

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

---

*Abgabedatum:* 13. September 2022

*Autor:*

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.

# Danksagungen

Bevor der Inhalt dieser Schriftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie „mein aufrichtigster Dank“ oder „aus tiefstem Herzen“, sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbstverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das hinaus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschaft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias<sup>1</sup> konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe<sup>2</sup>, er hat sich bei der Korrektur dieser Schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbstverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dageben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

---

<sup>1</sup>Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

<sup>2</sup>Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil<sup>3 4</sup> weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbstverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler auseinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Misst baue. Der RETI-Emulator von Michel Giehl ist unter [Link](#)<sup>5</sup> zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein persönliches Interessengebiet fällt<sup>6</sup>. Das Aufschreiben dieser Schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht<sup>7</sup>. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiersprache kann in der Verwendung ziemlich willkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

---

<sup>3</sup>Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

<sup>4</sup>Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes 🤔.

<sup>5</sup><https://github.com/michel-giehl/Reti-Emulator>.

<sup>6</sup>Womit nicht alle Studenten so viel Glück haben.

<sup>7</sup>Dieses ständige überlegen, wo man möglicherweise eine Erklärücke hat, ob man nicht was wichtiges ausgelassen hat usw.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>I</b>
<b>Codeverzeichnis</b>	<b>II</b>
<b>Tabellenverzeichnis</b>	<b>III</b>
<b>Definitionsverzeichnis</b>	<b>IV</b>
<b>Grammatikverzeichnis</b>	<b>V</b>
0.0.1  Umsetzung von Funktionen . . . . .	1
0.0.1.1  Mehrere Funktionen . . . . .	1
0.0.1.1.1  Sprung zur Main Funktion . . . . .	6
0.0.1.2  Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen	9
0.0.1.3  Funktionsaufruf . . . . .	11
0.0.1.3.1  Rückgabewert . . . . .	20
0.0.1.3.2  Umsetzung der Übergabe eines Feldes . . . . .	25
0.0.1.3.3  Umsetzung der Übergabe eines Verbundes . . . . .	29
<b>Literatur</b>	<b>A</b>

# Abbildungsverzeichnis

1	Veranschaulichung der Distanzberechnung . . . . .	19
---	---	----

# Codeverzeichnis

0.1	PicoC-Code für 3 Funktionen. . . . .	1
0.2	Abstrakter Syntaxbaum für 3 Funktionen. . . . .	2
0.3	PicoC-Blocks Pass für 3 Funktionen. . . . .	3
0.4	PicoC-ANF Pass für 3 Funktionen. . . . .	4
0.5	RETI-Blocks Pass für 3 Funktionen. . . . .	6
0.6	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . . .	6
0.7	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . .	7
0.8	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . .	8
0.9	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist. . . . .	8
0.10	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss. . . . .	9
0.11	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss. . . . .	11
0.12	PicoC-Code für Funktionsaufruf ohne Rückgabewert. . . . .	11
0.13	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert. . . . .	12
0.14	Symboltabelle für Funktionsaufruf ohne Rückgabewert. . . . .	15
0.15	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert. . . . .	17
0.16	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert. . . . .	18
0.17	RETI-Pass für Funktionsaufruf ohne Rückgabewert. . . . .	20
0.18	PicoC-Code für Funktionsaufruf mit Rückgabewert. . . . .	21
0.19	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert. . . . .	21
0.20	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert. . . . .	23
0.21	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert. . . . .	25
0.22	PicoC-Code für die Übergabe eines Feldes. . . . .	25
0.23	Symboltabelle für die Übergabe eines Feldes. . . . .	27
0.24	PicoC-ANF Pass für die Übergabe eines Feldes. . . . .	28
0.25	RETI-Block Pass für die Übergabe eines Feldes. . . . .	29
0.26	PicoC-Code für die Übergabe eines Verbundes. . . . .	30
0.27	PicoC-ANF Pass für die Übergabe eines Verbundes. . . . .	31
0.28	RETI-Block Pass für die Übergabe eines Verbundes. . . . .	32

# Tabellenverzeichnis

1	Datensegment mit Stackframe. . . . .	13
2	Aufbau Stackframe . . . . .	13
3	Knoten für Funktionsaufruf. . . . .	16



# Definitionsverzeichnis

0.1 Stackframe . . . . . 13

# Grammatikverzeichnis

## 0.0.1 Umsetzung von Funktionen

Um die **Umsetzung** von **Funktionen** zu verstehen, ist es erstmal wichtig zu verstehen, wie **Funktionen** später im **RETI-Code** aussehen (Unterkapitel 0.0.1.1), wie Funktionen **deklariert** (Definition ??) und **definiert** (Definition ??) werden können und hierbei **Sichtbarkeitsbereiche** (Definition ??) umgesetzt sind (Unterkapitel 0.0.1.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines **Funktionsaufrufes** erklärt werden (Unterkapitel 0.0.1.3). Beim Thema **Funktionsaufruf** wird im speziellen darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 0.0.1.3.1) umgesetzt sind und die **Übergabe** von **Zusammengesetzten Datentypen**, die mehr als eine Speicherzelle belegen, wie **Verbunden** (Unterkapitel 0.0.1.3.3) und **Feldern** (Unterkapitel 0.0.1.3.2) umgesetzt ist.

### 0.0.1.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 0.1 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** übersetzt werden.

```

1 void main() {
2     return;
3 }
4
5 void fun1() {
6     int var = 41;
7     if(1) {
8         var = 42;
9     }
10 }
11
12 int fun2() {
13     return 1;
14 }
```

Code 0.1: PicoC-Code für 3 Funktionen.

Im **Abstrakten Syntaxbaum** in Code 0.2 werden die 3 **Funktionen** durch entsprechende Knoten dargestellt. Am Beispiel der **Funktion** `void fun2() {return 1;}` wären die hierzu passenden **Knoten** `FunDef(VoidType(), Name('fun2'), [], [Return(Num('1'))])`.<sup>8</sup>

```

1 File
2   Name './verbose_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return
10          Empty
11      ],
12     FunDef
```

<sup>8</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

```

13     VoidType 'void',
14     Name 'fun1',
15     [],
16     [
17         Assign
18         Alloc
19         Writeable,
20         IntType 'int',
21         Name 'var',
22         Num '41',
23     If
24         Num '1',
25         [
26             Assign
27             Name 'var',
28             Num '42'
29         ]
30     ],
31     FunDef
32     IntType 'int',
33     Name 'fun2',
34     [],
35     [
36         Return
37         Num '1'
38     ]
39 ]

```

**Code 0.2:** Abstrakter Syntaxbaum für 3 Funktionen.

Im **PicoC-Blocks Pass** in Code 0.3 werden die **Anweisungen** von Funktionen in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Hierbei bekommt ein Block `Block(name, stmts_instrs)`, der die Anweisungen einer Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`<sup>9</sup> beinhaltet den **Bezeichner** bzw. den `Name(str)`-Knoten der Funktion an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** `<number>` angehängt `<name>.<number>`<sup>10,11</sup>.

Es werden parallel dazu neue Zuordnungen im **Assoziativen Feld** `fun_name_to_block_name` hinzugefügt. Das **Assoziative Feld** `fun_name_to_block_name` ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der die erste **Anweisung** der Funktion enthält zu. Der **Bezeichner** des Blockes `<name>.<number>` ist dabei bis auf die angehängte **Nummer** `<number>` identisch zu dem der Funktion. Diese Zuordnung ist nötig, da **Blöcke** eine **Nummer** `<number>` an ihren Bezeichner angehängt haben `<name>.<number>`, die auf anderem Wege **nicht** ohne großen Aufwand herausgefunden werden kann.

```

1 File
2   Name './verbose_3_funs.picoc_blocks',
3   [
4       FunDef
5       VoidType 'void',

```

<sup>9</sup>Eine **Erklärung** dazu ist in Unterkapitel ?? zu finden.

<sup>10</sup>Der **Grund** dafür kann im Unterkapitel ?? nachgelesen werden.

