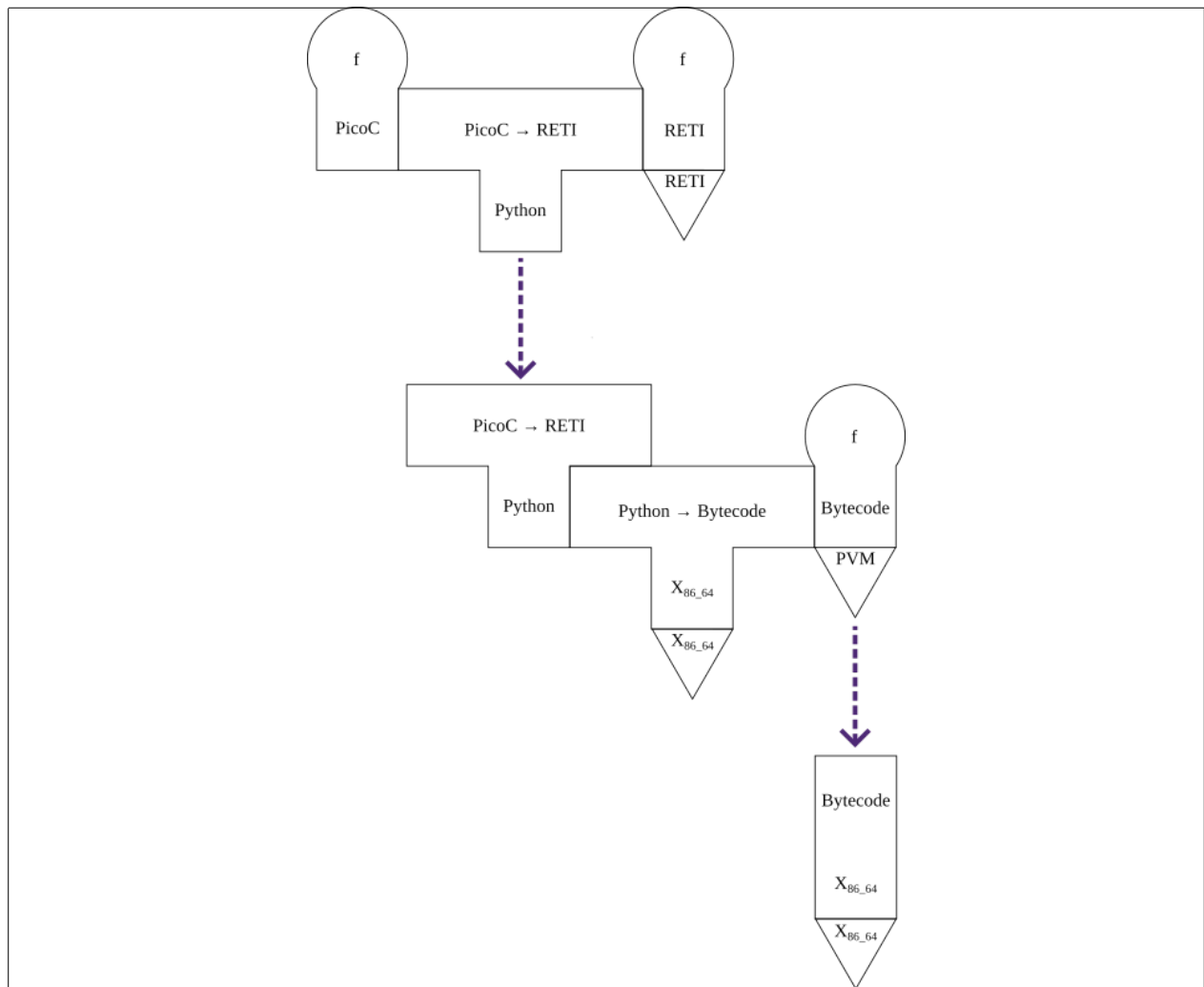


Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

Dieses längliche **T-Diagramm** in Abbildung 1.1 lässt sich zusammenfassen, sodass man das **T-Diagramm** in Abbildung 1.2 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in X_{86_64} -Maschinensprache geschrieben ist.

