

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 13. September 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

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.

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, bin ich keine Person, die irgendwelche Dinge gerne so macht wie es üblich ist, ich schreibe meine Danksagung nicht auf eine bestimmte Weise, nur weil sich irgendwann mal etabliert hat wie eine Danksagung üblicherweise aussieht. Ich halte nicht viel von 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.

Ich halte es eher so, dass wenn mir wirklich etwas am Dank gegenüber Personen liegt, ich mir wirklich den Aufwand mache einen Text zu schreiben in dem ich diesen zum Ausdruck bringe, im anderen Fall kann man sich bei mir auf die typischen Standardfloskeln einstellen. Bei dieser Bachelorarbeit kann ich nur auf ersteres Zurückgreifen. 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 sehr respektvoll, 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 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 habe¹, er hat sich bei der Korrektur dieser Schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere Textkommentare zu verfassen und das trotz dessen, dass meine Bachelorarbeit recht Umfangreich zu lesen ist² und war sich nicht zu Schade die Rolle des Nachrichtenübersetzers 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. Ich hab während meines Bachelorprojekts und meiner Bachelorarbeit wahrscheinlich einen ziemlich eigensinnigen Eindruck gemacht, bei der Weise, wie ich bestimmte Dinge umsetzen wollte. 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.

¹Wofür ich mich auch nochmal Entschuldigen will.

²Wobei er sich kein einziges Mal in geringster Weise entnervt darüber gezeigt hat.

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 Michel Giehl sich netterweise zur Verfügung gestellt hat. 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 Mist baue. Der RETI-Emulator von Michel Giehl ist unter [Link](#)³ zu finden.

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, was man auch selten im Studium erlebt, dass dem 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 ziemlich viel unerwartete Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl nichts geworden. Man hat daran gemerkt, dass Prof. Dr. Scholl das Wohlergehen der Studenten wichtig ist.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein persönliches Interessengebiet fällt. Das Aufschreiben dieser Schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁴. 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, die sonst ziemlich willkürlich erscheinen würde, 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 großartigen Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

³<https://github.com/michel-giehl/Reti-Emulator>.

⁴Da es recht stressig ist im Kopf zu behalten, was man schon erklärt hat und wo noch eine Verständnislücke vorliegen könnte.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
0.0.1 Umsetzung von Funktionen	1
0.0.1.1 Mehrere Funktionen	1
0.0.1.1.1 Sprung zur Main Funktion	4
0.0.1.2 Funktionsdeklaration und -definition und Umsetzung von Sichtbarkeitsbereichen	7
0.0.1.3 Funktionsaufruf	9
0.0.1.3.1 Rückgabewert	14
0.0.1.3.2 Umsetzung der Übergabe eines Feldes	18
0.0.1.3.3 Umsetzung einer Übergabe eines Verbundes	22
Literatur	A

Abbildungsverzeichnis

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.	3
0.5	RETI-Blocks Pass für 3 Funktionen.	4
0.6	PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	4
0.7	RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	5
0.8	RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	6
0.9	RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.	7
0.10	PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.	7
0.11	Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.	9
0.12	PicoC-Code für Funktionsaufruf ohne Rückgabewert.	9
0.13	Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.	10
0.14	PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.	11
0.15	RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.	12
0.16	RETI-Pass für Funktionsaufruf ohne Rückgabewert.	14
0.17	PicoC-Code für Funktionsaufruf mit Rückgabewert.	14
0.18	Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.	15
0.19	PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.	16
0.20	RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.	18
0.21	PicoC-Code für die Übergabe eines Feldes.	18
0.22	Symboltabelle für die Übergabe eines Feldes.	20
0.23	PicoC-ANF Pass	20
0.24	RETI-Block Pass für die Übergabe eines Feldes.	22
0.25	PicoC-Code für Call by Value für Structs.	22
0.26	PicoC-ANF Pass für Call by Value für Structs.	23
0.27	RETI-Block Pass für Call by Value für Structs.	25

Tabellenverzeichnis

Definitionsverzeichnis

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 **dekliert** und **definiert** werden können und hierbei **Sichtbarkeitsbereiche** 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** muss im speziellen nochmal darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 0.0.1.3.1) umgesetzt sind und die **Übergabe** von **Abgeleiteten 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** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

```

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

```

Code 0.1: PicoC-Code für 3 Funktionen.