<sup>11</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

```

6      Name 'main',
7      [],
8      [
9          Block
10         Name 'main.4',
11         [
12             Return(Empty())
13         ]
14     ],
15     FunDef
16     VoidType 'void',
17     Name 'fun1',
18     [],
19     [
20         Block
21         Name 'fun1.3',
22         [
23             Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('41'))
24             // If(Num('1'), []),
25             IfElse
26             Num '1',
27             [
28                 GoTo
29                 Name 'if.2'
30             ],
31             [
32                 GoTo
33                 Name 'if_else_after.1'
34             ]
35         ],
36         Block
37         Name 'if.2',
38         [
39             Assign(Name('var'), Num('42'))
40             GoTo(Name('if_else_after.1'))
41         ],
42         Block
43         Name 'if_else_after.1',
44         []
45     ],
46     FunDef
47     IntType 'int',
48     Name 'fun2',
49     [],
50     [
51         Block
52         Name 'fun2.0',
53         [
54             Return(Num('1'))
55         ]
56     ]
57 ]

```

**Code 0.3:** *PicoC-Blocks Pass für 3 Funktionen.*

Im **PicoC-ANF Pass** in Code 0.4 werden die FunDef(datatype, name, allocs, stmts)-Knoten komplett

aufgelöst, sodass sich im `File(name, decls.defs.blocks)`-Knoten nur noch **Blöcke** befinden.

```

1 File
2   Name './verbose_3_funs.picoc_mon',
3   [
4     Block
5       Name 'main.4',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun1.3',
11      [
12        // Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('41'))
13        // Assign(Name('var'), Num('41'))
14        Exp(Num('41'))
15        Assign(Stackframe(Num('0')), Stack(Num('1')))
16        // If(Num('1'), [])
17        // IfElse(Num('1'), [], [])
18        Exp(Num('1')),
19        IfElse
20          Stack
21            Num '1',
22            [
23              GoTo
24                Name 'if.2'
25            ],
26            [
27              GoTo
28                Name 'if_else_after.1'
29            ]
30        ],
31      Block
32        Name 'if.2',
33        [
34          // Assign(Name('var'), Num('42'))
35          Exp(Num('42'))
36          Assign(Stackframe(Num('0')), Stack(Num('1')))
37          Exp(GoTo(Name('if_else_after.1')))
38        ],
39      Block
40        Name 'if_else_after.1',
41        [
42          Return(Empty())
43        ],
44      Block
45        Name 'fun2.0',
46        [
47          // Return(Num('1'))
48          Exp(Num('1'))
49          Return(Stack(Num('1')))
50        ]
51  ]

```

**Code 0.4:** *PicoC-ANF Pass für 3 Funktionen.*

Nach dem **RETI Pass** in Code 0.5 gibt es nur noch **RETI-Befehle**, die Blöcke wurden entfernt. Die **RETI-Befehle** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man Folgen von RETI-Befehlen **nicht** mehr direkt Funktionen zuordnen. Die **Kommentare** enthalten die **Bezeichner** <name>.<number> der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer <number>, dem **Bezeichner** einer jeweiligen **Funktion** entsprechen und somit die Anfänge von **Funktionen erkennbar** machen.

Da es in der main-Funktion keinen **Funktionsaufruf** gab, wird der Code, der nach dem **Befehl** in der **markierten Zeile** in Code 0.5 kommt nicht mehr betreten. Funktionen sind im **RETI-Code** nur dadurch existent, dass im RETI-Code **Sprünge** (z.B. JUMP<rel> <im>) zu den jeweils richtigen **Adressen** gemacht werden. Die Sprünge werden zu den Adressen gemacht, wo die Folge von **RETI-Befehlen** anfängt, die aus den **Anweisungen** einer **Funktion** kompiliert wurden.

```

1 # // Block(Name('start.5'), [])
2 # // Exp(GoTo(Name('main.4'))))
3 # // not included Exp(GoTo(Name('main.4'))))
4 # // Block(Name('main.4'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.3'), [])
8 # // Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('41'))
9 # // Assign(Name('var'), Num('41'))
10 # Exp(Num('41'))
11 SUBI SP 1;
12 LOADI ACC 41;
13 STOREIN SP ACC 1;
14 # Assign(Stackframe(Num('0')), Stack(Num('1'))))
15 LOADIN SP ACC 1;
16 STOREIN BAF ACC -2;
17 ADDI SP 1;
18 # // If(Num('1'), [])
19 # // IfElse(Num('1'), [], [])
20 # Exp(Num('1'))
21 SUBI SP 1;
22 LOADI ACC 1;
23 STOREIN SP ACC 1;
24 # IfElse(Stack(Num('1')), [], [])
25 LOADIN SP ACC 1;
26 ADDI SP 1;
27 # JUMP== GoTo(Name('if_else_after.1'));
28 JUMP== 7;
29 # GoTo(Name('if.2'))
30 # // not included Exp(GoTo(Name('if.2'))))
31 # // Block(Name('if.2'), [])
32 # // Assign(Name('var'), Num('42'))
33 # Exp(Num('42'))
34 SUBI SP 1;
35 LOADI ACC 42;
36 STOREIN SP ACC 1;
37 # Assign(Stackframe(Num('0')), Stack(Num('1'))))
38 LOADIN SP ACC 1;
39 STOREIN BAF ACC -2;
40 ADDI SP 1;
41 # Exp(GoTo(Name('if_else_after.1'))))
42 # // not included Exp(GoTo(Name('if_else_after.1'))))
43 # // Block(Name('if_else_after.1'), [])

```

```

44 # Return(Empty())
45 LOADIN BAF PC -1;
46 # // Block(Name('fun2.0'), [])
47 # // Return(Num('1'))
48 # Exp(Num('1'))
49 SUBI SP 1;
50 LOADI ACC 1;
51 STOREIN SP ACC 1;
52 # Return(Stack(Num('1'))))
53 LOADIN SP ACC 1;
54 ADDI SP 1;
55 LOADIN BAF PC -1;

```

**Code 0.5:** *RETI-Blocks Pass für 3 Funktionen.*

#### 0.0.1.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 0.1 war die `main`-Funktion die **erste** Funktion, die im Code vorkam. Dadurch konnte die `main`-Funktion direkt betreten werden, da die **Ausführung** eines Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 0.6 ist die `main`-Funktion allerdings **nicht** die **erste** Funktion. Daher muss dafür gesorgt werden, dass die `main`-Funktion die erste Funktion ist, die ausgeführt wird.

```

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

```

**Code 0.6:** *PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI-Blocks Pass** in Code 0.7 sind die **Funktionen** durch **Blöcke** umgesetzt.

```

1 File
2   Name './verbose_3_funs_main.reti_blocks',
3   [
4     Block
5       Name 'fun1.2',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block

```



```

11     Name 'fun2.1',
12     [
13         # // Return(Num('1'))
14         # Exp(Num('1'))
15         SUBI SP 1;
16         LOADI ACC 1;
17         STOREIN SP ACC 1;
18         # Return(Stack(Num('1')))
19         LOADIN SP ACC 1;
20         ADDI SP 1;
21         LOADIN BAF PC -1;
22     ],
23     Block
24     Name 'main.0',
25     [
26         # Return(Empty())
27         LOADIN BAF PC -1;
28     ]
29 ]

```

**Code 0.7:** *RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

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

Die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** allerdings entschieden wurde, war es, wenn die `main`-Funktion **nicht** die **erste** Funktion im `decls_defs`-Attribut des `File(name, decls_defs)`-Knoten ist, einen `start.<number>`-Block als ersten Block einzufügen. Dieser `start.<number>`-Block enthält einen `GoTo(Name('main.<number>'))`-Knoten, der im **RETI Pass** in Code 0.9 in einen Sprung zum **Block** der `main`-Funktion übersetzt wird.<sup>12</sup>

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für Verwender<sup>13</sup> des **PicoC-Compilers** vermutlich am **intuitivsten** ist, wenn der **RETI-Code** für die Funktionen an denselben Stellen relativ zueinander verortet ist, wie die Funktionsdefinitionen im **PicoC-Code**.

Das **Einfügen** des `start.<number>`-Blockes erfolgt im **RETI-Patch Pass** in Code 0.8. Der **RETI-Patch** Pass ist der Pass, der für das **Ausbessern**<sup>14</sup> des **Abstrakten Syntaxbaumes** zuständig ist, wenn z.B. wie hier die `main`-Funktion nicht die erste Funktion ist.

```

1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5     Name 'start.3',
6     [
7       # // Exp(GoTo(Name('main.0')))

```

<sup>12</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

<sup>13</sup>Also die kommenden **Studentengenerationen**.

<sup>14</sup>In engl. to patch.

```

8      Exp(GoTo(Name('main.0'))))
9      ],
10     Block
11       Name 'fun1.2',
12       [
13         # Return(Empty())
14         LOADIN BAF PC -1;
15       ],
16     Block
17       Name 'fun2.1',
18       [
19         # // Return(Num('1'))
20         # Exp(Num('1'))
21         SUBI SP 1;
22         LOADI ACC 1;
23         STOREIN SP ACC 1;
24         # Return(Stack(Num('1'))))
25         LOADIN SP ACC 1;
26         ADDI SP 1;
27         LOADIN BAF PC -1;
28       ],
29     Block
30       Name 'main.0',
31       [
32         # Return(Empty())
33         LOADIN BAF PC -1;
34       ]
35 ]

```

**Code 0.8:** RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

Im **RETI Pass** in Code 0.9 werden die Knoten `Exp(GoTo(Name('main.<number>')))` durch den entsprechenden **Sprung** `JUMP <distance_to_main_function>` ersetzt und es werden die **Blöcke entfernt**.

```

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

```

**Code 0.9:** RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.

### 0.0.1.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen

In der Programmiersprache  $L_C$  und somit auch  $L_{PicoC}$  ist es notwendig, dass eine Funktion **deklariert** ist, bevor man einen **Funktionsaufruf** zu dieser Funktion machen kann. Das ist notwendig, damit **Fehlermeldungen** ausgegeben werden können, wenn der **Prototyp** (Definition ??) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen Reihenfolge** übergeben werden.

