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

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

Bevor der Inhalt dieser Schrifftlichen 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 Selbverständlichkeit 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 heraus, 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 Leidenschafft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias[1] 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[2], er hat sich bei der Korrektur dieser Schrifftlichen 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 Selbverständlichkeit, 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 dagegeben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

[1] Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.
[2] Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil[3] [4] 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 Selbverständlichkeit, 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 ausgeinandergesetzt 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[5] zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt[6]. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht[7]. 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 Programmiesprache kann in der Verwendung ziemlich wilkürlich 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.

[3] Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.
[4] 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 ☺.
[5] https://github.com/michel-giehl/Reti-Emulator.
[6] Womit nicht alle Studenten so viel Glück haben.
[7] Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 Umsetzung von Zeigern

Die **Zeigern** ist in diesem Unterkapitel schnell erklärt, auch Dank eines kleinen **Taschenspielertricks**[8]. Hierbei sind nur die **Operationen** für **Referenzierung** und **Dereferenzierung** in den Unterkapiteln 0.0.1.1 und 0.0.1.2 zu erläutern. **Referenzierung** kann dazu genutzt werden einen **Zeiger** zu **initialisieren** und **Dereferenzierung** kann dazu genutzt werden, um auf diesen später zuzugreifen.

0.0.1.1 Referenzierung

Die **Referenzierung** (z.B. `&var`) ist eine **Operation** bei der ein **Zeiger** auf eine **Location** (Definition **??**) in Form der **Anfangsadresse** dieser Location als Ergebnis zurückgegeben wird. Die Umsetzung der **Referenzierung** wird im Folgenden anhand des Beispiels in Code 0.1 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4 }
```

Code 0.1: *PicoC-Code für Zeigerreferenzierung.*

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 0.2 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Zeiger `*pntr`.

```
1 File
2   Name './example_pntr_ref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
          ↪  Ref(Name('var')))
11      ]
12  ]
```

Code 0.2: *Abstrakter Syntaxbaum für Zeigerreferenzierung.*

Bevor man einem **Zeiger** eine **Adresse** (z.B. `&var`) zuweisen kann, muss dieser erstmal **definiert** sein. Dafür braucht es einen Eintrag in der **Symboltabelle** in Code 0.3.

Anmerkung

Die **Anzahl Speicherzellen**[a], die ein Zeiger[b] `datatype *pntr` belegt ist dabei immer[c]: $size(type(\texttt{pntr})) = 1$ *Speicherzelle.*[def]

[a] Die im `size`-Attribut der **Symboltabelle** eingetragen ist.
[b] Z.B. ein Zeiger auf ein Feld von Integern: `int (*pntr)[3]`.
[c] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas

[8] Später mehr dazu.

unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.

[d]Eine **Speicherzelle** ist in der **RETI-Architektur**, wie in Unterkapitel ?? erklärt 4 *Byte* breit.

[e]Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein **Datentyp** belegt.

[f]Die Funktion *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('var@main')
        value or address:       Num('0')
        position:               Pos(Num('2'), Num('6'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               PntrDecl(Num('1'), IntType('int'))
        name:                   Name('pntr@main')
        value or address:       Num('1')
        position:               Pos(Num('3'), Num('7'))
        size:                   Num('1')
      }
  ]
```

Code 0.3: *Symboltabelle für Zeigerreferenzierung.*

Im **PicoC-ANF Pass** in Code 0.4 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')),Tmp(Num('1')))` ersetzt. Im Fall, dass in `Ref(exp))` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var'),Name('attr')),Num('1'))` ist, sind noch **weitere Anweisungen** zwischen den Zeilen `11` und `12` nötig. Diese weiteren Anweisungen würden sich bei z.B. `Subscr(Attr(Name('var'),Name('attr')),Num('1'))` um das Übersetzen von `Subscr(exp)` und `Attr(exp,name)` nach dem Schema in Unterkapitel 0.0.4.2 kümmern.[9]

```
File
  Name './example_pntr_ref.picoc_mon',
```

[9]Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel ??, ?? und ?? erläutert.

```
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('pntr'), Ref(Name('var')))
11        Ref(Global(Num('0')))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        Return(Empty())
14      ]
15  ]
```

Code 0.4: *PicoC-ANF Pass für Zeigerreferenzierung.*

Im **RETI-Blocks Pass** in Code 0.5 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und `Assign(Global(Num('1')),Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_pntr_ref.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('pntr'), Ref(Name('var')))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # Return(Empty())
27        LOADIN BAF PC -1;
28      ]
29  ]
```

Code 0.5: *RETI-Blocks Pass für Zeigerreferenzierung.*

0.0.1.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

Die **Dereferenzierung** (z.B. `*var`) ist eine **Operation** bei der einem **Zeiger** zur Location (Definition **??**) hin **gefolgt** wird, auf welche dieser zeigt und das Ergebnis z.B. der **Inhalt** der ersten Speicherzelle der referenzierten Location ist. Die Umsetzung von **Dereferenzierung** wird im Folgenden anhand des Beispiels in Code 0.6 erklärt.

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4   *pntr;
5 }
```

Code 0.6: *PicoC-Code für Zeigerdereferenzierung.*

Der Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 0.7 eine **Dereferenzierung** `*var`. Es gibt hierbei 3 Fälle. Bei der Anwendung von **Zeigerarithmetik**, wie z.B. `*(var + 2 - 1)` übersetzt sich diese zu `Deref(Name('var'), BinOp(Num('2'), Sub(), Num('1')))` und bei z.B. `*(var - 2 - 1)` zu `Deref(Name('var'), UnOp(Minus(), BinOp(Num('2'), Sub(), Num('1'))))`. Bei einer normalen **Dereferenzierung**, wie z.B. `*var`, übersetzt sich diese zu `Deref(Name('var'), Num('0'))`[10].

```
1 File
2   Name './example_pntr_deref.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
          ↪  Ref(Name('var')))
11        Exp(Deref(Name('pntr'), Num('0')))
12      ]
13  ]
```

Code 0.7: *Abstrakter Syntaxbaum für Zeigerdereferenzierung.*

Im **PicoC-Shrink Pass** in Code 0.8 wird ein **Trick** angewandet, bei dem jeder Knoten `Deref(exp1, exp2)` einfach durch den Knoten `Subscr(exp1, exp2)` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzierungsoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Feldindex** (z.B. `var[1]`) verhält, wie es bereits im Unterkapitel **??** erläutert wurde. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die Übersetzung der **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Feldindex** (z.B. `var[1]`) übernehmen lassen. Das Vorgehen bei der Umsetzung eines **Zugriffs auf einen Feldindex** (z.B. `*(var + 1)`) wird in Unterkapitel 0.0.2.2 erläutert.[11]

[10] Das `Num('0')` steht dafür, dass dem **Zeiger gefolgt** wird, aber danach **nicht** noch mit einem **Versatz** von der **Größe des Unterdatentyps** (Definition 0.1) auf eine nebenliegende Location zugegriffen wird.

[11] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

```
File
  Name './example_pntr_deref.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Subscr(Name('pntr'), Num('0')))
      ]
  ]
```

Code 0.8: *PicoC-Shrink Pass für Zeigerdereferenzierung.*

0.0.2 Umsetzung von Feldern

Bei Feldern ist in diesem Unterkapitel die Umsetzung der **Innitialisierung eines Feldes** 0.0.2.1, des **Zugriffs auf einen Feldindex** 0.0.2.2 und der **Zuweisung an einen Feldindex** 0.0.2.3 zu klären.

0.0.2.1 Initialisierung eines Feldes

Die Umsetzung der **Initialisierung** eines **Feldes** (z.B. `int ar[2][1] = {{3+1}, {5}}`) wird im Folgenden anhand des Beispiels in Code 0.9 erklärt.

```
void fun() {
  int ar[2][2] = {{3, 4}, {5, 6}};
}

void main() {
  int ar[2][1] = {{3+1}, {5}};
}
```

Code 0.9: *PicoC-Code für die Initialisierung eines Feldes.*

Die **Initialisierung** eines **Feldes** `intar[2][1]={{3+1},{5}}` wird im **Abstrakten Syntaxbaum** in Code 0.10 mithilfe der Knoten `Assign(Alloc(Writeable(),ArrayDecl([Num('2'),Num('1')],IntType('int')),Name('ar')),Array([Array([BinOp(Num('3'),Add('+'),Num('1'))]),Array([Num('5')])]))` dargestellt.

```
File
  Name './example_array_init.ast',
  [
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
```

```
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
          ↪  Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
10      ],
11    FunDef
12      VoidType 'void',
13      Name 'main',
14      [],
15      [
16        Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
          ↪  Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('5')])]))
17      ]
18  ]
```

Code 0.10: *Abstrakter Syntaxbaum für die Initialisierung eines Feldes.*

Bei der **Initialisierung** eines **Feldes** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[12]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 0.11 dargestellt ist.

Auf dem **Stackframe** wird ein Feld verglichen zur Wachstumrichtung des Stacks **rückwärts** in den Stackframe geschrieben und die **relative Adresse des ersten Elements** als Adresse des Feldes in der **Symboltabelle** in Code 0.11 genommen. Dies ist in Tabelle 1 für ein **Datensegment** der Größe 8 und das Beispiel aus Code 0.9 dargestellt. Es wird hier so getann als würde die **Funktion** `fun` ebenfalls **aufgerufen** werden. Der **Stack** wächst zwar verglichen zu den **Globalen Statischen Daten** in die entgegengesetzte Richtung, aber Felder in den **Globalen Statischen Daten** und in einem **Stackframe** haben die gleiche Ausrichtung. Das macht den **Zugriff auf einen Feldindex** in Unterkapitel 0.0.2.2 deutlich unkomplizierter. Auf diese Weise muss beim **Zugriff auf einen Feldindex** nicht zwischen **Stackframe** und **Globalen Statischen Daten** unterschieden werden.

Relativ-adresse	Wert	Register
0	4	CS
1	5	
...	...	
3	3	
2	4	
1	5	
0	6	
...	...	
...	...	BAF

Tabelle 1: *Datensegment nach der Initialisierung beider Felder.*

Anmerkung

Die **Anzahl Speicherzellen**, die ein Feld[a] `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_n]$ belegt[b], berechnet sich aus der **Mächtigkeit** der einzelnen **Dimensionen** des Feldes und der **Anzahl Speicherzellen**, die der

[12] Das widerspricht der üblichen Auswertungsreihenfolge beim **Zuweisungsoperator** `=`, der **rechtsassoziativ** ist. Der **Zuweisungsoperator** `=` tritt allerdings erst später in Aktion.

Datentyp, den alle **Feldelemente** haben belegt: $size(type(\texttt{ar})) = \left(\prod_{j=1}^{n} \texttt{dim}_\texttt{j}\right) \cdot size(\texttt{datatype})$.[cd]

[a]Die im `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.
[b]Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von $\texttt{L}_\texttt{C}$ übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[c]Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[d]Die **Funktion** *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

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

Code 0.11: *Symboltabelle für die Initialisierung eines Feldes.*

Im **PiocC-ANF Pass** in Code 0.12 werden zuerst die Knoten für die **Logischen Ausdrücke** in den **Blättern** des Teilbaumes, dessen Wurzel der **Feld-Initialisierer**-Knoten `Array([Array([BinOp(Num('3'), Add('+'),`

`Num('1'))]), Array([Num('5')])])` ist ausgewertet. Die Auswertung geschieht hierbei nach dem Prinzip der **Tiefensuche**, von **links-nach-rechts**. Bei dieser Auswertung werden diese Knoten für die **Logischen Ausdrücke** durch Knoten erstetzt, welche das Ergebnis dieser Ausdrücke auf den **Stack** schreiben[13].

Im finalen Schritt muss zwischen den **Globalen Statischen Daten** der `main`-Funktion und dem **Stackframe** der Funktion `fun` unterschieden werden. Die auf dem **Stack** ausgewerteten **Logischen Ausdrücke** werden mittels der Knoten `Assign(Global(Num('0')), Stack(Num('2')))` (für **Globale Statische Daten**) bzw. `Assign(Stackframe(Num('3')), Stack(Num('5')))` (für **Stackframe**) zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.[14]

Zur Veranschaulichung ist in Tabelle 2 ein **Ausschnitt des Datensegments** nach der Initialisierung des Feldes der Funktion `main`-Funktion dargestellt. Die auf den **Stack** ausgewerteten **Logischen Ausdrücke** sind in grauer Farbe markiert. Die **Kopien** dieser ausgewerteten Logischen Ausdrücke in den **Globalen Statischen Daten**, welche die einzelnen Elemente des Feldes darstellen sind in roter Farbe markiert. In Tabelle 3 ist das gleiche, allerdings für die Funktion `fun` und den **Stackframe** der Funktion `fun` dargestellt.

Relativ-adresse	Wert	Register
0	4	CS
1	5	
...	...	
1	5	
2	4	SP
...	...	

Tabelle 2: *Ausschnitt des Datensegments nach der Initialisierung des Feldes in der `main`-Funktion.*

Relativ-adresse	Wert	Register
...	...	
1	6	
2	5	
3	4	
4	3	SP
3	3	
2	4	
1	5	
0	6	
...	...	
...	...	BAF

Tabelle 3: *Ausschnitt des Datensegments nach der Initialisierung des Feldes in der Funktion `fun`.*

Der **Trick** ist hier, dass egal wieviele **Dimensionen** und was für einen **grundlegenden Datentyp**[15] das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man z.B. mit den **Knoten** `Assign(Global(Num('0')), Stack(Num('2')))` einfach so viele Speicherzellen rüberkopiert, wie das Feld **Speicherzellen** belegt.