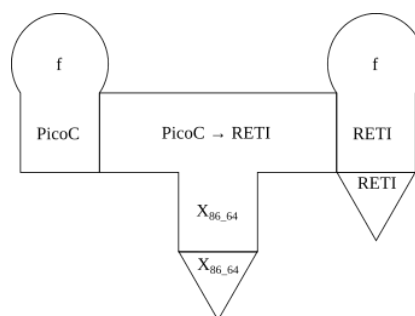


Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

Nachdem der Kompilierprozess des **PicoC-Compiler** im **vertikalen** nun genauer angesehen wurde, wird

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

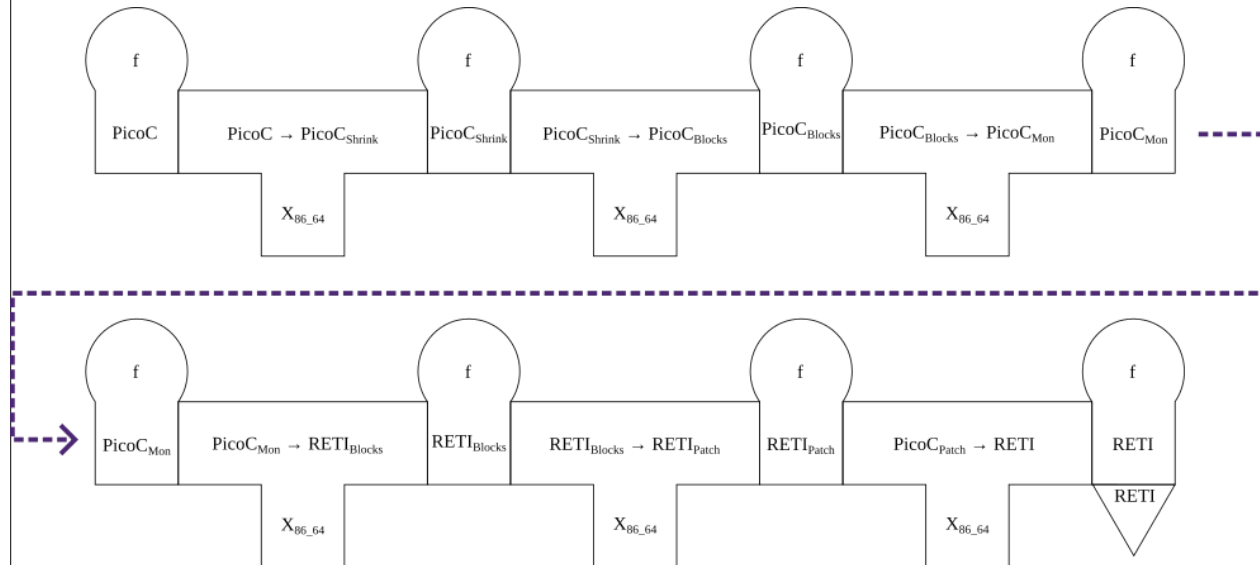


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 1.3.2 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. Die von den **Passes** generierten **Abstract Syntax Trees** werden dabei mit jedem **Pass** der **Syntax** des **RETI-Code's** immer ähnlicher werden. Jeder Pass sollte dabei möglichst eine Aufgabe übernehmen, da der Sinn von **Passes** ist, die Kompilierung in mehrere kleinschrittige Aufgaben runterzuberechnen. Wie es auch schon der Zweck des **Derivation Tree** in der Syntaktischen Analyse war, eine Zwischenstufe zum **Abstract Syntax Tree** darzustellen, aus der sich unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstract Syntax Tree** generieren lies.

In den darauffolgenden Unterkapiteln ??, ??, ?? und 1.3.3 werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 1.3.2 nicht ausreichend vertieft wurden. Viele der verwendeten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird². Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem **Link**³ finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind⁴.

²Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

³https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples

⁴Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstract Syntax Trees** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

1.3.2 Passes

1.3.2.1 PicoC-Shrink Pass

1.3.2.1.1 Zweck

1.3.2.1.2 Codebeispiel

```
1 // based on a example program from Christoph Scholl's 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 Zweck

Der Zweck dieses **Passes** ist die die Container-Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von Blöcken, `GoTo(label)`-Statements und nur noch IF-Else-Container-Knoten für die **Condition** umzusetzen.

1.3.2.2.2 Abstrakte Syntax

<i>decl_def</i>	$::=$	<i>FunDef</i> ($\langle datatype \rangle$, <i>Name</i> (<i>str</i>), <i>Alloc</i> (<i>Writable</i> ()), $\langle datatype \rangle$, <i>Name</i> (<i>str</i>))* , $\langle block \rangle$ *)	<i>L_Fun</i>
<i>block</i>	$::=$	<i>Block</i> (<i>Name</i> (<i>str</i>), $\langle stmt \rangle$ *)	<i>L_Blocks</i>
<i>stmt</i>	$::=$	<i>GoTo</i> (<i>Name</i> (<i>str</i>)) <i>NewStackframe</i> (<i>Name</i> ()), <i>GoTo</i> (<i>str</i>) <i>RemoveStackframe</i> () <i>SetScope</i> (<i>Name</i> (<i>str</i>) <i>SingleLineComment</i> (<i>str</i> , <i>str</i>)	

Grammar 1.3.1: Abstrakte Syntax für L_{PicoC_Blocks}

1.3.2.2.3 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(Writable(), IntType('int'), Name('n')), Num('4'))
13            Assign(Alloc(Writable(), 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 Zweck

1.3.2.3.2 Abstrakte Syntax

<i>ref_loc</i>	$::=$	<i>Stack</i> (<i>Num</i> (<i>str</i>)) <i>Global</i> (<i>Num</i> (<i>str</i>))	<i>L_Assign_Alloc</i>
		<i>Stackframe</i> (<i>Num</i> (<i>str</i>))	
<i>error_data</i>	$::=$	$\langle exp \rangle$ <i>Pos</i> (<i>Num</i> (<i>str</i>), <i>Num</i> (<i>str</i>))	
<i>exp</i>	$::=$	<i>Stack</i> (<i>Num</i> (<i>str</i>)) <i>Ref</i> ($\langle ref_loc \rangle$, $\langle datatype \rangle$, $\langle error_data \rangle$)	
<i>stmt</i>	$::=$	<i>Exp</i> ($\langle exp \rangle$)	
		<i>Assign</i> (<i>Alloc</i> (<i>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 1.5: Symboltabelle

1.3.2.3.3 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 Zweck

1.3.2.4.2 Abstrakte Syntax

<i>program</i>	$::=$	$Program(Name(str), \langle block \rangle^*)$	$L_Program$
<i>exp_stmts</i>	$::=$	$GoTo(str)$	L_Blocks
<i>instrs_before</i>	$::=$	$Num(str)$	
<i>num_instrs</i>	$::=$	$Num(str)$	
<i>block</i>	$::=$	$Block(Name(str), \langle instr \rangle^*, \langle instrs_before \rangle, \langle num_instrs \rangle)$	
<i>instr</i>	$::=$	$GoTo(Name(str))$	

Grammar 1.3.3: Abstrakte Syntax für L_{RETI_Blocks}

1.3.2.4.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4       Block
5         Name '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 Codebeispiel

1.3.2.5 RETI-Patch Pass

1.3.2.5.1 Zweck

1.3.2.5.2 Abstrakte Syntax

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

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

1.3.2.5.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.6',
6       [
7         # // Exp(GoTo(Name('main.5')))
8         # // not included 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        # // not included 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        # // not included 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     # // not included 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 Zweck

1.3.2.6.2 Konkrete und Abstrakte Syntax

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

Grammar 1.3.5: Konkrete Syntax für $L_{RETI.Lex}$


```

16 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 # // not included 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 # // not included 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 # // not included 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 Funktionen

1.3.3.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 1.13 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

```

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