Die **Deklaration** einer Funktion kann **explizit** erfolgen (z.B. `int fun2(int var);`), wie bei der in Code 0.10 **markierten Zeile 1** oder **indirekt** zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen 3-4**.

In dem Beispiel in Code 0.10 erfolgt in Zeile 7 ein **Funktionsaufruf** der Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdeklaration**, wie in der **markierten Zeile 1** notwendig. Beim **Funktionsaufruf** der Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen 3-4** zu sehen ist.

```

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

**Code 0.10:** PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.

Die **Deklaration** einer **Funktion** erfolgt mithilfe einer **Symboltabelle** (Definition ??), die in Code 0.11 für das Beispiel in Code 0.10 dargestellt ist. Die Symboltabelle ist als **Assoziatives Feld** umgesetzt, indem über einen **Schlüssel** ein **Symboltabelleneintrag** mit verschiedenen **Attributen**, welche bereits in Tabelle ?? erklärt wurden für eine Variable, eine Konstante, eine Funktion, einen Datentyp usw. nachgeschlagen werden kann. Für z.B. die Funktion `int fun2(int var)` werden die **Attribute** des **Symboltabelleneintrags** `Symbol(type_qual, datatype, name, val_addr, pos, size)` wie üblich gesetzt. Dem `datatype`-Attribut werden dabei einfach die Knoten des kompletten **Funktionsprototypes** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(), IntType('int'), Name('var'))])` zugewiesen.

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` in Code 0.11 haben unterschiedliche **Sichtbarkeitsbereiche** (Definition ??), nämlich den der jeweiligen Funktion, also `main` und `fun2`. Die **Sichtbarkeitsbereiche** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` angehängt wird: `var@<fun_name>`. Dieser **Bezeichner** mit **angehängten Suffix** `<var_name>@<scope>` wird als **Schlüssel** zum Nachschlagen von Symboltabelleneinträgen in der Symboltabelle verwendet. Im Fall von Funktionen handelt es sich bei diesem **Suffix** `@<scope>` um den **Bezeichner** der jeweiligen Funktion, der aus dem `name`-Attribut des `FunDef(datatype, name, params, stmts_blocks)`-Knotens entnommen wird.

Dieser **Suffix** wird geändert, sobald beim **Top-Down**<sup>15</sup>-Iterieren über den **Abstrakten Syntaxbaum** ein neuer **FunDef**(datatype, name, allocs, stmts\_blocks)-Knoten betreten wird und über dessen Anweisungsknoten im **stmts\_blocks**-Attribut iteriert wird. Die **Schlüssel** der Symboltabelle sind **identisch** zur Zeichenkette, die im **Name(str)**-Knoten des **name**-Attributes im Symboltabelleneintrag **Symbol**(type\_qual, datatype, name, val\_addr, pos, size) gespeichert ist. Dadurch lässt sich aus dem **Symboltabelleneintrag** einer **Variable** direkt ihr **Sichtbarkeitsbereich**, in dem sie definiert wurde ablesen. Dies ist in Code 0.11 **markiert**.

Die Variable **var@main**, bei der es sich um eine **Lokale Variable** der **main**-Funktion handelt, ist nur innerhalb des **Codeblocks** {} der **main**-Funktion **sichtbar** und die Variable **var@fun2** bei der es sich um einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** {} der Funktion **fun2** **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symboltabelleneintrags** für eine Variable, an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird und nicht nur beim Definieren der Variable.

Ein Grund, warum **Sichtbarkeitsbereiche** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel** der **Symboltabelle** eindeutig sind, denn z.B. bei Funktionen ist es **nicht** möglich zwei Funktionen mit dem **gleichen Bezeichner** zu deklarieren / definieren<sup>16</sup>. Folglich kann eine Variable nur in genau der Funktion **nachgeschlagen**, in der sie **definiert** wurde. Das Symbol '@' wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel ?? erläutert wurde.

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
7         ↪ IntType('int'), Name('var'))])
8       name:                Name('fun2')
9       value or address:    Empty()
10      position:            Pos(Num('1'), Num('4'))
11      size:                Empty()
12    },
13    Symbol
14    {
15      type qualifier:      Empty()
16      datatype:            FunDecl(VoidType('void'), Name('fun1'), [])
17      name:                Name('fun1')
18      value or address:    Empty()
19      position:            Pos(Num('3'), Num('5'))
20      size:                Empty()
21    },
22    Symbol
23    {
24      type qualifier:      Empty()
25      datatype:            FunDecl(VoidType('void'), Name('main'), [])
26      name:                Name('main')
27      value or address:    Empty()
28      position:            Pos(Num('6'), Num('5'))
29      size:                Empty()
30    },
31    Symbol
32    {

```

<sup>15</sup>D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

<sup>16</sup>Sonst gibt es eine **Fehlermeldung**, wie **ReDeclarationOrDefinition**.

```

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

```

**Code 0.11:** *Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

### 0.0.1.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(1+1)`) wird im Folgenden mithilfe des Beispiels in Code 0.12 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und das Beispiel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 0.0.1.3.1 erklärt. Zudem wurde, um die **Adressberechnung anschaulicher** zu machen als **Datentyp** für die beiden **lokalen Variablen** `local_var` in beiden Funktion `main` und `stack_fun` ein **Verbund** gewählt, der **mehrere Speicherzellen** im Hauptspeicher einnimmt.

```

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

```

**Code 0.12:** *PicoC-Code für Funktionsaufruf ohne Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 0.13 wird ein **Funktionsaufruf** `stack_fun(1+1)` durch die **Knoten** `Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))` dargestellt.

Der **Parameter** `param` der Funktion `stack_fun` wird, wie in Code 0.13 markiert ist mithilfe der Knoten `Alloc(Writeable(), IntType('int'), Name('param'))` dargestellt, die normalerweise eine **Allokation** darstellen, wie z.B. bei der Lokalen Variable `local_var`: `Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))), Name('local_var')))`.

Das hat den Grund, dass **Parameter** einer Funktion auf diese Weise als `Exp(Alloc(type_qual, datatype, name))` in der **Symboltabelle** definiert werden können. Hierzu werden die **Parameter**, die als `Alloc(type_qual, datatype, name)`-Knoten dargestellt sind `exp`-Attributen der `Exp(exp)`-Knoten zugewiesen. Die durch die `Exp(Alloc(type_qual, datatype, name))`-Knoten dargestellten Allokationsanweisungen werden im **PicoC-ANF Pass** dann als **erstes** im `stmts_blocks`-Attribut des `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten verarbeitet.

```

1 File
2   Name './example_fun_call_no_return_value.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8       ],
9     FunDecl
10      VoidType 'void',
11      Name 'stack_fun',
12      [
13        Alloc
14          Writeable,
15          IntType 'int',
16          Name 'param'
17      ],
18    FunDef
19      VoidType 'void',
20      Name 'main',
21      [],
22      [
23        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
24          ↪ Name('local_var')))
25        Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))
26        Return(Empty())
27      ],
28    FunDef
29      VoidType 'void',
30      Name 'stack_fun',
31      [
32        Alloc(Writeable(), IntType('int'), Name('param'))
33      ],
34      [
35        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
36          ↪ Name('local_var')))
37      ]
38  ]

```

**Code 0.13:** Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.

Alle Funktionen **außer** der `main`-Funktion besitzen einen **Stackframe** (Definition 0.1). Bei der `main`-Funktion werden **Lokale Variablen** einfach zu den **Globalen Statischen Daten** geschrieben.

In Tabelle 1 ist für das Beispiel in Code 0.12 das **Datensegment** inklusive **Stackframe** der Funktion `stack_fun` mit allen **allokierten Variablen** dargestellt. Mithilfe der Spalte **Relativadresse** in der Tabelle 1 erklären sich auch die **Relativadressen** der Variablen `local_var@main`, `local_var@stack_fun`, `param@stack_fun` in den `value` or `address`-Attributen der markierten **Symboltabelleneinträge** in der **Symboltabelle** in

Code 0.14. Bei **Stackframes** fangen die **Relativadressen** erst 2 Speicherzellen relativ zum **BAF-Register** an, da die **Rücksprungadresse** und die **Startadresse des Vorgängerframes** Platz brauchen.

Dadurch, dass die Allokationsanweisungen `Exp(Alloc(type_qual, datatype, name))` der **Parameter** wie vorhin erwähnt als erstes **vor** allen anderen Anweisungen einer Funktion ausgeführt werden, wird z.B. der **Parameter param** der Funktion `stack_fun` als **erstes** auf dem dem **Stackframe** in Tabelle 1 allokiert und erst danach folgt die **Lokale Variable local\_var**<sup>17</sup>.