[13] Da der **Zuweisungsoperator** `=` **rechtsassoziativ** ist und auch rein **logisch**, weil man nichts zuweisen kann, was man noch nicht berechnet hat.

[14] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

[15] Z.B. ein **Verbund**, sodass es ein **„Feld von Verbunden“** ist.

In die Knoten `Global('0')` und `Stackframe('3')` wird hierbei die **Startadresse** des jeweiligen Feldes geschrieben. Daher müssen nach dem **PicoC-ANF Pass** nie mehr Variablen in der **Symboltabelle** nachgesehen werden und es ist möglich direkt abzulesen, ob diese in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

```
File
  Name './example_array_init.picoc_mon',
  [
    Block
      Name 'fun.1',
      [
        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        Exp(Num('3'))
        Exp(Num('4'))
        Exp(Num('5'))
        Exp(Num('6'))
        Assign(Stackframe(Num('3')), Stack(Num('4')))
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('5')])]))
        Exp(Num('3'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Exp(Num('5'))
        Assign(Global(Num('0')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 0.12: *PicoC-ANF Pass für die Initialisierung eines Feldes.*

Im **RETI-Blocks Pass** in Code 0.13 werden die **PicoC-Knoten** `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('5')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'fun.1',
      [
        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
        ↪  Num('6')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('4'))
```

```
13        SUBI SP 1;
14        LOADI ACC 4;
15        STOREIN SP ACC 1;
16        # Exp(Num('5'))
17        SUBI SP 1;
18        LOADI ACC 5;
19        STOREIN SP ACC 1;
20        # Exp(Num('6'))
21        SUBI SP 1;
22        LOADI ACC 6;
23        STOREIN SP ACC 1;
24        # Assign(Stackframe(Num('3')), Stack(Num('4')))
25        LOADIN SP ACC 1;
26        STOREIN BAF ACC -2;
27        LOADIN SP ACC 2;
28        STOREIN BAF ACC -3;
29        LOADIN SP ACC 3;
30        STOREIN BAF ACC -4;
31        LOADIN SP ACC 4;
32        STOREIN BAF ACC -5;
33        ADDI SP 4;
34        # Return(Empty())
35        LOADIN BAF PC -1;
36      ],
37    Block
38      Name 'main.0',
39      [
40        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('5')])]))
41        # Exp(Num('3'))
42        SUBI SP 1;
43        LOADI ACC 3;
44        STOREIN SP ACC 1;
45        # Exp(Num('1'))
46        SUBI SP 1;
47        LOADI ACC 1;
48        STOREIN SP ACC 1;
49        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
50        LOADIN SP ACC 2;
51        LOADIN SP IN2 1;
52        ADD ACC IN2;
53        STOREIN SP ACC 2;
54        ADDI SP 1;
55        # Exp(Num('5'))
56        SUBI SP 1;
57        LOADI ACC 5;
58        STOREIN SP ACC 1;
59        # Assign(Global(Num('0')), Stack(Num('2')))
60        LOADIN SP ACC 1;
61        STOREIN DS ACC 1;
62        LOADIN SP ACC 2;
63        STOREIN DS ACC 0;
64        ADDI SP 2;
65        # Return(Empty())
66        LOADIN BAF PC -1;
67      ]
68  ]
```

Code 0.13: *RETI-Blocks Pass für die Initialisierung eines Feldes.*

0.0.2.2 Zugriff auf einen Feldindex

Die Umsetzung des **Zugriffs auf einen Feldindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 0.14 erklärt.

```
1 void fun() {
2   int ar[1] = {42};
3   ar[0];
4 }
5
6 void main() {
7   int ar[3] = {1, 2, 3};
8   ar[1+1];
9 }
```

Code 0.14: *PicoC-Code für Zugriff auf einen Feldindex.*

Der **Zugriff auf einen Feldindex** `ar[0]` wird im **Abstrakten Syntaxbaum** in Code 0.15 mithilfe des Knotens `Subscr(Name('ar'), Num('0'))` dargestellt.

```
1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'fun',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
          ↪  Array([Num('42')]))
10        Exp(Subscr(Name('ar'), Num('0')))
11      ],
12    FunDef
13      VoidType 'void',
14      Name 'main',
15      [],
16      [
17        Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
          ↪  Array([Num('1'), Num('2'), Num('3')]))
18        Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
19      ]
20  ]
```

Code 0.15: *Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.*

Im **PicoC-ANF Pass** in Code 0.16 wird zuerst das Schreiben der **Adresse** einer Variable `Name('ar')` des Knoten `Subscr(Name('ar'), Num('0'))` auf den **Stack** dargestellt. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Knoten `Ref(Global(Num('0')))` dargestellt und beim **Stackframe**

der Funktionm `fun` wird das durch die Knoten `Ref(Stackframe(Num('2')))` dargestellt. Diese Phase wird als **Anfangsteil** 0.0.4.1 bezeichnet.

Die nächste Phase wird als **Mittelteil** 0.0.4.2 bezeichnet. In dieser Phase wird die **Adresse** ab der das **Feldelement**, des Feldes auf das zugegriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Feldes, in dem dieses **Feldelement** liegt auf den **Stack** gelegt. Ein **Index** eines Feldelements auf das zugegriffen werden soll kann auch durch das Ergebnis eines **komplexeren Ausdrucks**, wie z.B. `ar[1 + var]` bestimmt sein, in dem auch **Variablen** vorkommen. Aus diesem Grund kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll bestimmt werden, was z.B. im einfachsten Fall durch `Exp(Num('0'))` dargestellt wird. Danach kann die **Adresse des Index** berechnet werden, was durch die Knoten `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird.

In Tabelle 4 ist das ganze **veranschaulicht**. In dem Auschnitt liegt die **Startadresse** $2^{31} + 67$ des Felds `int ar[3] = {1, 2, 3}` auf dem **Stack** und darüber wurde der **Wert des Index** `[1+1]` berechnet und auf dem Stack gespeichert (in rot markiert). Der Wert des Index wurde noch **nicht** auf auf die **Startadresse** des Felds **draufaddiert**.[16]

Absolutadresse	Wert	Register
...	...	
$2^{31} + 64$	1	SP
$2^{31} + 65$	2	
$2^{31} + 66$	$2^{31} + 67$	
$2^{31} + 67$	1	
$2^{31} + 68$	2	
$2^{31} + 69$	3	
...	...	
...	...	BAF

Tabelle 4: *Ausschnitt des Datensegments bei der Adressberechnung.*

Zur **Adressberechnung** ist es notwendig auf die **Dimensionen** (z.B. `[Num('3')]`) des Feldes, auf dessen **Feldelement** zugegriffen werden soll, zugreifen zu können. Daher ist der **Felddatentyp** (z.B. `ArrayDecl([Num('3')], IntType('int'))`) dem Knoten `Ref(exp, datatype)` als verstecktes Attribut `datatype` angehängt. Das versteckte Attribut wird zuvor, während des Kompiliervorgangs im **PiocC-ANF Pass** dem Knoten `Ref(exp, datatype)` angehängt.

Je nachdem, ob mehrere `Subscr(exp,exp)` eine Komposition bilden (z.B. `Subscr(Subscr(Name('var'), Num('1')), Num('1'))`) ist es notwendig mehrere **Adressberechnungsschritte für den Index** `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` einzuleiten. Es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und einen **Zugriff auf einen Arryindex** `var[1]` miteinander zu kombinieren, was in Unterkapitel 0.0.4.2 allgemein erklärt wird.

Die letzte Phase wird als **Schlussteil** 0.0.4.3 bezeichnet. In dieser Phase wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde nun auf den **Stack** geschrieben. Hierfür wird die **Adresse**, die in den vorherigen Schritten auf dem **Stack** berechnet wurde verwendet. Beim Schreiben des **Inhalts dieses Index** auf den **Stack**, wird dieser die **Adresse** auf dem Stack ersetzen, die in den vorherigen Schritten berechnet wurde. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. In Tabelle 5 ist das ganze veranschaulicht. In rot ist der **Inhalt des Feldindex** markiert, der auf den **Stack** geschrieben wurde.

[16] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	1	
$2^{31}+65$	2	SP
$2^{31}+66$	3	
$2^{31}+67$	1	
$2^{31}+68$	2	
$2^{31}+69$	3	
...	...	
...	...	BAF

Tabelle 5: *Ausschnitt des Datensegments nach Schlussteil.*

Je nachdem auf welchen **Unterdatentyp** (Definition 0.1) im **Kontext** zuletzt zugegriffen wird, abhängig davon wird der **PicoC-Knoten** `Exp(Stack(Num('1')))` durch andere **semantisch** entsprechende **RETI-Knoten** ersetzt (siehe Unterkapitel 0.0.4.3 für genauere Erklärung). Der **Unterdatentyp** ist dabei über das versteckte Attribut `datatype` des `Exp(exp, datatype)`-Knoten zugänglich.

Definition 0.1: Unterdatentyp

***Datentyp**, der durch einen **Teilbaum** dargestellt wird. Dieser Teilbaum ist ein **Teil** eines Baumes ist, der einen gesamten **Datentyp** darstellt.*

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Feldindex** (z.B. `ar[1]`) in der `main`-Funktion oder der Funktion `fun` erfolgt, ist eigentlich nur beim **Anfangsteil**, beim Schreiben der **Adresse** der Variable `ar` auf den **Stack** zu finden. Hierbei werden, je nachdem, ob eine Variable in den **Globalen Statischen Daten** liegt oder sie auf dem **Stackframe** liegt unterschiedliche **semantisch** entsprechende **RETI-Befehle** erzeugt.

Anmerkung

Die Berechnung der **Adresse**, ab der ein **Feldelement** am Ende einer Aneinanderreihung von Zugriffen auf **Feldelemente** eines Feldes `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist[a], kann mittels der Formel 0.0.1:

$$ref(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n]) = ref(\texttt{ar}) + \left(\sum_{i=1}^{n}\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\right)\cdot size(\texttt{datatype}) \tag{0.0.1}$$

aus der Betriebssysteme Vorlesung Scholl, „Betriebssysteme“ berechnet werden[b c].

Die Knoten `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentieren dabei den Summanden für die **Anfangsadresse** $ref(\texttt{ar})$ in der Formel.

Der Knoten `Exp(num)` repräsentiert dabei einen **Index** beim **Zugriff auf ein Feldelement** (z.B. `j` in `a[i][j][k]`), der als Faktor $\texttt{idx}_i$ in der Formel auftaucht.

Die Knoten `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` repräsentieren dabei einen **ausmultiplizierten Summanden** $\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\cdot size(\texttt{datatpye})$ in der Formel.

Die Knoten `Exp(Stack(Num('1')))` repräsentieren dabei das Lesen des **Inhalts**

$M\,[ref(\texttt{ar}[\texttt{idx}_1]\ldots[\texttt{idx}_n])]$ der Speicherzelle an der finalen **Adresse** $ref(\texttt{ar}[\texttt{idx}_1]\ldots[\texttt{idx}_n])$.

