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 Lexer erstellen

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	::=	"!"	
INT_DT.2	::=	"*int*"	
INT_NAME.3	::=	"*int*" (*LETTER* | *DIG_WITH_0* | "_")+	*L_Assign_Alloc*
CHAR_DT.2	::=	"*char*"	
CHAR_NAME.3	::=	"*char*" (*LETTER* | *DIG_WITH_0* | "_")+	
VOID_DT.2	::=	"*void*"	
VOID_NAME.3	::=	"*void*" (*LETTER* | *DIG_WITH_0* | "_")+	
prim_dt	::=	*INT_DT* | *CHAR_DT* | *VOID_DT*	

Grammar 1.1.1: Konkrette Syntax des Lexers in EBNF

1.1.2 Basic Lexer

1.2 Syntaktische Analyse

1.2.1 Konkrette Syntax für Parser erstellen

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 des Parsers 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 des Parsers 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()`	Steht für einen **Identifier**, aber `Name()`
`Num()`	
`Char()`	
`Minus()`	
`Not()`	
`Add()`	
`Sub()`	
`Mul()`	
`Div()`	
`Mod()`	
`Oplus()`	
`And()`	
`Or()`	
`DerefOp()`	
`RefOp()`	
`Eq()`	
`NEq()`	
`Lt()`	
`Gt()`	
`LtE()`	
`GtE()`	
`LogicAnd()`	
`LogicOr()`	
`LogicNot()`	
`Const()`	
`Writeable()`	
`IntType()`	
`CharType()`	
`VoidType()`	
`BinOp(exp1, bin_op, exp2)`	
`UnOp(un_op, exp)`	
`Exit(num)`	
`Atom(exp1, rel, exp2)`	
`ToBool(exp)`	
`Alloc(type_qual, datatype, name, local_var_or_param)`	
`Assign(lhs, exp)`	
`Exp(exp, datatype, error_data)`	
`Tmp(num)`	
`StackRead()`	
`StackWrite()`	
`GlobalRead()`	
`GlobalWrite()`	
`StackMalloc()`	

Tabelle 1.2: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
`PntrDecl()`	
`Ref()`	
`Deref()`	
`ArrayDecl()`	
`Array()`	
`Subscr()`	
`StructSpec()`	
`Attr()`	
`Struct()`	
`StructDecl()`	
`If()`	
`IfElse()`	
`While()`	
`DoWhile()`	
`Call()`	
`Return()`	
`FunDecl()`	
`FunDef()`	
`NewStackframe()`	
`RemoveStackframe()`	
`File()`	
`Block()`	
`GoTo()`	
`SingleLineComment()`	
`RETIComment()`	
`Placeholder()`	
`Tmp(Num('addr'))`	Steht für ein `exp`, dessen Wert bereits **vorher** berechnet wurde und `x` Speicherzellen **relativ** zum **Stackpointer** `SP` steht.

Tabelle 1.3: PicoC-Knoten Teil 2

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 ein verstecktes Attribut zugewiesen bekommen, dass im weiteren Kompiliervorgang Informationen transportiert, die später im Kompiliervorgang nicht mehr so leicht zugänglich wären.