```

Code 1.13: PicoC-Code für 3 Funktionen

Im **Abstract Syntax Tree** in Code 1.14 wird eine **Funktion**, wie z.B. `void fun(int param){ return param; }` mit der Komposition `FunDef(IntType(), Name('fun'), [Alloc(Writeable(), IntType()), Name('fun')])`, `[Return(Exp(Name('param')))]` dargestellt. Die einzelnen **Attribute** dieses Container-Knoten sind in Tabelle 1.5 erklärt.

```

1 File
2   Name './verbose_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return
10          Empty
11      ],
12     FunDef
13       VoidType 'void',
14       Name 'fun1',
15       [],
16       [],
17     FunDef
18       IntType 'int',
19       Name 'fun2',
20       [],
21       [
22         Return
23           Num '1'
24       ]
25   ]

```

Code 1.14: Abstract Syntax Tree für 3 Funktionen

Im **PicoC-Blocks Pass** in Code 1.15 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`⁵ beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<number>`⁶.

Es werden parallel dazu neue Zuordnungen im **Dictionary** `fun_name_to_block_name` hinzugefügt. Das **Dictionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<number>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu⁷. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<number>` angehängt haben.

```

1 File
2   Name './verbose_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        ]

```

⁵Eine Erklärung dazu ist in Unterkapitel 1.3.2.2.1 zu finden.

⁶Der **Grund** dafür kann im Unterkapitel 1.3.2.2.1 nachgelesen werden.

⁷Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 1.15: PicoC-Blocks Pass für 3 Funktionen

Im **PicoC-Mon Pass** in Code 1.16 werden die `FunDef(datatype, name, allocs, stmts)`-Container-Knoten komplett aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Container-Knoten nur noch Blöcke befinden.

```

1 File
2   Name './verbose_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 1.16: PicoC-Mon Pass für 3 Funktionen

Nach dem **RETI Pass** in Code 1.17 gibt es nur noch **RETI-Instructions**, die Blöcke wurden entfernt und die **RETI-Instructions** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man die Funktionen nicht mehr direkt ausmachen, denn die **Kommentare** enthalten die **Labelbezeichner** `<name>.<nummer>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer, dem **Namen** der jeweiligen **Funktion** entsprechen.

Da es in der `main`-Funktion keinen **Funktionsaufruf** gab, wird der Code, der nach der **Instruction** in der **markierten Zeile** kommt nicht mehr betreten. Funktionen sind im **RETI-Code** nur dadurch existent, dass im RETI-Code **Sprünge** (z.B. `JUMP<rel> <im>`) zu den jeweils richtigen Positionen gemacht werden, nämlich dorthin, wo die **RETI-Instructions**, die aus den **Statemens** einer **Funktion** kompiliert wurden anfangen.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.2'))))
3 # // not included Exp(GoTo(Name('main.2'))))
4 # // Block(Name('main.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.1'), [])
8 # Return(Empty())

```

```

9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1')))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 LOADIN BAF PC -1;

```

Code 1.17: RETI-Blocks Pass für 3 Funktionen

1.3.3.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 1.13 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 1.18 ist die `main`-Funktion allerdings **nicht** die **erste** Funktion. Daher muss dafür gesorgt werden, dass die `main`-Funktion die erste Funktion ist, die ausgeführt wird.

```

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

```

Code 1.18: PicoC-Code für Funktionen, wobei die `main` Funktion nicht die erste Funktion ist

Im **RETI-Blocks Pass** in Code 1.19 sind die **Funktionen** nur noch durch **Blöcke** umgesetzt.

```

1 File
2   Name './verbose_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 1.19: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Eine simple Möglichkeit ist es, die **main-Funktion** einfach nach **vorne** zu schieben, damit diese als **erstes** ausgeführt wird. Im `File(name, decls_defs)`-Container-Knoten muss dazu im `decls_defs`-Attribut, welches eine **Liste von Funktionen** ist, die **main-Funktion** an Index 0 geschoben werden.

Eine andere Möglichkeit und die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die **main-Funktion** nicht die erste auftauchende Funktion ist, einen `start.<number>`-Block als ersten Block einzufügen, der einen `GoTo(Name('main.<number>'))`-Container-Knoten enthält, der im **RETI Pass 1.21** in einen Sprung zur **main-Funktion** übersetzt wird.

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für **Studenten**, welche die Verwender des **Piocc-Compilers** sein werden vermutlich am **intuitivsten** ist, wenn der **RETI-Code** für die Funktionen an denselben Stellen relativ zueinander verortet ist, wie die Funktionsdefinitionen im **PicoC-Code**.

Das **Einsetzen** des `start.<number>`-Blockes erfolgt im **RETI-Patch Pass** in Code 1.20, da der **RETI-Patch**-Pass der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die **main-Funktion** nicht die erste Funktion ist.

