
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
1 Implementierung	1
1.1 Lexikalische Analyse	1
1.1.1 Teil der Konkreten Syntax für die Lexikalische Analyse	1
1.1.2 Basic Lexer	2
1.2 Syntaktische Analyse	2
1.2.1 Teil der Konkreten Syntax für die Syntaktische Analyse	2
1.2.2 Umsetzung von Präzidenz	4
1.2.3 Derivation Tree Generierung	5
1.2.3.1 Early Parser	5
1.2.3.2 Codebeispiel	5
1.2.4 Derivation Tree Vereinfachung	6
1.2.4.1 Visitor	6
1.2.4.2 Codebeispiel	6
1.2.5 Abstrakt Syntax Tree Generierung	8
1.2.5.1 PicoC-Knoten	8
1.2.5.2 RETI-Knoten	13
1.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	14
1.2.5.4 Abstrakte Syntax	16
1.2.5.5 Transformer	18
1.2.5.6 Codebeispiel	18
1.3 Code Generierung	18
1.3.1 Übersicht	18
1.3.2 Passes	21
1.3.2.1 PicoC-Shrink Pass	21
1.3.2.1.1 Aufgabe	21
1.3.2.1.2 Abstrakte Syntax	21
1.3.2.1.3 Codebeispiel	22
1.3.2.2 PicoC-Blocks Pass	24
1.3.2.2.1 Aufgabe	24
1.3.2.2.2 Abstrakte Syntax	24
1.3.2.2.3 Codebeispiel	26
1.3.2.3 PicoC-ANF Pass	27
1.3.2.3.1 Aufgabe	27
1.3.2.3.2 Abstrakte Syntax	27
1.3.2.3.3 Codebeispiel	29
1.3.2.4 RETI-Blocks Pass	30
1.3.2.4.1 Aufgaben	30

1.3.2.4.2	Abstrakte Syntax	30
1.3.2.4.3	Codebeispiel	30
1.3.2.5	RETI-Patch Pass	33
1.3.2.5.1	Aufgaben	33
1.3.2.5.2	Abstrakte Syntax	33
1.3.2.5.3	Codebeispiel	34
1.3.2.6	RETI Pass	37
1.3.2.6.1	Aufgaben	37
1.3.2.6.2	Konkrete und Abstrakte Syntax	37
1.3.2.6.3	Codebeispiel	38
Appendix		A
Danksagungen		B
Literatur		C

Abbildungsverzeichnis

1.1	Cross-Compiler Kompiliervorgang ausgeschrieben	19
1.2	Cross-Compiler Kompiliervorgang Kurzform	20
1.3	Architektur mit allen Passes ausgeschrieben	20

Codeverzeichnis

1.1	PicoC Code für Derivation Tree Generierung	5
1.2	Derivation Tree nach Derivation Tree Generierung	6
1.3	Derivation Tree nach Derivation Tree Vereinfachung	7
1.4	Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert	18
1.5	PicoC Code für Codebeispiel	23
1.6	Abstract Syntax Tree für Codebeispiel	24
1.7	PicoC-Blocks Pass für Codebeispiel	27
1.8	PicoC-ANF Pass für Codebeispiel	30
1.9	RETI-Blocks Pass für Codebeispiel	33
1.10	RETI-Patch Pass für Codebeispiel	36
1.11	RETI Pass für Codebeispiel	40

Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC	5
1.2	PicoC-Knoten Teil 1	8
1.3	PicoC-Knoten Teil 2	9
1.4	PicoC-Knoten Teil 3	10
1.5	PicoC-Knoten Teil 4	11
1.6	RETI-Knoten	13
1.7	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	15

Definitionsverzeichnis

1.1	Label	12
1.2	Token-Knoten	12
1.3	Container-Knoten	12
1.4	Symboltabelle	28

Grammatikverzeichnis

1.1.1 Teil der Konkreten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF, Teil 1	1
1.1.2 Teil der Konkreten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF, Teil 2	2
1.2.1 Teil der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1	3
1.2.2 Teil der Konkreten Syntax der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2	4
1.2.3 Abstrakte Syntax der Sprache L_{PiocC}	17
1.3.1 Abstrakte Syntax der Sprache L_{PiocC_Shrink}	22
1.3.2 Abstrakte Syntax der Sprache L_{PiocC_Blocks}	25
1.3.3 Abstrakte Syntax der Sprache L_{PiocC_ANF}	28
1.3.4 Abstrakte Syntax für L_{RETI_Blocks}	30
1.3.5 Abstrakte Syntax für L_{RETI_Patch}	33
1.3.6 Konkrete Syntax für L_{RETI_Lex}	37
1.3.7 Konkrete Syntax für L_{RETI_Parse}	37
1.3.8 Abstrakte Syntax für L_{RETI}	38

1 Implementierung

1.1 Lexikalische Analyse

1.1.1 Teil der Konkreten Syntax für die Lexikalische Analyse

<i>COMMENT</i>	::=	<code>"/" /[\n]* / "/" /(. \n)*? / "*" /</code>	<i>L_Comment</i>
<i>RETI.COMMENT.2</i>	::=	<code>"/" " " "?" "#" /[\n]* /</code>	
<i>DIG.NO_0</i>	::=	<code>"1" "2" "3" "4" "5"</code> <code> "6" "7" "8" "9"</code>	<i>L_Arith</i>
<i>DIG.WITH_0</i>	::=	<code>"0" DIG.NO_0</code>	
<i>NUM</i>	::=	<code>"0" DIG.NO_0 DIG.WITH_0*</code>	
<i>ASCII.CHAR</i>	::=	<code>" " .. " " ~ "</code>	
<i>CHAR</i>	::=	<code>" " ASCII.CHAR " " "</code>	
<i>FILENAME</i>	::=	<code>ASCII.CHAR + ".picoc"</code>	
<i>LETTER</i>	::=	<code>"a" .. "z" "A" .. "Z"</code>	
<i>NAME</i>	::=	<code>(LETTER " ")</code> <code>(LETTER — DIG.WITH_0 — " ")*</code>	
<i>name</i>	::=	<code>NAME INT.NAME CHAR.NAME</code> <code> VOID.NAME</code>	
<i>NOT</i>	::=	<code>" ~ "</code>	
<i>REF_AND</i>	::=	<code>" & "</code>	
<i>un_op</i>	::=	<code>SUB.MINUS LOGIC.NOT NOT</code> <code> MUL.DEREF.PNTR REF_AND</code>	
<i>MUL.DEREF.PNTR</i>	::=	<code>" * "</code>	
<i>DIV</i>	::=	<code>" / "</code>	
<i>MOD</i>	::=	<code>" % "</code>	
<i>prec1_op</i>	::=	<code>MUL.DEREF.PNTR DIV MOD</code>	
<i>ADD</i>	::=	<code>" + "</code>	
<i>SUB.MINUS</i>	::=	<code>" - "</code>	
<i>prec2_op</i>	::=	<code>ADD SUB.MINUS</code>	
<i>LT</i>	::=	<code>" < "</code>	<i>L_Logic</i>
<i>LTE</i>	::=	<code>" < = "</code>	
<i>GT</i>	::=	<code>" > "</code>	
<i>GTE</i>	::=	<code>" > = "</code>	
<i>rel_op</i>	::=	<code>LT LTE GT GTE</code>	
<i>EQ</i>	::=	<code>" = = "</code>	
<i>NEQ</i>	::=	<code>" ! = "</code>	
<i>eq_op</i>	::=	<code>EQ NEQ</code>	
<i>LOGIC.NOT</i>	::=	<code>" ! "</code>	