Im **Abstrakten Syntaxbaum** in Code 0.2 wird eine **Funktion**, wie z.B. `voidfun(intparam;){ returnparam; }` mit der Komposition `FunDef(IntType(), Name('fun'), [Alloc(Writeable(), IntType(), Name('fun'))], [Return(Exp(Name('param')))])` dargestellt. Die einzelnen **Attribute** dieses Knoten sind in Tabelle ?? erklärt.

```

1 File
2   Name './verbose_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return
10        Empty
11      ],
12     FunDef
13       VoidType 'void',
14       Name 'fun1',
15       [],
16       []

```

```

17     FunDef
18         IntType 'int',
19         Name 'fun2',
20         [],
21         [
22             Return
23             Num '1'
24         ]
25 ]

```

Code 0.2: Abstrakter Syntaxbaum für 3 Funktionen.

Im **PicoC-Blocks Pass** in Code 0.3 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`⁵ beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`⁶.

Es werden parallel dazu neue Zuordnungen im **Assoziativen Feld** `fun_name_to_block_name` hinzugefügt. Das **Dicionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu⁷. Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```

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

```

⁵Eine Erklärung dazu ist in Unterkapitel ?? zu finden.

⁶Der **Grund** dafür kann im Unterkapitel ?? nachgelesen werden.

⁷Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

```

26     Name 'fun2',
27     [],
28     [
29         Block
30         Name 'fun2.0',
31         [
32             Return(Num('1'))
33         ]
34     ]
35 ]

```

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

```

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

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

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.2'))))
3 # // not included Exp(GoTo(Name('main.2'))))
4 # // Block(Name('main.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.1'), [])
8 # Return(Empty())
9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1'))))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 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** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 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** nur noch 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 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 Index 0 geschoben werden.

Eine andere Möglichkeit und die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die **main-Funktion** nicht die erste auftauchende Funktion ist, einen `start.<nummer>`-Block als ersten Block einzufügen, der einen `GoTo(Name('main.<nummer>'))`-Knoten enthält, der im **RETI Pass 0.9** in einen Sprung zur **main-Funktion** übersetzt wird.

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

Das **Einsetzen** des `start.<nummer>`-Blockes erfolgt im **RETI-Patch Pass** in Code 0.8, da der **RETI-Patch-Pass** der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die **main-Funktion** nicht die erste Funktion ist.

```

1 File
2   Name './verbose_3_funs_main.reti_patch',
3   [
4     Block
5       Name 'start.3',
6       [

```

```

7      # // Exp(GoTo(Name('main.0')))
8      Exp(GoTo(Name('main.0')))
9      ],
10     Block
11       Name 'fun1.2',
12       [
13         # Return(Empty())
14         LOADIN BAF PC -1;
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 wird das `GoTo(Name('main.<number>'))` durch den entsprechenden Sprung `JUMP <distanz_zur_main_funktion>` ersetzt und 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 in der im Beispiel in Code 0.10 **markierten Zeile 1** oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen 3-4**.

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

```

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 der **Symboltabelle**, die in Code 0.11 für das Beispiel in Code 0.10 dargestellt ist. Die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` werden wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette Komposition der **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writable(), IntType('int'), Name('var'))])` zugewiesen.

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Sichtbarkeitsbereiche** (Definition ??). Die **Sichtbarkeitsbereiche** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**⁸ Durchiterieren über den **Abstrakten Syntaxbaum** des aktuellen **Passes** ein **Funktionswechsel** eintritt und über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

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

Ein Grund, warum **Sichtbarkeitsbereiche** über das Anhängen eines **Suffix** an den **Bezeichner** gelöst sind, ist, dass auf diese Weise die **Schlüssel**, die aus dem **Bezeichner** einer Variable und einem angehängten **Suffix** bestehen, in der als **Assoziatives Feld** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt ihren **Sichtbarkeitsbereich** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im **Name(str)**-Knoten des **name**-Attributbes eines **Symbols** der Symboltabelle angehängt. Zur beseren Vorstellung ist dies 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 im einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** {} der Funktion **fun2** **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symbols** in der **Symboltabelle** an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird. Und da die Zuordnungen im **Assoziativen Feld eindeutig** sind, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

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

```

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    {
33      type qualifier:      Writeable()
34      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(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 0.12 erklärt. Das Beispiel ist so gewählt, dass alleinig der **Funktionsaufruf** im **Vordergrund** steht und dieses Kapitel nicht auch noch mit z.B. Aspekten wie der Umsetzung eines **Rückgabewertes** überladen ist. Der Aspekt der Umsetzung eines **Rückgabewertes** wird erst im nächsten Unterkapitel 0.0.1.3.1 erklärt.

