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** (Definition 1.2) auf den Stack schreiben soll, die über `exp` eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` im **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Deref(lhs, exp)`	Container für den **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)`	Container für ein **If Statement** `if(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`IfElse(exp, stmts1, stmts2)`	Container für ein **If-Else Statement** `if(<exp>) { <stmts2> } else { <stmts2> }` inklusive **Codition** `exp` und 2 **Branches** `stmts1` und `stmts2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Statements** oder `GoTo(Name('block.xyz'))`'s stehen können.
`While(exp, stmts)`	Container für ein **While-Statement** `while(<exp>) { <stmts> }` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`DoWhile(exp, stmts)`	Container für ein **Do-While-Statement** `do { <stmts> } while(<exp>);` inklusive **Condition** `exp` und einem **Branch** `stmts`, indem eine **Liste von Statements** stehen kann oder ein einzelnes `GoTo(Name('block.xyz'))`.
`Call(name, exps)`	Container für einen **Funktionsaufruf**: `fun_name(exps)`, wobei `name` der **Bezeichner** der Funktion ist, die aufgerufen werden soll und `exps` eine **Liste von Argumenten** ist, die an die Funktion übergeben werden soll.
`Return(exp)`	Container für ein `Return-Statement`: `return <exp>`, wobei das **Attribut** `exp` einen **Logischen Ausdruck** darstellt, dessen Ergebnis vom **Return-Statement** zurückgegeben wird.
`FunDecl(datatype, name, allocs)`	Container für eine **Funktionsdeklaration**, z.B. `<datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>)`, wobei `datatype` der **Rückgabewert** der Funktion ist, `name` der **Bezeichner** der Funktion ist und `allocs` die **Parameter** der Funktion sind, wobei der **Container-Knoten** `Alloc(type_spec, datatype, name)` als 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** und Speicherung des Werts des `BAF`-Registers der **aufrufenden Funktion** und der **Rücksprungadresse** nacheinander an den **Anfang** des neuen **Stackframes**. Das Attribut `fun_name` stehte dabei für den Bezeichner der Funktion, für die ein neuer **Stackframe** erstellt werden soll. Das Attribut `fun_name` dient später dazu den **Block** dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attributen gespeichert hat. Des Weiteren ist das Attribut `goto_after_call` ein `GoTo(Name('addr@next_instr'))`, welches später durch die **Adresse** des Befehls, der direkt auf die **Jump Instruction** folgt, ersetzt wird.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes** durch das **Wiederherstellen** des im noch **aktuellen Stackframe** gespeicherten Werts des `BAF`-Registes der **aufrufenden Funktion** und das Setzen des `SP`-Registers auf den Wert des `BAF`-Registesr **vor** der Wiederherstellung.
`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** (Definition 1.1) des Blocks ist und `stmts_instrs` eine **Liste von Statements** oder **Instructions**. Zudem besitzt er noch 3 versteckte Attribute, wobei `instrs_before` die Zahl der **Instructions** vor diesem **Block** zählt, `num_instrs` die Zahl der Instructions ohne Kommentare in diesem Block zählt, `param_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **Parameter** der Funktion belegt werden müssen und `local_vars_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **lokalen Variablen** der Funktion belegt werden müssen.
`GoTo(name)`	Container für ein `Goto` zu einem anderen **Block**, wobei das Attribut `name` der Bezeichner des **Labels** des Blocks ist zu dem Gesprungen werden soll.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar**, den der Compiler selber während des **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

Definition 1.1: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a] **tab:picoc˙knoten˙teil˙4**.

Definition 1.2: Location

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

*Im Grunde genommen alles, was mit einem **Programm zu tuen** hat und irgendwo **gespeichert** ist.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Die ausgegrauten Attribute der PicoC-Nodes sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Nodes** mit einem Wert **initialisiert** werden, sondern im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen bekommen, dass im weiteren 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 1.3) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.4) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

Definition 1.3: Token-Knoten

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

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

[a] Thiemann, „Compilerbau“.