[a]Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[b]ref(`exp`) steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte.
[c]Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.

```
File
  Name './example_array_access.picoc_mon',
  [
    Block
      Name 'fun.1',
      [
        // Assign(Name('ar'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        // Exp(Subscr(Name('ar'), Num('0')))
        Ref(Stackframe(Num('0')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ],
    Block
      Name 'main.0',
      [
        // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
        Exp(Num('1'))
        Exp(Num('2'))
        Exp(Num('3'))
        Assign(Global(Num('0')), Stack(Num('3')))
        // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
        Ref(Global(Num('0')))
        Exp(Num('1'))
        Exp(Num('1'))
        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 0.16: *PicoC-ANF Pass für Zugriff auf einen Feldindex.*

Im **RETI-Blocks Pass** in Code 0.17 werden die **PicoC-Knoten** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2'))undStack(Num('1'))))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
```

```
5       Name 'fun.1',
6       [
7         # // Assign(Name('ar'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Stackframe(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN BAF ACC -2;
15        ADDI SP 1;
16        # // Exp(Subscr(Name('ar'), Num('0')))
17        # Ref(Stackframe(Num('0')))
18        SUBI SP 1;
19        MOVE BAF IN1;
20        SUBI IN1 2;
21        STOREIN SP IN1 1;
22        # Exp(Num('0'))
23        SUBI SP 1;
24        LOADI ACC 0;
25        STOREIN SP ACC 1;
26        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        LOADIN SP IN1 2;
28        LOADIN SP IN2 1;
29        MULTI IN2 1;
30        ADD IN1 IN2;
31        ADDI SP 1;
32        STOREIN SP IN1 1;
33        # Exp(Stack(Num('1')))
34        LOADIN SP IN1 1;
35        LOADIN IN1 ACC 0;
36        STOREIN SP ACC 1;
37        # Return(Empty())
38        LOADIN BAF PC -1;
39      ],
40    Block
41      Name 'main.0',
42      [
43        # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44        # Exp(Num('1'))
45        SUBI SP 1;
46        LOADI ACC 1;
47        STOREIN SP ACC 1;
48        # Exp(Num('2'))
49        SUBI SP 1;
50        LOADI ACC 2;
51        STOREIN SP ACC 1;
52        # Exp(Num('3'))
53        SUBI SP 1;
54        LOADI ACC 3;
55        STOREIN SP ACC 1;
56        # Assign(Global(Num('0')), Stack(Num('3')))
57        LOADIN SP ACC 1;
58        STOREIN DS ACC 2;
59        LOADIN SP ACC 2;
60        STOREIN DS ACC 1;
61        LOADIN SP ACC 3;
```

```
62          STOREIN DS ACC 0;
63          ADDI SP 3;
64          # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65          # Ref(Global(Num('0')))
66          SUBI SP 1;
67          LOADI IN1 0;
68          ADD IN1 DS;
69          STOREIN SP IN1 1;
70          # Exp(Num('1'))
71          SUBI SP 1;
72          LOADI ACC 1;
73          STOREIN SP ACC 1;
74          # Exp(Num('1'))
75          SUBI SP 1;
76          LOADI ACC 1;
77          STOREIN SP ACC 1;
78          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79          LOADIN SP ACC 2;
80          LOADIN SP IN2 1;
81          ADD ACC IN2;
82          STOREIN SP ACC 2;
83          ADDI SP 1;
84          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85          LOADIN SP IN1 2;
86          LOADIN SP IN2 1;
87          MULTI IN2 1;
88          ADD IN1 IN2;
89          ADDI SP 1;
90          STOREIN SP IN1 1;
91          # Exp(Stack(Num('1')))
92          LOADIN SP IN1 1;
93          LOADIN IN1 ACC 0;
94          STOREIN SP ACC 1;
95          # Return(Empty())
96          LOADIN BAF PC -1;
97        ]
98    ]
```

Code 0.17: *RETI-Blocks Pass für Zugriff auf einen Feldindex.*

0.0.2.3 Zuweisung an Feldindex

Die Umsetzung einer **Zuweisung** eines Wertes an einen **Feldindex** (z.B. `ar[2] = 42;`) wird im Folgenden anhand des Beispiels in Code 0.18 erläutert.

```
1 void main() {
2   int ar[2];
3   ar[1] = 42;
4 }
```

Code 0.18: *PicoC-Code für Zuweisung an Feldindex.*

Im **Abstrakten Syntaxbaum** in Code 0.19 wird eine **Zuweisung** an einen **Feldindex** `ar[2] = 42;` durch

die Knoten `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

```
File
  Name './example_array_assignment.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        Assign(Subscr(Name('ar'), Num('1')), Num('42'))
      ]
  ]
```

Code 0.19: *Abstrakter Syntaxbaum für Zuweisung an Feldindex.*

Im **PicoC-ANF Pass** in Code 0.20 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=` bzw. des Knotens der diesen darstellt ausgewertet: `Exp(Num('42'))`. Dies ist in Tabelle 6 für das Beispiel in Code 0.18 veranschaulicht. Der **Wert** 42 (in rot markiert) wurde auf den **Stack** geschrieben.

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	...	
$2^{31}+65$	...	SP
$2^{31}+66$	42	
$2^{31}+67$	...	
$2^{31}+68$	...	
$2^{31}+69$	...	
...	...	
...	...	BAF

Tabelle 6: *Ausschnitt des Datensegments nach Auswerten der rechten Seite.*

Danach ist das Vorgehen und die damit verbundenen Knoten, die dieses Vorgehen darstellen: `Ref(Global(Num('0')))`, `Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum **Anfangsteil** und **Mittelteil** aus dem vorherigen Unterkapitel 0.0.2.2. Die eben genannten Knoten stellen die Berechnung der **Adresse** des **Index**, dem das Ergebnis des Logischen Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird dar. Dies ist in Tabelle 7 für das Beispiel in Code 0.18 veranschaulicht. Die **Adresse** $2^{31}+68$ (in rot markiert) des Index wurde auf dem **Stack** berechnet.

Absolutadresse	Wert	Register
...	...	
$2^{31} + 64$	1	SP
$2^{31} + 65$	$2^{31} + 68$	
$2^{31} + 66$	42	
$2^{31} + 67$	...	
$2^{31} + 68$	...	
$2^{31} + 69$	...	
...	...	
...	...	BAF

Tabelle 7: *Ausschnitt des Datensegments vor Zuweisung.*

Zum Schluss stellen die Knoten `Assign(Stack(Num('1')), Stack(Num('2')))` die **Zuweisung** `stack(1) = stack(2)` des Ergebnisses des Ausdrucks auf der **rechten Seite der Zuweisung** zum **Feldindex** dar. Die **Adresse** des Feldindex wurde im Schritt davor berechnet. Die **Zuweisung** des Wertes 42 an den **Feldindex** `[1]` ist in Tabelle 8 veranschaulicht (in rot markiert).[17]

Absolutadresse	Wert	Register
...	...	
$2^{31} + 64$	1	
$2^{31} + 65$	$2^{31} + 68$	
$2^{31} + 66$	42	SP
$2^{31} + 67$	...	
$2^{31} + 68$	42	
$2^{31} + 69$	...	
...	...	
...	...	BAF

Tabelle 8: *Ausschnitt des Datensegments nach Zuweisung.*

```
1  File
2    Name './example_array_assignment.picoc_mon',
3    [
4      Block
5        Name 'main.0',
6        [
7          // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
8          Exp(Num('42'))
9          Ref(Global(Num('0')))
10         Exp(Num('1'))
11         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
12         Assign(Stack(Num('1')), Stack(Num('2')))
13         Return(Empty())
14       ]
15   ]
```

Code 0.20: *PicoC-ANF Pass für Zuweisung an Feldindex.*

[17] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

Im **RETI-Blocks Pass** in Code 0.21 werden die **PicoC-Knoten** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Exp(Num('1'))`, `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` und `Assign(Stack(Num('1')),Stack(Num('2')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        LOADIN SP IN1 2;
        LOADIN SP IN2 1;
        MULTI IN2 1;
        ADD IN1 IN2;
        ADDI SP 1;
        STOREIN SP IN1 1;
        # Assign(Stack(Num('1')), Stack(Num('2')))
        LOADIN SP IN1 1;
        LOADIN SP ACC 2;
        ADDI SP 2;
        STOREIN IN1 ACC 0;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 0.21: *RETI-Blocks Pass für Zuweisung an Feldindex.*

0.0.3 Umsetzung von Verbunden

Bei Verbunden wird in diesem Unterkapitel zunächst geklärt, wie die **Deklaration von Verbundstypen** umgesetzt ist. Ist ein **Verbundstyp** deklariert, kann damit einhergehend ein **Verbund** mit diesem Verbundstyp **definiert** werden. Die Umsetzung von beidem wird in Unterkapitel 0.0.3.1 erläutert. Des Weiteren ist die Umsetzung der **Innitialisierung eines Verbundes** 0.0.3.2, des **Zugriffs auf ein Verbundsattribut** 0.0.3.3 und der **Zuweisung an ein Verbundsattribut** 0.0.3.4 zu klären.

0.0.3.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Umsetzung der **Deklaration** (Definition **??**) eines neuen **Verbundstyps** (z.B. `struct st {int len; int ar[2];}`) und der **Definition** (Definition **??**) eines **Verbundes** mit diesem **Verbundstyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 0.22 erläutert.

```
1 struct st {int len; int ar[2];};
2
3 void main() {
4   struct st st_var;
5 }
```

Code 0.22: *PicoC-Code für die Deklaration eines Verbundstyps.*

Bevor ein Verbund definiert werden kann, muss erstmal ein **Verbundstyp** deklariert werden. Im **Abstrakten Syntaxbaum** in Code 0.24 wird die **Deklaration eines Verbundstyps** `struct st {int len; int ar[2];}` durch die Knoten `StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` dargestellt.

Die **Definition** einer Variable mit diesem **Verbundstyp** `struct st st_var;` wird durch die Knoten `Alloc(Writeable(), StructSpec(Name('st')), Name('st_var'))` dargestellt.

```
1  File
2    Name './example_struct_decl_def.ast',
3    [
4      StructDecl
5        Name 'st',
6        [
7          Alloc(Writeable(), IntType('int'), Name('len'))
8          Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Exp(Alloc(Writeable(), StructSpec(Name('st')), Name('st_var')))
16       ]
17   ]
```

Code 0.23: *Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.*

Für den **Verbundstyp** selbst und seine **Verbundsattribute** werden in der **Symboltabelle**, die in Code 0.24 dargestellt ist Symboltabelleneintrage mit den **Schlüsseln** `st`, `len@st` und `ar@st` erstellt. Die Schlüssel der

Verbundsattribute haben einen **Suffix** `@st` angehängt, welcher für die Verbundsattribute einen Verbundstyps **indirekt** einen **Sichtbarkeitsbereich** (Definition **??**) über den **Verbundstyp** selbst erzeugt. Im Unterkapitel 0.0.5.2 wird die **Funktionsweise** von **Sichtbarkeitsbereichen** genauer erläutert. Es gilt folglich, dass **innerhalb** eines **Verbundstyps** zwei Verbundsattribute **nicht** gleich benannt werden können, aber dafür zwei **unterschiedliche Verbundstypen** ihre Verbundsattribute gleich benennen können.

Das Symbol `'@'` wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel **??** erläutert wurde.

Die Attribute[18] der Symboltabelleneinträge für die **Verbundsattribute** sind genauso belegt wie bei **üblichen Variablen**. Die Attribute des Symboltabelleneintrags für den **Verbundstyp** `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt. In dem `value_address`-Attribut des Symboltabelleneintrags für den **Verbundstyp** sind die **Verbundsattribute** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Verbundstyp** `st` als Schlüssel die **Verbundsattribute** des Verbundstyps in der **Symboltabelle** nachschlagen kann.

Für die **Definition** einer Variable `st_var@main` mit diesem **Verbundstyp** `st` wird ein **Symboltabelleneintrag** in der **Symboltabelle** angelegt. Das `datatype`-Attribut dieses **Symboltabelleneintrags** enthält dabei den Namen des **Verbundstyps** als `StructSpec(Name('st'))`. Dadurch können jederzeit alle wichtigen Informationen zu diesem **Verbundstyp**[19] und seinen **Verbundsattributen** in der **Symboltabelle** nachgeschlagen werden.

Anmerkung

Die **Anzahl Speicherzellen**, die ein **Verbund** `struct st st_var` belegt[a], der mit dem **Verbundstyp** `struct st` {$\texttt{datatype}_1$ $\texttt{attr}_1$; ...; $\texttt{datatype}_n$ $\texttt{attr}_n$;} definiert ist[b], berechnet sich aus der Summe der **Anzahl Speicherzellen**, welche die einzelnen **Datentypen** $\texttt{datatype}_1$ ... $\texttt{datatype}_n$ der **Verbundsattribute** $\texttt{attr}_1$, ... $\texttt{attr}_n$ des **Verbundstyps** belegen: $size(type(\texttt{st\_var})) = \sum_{i=1}^{n} size(\texttt{datatype}_i)$.[c][d]

[a] Die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.
[b] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} **nicht** die manchmal etwas unpraktische Designentscheidung, die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von L_C übernommen. Es wird so getan, als würde der restliche **Datentyp** komplett **vor** der Variable stehen: `datatype var`.
[c] Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein **Datentyp** belegt.
[d] Die Funktion *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:         Empty()
6          datatype:               IntType('int')
7          name:                   Name('len@st')
8          value or address:       Empty()
9          position:               Pos(Num('1'), Num('15'))
10         size:                   Num('1')
11       },
12     Symbol
13       {
```

[18] Die über einen **Bezeichner** selektierbaren Elemente eines **Symboltabelleneintrags** und eines **Verbunds** heißen bei beiden **Attribute**.
[19] Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Verbundstyp** einnimmt.

```
14        type qualifier:         Empty()
15        datatype:               ArrayDecl([Num('2')], IntType('int'))
16        name:                   Name('ar@st')
17        value or address:       Empty()
18        position:               Pos(Num('1'), Num('24'))
19        size:                   Num('2')
20      },
21    Symbol
22      {
23        type qualifier:         Empty()
24        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
          ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
          ↪  Name('ar'))])
25        name:                   Name('st')
26        value or address:       [Name('len@st'), Name('ar@st')]
27        position:               Pos(Num('1'), Num('7'))
28        size:                   Num('3')
29      },
30    Symbol
31      {
32        type qualifier:         Empty()
33        datatype:               FunDecl(VoidType('void'), Name('main'), [])
34        name:                   Name('main')
35        value or address:       Empty()
36        position:               Pos(Num('3'), Num('5'))
37        size:                   Empty()
38      },
39    Symbol
40      {
41        type qualifier:         Writeable()
42        datatype:               StructSpec(Name('st'))
43        name:                   Name('st_var@main')
44        value or address:       Num('0')
45        position:               Pos(Num('4'), Num('12'))
46        size:                   Num('3')
47      }
48  ]
```

Code 0.24: *Symboltabelle für die Deklaration eines Verbundstyps.*

0.0.3.2 Initialisierung von Verbunden

Die Umsetzung der **Initialisierung eines Verbundes** wird im Folgenden mithilfe des Beispiels in Code 0.25 erklärt.