1.2.5.2 RETI-Knoten

RETI-Knoten	Beschreibung
	asdf

Tabelle 1.4: 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(GlobalRead(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Datensegment Register** `DS` steht auf den **Stack**.
`Ref(StackRead(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Begin-Aktive-Funktion Register** `BAF` steht auf den **Stack**.
`Ref(Subscr(Tmp(Num('addr1')), Tmp(Num('addr2'))))`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Tmp(Num('addr1'))` steht und dem **Subscript Index**, der an Speicherzelle `Tmp(Num('addr2'))` steht und speichert diese auf den **Stack**. Die Berechnung ist abhängig davon ob der **Datentyp** `ArrayDecl(datatype)` oder `PntrDecl(datatype)` ist. Der **Datentyp** ist ein **verstecktes Attribut** von `Ref(exp)`.
`Ref(Attr(Tmp(Num('addr1')), Name('attr')))`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Tmp(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(Tmp(Num('size')), GlobalRead(Num('addr')))`	Schreibt `Num('size')` viele Speicherzellen, die ab `GlobalRead(Num('addr'))` relativ zum **Datensegment Register** `DS` stehen, versetzt genauso auf den **Stack**.
`Assign(Tmp(Num('size')), StackRead(Num('addr')))`	Schreibt `Num('size')` viele Speicherzellen, die ab `GlobalRead(Num('addr'))` relativ zum **Begin-Aktive-Funktion Register** `BAF` stehen, versetzt genauso auf den **Stack**.
`Exp(GlobalRead(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Datensegment Register** `DS` steht auf den **Stack**.
`Exp(StackRead(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Begin-Aktive-Funktion Register** `BAF` steht auf den **Stack**.
`Exp(Tmp(Num('addr')))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum **Stackpointer Register** `SP` steht auf den **Stack**.
`Assign(Tmp(Num('addr1')), Tmp(Num('addr2')))`	Speichert **Inhalt** der Speicherzelle `Tmp(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(GlobalWrite(Num('addr')), Tmp(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(StackWrite(Num('addr')), Tmp(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 1.5: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 1.5 nicht mit unnötig viel repetetiven Inhalt zu füllen, wurden die zahlreichen Kompostionen **ausgelassen**, bei denen einfach nur `exp` durch $\texttt{Tmp}(\texttt{Num}(\texttt{'x'})), \texttt{x} \in \mathbb{N}$ 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	$::=$	$Alloc(\langle type\_qual\rangle, \langle dataype\rangle, Name(str))$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
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*
$datatype$	$::=$	$IntType() \mid CharType() \mid VoidType()$	
$assign\_lhs$	$::=$	$Alloc(\langle type\_qual\rangle, \langle dataype\rangle, Name(str)) \mid \langle rel\_loc\rangle$	
exp_stmts	$::=$	$Alloc(\langle type\_qual\rangle, \langle dataype\rangle, Name(str))$	
$stmt$	$::=$	$Assign(\langle assign\_lhs\rangle, \langle exp\rangle)$	
	$\mid$	$Exp(\langle exp\_stmts\rangle)$	
$datatype$	$::=$	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
$deref\_loc$	$::=$	$Ref(\langle ref\_loc\rangle) \mid \langle ref\_loc\rangle$	
ref_loc	$::=$	$Name(str)$	
	$\mid$	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Attr(\langle ref\_loc\rangle, Name(str))$	
exp	$::=$	$Deref(\langle deref\_loc\rangle, \langle exp\rangle)$	
	$\mid$	$Ref(\langle ref\_loc\rangle)$	
$datatype$	$::=$	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	$::=$	$Subscr(\langle deref\_loc\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	$::=$	$StructSpec(Name(str))$	*L_Struct*
exp	$::=$	$Attr(\langle ref\_loc\rangle, Name(str))$	
	$\mid$	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
$decl\_def$	$::=$	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
$stmt$	$::=$	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	$\mid$	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
$stmt$	$::=$	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	$\mid$	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	$::=$	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
exp_stmts	$::=$	$Call(Name(str), \langle exp\rangle*)$	
$stmt$	$::=$	$Return(\langle exp\rangle)$	
$decl\_def$	$::=$	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
$file$	$::=$	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

Grammar 1.2.3: Abstrakte Syntax für L_{PiocC}

1.2.5.5 Transformer

1.2.5.6 Codebeispiel

Beispiel welches in Subkapitel 1.2.3.2 angefangen wurde, wird hier fortgeführt.