Definition 1.4: Container-Knoten

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

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

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

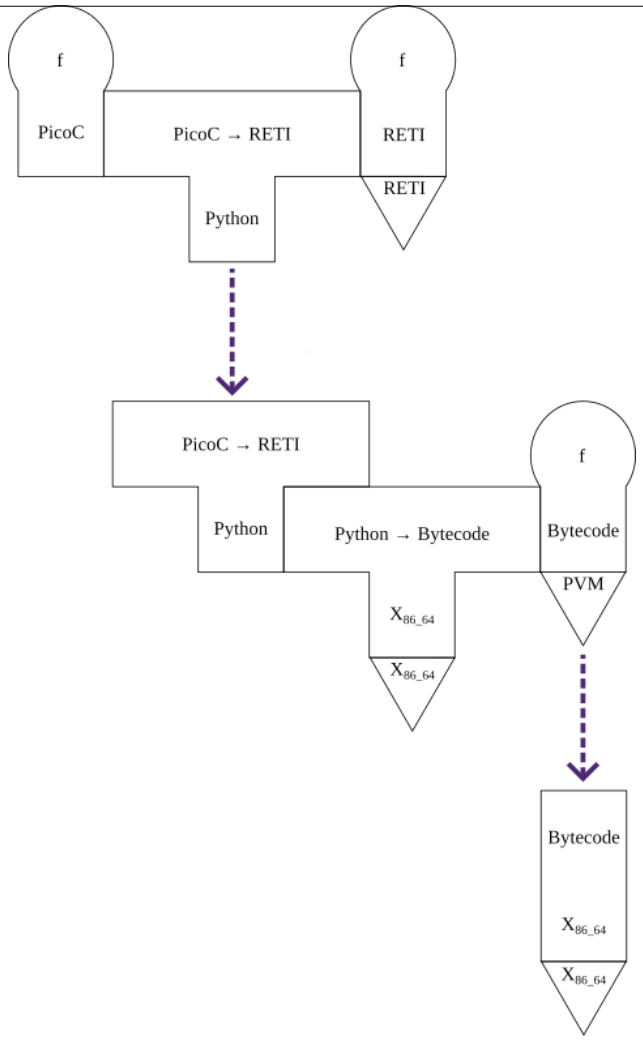

Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

Dieses längliche **T-Diagram** in Abbildung 1.1 lässt sich zusammenfassen, sodass man das **T-Diagram** in Abbildung 1.2 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in $\mathtt{X}_{86\_64}$-Maschienensprache geschrieben ist.

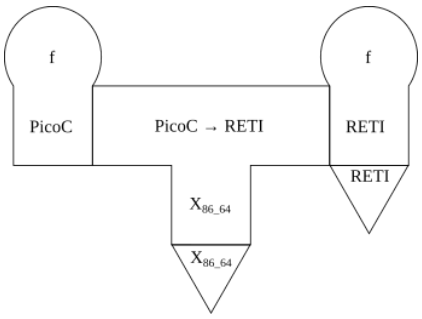

Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

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

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

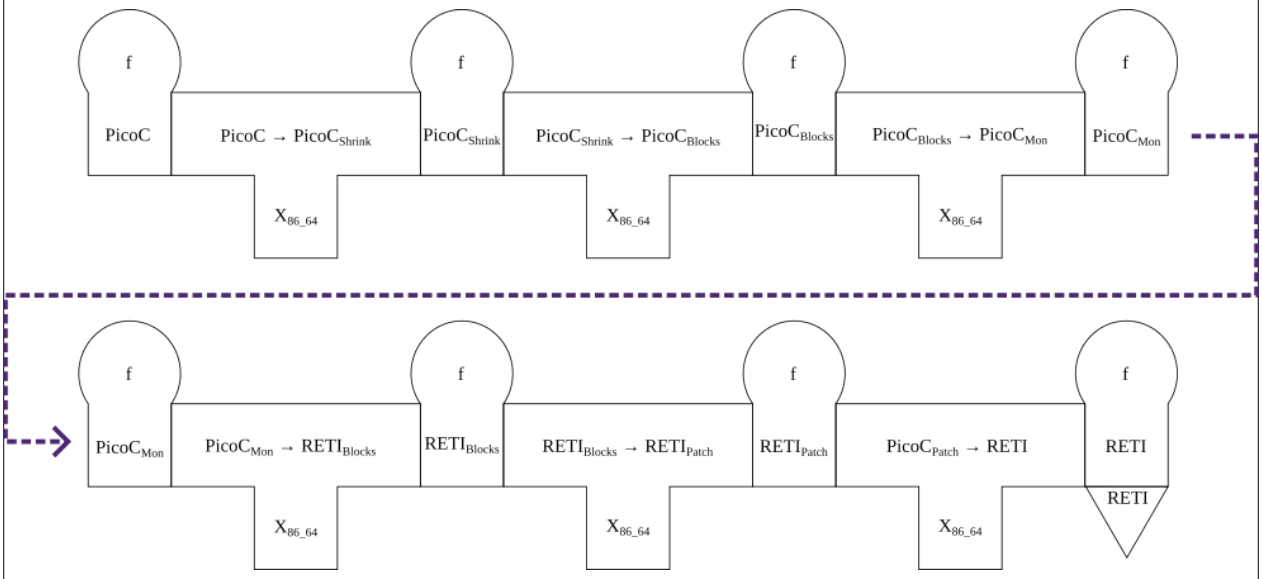

Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

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

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

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird[2]. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[3] finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[4].

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

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

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

1.3.2 Passes

1.3.2.1 PicoC-Shrink Pass

1.3.2.1.1 Zweck

1.3.2.1.2 Codebeispiel

```
// based on a example program from Christoph Scholl's Operating Systems lecture

void main() {
  int n = 4;
  int res = 1;
  while (1) {
    if (n == 1) {
      return;
    }
    res = n * res;
    n = n - 1;
  }
}
```

Code 1.5: PicoC Code für Codebespiel

```
File
  Name './example_faculty_it.ast',
  [
    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 1.6: 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 Zweck

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

1.3.2.2.2 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}

1.3.2.2.3 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'))
              ]
          ],
        Block
          Name 'if.2',
          [
            Return(Empty())
          ],
        Block
          Name 'if_else_after.1',
          [
            Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
            Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
            GoTo(Name('condition_check.4'))
          ],
        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 Zweck

1.3.2.3.2 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 1.5: Symboltabelle

1.3.2.3.3 Codebeispiel