Relativ-adresse	Inhalt	Register
0	$\langle local\_var@main \rangle$	CS
1		
2		
3		
...	...	
...	...	SP
4	$\langle local\_var@stack\_fun \rangle$	
3		
2		
1		
0	$\langle param@stack\_fun \rangle$	
...	Rücksprungadresse	
...	Startadresse Vorgängerframe	BAF

**Tabelle 1:** Datensegment mit Stackframe.

#### Definition 0.1: Stackframe



Eine **Datenstruktur**, die dazu dient während der **Laufzeit** eines Programmes den **Zustand** einer Funktion „konservieren“ zu können, um die **Ausführung** dieser Funktion später im **selben Zustand fortsetzen** zu können. **Stackframes** werden dabei in einem Stack **übereinander gestapelt** und in die **entgegengesetzte Richtung** wieder abgebaut, wenn sie nicht mehr benötigt werden. Der **Aufbau** eines **Stackframes** ist in Tabelle 2 dargestellt.<sup>a</sup>

...	← SP
<hr/>	
Temporäre Berechnungen	
Lokale Variablen	
Parameter	
Rücksprungadresse	
Startadresse Vorgängerframe	← BAF

**Tabelle 2:** Aufbau Stackframe

Üblicherweise steht als **erstes**<sup>b</sup> in einem Stackframe die **Startadresse des Vorgängerframes**. Diese ist notwendig, damit beim **Rücksprung** aus einer **aufgerufenen Funktion**, zurück zur **aufgerufenen Funktion** das **BAF-Register** wieder so gesetzt werden kann, dass es auf den **Stackframe der aufrufenden Funktion** zeigt.

Als **zweites** steht in einem Stackframe üblicherweise die **Rücksprungadresse**. Die

<sup>17</sup>Der **Stack** wächst in Tabelle 1 von **unten-nach-oben**, daher wird ein **Stackframe** ebenfalls von **unten-nach-oben** gelesen.

*Rücksprungadresse* ist die Adresse *relativ* zum Anfang des *Codesegments*, an welcher die *Ausführung* der *aufgerufenen Funktion* nach einem Funktionsaufruf *fortgesetzt* wird.

Die *Startadresse* des *Vorgängerframes* und die *Rücksprungadresse* stehen beide im Stackframe der *aufgerufenen Funktion*, weil eine Funktion von allen möglichen Funktionen aufgerufen werden kann und diese beiden Informationen der *aufgerufenen Funktion* für den *Rücksprung* im Stackframe der *aufgerufenen Funktion* benötigt werden, da hier auf die *einfachste Weise* darauf zugegriffen werden kann. Alles weitere in Tabelle 2 ist selbsterklärend.<sup>c</sup>

<sup>a</sup>Wenn von „auf den Stack schreiben“ gesprochen wird, dann wird damit gemeint, dass der Bereich für **Temporäre Berechnungen** (nach Tabelle 2) **vergrößert** wird und ein **Wert** hinein **geschrieben** wird.

<sup>b</sup>Die Tabelle 2 ist von **unten-nach-oben** zu lesen, da im **PicoC-Compiler** Stackframes in einem **Stack** untergebracht werden, der von **unten-nach-oben** wächst. Alles soll **konsistent** dazu gehalten werden, wie es im **PicoC-Compiler** umgesetzt ist.

<sup>c</sup>Scholl, „Betriebssysteme“.

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            ArrayDecl([Num('2')], IntType('int'))
7       name:                Name('attr@st')
8       value or address:    Empty()
9       position:            Pos(Num('1'), Num('15'))
10      size:                 Num('2')
11    },
12    Symbol
13    {
14      type qualifier:      Empty()
15      datatype:            StructDecl(Name('st'), [Alloc(Writeable(),
16      ↪ ArrayDecl([Num('2')], IntType('int')), Name('attr'))])
17      name:                Name('st')
18      value or address:    [Name('attr@st')]
19      position:            Pos(Num('1'), Num('7'))
20      size:                 Num('2')
21    },
22    Symbol
23    {
24      type qualifier:      Empty()
25      datatype:            FunDecl(VoidType('void'), Name('stack_fun'),
26      ↪ [Alloc(Writeable(), IntType('int'), Name('param'))])
27      name:                Name('stack_fun')
28      value or address:    Empty()
29      position:            Pos(Num('3'), Num('5'))
30      size:                 Empty()
31    },
32    Symbol
33    {
34      type qualifier:      Empty()
35      datatype:            FunDecl(VoidType('void'), Name('main'), [])
36      name:                Name('main')
37      value or address:    Empty()
38      position:            Pos(Num('5'), Num('5'))
39      size:                 Empty()
40    },
41  ]

```



```

39 Symbol
40 {
41     type qualifier:      Writeable()
42     datatype:            ArrayDecl([Num('2')], StructSpec(Name('st')))
43     name:                Name('local_var@main')
44     value or address:    Num('0')
45     position:            Pos(Num('6'), Num('12'))
46     size:                Num('4')
47 },
48 Symbol
49 {
50     type qualifier:      Writeable()
51     datatype:            IntType('int')
52     name:                Name('param@stack_fun')
53     value or address:    Num('0')
54     position:            Pos(Num('11'), Num('19'))
55     size:                Num('1')
56 },
57 Symbol
58 {
59     type qualifier:      Writeable()
60     datatype:            ArrayDecl([Num('2')], StructSpec(Name('st')))
61     name:                Name('local_var@stack_fun')
62     value or address:    Num('4')
63     position:            Pos(Num('12'), Num('12'))
64     size:                Num('4')
65 }
66 ]

```

**Code 0.14:** *Symboltabelle für Funktionsaufruf ohne Rückgabewert.*

Im **PicoC-ANF Pass** in Code 0.15 werden die Knoten `Exp(Call(Name('stack_fun'), [Name('local_var')]))` durch die Knoten `StackMalloc(Num('2'))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` usw. ersetzt. Die **Bedeutung** all dieser **Knoten** und **Kompositionen** von Knoten wird in Tabelle 3 erläutert.

Knoten	Beschreibung
StackMalloc(num)	Steht für das <b>Allokieren</b> von num Speicherzellen auf dem <b>Stack</b> .
NewStackframe(fun_name, goto_after_call)	<b>Erstellt</b> einen neuen <b>Stackframe</b> und speichert den Wert des BAF-Registers der <b>aufrufenden Funktion</b> und die <b>Rücksprungadresse</b> nacheinander an den <b>Anfang</b> des neuen <b>Stackframes</b> . Das Attribut <b>fun_name</b> steht dabei für den Bezeichner der Funktion, für die ein neuer <b>Stackframe</b> erstellt werden soll. Das Attribut <b>fun_name</b> dient später dazu den <b>Block</b> dieser Funktion zu finden, weil dieser für den weiteren Kompilervorwärtsschritt wichtige Information in seinen <b>versteckten Attributen</b> gespeichert hat. Des Weiteren enthält das Attribut <b>goto_after_call</b> ein <b>GoTo(Name('addr@next_instr'))</b> , welches später durch die <b>Adresse</b> des Befehls, der direkt auf den <b>Sprungbefehl</b> folgt, ersetzt wird.
Exp(GoTo(name))	Sprung zu einem anderen <b>Block</b> durch Angabe des <b>Bezeichners</b> des Blocks, wobei das Attribut <b>name</b> der <b>Bezeichner</b> des Blocks ist, zu dem gesprungen werden soll.
Instr(Loadi(), [Reg(reg), GoTo(Name('addr@next_instr'))])	Lädt in das <b>reg</b> -Register die <b>Adresse</b> des Befehls, der direkt <b>auf</b> dem nächsten <b>Sprung</b> zum Block einer anderen Funktion folgt. Die Adresse ist dabei <b>relativ</b> zum <b>Anfang</b> des <b>Codesegments</b> .
RemoveStackframe()	Container für das <b>Entfernen</b> des aktuellen <b>Stackframes</b> , durch das <b>Wiederherstellen</b> des im <b>Stackframe</b> der <b>aufgerufenen Funktion</b> gespeicherten Werts des BAF-Registers der <b>aufrufenden Funktion</b> und das Setzen des SP-Registers auf den Wert des BAF-Registers der <b>aufgerufenen Funktion</b> .

Tabelle 3: Knoten für Funktionsaufruf.

Die Knoten `StackMalloc(Num('2'))` sind notwendig, weil auf dem **Stackframe** für den Wert des BAF-Registers der **aufrufenden Funktion** und die **Rücksprungadresse** am **Anfang** des **Stackframes** 2 Speicherzellen Platz gelassen werden müssen. Das wird durch den Knoten `StackMalloc(Num('2'))` umgesetzt, indem das SP-Register einfach um zwei Speicherzellen **dekrementiert** wird und somit Speicher auf dem **Stack** allokiert wird.

1	File
2	Name './example_fun_call_no_return_value.picoc_mon',
3	[
4	Block
5	Name 'main.1',
6	[
7	StackMalloc(Num('2'))
8	Exp(Num('1'))
9	Exp(Num('1'))
10	Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
11	NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))
12	Exp(GoTo(Name('stack_fun.0')))
13	RemoveStackframe()
14	Return(Empty())
15	],
16	Block