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

Code 1.4: Abstract Syntax Tree aus vereinfachtem Derivarion Tree generiert

1.3 Code Generierung

1.3.1 Übersicht

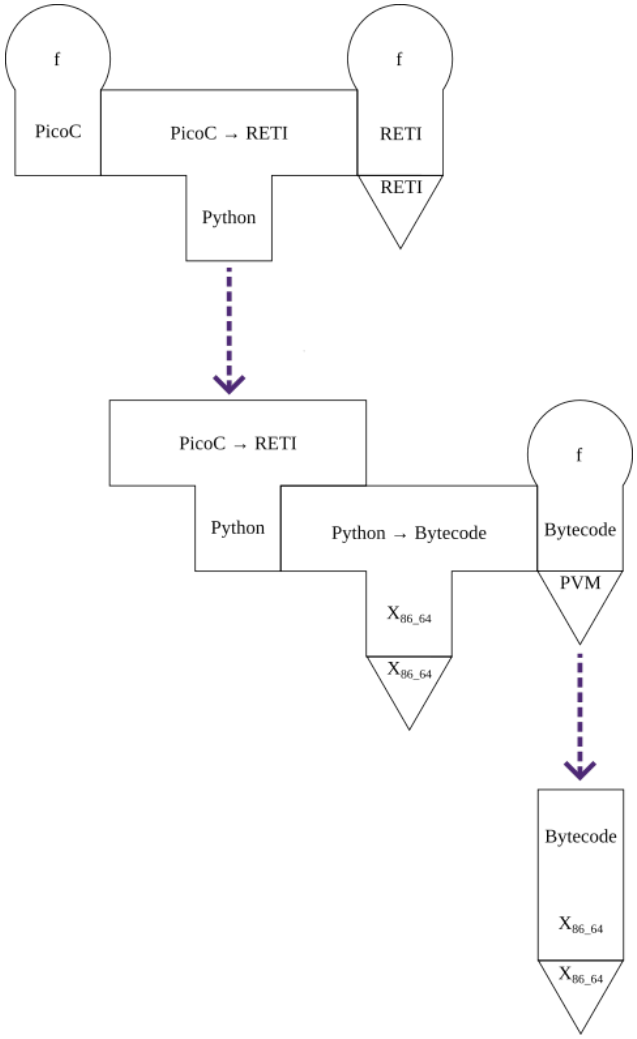

Abbildung 1.1: Cross-Compiler Kompiliervorgang ausgeschrieben

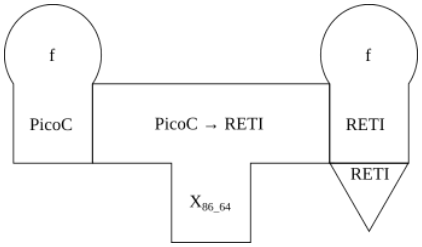

Abbildung 1.2: Cross-Compiler Kompiliervorgang Kurzform

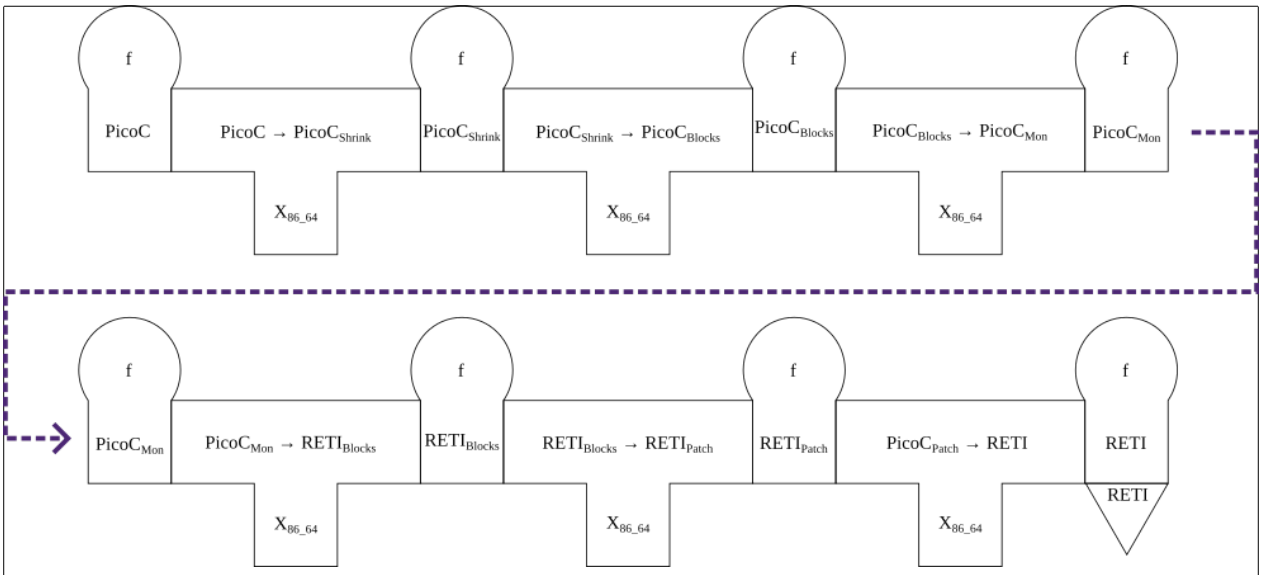

Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

1.3.2 Passes

1.3.2.1 PicoC-Shrink Pass

1.3.2.1.1 Codebeispiel

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

Code 1.5: PicoC Code für Codebespiel

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

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

Code 1.6: Abstract Syntax Tree für Codebespiel

```
File
 Name './example_faculty_it.picoc_shrink',
 [
   FunDef
     VoidType 'void',
     Name 'main',
     [],
     [
       Assign
         Alloc
           Writeable,
           IntType 'int',
```

```
13            Name 'n',
14          Num '4',
15        Assign
16          Alloc
17            Writeable,
18            IntType 'int',
19            Name 'res',
20          Num '1',
21        While
22          Num '1',
23          [
24            If
25              Atom
26                Name 'n',
27                Eq '==',
28                Num '1',
29              [
30                Return
31                  Empty
32              ],
33            Assign
34              Name 'res',
35              BinOp
36                Name 'n',
37                Mul '*',
38                Name 'res',
39            Assign
40              Name 'n',
41              BinOp
42                Name 'n',
43                Sub '-',
44                Num '1'
45          ]
46      ]
47  ]
```

Code 1.7: PicoC Shrink Pass für Codebespiel

1.3.2.2 PicoC-Blocks Pass

1.3.2.2.1 Abstrakte Syntax

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

Grammar 1.3.1: Abstrakte Syntax für L_{PicoC_Blocks}

1.3.2.2.2 Codebeispiel

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

```
58        Block
59          Name 'if_else_after.1',
60          [
61            Assign
62              Name 'res',
63              BinOp
64                Name 'n',
65                Mul '*',
66                Name 'res',
67            Assign
68              Name 'n',
69              BinOp
70                Name 'n',