```

1 File
2   Name './verbose_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 1.20: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

Im **RETI Pass** in Code 1.21 wird das `GoTo(Name('main.<nummer>'))` durch den entsprechenden Sprung `JUMP <distanz_zur_main_funktion>` ersetzt und die Blöcke entfernt.

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.0'))))
3 JUMP 8;
4 # // Block(Name('fun1.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun2.1'), [])
8 # // Return(Num('1'))
9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Return(Stack(Num('1')))
14 LOADIN SP ACC 1;
15 ADDI SP 1;
16 LOADIN BAF PC -1;
17 # // Block(Name('main.0'), [])
18 # Return(Empty())
19 LOADIN BAF PC -1;

```

Code 1.21: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

1.3.3.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

In der Programmiersprache L_C und somit auch L_{PicoC} ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehler-**

meldungen ausgegeben werden können, wenn der **Prototyp** (Definition 1.6) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

Die Deklaration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 1.22 **markierten Zeile 1** oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen 3-4**.

In dem Beispiel in Code 1.22 erfolgt ein **Funktionsaufruf** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdeklaration**, wie in der **markierten Zeile 1** notwendig. Beim **Funktionsaufruf** zur Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen 3-4** zu sehen ist.

Definition 1.6: Funktionsprototyp

*Deklaration einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parameterreihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktionendeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.^{a,b}*

^aDer **Funktionsprototyp** ist von der **FunktionsSignatur** zu unterscheiden, die in Programmiersprache wie C++ und Java für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentypen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentypen** usw. unterscheiden.

^bWhat is the difference between function prototype and function signature?

```

1 int fun2(int var);
2
3 void fun1() {
4 }
5
6 void main() {
7     fun1();
8     int var = fun2(42);
9     return;
10 }
11
12 int fun2(int var) {
13     return var;
14 }
```

Code 1.22: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 1.23 für das Beispiel in Code 1.22 dargestellt ist. Die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` werden wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette Komposition der **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(), IntType('int'), Name('var'))])` zugewiesen.

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 1.7). Die **Scopes** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**⁸ Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** ein **Funktionswechsel** eintritt und

⁸D.h. von der **Wurzel** zu den **Blättern** eines Baumes

über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

Ein Grund, warum **Scopes** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel**, die aus dem **Bezeichner** einer Variable und einem angehängten **Suffix** bestehen, in der als **Dictionary** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt den **Scope** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im `Name(str)`-Token-Knoten des `name`-Attributtes eines **Symbols** der Symboltabelle angehängt. Zur besseren Vorstellung ist dies in Code 1.23 **markiert**.

Die Variable `var@main`, bei der es sich um eine **Lokale Variable** der `main`-Funktion handelt, ist nur innerhalb des **Codeblocks** {} der `main`-Funktion **sichtbar** und die Variable `var@fun2` bei der es sich um einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** {} der Funktion `fun2` **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symbols** in der **Symboltabelle** an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird. Und da die Zuordnungen im **Dictionary** **eindeutig** sind, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

Das Zeichen @ wurde aus einem bestimmten Grund als **Trennzeichen** verwendet, nämlich, weil kein Bezeichner das Zeichen @ jemals selbst enthalten kann. Damit ist ausgeschlossen, dass falls ein **Benutzer** des PicoC-Compilers zufällig auf die Idee kommt seine Funktion genauso zu nennen (z.B. `var@fun2` als Funktionsname), es zu Problemen kommt, weil bei einem Nachschlagen der **Variable** die **Funktion** nachgeschlagen wird.

Definition 1.7: Scope (bzw. Sichtbarkeitsbereich)

*Bereich in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.*^a

^aThiemann, „Einführung in die Programmierung“.