Grammar 1.1.1: Teil der Konkreten Syntax der Sprache L_{Picoc} für die Lexikalische Analyse in EBNF, Teil 1

<i>INT_DT.2</i>	::=	"int"	<i>L_Assign_Alloc</i>
<i>INT_NAME.3</i>	::=	"int" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" (<i>LETTER</i> <i>DIG_WITH_0</i> "_")+	
<i>prim_dt</i>	::=	<i>INT_DT</i> <i>CHAR_DT</i> <i>VOID_DT</i>	

Grammar 1.1.2: Teil der Konkretten Syntax der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF, Teil 2

1.1.2 Basic Lexer

1.2 Syntaktische Analyse

1.2.1 Teil der Konkretten Syntax für die Syntaktische Analyse

In 1.2.1

<i>prim_exp</i>	::=	<i>name</i> <i>NUM</i> <i>CHAR</i> "(" <i>logic_or</i> ")"	<i>L_Arith</i> +
<i>post_exp</i>	::=	<i>array_subscr</i> <i>struct_attr</i> <i>fun_call</i>	<i>L_Array</i> +
		<i>input_exp</i> <i>print_exp</i> <i>prim_exp</i>	<i>L_Pntr</i> +
<i>un_exp</i>	::=	<i>un_op</i> <i>un_exp</i> <i>post_exp</i>	<i>L_Struct</i> + <i>L_Fun</i>
<i>input_exp</i>	::=	"input" "(" ")"	<i>L_Arith</i>
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i> <i>un_exp</i>	
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i> <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i> <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i> <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i> <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i> <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_op</i> <i>rel_exp</i> <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i> <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> " " <i>logic_and</i> <i>logic_and</i>	
<i>type_spec</i>	::=	<i>prim_dt</i> <i>struct_spec</i>	<i>L_Assign_Alloc</i>
<i>alloc</i>	::=	<i>type_spec</i> <i>pntr_decl</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	
<i>initializer</i>	::=	<i>logic_or</i> <i>array_init</i> <i>struct_init</i>	
<i>init_stmt</i>	::=	<i>alloc</i> "=" <i>initializer</i> ";"	
<i>const_init_stmt</i>	::=	"const" <i>type_spec</i> <i>name</i> "=" <i>NUM</i> ";"	
<i>pntr_deg</i>	::=	"*" *	<i>L_Pntr</i>
<i>pntr_decl</i>	::=	<i>pntr_deg</i> <i>array_decl</i> <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]") *	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name</i> <i>array_dims</i> "(" <i>pntr_decl</i> ")" <i>array_dims</i>	
<i>array_init</i>	::=	"{" <i>initializer</i> ("," <i>initializer</i>) * "}"	
<i>array_subscr</i>	::=	<i>post_exp</i> "[" <i>logic_or</i> "]"	
<i>struct_spec</i>	::=	"struct" <i>name</i>	<i>L_Struct</i>
<i>struct_params</i>	::=	(<i>alloc</i> ";") +	
<i>struct_decl</i>	::=	"struct" <i>name</i> "{" <i>struct_params</i> "}"	
<i>struct_init</i>	::=	"{" "." <i>name</i> "=" <i>initializer</i> ("," "." <i>name</i> "=" <i>initializer</i>) * "}"	
<i>struct_attr</i>	::=	<i>post_exp</i> "." <i>name</i>	
<i>if_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_If_Else</i>
<i>if_else_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i> "else" <i>exec_part</i>	
<i>while_stmt</i>	::=	"while" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_Loop</i>
<i>do_while_stmt</i>	::=	"do" <i>exec_part</i> "while" "(" <i>logic_or</i> ")" ";"	

Grammar 1.2.1: Teil der Konkreten Syntax der Sprache L_{Picoc} für die Syntaktische Analyse in EBNF, Teil 1

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

Grammar 1.2.2: Teil der Konkretten Syntax der Sprache L_{PicoC} 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.

¹C Operator Precedence - cppreference.com.

Präzidenz	Operator	Beschreibung	Assoziativität
1	a()	Funktionsaufruf	Links, dann rechts →
	a[]	Indezzugriff	
	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	Logisches 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