```

17     Name 'stack_fun.0',
18     [
19         Return(Empty())
20     ]
21 ]

```

**Code 0.15:** *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.16 werden die **PicoC-Knoten** `StackMalloc(Num('2'))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch **semantisch** entsprechende **Knoten** ersetzt.

Die Knoten `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` in Zeile 29 und 33 in Code 0.16 bestehen noch nicht ausschließlich aus **RETI-Knoten** und werden erst später in dem für sie vorgesehenen **RETI-Pass** in Code 0.17 passend ersetzt. Der **Bezeichner** `stack_fun.0` für `NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))` und `Exp(GoTo(Name('stack_fun.0')))`, wird mithilfe des **Assoziativen Feldes** `fun_name.to.block_name`<sup>18</sup> und dem **Schlüssel** `stack_fun`<sup>19</sup> aus `Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))` nachgeschlagen.<sup>20</sup>

```

1 File
2   Name './example_fun_call_no_return_value.reti_blocks',
3   [
4       Block
5       Name 'main.1',
6       [
7           # StackMalloc(Num('2'))
8           SUBI SP 2;
9           # Exp(Num('1'))
10          SUBI SP 1;
11          LOADI ACC 1;
12          STOREIN SP ACC 1;
13          # Exp(Num('1'))
14          SUBI SP 1;
15          LOADI ACC 1;
16          STOREIN SP ACC 1;
17          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
18          LOADIN SP ACC 2;
19          LOADIN SP IN2 1;
20          ADD ACC IN2;
21          STOREIN SP ACC 2;
22          ADDI SP 1;
23          # NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr'))
24          MOVE BAF ACC;
25          ADDI SP 3;
26          MOVE SP BAF;
27          SUBI SP 7;
28          STOREIN BAF ACC 0;
29          LOADI ACC GoTo(Name('addr@next_instr'));
30          ADD ACC CS;

```

<sup>18</sup>Welches in Unterkapitel 0.0.1.1 eingeführt wurde.

<sup>19</sup>Dem **Bezeichner** der **Funktion** zu der gesprungen werden soll.

<sup>20</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

```

31     STOREIN BAF ACC -1;
32     # Exp(GoTo(Name('stack_fun.0')))
33     Exp(GoTo(Name('stack_fun.0')))
34     # RemoveStackframe()
35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Return(Empty())
39     LOADIN BAF PC -1;
40 ],
41 Block
42   Name 'stack_fun.0',
43   [
44     # Return(Empty())
45     LOADIN BAF PC -1;
46   ]
47 ]

```

**Code 0.16:** *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 0.17 wird nun der finale **RETI-Code** generiert. Die **RETI-Befehle** aus den **Blöcken** sind nun zusammengefügt und es gibt keine **Blöcke** mehr. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in `LOADI ACC GoTo(Name('addr@next_instr'))` durch die **Adresse** des nächsten Befehls direkt nach dem Befehl `JUMP 5`<sup>21 22</sup> ersetzt: `LOADI ACC 21`. Die Knoten, die den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellen, werden durch den Befehl `JUMP 5` ersetzt, der die **Distanz** als die **Anzahl Speicherzellen** 5 bis zum **ersten** Befehl der Funktion `stack_fun` enthält.

Die **Distanz** 5 im **RETI-Knoten** `JUMP 5` wird mithilfe des versteckten `instrs.before`-Attributs des **Zielblocks** `Block(name, stmts_instrs, instrs.before, num_instrs, param_size, local_vars_size)`<sup>23</sup> und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` selbst liegt berechnet.

Die **relative Adresse** 21 des Befehls `LOADI ACC 21` wird ebenfalls mithilfe des versteckten `instrs.before`-Attributs des **aktuellen Blocks** `Block(name, stmts_instrs, instrs.before, num_instrs, param_size, local_vars_size)` berechnet. Es handelt sich bei 21 um eine **relative Adresse**, die **relativ** zum CS-Register<sup>24 25 26</sup> berechnet wird.

#### Anmerkung 🔍

Die Berechnung der **Adresse**  $adr_{danach}$  bzw. `<addr@next_instr>` des Befehls nach dem nächsten **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt mithilfe der folgenden Formel 0.0.1:

$$adr_{danach} = \#Bef_{vor\ akt.\ Bl.} + idx + 4 \quad (0.0.1)$$

wobei:

- es sich bei  $adr_{danach}$  um eine **relative Adresse** handelt, die **relativ** zum **Anfang** des **Codesegments** berechnet wird.

<sup>21</sup>Der für den **Sprung zur gewünschten Funktion** verantwortlich ist.

<sup>22</sup>Also der Befehl, der bisher durch die Knoten `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

<sup>23</sup>Welcher den **ersten Befehl** der gewünschten Funktion enthält.

<sup>24</sup>Also relativ zum **Anfang** des **Codesegments**.

<sup>25</sup>Welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

<sup>26</sup>Wobei man bei den **Adressen** bei 0 anfängt zu zählen.

- $\#Bef_{vor\ akt.\ Bl.} \hat{=}$  **Anzahl** Befehle vor dem aktuellen Block. Es handelt sich hierbei um ein verstecktes Attribut `instrs.before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs.before, num_instrs, param_size, local_vars_size)`, welches im **RETI-Patch-Pass** gesetzt wird. Der Grund dafür, dass das Zuweisen dieses versteckten Attributes `instrs.before` im **RETI-Patch-Pass** erfolgt, ist, weil erst im **RETI-Patch-Pass** die finale **Anzahl** an Befehlen in einem Block feststeht. Das liegt darin begründet, dass im **RETI-Patch-Pass** `GoTo()`'s entfernt werden, deren Sprung nur **eine** Adresse weiterspringen würde. Die finale **Anzahl** an Befehlen kann sich in diesem **Pass** also noch ändern und muss daher im letzten Schritt dieses **Pass** berechnet werden.
- $idx \hat{=}$  **relativer Index** des Befehls `LOADI ACC <addr@next_instr>`, für den die **Adresse**  $adr_{danach}$  bzw. `<addr@next_instr>` gerade berechnet werden soll im **aktuellen Block**.
- $4 \hat{=}$  **Distanz**, die zwischen den in Code 0.17 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` gesprungen werden muss und noch **eins** mehr, weil man ja zum Befehl nach dem Sprung `JUMP <im>` springen will.

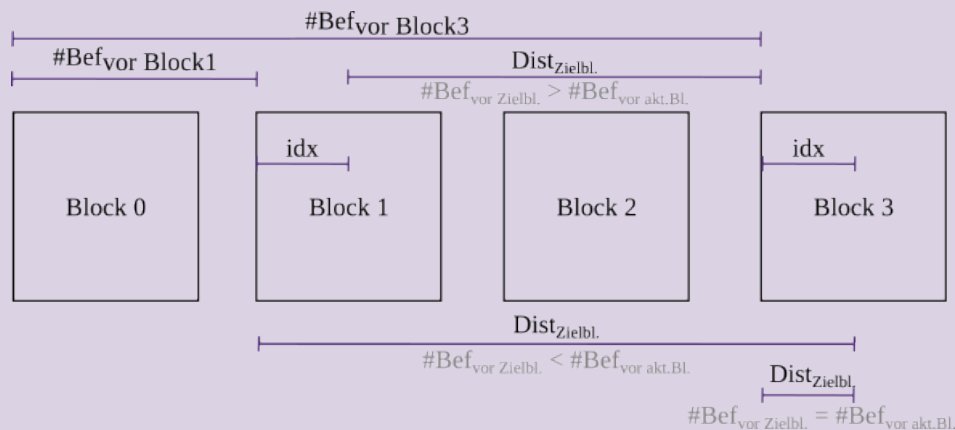
Die Berechnung der **Sprungdistanz**<sup>a</sup>  $Dist_{Zielbl.}$  bzw. `<distance>` zum **ersten** Befehl eines im vorhergehenden **Pass** **existenten Blockes**<sup>b</sup> für den Sprungbefehl `JUMP <distance>` erfolgt nach der folgenden Formel 0.0.2:

$$Dist_{Zielbl.} = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} \neq \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases} \quad (0.0.2)$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$  **Anzahl** Befehle vor dem **Zielblock** zu dem gesprungen werden soll. Es handelt sich hierbei um ein verstecktes Attribut `instrs.before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs.before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$  und  $idx$  haben die **gleiche Bedeutung**, wie in der Formel 0.0.1.
- $idx \hat{=}$  **relativer Index** des Befehls `JUMP <distance>`, für den gerade die **Sprungdistanz**  $Dist_{Zielbl.}$  bzw. `<distance>` gerade berechnet werden soll im **aktuellen Block**.

In Abbildung 1 sind alle 3 möglichen **Konstellationen**  $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$ ,  $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$  und  $\#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.}$  veranschaulicht, welche durch die Formel 0.0.2 abgedeckt sind<sup>c</sup>.



**Abbildung 1:** Veranschaulichung der Distanzberechnung

<sup>a</sup>Diese **Distanz** kann auch **negativ** werden.

<sup>b</sup>Im **RETI-Pass** gibt es **keine** Blöcke mehr.

<sup>c</sup>Die **Konstellationen**  $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt. Bl.}$ ,  $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt. Bl.}$  sind im Fall  $\#Bef_{vor\ Zielbl.} \neq \#Bef_{vor\ akt. Bl.}$  zusammengefasst, da sie auf die **gleiche Weise** berechnet werden.