```

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')

```

```

26     value or address:    Empty()
27     position:           Pos(Num('6'), Num('5'))
28     size:               Empty()
29   },
30   Symbol
31   {
32     type qualifier:      Writeable()
33     datatype:            IntType('int')
34     name:                Name('var@main')
35     value or address:    Num('0')
36     position:            Pos(Num('7'), Num('6'))
37     size:                Num('1')
38   },
39   Symbol
40   {
41     type qualifier:      Writeable()
42     datatype:            IntType('int')
43     name:                Name('var@fun2')
44     value or address:    Num('0')
45     position:            Pos(Num('11'), Num('13'))
46     size:                Num('1')
47   }
48 ]

```

Code 1.23: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

1.3.3.3 Funktionsaufruf

1.3.3.3.1 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 1.24: PicoC-Code für Funktionsaufruf ohne Rückgabewert

```

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

```

Code 1.25: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

```

1 File
2   Name './verbose_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        # // // Call(Name('stack_fun'), [Name('local_var')])
11        # StackMalloc(Num('2'))
12        SUBI SP 2;
13        # Ref(Global(Num('0')))
14        SUBI SP 1;
15        LOADi IN1 0;
16        ADD IN1 DS;
17        STOREIN SP IN1 1;
18        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))
19        MOVE BAF ACC;
20        ADDI SP 3;
21        MOVE SP BAF;
22        SUBI SP 4;
23        STOREIN BAF ACC 0;
24        LOADi ACC GoTo(Name('addr@next_instr'));
25        ADD ACC CS;
26        STOREIN BAF ACC -1;
27        # Exp(GoTo(Name('stack_fun.0'))))
28        Exp(GoTo(Name('stack_fun.0'))))
29        # RemoveStackframe()
30        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   ]

```

Code 1.26: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

```

1 # // Block(Name('start.2'), [])
2 # // Exp(GoTo(Name('main.1'))))
3 # // not included Exp(GoTo(Name('main.1'))))
4 # // Block(Name('main.1'), [])
5 # // Exp(Alloc(Writable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
6 ↪   Name('local_var')))
7 # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
8 # // // Call(Name('stack_fun'), [Name('local_var')])
9 # StackMalloc(Num('2'))
10 SUBI SP 2;
11 # Ref(Global(Num('0'))))
12 SUBI SP 1;
13 LOADI IN1 0;
14 ADD IN1 DS;
15 STOREIN SP IN1 1;
16 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
17 MOVE BAF ACC;
18 ADDI SP 3;
19 MOVE SP BAF;
20 SUBI SP 4;
21 STOREIN BAF ACC 0;
22 # LOADI ACC GoTo(Name('addr@next_instr'));
23 LOADI ACC 14;
24 ADD ACC CS;
25 STOREIN BAF ACC -1;
26 # Exp(GoTo(Name('stack_fun.0'))))
27 JUMP 5;
28 # RemoveStackframe()
29 MOVE BAF IN1;
30 LOADIN IN1 BAF 0;
31 MOVE IN1 SP;
32 # Return(Empty())
33 LOADIN BAF PC -1;
34 # // Block(Name('stack_fun.0'), [])
35 # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('st'))), Name('param')))
36 # // Exp(Alloc(Writable(), IntType('int'), Name('local_var')))

```

```

36 # Return(Empty())
37 LOADIN BAF PC -1;

```

Code 1.27: RETI-Pass für Funktionsaufruf ohne Rückgabewert

1.3.3.3.2 Mit Rückgabewert

```

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

```

Code 1.28: PicoC-Code für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './verbose_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(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('stack_fun'),
15         ↪      []))
16         // Assign(Name('var'), Call(Name('stack_fun'), []))
17         // // Call(Name('stack_fun'), [])
18         StackMalloc(Num('2'))
19         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
20         Exp(GoTo(Name('stack_fun.1')))
21         RemoveStackframe()
22         Assign(Global(Num('0')), Stack(Num('1')))
23         Return(Empty())
24      ]
25   ]

