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

Betreung:
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

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Implementierung

1.1 Lexikalische Analyse

1.1.1 Konkrette Syntax für die Lexikalische Analyse

COMMENT	::=	"//" /[^\n]*/ | "/*" /(. | \n)*?/ "*/"	*L_Comment*
RETI_COMMENT.2	::=	"//""␣"?"#"/[^\n]*/	
DIG_NO_0	::=	"1" | "2" | "3" | "4" | "5"	*L_Arith*
	|	"6" | "7" | "8" | "9"	
DIG_WITH_0	::=	"0" | *DIG_NO_0*	
NUM	::=	"0" | *DIG_NO_0DIG_WITH_0*∗	
ASCII_CHAR	::=	"␣".."~"	
CHAR	::=	"'"*ASCII_CHAR*"'"	
FILENAME	::=	*ASCII_CHAR* + ".*picoc*"	
LETTER	::=	"*a*".."*z*" | "*A*".."*Z*"	
NAME	::=	(*LETTER* | "_") (LETTER — DIG_WITH_0 — "_")*	
name	::=	*NAME* | *INT_NAME* | *CHAR_NAME*	
	|	*VOID_NAME*	
NOT	::=	"~"	
REF_AND	::=	"&"	
un_op	::=	*SUB_MINUS* | *LOGIC_NOT* | *NOT*	
	|	*MUL_DEREF_PNTR* | *REF_AND*	
MUL_DEREF_PNTR	::=	"∗"	
DIV	::=	"/"	
MOD	::=	"%"	
*prec*1*_op*	::=	*MUL_DEREF_PNTR* | *DIV* | *MOD*	
ADD	::=	"+"	
SUB_MINUS	::=	"−"	
*prec*2*_op*	::=	*ADD* | *SUB_MINUS*	
LT	::=	"<"	*L_Logic*
LTE	::=	"<="	
GT	::=	">"	
GTE	::=	">="	
rel_op	::=	*LT* | *LTE* | *GT* | *GTE*	
EQ	::=	"=="	
NEQ	::=	"!="	
eq_op	::=	*EQ* | *NEQ*	
LOGIC_NOT	::=	"!"	

Grammar 1.1.1: Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 1

$INT\_DT.2$	::=	"*int*"	*L_Assign_Alloc*
$INT\_NAME.3$	::=	"*int*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
$CHAR\_DT.2$	::=	"*char*"	
$CHAR\_NAME.3$	::=	"*char*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
$VOID\_DT.2$	::=	"*void*"	
$VOID\_NAME.3$	::=	"*void*" $(LETTER \mid DIG\_WITH\_0 \mid$ "_")+	
prim_dt	::=	$INT\_DT \mid CHAR\_DT \mid VOID\_DT$	

Grammar 1.1.2: Konkrette Syntax für die Lexikalische Analyse in EBNF, Teil 2

1.1.2 Basic Lexer

1.2 Syntaktische Analyse

1.2.1 Konkrette Syntax für die Syntaktische Analyse

In 1.2.1

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith +*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*L_Array +*
		| *input_exp* | *print_exp* | *prim_exp*	*L_Pntr +*
un_exp	::=	*un_opun_exp* | *post_exp*	*L_Struct + L_Fun*
input_exp	::=	"*input*""("")"	*L_Arith*
print_exp	::=	"*print*""("*logic_or*")"	
*arith_prec*1	::=	*arith_prec*1 *prec*1*_op un_exp* | *un_exp*	
*arith_prec*2	::=	*arith_prec*2 *prec*2*_op arith_prec*1 | *arith_prec*1	
arith_and	::=	*arith_and* "&" *arith_prec*2 | *arith_prec*2	
arith_oplus	::=	*arith_oplus* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_oplus* | *arith_oplus*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_oprel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
type_spec	::=	*prim_dt* | *struct_spec*	*L_Assign_Alloc*
alloc	::=	*type_spec pntr_decl*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	
initializer	::=	*logic_or* | *array_init* | *struct_init*	
init_stmt	::=	*alloc* "=" *initializer*";"	
const_init_stmt	::=	"*const*" *type_spec name* "=" *NUM*";"	
pntr_deg	::=	"*"*	*L_Pntr*
pntr_decl	::=	*pntr_deg array_decl* | *array_decl*	
array_dims	::=	("["*NUM*"]")*	*L_Array*
array_decl	::=	*name array_dims* | "("*pntr_decl*")"*array_dims*	
array_init	::=	"{"*initializer*("," *initializer*) * "}"	
array_subscr	::=	*post_exp*"["*logic_or*"]"	
struct_spec	::=	"*struct*" *name*	*L_Struct*
struct_params	::=	(*alloc*";")+	
struct_decl	::=	"*struct*" *name* "{"*struct_params*"}"	
struct_init	::=	"{""."*name*"="*initializer* ("," "."name"="initializer)*"}"	
struct_attr	::=	*post_exp*"."*name*	
if_stmt	::=	"*if*""("*logic_or*")" *exec_part*	*L_If_Else*
if_else_stmt	::=	"*if*""("*logic_or*")" *exec_part* "*else*" *exec_part*	
while_stmt	::=	"*while*""("*logic_or*")" *exec_part*	*L_Loop*
do_while_stmt	::=	"*do*" *exec_part* "*while*""("*logic_or*")"";"	

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

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

Grammar 1.2.2: Konkrette 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	`a()`	Funktionsaufruf	Links, dann rechts →
	`a[]`	Indexzugriff	
	`a.b`	Attributzugriff	
2	`-a`	Unäres Minus	Rechts, dann links ←
	`!a ~a`	Logisches NOT und Bitweise NOT	
	`*a &a`	Dereferenz und Referenz, auch Adresse-von	
3	`a*b a/b a%b`	Multiplikation, Division und Modulo	Links, dann rechts →
4	`a+b a-b`	Addition und Subtraktion	
5	`a<b a<=b a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	`a==b a!=b`	Gleichheit und Ungleichheit	
7	`a&b`	Bitweise UND	
8	`a^b`	Bitweise XOR (exclusive or)	
9	`a|b`	Bitweise ODER (inclusive or)	
10	`a&&b`	Logiches UND	
11	$a \| \| b$	Logisches ODER	
12	`a=b`	Zuweisung	Rechts, dann links ←
13	`a,b`	Komma	Links, dann rechts →