71                Sub '-',
72                Num '1',
73            GoTo
74              Name 'condition_check.4'
75          ],
76        Block
77          Name 'while_after.0',
78          []
79      ]
80  ]
```

Code 1.8: PicoC-Blocks Pass für Codebespiel

1.3.2.3 PicoC-Mon Pass

1.3.2.3.1 Abstrakte Syntax

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

Grammar 1.3.2: Abstrakte Syntax für L_{PicoC_Mon}

Definition 1.1: Symboltabelle

1.3.2.3.2 Codebeispiel

```
File
  Name './example_faculty_it.picoc_mon',
  [
    Block
      Name 'main.5',
      [
        // Assign(Name('n'), Num('4')),
        Exp
          Num '4',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '1',
        // Assign(Name('res'), Num('1')),
        Exp
          Num '1',
        Assign
          GlobalWrite
            Num '1',
          Tmp
            Num '1',
        // While(Num('1'), []),
        Exp
          GoTo
            Name 'condition_check.4'
      ],
    Block
      Name 'condition_check.4',
      [
        // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
        Exp
          Num '1',
        IfElse
          Tmp
            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'))),
        Exp
          GlobalRead
            Num '0',
```

```
      Exp
        Num '1',
      Exp
        Atom
          Tmp
            Num '2',
          Eq '==',
          Tmp
            Num '1',
      IfElse
        Tmp
          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'))),
      Exp
        GlobalRead
          Num '0',
      Exp
        GlobalRead
          Num '1',
      Exp
        BinOp
          Tmp
            Num '2',
          Mul '*',
          Tmp
            Num '1',
      Assign
        GlobalWrite
          Num '1',
        Tmp
          Num '1',
      // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1'))),
      Exp
        GlobalRead
          Num '0',
      Exp
        Num '1',
      Exp
        BinOp
          Tmp
            Num '2',
          Sub '-',
          Tmp
```

```
107             Num '1',
108       Assign
109         GlobalWrite
110           Num '0',
111         Tmp
112           Num '1',
113       Exp
114         GoTo
115           Name 'condition_check.4'
116     ],
117   Block
118     Name 'while_after.0',
119     [
120       Return
121         Empty
122     ]
123 ]
```