```

1 # // Exp(GoTo(Name('main.1')))
2 # // not included Exp(GoTo(Name('main.1')))
3 # StackMalloc(Num('2'))
4 SUBI SP 2;
5 # Exp(Num('1'))
6 SUBI SP 1;
7 LOADI ACC 1;
8 STOREIN SP ACC 1;
9 # Exp(Num('1'))
10 SUBI SP 1;
11 LOADI ACC 1;
12 STOREIN SP ACC 1;
13 # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
14 LOADIN SP ACC 2;
15 LOADIN SP IN2 1;
16 ADD ACC IN2;
17 STOREIN SP ACC 2;
18 ADDI SP 1;
19 # NewStackframe(Name('stack_fun.0'), GoTo(Name('addr@next_instr')))
20 MOVE BAF ACC;
21 ADDI SP 3;
22 MOVE SP BAF;
23 SUBI SP 7;
24 STOREIN BAF ACC 0;
25 LOADI ACC 21;
26 ADD ACC CS;
27 STOREIN BAF ACC -1;
28 # Exp(GoTo(Name('stack_fun.0')))
29 JUMP 5;
30 # RemoveStackframe()
31 MOVE BAF IN1;
32 LOADIN IN1 BAF 0;
33 MOVE IN1 SP;
34 # Return(Empty())
35 LOADIN BAF PC -1;
36 # Return(Empty())
37 LOADIN BAF PC -1;

```

**Code 0.17:** *RETI-Pass für Funktionsaufruf ohne Rückgabewert.*

#### 0.0.1.3.1 Rückgabewert

Die Umsetzung eines **Funktionsaufrufs inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 0.18 erklärt.

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

```

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

```

**Code 0.18:** *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 0.19 wird eine **Return-Anweisung mit Rückgabewert** `return 21 * 2` mit den Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, eine **Return-Anweisung ohne Rückgabewert** `return` mit den Knoten `Return(Empty())` und ein **Funktionsaufruf inklusive Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` mit den Knoten `Assign(Alloc(Writable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))`.

```

1 File
2   Name './example_fun_call_with_return_value.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'fun_with_return_value',
7       [],
8       [
9         Return(BinOp(Num('21'), Mul('*'), Num('2')))
10      ],
11     FunDef
12       VoidType 'void',
13       Name 'fun_no_return_value',
14       [],
15       [
16         Return(Empty())
17      ],
18     FunDef
19       VoidType 'void',
20       Name 'main',
21       [],
22       [
23         Assign(Alloc(Writable(), IntType('int'), Name('var')),
24               ↪ Call(Name('fun_with_return_value'), []))
25         Exp(Call(Name('fun_no_return_value'), []))
26      ]
27   ]

```

**Code 0.19:** *Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.*

Im **PicoC-ANF Pass** in Code 0.20 werden bei den Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die Knoten `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierfür erstellten Knoten

`Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis, welches den **Rückgabewert** darstellt, wird dann von den **Knoten** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register ACC** geschrieben. Des Weiteren wird vom `Return(Stack(Num('1')))`-Knoten die **Rücksprungsadresse**, die im **Stackframe** der **aufgerufenen Funktion** gespeichert ist in das PC-Register geladen<sup>27</sup>, um wieder zur **aufzurufenden Funktion** zurückzuspringen.

Ein wichtiges Detail bei der **Funktion** `int fun_with_return_value() { return 21*2; }` ist, dass der **Funktionsaufruf** `Call(Name('fun_with_return_value'), [])` anders übersetzt wird<sup>28</sup>, da diese **Funktion** einen Rückgabewert vom **Datentyp** `IntType()` und nicht `VoidType()` hat. Bei dieser Übersetzung wird durch die Knoten `Exp(ACC)` der **Rückgabewert** der **aufgerufenen Funktion** für die **aufzurufende Funktion**, deren **Stackframe** nun wieder der aktuelle ist vom ACC-Register auf den **Stack** geschrieben. Der **Rückgabewert** wurde zuvor in der **aufgerufenen Funktion** durch die Knoten `Return(BinOp(Num('21'), Mul('*'), Num('2')))` in das ACC-Register geschrieben.

Dieser Trick mit dem Speichern des **Rückgabewerts** im ACC-Register ist notwendig, da der Rückgabewert **nicht** einfach auf den **Stack** gespeichert werden kann. Nach dem **Entfernen** des **Stackframes** der **aufgerufenen Funktion** zeigt das SP-Register nicht mehr an die gleiche Stelle. Daher sind alle **temporären** Werte, die in der **aufgerufenen Funktion** auf den **Stack** geschrieben wurden unzugänglich. Man kann nicht direkt ohne weiteres berechnen, um wieviel die Adresse im SP-Register verglichen zu vorher verschoben ist, weil der Speicherplatz, den Parameter und Lokale Variablen im **Stackframe** einnehmen bei unterschiedlichen **aufgerufenen Funktionen** **unterschiedlich groß** sein kann.

Die **Knoten** `Assign(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))` vereinen **mehrere Aufgaben**. Mittels `Alloc(Writeable(), IntType('int'), Name('var'))` wird die Variable `Name('var')` **allokiert**. Die Knoten `Assign(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))` werden durch die Knoten `Assign(Global(Num('0')), Stack(Num('1')))` ersetzt, welche den **Rückgabewert** der Funktion `'fun_with_return_value'` nun vom **Stack** in die Speicherzelle der Variable `Name('var')` in den **Globalen Statischen Daten** speichern. Hierzu muss die **Adresse** der gerade allokierten Variable `Name('var')` in der **Symboltabelle** nachgeschlagen werden. Der **Rückgabewert** der Funktion `'fun_with_return_value'` wurde zuvor durch die **Knoten** `Exp(ACC)` aus dem ACC-Register auf den **Stack** geschrieben.<sup>29</sup>

Der Umgang mit einer **Funktion ohne Rückgabewert** wurde am Anfang dieses Unterkapitels 0.0.1.3 bereits besprochen. Für ein **return ohne Rückgabewert** bleiben die **Knoten** `Return(Empty())` in diesem Pass unverändert, sie stellen nur das Laden der **Rücksprungsadresse** in das PC-Register dar.

Des Weiteren kann anhand der `main`-Funktion beobachtet werden, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` keine `return`-Anweisung explizit ans Ende geschrieben wird, im **PicoC-ANF Pass** eine in Form der Knoten `Return(Empty())` hinzugefügt wird. Bei **Nicht-Angeben** wird im Falle eines Rückgabedatentyps, der **nicht** `void` ist allerdings eine **MissingReturn-Fehlermeldung** ausgelöst.

```
1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'fun_with_return_value.2',
6       [
```

<sup>27</sup>Die **Rücksprungsadresse** wurde zuvor durch den `NewStackframe()`-Knoten eine Speicherzelle **vor** der Speicherzelle auf die das BAF-Register zeigt im **Stackframe** gespeichert (siehe Unterkapitel 0.0.1.3 für Zusammenhang).

<sup>28</sup>Als in Unterkapitel 0.0.1.3 **bisher** erklärt wurde.

<sup>29</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.



```

7      // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8      Exp(Num('21'))
9      Exp(Num('2'))
10     Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
11     Return(Stack(Num('1')))
12 ],
13 Block
14     Name 'fun_no_return_value.1',
15     [
16         Return(Empty())
17     ],
18 Block
19     Name 'main.0',
20     [
21         // Assign(Name('var'), Call(Name('fun_with_return_value'), []))
22         StackMalloc(Num('2'))
23         NewStackframe(Name('fun_with_return_value.2'), GoTo(Name('addr@next_instr')))
24         Exp(GoTo(Name('fun_with_return_value.2')))
25         RemoveStackframe()
26         Exp(ACC)
27         Assign(Global(Num('0')), Stack(Num('1')))
28         StackMalloc(Num('2'))
29         NewStackframe(Name('fun_no_return_value.1'), GoTo(Name('addr@next_instr')))
30         Exp(GoTo(Name('fun_no_return_value.1')))
31         RemoveStackframe()
32         Return(Empty())
33     ]
34 ]

```

**Code 0.20:** *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.21 werden die **PicoC-Knoten** `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))`, `Return(Stack(Num('1')))`, `Return(Empty())`, `Exp(ACC)` und `Assign(Global(Num('0')), Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2     Name './example_fun_call_with_return_value.reti_blocks',
3     [
4         Block
5             Name 'fun_with_return_value.2',
6             [
7                 # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
8                 # Exp(Num('21'))
9                 SUBI SP 1;
10                LOADI ACC 21;
11                STOREIN SP ACC 1;
12                # Exp(Num('2'))
13                SUBI SP 1;
14                LOADI ACC 2;
15                STOREIN SP ACC 1;
16                # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
17                LOADIN SP ACC 2;
18                LOADIN SP IN2 1;

```

```

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

```

```

76      LOADIN IN1 BAF 0;
77      MOVE IN1 SP;
78      # Return(Empty())
79      LOADIN BAF PC -1;
80  ]
81  ]