Tabelle 1.1: Präzidenzregeln von PicoC

[1] *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             pntr_decl
10              pntr_deg  *
11              array_decl
12                array_dims
13                  5
14                  6
15                pntr_decl
16                  pntr_deg  *
17                  array_decl
18                    array_dims
19                    type_spec
20                      prim_dt  int
21            name  attr
22    decl_def
23      fun_def
24        type_spec
25          prim_dt  void
26        pntr_deg
27        name  main
28        fun_params
29        decl_exec_stmts
30          decl_part
31            decl_exp_stmt
32              alloc
33                pntr_decl
34                  pntr_deg  *
35                  array_decl
36                    array_dims
```

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

Code 1.3: Derivation Tree nach Derivation Tree Vereinfachung

1.2.5 Abstrakt Syntax Tree Generierung

1.2.5.1 PicoC-Knoten

PiocC-Knoten	Beschreibung
`Name(val)`	Ein **Bezeichner**, z.B. `my_fun, my_var` usw. , aber da es keine gute Kurzform für `Identifier()` (englisches Wort für Bezeichner) gibt, wurde dieser Knoten `Name()` genannt.
`Num(val)`	Eine **Zahl**, z.B. `42, -3` usw.
`Char(val)`	Ein **Zeichen** der **ASCII-Zeichenkodierung**, z.B. `'c', '*'` usw.
`Minus(), Not(), DerefOp(), RefOp(), LogicNot()`	Die **unären Operatoren** `un_op`: `-a, ~a, *a, &a !a`.
`Add(), Sub(), Mul(), Div(), Mod(), Oplus(), And(), Or(), LogicAnd(), LogicOr()`	Die **binären Operatoren** `bin_op`: `a + b, a - b, a * b, a / b, a % b, a ∧ b, a & b, a | b, a && b, a || b`.
`Eq(), NEq(), Lt(), LtE(), Gt(), GtE()`	Die **Relationen** `rel`: `a == b, a != b, a < b, a <= b, a > b, a >= b`.
`Const(), Writeable()`	Die **Type Qualifier** `type_qual`: `const`, was für ein **nicht beschreibbare Konstante** steht und das **nicht** Angeben von `const`, 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** `datatype` eingeordnet werden: `int`, `char`, `void`.
`Placeholder()`	**Platzhalter** für einen Knoten, der diesen später **ersetzt**.
`BinOp(exp, bin_op, exp)`	Container für eine **binäre Operation** mit 2 Expressions: `<exp1> <bin_op> <exp2>`
`UnOp(un_op, exp)`	Container für eine **unäre Operation** mit einer Expression: `<un_op> <exp>`.
`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: `<exp1> <rel> <exp2>`
`ToBool(exp)`	Container für einen **Arithmetischen Ausdruck**, wie z.B. `1 + 3` oder einfach nur `3`, der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis $x > 1$ auf 1 abgebildet wird.
`Alloc(type_qual, datatype, name, local_var_or_param)`	**Container** für eine **Allokation** `<type_qual> <datatype> <name>` mit den notwendigen Knoten `type_qual`, `datatype` und `name`, die alle für einen Eintrag in der **Symboltabelle** notwendigen Informationen enthalten. Zudem besitzt er ein verstecktes Attribut `local_var_or_param`, 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 `lhs` ein `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` sein kann und `exp` ein beliebiger **Logischer Ausdruck** sein kann: `lhs = exp`.

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 **Pointerdatentyp** steht: `<prim_dt> *<var>`, 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** `&<var>` steht und die **Adresse** einer **Location** 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 **Indexzugriff** auf einen **Array-** oder **Pointerdatentyp**: `<var>[<i>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`ArrayDecl(nums, datatype)`	Container, der für den **Arraydatentyp** steht: `<prim_dt> <var>[<i>]`, wobei das **Attribut** `nums` eine Liste von `Num('x')` ist, die die **Dimensionen** des Arrays angibt und `datatype` der Datentyp ist, der über das Anwenden von `Subscript()` auf das Array zugreifbar ist.
`Array(exps, datatype)`	Container für den **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. `{{1, 2}, {3, 4}}`. 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 **Indexzugriff** auf einen **Array-** oder **Pointerdatentyp**: `<var>[<i>]`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `exp2` der Index ist auf den zugegriffen werden soll.
`StructSpec(name)`	Container für einen selbst definierten **Structdatentyp**: `struct <name>`, wobei das **Attribut** `name` festlegt, welchen **selbst definierte** Structdatentyp dieser Container-Knoten repräsentiert.
`Attr(exp, name)`	Container für den **Attributzugriff** auf einen **Structdatentyp**: `<var>.<attr>`, wobei `exp1` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation sein kann oder ein `Name('var')` sein kann und `name` das Attribut ist, auf das zugegriffen werden soll.

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 versteckte 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_goto)`	Container für ein **If Statement** `if(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`IfElse(exp, stmts_goto1, stmts_goto2)`	Container für ein **If-Else Statement** `if(<exp>) { <stmts2> } else { <stmts2> }` inklusive **Codition** `exp` und 2 **Branches** `stmts_goto1` und `stmts_goto2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Statements** oder `GoTo(Name('block.xyz'))`'s stehen können.
`While(exp, stmts_goto)`	Container für ein **While-Statement** `while(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`DoWhile(exp, stmts_goto)`	Container für ein **Do-While-Statement** `do { <stmts> } while(<exp>);` inklusive **Condition** `exp` und einem **Branch** `stmts_goto`, 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 Cotainer 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 Cotainer für die Parameter dient und `stmts_blocks` eine Liste von **Statemetns** 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 Kompiliervorang wichtige Information in seinen versteckte 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 **Funkionen** 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** 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 **Kompiliervorangs** 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.
`RETIComment(value)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** 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

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

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

Definition 1.1: Token-Knoten

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

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

[a] Thiemann, „Compilerbau“.

Definition 1.2: Container-Knoten

*Dient als **Container** für andere **Container-Knoten** und **Token-Knoten**. Die **Container-Knoten** werden optimalerweise immer so gewählt, dass sie **mehrere Produktionen** der **Konkretten 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]

[a] Wie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. `1 % 3` und **Logischen Ausdrücke**, wie z.B. `1 && 2 < 3`, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.
[b] Thiemann, „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 **Interrruptvektornummer** (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 **Programnamen**, 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()`	**R**eturn-**F**rom-**I**nterrupt 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 versteckes 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 repetetiven Inhalt zu füllen, wurden die zahlreichen Kompostionen **ausgelassen**, bei denen einfach nur `exp` durch $\texttt{Stack}(\texttt{Num}(\texttt{'x'})), \texttt{x} \in \mathbb{N}$ 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

un_op	::=	$Minus() \mid Not()$	*L_Arith*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(str) \mid Num(str) \mid Char(str)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), None)$	
exp_stmts	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
un_op	::=	$LogicNot()$	*L_Logic*
rel	::=	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	::=	$LogicAnd() \mid LogicOr()$	
exp	::=	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle)$	
	$\mid$	$ToBool(\langle exp\rangle)$	
$type\_qual$	::=	$Const() \mid Writeable()$	*L_Assign_Alloc*
$datatype$	::=	$IntType() \mid CharType() \mid VoidType()$	
lhs	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str)) \mid \langle rel\_loc\rangle$	
exp_stmts	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
$stmt$	::=	$Assign(\langle lhs\rangle, \langle exp\rangle)$	
	$\mid$	$Exp(\langle exp\_stmts\rangle)$	
$datatype$	::=	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
$deref\_loc$	::=	$Ref(\langle ref\_loc\rangle) \mid \langle ref\_loc\rangle$	
ref_loc	::=	$Name(str)$	
	$\mid$	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Attr(\langle ref\_loc\rangle, Name(str))$	
exp	::=	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Ref(\langle ref\_loc\rangle)$	
$datatype$	::=	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	::=	$StructSpec(Name(str))$	*L_Struct*
exp	::=	$Attr(\langle ref\_loc\rangle, Name(str))$	
	$\mid$	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
$decl\_def$	::=	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
$stmt$	::=	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	$\mid$	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
$stmt$	::=	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	$\mid$	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	::=	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
exp_stmts	::=	$Call(Name(str), \langle exp\rangle*)$	
$stmt$	::=	$Return(\langle exp\rangle)$	
$decl\_def$	::=	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
$file$	::=	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

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.