Code 1.9: PicoC-Mon Pass für Codebespiel

1.3.2.4 RETI-Blocks Pass

1.3.2.4.1 Abstrakte Syntax

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

Grammar 1.3.3: Abstrakte Syntax für L_{RETI_Blocks}

1.3.2.4.2 Codebeispiel

```
File
 Name './example_faculty_it.reti_blocks',
 [
   Block
     Name 'main.5',
     [
       # // Assign(Name('n'), Num('4')),
       # Exp(Num('4')),
       SUBI SP 1,
       LOADI ACC 4,
       STOREIN SP ACC 1,
       # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
       LOADIN SP ACC 1,
       STOREIN DS ACC 0,
       ADDI SP 1,
       # // Assign(Name('res'), Num('1')),
       # Exp(Num('1')),
```

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

```
74        Exp
75          GoTo
76            Name 'if.2'
77      ],
78    Block
79      Name 'if.2',
80      [
81        # Return(Empty()),
82        LOADIN BAF PC -1
83      ],
84    Block
85      Name 'if_else_after.1',
86      [
87        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res'))),
88        # Exp(GlobalRead(Num('0'))),
89        SUBI SP 1,
90        LOADIN DS ACC 0,
91        STOREIN SP ACC 1,
92        # Exp(GlobalRead(Num('1'))),
93        SUBI SP 1,
94        LOADIN DS ACC 1,
95        STOREIN SP ACC 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(GlobalWrite(Num('1')), Tmp(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(GlobalRead(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       LOADIN SP ACC 2,
115       LOADIN SP IN2 1,
116       SUB ACC IN2,
117       STOREIN SP ACC 2,
118       ADDI SP 1,
119       # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
120       LOADIN SP ACC 1,
121       STOREIN DS ACC 0,
122       ADDI SP 1,
123       Exp
124         GoTo
125           Name 'condition_check.4'
126     ],
127   Block
128     Name 'while_after.0',
129     [
130       # Return(Empty()),
```

```
131         LOADIN BAF PC -1
132       ]
133   ]
```