```
1  File
2    Name './example_faculty_it.picoc_mon',
3    [
4      Block
5        Name 'main.5',
6        [
7          // Assign(Name('n'), Num('4'))
8          Exp(Num('4'))
9          Assign(Global(Num('0')), Stack(Num('1')))
10         // Assign(Name('res'), Num('1'))
11         Exp(Num('1'))
12         Assign(Global(Num('1')), Stack(Num('1')))
```

```
13        // While(Num('1'), [])
14        Exp(GoTo(Name('condition_check.4')))
15      ],
16    Block
17      Name 'condition_check.4',
18      [
19        // IfElse(Num('1'), [], [])
20        Exp(Num('1')),
21        IfElse
22          Stack
23            Num '1',
24          [
25            GoTo(Name('while_branch.3'))
26          ],
27          [
28            GoTo(Name('while_after.0'))
29          ]
30      ],
31    Block
32      Name 'while_branch.3',
33      [
34        // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35        // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36        Exp(Global(Num('0')))
37        Exp(Num('1'))
38        Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
39        IfElse
40          Stack
41            Num '1',
42          [
43            GoTo(Name('if.2'))
44          ],
45          [
46            GoTo(Name('if_else_after.1'))
47          ]
48      ],
49    Block
50      Name 'if.2',
51      [
52        Return(Empty())
53      ],
54    Block
55      Name 'if_else_after.1',
56      [
57        // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
58        Exp(Global(Num('0')))
59        Exp(Global(Num('1')))
60        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
61        Assign(Global(Num('1')), Stack(Num('1')))
62        // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
63        Exp(Global(Num('0')))
64        Exp(Num('1'))
65        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
66        Assign(Global(Num('0')), Stack(Num('1')))
67        Exp(GoTo(Name('condition_check.4')))
68      ],
69    Block
```

```
70      Name 'while_after.0',
71      [
72        Return(Empty())
73      ]
74  ]
```

Code 1.9: PicoC-Mon Pass für Codebespiel

1.3.2.4 RETI-Blocks Pass

1.3.2.4.1 Zweck

1.3.2.4.2 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}

1.3.2.4.3 Codebeispiel

```
1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'main.5',
6       [
7         # // Assign(Name('n'), Num('4'))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('res'), Num('1'))
17        # Exp(Num('1'))
18        SUBI SP 1;
19        LOADI ACC 1;
20        STOREIN SP ACC 1;
21        # Assign(Global(Num('1')), Stack(Num('1')))
22        LOADIN SP ACC 1;
23        STOREIN DS ACC 1;
24        ADDI SP 1;
25        # // While(Num('1'), [])
26        # Exp(GoTo(Name('condition_check.4')))
27        Exp(GoTo(Name('condition_check.4')))
```

```
28      ],
29    Block
30      Name 'condition_check.4',
31      [
32        # // IfElse(Num('1'), [], [])
33        # Exp(Num('1'))
34        SUBI SP 1;
35        LOADI ACC 1;
36        STOREIN SP ACC 1;
37        # IfElse(Stack(Num('1')), [], [])
38        LOADIN SP ACC 1;
39        ADDI SP 1;
40        JUMP== GoTo(Name('while_after.0'));
41        Exp(GoTo(Name('while_branch.3')))
42      ],
43    Block
44      Name 'while_branch.3',
45      [
46        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
47        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
48        # Exp(Global(Num('0')))
49        SUBI SP 1;
50        LOADIN DS ACC 0;
51        STOREIN SP ACC 1;
52        # Exp(Num('1'))
53        SUBI SP 1;
54        LOADI ACC 1;
55        STOREIN SP ACC 1;
56        LOADIN SP ACC 2;
57        LOADIN SP IN2 1;
58        SUB ACC IN2;
59        JUMP== 3;
60        LOADI ACC 0;
61        JUMP 2;
62        LOADI ACC 1;
63        STOREIN SP ACC 2;
64        ADDI SP 1;
65        # IfElse(Stack(Num('1')), [], [])
66        LOADIN SP ACC 1;
67        ADDI SP 1;
68        JUMP== GoTo(Name('if_else_after.1'));
69        Exp(GoTo(Name('if.2')))
70      ],
71    Block
72      Name 'if.2',
73      [
74        # Return(Empty())
75        LOADIN BAF PC -1;
76      ],
77    Block
78      Name 'if_else_after.1',
79      [
80        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
81        # Exp(Global(Num('0')))
82        SUBI SP 1;
83        LOADIN DS ACC 0;
84        STOREIN SP ACC 1;
```

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

Code 1.10: RETI-Blocks Pass für Codebespiel

1.3.2.5 RETI-Patch Pass

1.3.2.5.1 Zweck

1.3.2.5.2 Abstrakte Syntax

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

Grammar 1.3.4: Abstrakte Syntax für L_{RETI_Patch}

1.3.2.5.3 Codebeispiel