```
File
  Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')],
        ↪  PntrDecl(Num('1'), IntType('int')))), Name('attr'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
        ↪  PntrDecl(Num('1'), StructSpec(Name('st'))))), Name('var')))
      ]
  ]
```

Code 1.4: Abstract Syntax Tree aus vereinfachtem Derivarion Tree generiert

1.3 Code Generierung

1.3.1 Übersicht

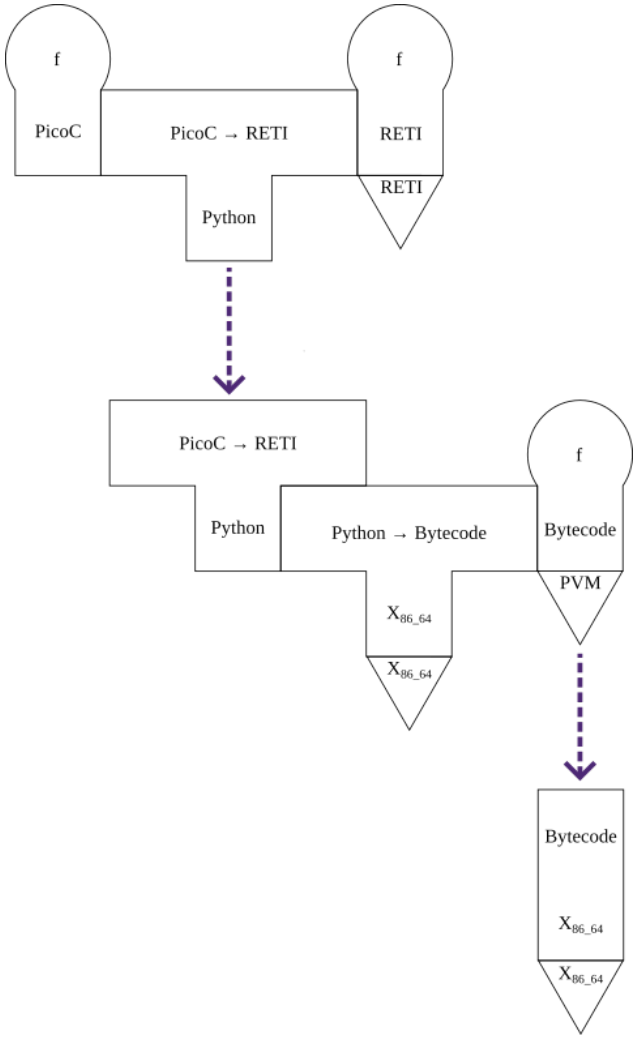

Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

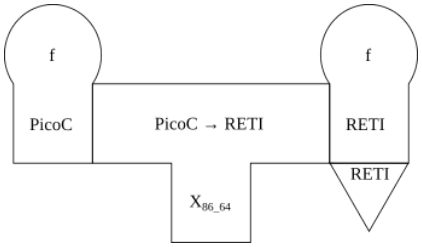

Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

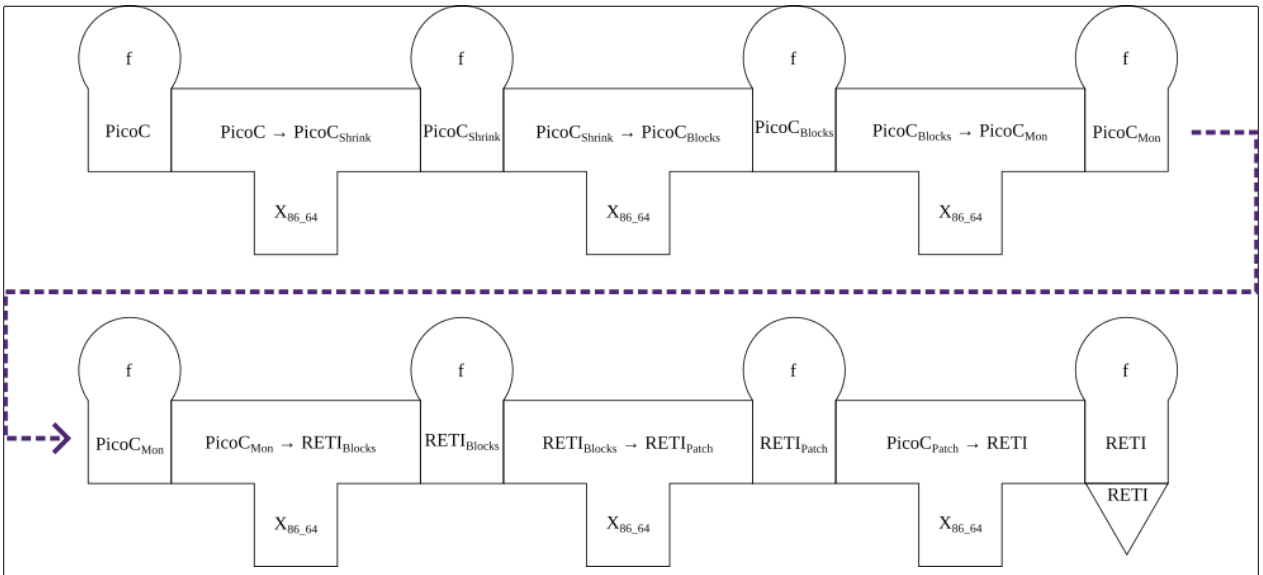

Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

1.3.2 Passes

1.3.2.1 PicoC-Shrink Pass

1.3.2.1.1 Codebeispiel

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

Code 1.5: PicoC Code für Codebespiel

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

```
      Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
      Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
      While
        Num '1',
        [
          If
            Atom
              Name 'n',
              Eq '==',
              Num '1',
            [
              Return(Empty())
            ]
          Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
          Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
        ]
    ]
]
```

Code 1.6: Abstract Syntax Tree für Codebespiel