```

Code 1.29: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './verbose_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(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('stack_fun'),
21       ↪   []))
22       # // Assign(Name('var'), Call(Name('stack_fun'), []))
23       # // // Call(Name('stack_fun'), [])
24       # StackMalloc(Num('2'))
25       SUBI SP 2;
26       # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
27       MOVE BAF ACC;
28       ADDI SP 2;
29       MOVE SP BAF;
30       SUBI SP 2;
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       # Assign(Global(Num('0')), Stack(Num('1')))
42       LOADIN SP ACC 1;
43       STOREIN DS ACC 0;
44       ADDI SP 1;
45       # Return(Empty())
46       LOADIN BAF PC -1;
47     ]
48   ]

```

Code 1.30: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1 # // Block(Name('start.2'), [])
2 # // Exp(GoTo(Name('main.0')))
3 JUMP 7;
4 # // Block(Name('stack_fun.1'), [])
5 # // Return(Num('42'))

```



```

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

```

Code 1.31: RETI-Pass für Funktionsaufruf mit Rückgabewert

1.3.3.3 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 1.32: PicoC-Code für Call by Sharing für Arrays

```

1 File
2   Name './verbose_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        // // Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')])
21        StackMalloc(Num('2'))
22        Ref(Global(Num('0'))))
23        Ref(Global(Num('6'))))
24        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
25        Exp(GoTo(Name('stack_fun.1'))))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29    ]
30  ]

```

Code 1.33: 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'),
7       ↪ [Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8       ↪ Name('param1')), Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')),
9       ↪ Name('param2'))])
10      name:                 Name('stack_fun')
11      value or address:     Empty()
12      position:             Pos(Num('1'), Num('5'))
13      size:                 Empty()
14    },
15    Symbol
16    {
17      type qualifier:       Writable()
18      datatype:             PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
19      name:                 Name('param1@stack_fun')
20      value or address:     Num('0')
21      position:             Pos(Num('1'), Num('21'))
22      size:                 Num('1')
23    }
24  ]

```

```

20     },
21     Symbol
22     {
23         type qualifier:      Writeable()
24         datatype:            PntrDecl(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:      Writeable()
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:      Writeable()
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 1.34: Symboltabelle für Call by Sharing für Arrays

```

1 File
2   Name './verbose_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
8           ↪ IntType('int'))), Name('param1')))
9         # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
10        # Return(Empty())
11        LOADIN BAF PC -1;
12      ],
13    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         # // // 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 ]

```

Code 1.35: RETI-Block Pass für Call by Sharing für Arrays

1.3.3.3.4 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 1.36: PicoC-Code für Call by Value für Structs

```

1 File
2   Name './verbose_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        // // Call(Name('stack_fun'), [Name('local_var')])
16        StackMalloc(Num('2'))
17        Assign(Stack(Num('3')), Global(Num('0')))
18        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
19        Exp(GoTo(Name('stack_fun.1')))
20        RemoveStackframe()
21        Return(Empty())
22      ]
23    ]

```

Code 1.37: PicoC-Mon Pass für Call by Value für Structs

```

1 File
2   Name './verbose_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        # // // Call(Name('stack_fun'), [Name('local_var')])
17        # StackMalloc(Num('2'))
18        SUBI SP 2;
19        # Assign(Stack(Num('3')), Global(Num('0')))
20        SUBI SP 3;
21        LOADIN DS ACC 0;
22        STOREIN SP ACC 1;
23        LOADIN DS ACC 1;
24        STOREIN SP ACC 2;
25        LOADIN DS ACC 2;

```

```
26     STOREIN SP ACC 3;
27     # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
28     MOVE BAF ACC;
29     ADDI SP 5;
30     MOVE SP BAF;
31     SUBI SP 5;
32     STOREIN BAF ACC 0;
33     LOADI ACC GoTo(Name('addr@next_instr'));
34     ADD ACC CS;
35     STOREIN BAF ACC -1;
36     # Exp(GoTo(Name('stack_fun.1')))
37     Exp(GoTo(Name('stack_fun.1')))
38     # RemoveStackframe()
39     MOVE BAF IN1;
40     LOADIN IN1 BAF 0;
41     MOVE IN1 SP;
42     # Return(Empty())
43     LOADIN BAF PC -1;
44 ]
45 ]
```

Code 1.38: RETI-Block Pass für Call by Value für Structs

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 (besucht am 27.04.2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: <https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/> (besucht am 18.07.2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).

Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (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).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).