```
1 struct st1 {int *attr[2];};
2
3 struct st2 {int attr1; struct st1 attr2;};
4
5 void main() {
6   int var = 42;
7   struct st2 st = {.attr1=var, .attr2={.attr={&var, &var}}};
8 }
```

Code 0.25: *PicoC-Code für Initialisierung von Verbunden.*

Im **Abstrakten Syntaxbaum** in Code 0.26 wird die **Initialisierung eines Verbundes** `struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}}` mithilfe der **Knoten** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` dargestellt.

```
File
  Name './example_struct_init.ast',
  [
    StructDecl
      Name 'st1',
      [
        Alloc(Writeable(), ArrayDecl([Num('2')], PntrDecl(Num('1'), IntType('int'))),
        ↪  Name('attr'))
      ],
    StructDecl
      Name 'st2',
      [
        Alloc(Writeable(), IntType('int'), Name('attr1'))
        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
        ↪  Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
        ↪  Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))])])])])])
      ]
  ]
```

Code 0.26: *Abstrakter Syntaxbaum für Initialisierung von Verbunden.*

Im **PicoC-ANF Pass** in Code 0.27 wird `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Feldes** in Unterkapitel 0.0.2.1. Für **genauere Details** wird an dieser Stelle daher auf Unterkapitel 0.0.2.1 verwiesen. Um das Ganze interessanter zu gestalten, wurde das Beispiel in Code 0.25 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der Teilbaum `Struct([Assign(Name('attr1'),Name('var')),Assign(Name('attr2'),Struct([Assign(Name('attr'),Array([Array([Ref(Name('var')),Ref(Name('var'))])])])])])`, der beim äußersten **Verbund-Initialisierer**-Knoten `Struct(...)` anfängt, wird auf dieselbe Weise nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Feldes** in Unterkapitel 0.0.2.1 bereits erklärt wurde. Beim **Iterieren** über den **Teilbaum**, muss bei einem **Verbund-Initialisierer**-Knoten `Struct(...)` nur beachtet werden, dass bei den `Assign(exp1, exp2)`-Knoten[20] der Teilbaum beim rechten `exp` Attribut weitergeht.

Im Allgemeinen gibt es im Teilbaum beim **Initialisieren** eines **Feldes** oder **Verbundes** auf der **rechten Seite** immer nur 3 Fälle. Auf der **rechten** Seite hat man es entweder mit einem **Verbund-Initialiser**, einem

[20] Über welche die **Attributzuweisung** (z.B. `attr1=var`) als z.B. `Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])])]))` dargestellt wird.

Feld-Initialiser oder einem **Logischen Ausdruck** zu tun. Bei einem **Feld-** oder **Verbund-Initialiser** wird über diesen nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** iteriert und mithilfe von `Exp(exp)`-Knoten die Auswertung der **Logischen Ausdrücke** in den **Blättern** auf den **Stack** dargestellt. Der Fall, dass ein **Logischer Ausdruck** vorliegt erübrigt sich hiermit.

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
        ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
        ↪  Ref(Name('var'))]))]))]))
        Exp(Global(Num('0')))
        Ref(Global(Num('0')))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('3')))
        Return(Empty())
      ]
  ]
```

Code 0.27: *PicoC-ANF Pass für Initialisierung von Verbunden.*

Im **RETI-Blocks Pass** in Code 0.28 werden die **PicoC-Knoten** `Exp(Global(Num('0')))`, `Ref(Global(Num('0')))` und `Assign(Global(Num('1')),Stack(Num('3')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_init.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
        ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var')),
        ↪  Ref(Name('var'))]))]))]))
        # Exp(Global(Num('0')))
        SUBI SP 1;
        LOADIN DS ACC 0;
        STOREIN SP ACC 1;
```

```
21        # Ref(Global(Num('0')))
22        SUBI SP 1;
23        LOADI IN1 0;
24        ADD IN1 DS;
25        STOREIN SP IN1 1;
26        # Ref(Global(Num('0')))
27        SUBI SP 1;
28        LOADI IN1 0;
29        ADD IN1 DS;
30        STOREIN SP IN1 1;
31        # Assign(Global(Num('1')), Stack(Num('3')))
32        LOADIN SP ACC 1;
33        STOREIN DS ACC 3;
34        LOADIN SP ACC 2;
35        STOREIN DS ACC 2;
36        LOADIN SP ACC 3;
37        STOREIN DS ACC 1;
38        ADDI SP 3;
39        # Return(Empty())
40        LOADIN BAF PC -1;
41      ]
42  ]
```

Code 0.28: *RETI-Blocks Pass für Initialisierung von Verbunden.*

0.0.3.3 Zugriff auf Verbundsattribut

Die Umsetzung des **Zugriffs auf ein Verbundsattribut** (z.B. `st.y`) wird im Folgenden mithilfe des Beispiels in Code 0.29 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y;
6 }
```

Code 0.29: *PicoC-Code für Zugriff auf Verbundsattribut.*

Im **Abstrakten Syntaxbaum** in Code 0.30 wird der **Zugriff auf ein Verbundsattribut** `st.y` mithilfe der Knoten `Exp(Attr(Name('st'), Name('y')))` dargestellt.

```
1 File
2   Name './example_struct_attr_access.ast',
3   [
4     StructDecl
5       Name 'pos',
6       [
7         Alloc(Writeable(), IntType('int'), Name('x'))
8         Alloc(Writeable(), IntType('int'), Name('y'))
9       ],
```

```
10      FunDef
11        VoidType 'void',
12        Name 'main',
13        [],
14        [
15          Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
            ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
16          Exp(Attr(Name('st'), Name('y')))
17        ]
18    ]
```

Code 0.30: *Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 0.31 werden die Knoten `Exp(Attr(Name('st'), Name('y')))` auf eine ähnliche Weise ausgewertet, wie die Knoten `Exp(Subscr(Name('ar'), Num('0')))`, die in Unterkapitel 0.0.2.2 einen **Zugriff auf ein Feldelement** darstellen. Daher wird hier, um Redundanz zu vermeiden, nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 0.0.2.2 verwiesen.

Die Knoten `Exp(Attr(Name('st'), Name('y')))` werden genauso, wie in Unterkapitel 0.0.2.2 durch Knoten ersetzt, die sich in **Anfangsteil** 0.0.4.1, **Mittelteil** 0.0.4.2 und **Schlussteil** 0.0.4.3 aufteilen lassen. In diesem Fall sind es `Ref(Global(Num('0')))` (**Anfangsteil**), `Ref(Attr(Stack(Num('1')), Name('y')))` (**Mittelteil**) und `Exp(Stack(Num('1')))` (**Schlussteil**). Der **Anfangsteil** und **Schlussteil** sind genau gleich, wie in Unterkapitel 0.0.2.2.

Nur für den **Mittelteil** werden andere Knoten `Ref(Attr(Stack(Num('1')), Name('y')))` gebraucht. Diese Knoten `Ref(Attr(Stack(Num('1')), Name('y')))` stellen die Aufgabe dar, die **Anfangsadresse** des **Attributs** auf welches zugegriffen wird zu berechnen und auf den Stack zu legen. Hierfür wird die **Anfangsadresse** des **Verbundes**, in dem dieses Attribut liegt verwendet. Das auf den **Stack**-Speichern dieser **Anfangsadresse** wird durch Knoten des **Anfangsteils** dargestellt.

Beim **Zugriff auf einen Feldindex** muss vorher durch z.B. `Exp(Num('3'))` die **Berechnung des Indexwerts** und das auf den **Stack** legen des Ergebnisses dargestellt werden. Beim **Zugriff auf ein Verbundsattribut** steht der Bezeichner des **Verbundsattributs** `Name('y')` dagegen bereits während des Kompilierens in `Ref(Attr(Stack(Num('1')), Name('y')))` zur Verfügung. Der **Verbundstyp**, dem dieses Attribut gehört, wird im **Mittelteil** aus dem versteckten Attribut `datatype` des Knoten `Ref(exp, datatype)` herausgelesen. Der **Verbundstyp** wird während des Kompiliervorgangs im **PiocC-ANF Pass** dem Knoten `Ref(exp, datatype)` über das versteckten Attribut `datatype` angehängt.

Anmerkung

Im Unterkapitel 0.0.4.2 wird mit der **allgemeinen Formel** 0.0.3 ein allgemeines Vorgehen zur **Adressberechnung** für alle möglichen Aneinanderreihungen von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** erklärt. Um die **Adresse**, ab der ein **Verbundsattribut** am Ende einer Aneinanderreihung von Zugriffen auf **Verbundsattribute** abgespeichert ist, zu berechnen, kann diese **allgemeine Formel** 0.0.3 ebenfalls genutzt werden. Im Gegensatz zu **Feldern**, lässt sich bei Verbunden **keine** vereinfachte Formel aus der allgemeinen Formel bilden, da die **Verbundsattribute** eines Verbunds **unterschiedlich viele Speicherzellen** belegen.

```
1 File
2   Name './example_struct_attr_access.picoc_mon',
3   [
```

```
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         Exp(Num('4'))
9         Exp(Num('2'))
10        Assign(Global(Num('0')), Stack(Num('2')))
11        // Exp(Attr(Name('st'), Name('y')))
12        Ref(Global(Num('0')))
13        Ref(Attr(Stack(Num('1')), Name('y')))
14        Exp(Stack(Num('1')))
15        Return(Empty())
16      ]
17  ]
```

Code 0.31: *PicoC-ANF Pass für Zugriff auf Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 0.32 werden die **PicoC-Knoten** `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')),Name('y')))` und `Exp(Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_struct_attr_access.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
          ↪  Num('2'))]))
8         # Exp(Num('4'))
9         SUBI SP 1;
10        LOADI ACC 4;
11        STOREIN SP ACC 1;
12        # Exp(Num('2'))
13        SUBI SP 1;
14        LOADI ACC 2;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('0')), Stack(Num('2')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 1;
19        LOADIN SP ACC 2;
20        STOREIN DS ACC 0;
21        ADDI SP 2;
22        # // Exp(Attr(Name('st'), Name('y')))
23        # Ref(Global(Num('0')))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        STOREIN SP IN1 1;
28        # Ref(Attr(Stack(Num('1')), Name('y')))
29        LOADIN SP IN1 1;
30        ADDI IN1 1;
31        STOREIN SP IN1 1;
```

```
32        # Exp(Stack(Num('1')))
33        LOADIN SP IN1 1;
34        LOADIN IN1 ACC 0;
35        STOREIN SP ACC 1;
36        # Return(Empty())
37        LOADIN BAF PC -1;
38      ]
39  ]
```

Code 0.32: *RETI-Blocks Pass für Zugriff auf Verbundsattribut.*

0.0.3.4 Zuweisung an Verbundsattribut

Die Umsetzung der **Zuweisung an ein Verbundsattribut** (z.B. `st.y = 42`) wird im Folgenden anhand des Beispiels in Code 0.33 erklärt.

```
1 struct pos {int x; int y;};
2
3 void main() {
4   struct pos st = {.x=4, .y=2};
5   st.y = 42;
6 }
```

Code 0.33: *PicoC-Code für Zuweisung an Verbundsattribut.*

Im **Abstrakten Syntaxbaum** wird eine **Zuweisung an ein Verbundsattribut** `st.y = 42` durch die Knoten `Assign(Attr(Name('st'), Name('y')), Num('42'))` dargestellt.