```
File
  Name './example_faculty_it.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
        Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1')),
        While
          Num '1',
          [
            If
              Atom
                Name 'n',
                Eq '==',
                Num '1',
              [
                Return(Empty())
              ]
            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
          ]
      ]
  ]
```

Code 1.7: PicoC Shrink Pass für Codebespiel

1.3.2.2 PicoC-Blocks Pass

1.3.2.2.1 Abstrakte Syntax

$decl\_def$	$::=$	$FunDef(\langle datatype\rangle, Name(str),$ $Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle block\rangle*)$	L_Fun
$block$	$::=$	$Block(Name(str), \langle stmt\rangle*)$	L_Blocks
$stmt$	$::=$	$GoTo(Name(str)) \mid NewStackframe(Name(), GoTo(str))$	
	$\mid$	$RemoveStackframe() \mid SetScope(Name(str))$	
	$\mid$	$SingleLineComment(str, str)$	

Grammar 1.3.1: Abstrakte Syntax für L_{PicoC_Blocks}

1.3.2.2.2 Codebeispiel

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Block
          Name 'main.5',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('n')), Num('4'))
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.4'))
          ],
        Block
          Name 'condition_check.4',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.3'))
              ],
              [
                GoTo(Name('while_after.0'))
              ]
          ],
        Block
          Name 'while_branch.3',
          [
            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
            IfElse
              Atom
                Name 'n',
                Eq '==',
                Num '1',
              [
                GoTo(Name('if.2'))
              ],
              [
                GoTo(Name('if_else_after.1'))
              ]
```

```
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 Codebespiel

1.3.2.3 PicoC-Mon Pass

1.3.2.3.1 Abstrakte Syntax

ref_loc	::=	$Stack(Num(str)) \mid Global(Num(str))$	*L_Assign_Alloc*
	$\mid$	$Stackframe(Num(str))$	
$error\_data$	::=	$\langle exp \rangle \mid Pos(Num(str), Num(str))$	
exp	::=	$Stack(Num(str)) \mid Ref(\langle ref_loc \rangle, \langle datatype \rangle, \langle error_data \rangle)$	
$stmt$	::=	$Exp(\langle exp \rangle)$	
	$\mid$	$Assign(Alloc(Writeable(), StructSpec(Name(str)), Name(str)),$ $Struct(Assign(Name(str), \langle exp \rangle)+, \langle datatype \rangle))$	
	$\mid$	$Assign(Alloc(Writeable(), ArrayDecl(Num(str)+, \langle datatype \rangle)),$ $Name(str)), Array(\langle exp \rangle+, \langle datatype \rangle))$	
$symbol\_table$	::=	$SymbolTable(\langle symbol \rangle)$	*L_Symbol_Table*
$symbol$	::=	$Symbol(\langle type_qual \rangle, \langle datatype \rangle, \langle name \rangle, \langle val \rangle, \langle pos \rangle, \langle size \rangle)$	
$type\_qual$	::=	$Empty()$	
$datatype$	::=	$BuiltIn() \mid SelfDefined()$	
$name$	::=	$Name(str)$	
val	::=	$Num(str) \mid Empty()$	
pos	::=	$Pos(Num(str), Num(str)) \mid Empty()$	
$size$	::=	$Num(str) \mid Empty()$	

Grammar 1.3.2: Abstrakte Syntax für L_{PicoC_Mon}

Definition 1.3: Symboltabelle

1.3.2.3.2 Codebeispiel

```
File
  Name './example_faculty_it.picoc_mon',
  [
    Block
      Name 'main.5',
      [
        // Assign(Name('n'), Num('4'))
        Exp(Num('4'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('res'), Num('1'))
        Exp(Num('1'))
        Assign(Global(Num('1')), Stack(Num('1')))
        // While(Num('1'), [])
        Exp(GoTo(Name('condition_check.4')))
      ],
    Block
      Name 'condition_check.4',
      [
        // IfElse(Num('1'), [], [])
        Exp(Num('1')),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('while_branch.3'))
          ],
          [
            GoTo(Name('while_after.0'))
          ]
      ],
    Block
      Name 'while_branch.3',
      [
        // If(Atom(Name('n'), Eq('==')), Num('1')), [])
        // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
        Exp(Global(Num('0')))
        Exp(Num('1'))
        Exp(Atom(Stack(Num('2')), Eq('==')), Stack(Num('1')))),
        IfElse
          Stack
            Num '1',
          [
            GoTo(Name('if.2'))
          ],
          [
            GoTo(Name('if_else_after.1'))
          ]
      ],
    Block
      Name 'if.2',
      [
        Return(Empty())
      ],
    Block
      Name 'if_else_after.1',
      [
        // 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 Codebespiel

1.3.2.4 RETI-Blocks Pass

1.3.2.4.1 Abstrakte Syntax

$program$	$::=$	$Program(Name(str), \langle block \rangle *)$	*L_Program*
exp_stmts	$::=$	$GoTo(str)$	*L_Blocks*
$instrs\_before$	$::=$	$Num(str)$	
num_instrs	$::=$	$Num(str)$	
$block$	$::=$	$Block(Name(str), \langle instr \rangle *, \langle instrs\_before \rangle, \langle num\_instrs \rangle)$	
$instr$	$::=$	$GoTo(Name(str))$	

Grammar 1.3.3: Abstrakte Syntax für L_{RETI_Blocks}

1.3.2.4.2 Codebeispiel

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

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

```
75        # Return(Empty())
76        LOADIN BAF PC -1;
77      ],
78    Block
79      Name 'if_else_after.1',
80      [
81        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
82        # Exp(Global(Num('0')))
83        SUBI SP 1;
84        LOADIN DS ACC 0;
85        STOREIN SP ACC 1;
86        # Exp(Global(Num('1')))
87        SUBI SP 1;
88        LOADIN DS ACC 1;
89        STOREIN SP ACC 1;
90        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
91        LOADIN SP ACC 2;
92        LOADIN SP IN2 1;
93        MULT ACC IN2;
94        STOREIN SP ACC 2;
95        ADDI SP 1;
96        # Assign(Global(Num('1')), Stack(Num('1')))
97        LOADIN SP ACC 1;
98        STOREIN DS ACC 1;
99        ADDI SP 1;
100       # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
101       # Exp(Global(Num('0')))
102       SUBI SP 1;
103       LOADIN DS ACC 0;
104       STOREIN SP ACC 1;
105       # Exp(Num('1'))
106       SUBI SP 1;
107       LOADI ACC 1;
108       STOREIN SP ACC 1;
109       # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
110       LOADIN SP ACC 2;
111       LOADIN SP IN2 1;
112       SUB ACC IN2;
113       STOREIN SP ACC 2;
114       ADDI SP 1;
115       # Assign(Global(Num('0')), Stack(Num('1')))
116       LOADIN SP ACC 1;
117       STOREIN DS ACC 0;
118       ADDI SP 1;
119       # Exp(GoTo(Name('condition_check.4')))
120       Exp(GoTo(Name('condition_check.4')))
121     ],
122   Block
123     Name 'while_after.0',
124     [
125       # Return(Empty())
126       LOADIN BAF PC -1;
127     ]
128 ]
```

Code 1.10: RETI-Blocks Pass für Codebespiel

1.3.2.5 RETI-Patch Pass

1.3.2.5.1 Abstrakte Syntax

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

Grammar 1.3.4: Abstrakte Syntax für L_{RETI_Patch}

1.3.2.5.2 Codebeispiel

```
File
 Name './example_faculty_it.reti_patch',
 [
   Block
     Name 'start.6',
     [
       # // Exp(GoTo(Name('main.5')))
       # // patched out Exp(GoTo(Name('main.5')))
     ],
   Block
     Name 'main.5',
     [
       # // Assign(Name('n'), Num('4'))
       # Exp(Num('4'))
       SUBI SP 1;
       LOADI ACC 4;
       STOREIN SP ACC 1;
       # Assign(Global(Num('0')), Stack(Num('1')))
       LOADIN SP ACC 1;
       STOREIN DS ACC 0;
       ADDI SP 1;
       # // Assign(Name('res'), Num('1'))
       # Exp(Num('1'))
       SUBI SP 1;
       LOADI ACC 1;
       STOREIN SP ACC 1;
       # Assign(Global(Num('1')), Stack(Num('1')))
       LOADIN SP ACC 1;
       STOREIN DS ACC 1;
       ADDI SP 1;
       # // While(Num('1'), [])
       # Exp(GoTo(Name('condition_check.4')))
       # // patched out Exp(GoTo(Name('condition_check.4')))
     ],
   Block
     Name 'condition_check.4',
     [
       # // IfElse(Num('1'), [], [])
       # Exp(Num('1'))
       SUBI SP 1;
       LOADI ACC 1;
       STOREIN SP ACC 1;
       # IfElse(Stack(Num('1')), [], [])
       LOADIN SP ACC 1;
       ADDI SP 1;
```