```

**Code 0.21:** RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.

#### 0.0.1.3.2 Umsetzung der Übergabe eines Feldes

Die Eigenheit, dass bei der **Übergabe** eines **Feldes** an eine andere Funktion, dieses als **Zeiger** übergeben wird, wurde bereits im Unterkapitel ?? erläutert. Die Umsetzung der **Übergabe** eines **Feldes** an eine andere Funktion wird im Folgenden mithilfe des Beispiels in Code 0.22 erklärt.

```

1 void fun_array_from_stackframe(int (*param)[3]) {
2 }
3
4 void fun_array_from_global_data(int param[2][3]) {
5     int local_var[2][3];
6     fun_array_from_stackframe(local_var);
7 }
8
9 void main() {
10     int local_var[2][3];
11     fun_array_from_global_data(local_var);
12 }

```

**Code 0.22:** PicoC-Code für die Übergabe eines Feldes.

In der **Symboltabelle** in Code 0.23 kann die **Umwandlung** eines **Feldes** zu einem **Zeiger** beobachtet werden. Die **lokalen Variablen** `local.var@main` und `local.var@fun_array_from_global_data` sind beide vom Datentyp `ArrayDecl([Num('2'), Num('3')], IntType('int'))` und bei der Übergabe werden sie an **Parameter** `'param@fun_array_from_global_data'` und `'param@fun_array_from_stackframe'` mit dem Datentyp `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))` gebunden. Die **Größe** dieser **Parameter** beträgt dabei, wie in der Symboltabelle in Code 0.23 gesehen werden kann `Num('1')`, da ein **Zeiger** nur **eine** Speicherzelle einnimmt.

Wie in Unterkapitel 0.0.1.3 erklärt wurde, werden **Parameter** zu Allokationsanweisungen `Exp(Alloc(type_qual, datatype, name))` umgewandelt und als **erstes** im `stmts.blocks`-Attribut des `FunDef(datatype, name, allocs, stmts.blocks)`-Knoten verarbeitet. Bei der gerade erwähnte **Änderung** des **Datentyps** von einem **Feld-Datentyp** zu einem **Zeiger-Datentyp**, soll nur bei Allokationsanweisungen von **Parametern** der Feld-Datentyp zu einem Zeiger-Datentyp umgewandelt werden. Bei Allokationsanweisungen von **Lokalen Variablen** soll **keine Umwandlung** des Datentyps stattfinden. Um **unterscheiden** zu können, ob es sich bei der Allokationsanweisung um die Allokation eines **Parameters** oder eine **Lokalen Variablen** handelt, wird das versteckte Attribut `local_var_or_param`-Attribut des `Alloc(type_qual, datatype, name, local_var_or_param)`-Knoten zu Beginn des **PicoC-ANF Pass** mit z.B. einem entsprechenden `Name('param')`-Knoten belegt, sodass in diesem Fall klar ist, dass es sich um einen **Parameter** handelt.

Um die Änderung des Datentyps von einem **Feld-Datentyp** zu einem **Zeiger-Datentyp** umzusetzen, wird bei der **Erstellung** eines **Symboltableneintrags** für einen **Feld-Datentyp**, vor dem Zuweisen an das

`datatype`-Attribut eines Symboltabelleneintrags, der **oberste Knoten** des Teilbaums, der den **Feld-Datentyp** `ArrayDecl(nums, datatype)` darstellt zu einem **Zeiger-Datentyp** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut des `ArrayDecl(nums, datatype)`-Knoten hängt, wird an das `datatype`-Attribut des **Zeigers-Datentyps** `PntrDecl(num, datatype)` gehängt. Bei einem **Mehrdimensionalen Feld**, fällt eine **Dimension** an den Zeiger **weg** und der Rest des **Feld-Datentyps** `ArrayDecl(nums, datatype)` wird an das `datatype`-Attribut des **Zeiger-Datentyps** `PntrDecl(num, datatype)` gehängt.

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
7         ↪ [Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param'))])
9       name:                Name('fun_array_from_stackframe')
10      value or address:     Empty()
11      position:            Pos(Num('1'), Num('5'))
12      size:                Empty()
13    },
14    Symbol
15    {
16      type qualifier:      Writable()
17      datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
18      name:                Name('param@fun_array_from_stackframe')
19      value or address:     Num('0')
20      position:            Pos(Num('1'), Num('37'))
21      size:                Num('1')
22    },
23    Symbol
24    {
25      type qualifier:      Empty()
26      datatype:            FunDecl(VoidType('void'), Name('fun_array_from_global_data'),
27        ↪ [Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')), Name('param'))])
28      name:                Name('fun_array_from_global_data')
29      value or address:     Empty()
30      position:            Pos(Num('4'), Num('5'))
31      size:                Empty()
32    },
33    Symbol
34    {
35      type qualifier:      Writable()
36      datatype:            PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
37      name:                Name('param@fun_array_from_global_data')
38      value or address:     Num('0')
39      position:            Pos(Num('4'), Num('36'))
40      size:                Num('1')
41    },
42    Symbol
43    {
44      type qualifier:      Writable()
45      datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
46      name:                Name('local_var@fun_array_from_global_data')
47      value or address:     Num('6')
48      position:            Pos(Num('5'), Num('6'))
49      size:                Num('6')
50    }
51  ]

```

```

47     },
48     Symbol
49     {
50         type qualifier:      Empty()
51         datatype:            FunDecl(VoidType('void'), Name('main'), [])
52         name:                 Name('main')
53         value or address:     Empty()
54         position:             Pos(Num('9'), Num('5'))
55         size:                 Empty()
56     },
57     Symbol
58     {
59         type qualifier:      Writeable()
60         datatype:            ArrayDecl([Num('2'), Num('3')], IntType('int'))
61         name:                 Name('local_var@main')
62         value or address:     Num('0')
63         position:             Pos(Num('10'), Num('6'))
64         size:                 Num('6')
65     }
66 ]

```

**Code 0.23:** *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 0.24 ist zu sehen, dass zur Übergabe der beiden Felder `local_var@main` und `local_var@fun_array_from_global_data` die **Adressen** der Felder mithilfe der Knoten `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` auf den **Stack** geschrieben werden. Die Knoten `Ref(Global(Num('0')))` sind für die **Variable** `local_var` aus der `main`-Funktion, da diese in den **Globalen Statischen Daten** liegt und die Knoten `Ref(Stackframe(Num('6')))` sind für die Variable `local_var` aus der Funktion `fun_array_from_global_data`, da diese auf dem **Stackframe** dieser Funktion liegt.

Die Knoten `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` werden später im **RETI-Blocks Pass** in Code 0.25 durch unterschiedliche **RETI-Befehle** ersetzt. Hierbei stellen die Zahlen '0' bzw. '6' in den Knoten `Global(num)` bzw. `Stackframe(num)`, die aus der **Symboltabelle** in Code 0.23 entnommen sind **relative Adressen** relativ zum DS-Register bzw. SP-Register dar. Die Zahl '6' ergibt sich dadurch, dass das Feld `local_var` der Funktion `fun_array_from_global_data` die **Dimensionen**  $2 \times 3$  hat und ein Feld von **Integern** ist, also  $size(type(local\_var)) = \left(\prod_{j=1}^n \dim_j\right) \cdot size(int) = 2 \cdot 3 \cdot 1 = 6$  Speicherzellen.

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'fun_array_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_array_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Ref(Stackframe(Num('6')))
14        NewStackframe(Name('fun_array_from_stackframe.2'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('fun_array_from_stackframe.2')))

```

```

16     RemoveStackframe()
17     Return(Empty())
18 ],
19 Block
20     Name 'main.0',
21     [
22         StackMalloc(Num('2'))
23         Ref(Global(Num('0'))))
24         NewStackframe(Name('fun_array_from_global_data.1'), GoTo(Name('addr@next_instr'))
25         Exp(GoTo(Name('fun_array_from_global_data.1'))))
26         RemoveStackframe()
27         Return(Empty())
28     ]
29 ]

```

**Code 0.24:** *PicoC-ANF Pass für die Übergabe eines Feldes.*

Im **RETI-Blocks Pass** in Code 0.25 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2     Name './example_fun_call_by_sharing_array.reti_blocks',
3     [
4         Block
5             Name 'fun_array_from_stackframe.2',
6             [
7                 # Return(Empty())
8                 LOADIN BAF PC -1;
9             ],
10        Block
11            Name 'fun_array_from_global_data.1',
12            [
13                # StackMalloc(Num('2'))
14                SUBI SP 2;
15                # Ref(Stackframe(Num('6'))))
16                SUBI SP 1;
17                MOVE BAF IN1;
18                SUBI IN1 8;
19                STOREIN SP IN1 1;
20                # NewStackframe(Name('fun_array_from_stackframe.2'), GoTo(Name('addr@next_instr'))
21                MOVE BAF ACC;
22                ADDI SP 3;
23                MOVE SP BAF;
24                SUBI SP 3;
25                STOREIN BAF ACC 0;
26                LOADI ACC GoTo(Name('addr@next_instr'));
27                ADD ACC CS;
28                STOREIN BAF ACC -1;
29                # Exp(GoTo(Name('fun_array_from_stackframe.2'))))
30                Exp(GoTo(Name('fun_array_from_stackframe.2'))))
31                # RemoveStackframe()
32                MOVE BAF IN1;
33                LOADIN IN1 BAF 0;
34                MOVE IN1 SP;

```