```
File
  Name './example_faculty_it.reti_patch',
  [
    Block
      Name 'start.6',
      [
        # // Exp(GoTo(Name('main.5')))
        # // not included 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')))
        # // not included 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;
        JUMP== GoTo(Name('while_after.0'));
        # // not included Exp(GoTo(Name('while_branch.3')))
      ],
    Block
      Name 'while_branch.3',
      [
        # // If(Atom(Name('n'), Eq('==')), Num('1')), [])
        # // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
        # Exp(Global(Num('0')))
        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        LOADIN SP ACC 2;
63        LOADIN SP IN2 1;
64        SUB ACC IN2;
65        JUMP== 3;
66        LOADI ACC 0;
67        JUMP 2;
68        LOADI ACC 1;
69        STOREIN SP ACC 2;
70        ADDI SP 1;
71        # IfElse(Stack(Num('1')), [], [])
72        LOADIN SP ACC 1;
73        ADDI SP 1;
74        JUMP== GoTo(Name('if_else_after.1'));
75        # // not included Exp(GoTo(Name('if.2')))
76      ],
77    Block
78      Name 'if.2',
79      [
80        # Return(Empty())
81        LOADIN BAF PC -1;
82      ],
83    Block
84      Name 'if_else_after.1',
85      [
86        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
87        # Exp(Global(Num('0')))
88        SUBI SP 1;
89        LOADIN DS ACC 0;
90        STOREIN SP ACC 1;
91        # Exp(Global(Num('1')))
92        SUBI SP 1;
93        LOADIN DS ACC 1;
94        STOREIN SP ACC 1;
95        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
96        LOADIN SP ACC 2;
97        LOADIN SP IN2 1;
98        MULT ACC IN2;
99        STOREIN SP ACC 2;
100       ADDI SP 1;
101       # Assign(Global(Num('1')), Stack(Num('1')))
102       LOADIN SP ACC 1;
103       STOREIN DS ACC 1;
104       ADDI SP 1;
105       # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
106       # Exp(Global(Num('0')))
107       SUBI SP 1;
108       LOADIN DS ACC 0;
109       STOREIN SP ACC 1;
110       # Exp(Num('1'))
111       SUBI SP 1;
112       LOADI ACC 1;
```

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

Code 1.11: RETI-Patch Pass für Codebespiel

1.3.2.6 RETI Pass

1.3.2.6.1 Zweck

1.3.2.6.2 Konkrette und Abstrakte Syntax

dig_no_0	::=	”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”	*L_Program*
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}

1.3.2.6.3 Codebeispiel

```
1  # // Exp(GoTo(Name('main.5')))
2  # // not included Exp(GoTo(Name('main.5')))
3  # // Assign(Name('n'), Num('4'))
4  # Exp(Num('4'))
5  SUBI SP 1;
6  LOADI ACC 4;
7  STOREIN SP ACC 1;
8  # Assign(Global(Num('0')), Stack(Num('1')))
9  LOADIN SP ACC 1;
10 STOREIN DS ACC 0;
11 ADDI SP 1;
12 # // Assign(Name('res'), Num('1'))
13 # Exp(Num('1'))
14 SUBI SP 1;
15 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 # // not included Exp(GoTo(Name('condition_check.4')))
24 # // IfElse(Num('1'), [], [])
25 # Exp(Num('1'))
26 SUBI SP 1;
27 LOADI ACC 1;
28 STOREIN SP ACC 1;
29 # IfElse(Stack(Num('1')), [], [])
30 LOADIN SP ACC 1;
31 ADDI SP 1;
32 JUMP== 49;
33 # // not included Exp(GoTo(Name('while_branch.3')))
34 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
35 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
36 # Exp(Global(Num('0')))
37 SUBI SP 1;
38 LOADIN DS ACC 0;
39 STOREIN SP ACC 1;
40 # Exp(Num('1'))
41 SUBI SP 1;
42 LOADI ACC 1;
43 STOREIN SP ACC 1;
44 LOADIN SP ACC 2;
45 LOADIN SP IN2 1;
46 SUB ACC IN2;
47 JUMP== 3;
48 LOADI ACC 0;
49 JUMP 2;
50 LOADI ACC 1;
51 STOREIN SP ACC 2;
52 ADDI SP 1;
53 # IfElse(Stack(Num('1')), [], [])
54 LOADIN SP ACC 1;
55 ADDI SP 1;
56 JUMP== 2;
57 # // not included Exp(GoTo(Name('if.2')))
58 # Return(Empty())
59 LOADIN BAF PC -1;
60 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
61 # Exp(Global(Num('0')))
62 SUBI SP 1;
63 LOADIN DS ACC 0;
64 STOREIN SP ACC 1;
65 # Exp(Global(Num('1')))
66 SUBI SP 1;
67 LOADIN DS ACC 1;
68 STOREIN SP ACC 1;
69 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
70 LOADIN SP ACC 2;
71 LOADIN SP IN2 1;
72 MULT ACC IN2;
```

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

Code 1.12: RETI Pass für Codebespiel

1.3.3 Umsetzung von Funktionen

1.3.3.1 Mehrere Funktionen

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

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

Code 1.13: PicoC-Code für 3 Funktionen

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

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

Code 1.14: Abstract Syntax Tree für 3 Funktionen

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

Es werden parallel dazu neue Zuordnungen im **Dictionary** `fun_name_to_block_name` hinzugefügt. Das **Dicionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu[7]. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```
File
 Name './verbose_3_funs.picoc_blocks',
 [
   FunDef
     VoidType 'void',
     Name 'main',
     [],
     [
       Block
         Name 'main.2',
         [
           Return(Empty())
         ]
     ],
   FunDef
     VoidType 'void',
     Name 'fun1',
     [],
     [
       Block
         Name 'fun1.1',
         []
     ],
   FunDef
     IntType 'int',
     Name 'fun2',
     [],
     [
       Block
         Name 'fun2.0',
         [
           Return(Num('1'))
         ]
     ]
 ]
```