1.2.3 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         ptr_deg *
11         array_decl
12         array_dims
13         5
14         6
15         ptr_decl
16         ptr_deg *
17         array_decl
18         array_dims
19         type_spec
20         prim_dt int
21         name attr
22 decl_def
23 fun_def
24     type_spec
25     prim_dt void
26     ptr_deg
27     name main
28     fun_params
29     decl_exec_stmts
30     decl_part
31     decl_exp_stmt
32     alloc
33     ptr_decl
34     ptr_deg *
35     array_decl
36     array_dims
37     3
38     2
39     ptr_decl
40     ptr_deg *
41     array_decl
42     array_dims
43     type_spec
44     struct_spec
45         name st
46     name var

```

Code 1.3: *Derivation Tree nach Derivation Tree Vereinfachung*

1.2.5 Abstrakt Syntax Tree Generierung

1.2.5.1 PicoC-Knoten

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

Tabelle 1.2: PicoC-Knoten Teil 1

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

Tabelle 1.3: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den Initializer eines Structs , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines Attributezeichners , zu einem weiteren Initializer für eine Array-Dimension oder zu einem Initializer für ein Struct oder zu einem Logischen Ausdruck . Des Weiteren besitzt er ein verstecktes Attribut datatype, welches für den PicoC-ANF Pass Informationen transportiert, die für Fehlermeldungen wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines selbstdefinierten Structdatentyps , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der Bezeichner des Structdatentyps ist und allocs eine Liste von Bezeichnern der Attribute des Structdatentyps mit dazugehörigem Datentyp , wofür sich der Container-Knoten Alloc(type_qual, datatype, name) sehr gut als Container eignet.
If(exp, stmts)	Container für ein If Statement if(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts1, stmts2)	Container für ein If-Else Statement if(<exp>) { <stmts2> } else { <stmts2> } inklusive Condition exp und 2 Branches stmts1 und stmts2, die zwei Alternativen darstellen in denen jeweils Listen von Statements oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts)	Container für ein While-Statement while(<exp>) { <stmts> } inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts)	Container für ein Do-While-Statement do { <stmts> } while(<exp>); inklusive Condition exp und einem Branch stmts, indem eine Liste von Statements stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen Funktionsaufruf : fun_name(exps), wobei name der Bezeichner der Funktion ist, die aufgerufen werden soll und exps eine Liste von Argumenten ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein Return-Statement : return <exp>, wobei das Attribut exp einen Logischen Ausdruck darstellt, dessen Ergebnis vom Return-Statement zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine Funktionsdeklaration , z.B. <datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist und allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient.

Tabelle 1.4: PicoC-Knoten Teil 3

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts_blocks)	Container für eine Funktionsdefinition , z.B. <datatype> <fun_name>(<datatype> <param>) {<stmts>}, wobei datatype der Rückgabewert der Funktion ist, name der Bezeichner der Funktion ist, allocs die Parameter der Funktion sind, wobei der Container-Knoten Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts_blocks eine Liste von Statements bzw. Blöcken ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die Erstellung eines neuen Stackframes und Speicherung des Werts des BAF-Registers der aufgerufenen Funktion und der Rücksprungsadresse nacheinander an den Anfang des neuen Stackframes . Das Attribut fun_name steht 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 Kompilierungsvorgang wichtige Information in seinen versteckten 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-Registers der aufgerufenen Funktion und das Setzen des SP-Registers auf den Wert des BAF-Registers vor der Wiederherstellung.
File(name, decls_defs_blocks)	Container für alle Funktionen oder Blöcke , welche eine Datei als Ursprung haben, wobei name der Dateiname der Datei ist, die erstellt wird und decls_defs_blocks eine Liste von Funktionen bzw. Blöcken ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für Statements , der auch als Block bezeichnet wird, wobei das Attribut name der Bezeichner 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 Kompilierungsvorgangs 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-Interpreter später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer RETI-CPU nicht umsetzbar ist und auch nicht sinnvoll wäre umzusetzen. Der Kommentar ist im Attribut value , welches jeder Knoten besitzt gespeichert.

Tabelle 1.5: PicoC-Knoten Teil 4

Definition 1.1: Label

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

^aThiemann, „Compilerbau“.

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

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 1.2) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.3) unbesetzt ist. Das **Attribut** **position** wird später für Fehlermeldungen gebraucht.

Definition 1.2: 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

^aThiemann, „Compilerbau“.

Definition 1.3: Container-Knoten

Dient als **Container** für andere **Container-Knoten** und **Token-Knoten**. Die **Container-Knoten** werden optimalerweise immer so gewählt, dass sie **mehrere Produktionen** der **Konkreten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.^a

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

^aWie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. **1 % 3** und **Logischen Ausdrücke**, wie z.B. **1 && 2 < 3**, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.

^bThiemann, „Compilerbau“.

1.2.5.2 RETI-Knoten

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

^a Scholl, „Betriebssysteme“

Tabelle 1.6: RETI-Knoten

1.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Hier sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax 1.2.1** enthalten sind.

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

Tabelle 1.7: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 1.7 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch **Stack(Num('x'))**, $x \in \mathbb{N}$ ersetzt wurde.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen einfach nur eine **Expression** an ein **Exp(exp)** bzw. **Ref(exp)** drangehängt wurde.

1.2.5.4 Abstrakte Syntax

<i>stmt</i>	::=	<i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp</i> (<i><exp></i> , <i><bin_op></i> , <i><exp></i>) <i>UnOp</i> (<i><un_op></i> , <i><exp></i>) <i>Call</i> (<i>Name('input')</i> , <i>Empty()</i>)	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom</i> (<i><exp></i> , <i><rel></i> , <i><exp></i>) <i>ToBool</i> (<i><exp></i>)	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>)	
<i>stmt</i>	::=	<i>Assign</i> (<i><exp></i> , <i><exp></i>) <i>Exp</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num(str)</i> , <i><datatype></i>)	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref</i> (<i><exp></i> , <i><exp></i>) <i>Ref</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num(str)</i> +, <i><datatype></i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i><exp></i> , <i><exp></i>) <i>Array</i> (<i><exp></i> +))	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name(str)</i>)	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i><exp></i> , <i>Name(str)</i>) <i>StructAssign</i> (<i>Name(str)</i> , <i><exp></i>)+)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)+)	
<i>stmt</i>	::=	<i>If</i> (<i><exp></i> , <i><stmt></i> *) <i>IfElse</i> (<i><exp></i> , <i><stmt></i> *, <i><stmt></i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> (<i><exp></i> , <i><stmt></i> *) <i>DoWhile</i> (<i><exp></i> , <i><stmt></i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> (<i>Name(str)</i> , <i><exp></i> *)	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return</i> (<i><exp></i>)	
<i>decl_def</i>	::=	<i>FunDecl</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*) <i>FunDef</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*, <i><stmt></i> *)	
<i>file</i>	::=	<i>File</i> (<i>Name(str)</i> , <i><decl_def></i> *)	<i>L_File</i>

Grammar 1.2.3: Abstrakte Syntax der Sprache L_{PicoC}

Man spricht hier von der „**Abstrakten Syntax der Sprache L_{PicoC}** “ und meint hier mit der Sprache L_{PicoC} **nicht** die Sprache, welche durch die **Abstrakte Syntax** beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck die **Abstrakte Syntax** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Syntax** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Redeart** wurde aus der **Quelle** G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

Das Ausgeben eines **Abstract Syntax Trees** wird in Python über die **Magische Methode** `__repr__()`² umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine Textrepräsentation seiner selbst und all seiner Knoten mit an den richtigen Stellen passenden gesetzten **runden öffnenden** (und **schließenden**) **Klammern**, sowie **Kommas** , und **Semikolons** ; zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgeben**.

1.2.5.5 Transformer

1.2.5.6 Codebeispiel

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