```
1  File
2    Name './example_struct_attr_assignment.ast',
3    [
4      StructDecl
5        Name 'pos',
6        [
7          Alloc(Writeable(), IntType('int'), Name('x'))
8          Alloc(Writeable(), IntType('int'), Name('y'))
9        ],
10     FunDef
11       VoidType 'void',
12       Name 'main',
13       [],
14       [
15         Assign(Alloc(Writeable(), StructSpec(Name('pos')), Name('st')),
           ↪  Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
16         Assign(Attr(Name('st'), Name('y')), Num('42'))
17       ]
18   ]
```

Code 0.34: *Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.*

Im **PicoC-ANF Pass** in Code 0.35 werden die Knoten `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf eine ähnliche Weise ausgewertet, wie die Knoten `Assign(Subscr(Name('ar'), Num('2')), Num('42'))`, die in Unterkapitel 0.0.2.3 einen **Zugriff auf ein Feldelement** darstellen. Daher wird hier, um Redundanz zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel 0.0.2.3 verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel 0.0.2.3 muss hier zum Auswerten des **linken** Knoten `Attr(Name('st'),Name('y'))` von `Assign(Attr(Name('st'),Name('y')),Num('42'))` wie in Unterkapitel 0.0.3.3 vorgegangen werden.

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        Exp(Num('4'))
        Exp(Num('2'))
        Assign(Global(Num('0')), Stack(Num('2')))
        // Assign(Attr(Name('st'), Name('y')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 0.35: *PicoC-ANF Pass für Zuweisung an Verbundsattribut.*

Im **RETI-Blocks Pass** in Code 0.36 werden die **PicoC-Knoten** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')),Name('y')))` und `Assign(Stack(Num('1')),Stack(Num('2')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_struct_attr_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
        ↪  Num('2'))]))
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
```

```
18          STOREIN DS ACC 1;
19          LOADIN SP ACC 2;
20          STOREIN DS ACC 0;
21          ADDI SP 2;
22          # // Assign(Attr(Name('st'), Name('y')), Num('42'))
23          # Exp(Num('42'))
24          SUBI SP 1;
25          LOADI ACC 42;
26          STOREIN SP ACC 1;
27          # Ref(Global(Num('0')))
28          SUBI SP 1;
29          LOADI IN1 0;
30          ADD IN1 DS;
31          STOREIN SP IN1 1;
32          # Ref(Attr(Stack(Num('1')), Name('y')))
33          LOADIN SP IN1 1;
34          ADDI IN1 1;
35          STOREIN SP IN1 1;
36          # Assign(Stack(Num('1')), Stack(Num('2')))
37          LOADIN SP IN1 1;
38          LOADIN SP ACC 2;
39          ADDI SP 2;
40          STOREIN IN1 ACC 0;
41          # Return(Empty())
42          LOADIN BAF PC -1;
43        ]
44    ]
```

Code 0.36: *RETI-Blocks Pass für Zuweisung an Verbndsattribut.*

0.0.4 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

In den Unterkapiteln 0.0.1, 0.0.2 und 0.0.3 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln vorkommenden Datentypen (**Zeiger**, **Feld** und **Verbund**) sehr **ähnlich** abläuft. Es lässt sich ein **allgemeines Vorgehen**, bestehend aus einem **Anfangsteil** 0.0.4.1 , **Mittelteil** 0.0.4.2 und **Schlussteil** 0.0.4.3 darin erkennen. In diesem **allgemeinen Vorgehen** lassen sich die verschiedenen Zugriffsarten für Elemente bzw. Attribute von **Zeigern** (z.B. `*(pntr + i)`), **Feldern** (z.B. `ar[i]`) und **Verbunden** (z.B. `st.attr`) miteinander kombinieren und so **gemischte Ausdrücke**, wie z.B. `(*st_first.ar)[0]` bilden. Dieses **allgemeine Vorgehen** ist in Abbildung 1 veranschaulicht.

Gemischte Ausdrücke sind möglich, indem im **Mittelteil**, je nachdem, ob das versteckte Attribut `datatype` des `Ref(exp, datatype)`-Knotens ein `ArrayDecl(nums, datatype)`, ein `PntrDecl(num, datatype)` oder `StructSpec(name)` beinhaltet ein anderer **RETI-Code** generiert wird. Hierzu muss im `exp`-Attribut des `Ref(exp, datatype)`-Knoten die passende **Zugriffsoperation** `Subscr(exp1, exp2)` oder `Attr(exp, name)` vorliegen.

Der gerade erwähnte **RETI-Code** berechnet die **Startadresse** eines gewünschten **Zeigerelements**, **Feldelements** oder **Verbundsattributs**. Zur Berechnung wird die **Startadresse** des **Zeigers**, **Feldes** oder **Verbundes**, dessen **Attribut** oder **Element** berechnet werden soll verwendet. Die **Startadresse** wird in einem **vorherigen Berechnungschritt** oder im **Anfangsteil** auf den **Stack** geschrieben. Bei einem **Zugriff** auf einen **Feldindex** wird zudem mithilfe von entsprechendem **RETI-Code** dafür gesorgt, dass beim Ausführen zur **Laufzeit** der **Wert** des **Index** berechnet wird und nach der **Startadresse** auf den **Stack**

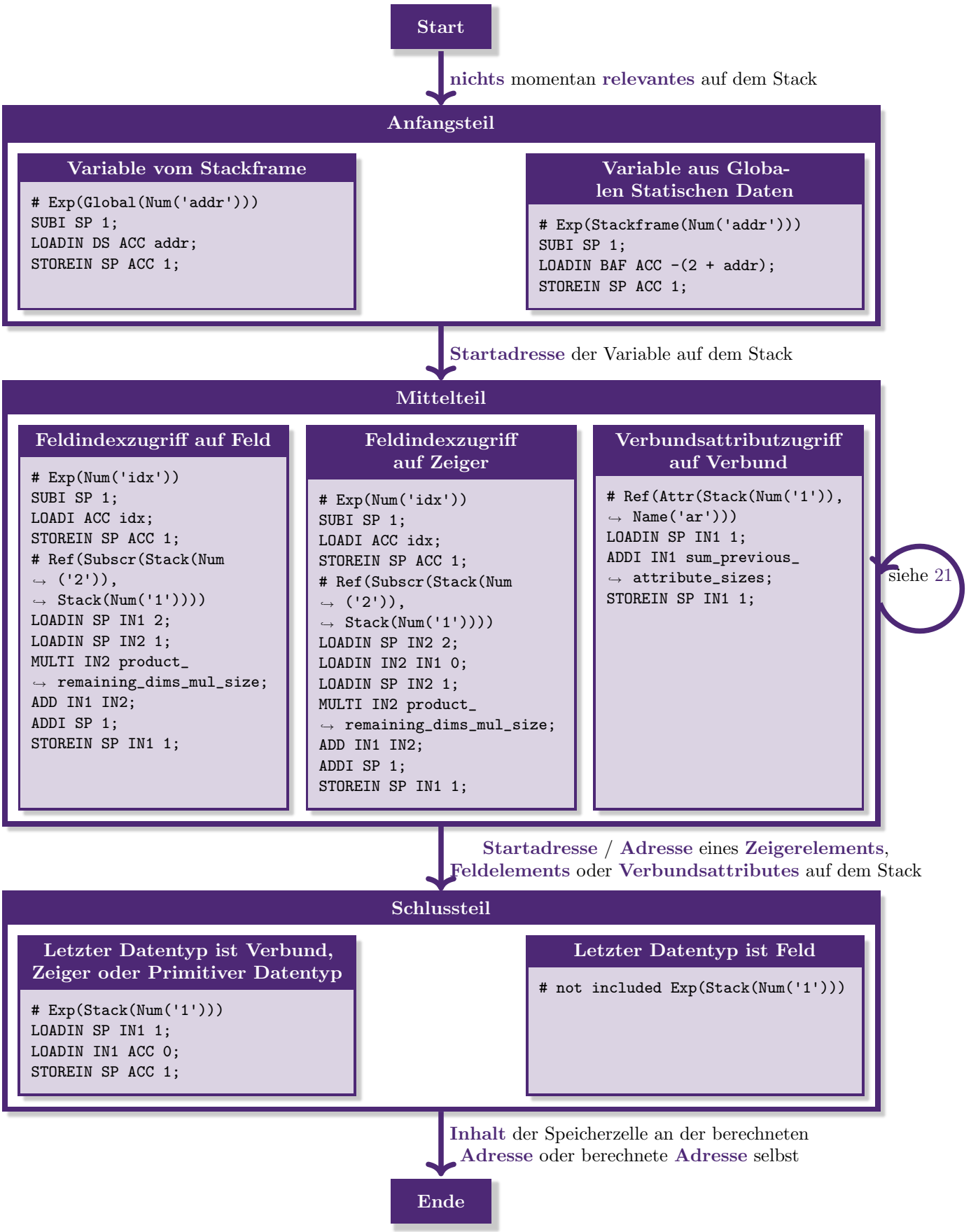


Abbildung 1: *Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.*

geschrieben wird. Dies wurde in Unterkapitel 0.0.2.2 bereits veranschaulicht.

Würde man bei einer Operation `Subsc(Name('var'), Num('0'))` den Datentyp der Variable `Name('var')` von `ArrayDecl([Num('3')], IntType())` zu `PointerDecl(Num('1'), IntType())` ändern, müssten beim generierten **RETI-Code** nur die **RETI-Befehle** des **Mittelteils** ausgetauscht werden. Die **RETI-Befehle** des **Anfangsteils** würden unverändert bleiben, da die Variable immer noch entweder in den **Globalen Statischen Daten** oder in einem **Stackframe** abgespeichert ist. Die **RETI-Befehle** des **Schlussteils** würden unverändert bleiben, da der **letzte Datentyp** auf den Zugegriffen wird immer noch `IntType()` ist.

Im `Ref(exp, datatype)`-Knoten muss die **Zugriffsoperation** im `exp`-Attribut zum **Datentyp** im versteckten Attribut `datatype` passen. Im Fall, dass Operation und Datentyp **nicht zusammenpassen**, gibt es eine `DatatypeMismatch`-**Fehlermeldung**. Ein **Zugriff auf einen Feldindex** `Subscr(exp1, exp2)` kann dabei mit den Datentypen **Feld** `ArrayDecl(nums, datatype)` und **Zeiger** `PntrDecl(num, datatype)` kombiniert werden. Allerdings wird für beide Kombinationen unterschiedlicher **RETI-Code** generiert. Das liegt daran, dass in der Speicherzelle des **Zeigers** `PntrDecl(num, datatype)` eine **Adresse** steht und das **gewünschte Element** erst zu finden ist, wenn man dieser **Adresse** folgt. Hierfür muss ein anderer **RETI-Code** erzeugt werden, wie für ein **Feld** `ArrayDecl(nums, datatype)`, bei dem direkt auf dessen **Elemente** zugegriffen werden kann. Ein **Zugriff auf ein Verbundsattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.[22]

Anmerkung

Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine **Dereferenzierung** in der Form `Deref(exp1, exp2)` nicht mehr existiert. In Unterkapitel 0.0.1.2 wurde bereits erklärt, dass alle Knoten `Deref(exp1, exp2)` im **PicoC-Shrink Pass** durch `Subscr(exp1, exp2)` ersetzt wurden. Das hatte den Zweck, **doppelten Code** zu vermeiden, da die **Dereferenzierung** und der **Zugriff auf ein Feldelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Feldindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

Der **Anfangsteil**, der durch die Knoten `Ref(Name('var'))` repräsentiert wird, ist dafür zuständig die **Startadresse** der Variablen `Name('var')` auf den **Stack** zu schreiben. Je nachdem, ob diese Variable in den **Globalen Statischen Daten** oder auf einem **Stackframe** liegt, wird ein anderer **RETI-Code** generiert.

Der **Schlussteil** wird durch die Knoten `Exp(Stack(Num('1')), datatype)` dargestellt. Wenn das versteckte Attribut `datatype` ein `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructType(name)` ist, wird ein entsprechender **RETI-Code** generiert. Dieser **RETI-Code** nutzt die **Adresse**, die in den vorherigen Phasen auf dem **Stack** berechnet wurde dazu, um den **Inhalt** der Speicherzelle an dieser **Adresse** auf den **Stack** zu schreiben. Hierbei wird die Speicherzelle, in welcher die **Adresse** steht mit dem **Inhalt** auf den sie selbst zeigt überschrieben. Bei einem `ArrayDecl(nums, datatype)` hingegen wird kein weiterer **RETI-Code** generiert, die **Adresse**, die auf dem **Stack** liegt, stellt bereits das gewünschte Ergebnis dar.

Felder haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes **Feld** zugegriffen wird[23], die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache L_{PicoC} implementieren Datentypen[24] wird immer der **Inhalt** der Speicherzelle der ersten Elements bzw. Elements ausgegeben.

0.0.4.1 Anfangsteil

Die Umsetzung des **Anfangsteils**, bei dem die **Startadresse** einer Variable auf den **Stack** geschrieben wird (z.B. `&st`), wird im Folgenden mithilfe des Beispiels in Code 0.37 erklärt.

[21] **Startadresse** / **Adresse** eines **Zeigerelements**, **Feldelements** oder **Verbundsattributes** auf dem Stack.
[22] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.
[23] Und nicht auf ein **Element** des Feldes, welches den Datentyp `CharType()` oder `IntType()`, `PntrDecl(num, datatype)` oder `StructType(name)` hat.
[24] Also `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructType(name)`.

```c
struct ar_with_len {int len; int ar[2];};

void main() {
  struct ar_with_len st_ar[3];
  int *(*complex_var)[3];
  &complex_var;
}

void fun() {
  struct ar_with_len st_ar[3];
  int (*complex_var)[3];
  &complex_var;
}
```

Code 0.37: *PicoC-Code für den Anfangsteil.*

Im **Abstrakten Syntaxbaum** in Code 0.38 wird die **Refererenzierung** `&complex_var` mit den Knoten `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird für eine **Referenzierung** einfach nur `Ref(Name('complex_var')` geschrieben, aber da beim Erstellen des **Abstrakten Syntaxbaums** jeder **Logische Ausdruck** in ein `Exp(exp)` eingebettet wird, ist das `Ref(Name('complex_var')` in ein `Exp(exp)` eingebettet. **Semantisch** macht es in diesem Zwischenschritt der Kompilierung **keinen Unterschied**, ob an einer Stelle `Ref(Name('complex_var')` oder `Exp(Ref(Name('complex_var')))` steht. Man müsste an vielen Stellen eine gesonderte **Fallunterschiedung** aufstellen, um bei `Exp(Ref(Name('complex_var')))` das `Exp(exp)` zu entfernen. Das `Exp(exp)` wird allerdings in den darauffolgenden **Passes** sowieso herausgefiltet. Daher wurde darauf verzichtet den Code ohne triftigen Grund **komplexer** zu machen.

```
File
  Name './example_derived_dts_introduction_part.ast',
  [
    StructDecl
      Name 'ar_with_len',
      [
        Alloc(Writeable(), IntType('int'), Name('len'))
        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
        ↪  IntType('int')))), Name('complex_var')))
        Exp(Ref(Name('complex_var')))
      ],
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
```

```
24          Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
            ↪  Name('st_ar')))
25          Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
            ↪  Name('complex_var')))
26          Exp(Ref(Name('complex_var')))
27        ]
28    ]
```