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

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

Code 1.11: RETI-Patch Pass für Codebespiel

1.3.2.6 RETI Pass

1.3.2.6.1 Konkrette und Abstrakte Syntax

dig_no_0	::=	"1" | "2" | "3" | "4" | "5" | "6"	*L_Program*
		| "7" | "8" | "9"	
dig_with_0	::=	"0" | *dig_no_0*	
num	::=	"0" | *dig_no_0dig_with_0* $*$ | "$-$"*dig_with_0*$*$	
letter	::=	"*a*"..."*Z*"	
name	::=	*letter*(*letter* | *dig_with_0* | _)$*$	
reg	::=	"*ACC*" | "*IN*1" | "*IN*2" | "*PC*" | "*SP*"	
		| "*BAF*" | "*CS*" | "*DS*"	
arg	::=	*reg* | *num*	
rel	::=	"==" | "!=" | "<" | "<=" | ">"	
		| ">=" | "_*NOP*"	

Grammar 1.3.5: Konkrette Syntax für L_{RETI_Lex}

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

Grammar 1.3.6: Konkrette Syntax für L_{RETI_Parse}

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

Grammar 1.3.7: Abstrakte Syntax für L_{RETI}

1.3.2.6.2 Codebeispiel

```
# // Exp(GoTo(Name('main.5')))
# // patched out Exp(GoTo(Name('main.5')))
# // Assign(Name('n'), Num('4'))
# Exp(Num('4'))
SUBI SP 1;
LOADI ACC 4;
STOREIN SP ACC 1;
# Assign(Global(Num('0')), Stack(Num('1')))
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
# // Assign(Name('res'), Num('1'))
# Exp(Num('1'))
SUBI SP 1;
LOADI ACC 1;
```

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

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

Code 1.12: RETI Pass für Codebespiel

1.3.3 Umsetzung von Pointern

1.3.3.1 Referenzierung

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

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

Code 1.13: PicoC-Code für Pointer Referenzierung

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

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

Code 1.14: Abstract Syntax Tree für Pointer Referenzierung

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

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

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

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

Code 1.15: Symboltabelle für Pointer Referenzierung

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

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

Code 1.16: PicoC-Mon Pass für Pointer Referenzierung

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

```
File
  Name './example_pntr_ref.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('pntr'), Ref(Name('var')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.17: RETI-Blocks Pass für Pointer Referenzierung

1.3.3.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

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

```
void main() {
  int var = 42;
  int *pntr = &var;
  *pntr;
}
```

Code 1.18: PicoC-Code für Pointer Dereferenzierung

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

```
File
  Name './example_pntr_deref.ast',
  [
    FunDef
```

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

Code 1.19: Abstract Syntax Tree für Pointer Dereferenzierung

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

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

Code 1.20: PicoC-Shrink Pass für Pointer Dereferenzierung

1.3.4 Umsetzung von Arrays

1.3.4.1 Initialisierung von Arrays

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

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

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

Code 1.21: PicoC-Code für Array Initialisierung

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

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

Code 1.22: Abstract Syntax Tree für Array Initialisierung

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

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

[3] Das Widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.

Statischen Daten[4].

Das **Größe** des Arrays `datatype ar`$[\mathtt{dim}_1]\dots[\mathtt{dim}_k]$, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Arrays multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Arrayelemente**: $\mathtt{size}(\mathtt{datatype}(\mathtt{ar})) = \left(\prod_{j=1}^{n} \mathtt{dim}_j\right) \cdot \mathtt{size}(\mathtt{datatype})$[a].

[a] Die **Funktion** `type` ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die **Funktion** `size` nur bei einem **Datentyp** als **Funktionsargument** die **Größe dieses Datentyps** als **Zielwert** liefert

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('1')], IntType('int'))
        name:                   Name('ar@main')
        value or address:       Num('0')
        position:               Pos(Num('2'), Num('6'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun'), [])
        name:                   Name('fun')
        value or address:       Empty()
        position:               Pos(Num('5'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('2')], IntType('int'))
        name:                   Name('ar@fun')
        value or address:       Num('3')
        position:               Pos(Num('6'), Num('6'))
        size:                   Num('4')
      }
  ]
```

Code 1.23: Symboltabelle für Array Initialisierung

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

Im **PiocC-Mon Pass** in Code 1.24 werden zuerst die **Logischen Ausdrücke** in den Blättern des vom **Array-Initializers Container-Knoten** ausgehenden Baumes `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben[5].

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

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

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

```
File
  Name './example_array_init.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
        Exp(Num('3'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Exp(Num('4'))
        Assign(Global(Num('0')), Stack(Num('2')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        Exp(Num('3'))
        Exp(Num('4'))
        Exp(Num('5'))
        Exp(Num('6'))
        Assign(Stackframe(Num('3')), Stack(Num('4')))
        Return(Empty())
      ]
  ]
```

Code 1.24: PicoC-Mon Pass für Array Initialisierung

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

[5] Da der **Zuweisungsoperator** `=` **rechtsassoziativ** ist und auch rein **logisch**, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

`Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        LOADIN SP ACC 2;
        LOADIN SP IN2 1;
        ADD ACC IN2;
        STOREIN SP ACC 2;
        ADDI SP 1;
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('5'))
        SUBI SP 1;
        LOADI ACC 5;
        STOREIN SP ACC 1;
        # Exp(Num('6'))
        SUBI SP 1;
        LOADI ACC 6;
```

```
54          STOREIN SP ACC 1;
55          # Assign(Stackframe(Num('3')), Stack(Num('4')))
56          LOADIN SP ACC 1;
57          STOREIN BAF ACC -2;
58          LOADIN SP ACC 2;
59          STOREIN BAF ACC -3;
60          LOADIN SP ACC 3;
61          STOREIN BAF ACC -4;
62          LOADIN SP ACC 4;
63          STOREIN BAF ACC -5;
64          ADDI SP 4;
65          # Return(Empty())
66          LOADIN BAF PC -1;
67        ]
68    ]
```

Code 1.25: RETI-Blocks Pass für Array Initialisierung

1.3.4.2 Zugriff auf einen Arrayindex

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

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

Code 1.26: PicoC-Code für Zugriff auf einen Arrayindex

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

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

```
13      VoidType 'void',
14      Name 'fun',
15      [],
16      [
17        Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
          ↪  Array([Num('1'), Num('2'), Num('3')]))
18        Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19      ]
20  ]
```

Code 1.27: Abstract Syntax Tree für Zugriff auf einen Arrayindex

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

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

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

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

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

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

```
1 File
2   Name './example_array_access.picoc_mon',
3   [
4     Block
```

```
5       Name 'main.1',
6       [
7         // Assign(Name('ar'), Array([Num('42')]))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Exp(Subscr(Name('ar'), Num('0')))
11        Ref(Global(Num('0')))
12        Exp(Num('0'))
13        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
14        Exp(Stack(Num('1')))
15        Return(Empty())
16      ],
17    Block
18      Name 'fun.0',
19      [
20        // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
21        Exp(Num('1'))
22        Exp(Num('2'))
23        Exp(Num('3'))
24        Assign(Stackframe(Num('2')), Stack(Num('3')))
25        // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
26        Ref(Stackframe(Num('2')))
27        Exp(Num('1'))
28        Exp(Num('1'))
29        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
30        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
31        Exp(Stack(Num('1')))
32        Return(Empty())
33      ]
34  ]
```

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

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

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

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

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

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

1.3.4.3 Zuweisung an Arrayindex

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

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

Code 1.30: PicoC-Code für Zuweisung an Arrayindex

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

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