```

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

```

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

Im **Abstrakten Syntaxbaum** in Code 0.13 wird ein **Funktionsaufruf** `stack_fun(local_var)` durch die **Komposition** `Exp(Call(Name('stack_fun'), [Name('local_var')]))` dargestellt.

```

1 File
2   Name './example_fun_call_no_return_value.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), IntType('int'), Name('attr1'))

```

```

8      Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr2'))
9  ],
10  FunDecl
11      VoidType 'void',
12      Name 'stack_fun',
13      [
14          Alloc
15              Writeable,
16              ArrayDecl
17                  [
18                      Num '2',
19                      Num '3'
20                  ],
21              StructSpec
22                  Name 'st',
23                  Name 'param'
24      ],
25  FunDef
26      VoidType 'void',
27      Name 'main',
28      [],
29      [
30          Exp(Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
31              ↪ Name('local_var')))
32          Exp(Call(Name('stack_fun'), [Name('local_var')]))
33          Return(Empty())
34      ],
35  FunDef
36      VoidType 'void',
37      Name 'stack_fun',
38      [
39          Alloc(Writeable(), ArrayDecl([Num('2'), Num('3')], StructSpec(Name('st'))),
40              ↪ Name('param'))
41      ],
42      [
43          Exp(Alloc(Writeable(), IntType('int'), Name('local_var')))
44      ]
45  ]

```

Code 0.13: Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.

Im **PicoC-ANF Pass** in Code 0.14 wird die Komposition `Exp(Call(Name('stack_fun'), [Name('local_var')]))` durch die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'))`, `GoTo(Name('addr@next_instr'))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` ersetzt, welche in den Tabellen ?? und ?? genauer erklärt sind.

Der Knoten `StackMalloc(Num('2'))` ist notwendig, weil auf dem **Stackframe** für den Wert des BAF-Registers der **aufzufendenden Funktion** und die **Rücksprungadresse** 2 Speicherzellen Platz am **Anfang** des **Stackframes** gelassen werden muss. 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** belegt wird⁹.

⁹Wobei hier "reserviert" besser passen würde.

```

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         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
10        Exp(GoTo(Name('stack_fun.0')))
11        RemoveStackframe()
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]

```

Code 0.14: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.15 werden die Kompositionen `StackMalloc(Num('2'))`, `Ref(Global(Num('0')))`, `NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))`, `Exp(GoTo(Name('stack_fun.0')))` und `RemoveStackframe()` durch ihre entsprechenden **RETI-Knoten** ersetzt.

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

Für den **Bezeichner des Blocks** `stack_fun.0` in der Komposition `Exp(GoTo(Name('stack_fun.0')))` wird im **Assoziativen Feld** `fun_name.to.block_name`¹⁰ mit dem Schlüssel `stack_fun`, dem **Bezeichner der Funktion**, der im Knoten `NewStackframe(Name('stack_fun'))` gespeichert ist nachgeschlagen.

```

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         # Ref(Global(Num('0')))
10        SUBI SP 1;
11        LOADI IN1 0;
12        ADD IN1 DS;
13        STOREIN SP IN1 1;
14        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        MOVE BAF ACC;
16        ADDI SP 3;
17        MOVE SP BAF;
18        SUBI SP 4;

```

¹⁰Dieses Assoziative Feld wurde in Unterkapitel 0.0.1.1 eingeführt.