Code 0.38: *Abstrakter Syntaxbaum für den Anfangsteil.*

Im **PicoC-ANF Pass** in Code 0.39 werden die Knoten `Exp(Ref(Name('complex_var')))`, je nachdem, ob die Variable `Name('complex_var')` in den **Globalen Statischen Daten** oder in einem **Stackframe** liegt durch die Knoten `Ref(Global(Num('9')))` oder `Ref(Stackframe(Num('9')))` ersetzt.[25]

```
1  File
2    Name './example_derived_dts_introduction_part.picoc_mon',
3    [
4      Block
5        Name 'main.1',
6        [
7          // Exp(Ref(Name('complex_var')))
8          Ref(Global(Num('9')))
9          Return(Empty())
10       ],
11     Block
12       Name 'fun.0',
13       [
14         // Exp(Ref(Name('complex_var')))
15         Ref(Stackframe(Num('9')))
16         Return(Empty())
17       ]
18   ]
```

Code 0.39: *PicoC-ANF Pass für den Anfangsteil.*

Im **RETI-Blocks Pass** in Code 0.40 werden die **PicoC-Knoten** `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
1  File
2    Name './example_derived_dts_introduction_part.reti_blocks',
3    [
4      Block
5        Name 'main.1',
6        [
7          # // Exp(Ref(Name('complex_var')))
8          # Ref(Global(Num('9')))
9          SUBI SP 1;
10         LOADI IN1 9;
```

[25] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

```
11        ADD IN1 DS;
12        STOREIN SP IN1 1;
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16    Block
17      Name 'fun.0',
18      [
19        # // Exp(Ref(Name('complex_var')))
20        # Ref(Stackframe(Num('9')))
21        SUBI SP 1;
22        MOVE BAF IN1;
23        SUBI IN1 11;
24        STOREIN SP IN1 1;
25        # Return(Empty())
26        LOADIN BAF PC -1;
27      ]
28  ]
```

Code 0.40: *RETI-Blocks Pass für den Anfangsteil.*

0.0.4.2 Mittelteil

Der Umsetzung des **Mittelteils**, bei dem die **Startadresse** bzw. **Adresse** des letzten Attributs oder Elements einer Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute** berechnet wird (z.B. `(*complex_var.ar)[2-2]`), wird im Folgenden mithilfe des Beispiels in Code 0.41 erklärt.

```
1 struct st {int (*ar)[1];};
2
3 void main() {
4   int var[1] = {42};
5   struct st complex_var = {.ar=&var};
6   (*complex_var.ar)[2-2];
7 }
```

Code 0.41: *PicoC-Code für den Mittelteil.*

Im **Abstrakten Syntaxbaum** in Code 0.42 wird die Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** `(*complex_var.ar)[2-2]` durch die Knoten `Exp(Subscr(Deref(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2'))))` dargestellt.

```
1 File
2   Name './example_derived_dts_main_part.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
          ↪  Name('ar'))
8       ],
```

```
 9     FunDef
10       VoidType 'void',
11       Name 'main',
12       [],
13       [
14         Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
           ↪  Array([Num('42')]))
15         Assign(Alloc(Writeable(), StructSpec(Name('st')), Name('complex_var')),
           ↪  Struct([Assign(Name('ar'), Ref(Name('var')))]))
16         Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
           ↪  Sub('-'), Num('2'))))
17       ]
18   ]
```

Code 0.42: *Abstrakter Syntaxbaum für den Mittelteil.*

Im **PicoC-ANF Pass** in Code 0.43 werden die Knoten `Exp(Subscr(Deref(Attr(Name('complex_var'),Name('ar')),Num('0')),BinOp(Num('2'),Sub('-'),Num('2'))))` durch die Knoten `Ref(Attr(Stack(Num('1')),Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Sub('-'),Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` ersetzt. Bei z.B. dem `Subscr(exp1,exp2)`-Knoten wird dieser einfach dem `exp`-Attribut des `Ref(exp)`-Knoten zugewiesen und die **Indexberechnung** für `exp2` davor gezogen. Bei `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` wird über `Stack(Num('1'))` auf das Ergebnis der **Indexberechnung** auf dem **Stack** zugegriffen und über `Stack(Num('2'))` auf das Ergebnis der **Adressberechnung** auf dem **Stack** zugegriffen. Die gerade erwähnte **Indexberechnung** wird in diesem Fall durch die Knoten `Exp(Num(str))` und `Exp(BinOp(Stack(Num('2')),Sub('-'),Stack(Num('1'))))` dargestellt.

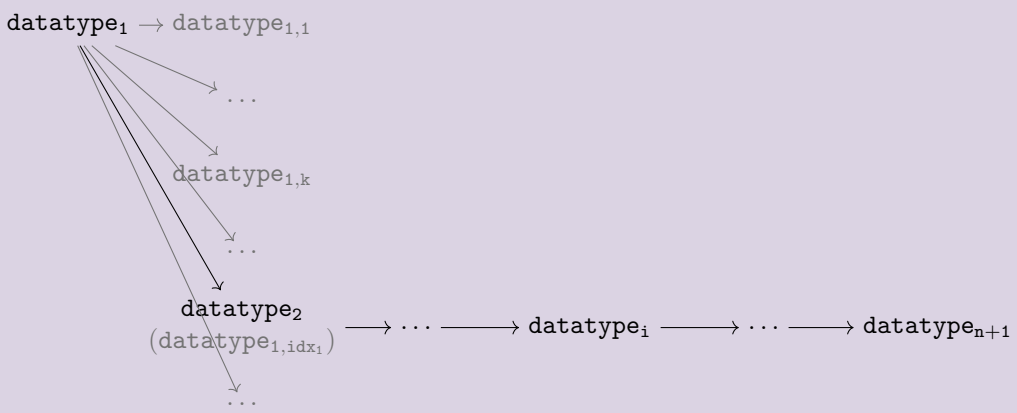
Anmerkung

Sei $\texttt{datatype}_\texttt{i}$ ein **Folgeglied** einer **Folge** $(\texttt{datatype}_\texttt{i})_{i=1,\dots,n+1}$, dessen erstes Folgeglied $\texttt{datatype}_\texttt{1}$ ist. Dabei steht `i` für eine **Ebene** eines Baumes. Die Folgeglieder der Folge lassen sich **Startadressen** $ref(\texttt{datatype}_\texttt{i})$ von Speicherbereichen $ref(\texttt{datatype}_\texttt{i})\ \dots\ ref(\texttt{datatype}_\texttt{i}) + size(\texttt{datatype}_\texttt{i})$ im **Hauptspeicher** zuordnen. Hierbei gilt, dass $ref(\texttt{datatype}_\texttt{i}) \leq ref(\texttt{datatype}_{\texttt{i+1}}) < ref(\texttt{datatype}_\texttt{i}) + size(\texttt{datatype}_\texttt{i})$.[ab]

Sei $\texttt{datatype}_{\texttt{i,k}}$ ein beliebiges **Element / Attribut** des **Datentyps** $\texttt{datatype}_\texttt{i}$. Dabei gilt: $ref(\texttt{datatype}_{\texttt{i,k}}) < ref(\texttt{datatype}_{\texttt{i,k+1}})$ und $ref(\texttt{datatype}_\texttt{i}) \leq ref(\texttt{datatype}_{\texttt{i,k}}) < ref(\texttt{datatype}_\texttt{i}) + size(\texttt{datatype}_\texttt{i})$.

Sei $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ das **Element / Attribut** des **Datentyps** $\texttt{datatype}_\texttt{i}$ für das gilt: $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}} = \texttt{datatype}_{\texttt{i+1}}$. Hierbei ist $\texttt{idx}_\texttt{i}$ der **Index**[c] des **Elements / Attributs** auf welches zugegriffen wird innerhalb des Datentyps $\texttt{datatype}_\texttt{i}$.

In Abbildung 0.0.2 ist das ganze veranschaulicht. Die ausgegrauten Knoten stellen die verschiedenen **Elemente / Attribute** $\texttt{datatype}_{\texttt{i,k}}$ des **Datentyps** $\texttt{datatype}_\texttt{i}$ dar. Allerdings können nur die Knoten $\texttt{datatype}_\texttt{i}$ **Folgeglieder** der **Folge** $(\texttt{datatype}_\texttt{i})_{i=1,\dots,n+1}$ darstellen.



(0.0.2)

Die **Adresse**, ab der ein **Element** / **Attribut** am Ende einer Folge $(\texttt{datatype}_{\texttt{i,idx}_\texttt{i}})_{i=1,\dots,n}$ verschiedener **Elemente** / **Attribute** abgespeichert ist, kann mittels der Formel 0.0.3 berechnet werden. Diese Folge ist das Resultat einer Aneinanderreihung von **Zugriffen** auf **Feldelemente** und **Verbundsattributte** unterschiedlicher Datentypen $\texttt{datatype}_\texttt{i}$ (z.B. `*complex_var.attr3[2]`).

$$ref(\texttt{datatype}_{\texttt{1,idx}_\texttt{1}},\ \dots,\ \texttt{datatype}_{\texttt{n,idx}_\texttt{n}}) = ref(\texttt{datatype}_\texttt{1}) + \sum_{i=1}^{n} \sum_{k=1}^{idx_i - 1} size(\texttt{datatype}_{\texttt{i,k}}) \quad (0.0.3)$$

Die **äußere Schleife** iteriert nacheinander über die Folge von **Attributen** / **Elementen** $(\texttt{datatype}_{\texttt{i,idx}_\texttt{i}})_{i=1,\dots,n}$, die aus den **Zugriffen** auf **Feldelemente** oder **Verbundsattribute** resultiert (z.B. `*complex_var.attr3[2]`). Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** $\texttt{datatype}_{\texttt{i,k}}$ des momentan betrachteten **Datentyps** $\texttt{datatype}_\texttt{i}$, die vor dem **Element** / **Attribut** $\texttt{datatype}_{\texttt{i,idx}_\texttt{i}}$ liegen.

Dabei darf nur das letzte Folgenglied $\texttt{datatype}_{\texttt{n+1}}$ vom Datentyp **Zeiger** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Zeiger**, der nicht der **letzte Datentyp** $\texttt{datatype}_{\texttt{n+1}}$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden. Dabei geht die **erste Adressberechnung** vom ersten Datentyp $\texttt{datatype}_\texttt{1}$ bis direkt zum Zeiger-Datentyp $\texttt{datatype}_\texttt{pntr}$ und die **zweite Adressberechnung** fängt einen Datentyp nach dem Zeiger-Datentyp $\texttt{datatype}_{\texttt{pntr+1}}$ an und geht bis zum letzten Datenyp $\texttt{datatype}_\texttt{n}$. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $ref(\texttt{datatype}_\texttt{1})$ des Summanden aus der Formel 0.0.3 auf den **Inhalt**[d] der Speicherzelle an der **Adresse**, welche in der **ersten Adressberechnung**[e] $ref(\texttt{datatype}_\texttt{1},\ \dots,\ \texttt{datatype}_\texttt{pntr})$ berechnet wurde gesetzt werden: $M\left[ref(\texttt{datatype}_\texttt{1},\ \dots,\ \texttt{datatype}_\texttt{pntr})\right]$.

Die Formel 0.0.3 stellt dabei eine **Verallgemeinerung** der Formel 0.0.1 dar, die für alle möglichen Aneinanderreihungen von Zugriffen auf **Feldelemente** und **Verbundsattribute** funktioniert (z.B. `(*complex_var.attr2)[3]`). Da die Formel **allgemein** sein muss, lässt sie sich nicht so elegant mit einem Produkt $\prod$ schreiben, wie die Formel 0.0.1, da man **nicht** davon ausgehen kann, dass alle Elemente / Attribute den **gleichen Datentyp** haben[f].

Die Knoten `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentieren dabei den Summanden $ref(\texttt{datatype}_\texttt{1})$ in der Formel.

Die Knoten `Ref(Attr(Stack(Num('1')), name))` bzw. `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))`

repräsentieren dabei einen Summanden $\sum_{k=1}^{idx_i-1} size(\texttt{datatype}_{\texttt{i,k}})$ in der Formel.

Die Knoten `Exp(Stack(Num('1')))` repräsentieren dabei das Lesen des **Inhalts** $M\,[ref(\texttt{datatype}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{datatype}_{\texttt{n,idx}_\texttt{n}})]$ der Speicherzelle an der finalen **Adresse** $ref(\texttt{datatype}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{datatype}_{\texttt{n,idx}_\texttt{n}})$.

[a] $ref(\texttt{datatype})$ ordent dabei dem **Datentyp** `datatype` eine **Startadresse** zu.
[b] Die Funktion `size` berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[c] Man fängt hier bei den **Indices** von 1 zu zählen an.
[d] Der **Inhalt** dieser Speicherzelle ist eine **Adresse**, da im momentanen Kontext ein **Zeiger** betrachtet wird.
[e] Hierbei kommt die **Adresse** des **Zeigers** selbst raus.
[f] Verbundsattribute haben z.B. **unterschiedliche** Größen.