[5] Eine Erklärung dazu ist in Unterkapitel 1.3.2.2.1 zu finden.
[6] Der **Grund** dafür kann im Unterkapitel 1.3.2.2.1 nachgelesen werden.
[7] Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 1.15: PicoC-Blocks Pass für 3 Funktionen

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

```
File
  Name './verbose_3_funs.picoc_mon',
  [
    Block
      Name 'main.2',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 1.16: PicoC-Mon Pass für 3 Funktionen

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

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

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.2')))
# // not included Exp(GoTo(Name('main.2')))
# // 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 1.17: RETI-Blocks Pass für 3 Funktionen

1.3.3.1.1 Sprung zur Main Funktion

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

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

```
void fun1() {
}

int fun2() {
  return 1;
}

void main() {
  return;
}
```

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

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

```
File
  Name './verbose_3_funs_main.reti_blocks',
  [
    Block
      Name 'fun1.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun2.1',
```

```
12      [
13        # // Return(Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Return(Stack(Num('1')))
19        LOADIN SP ACC 1;
20        ADDI SP 1;
21        LOADIN BAF PC -1;
22      ],
23    Block
24      Name 'main.0',
25      [
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 1.19: RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

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

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

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

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

```
1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'fun1.2',
12      [
13        # Return(Empty())
14        LOADIN BAF PC -1;
```

```
      ],
    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 1.20: RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

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

```
# // Block(Name('start.3'), [])
# // Exp(GoTo(Name('main.0')))
JUMP 8;
# // 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 1.21: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

1.3.3.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

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

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

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

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

Definition 1.6: Funktionsprototyp

***Deklaration** einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parametereihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktiondeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.*[a] [b]

[a] Der **Funktionsprototyp** ist von der **Funktionsignatur** zu unterschieden, die in Programmiersprache wie `C++` und `Java` für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentpyen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentpyen** usw. unterschieden.

[b] *What is the difference between function prototype and function signature?*

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

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

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

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

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

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

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

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

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

Definition 1.7: Scope (bzw. Sichtbarkeitsbereich)

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

[a] Thiemann, „Einführung in die Programmierung“.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
           ↪  IntType('int'), Name('var'))])
7          name:                   Name('fun2')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('4'))
10         size:                   Empty()
11       },
12     Symbol
13       {
14         type qualifier:         Empty()
15         datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
16         name:                   Name('fun1')
17         value or address:       Empty()
18         position:               Pos(Num('3'), Num('5'))
19         size:                   Empty()
20       },
21     Symbol
22       {
23         type qualifier:         Empty()
24         datatype:               FunDecl(VoidType('void'), Name('main'), [])
```

```
25        name:                   Name('main')
26        value or address:       Empty()
27        position:               Pos(Num('6'), Num('5'))
28        size:                   Empty()
29      },
30    Symbol
31      {
32        type qualifier:         Writeable()
33        datatype:               IntType('int')
34        name:                   Name('var@main')
35        value or address:       Num('0')
36        position:               Pos(Num('7'), Num('6'))
37        size:                   Num('1')
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               IntType('int')
43        name:                   Name('var@fun2')
44        value or address:       Num('0')
45        position:               Pos(Num('12'), Num('13'))
46        size:                   Num('1')
47      }
48  ]
```

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

1.3.3.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 1.24 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und dieses Kapitel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 1.3.3.3.1 erklärt.

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

Code 1.24: PicoC-Code für Funktionsaufruf ohne Rückgabewert

Im **Abstract Syntax Tree** in Code 1.25 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```
File
  Name './example_fun_call_no_return_value.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
      ],
    FunDecl
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '2',
              Num '3'
            ],
            StructSpec
              Name 'st',
          Name 'param'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('local_var')))
        Exp(Call(Name('stack_fun'), [Name('local_var')]))
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'stack_fun',
      [
        Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
        ↪  Name('param'))
      ],
      [
        Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
      ]
  ]
```

Code 1.25: Abstract Syntax Tree für Funktionsaufruf ohne Rückgabewert

Im **PicoC-Mon Pass** in Code 1.26 wird die Komposition und Container-Knoten `Exp(Call(Name('stack_fun'), [Name('local_var')]))` durch die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` ersetzt, welche in den Tabellen 1.7 und 1.2 genauer erklärt sind.

```
File
  Name './example_fun_call_no_return_value.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.0')))
        RemoveStackframe()
        Return(Empty())
      ],
    Block
      Name 'stack_fun.0',
      [
        Return(Empty())
      ]
  ]
```

Code 1.26: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI-Blocks Pass** in Code 1.27 werden die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun')`, `GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch ihre entsprechenden **RETI-Knoten** ersetzt.

Unter den **RETI-Knoten** entsprechen die **Kompostionen** `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` noch keine fertigen **RETI-Instructions** und werden später in dem für sie vorgesehenen **RETI-Pass** passend ergänzt bzw. ersetzt.