```

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

```

Code 0.15: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 0.15 wird nun der finale **RETI-Code** erstellt. Eine Änderung, die direkt auffällt, ist, dass die **RETI-Befehle** aus den **Blöcken** nun zusammengefügt sind und es keine **Blöcke** mehr gibt. Des Weiteren wird das `GoTo(Name('addr@next_instr'))` in der Komposition `LOADI ACC GoTo(Name('addr@next_instr'))` nun durch die **Adresse** des nächsten Befehls direkt nach dem dem Befehl `JUMP 5`, der für den **Sprung zur gewünschten Funktion** verantwortlich ist¹¹ ersetzt: `LOADI ACC 14`. Und auch der Knoten, der den Sprung `Exp(GoTo(Name('stack_fun.0')))` darstellt wird durch den Knoten `JUMP 5` ersetzt.

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

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

Anmerkung

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

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

wobei:

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

¹¹Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.

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

- $idx \hat{=}$ relativer Index des Befehls `LOADI ACC <addr@next_instr>` selbst im Block.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 0.16 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

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

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

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock**, der den **ersten** Befehl einer Funktion enthält und 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.

```

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

```

```
28 LOADIN BAF PC -1;
```

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

0.0.1.3.1 Rückgabewert

Ein **Funktionsaufruf inklusive Zuweisung eines Rückgabewertes** (z.B. `int var = fun_with_return_value()`) wird im Folgenden mithilfe des Beispiels in Code 0.17 erklärt.

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

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

Code 0.17: *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 0.18 wird ein **Return-Statement mit Rückgabewert** `return 21 * 2` mit der Komposition `Return(BinOp(Num('21'), Mul('*'), Num('2')))` dargestellt, ein **Return-Statement ohne Rückgabewert** `return` mit der Komposition `Return(Empty())` und ein **Funktionsaufruf inklusive Zuweisung des Rückgabewertes** `int var = fun_with_return_value()` durch die Komposition `Assign(Alloc(Writeable(), IntType('int'), Name('var')), Call(Name('fun_with_return_value'), []))`.

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



```

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

```

Code 0.18: Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.

Im **PicoC-ANF Pass** in Code 0.19 wird bei der **Komposition** `Return(BinOp(Num('21'), Mul('*'), Num('2')))` erst die **Expression** `BinOp(Num('21'), Mul('*'), Num('2'))` ausgewertet. Die hierfür erstellten Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))` berechnen das Ergebnis des Ausdrucks `21*2` auf dem **Stack**. Dieses Ergebnis wird dann von der **Komposition** `Return(Stack(Num('1')))` vom **Stack** gelesen und in das **Register ACC** geschrieben und als letztes wird die **Rücksprungsadresse** in das PC-Register geladen, die durch den `NewStackframe()`-Token-Knoten eine Speicherzelle nach dem Wert des BAF-Registers der aufrufenden Funktion im **Stackframe** gespeichert ist.

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

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

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

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

Des Weiteren ist zu beobachten, dass wenn bei einer Funktion mit dem **Rückgabedatentyp** `void` kein **return**-Statement explizit ans Ende geschrieben wird, im **PicoC-ANF Pass** eines hinzugefügt wird in Form der Komposition `Return(Empty())`. Beim Nicht-Angeben im Falle eines Dantentyps, der **nicht** `void` ist, wird 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       [
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'), 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'), GoTo(Name('addr@next_instr')))
30        Exp(GoTo(Name('fun_no_return_value.1')))
31        RemoveStackframe()
32        Return(Empty())
33      ]
34    ]

```

Code 0.19: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.20 werden die Kompositionen `Exp(Num('21'))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))`, `Return(Stack(Num('1')))` und `Assign(Global(Num('0')), Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

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'), 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'), 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 0.20: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.*

0.0.1.3.2 Umsetzung der Übergabe eines Feldes

Die **Call by Reference** (Definition ??) Übergabe eines Feldes an eine andere Funktion, wird im Folgenden mithilfe des Beispiels in Code 0.21 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 0.21: *PicoC-Code für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** wird im Fall dessen, dass der **oberste Knoten** im Teilbaum, der den Datentyp darstellt und an die Funktion übergeben wird ein **Feld** `ArrayDecl(nums, datatype)` ist, dieser zu einem **Zeiger** `PntrDecl(num, datatype)` umgewandelt und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, an das `datatype`-Attribut des **Zeigers** `PntrDecl(num, datatype)` drangehängt.