```

1 File
2   Name './example_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('5'), Num('6')],
8           ↪ PtrDecl(Num('1'), IntType('int')))), Name('attr'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3'), Num('2')],
16          ↪ PtrDecl(Num('1'), StructSpec(Name('st'))))), Name('var'))
17      ]
18  ]

```

Code 1.4: *Abstract Syntax Tree aus vereinfachtem Derivation Tree generiert*

1.3 Code Generierung

1.3.1 Übersicht

Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 1.2, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **Container-Knoten** und **Token-Knoten** im **Abstract Syntax Tree** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstract Syntax Tree** der Sprache L_{PicoC} , der durch die **Abstrakte Syntax** in Grammatik 1.2.3 spezifiziert ist in einen entsprechenden **Abstract Syntax Tree** der Sprache L_{RETI} umzuformen. Das ganze lässt sich, wie in Unterkapitel ?? bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition ??) herunterbricht.

²Spezielle Methode, die immer aufgerufen wird, wenn das **Object**, dass in Besitz dieser Methode ist als **String** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.

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

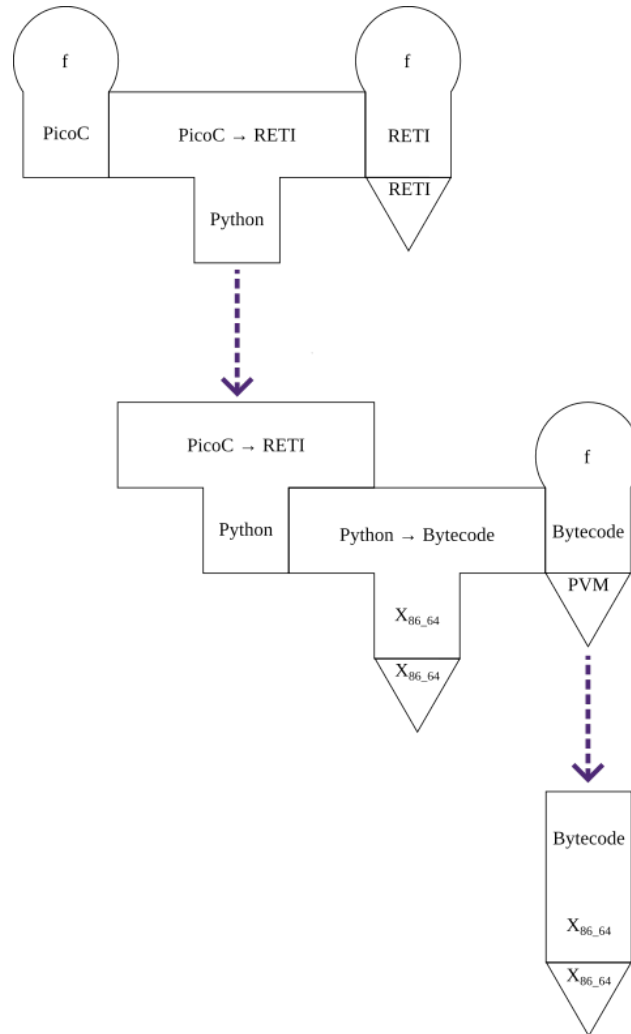


Abbildung 1.1: *Cross-Compiler Kompiliervorgang ausgeschrieben*

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

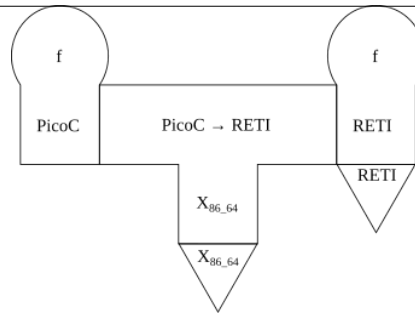


Abbildung 1.2: Cross-Compiler Kompilervorgang Kurzform

Nachdem der Kompilierprozess des **PicoC-Compiler** im **vertikalen** nun genauer angesehen wurde, wird der Kompilierprozess im Folgenden im **horizontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 1.3 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (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.

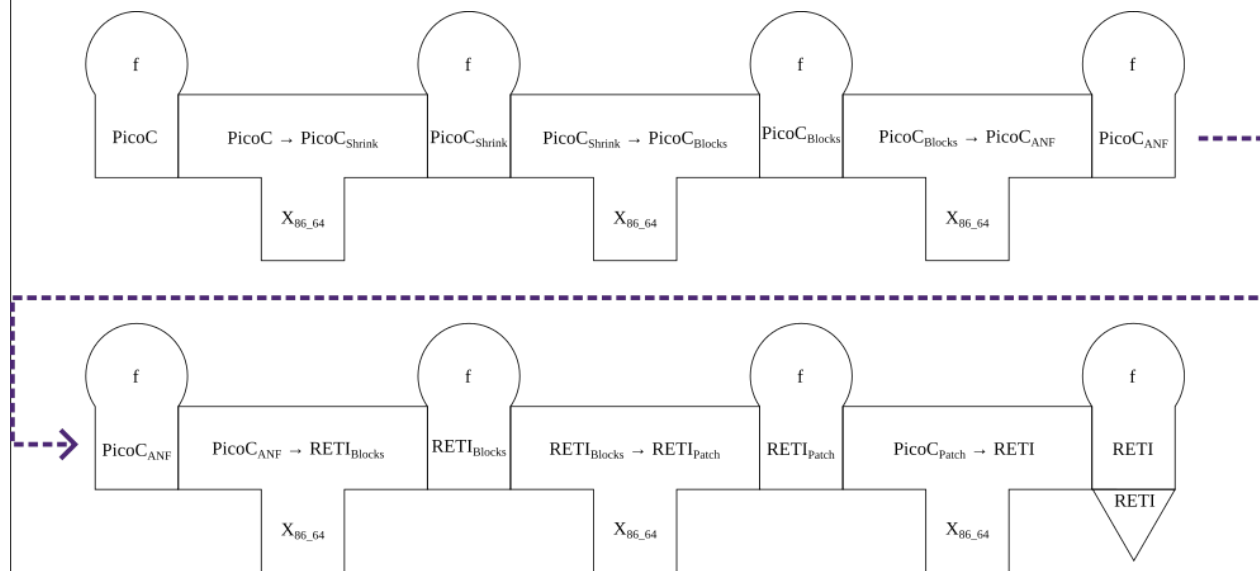


Abbildung 1.3: Architektur mit allen Passes ausgeschrieben

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

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird³. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und

³Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

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**⁴ finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind⁵.

1.3.2 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der verschiedenen Aspekte⁶ der **Bachelorarbeit** wichtig.

Auf jedes Detail der einzelnen **Passes** wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** im Detail erklärt sind und andererseits viele Aufgaben dieser **Passes** eher dem **Bachelorprojekt** zuzurechnen sind.

1.3.2.1 PicoC-Shrink Pass

1.3.2.1.1 Aufgabe

Der Aufgabe des **PicoC-Shrink Pass** ist in Unterkapitel ?? ausführlich an einem Beispiel erklärt. Kurzgefasst hat der **PicoC-Shrink Pass** die Aufgabe, die Eigenheit auszunutzen, dass der **Dereferenzierungoperator** `*pntr` und die damit einhergehende **Pointer Arithmetik** `*(pntr + i)` sich in der Untermenge der Sprache L_C , welche die Sprache L_{PicoC} darstellt genau gleich verhält, wie der **Operator** für den **Zugriff** auf den **Index** eines **Arrays** `ar[i]`.

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** `Deref(exp, i)` im jeweiligen **Abstract Syntax Tree** in **Knoten** `Subscr(exp, i)` um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung sparen lassen, denn man kann die **Dereferenzierung** `*(var + i)` einfach von den Routinen für einen **Zugriff auf einen Arrayindex** `var[i]` übernehmen lassen.

1.3.2.1.2 Abstrakte Syntax

⁴https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples.

⁵Es 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.

⁶In kurz: **Pointer**, **Arrays**, **Structs** und **Funktionen**.

<i>stmt</i>	::=	<i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp</i> (<i><exp></i> , <i><bin_op></i> , <i><exp></i>) <i>UnOp</i> (<i><un_op></i> , <i><exp></i>) <i>Call</i> (<i>Name('input')</i> , <i>Empty()</i>)	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom</i> (<i><exp></i> , <i><rel></i> , <i><exp></i>) <i>ToBool</i> (<i><exp></i>)	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>)	
<i>stmt</i>	::=	<i>Assign</i> (<i><exp></i> , <i><exp></i>) <i>Exp</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num(str)</i> , <i><datatype></i>)	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref</i> (<i><exp></i> , <i><exp></i>) <i>Ref</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num(str)</i> +, <i><datatype></i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i><exp></i> , <i><exp></i>) <i>Array</i> (<i><exp></i> +))	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name(str)</i>)	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i><exp></i> , <i>Name(str)</i>) <i>StructAssign</i> (<i>Name(str)</i> , <i><exp></i>)+)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)+)	
<i>stmt</i>	::=	<i>If</i> (<i><exp></i> , <i><stmt></i> *) <i>IfElse</i> (<i><exp></i> , <i><stmt></i> *, <i><stmt></i> *)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> (<i><exp></i> , <i><stmt></i> *) <i>DoWhile</i> (<i><exp></i> , <i><stmt></i> *)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> (<i>Name(str)</i> , <i><exp></i> *)	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return</i> (<i><exp></i>)	
<i>decl_def</i>	::=	<i>FunDecl</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*) <i>FunDef</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>)*, <i><stmt></i> *)	
<i>file</i>	::=	<i>File</i> (<i>Name(str)</i> , <i><decl_def></i> *)	<i>L_File</i>

Grammar 1.3.1: Abstrakte Syntax der Sprache *L_{PiocShrink}*

Die **Abstrakte Syntax** der Sprache *L_{PiocShrink}* ist **identisch** mit der **Abstrakten Syntax** der Sprache *L_{Pioc}* aus Tabelle 1.2.3, nach welcher der **erste** Abstract Syntax Tree in der **Syntaktischen Analyse** generiert wurde. Das liegt daran, dass dieser Pass nur alle **Vorkommnisse** eines Knoten **Deref**(*exp*, *i*) durch den Knoten **Subscr**(*exp*, *i*) auswechselt, der ebenfalls bereits in der **Abstrakten Syntax** der Sprache *L_{Pioc}* definiert ist.

1.3.2.1.3 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 1.5 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 1.5 ist in der Funktion **faculty** ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen **Arguments** berechnet. Der Algorithmus basiert auf einem **Beispielprogramm** aus der

Vorlesung Scholl, „Betriebssysteme“, der in der Vorlesung allerdings **rekursiv** implementiert war.

Dieser **rekursive** Algorithmus ist allerdings **kein** gutes **Anschauungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von **if**-, **if-else**-, **while**- und **do-while**-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 1.5 umgeschrieben, um **if**- und **while**-Statemens zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter [Link⁷](#), unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** mit eigenen **Codebeispielen** erklärt werden und alle sonstigen Details dem **Bachelorprojekt** zuzurechnen sind.