```

35     # Return(Empty())
36     LOADIN BAF PC -1;
37 ],
38 Block
39     Name 'main.O',
40     [
41         # StackMalloc(Num('2'))
42         SUBI SP 2;
43         # Ref(Global(Num('0')))
44         SUBI SP 1;
45         LOADI IN1 0;
46         ADD IN1 DS;
47         STOREIN SP IN1 1;
48         # NewStackframe(Name('fun_array_from_global_data.1'), GoTo(Name('addr@next_instr'))
49         MOVE BAF ACC;
50         ADDI SP 3;
51         MOVE SP BAF;
52         SUBI SP 9;
53         STOREIN BAF ACC 0;
54         LOADI ACC GoTo(Name('addr@next_instr'));
55         ADD ACC CS;
56         STOREIN BAF ACC -1;
57         # Exp(GoTo(Name('fun_array_from_global_data.1')))
58         Exp(GoTo(Name('fun_array_from_global_data.1')))
59         # RemoveStackframe()
60         MOVE BAF IN1;
61         LOADIN IN1 BAF 0;
62         MOVE IN1 SP;
63         # Return(Empty())
64         LOADIN BAF PC -1;
65     ]
66 ]

```

Code 0.25: RETI-Block Pass für die Übergabe eines Feldes.

### 0.0.1.3.3 Umsetzung der Übergabe eines Verbundes

Die Eigenheit, dass ein **Verbund** als Argument beim **Funktionsaufruf** einer anderen Funktion in den **Stackframe** der **aufgerufenen** Funktion **kopiert** wird, wurde bereits im Unterkapitel ?? erläutert. Die Umsetzung der **Übergabe** eines **Verbundes** wird im Folgenden mithilfe des Beispiels in Code 0.26 erklärt.

```

1 struct st {int attr1; int attr2[2];};
2
3
4 void fun_struct_from_stackframe(struct st param) {
5 }
6
7 void fun_struct_from_global_data(struct st param) {
8     fun_struct_from_stackframe(param);
9 }
10
11
12 void main() {

```

```

13 struct st local_var;
14 fun_struct_from_global_data(local_var);
15 }

```

**Code 0.26:** *PicoC-Code für die Übergabe eines Verbundes.*

Im **PicoC-ANF Pass** in Code 0.27 werden zur **Übergabe der beiden Verbunde** `local_var@main` und `param@fun_array_from_global_data`, diese mittels der Knoten `Assign(Stack(Num('3')), Global(Num('0')))` bzw. `Assign(Stack(Num('3')), Stackframe(Num('2')))` jeweils auf den **Stack** kopiert.

Bei der **Übergabe** an eine **Funktion** wird der Zugriff auf einen gesamten **Verbund** anders gehandhabt als bei einem **Feld**<sup>30</sup>. Beim einem **Feld** wurde bei der **Übergabe an eine Funktion** die **Adresse des ersten Feldelements** auf den **Stack** geschrieben. Bei einem **Verbund** wird bei der **Übergabe an eine Funktion** dagegen der gesamte **Verbund** auf den **Stack** kopiert.

Des Weiteren muss unterschieden werden, ob ein **Verbund** an eine **Funktion übergeben** wird oder einfach nur **normal** auf einen Verbund **zugegriffen** wird. Wenn **normal** auf einen Verbund **zugegriffen** wird, dann wird das **erste Attribut** auf den Stack geschrieben. Wenn ein Verbund dagegen an eine **Funktion übergeben** wird, wird dieser **komplett** auf den **Stack** kopiert, um später Teil des **Stackframes** der **aufgerufenen Funktion** zu werden.

Das wird durch eine globale Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange ein **Funktionsaufruf** im **Picoc-ANF Pass** übersetzt wird und wieder auf `false` gesetzt, wenn die Übersetzung des **Funktionsaufrufs** abgeschlossen ist. Solange die Variable `argmode_on` auf `true` gesetzt ist, werden die **Knoten** `Assign(Stack(Num('3')), Global(Num('0')))` bzw. `Assign(Stack(Num('3')), Stackframe(Num('2')))` für die Ersetzung verwendet. Ist die Variable `argmode_on` auf `false`, werden die Knoten `Exp(Global(num))` bzw. `Exp(Stackframe(num))` für die Ersetzung verwendet.<sup>31</sup>

Die Knoten `Assign(Stack(Num('3')), Global(Num('0')))` werden verwendet, da die Verbundsvariable `local_var` der `main`-Funktion in den **Globalen Statischen Daten** liegt und die Knoten `Assign(Stack(Num('3')), Stackframe(Num('2')))` werden verwendet, da die Verbundsvariable `local_var` der Funktion `fun_struct_from_global_data` im **Stackframe** der Funktion `fun_struct_from_global_data` liegt.

```

1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'fun_struct_from_stackframe.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun_struct_from_global_data.1',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Stackframe(Num('2'))))
14        NewStackframe(Name('fun_struct_from_stackframe.2'), GoTo(Name('addr@next_instr'))))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2'))))

```

<sup>30</sup>Wie es in Unterkapitel 0.0.1.3.2 erklärt wurde

<sup>31</sup>Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.



```

16     RemoveStackframe()
17     Return(Empty())
18 ],
19 Block
20     Name 'main.0',
21     [
22         StackMalloc(Num('2'))
23         Assign(Stack(Num('3')), Global(Num('0'))))
24         NewStackframe(Name('fun_struct_from_global_data.1'), GoTo(Name('addr@next_instr'))
25         Exp(GoTo(Name('fun_struct_from_global_data.1'))))
26         RemoveStackframe()
27         Return(Empty())
28     ]
29 ]

```

**Code 0.27:** *PicoC-ANF Pass für die Übergabe eines Verbundes.*

Im **RETI-Blocks Pass** in Code 0.28 werden die **PicoC-Knoten** `Assign(Stack(Num('3')),Stackframe(Num('2')))` und `Assign(Stack(Num('3')),Global(Num('0')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2     Name './example_fun_call_by_value_struct.reti_blocks',
3     [
4         Block
5             Name 'fun_struct_from_stackframe.2',
6             [
7                 # Return(Empty())
8                 LOADIN BAF PC -1;
9             ],
10        Block
11            Name 'fun_struct_from_global_data.1',
12            [
13                # StackMalloc(Num('2'))
14                SUBI SP 2;
15                # Assign(Stack(Num('3')), Stackframe(Num('2'))))
16                SUBI SP 3;
17                LOADIN BAF ACC -4;
18                STOREIN SP ACC 1;
19                LOADIN BAF ACC -3;
20                STOREIN SP ACC 2;
21                LOADIN BAF ACC -2;
22                STOREIN SP ACC 3;
23                # NewStackframe(Name('fun_struct_from_stackframe.2'), GoTo(Name('addr@next_instr'))
24                MOVE BAF ACC;
25                ADDI SP 5;
26                MOVE SP BAF;
27                SUBI SP 5;
28                STOREIN BAF ACC 0;
29                LOADI ACC GoTo(Name('addr@next_instr'));
30                ADD ACC CS;
31                STOREIN BAF ACC -1;
32                # Exp(GoTo(Name('fun_struct_from_stackframe.2'))))
33                Exp(GoTo(Name('fun_struct_from_stackframe.2'))))

```

```

34     # RemoveStackframe()
35     MOVE BAF IN1;
36     LOADIN IN1 BAF 0;
37     MOVE IN1 SP;
38     # Return(Empty())
39     LOADIN BAF PC -1;
40 ],
41 Block
42     Name 'main.0',
43     [
44         # StackMalloc(Num('2'))
45         SUBI SP 2;
46         # Assign(Stack(Num('3')), Global(Num('0')))
47         SUBI SP 3;
48         LOADIN DS ACC 0;
49         STOREIN SP ACC 1;
50         LOADIN DS ACC 1;
51         STOREIN SP ACC 2;
52         LOADIN DS ACC 2;
53         STOREIN SP ACC 3;
54         # NewStackframe(Name('fun_struct_from_global_data.1'), GoTo(Name('addr@next_instr')))
55         MOVE BAF ACC;
56         ADDI SP 5;
57         MOVE SP BAF;
58         SUBI SP 5;
59         STOREIN BAF ACC 0;
60         LOADI ACC GoTo(Name('addr@next_instr'));
61         ADD ACC CS;
62         STOREIN BAF ACC -1;
63         # Exp(GoTo(Name('fun_struct_from_global_data.1')))
64         Exp(GoTo(Name('fun_struct_from_global_data.1')))
65         # RemoveStackframe()
66         MOVE BAF IN1;
67         LOADIN IN1 BAF 0;
68         MOVE IN1 SP;
69         # Return(Empty())
70         LOADIN BAF PC -1;
71     ]
72 ]

```

**Code 0.28:** *RETI-Block Pass für die Übergabe eines Verbundes.*

# Literatur

## Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [https://abs.informatik.uni-freiburg.de/src/teach\\_main.php?id=157](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157) (besucht am 09.07.2022).