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Dictionary** `fun_name_to_block_name`[9] mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Container-Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```
File
  Name './example_fun_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # 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;
```

[9] Dieses Dictionary wurde in Unterkapitel 1.3.3.1 eingeführt.

```
19          STOREIN BAF ACC 0;
20          LOADI ACC GoTo(Name('addr@next_instr'));
21          ADD ACC CS;
22          STOREIN BAF ACC -1;
23          # Exp(GoTo(Name('stack_fun.0')))
24          Exp(GoTo(Name('stack_fun.0')))
25          # RemoveStackframe()
26          MOVE BAF IN1;
27          LOADIN IN1 BAF 0;
28          MOVE IN1 SP;
29          # Return(Empty())
30          LOADIN BAF PC -1;
31        ],
32      Block
33        Name 'stack_fun.0',
34        [
35          # Return(Empty())
36          LOADIN BAF PC -1;
37        ]
38    ]
```

Code 1.27: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

Im **RETI Pass** in Code 1.27 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist[10] ersetzt: `LOADI ACC 14`. Und auch der **Container-Knoten**, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den **Container-Knoten** `JUMP 5` ersetzt.

Die **Distanz** `5` im **RETI-Knoten** `JUMP 5` wird mithilfe des `instrs_before`-Attribute des **Zielblocks**, der den ersten Befehl der gewünschten Funktion enthält und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` enthalten ist berechnet.

Die **relative Adresse** `14` direkt nach dem Befehl `JUMP 5` wird ebenfalls mithilfe des `instrs_before`-Attributs des **aktuellen Blocks** berechnet. Es handelt sich bei bei 14 um eine **relative Adresse**, die **relativ** zum `CS`-Register berechnet wird, welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

Die Berechnung der **Adresse** `'<addr@next_instr>'` (bzw. in der Formel adr_{danach}) des Befehls nach dem **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt dabei mithilfe der folgenden Formel:

$$adr_{danach} = \#Bef_{vor\,akt.\,Bl.} + idx + 4 \tag{1.3.1}$$

wobei:

- es sich bei bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum `CS`-Register berechnet wird.
- $\#Bef_{vor\,akt.\,Bl.} \hat{=}$ **Anzahl** Befehle vor dem momentanen Block. Es handelt sich hierbei um ein versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before,`

[10] Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

num_instrs, param_size, local_vars_size), welches im **RETI-Patch**-Pass gesetzt wird. Der Grund dafür, dass das Zuweisen dieses versteckten Attributes `instrs_before` im **RETI-Patch** Pass erfolgt ist, weil erst im **RETI-Patch** Pass die **finale Anzahl** an Befehlen in einem Block feststeht, da im **RETI-Patch** Pass `GoTo()`'s entfernt werden, deren Sprung nur **eine** Adresse weiterspringen würde. Die **finale Anzahl** an Befehlen kann sich in diesem **Pass** also noch ändern und steht erst nach diesem **Pass** fest.

- $idx \hat{=}$ relativer Index des Befehls `LOADI ACC <addr@next_instr>` selbst im Block.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 1.28 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

Die Berechnug der **Distanz** `<distanz>` für den Sprung `JUMP <distanz>` zum **ersten** Befehl eines im **Pass** zuvor **existenten Blockes** erfolgt dabei nach der folgenden Formel:

$$distanz = \begin{cases} -\#Bef_{vor\ akt.\ Bl.} + \#Bef_{vor\ Zielbl.} - idx & \#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \\ \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.} \end{cases} \quad (1.3.2)$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und zu dem gesprungen werden soll. Es handelt sich hierbei um ein versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung** wie in der Formel 1.3.1.

```
1  # // Exp(GoTo(Name('main.1')))
2  # // not included Exp(GoTo(Name('main.1')))
3  # StackMalloc(Num('2'))
4  SUBI SP 2;
5  # Ref(Global(Num('0')))
6  SUBI SP 1;
7  LOADI IN1 0;
8  ADD IN1 DS;
9  STOREIN SP IN1 1;
10 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
11 MOVE BAF ACC;
12 ADDI SP 3;
13 MOVE SP BAF;
14 SUBI SP 4;
15 STOREIN BAF ACC 0;
16 LOADI ACC 14;
17 ADD ACC CS;
18 STOREIN BAF ACC -1;
19 # Exp(GoTo(Name('stack_fun.0')))
20 JUMP 5;
21 # RemoveStackframe()
22 MOVE BAF IN1;
23 LOADIN IN1 BAF 0;
24 MOVE IN1 SP;
25 # Return(Empty())
26 LOADIN BAF PC -1;
```

```
27 # Return(Empty())
28 LOADIN BAF PC -1;
```

Code 1.28: RETI-Pass für Funktionsaufruf ohne Rückgabewert

1.3.3.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 1.29 erklärt.

Um den Unterschied zwischen einem `return` ohne **Rückgabewert** und einem `return 21 * 2` mit **Rückgabewert** hervorzuheben, wurde ist auch eine Funktion `fun_no_return_value`, die **keinen** Rückgabewert hat in das Beispiel integriert.