```

1 // based on a example program from Christoph Scholl's Operating Systems lecture
2
3 int faculty(int n){
4     int res = 1;
5     while (1) {
6         if (n == 1) {
7             return res;
8         }
9         res = n * res;
10        n = n - 1;
11    }
12 }
13
14 void main() {
15     print(faculty(4));
16 }
```

Code 1.5: PicoC Code für Codebespiel

In Code 1.6 sieht man den **Abstract Syntax Tree**, der in der **Syntaktischen Analyse** generiert wurde.

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
10    ]
```

⁷https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.


```

11     Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
12     While
13         Num '1',
14         [
15             If
16                 Atom(Name('n'), Eq('=='), Num('1')),
17                 [
18                     Return(Name('res'))
19                 ]
20                 Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21                 Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22             ]
23         ],
24     FunDef
25         VoidType 'void',
26         Name 'main',
27         [],
28         [
29             Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30         ]
31 ]

```

Code 1.6: Abstract Syntax Tree für Codebeispiel

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstract Syntax Tree** in Code 1.6, da das Codebeispiel keine **Dereferenzierung** enthält.

1.3.2.2 PicoC-Blocks Pass

1.3.2.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** ist die Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs)`, `GoTo(label)`- und `IfElse(exp, stmts1, stmts2)`-Knoten umzusetzen. Der `IfElse(exp, stmts1, stmts2)`-Knoten wird zur Umsetzung der **Bedingung** verwendet und es wird, je nachdem, ob die Bedingung **wahr** oder **falsch** ist mithilfe der `GoTo(label)`-Knoten in einen von zwei **alternativen Branches** gesprungen oder ein **Branch** erneut aufgerufen usw.

1.3.2.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax 1.2.3** um die Knoten zu erweitern, die im Unterkapitel 1.3.2.2.1 erwähnt wurden. Des Weiteren wird für die **Kommentare**, die in vielen Codebeispielen zur leichten Verständlichkeit eingefügt wurden ein `SingleLineComment(prefix, content)`-Knoten benötigt. Die **Funktionsdefinition** `FunDef(<datatype>, Name(str), Alloc(Writable(), <datatype>, Name(str))* , <block>*)` ist nun ein Container für **Blöcke** `Block(Name(str), <stmt>*)` und keine Statements `stmt` mehr.

<i>stmt</i>	::=	<i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>BinOp</i> (<i><exp></i> , <i><bin_op></i> , <i><exp></i>) <i>UnOp</i> (<i><un_op></i> , <i><exp></i>) <i>Call</i> (<i>Name('input')</i> , <i>Empty()</i>)	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom</i> (<i><exp></i> , <i><rel></i> , <i><exp></i>) <i>ToBool</i> (<i><exp></i>)	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc</i> (<i><type_qual></i> , <i><datatype></i> , <i>Name(str)</i>)	
<i>stmt</i>	::=	<i>Assign</i> (<i><exp></i> , <i><exp></i>) <i>Exp</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>PntrDecl</i> (<i>Num(str)</i> , <i><datatype></i>)	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Ref</i> (<i><exp></i>)	
<i>datatype</i>	::=	<i>ArrayDecl</i> (<i>Num(str)+</i> , <i><datatype></i>)	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr</i> (<i><exp></i> , <i><exp></i>) <i>Array</i> (<i><exp>+</i>)	
<i>datatype</i>	::=	<i>StructSpec</i> (<i>Name(str)</i>)	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr</i> (<i><exp></i> , <i>Name(str)</i>) <i>StructAssign</i> (<i>Name(str)</i> , <i><exp>+</i>)	
<i>decl_def</i>	::=	<i>StructDecl</i> (<i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>) <i>+</i>)	
<i>stmt</i>	::=	<i>If</i> (<i><exp></i> , <i><stmt>*</i>) <i>IfElse</i> (<i><exp></i> , <i><stmt>*</i> , <i><stmt>*</i>)	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While</i> (<i><exp></i> , <i><stmt>*</i>) <i>DoWhile</i> (<i><exp></i> , <i><stmt>*</i>)	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call</i> (<i>Name(str)</i> , <i><exp>*</i>)	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return</i> (<i><exp></i>)	
<i>decl_def</i>	::=	<i>FunDecl</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>) <i>*)</i> <i>FunDef</i> (<i><datatype></i> , <i>Name(str)</i> , <i>Alloc</i> (<i>Writeable()</i> , <i><datatype></i> , <i>Name(str)</i>) <i>*, <block>*</i>)	
<i>block</i>	::=	<i>Block</i> (<i>Name(str)</i> , <i><stmt>*</i>)	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo</i> (<i>Name(str)</i>) <i>SingleLineComment</i> (<i>str</i> , <i>str</i>)	
<i>file</i>	::=	<i>File</i> (<i>Name(str)</i> , <i><decl_def>*</i>)	<i>L_File</i>