```
File
  Name './example_derived_dts_main_part.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Array([Num('42')]))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
        ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
        Ref(Global(Num('1')))
        Ref(Attr(Stack(Num('1')), Name('ar')))
        Exp(Num('0'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Num('2'))
        Exp(Num('2'))
        Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 0.43: *PicoC-ANF Pass für den Mittelteil.*

Im **RETI-Blocks Pass** in Code 0.44 werden die **PicoC-Knoten** `Ref(Attr(Stack(Num('1')),Name('ar')))`, `Exp(Num('2'))`, `Exp(BinOp(Stack(Num('2')),Sub('-'),Stack(Num('1'))))`, `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` und `Ref(Subscr(Stack(Num('2')),Stack(Num('1'))))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt. Bei der Generierung des **RETI-Code** muss auch das versteckte Attribut `datatype` des `Ref(exp, datatpye)`-Knoten berücksichtigt werden, wie es am Anfang dieses Unterkapitels 0.0.4 zusammen mit der Abbildung 1 bereits erklärt wurde.

```
File
  Name './example_derived_dts_main_part.reti_blocks',
  [
```

```
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Array([Num('42')]))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
17        # Ref(Global(Num('0')))
18        SUBI SP 1;
19        LOADI IN1 0;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('1')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 1;
25        ADDI SP 1;
26        # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
          ↪  BinOp(Num('2'), Sub('-'), Num('2'))))
27        # Ref(Global(Num('1')))
28        SUBI SP 1;
29        LOADI IN1 1;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Ref(Attr(Stack(Num('1')), Name('ar')))
33        LOADIN SP IN1 1;
34        ADDI IN1 0;
35        STOREIN SP IN1 1;
36        # Exp(Num('0'))
37        SUBI SP 1;
38        LOADI ACC 0;
39        STOREIN SP ACC 1;
40        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
41        LOADIN SP IN2 2;
42        LOADIN IN2 IN1 0;
43        LOADIN SP IN2 1;
44        MULTI IN2 1;
45        ADD IN1 IN2;
46        ADDI SP 1;
47        STOREIN SP IN1 1;
48        # Exp(Num('2'))
49        SUBI SP 1;
50        LOADI ACC 2;
51        STOREIN SP ACC 1;
52        # Exp(Num('2'))
53        SUBI SP 1;
54        LOADI ACC 2;
55        STOREIN SP ACC 1;
56        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
57        LOADIN SP ACC 2;
58        LOADIN SP IN2 1;
59        SUB ACC IN2;
```

```
60          STOREIN SP ACC 2;
61          ADDI SP 1;
62          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63          LOADIN SP IN1 2;
64          LOADIN SP IN2 1;
65          MULTI IN2 1;
66          ADD IN1 IN2;
67          ADDI SP 1;
68          STOREIN SP IN1 1;
69          # Exp(Stack(Num('1')))
70          LOADIN SP IN1 1;
71          LOADIN IN1 ACC 0;
72          STOREIN SP ACC 1;
73          # Return(Empty())
74          LOADIN BAF PC -1;
75        ]
76    ]
```

Code 0.44: *RETI-Blocks Pass für den Mittelteil.*

0.0.4.3 Schlussteil

Die Umsetzung des **Schlussteils**, bei dem ein **Attribut** oder **Element**, dessen **Adresse** im **Anfangsteil** 0.0.4.1 und **Mittelteil** 0.0.4.2 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird[26], wird im Folgenden mithilfe des Beispiels in Code 0.45 erklärt.

```
1  struct st {int attr[2];};
2
3  void main() {
4    int complex_var1[1][2];
5    struct st complex_var2[1];
6    int var = 42;
7    int *pntr1 = &var;
8    int **complex_var3 = &pntr1;
9
10   complex_var1[0];
11   complex_var2[0];
12   *complex_var3;
13 }
```

Code 0.45: *PicoC-Code für den Schlussteil.*

Die Generierung des **Abstrakten Syntaxbaumes** in Code 0.46 verläuft wie üblich.

```
1  File
2    Name './example_derived_dts_final_part.ast',
3    [
4      StructDecl
5        Name 'st',
```

[26] Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

```
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14        Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
          ↪  Name('complex_var1')))
15        Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
          ↪  Name('complex_var2')))
16        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
17        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr1')),
          ↪  Ref(Name('var')))
18        Assign(Alloc(Writeable(), PntrDecl(Num('2'), IntType('int')), Name('complex_var3')),
          ↪  Ref(Name('pntr1')))
19        Exp(Subscr(Name('complex_var1'), Num('0')))
20        Exp(Subscr(Name('complex_var2'), Num('0')))
21        Exp(Deref(Name('complex_var3'), Num('0')))
22      ]
23  ]
```

Code 0.46: *Abstrakter Syntaxbaum für den Schlussteil.*

Im **PicoC-ANF Pass** in Code 0.47 wird das am Anfang dieses **Unterkapitels** angesprochene auf den **Stack** speichern des **Attributs** oder **Elements**, dessen **Adresse** in den vorherigen Schritten auf dem **Stack** berechnet wurde mit den Knoten `Exp(Stack(Num('1')))` dargestellt.

```
1 File
2   Name './example_derived_dts_final_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('4')), Stack(Num('1')))
10        // Assign(Name('pntr1'), Ref(Name('var')))
11        Ref(Global(Num('4')))
12        Assign(Global(Num('5')), Stack(Num('1')))
13        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14        Ref(Global(Num('5')))
15        Assign(Global(Num('6')), Stack(Num('1')))
16        // Exp(Subscr(Name('complex_var1'), Num('0')))
17        Ref(Global(Num('0')))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        // Exp(Subscr(Name('complex_var2'), Num('0')))
22        Ref(Global(Num('2')))
23        Exp(Num('0'))
24        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
25        Exp(Stack(Num('1')))
```

```
26        // Exp(Subscr(Name('complex_var3'), Num('0')))
27        Ref(Global(Num('6')))
28        Exp(Num('0'))
29        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
30        Exp(Stack(Num('1')))
31        Return(Empty())
32      ]
33  ]
```

Code 0.47: *PicoC-ANF Pass für den Schlussteil.*

Im **RETI-Blocks Pass** in Code 0.48 werden die **PicoC-Knoten** `Exp(Stack(Num('1')))` durch **semantisch** entsprechende **RETI-Knoten** ersetzt, wenn das versteckte Attribut `datatype` im `Exp(exp,datatpye)`-Knoten kein **Feld** `ArrayDecl(nums, datatype)` enthält. Wenn doch, dann ist bei einem **Feld** die **Adresse**, die in vorherigen Schritten auf dem **Stack** berechnet wurde bereits das gewünschte Ergebnis. Genaueres wurde am Anfang dieses Unterkapitels 0.0.4 zusammen mit der Abbildung 1 bereits erklärt.

```
1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('4')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 4;
15        ADDI SP 1;
16        # // Assign(Name('pntr1'), Ref(Name('var')))
17        # Ref(Global(Num('4')))
18        SUBI SP 1;
19        LOADI IN1 4;
20        ADD IN1 DS;
21        STOREIN SP IN1 1;
22        # Assign(Global(Num('5')), Stack(Num('1')))
23        LOADIN SP ACC 1;
24        STOREIN DS ACC 5;
25        ADDI SP 1;
26        # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
27        # Ref(Global(Num('5')))
28        SUBI SP 1;
29        LOADI IN1 5;
30        ADD IN1 DS;
31        STOREIN SP IN1 1;
32        # Assign(Global(Num('6')), Stack(Num('1')))
33        LOADIN SP ACC 1;
34        STOREIN DS ACC 6;
35        ADDI SP 1;
36        # // Exp(Subscr(Name('complex_var1'), Num('0')))
37        # Ref(Global(Num('0')))
```

```
38          SUBI SP 1;
39          LOADI IN1 0;
40          ADD IN1 DS;
41          STOREIN SP IN1 1;
42          # Exp(Num('0'))
43          SUBI SP 1;
44          LOADI ACC 0;
45          STOREIN SP ACC 1;
46          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47          LOADIN SP IN1 2;
48          LOADIN SP IN2 1;
49          MULTI IN2 2;
50          ADD IN1 IN2;
51          ADDI SP 1;
52          STOREIN SP IN1 1;
53          # // not included Exp(Stack(Num('1')))
54          # // Exp(Subscr(Name('complex_var2'), Num('0')))
55          # Ref(Global(Num('2')))
56          SUBI SP 1;
57          LOADI IN1 2;
58          ADD IN1 DS;
59          STOREIN SP IN1 1;
60          # Exp(Num('0'))
61          SUBI SP 1;
62          LOADI ACC 0;
63          STOREIN SP ACC 1;
64          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65          LOADIN SP IN1 2;
66          LOADIN SP IN2 1;
67          MULTI IN2 2;
68          ADD IN1 IN2;
69          ADDI SP 1;
70          STOREIN SP IN1 1;
71          # Exp(Stack(Num('1')))
72          LOADIN SP IN1 1;
73          LOADIN IN1 ACC 0;
74          STOREIN SP ACC 1;
75          # // Exp(Subscr(Name('complex_var3'), Num('0')))
76          # Ref(Global(Num('6')))
77          SUBI SP 1;
78          LOADI IN1 6;
79          ADD IN1 DS;
80          STOREIN SP IN1 1;
81          # Exp(Num('0'))
82          SUBI SP 1;
83          LOADI ACC 0;
84          STOREIN SP ACC 1;
85          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86          LOADIN SP IN2 2;
87          LOADIN IN2 IN1 0;
88          LOADIN SP IN2 1;
89          MULTI IN2 1;
90          ADD IN1 IN2;
91          ADDI SP 1;
92          STOREIN SP IN1 1;
93          # Exp(Stack(Num('1')))
94          LOADIN SP IN1 1;
```

```
95          LOADIN IN1 ACC 0;
96          STOREIN SP ACC 1;
97          # Return(Empty())
98          LOADIN BAF PC -1;
99        ]
100   ]
```

Code 0.48: *RETI-Blocks Pass für den Schlussteil.*

0.0.5 Umsetzung von Funktionen

Um die **Umsetzung** von **Funktionen** zu verstehen, ist es erstmal wichtig zu verstehen, wie **Funktionen** später im **RETI-Code** aussehen (Unterkapitel 0.0.5.1), wie Funktionen **deklariert** (Definition **??**) und **definiert** (Definition **??**) werden können und hierbei **Sichtbarkeitsbereiche** (Definition **??**) umgesetzt sind (Unterkapitel 0.0.5.2). Aufbauend darauf können dann die notwendigen Schritte zur Umsetzung eines **Funktionsaufrufes** erklärt werden (Unterkapitel 0.0.5.3). Beim Thema **Funktionsaufruf** wird im speziellen darauf eingegangen werden, wie **Rückgabewerte** (Unterkapitel 0.0.5.3.1) umgesetzt sind und die **Übergabe** von **Zusammengesetzten Datentypen**, die mehr als eine Speicherzelle belegen, wie **Verbunden** (Unterkapitel 0.0.5.3.3) und **Feldern** (Unterkapitel 0.0.5.3.2) umgesetzt ist.

0.0.5.1 Mehrere Funktionen

Die Umsetzung **mehrerer Funktionen** wird im Folgenden mithilfe des Beispiels in Code 0.49 erklärt. Dieses Beispiel soll nur zeigen, wie Funktionen in verschiedenen, für die Kompilierung von Funktionen relevanten **Passes** übersetzt 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    int var = 41;
7    if(1) {
8      var = 42;
9    }
10 }
11
12 int fun2() {
13   return 1;
14 }
```

Code 0.49: *PicoC-Code für 3 Funktionen.*

Im **Abstrakten Syntaxbaum** in Code 0.50 werden die 3 **Funktionen** durch entsprechende Knoten dargestellt. Am Beispiel der **Funktion** `void fun2() {return 1;}` wäre der hierzu passende **Knoten** `FunDef(VoidType(), Name('fun2'), [], [Return(Num('1'))])`. Die einzelnen **Attribute** dieses `FunDef(datatype, name, allocs, stmts_blocks)`-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        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 0.50: *Abstrakter Syntaxbaum für 3 Funktionen.*

Im **PicoC-Blocks Pass** in Code 0.51 werden die **Anweisungen** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Hierbei bekommt ein Block `Block(name, stmts_instrs)`, der die Anweisungen 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)`[27] beinhaltet den **Bezeichner** bzw. den `Name(str)`-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** `<number>` angehängt `<name>.<number>`[28].[29]

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** an ihren Bezeichner `<name>.<number>` angehängt haben, 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',
```

[27] Eine **Erklärung** dazu ist in Unterkapitel **??** zu finden.
[28] Der **Grund** dafür kann im Unterkapitel **??** nachgelesen werden.
[29] 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(Writeable(), 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 0.51: *PicoC-Blocks Pass für 3 Funktionen.*