```
1  int fun_with_return_value() {
2    return 21 * 2;
3  }
4
5  void fun_no_return_value() {
6    return;
7  }
8
9  void main() {
10   int var = fun_with_return_value();
11   fun_no_return_value();
12 }
```

Code 1.29: PicoC-Code für Funktionsaufruf mit Rückgabewert

Im **Abstract Syntax Tree** in Code 1.30 wird ein **Return-Statement mit Rückgabewert** `return 21 * 2` mit der Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, ein **Return-Statement ohne Rückgabewert** `return` mit der Komposition `Return(Empty())` und ein **Funktionsaufruf** inklusive **Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` durch die Komposition `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))`.

```
1  File
2    Name './example_fun_call_with_return_value.ast',
3    [
4      FunDef
5        IntType 'int',
6        Name 'fun_with_return_value',
7        [],
8        [
9          Return(BinOp(Num('21'), Mul('*'), Num('2')))
10       ],
11     FunDef
12       VoidType 'void',
13       Name 'fun_no_return_value',
14       [],
15       [
```

```
16        Return(Empty())
17      ],
18    FunDef
19      VoidType 'void',
20      Name 'main',
21      [],
22      [
23        Assign(Alloc(Writeable(), IntType('int'), Name('var')),
          ↪  Call(Name('fun_with_return_value'), []))
24        Exp(Call(Name('fun_no_return_value'), []))
25      ]
26  ]
```

Code 1.30: Abstract Syntax Tree für Funktionsaufruf mit Rückgabewert

Im **PicoC-Mon Pass** in Code 1.31 wird bei der **Komposition** `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die **Expression** `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierführ erstellten Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis wird dann von der **Komposition** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register** `ACC` geschrieben und als letztes wird die **Rücksprungadresse** in das `PC`-Register geladen, die durch den `NewStackframe()`-Token-Knoten eine Speicherzelle nach dem Wert des `BAF`-Registers der aufrufenden Funktion im **Stackframe** gespeichert ist.

Ein wichtiges Detail bei der **Funktion** `fun_with_return_value` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird, da die **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Um den **Rückgabewert**, der durch die Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das `ACC`-Register geschrieben wurde für die aufrufende Funktion, deren Stackframe nun wieder das aktuelle ist auf den **Stack** zu schreiben, muss ein neue **Komposition** `Exp(ACC)` definiert werden. In Tabelle 1.7 ist die **Komposition** `Exp(ACC)` genauer erklärt.

Dieser Trick mit dem Speichern des **Rückgabewertes** im `ACC`-Register ist notwendidg, weil durch das **Entfernen** des **Stackframes** der **aufgerufenen Funktion** das `SP`-Register nicht mehr an der gleichen Stelle steht. Daher sind alle **temporären** Werte, die in der **aufgerufenen Funktion** auf den **Stack** geschrieben wurden unzugänglich, weil man nicht wissen kann, um wieviel die Adresse im `SP`-Register verglichen zu vorher verschoben ist, weil der **Stackframe** von unterschiedlichen **aufgerufenen Funktionen** unterschiedlich groß sein kann.

Die **Komposition** `Assign(Alloc(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))` wird nach dem **allokieren** der Variable `Name('var')` durch die Komposition `Assign(Global(Num('0')), Stack(Num('1')))` ersetzt, welche den **Rückgabewert** der Funktion `Name('fun_with_return_value')`, welcher durch die **Komposition** `Exp(Acc)` aus dem `ACC`-Register auf den **Stack** geschrieben wurde nun vom **Stack** in die Speicherzelle der Variable `Name('var')` speichert. Hierzu muss die **Adresse** der Variable `Name('var')` in der **Symboltabelle** nachgeschlagen werden.

Die **Komposition** `Return(Empty())` für ein `return` **ohne Rückgabewert** bleibt unverändert und stellt nur das Laden der **Rücksprungsadresse** in das `PC`-Register dar.

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` kein `return`-Statement explizit ans Ende geschrieben wird, im **PicoC-Mon Pass** eines hinzufügt wird in Form der `Komposition Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird allerdings eine `MissingReturn`-**Fehlermeldung** ausgelöst.

```
File
  Name './example_fun_call_with_return_value.picoc_mon',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        Exp(Num('21'))
        Exp(Num('2'))
        Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
        Return(Stack(Num('1')))
      ],
    Block
      Name 'fun_no_return_value.1',
      [
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_with_return_value.2')))
        RemoveStackframe()
        Exp(ACC)
        Assign(Global(Num('0')), Stack(Num('1')))
        StackMalloc(Num('2'))
        NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('fun_no_return_value.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.31: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

Im **RETI-Blocks Pass** in Code 1.32 werden die Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Mul('*'),Stack(Num('1'))))`, `Return(Stack(Num('1')))` und `Assign(Global(Num('0')),Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        # Exp(Num('21'))
        SUBI SP 1;
        LOADI ACC 21;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
```