Grammar 1.3.2: Abstrakte Syntax der Sprache *L_{Pioc}Blocks*

Die Abstrakte Syntax ist im Gegensatz zur Konkretten Syntax nur vom Programmierer verstanden werden, wenn man nicht darauf abzielt

Man bezeichnet hier die Abstrakte Syntax als „Abstrakte Syntax der Sprache *L_{Pioc}Blocks*“. Diese Sprache *L_{Pioc}Blocks* besitzt eine **Konkrete Syntax** und

1.3.2.2.3 Codebeispiel

In Code 1.7 sieht man den **Abstract-Syntax-Tree** des **PiocC-Blocks Passes** für das aus Unterkapitel 1.5 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion **faculty** und die **main**-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement oder bis zum ersten **Auftauchen** eines **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-**Knoten stehen. Je nachdem, ob ein **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-**Knoten auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```

1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writeable(), IntType('int'), Name('n'))
9       ],
10      [
11        Block
12          Name 'faculty.6',
13          [
14            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
15            // While(Num('1'), [])
16            GoTo(Name('condition_check.5'))
17          ],
18        Block
19          Name 'condition_check.5',
20          [
21            IfElse
22              Num '1',
23              [
24                GoTo(Name('while_branch.4'))
25              ],
26              [
27                GoTo(Name('while_after.1'))
28              ]
29          ],
30        Block
31          Name 'while_branch.4',
32          [
33            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
34            IfElse
35              Atom(Name('n'), Eq('=='), Num('1')),
36              [
37                GoTo(Name('if.3'))
38              ],
39              [
40                GoTo(Name('if_else_after.2'))
41              ]
42          ],
43        Block
44          Name 'if.3',
45          [
46            Return(Name('res'))

```

```
47     ],
48     Block
49       Name 'if_else_after.2',
50       [
51         Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52         Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53         GoTo(Name('condition_check.5'))
54       ],
55     Block
56       Name 'while_after.1',
57       []
58   ],
59   FunDef
60     VoidType 'void',
61     Name 'main',
62     [],
63     [
64       Block
65         Name 'main.0',
66         [
67           Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
68         ]
69     ]
70 ]
```

Code 1.7: *PicoC-Blocks Pass für Codebespiel*

1.3.2.3 PicoC-ANF Pass

1.3.2.3.1 Aufgabe

1.3.2.3.2 Abstrakte Syntax

<i>stmt</i>	::=	<i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i> <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i> <i>Sub()</i> <i>Mul()</i> <i>Div()</i> <i>Mod()</i> <i>Oplus()</i> <i>And()</i> <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i> <i>Num(str)</i> <i>Char(str)</i> <i>Global(Num(str))</i> <i>Stackframe(Num(str))</i> <i>Stack(Num(str))</i> <i>BinOp(Stack(Num(str)), <bin_op>, Stack(Num(str)))</i> <i>UnOp(<un_op>, Stack(Num(str)))</i> <i>Call(Name('input'), Empty())</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i> <i>NEq()</i> <i>Lt()</i> <i>LtE()</i> <i>Gt()</i> <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i> <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(Stack(Num(str)), <rel>, Stack(Num(str)))</i> <i>ToBool(Stack(Num(str)))</i>	
<i>type_qual</i>	::=	<i>Const()</i> <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i> <i>CharType()</i> <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(<type_qual>, <datatype>, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(<exp>, <exp>)</i> <i>Exp(<exp>)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), <datatype>)</i> <i>Ref(<exp>)</i>	<i>L_Pntr</i>
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, <datatype>)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(<exp>, Stack(Num(str)))</i> <i>Array(<exp>+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(<exp>, Name(str))</i> <i>Struct(Assign(Name(str), <exp>)+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), <datatype>, Name(str))+)</i>	
<i>stmt</i>	::=	<i>IfElse(Stack(Num(str)), <stmt>*, <stmt>*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>StackMalloc(Num(str))</i> <i>Ref(Global(Num(str)))</i> <i>NewStackframe(Name(str), GoTo(str))</i> <i>Exp(GoTo(Name(str)))</i> <i>RemoveStackframe()</i> <i>Return(Empty())</i> <i>Return(<exp>)</i>	<i>L_Fun</i>
<i>block</i>	::=	<i>Block(Name(str), <stmt>*)</i>	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo(Name(str))</i> <i>SingleLineComment(str, str)</i>	
<i>file</i>	::=	<i>File(Name(str), <decl_def>*)</i>	<i>L_File</i>
<i>symbol_table</i>	::=	<i>SymbolTable(<symbol>*)</i>	<i>L_Symbol_Table</i>
<i>symbol</i>	::=	<i>Symbol(<type_qual>, <datatype>, <name>, <val>, <pos>, <size>)</i>	
<i>type_qual</i>	::=	<i>Empty()</i>	
<i>datatype</i>	::=	<i>BuiltIn()</i> <i>SelfDefined()</i>	
<i>name</i>	::=	<i>Name(str)</i>	
<i>val</i>	::=	<i>Num(str)</i> <i>Empty()</i>	
<i>pos</i>	::=	<i>Pos(Num(str), Num(str))</i> <i>Empty()</i>	
<i>size</i>	::=	<i>Num(str)</i> <i>Empty()</i>	
Grammar 1.3.3: Abstrakte Syntax der Sprache L_{Pioc_ANF}			
Definition 1.4: Symboltabelle			

1.3.2.3.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         // Assign(Name('res'), Num('1'))
8         Exp(Num('1'))
9         Assign(Stackframe(Num('1')), Stack(Num('1')))
10        // While(Num('1'), [])
11        Exp(GoTo(Name('condition_check.5')))
12      ],
13    Block
14      Name 'condition_check.5',
15      [
16        // IfElse(Num('1'), [], [])
17        Exp(Num('1')),
18        IfElse
19          Stack
20            Num '1',
21            [
22              GoTo(Name('while_branch.4'))
23            ],
24            [
25              GoTo(Name('while_after.1'))
26            ]
27        ],
28      Block
29        Name 'while_branch.4',
30        [
31          // If(Atom(Name('n'), Eq('=='), Num('1')), [])
32          // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
33          Exp(Stackframe(Num('0')))
34          Exp(Num('1'))
35          Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
36          IfElse
37            Stack
38              Num '1',
39              [
40                GoTo(Name('if.3'))
41              ],
42              [
43                GoTo(Name('if_else_after.2'))
44              ]
45          ],
46        Block
47          Name 'if.3',
48          [
49            // Return(Name('res'))
50            Exp(Stackframe(Num('1')))
51            Return(Stack(Num('1')))
52          ],
53        Block
54          Name 'if_else_after.2',
55          [