Code 1.10: RETI-Blocks Pass für Codebespiel

1.3.2.5 RETI-Patch Pass

1.3.2.5.1 Abstrakte Syntax

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

Grammar 1.3.4: Abstrakte Syntax für L_{RETI_Patch}

1.3.2.5.2 Codebeispiel

```
1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.6',
6       [],
7     Block
8       Name 'main.5',
9       [
10        # // Assign(Name('n'), Num('4')),
11        # Exp(Num('4')),
12        SUBI SP 1,
13        LOADI ACC 4,
14        STOREIN SP ACC 1,
15        # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
16        LOADIN SP ACC 1,
17        STOREIN DS ACC 0,
18        ADDI SP 1,
19        # // Assign(Name('res'), Num('1')),
20        # Exp(Num('1')),
21        SUBI SP 1,
22        LOADI ACC 1,
23        STOREIN SP ACC 1,
24        # Assign(GlobalWrite(Num('1')), Tmp(Num('1'))),
25        LOADIN SP ACC 1,
26        STOREIN DS ACC 1,
27        ADDI SP 1,
28        # // While(Num('1'), [])
29      ],
30    Block
31      Name 'condition_check.4',
32      [
33        # // IfElse(Num('1'), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
34        # Exp(Num('1')),
35        SUBI SP 1,
36        LOADI ACC 1,
37        STOREIN SP ACC 1,
```

```
38        # IfElse(Tmp(Num('1')), GoTo(Name('while_branch.3')), GoTo(Name('while_after.0'))),
39        LOADIN SP ACC 1,
40        ADDI SP 1,
41        JUMP== GoTo
42                Name 'while_after.0';
43      ],
44    Block
45      Name 'while_branch.3',
46      [
47        # // If(Atom(Name('n'), Eq('==')), Num('1')), []),
48        # // IfElse(Atom(Name('n'), Eq('==')), Num('1')), GoTo(Name('if.2')),
          ↪  GoTo(Name('if_else_after.1'))),
49        # Exp(GlobalRead(Num('0'))),
50        SUBI SP 1,
51        LOADIN DS ACC 0,
52        STOREIN SP ACC 1,
53        # Exp(Num('1')),
54        SUBI SP 1,
55        LOADI ACC 1,
56        STOREIN SP ACC 1,
57        LOADIN SP ACC 2,
58        LOADIN SP IN2 1,
59        SUB ACC IN2,
60        JUMP== 3;,
61        LOADI ACC 0,
62        JUMP 2;,
63        LOADI ACC 1,
64        STOREIN SP ACC 2,
65        ADDI SP 1,
66        # IfElse(Tmp(Num('1')), GoTo(Name('if.2')), GoTo(Name('if_else_after.1'))),
67        LOADIN SP ACC 1,
68        ADDI SP 1,
69        JUMP== GoTo
70                Name 'if_else_after.1';
71      ],
72    Block
73      Name 'if.2',
74      [
75        # Return(Empty()),
76        LOADIN BAF PC -1
77      ],
78    Block
79      Name 'if_else_after.1',
80      [
81        # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res'))),
82        # Exp(GlobalRead(Num('0'))),
83        SUBI SP 1,
84        LOADIN DS ACC 0,
85        STOREIN SP ACC 1,
86        # Exp(GlobalRead(Num('1'))),
87        SUBI SP 1,
88        LOADIN DS ACC 1,
89        STOREIN SP ACC 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(GlobalWrite(Num('1')), Tmp(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(GlobalRead(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         LOADIN SP ACC 2,
109         LOADIN SP IN2 1,
110         SUB ACC IN2,
111         STOREIN SP ACC 2,
112         ADDI SP 1,
113         # Assign(GlobalWrite(Num('0')), Tmp(Num('1'))),
114         LOADIN SP ACC 1,
115         STOREIN DS ACC 0,
116         ADDI SP 1,
117         Exp
118           GoTo
119             Name 'condition_check.4'
120       ],
121     Block
122       Name 'while_after.0',
123       [
124         # Return(Empty()),
125         LOADIN BAF PC -1
126       ]
127   ]
```

Code 1.11: RETI-Patch Pass für Codebespiel

1.3.2.6 RETI Pass

1.3.2.6.1 Konkrette und Abstrakte Syntax

dig_no_0	::=	"1" | "2" | "3" | "4" | "5" | "6" | "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}

1.3.2.6.2 Codebeispiel

```
1  # // Assign(Name('n'), Num('4'))
2  # Exp(Num('4'))
3  SUBI SP 1;
4  LOADI ACC 4;
5  STOREIN SP ACC 1;
6  # Assign(GlobalWrite(Num('0')), Tmp(Num('1')))
7  LOADIN SP ACC 1;
8  STOREIN DS ACC 0;
9  ADDI SP 1;
10 # // Assign(Name('res'), Num('1'))
11 # Exp(Num('1'))
12 SUBI SP 1;
13 LOADI ACC 1;
14 STOREIN SP ACC 1;
15 # Assign(GlobalWrite(Num('1')), Tmp(Num('1')))
```

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

```
72 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
73 # Exp(GlobalRead(Num('0')))
74 SUBI SP 1;
75 LOADIN DS ACC 0;
76 STOREIN SP ACC 1;
77 # Exp(Num('1'))
78 SUBI SP 1;
79 LOADI ACC 1;
80 STOREIN SP ACC 1;
81 LOADIN SP ACC 2;
82 LOADIN SP IN2 1;
83 SUB ACC IN2;
84 STOREIN SP ACC 2;
85 ADDI SP 1;
86 # Assign(GlobalWrite(Num('0')), Tmp(Num('1')))
87 LOADIN SP ACC 1;
88 STOREIN DS ACC 0;
89 ADDI SP 1;
90 JUMP -53;
91 # Return(Empty())
92 LOADIN BAF PC -1;
```

Code 1.12: RETI Pass für Codebespiel

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).