```
10          Assign(Subscr(Name('ar'), Num('2')), Num('42'))
11        ]
12    ]
```

Code 1.31: Abstract Syntax Tree für Zuweisung an Arrayindex

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

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

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

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays $\texttt{datatype ar}[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist, kann mittels der Formel 1.3.1:

$$\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n]) = \texttt{ref}(\texttt{ar}) + \left(\sum_{i=1}^{n}\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\right)\cdot\text{size}(\texttt{datatype}) \tag{1.3.1}$$

aus der Betriebssysteme Vorlesung[a] berechnet werden[b].

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

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

Der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` repräsentiert dabei einen ausmultiplizierten Summanden $\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\cdot\texttt{size}(\texttt{datatpye})$ in der Formel.

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** $\texttt{M}[\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n])]$ der Speicherzelle an der finalen **Adresse** $\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n])$.

[a] Scholl, „Betriebssysteme“.
[b] `ref(exp)` steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte

```
1 File
2   Name './example_array_assignment.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
```

[6] Ist in Tabelle 1.7 genauer beschrieben ist

```
8         // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
9         Exp(Num('42'))
10        Ref(Global(Num('0')))
11        Exp(Num('2'))
12        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
13        Assign(Stack(Num('1')), Stack(Num('2')))
14        Return(Empty())
15      ]
16    ]
```

Code 1.32: PicoC-Mon Pass für Zuweisung an Arrayindex

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

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

Code 1.33: RETI-Blocks Pass für Zuweisung an Arrayindex

1.3.5 Umsetzung von Structs

1.3.5.1 Deklaration und Definition von Structtypen

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

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

Code 1.34: PicoC-Code für die Deklaration eines Structtyps

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

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

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

Code 1.35: Abstract Syntax Tree für die Deklaration eines Structtyps

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 1.36 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Felder dieses Eintrags `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Feld die Attribute des **Structtyps** `[Name('len@st'),`

`Name('ar@st')]` aufgelistet, sodass man über den **Structtyp** `st` die **Attribute** des Structtyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Structtyps** zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Structtypen** ihre Attribute gleich benennen können.

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

Für die **Definition** einer Variable `st_var@main` mit diesem **Structtyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatyp`-Feld enhält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structyp** und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Die **Größe** einer Variable `st_var`, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st` $\{$`datatype`$_1$ `attr`$_1$; ... `datatype`$_n$ `attr`$_n$;$\}$;[a] definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** `datatype`$_1$ ... `datatpye`$_n$ der **Attribute** `attr`$_1$, ... `attr`$_n$ des **Structtyps**: `size(st)` $= \sum_{i=1}^{n}$ `size(datatype`$_i$`)`.

[a] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die Fragwürdige Designentscheidung, auch die eckigen Klammern `[]` für die Definition eines Arrays **vor** die Variable zu schreiben von L_C übernommen. Es wird so getann, als würde der komplette **Datentyp** immer **hinter** der Variable stehen: `datatype var`.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               IntType('int')
        name:                   Name('len@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('ar@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('24'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
        ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
        ↪  Name('ar'))])
        name:                   Name('st')
        value or address:       [Name('len@st'), Name('ar@st')]
        position:               Pos(Num('1'), Num('7'))
```

```
      size:                   Num('3')
    },
  Symbol
    {
      type qualifier:         Empty()
      datatype:               FunDecl(VoidType('void'), Name('main'), [])
      name:                   Name('main')
      value or address:       Empty()
      position:               Pos(Num('3'), Num('5'))
      size:                   Empty()
    },
  Symbol
    {
      type qualifier:         Writeable()
      datatype:               StructSpec(Name('st'))
      name:                   Name('st_var@main')
      value or address:       Num('0')
      position:               Pos(Num('4'), Num('12'))
      size:                   Num('3')
    }
]
```

Code 1.36: Symboltabelle für die Deklaration eines Structtyps

1.3.5.2 Initialisierung von Structs

Die **Initialisierung eines Structs** wird im Folgenden mithilfe des Beispiels in Code 1.37 erklärt.

```
1 struct st1 {int *attr[2];};
2
3 struct st2 {int attr1; struct st1 attr2;};
4
5 void main() {
6   int var = 42;
7   struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}};
8 }
```

Code 1.37: PicoC-Code für Initialisierung von Structs

Im **Abstract Syntax Tree** in Code 1.38 wird die **Initialisierung eines Structs** `struct st1 st = .st=.pntr=&var;` mithilfe der **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1'))), Name('st')), Struct(...))` dargestellt.

```
1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
          ↪  Name('attr'))
```

```
8       ],
9     StructDecl
10      Name 'st2',
11      [
12        Alloc(Writeable(), IntType('int'), Name('attr1'))
13        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
14      ],
15    FunDef
16      VoidType 'void',
17      Name 'main',
18      [],
19      [
20        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
21        Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')),
          ↪  Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
          ↪  Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')),
          ↪  Ref(Name('var'))])]))]))]))
22      ]
23  ]
```

Code 1.38: Abstract Syntax Tree für Initialisierung von Structs

Im Folgenden ist der Ablauf identisch zur **Initialisierung eines Arrays** in Subkapitel 1.3.4.1, daher wird um keine Reptition zu betreiben auf Subkapitel 1.3.4.1 verwiesen. Der einzige Unterschied ist, dass der im **Abstract Syntax Tree** am weitesten oben liegende **Intializer**-Knoten ein **Struct Initializer** sein muss und kein **Array Initializer**. Dafür wird in diesem Beispiel in Code 1.38 eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklärt.

Im **PicoC-Mon Pass** in Code 1.39 werden die **Logischen Ausdrücke** in den Blättern des vom **Struct-Initializer Container-Knoten** ausgehenden Baumes `Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])]))]))])` genauso, wie der **Array Initializer**, wie in Subkapitel 1.3.4.1 nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet. Im **Struct-Initializer** sind weitere **Struct- oder Array-Initializer** und **Logische Ausdrücke** immer im rechten **exp** Eintrag des Arrays Container-Kontens zu finden: `Assign(Name('attr1'), Name('var'))`.

```
1 File
2   Name './example_struct_init.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'),
          ↪  Array([Array([Ref(Name('var')), Ref(Name('var'))])]))]))])
11        Exp(Global(Num('0')))
12        Ref(Global(Num('0')))
13        Ref(Global(Num('0')))
14        Assign(Global(Num('1')), Stack(Num('2')))
15        Return(Empty())
```

```
16      ]
17  ]
```