```

```

56      // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
57      Exp(Stackframe(Num('0')))
58      Exp(Stackframe(Num('1')))
59      Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
60      Assign(Stackframe(Num('1')), Stack(Num('1')))
61      // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
62      Exp(Stackframe(Num('0')))
63      Exp(Num('1'))
64      Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
65      Assign(Stackframe(Num('0')), Stack(Num('1')))
66      Exp(GoTo(Name('condition_check.5')))
67  ],
68  Block
69    Name 'while_after.1',
70    [
71      Return(Empty())
72    ],
73  Block
74    Name 'main.0',
75    [
76      StackMalloc(Num('2'))
77      Exp(Num('4'))
78      NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
79      Exp(GoTo(Name('faculty.6')))
80      RemoveStackframe()
81      Exp(ACC)
82      Exp(Call(Name('print'), [Stack(Num('1'))]))
83      Return(Empty())
84    ]
85  ]

```

Code 1.8: PicoC-ANF Pass für Codebeispiel

1.3.2.4 RETI-Blocks Pass

1.3.2.4.1 Aufgaben

1.3.2.4.2 Abstrakte Syntax

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

Grammar 1.3.4: 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 'faculty.6',
6       [
7         # // Assign(Name('res'), Num('1'))
8         # Exp(Num('1'))
9         SUBI SP 1;
10        LOADI ACC 1;
11        STOREIN SP ACC 1;
12        # Assign(Stackframe(Num('1')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN BAF ACC -3;
15        ADDI SP 1;
16        # // While(Num('1'), [])
17        # Exp(GoTo(Name('condition_check.5')))
18        Exp(GoTo(Name('condition_check.5')))
19      ],
20    Block
21      Name 'condition_check.5',
22      [
23        # // IfElse(Num('1'), [], [])
24        # Exp(Num('1'))
25        SUBI SP 1;
26        LOADI ACC 1;
27        STOREIN SP ACC 1;
28        # IfElse(Stack(Num('1')), [], [])
29        LOADIN SP ACC 1;
30        ADDI SP 1;
31        JUMP== GoTo(Name('while_after.1'));
32        Exp(GoTo(Name('while_branch.4')))
33      ],
34    Block
35      Name 'while_branch.4',
36      [
37        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
38        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39        # Exp(Stackframe(Num('0')))
40        SUBI SP 1;
41        LOADIN BAF ACC -2;
42        STOREIN SP ACC 1;
43        # Exp(Num('1'))
44        SUBI SP 1;
45        LOADI ACC 1;
46        STOREIN SP ACC 1;
47        LOADIN SP ACC 2;
48        LOADIN SP IN2 1;
49        SUB ACC IN2;
50        JUMP== 3;
51        LOADI ACC 0;
52        JUMP 2;
53        LOADI ACC 1;
54        STOREIN SP ACC 2;
55        ADDI SP 1;
56        # IfElse(Stack(Num('1')), [], [])
57        LOADIN SP ACC 1;

```



```

58     ADDI SP 1;
59     JUMP== GoTo(Name('if_else_after.2'));
60     Exp(GoTo(Name('if.3')))
61 ],
62 Block
63     Name 'if.3',
64     [
65         # // Return(Name('res'))
66         # Exp(Stackframe(Num('1')))
67         SUBI SP 1;
68         LOADIN BAF ACC -3;
69         STOREIN SP ACC 1;
70         # Return(Stack(Num('1')))
71         LOADIN SP ACC 1;
72         ADDI SP 1;
73         LOADIN BAF PC -1;
74     ],
75 Block
76     Name 'if_else_after.2',
77     [
78         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
79         # Exp(Stackframe(Num('0')))
80         SUBI SP 1;
81         LOADIN BAF ACC -2;
82         STOREIN SP ACC 1;
83         # Exp(Stackframe(Num('1')))
84         SUBI SP 1;
85         LOADIN BAF ACC -3;
86         STOREIN SP ACC 1;
87         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88         LOADIN SP ACC 2;
89         LOADIN SP IN2 1;
90         MULT ACC IN2;
91         STOREIN SP ACC 2;
92         ADDI SP 1;
93         # Assign(Stackframe(Num('1')), Stack(Num('1')))
94         LOADIN SP ACC 1;
95         STOREIN BAF ACC -3;
96         ADDI SP 1;
97         # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98         # Exp(Stackframe(Num('0')))
99         SUBI SP 1;
100        LOADIN BAF ACC -2;
101        STOREIN SP ACC 1;
102        # Exp(Num('1'))
103        SUBI SP 1;
104        LOADI ACC 1;
105        STOREIN SP ACC 1;
106        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107        LOADIN SP ACC 2;
108        LOADIN SP IN2 1;
109        SUB ACC IN2;
110        STOREIN SP ACC 2;
111        ADDI SP 1;
112        # Assign(Stackframe(Num('0')), Stack(Num('1')))
113        LOADIN SP ACC 1;
114        STOREIN BAF ACC -2;

```

```

115     ADDI SP 1;
116     # Exp(GoTo(Name('condition_check.5')))
117     Exp(GoTo(Name('condition_check.5')))
118 ],
119 Block
120   Name 'while_after.1',
121   [
122     # Return(Empty())
123     LOADIN BAF PC -1;
124   ],
125 Block
126   Name 'main.0',
127   [
128     # StackMalloc(Num('2'))
129     SUBI SP 2;
130     # Exp(Num('4'))
131     SUBI SP 1;
132     LOADI ACC 4;
133     STOREIN SP ACC 1;
134     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
135     MOVE BAF ACC;
136     ADDI SP 3;
137     MOVE SP BAF;
138     SUBI SP 4;
139     STOREIN BAF ACC 0;
140     LOADI ACC GoTo(Name('addr@next_instr'));
141     ADD ACC CS;
142     STOREIN BAF ACC -1;
143     # Exp(GoTo(Name('faculty.6')))
144     Exp(GoTo(Name('faculty.6')))
145     # RemoveStackframe()
146     MOVE BAF IN1;
147     LOADIN IN1 BAF 0;
148     MOVE IN1 SP;
149     # Exp(ACC)
150     SUBI SP 1;
151     STOREIN SP ACC 1;
152     LOADIN SP ACC 1;
153     ADDI SP 1;
154     CALL PRINT ACC;
155     # Return(Empty())
156     LOADIN BAF PC -1;
157   ]
158 ]

```

Code 1.9: RETI-Blocks Pass für Codebespiel

1.3.2.5 RETI-Patch Pass

1.3.2.5.1 Aufgaben

1.3.2.5.2 Abstrakte Syntax

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

Grammar 1.3.5: Abstrakte Syntax für L_{RETI_Patch}

1.3.2.5.3 Codebeispiel

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.7',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'faculty.6',
12      [
13        # // Assign(Name('res'), Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Assign(Stackframe(Num('1')), Stack(Num('1')))
19        LOADIN SP ACC 1;
20        STOREIN BAF ACC -3;
21        ADDI SP 1;
22        # // While(Num('1'), [])
23        # Exp(GoTo(Name('condition_check.5')))
24        # // not included Exp(GoTo(Name('condition_check.5')))
25      ],
26    Block
27      Name 'condition_check.5',
28      [
29        # // IfElse(Num('1'), [], [])
30        # Exp(Num('1'))
31        SUBI SP 1;
32        LOADI ACC 1;
33        STOREIN SP ACC 1;
34        # IfElse(Stack(Num('1')), [], [])
35        LOADIN SP ACC 1;
36        ADDI SP 1;
37        JUMP== GoTo(Name('while_after.1'));
38        # // not included Exp(GoTo(Name('while_branch.4')))
39      ],
40    Block
41      Name 'while_branch.4',
42      [
43        # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
44        # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
45        # Exp(Stackframe(Num('0')))
46        SUBI SP 1;
47        LOADIN BAF ACC -2;
48        STOREIN SP ACC 1;
49        # Exp(Num('1'))
50        SUBI SP 1;
51        LOADI ACC 1;
52        STOREIN SP ACC 1;
53        LOADIN SP ACC 2;
54        LOADIN SP IN2 1;
55        SUB ACC IN2;