Diese **Umwandlung** des **Datentyps** kann in der **Symboltabelle** in Code 0.22 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 ändert sich der Datentyp beider Variablen zu `PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))`. Die **Größe** dieser Variablen ändert sich damit zu `Num('1')`, da ein **Zeiger** nur eine **Speicherzelle** braucht.

```

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

```

```

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

```

```

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 0.22: *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 0.23 ist zu sehen, dass zur Übergabe der beiden Felder die **Adresse** des jeweiligen Feldes auf den **Stack** geschrieben wird. Die **Adressen** der beiden Felder auf den **Stack** zu schreiben wird durch die Kompositionen `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` repräsentiert.

Die Komposition `Ref(Global(Num('0')))` ist für Variablen in den **Globalen Statischen Daten** und die Komposition `Ref(Stackframe(Num('6')))` ist für Variablen aus dem **Stackframe**. Dabei stellen die Zahlen in den Knoten `Global(num)` bzw. `Stackframe(num)` die **relative Adressen** relativ zum DS-Register bzw. SP-Register dar, die aus der **Symboltabelle** entnommen sind.

```

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'), 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'), GoTo(Name('addr@next_instr'))
25        Exp(GoTo(Name('fun_array_from_global_data.1'))))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]

```

Code 0.23: *PicoC-ANF Pass .*

Im **RETI-Blocks Pass** in Code 0.24 werden Kompositionen `Ref(Global(Num('0')))` und

`Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RET-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'), 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.0',
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'), 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 0.24: *RETI-Block Pass für die Übergabe eines Feldes.*

0.0.1.3.3 Umsetzung einer Übergabe eines Verbundes

Die **Call by Value** (Definition ??) Übergabe eines **Structs** wird im Folgenden mithilfe des Beispiels in Code 0.25 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 0.25: *PicoC-Code für Call by Value für Structs.*

Im **PicoC-ANF Pass** in Code 0.26 wird zur **Übergabe eines Struct**, das komplette Struct auf den **Stack** kopiert. Das wird mittels der Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` dargestellt.

Bei der **Übergabe** an eine **Funktion** wird der Zugriff auf ein gesamtes **Struct** anders gehandhabt als sonst. Normalerweise wird beim Zugriff auf ein Struct die **Adresse** des **ersten Attributs** dieses Structs auf den **Stack** geschrieben. Bei der **Übergabe an eine Funktion** wird dagegen das gesamte **Struct** auf den **Stack** kopiert.

Das wird durch eine Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange der **Funktionsaufruf** im **Picoc-ANF Pass** verarbeitet wird und wieder auf `false` gesetzt, wenn die Verarbeitung des **Funktionsaufrufs** abgeschlossen ist. Solange die Variable `argmode_on` auf `true` gesetzt ist, wird immer die Komposition `Assign(Stack(Num('3')), Global(Num('0')))` bzw. der Komposition `Assign(Stack(Num('3')),`

`Stackframe(Num('2'))` für die Ersetzung verwendet. Ist die Variable `argmode_on` auf `false` wird die Komposition `Ref(Globalnum())` bzw. `Ref(Stackframe(num))` für die Ersetzung verwendet.

Die Komposition `Assign(Stack(Num('3')), Stackframe(Num('2')))` wird im Falle dessen, dass die Structvariable in den **Globalen Statischen Daten** liegt verwendet und die Komposition `Assign(Stack(Num('3')), Global(Num('0')))` wird im Falle, dessen, dass die Structvariable im **Stackframe** liegt verwendet.

```

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'), GoTo(Name('addr@next_instr'))
15        Exp(GoTo(Name('fun_struct_from_stackframe.2'))))
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'), GoTo(Name('addr@next_instr'))
25        Exp(GoTo(Name('fun_struct_from_global_data.1'))))
26        RemoveStackframe()
27        Return(Empty())
28      ]
29  ]

```

Code 0.26: *PicoC-ANF Pass für Call by Value für Structs.*

Im **RETI-Blocks Pass** in Code 0.27 werden die Kompositionen `Assign(Stack(Num('3')), Stackframe(Num('2')))` und `Assign(Stack(Num('3')), Global(Num('0')))` durch ihre 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'), 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'), 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 0.27: *RETI-Block Pass für Call by Value für Structs.*

Literatur