Code 1.39: PicoC-Mon Pass für Initialisierung von Structs

```
1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'),
          ↪  Array([Array([Ref(Name('var')), Ref(Name('var'))])])]))]))
17        # Exp(Global(Num('0')))
18        SUBI SP 1;
19        LOADIN DS ACC 0;
20        STOREIN SP ACC 1;
21        # Ref(Global(Num('0')))
22        SUBI SP 1;
23        LOADI IN1 0;
24        ADD IN1 DS;
25        STOREIN SP IN1 1;
26        # Ref(Global(Num('0')))
27        SUBI SP 1;
28        LOADI IN1 0;
29        ADD IN1 DS;
30        STOREIN SP IN1 1;
31        # Assign(Global(Num('1')), Stack(Num('2')))
32        LOADIN SP ACC 1;
33        STOREIN DS ACC 2;
34        LOADIN SP ACC 2;
35        STOREIN DS ACC 1;
36        ADDI SP 2;
37        # Return(Empty())
38        LOADIN BAF PC -1;
39      ]
40  ]
```

Code 1.40: RETI-Blocks Pass für Initialisierung von Structs

1.3.5.3 Zugriff auf Structattribut

```
struct pos {int x; int y;};

void main() {
  struct pos st = {.x=4, .y=2};
  st.y;
}
```

Code 1.41: PicoC-Code für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc(Writeable(), IntType('int'), Name('x'))
        Alloc(Writeable(), IntType('int'), Name('y'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
        ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
        Exp(Attr(Name('st'), Name('y')))
      ]
  ]
```

Code 1.42: Abstract Syntax Tree für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Exp(Attr(Name('st'), Name('y')))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 1.43: PicoC-Mon Pass für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # // Exp(Attr(Name('st'), Name('y')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Attr(Stack(Num('1')), Name('y')))
        LOADIN SP IN1 1;
        ADDI IN1 1;
        STOREIN SP IN1 1;
        # Exp(Stack(Num('1')))
        LOADIN SP IN1 1;
        LOADIN IN1 ACC 0;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.44: RETI-Blocks Pass für Zugriff auf Structattribut

1.3.5.4 Zuweisung an Structattribut

```c
struct pos {int x; int y;};

void main() {
  struct pos st = {.x=4, .y=2};
  st.y = 42;
```

```
}
```

Code 1.45: PicoC-Code für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc(Writeable(), IntType('int'), Name('x'))
        Alloc(Writeable(), IntType('int'), Name('y'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
        ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
        Assign(Attr(Name('st'), Name('y')), Num('42'))
      ]
  ]
```

Code 1.46: Abstract Syntax Tree für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Assign(Attr(Name('st'), Name('y')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 1.47: PicoC-Mon Pass für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # // Assign(Attr(Name('st'), Name('y')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Attr(Stack(Num('1')), Name('y')))
        LOADIN SP IN1 1;
        ADDI IN1 1;
        STOREIN SP IN1 1;
        # Assign(Stack(Num('1')), Stack(Num('2')))
        LOADIN SP IN1 1;
        LOADIN SP ACC 2;
        ADDI SP 2;
        STOREIN IN1 ACC 0;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.48: RETI-Blocks Pass für Zuweisung an Structattribut

1.3.6 Umsetzung der Derived Datatypes im Zusammenspiel

1.3.6.1 Anfangsteil für Globale Statische Daten und Stackframe

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

Code 1.49: PicoC-Code für den Anfangsteil

```
1  File
2    Name './example_derived_dts_introduction_part.ast',
3    [
4      StructDecl
5        Name 'ar_with_len',
6        [
7          Alloc(Writeable(), IntType('int'), Name('len'))
8          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           ↪  Name('st_ar')))
16         Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
           ↪  IntType('int')))), Name('pntr2')))
17         Exp(Name('pntr2'))
18       ],
19     FunDef
20       VoidType 'void',
21       Name 'fun',
22       [],
23       [
24         Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
           ↪  Name('st_ar')))
25         Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
           ↪  Name('pntr1')))
26         Exp(Name('pntr1'))
27       ]
28   ]
```

Code 1.50: Abstract Syntax Tree für den Anfangsteil

```
File
  Name './example_derived_dts_introduction_part.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
        ↪  IntType('int')))), Name('pntr2')))
        // Exp(Name('pntr2'))
        Exp(Global(Num('9')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('pntr1')))
        // Exp(Name('pntr1'))
        Exp(Stackframe(Num('9')))
        Return(Empty())
      ]
  ]
```

Code 1.51: PicoC-Mon Pass für den Anfangsteil

```
File
  Name './example_derived_dts_introduction_part.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
        ↪  PntrDecl(Num('1'), IntType('int')))), Name('pntr2')))
        # // Exp(Name('pntr2'))
        # Exp(Global(Num('9')))
        SUBI SP 1;
        LOADIN DS ACC 9;
        STOREIN SP ACC 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
```

```
20        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
21        # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
          ↪  IntType('int'))), Name('pntr1')))
22        # // Exp(Name('pntr1'))
23        # Exp(Stackframe(Num('9')))
24        SUBI SP 1;
25        LOADIN BAF ACC -11;
26        STOREIN SP ACC 1;
27        # Return(Empty())
28        LOADIN BAF PC -1;
29      ]
30  ]
```

Code 1.52: RETI-Blocks Pass für den Anfangsteil

1.3.6.2 Mittelteil für die verschiedenen Derived Datatypes

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

Code 1.53: PicoC-Code für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
          ↪  Name('ar'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14        Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
          ↪  Array([Num('42')]))
15        Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st_first')),
          ↪  Struct([Assign(Name('ar'), Ref(Name('var')))]))
16        Exp(Subscr(Deref(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
17      ]
18  ]
```

Code 1.54: Abstract Syntax Tree für den Mittelteil

```
File
  Name './example_derived_dts_main_part.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
        Ref(Global(Num('1')))
        Ref(Attr(Stack(Num('1')), Name('ar')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 1.55: PicoC-Mon Pass für den Mittelteil

```
File
  Name './example_derived_dts_main_part.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('st_first'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # // Exp(Subscr(Subscr(Attr(Name('st_first'), Name('ar')), Num('0')), Num('0')))
        # Ref(Global(Num('1')))
```

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

Code 1.56: RETI-Blocks Pass für den Mittelteil

1.3.6.3 Schlussteil für die verschiedenen Derived Datatypes

```c
struct st {int attr[2];};

void main() {
  int ar1[1][2] = {{42, 314}};
  struct st ar2[1] = {.attr={42, 314}};
  int var = 42;
  int *pntr1 = &var;
  int **pntr2 = &pntr1;

  ar1[0];
  ar2[0];
```

```
12   *pntr2;
13 }
```

Code 1.57: PicoC-Code für den Schlussteil

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

Code 1.58: Abstract Syntax Tree für den Schlussteil

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

```
16        Exp(Num('42'))
17        Assign(Global(Num('4')), Stack(Num('1')))
18        // Assign(Name('pntr1'), Ref(Name('var')))
19        Ref(Global(Num('4')))
20        Assign(Global(Num('5')), Stack(Num('1')))
21        // Assign(Name('pntr2'), Ref(Name('pntr1')))
22        Ref(Global(Num('5')))
23        Assign(Global(Num('6')), Stack(Num('1')))
24        // Exp(Subscr(Name('ar1'), Num('0')))
25        Ref(Global(Num('0')))
26        Exp(Num('0'))
27        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
28        Exp(Stack(Num('1')))
29        // Exp(Subscr(Name('ar2'), Num('0')))
30        Ref(Global(Num('2')))
31        Exp(Num('0'))
32        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
33        Exp(Stack(Num('1')))
34        // Exp(Subscr(Name('pntr2'), Num('0')))
35        Ref(Global(Num('6')))
36        Exp(Num('0'))
37        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
38        Exp(Stack(Num('1')))
39        Return(Empty())
40      ]
41  ]
```