```

```

56     JUMP== 3;
57     LOADI ACC 0;
58     JUMP 2;
59     LOADI ACC 1;
60     STOREIN SP ACC 2;
61     ADDI SP 1;
62     # IfElse(Stack(Num('1')), [], [])
63     LOADIN SP ACC 1;
64     ADDI SP 1;
65     JUMP== GoTo(Name('if_else_after.2'));
66     # // not included Exp(GoTo(Name('if.3')))
67 ],
68 Block
69     Name 'if.3',
70     [
71         # // Return(Name('res'))
72         # Exp(Stackframe(Num('1')))
73         SUBI SP 1;
74         LOADIN BAF ACC -3;
75         STOREIN SP ACC 1;
76         # Return(Stack(Num('1')))
77         LOADIN SP ACC 1;
78         ADDI SP 1;
79         LOADIN BAF PC -1;
80     ],
81 Block
82     Name 'if_else_after.2',
83     [
84         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85         # Exp(Stackframe(Num('0')))
86         SUBI SP 1;
87         LOADIN BAF ACC -2;
88         STOREIN SP ACC 1;
89         # Exp(Stackframe(Num('1')))
90         SUBI SP 1;
91         LOADIN BAF ACC -3;
92         STOREIN SP ACC 1;
93         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94         LOADIN SP ACC 2;
95         LOADIN SP IN2 1;
96         MULT ACC IN2;
97         STOREIN SP ACC 2;
98         ADDI SP 1;
99         # Assign(Stackframe(Num('1')), Stack(Num('1')))
100        LOADIN SP ACC 1;
101        STOREIN BAF ACC -3;
102        ADDI SP 1;
103        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104        # Exp(Stackframe(Num('0')))
105        SUBI SP 1;
106        LOADIN BAF ACC -2;
107        STOREIN SP ACC 1;
108        # Exp(Num('1'))
109        SUBI SP 1;
110        LOADI ACC 1;
111        STOREIN SP ACC 1;
112        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))

```

```

113     LOADIN SP ACC 2;
114     LOADIN SP IN2 1;
115     SUB ACC IN2;
116     STOREIN SP ACC 2;
117     ADDI SP 1;
118     # Assign(Stackframe(Num('0')), Stack(Num('1')))
119     LOADIN SP ACC 1;
120     STOREIN BAF ACC -2;
121     ADDI SP 1;
122     # Exp(GoTo(Name('condition_check.5')))
123     Exp(GoTo(Name('condition_check.5')))
124 ],
125 Block
126   Name 'while_after.1',
127   [
128     # Return(Empty())
129     LOADIN BAF PC -1;
130   ],
131 Block
132   Name 'main.0',
133   [
134     # StackMalloc(Num('2'))
135     SUBI SP 2;
136     # Exp(Num('4'))
137     SUBI SP 1;
138     LOADI ACC 4;
139     STOREIN SP ACC 1;
140     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
141     MOVE BAF ACC;
142     ADDI SP 3;
143     MOVE SP BAF;
144     SUBI SP 4;
145     STOREIN BAF ACC 0;
146     LOADI ACC GoTo(Name('addr@next_instr'));
147     ADD ACC CS;
148     STOREIN BAF ACC -1;
149     # Exp(GoTo(Name('faculty.6')))
150     Exp(GoTo(Name('faculty.6')))
151     # RemoveStackframe()
152     MOVE BAF IN1;
153     LOADIN IN1 BAF 0;
154     MOVE IN1 SP;
155     # Exp(ACC)
156     SUBI SP 1;
157     STOREIN SP ACC 1;
158     LOADIN SP ACC 1;
159     ADDI SP 1;
160     CALL PRINT ACC;
161     # Return(Empty())
162     LOADIN BAF PC -1;
163   ]
164 ]

```

Code 1.10: RETI-Patch Pass für Codebeispiel

1.3.2.6 RETI Pass**1.3.2.6.1 Aufgaben****1.3.2.6.2 Konkrete und Abstrakte Syntax**

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

Grammar 1.3.6: Konkrete Syntax für *L_{RETI_Lex}*

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

Grammar 1.3.7: Konkrete Syntax für *L_{RETI_Parse}*

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

Grammar 1.3.8: Abstrakte Syntax für *L_{RETI}*

1.3.2.6.3 Codebeispiel

```

1 # // Exp(GoTo(Name('main.0'))))
2 JUMP 67;
3 # // Assign(Name('res'), Num('1'))
4 # Exp(Num('1'))
5 SUBI SP 1;
6 LOADI ACC 1;
7 STOREIN SP ACC 1;
8 # Assign(Stackframe(Num('1')), Stack(Num('1')))
9 LOADIN SP ACC 1;
10 STOREIN BAF ACC -3;
11 ADDI SP 1;
12 # // While(Num('1'), [])
13 # Exp(GoTo(Name('condition_check.5'))))
14 # // not included Exp(GoTo(Name('condition_check.5'))))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4'))))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0'))))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;

```

```
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;
43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3;
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;
73 # Assign(Stackframe(Num('1')), Stack(Num('1')))
74 LOADIN SP ACC 1;
75 STOREIN BAF ACC -3;
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
```



```
92 # Assign(Stackframe(Num('0')), Stack(Num('1'))))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5'))))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;
100 # StackMalloc(Num('2'))
101 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr'))))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6'))))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

Code 1.11: *RETI Pass für Codebespiel*

Appendix

Konkrete und Abstrakte Syntax

Bedienungsanleitungen

PicoC-Compiler

Showmode

Entwicklertools

Danksagungen

Literatur

Online

- *C Operator Precedence* - *cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27.04.2022).

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).