```
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
19        MULT ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Return(Stack(Num('1')))
23        LOADIN SP ACC 1;
24        ADDI SP 1;
25        LOADIN BAF PC -1;
26      ],
27    Block
28      Name 'fun_no_return_value.1',
29      [
30        # Return(Empty())
31        LOADIN BAF PC -1;
32      ],
33    Block
34      Name 'main.0',
35      [
36        # // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
37        # StackMalloc(Num('2'))
38        SUBI SP 2;
39        # NewStackframe(Name('fun_with_return_value'), GoTo(Name('addr@next_instr')))
40        MOVE BAF ACC;
41        ADDI SP 2;
42        MOVE SP BAF;
43        SUBI SP 2;
44        STOREIN BAF ACC 0;
45        LOADI ACC GoTo(Name('addr@next_instr'));
46        ADD ACC CS;
47        STOREIN BAF ACC -1;
48        # Exp(GoTo(Name('fun_with_return_value.2')))
49        Exp(GoTo(Name('fun_with_return_value.2')))
50        # RemoveStackframe()
51        MOVE BAF IN1;
52        LOADIN IN1 BAF 0;
53        MOVE IN1 SP;
54        SUBI SP 1;
55        STOREIN SP ACC 1;
56        # Assign(Global(Num('0')), Stack(Num('1')))
57        LOADIN SP ACC 1;
58        STOREIN DS ACC 0;
59        ADDI SP 1;
60        # StackMalloc(Num('2'))
61        SUBI SP 2;
62        # NewStackframe(Name('fun_no_return_value'), GoTo(Name('addr@next_instr')))
63        MOVE BAF ACC;
64        ADDI SP 2;
65        MOVE SP BAF;
66        SUBI SP 2;
67        STOREIN BAF ACC 0;
68        LOADI ACC GoTo(Name('addr@next_instr'));
69        ADD ACC CS;
70        STOREIN BAF ACC -1;
```

```
71          # Exp(GoTo(Name('fun_no_return_value.1')))
72          Exp(GoTo(Name('fun_no_return_value.1')))
73          # RemoveStackframe()
74          MOVE BAF IN1;
75          LOADIN IN1 BAF 0;
76          MOVE IN1 SP;
77          # Return(Empty())
78          LOADIN BAF PC -1;
79        ]
80    ]
```

Code 1.32: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

1.3.3.3.2 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 1.33: PicoC-Code 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       {
```

```
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
        name:                   Name('param2@stack_fun')
        value or address:       Num('1')
        position:               Pos(Num('1'), Num('37'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('4'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
        name:                   Name('local_var1@main')
        value or address:       Num('0')
        position:               Pos(Num('5'), Num('6'))
        size:                   Num('6')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
        name:                   Name('local_var2@main')
        value or address:       Num('6')
        position:               Pos(Num('6'), Num('6'))
        size:                   Num('6')
      }
  ]
```

Code 1.34: Symboltabelle für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_sharing_array.picoc_mon',
  [
    Block
      Name 'stack_fun.1',
      [
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Ref(Global(Num('0')))
        Ref(Global(Num('6')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.1')))
```

```
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.35: PicoC-Mon Pass für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'main.0',
      [
        # 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 1.36: RETI-Block Pass für Call by Sharing für Arrays

1.3.3.3.3 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 1.37: PicoC-Code für Call by Value für Structs

```
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()
```

```
    },
  Symbol
    {
      type qualifier:         Writeable()
      datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
      name:                   Name('local_var1@main')
      value or address:       Num('0')
      position:               Pos(Num('5'), Num('6'))
      size:                   Num('6')
    },
  Symbol
    {
      type qualifier:         Writeable()
      datatype:               ArrayDecl([Num('2'), Num('3')], IntType('int'))
      name:                   Name('local_var2@main')
      value or address:       Num('6')
      position:               Pos(Num('6'), Num('6'))
      size:                   Num('6')
    }
]
```

Code 1.38: Symboltabelle für Call by Sharing für Arrays

```
File
  Name './example_fun_call_by_value_struct.picoc_mon',
  [
    Block
      Name 'stack_fun.1',
      [
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        StackMalloc(Num('2'))
        Assign(Stack(Num('3')), Global(Num('0')))
        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
        Exp(GoTo(Name('stack_fun.1')))
        RemoveStackframe()
        Return(Empty())
      ]
  ]
```

Code 1.39: PicoC-Mon Pass für Call by Value für Structs

```
File
  Name './example_fun_call_by_value_struct.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
```

```
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'main.0',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Assign(Stack(Num('3')), Global(Num('0')))
16        SUBI SP 3;
17        LOADIN DS ACC 0;
18        STOREIN SP ACC 1;
19        LOADIN DS ACC 1;
20        STOREIN SP ACC 2;
21        LOADIN DS ACC 2;
22        STOREIN SP ACC 3;
23        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24        MOVE BAF ACC;
25        ADDI SP 5;
26        MOVE SP BAF;
27        SUBI SP 5;
28        STOREIN BAF ACC 0;
29        LOADI ACC GoTo(Name('addr@next_instr'));
30        ADD ACC CS;
31        STOREIN BAF ACC -1;
32        # Exp(GoTo(Name('stack_fun.1')))
33        Exp(GoTo(Name('stack_fun.1')))
34        # RemoveStackframe()
35        MOVE BAF IN1;
36        LOADIN IN1 BAF 0;
37        MOVE IN1 SP;
38        # Return(Empty())
39        LOADIN BAF PC -1;
40      ]
41  ]
```

Code 1.40: RETI-Block Pass für Call by Value für Structs

1.4 Fehlermeldungen

1.4.1 Error Handler

1.4.2 Arten von Fehlermeldungen

1.4.2.1 Syntaxfehler

1.4.2.2 Laufzeitfehler

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/ (besucht am 18. 07. 2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).

Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).