Code 1.59: PicoC-Mon Pass für den Schlussteil

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

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

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

Code 1.60: RETI-Blocks Pass für den Schlussteil

1.3.7 Umsetzung von Funktionen

1.3.7.1 Funktionen auflösen zu RETI Code

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

Code 1.61: PicoC-Code für 3 Funktionen

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

Code 1.62: Abstract Syntax Tree für 3 Funktionen

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

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

Code 1.63: RETI-Blocks Pass für 3 Funktionen

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

Code 1.64: PicoC-Mon Pass für 3 Funktionen

```
File
  Name './example_3_funs.reti_blocks',
  [
    Block
      Name 'main.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun1.1',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.0',
      [
        # // Return(Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.65: RETI-Blocks Pass für 3 Funktionen

1.3.7.1.1 Sprung zur Main Funktion

```
void fun1() {
}

int fun2() {
  return 1;
}

void main() {
  return;
}
```

Code 1.66: PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
File
  Name './example_3_funs_main.picoc_mon',
```

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

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

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

Code 1.68: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

```
File
  Name './example_3_funs_main.reti_patch',
  [
    Block
      Name 'start.3',
      [
        # // Exp(GoTo(Name('main.0')))
        Exp(GoTo(Name('main.0')))
      ],
    Block
      Name 'fun1.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.1',
      [
        # // Return(Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.69: PicoC-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

1.3.7.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

```
int fun2(int var);

void fun1() {
}

void main() {
  int var = fun2(42);
  return;
}

int fun2(int var) {
  return var;
}
```

Code 1.70: PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss

Bei mehreren Funktionen werden die **Scopes** der unterschiedlichen **Funktionen** mittels eines **Suffix** `"<fun_name>@"` umgesetzt, der an den **Variablennamen** `<var>` drangehängt wird: `<var>@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**[7] Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** nach dem **Depth-First-Search** Schema über den

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
        ↪  IntType('int'), Name('var'))])
        name:                   Name('fun2')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('4'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
        name:                   Name('fun1')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('6'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('var@main')
        value or address:       Num('0')
        position:               Pos(Num('7'), Num('6'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('var@fun2')
        value or address:       Num('0')
        position:               Pos(Num('11'), Num('13'))
```

[7] D.h. von der Wurzel zu den Blättern eines Baumes

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

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

1.3.7.3 Funktionsaufruf

1.3.7.3.1 Ohne Rückgabewert

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

Code 1.72: PicoC-Code für Funktionsaufruf ohne Rückgabewert

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

Code 1.73: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

```
File
  Name './example_fun_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        # // Exp(Call(Name('stack_fun'), [Name('local_var')]))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 3;
        MOVE SP BAF;
        SUBI SP 4;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('stack_fun.0')))
        Exp(GoTo(Name('stack_fun.0')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'stack_fun.0',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('st'))),
        ↪  Name('param')))
        # // Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.74: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

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

```
# // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
↪  Name('local_var')))
# // Exp(Call(Name('stack_fun'), [Name('local_var')]))
# StackMalloc(Num('2'))
SUBI SP 2;
# Ref(Global(Num('0')))
SUBI SP 1;
LOADI IN1 0;
ADD IN1 DS;
STOREIN SP IN1 1;
# NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
MOVE BAF ACC;
ADDI SP 3;
MOVE SP BAF;
SUBI SP 4;
STOREIN BAF ACC 0;
LOADI ACC 14;
ADD ACC CS;
STOREIN BAF ACC -1;
# Exp(GoTo(Name('stack_fun.0')))
JUMP 5;
# RemoveStackframe()
MOVE BAF IN1;
LOADIN IN1 BAF 0;
MOVE IN1 SP;
# Return(Empty())
LOADIN BAF PC -1;
# // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('st'))), Name('param')))
# // Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
# Return(Empty())
LOADIN BAF PC -1;
```

Code 1.75: RETI-Pass für Funktionsaufruf ohne Rückgabewert

1.3.7.3.2 Mit Rückgabewert

```
void stack_fun() {
  return 42;
}

void main() {
  int var = stack_fun();
}
```

Code 1.76: PicoC-Code für Funktionsaufruf mit Rückgabewert

```
File
  Name './example_fun_call_with_return_value.picoc_mon',
  [
    Block
      Name 'stack_fun.1',
```

```
      [
        // Return(Num('42'))
        Exp(Num('42'))
        Return(Stack(Num('1')))
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Call(Name('stack_fun'), []))
        StackMalloc(Num('2'))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.1')))
        RemoveStackframe()
        Assign(Global(Num('0')), Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 1.77: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
        # // Return(Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Call(Name('stack_fun'), []))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 2;
        MOVE SP BAF;
        SUBI SP 2;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('stack_fun.1')))
        Exp(GoTo(Name('stack_fun.1')))
        # RemoveStackframe()
```

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

Code 1.78: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

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

Code 1.79: RETI-Pass für Funktionsaufruf mit Rückgabewert

1.3.7.3.3 Umsetzung von Call by Sharing für Arrays

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

Code 1.80: PicoC-Code für Call by Sharing für Arrays

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

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

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

```
6         datatype:               FunDecl(VoidType('void'), Name('stack_fun'),
          ↪  [Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
          ↪  Name('param1')), Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')),
          ↪  Name('param2'))])
7         name:                   Name('stack_fun')
8         value or address:       Empty()
9         position:               Pos(Num('1'), Num('5'))
10        size:                   Empty()
11      },
12    Symbol
13      {
14        type qualifier:         Writeable()
15        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16        name:                   Name('param1@stack_fun')
17        value or address:       Num('0')
18        position:               Pos(Num('1'), Num('21'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         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 1.82: Symboltabelle für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
        # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
        ↪  IntType('int'))), Name('param1')))
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('param2')))
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
        ↪  Name('local_var1')))
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], IntType('int')),
        ↪  Name('local_var2')))
        # // Exp(Call(Name('stack_fun'), [Name('local_var1'), Name('local_var2')]))
        # StackMalloc(Num('2'))
        SUBI SP 2;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Ref(Global(Num('6')))
        SUBI SP 1;
        LOADI IN1 6;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        MOVE BAF ACC;
        ADDI SP 4;
        MOVE SP BAF;
        SUBI SP 4;
        STOREIN BAF ACC 0;
        LOADI ACC GoTo(Name('addr@next_instr'));
        ADD ACC CS;
        STOREIN BAF ACC -1;
        # Exp(GoTo(Name('stack_fun.1')))
        Exp(GoTo(Name('stack_fun.1')))
        # RemoveStackframe()
        MOVE BAF IN1;
        LOADIN IN1 BAF 0;
        MOVE IN1 SP;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 1.83: RETI-Block Pass für Call by Sharing für Arrays

1.3.7.3.4 Umsetzung von Call by Value für Structs

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

Code 1.84: PicoC-Code für Call by Value für Structs

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

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

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

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

Code 1.86: RETI-Block Pass für Call by Value für Structs

1.3.8 Umsetzung kleinerer Details

1.4 Fehlermeldungen

1.4.1 Error Handler

1.4.2 Arten von Fehlermeldungen

1.4.2.1 Syntaxfehler

1.4.2.2 Laufzeitfehler

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 07. 2022).

Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (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).