Im **PicoC-ANF Pass** in Code 0.52 werden die `FunDef(datatype, name, allocs, stmts)`-Knoten komplett

aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Knoten nur noch Blöcke befinden.

```
File
  Name './verbose_3_funs.picoc_mon',
  [
    Block
      Name 'main.4',
      [
        Return(Empty())
      ],
    Block
      Name 'fun1.3',
      [
        // Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('41'))
        // Assign(Name('var'), Num('41'))
        Exp(Num('41'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        // If(Num('1'), [])
        // IfElse(Num('1'), [], [])
        Exp(Num('1')),
        IfElse
          Stack
            Num '1',
          [
            GoTo
              Name 'if.2'
          ],
          [
            GoTo
              Name 'if_else_after.1'
          ]
      ],
    Block
      Name 'if.2',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Stackframe(Num('0')), Stack(Num('1')))
        Exp(GoTo(Name('if_else_after.1')))
      ],
    Block
      Name 'if_else_after.1',
      [
        Return(Empty())
      ],
    Block
      Name 'fun2.0',
      [
        // Return(Num('1'))
        Exp(Num('1'))
        Return(Stack(Num('1')))
      ]
  ]
```

Code 0.52: *PicoC-ANF Pass für 3 Funktionen.*

Nach dem **RETI Pass** in Code 0.53 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 die RETI-Befehle nicht mehr direkt Funktionen zuordnen. Die **Kommentare** enthalten die **Bezeichner** `<name>.<number>` 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 **Adressen** gemacht werden. Die Sprünge werden zu den Adressen gemacht, wo die **RETI-Befehle** anfangen, 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(Writeable(), 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 0.53: *RETI-Blocks Pass für 3 Funktionen.*

0.0.5.1.1 Sprung zur Main Funktion

Im vorherigen Beispiel in Code 0.49 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 0.54 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 0.54: *PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

Im **RETI-Blocks Pass** in Code 0.55 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 0.55: *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, ist es, wenn die `main`-Funktion nicht die erste auftauchende Funktion ist, einen `start.<number>`-Block als ersten Block einzufügen. Dieser `start.<number>`-Block enthält einen `GoTo(Name('main.<number>'))`-Knoten, der im **RETI Pass** 0.57 in einen Sprung zur `main`-Funktion übersetzt wird.[30]

In der Implementierung des **PicoC-Compilers** wurde sich für diese Möglichkeit entschieden, da es für Verwender[31] 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 **Einsetzen** des `start.<number>`-Blockes erfolgt im **RETI-Patch Pass** in Code 0.56. Der **RETI-Patch** Pass ist der Pass, der für das **Ausbessern**[32] von Befehlen und Anweisungen 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
```

[30] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.
[31] Also die kommenden **Studentengenerationen**.
[32] In engl. to patch.

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

Im **RETI Pass** in Code 0.57 wird das `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 0.57: *RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist.*

0.0.5.2 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 **Dekleration** einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 0.58 **markierten Zeile** `1` oder zusammen mit der **Funktionsdefinition** (z.B. `void fun1(){}`), wie in den **markierten Zeilen** `3-4`.

In dem Beispiel in Code 0.58 erfolgt ein **Funktionsaufruf** der Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdekleration**, 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 0.58: *PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

Die **Deklaration** einer **Funktion** erfolgt mithilfe der **Symboltabelle**, die in Code 0.59 für das Beispiel in Code 0.58 dargestellt ist. Für z.B. die Funktion `int fun2(int var)` werden die **Attribute** des **Symbols** `Symbols(type_qual, datatype, name, val_addr, pos, size)` wie üblich gesetzt. Dem `datatype`-Attribut wird dabei einfach die komplette **Funktionsdeklaration** `FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(), 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` angehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert, sobald beim **Top-Down**[33]-Iterieren über den **Abstrakten Syntaxbaum** des aktuellen **Passes** ein neuer `FunDef(datatype, name, allocs, stmts_blocks)`-Knoten betreten wird und über dessen Anweisungen im `stmts`-Attribut iteriert wird. Beim **Iterieren** über die Anweisungen eines **Funktionsknotens** wird beim Erstellen neuer **Symboltabelleneinträge** an die **Schlüssel** ein **Suffix** angehängt, der aus dem `name`-Attribut des **Funktionsknotens** `FunDef(name, datatype, params, stmts_blocks)` entnommen wird.

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. Des Weiteren lässt sich

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

aus dem **Symboltabelleneintrag** einer **Variable** direkt ihr **Sichtbarkeitsbereich**, in dem sie definiert wurde ablesen. Der **Suffix** ist ebenfalls im `Name(str)`-Knoten des `name`-Attributes eines **Symboltabelleneintrags** der Symboltabelle angehängt. Dies ist in Code 0.59 **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** sein müssen[34], kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

Das Symbol `'@'` wird aus einem bestimmten Grund als **Trennzeichen** verwendet, welcher bereits in Unterkapitel **??** erläutert wurde.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(IntType('int'), Name('fun2'), [Alloc(Writeable(),
        ↪  IntType('int'), Name('var'))])
        name:                   Name('fun2')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('4'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('fun1'), [])
        name:                   Name('fun1')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('6'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('var@main')
        value or address:       Num('0')
        position:               Pos(Num('7'), Num('6'))
        size:                   Num('1')
      },
```

[34]Sonst gibt es eine **Fehlermeldung**, wie `ReDeclarationOrDefinition`.

```
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 0.59: *Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss.*

0.0.5.3 Funktionsaufruf

Ein **Funktionsaufruf** (z.B. `stack_fun(local_var)`) wird im Folgenden mithilfe des Beispiels in Code 0.60 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 0.0.5.3.1 erklärt. Zudem wurde, um die **Adressberechnung anschaulicher** zu machen als **Datentyp** für den **Parameter** `param` der Funktion `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 0.60: *PicoC-Code für Funktionsaufruf ohne Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 0.61 wird ein **Funktionsaufruf** `stack_fun(1+1)` durch die **Knoten** `Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))` dargestellt.

```
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'))),
          ↪  Name('local_var')))
24        Exp(Call(Name('stack_fun'), [BinOp(Num('1'), Add('+'), Num('1'))]))
25        Return(Empty())
26      ],
27    FunDef
28      VoidType 'void',
29      Name 'stack_fun',
30      [
31        Alloc(Writeable(), IntType('int'), Name('param'))
32      ],
33      [
34        Exp(Alloc(Writeable(), ArrayDecl([Num('2')], StructSpec(Name('st'))),
          ↪  Name('local_var')))
35      ]
36  ]
```

Code 0.61: *Abstrakter Syntaxbaum für Funktionsaufruf ohne Rückgabewert.*

Alle Funktionen **außer** der `main`-Funktion besitzen einen **Stackframe** (Definition 0.2). Bei der `main`-Funktion werden **Lokale Variablen** einfach zu den **Globalen Statischen Daten** geschrieben.

In Tabelle 9 ist für das Beispiel in Code 0.60 das **Datensegment** inklusive **Stackframe** der Funktion `stack_fun` mit allen **allokierten Variablen** dargestellt. Mithilfe der Spalte **Relativadresse** in der Tabelle 9 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 0.62. Bei **Stackframes** fangen die **Relativadressen** erst 2 Speicherzellen relativ zum `BAF`-Register an, da die **Rücksprungadresse** und die **Startadresse des Vorgängerframes** Platz brauchen.

Relativ-adresse	Inhalt	Register
0	⟨*local_var@main*⟩	CS
1		
2		
3		
...	...	
...	...	SP
4	⟨*local_var@stack_fun*⟩	
3		
2		
1		
0	⟨*param_var@stack_fun*⟩	
...	Rücksprungadresse	
...	Startadresse Vorgängerframe	BAF

Tabelle 9: *Datensegment mit Stackframe.*

Definition 0.2: 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 gestappelt** und in die **entgegengesetzte Richtung** wieder abgebaut, wenn sie nicht mehr benötigt werden. Der **Aufbau** eines **Stackframes** ist in Tabelle 10 dargestellt.*[a]

...	← SP
Temporäre Berechnungen	
Lokale Variablen	
Parameter	
Rücksprungadresse	
Startadresse Vorgängerframe	← BAF

Tabelle 10: *Aufbau Stackframe*

*Üblicherweise steht als **erstes**[b] in einem Stackframe die **Startadresse** des **Vorgängerframes**. Diese ist notwendig, damit beim **Rücksprung** aus einer **aufgerufenen Funktion**, zurück zur **aufrufenden 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 **Rücksprungadresse** ist die Adresse **relativ** zum Anfang des **Codesegments**, an welcher die **Ausführung** der aufrufenden Funktion nach einem Funktionsaufruf **fortgesetzt** wird.*

*Die **Startadresse** des **Vorgängerframes** und die **Rücksprungadresse** stehen beide im Stackframe der **aufgerufenen Funktion**, da andere Stackframes bei einem Funktionsaufruf **nicht** mehr so einfach **zugänglich** sind und diese beiden Informationen für den **Rücksprung benötigt** werden. Alles weitere in Tabelle 10 ist selbsterklärend.*[c]

[a] Wenn von **„auf den Stack schreiben“** gesprochen wird, dann wird damit gemeint, dass der Bereich für **Temporäre Berechnungen** (nach Tabelle 10) **vergrößert** wird und ein **Wert** hinein **geschrieben** wird.
[b] Die Tabelle 10 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.
[c]Scholl, „Betriebssysteme“.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('attr@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(),
        ↪  ArrayDecl([Num('2')], IntType('int')), Name('attr'))])
        name:                   Name('st')
        value or address:       [Name('attr@st')]
        position:               Pos(Num('1'), Num('7'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('stack_fun'),
        ↪  [Alloc(Writeable(), IntType('int'), Name('param'))])
        name:                   Name('stack_fun')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('5'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               ArrayDecl([Num('2')], StructSpec(Name('st')))
        name:                   Name('local_var@main')
        value or address:       Num('0')
        position:               Pos(Num('6'), Num('12'))
        size:                   Num('4')
      },
    Symbol
      {
        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 0.62: *Symboltabelle für Funktionsaufruf ohne Rückgabewert.*

Im **PicoC-ANF Pass** in Code 0.63 werden die Knoten `Exp(Call(Name('stack_fun'),[Name('local_var')]))` durch die Knoten `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. Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

Der Knoten `StackMalloc(Num('2'))` ist 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.[35]

```
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'), 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      ]
```

[35] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

```
21  ]
```

Code 0.63: *PicoC-ANF Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.64 werden die **PicoC-Knoten** `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 **semantisch** entsprechenden **RETI-Knoten** ersetzt.

Die Knoten `LOADI ACC GoTo(Name('addr@next_instr'))` und `Exp(GoTo(Name('stack_fun.0')))` sind noch keine **RETI-Knoten** und werden erst später in dem für sie vorgesehenen **RETI-Pass** passend ergänzt bzw. ersetzt.

Der **Bezeichner** des Blocks `stack_fun.0` in `Exp(GoTo(Name('stack_fun.0')))` wird im **Assoziativen Feld** `fun_name_to_block_name`[36] mit dem **Schlüssel** `stack_fun`[37], 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          # 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'), 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;
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] Dieses **Assoziative Feld** wurde in Unterkapitel 0.0.5.1 eingeführt.
[37] Dem **Bezeichner der Funktion**.

```
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 0.64: *RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert.*

Im **RETI Pass** in Code 0.64 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`[38] [39] ersetzt: `LOADI ACC 14`. 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 versteckten `instrs_before`-Attributs des **Zielblocks** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`[40] und des **aktuellen Blocks**, in dem der **RETI-Knoten** `JUMP 5` selbst liegt berechnet.

Die **relative Adresse** `14` des Befehls `LOADI ACC 14` 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 `14` um eine **relative Adresse**, die **relativ** zum `CS`-Register[41] berechnet wird.

Anmerkung

Die Berechnung der **Adresse** adr_{danach} bzw. `<addr@next_instr>` des Befehls nach dem **Sprung** `JUMP <distanz>` für den Befehl `LOADI ACC <addr@next_instr>` erfolgt mithilfe der folgenden Formel 0.0.4:

$$adr_{danach} = \#Bef_{vor\,akt.\,Bl.} + idx + 4 \tag{0.0.4}$$

wobei:

- es sich bei adr_{danach} um eine **relative Adresse** handelt, die **relativ** zum `CS`-Register berechnet wird.
- $\#Bef_{vor\,akt.\,Bl.} \hat{=}$ **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

[38] Der für den **Sprung zur gewünschten Funktion** verantwortlich ist.
[39] Also der Befehl, der bisher durch die Komposition `Exp(GoTo(Name('stack_fun.0')))` dargestellt wurde.
[40] Welcher den **ersten Befehl** der gewünschten Funktion enthält.
[41] Welches im **RETI-Interpreter** von einem **Startprogramm** im **EPROM** immer so gesetzt wird, dass es die **Adresse** enthält, an der das **Codesegment** anfängt.

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>` selbst im **aktuellen Block**.
- $4 \hat{=}$ **Distanz**, die zwischen den in Code 0.65 markierten Befehlen `LOADI ACC <im>` und `JUMP <im>` liegt und noch **eins** mehr, weil man ja zum nächsten Befehl will.

Die Berechnug der **Distanz**[a] $Dist_{Zielbl.}$ bzw. `<distance>` zum **ersten** Befehl eines im vorhergehenden **Pass existenten Blockes**[b] für den Sprungbefehl `JUMP <distance>` erfolgt nach der folgenden Formel 0.0.5:

$$Dist_{Zielbl.} = \begin{cases} \#Bef_{vor\ Zielbl.} - \#Bef_{vor\ akt.\ Bl.} - idx & \#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.} \\ -idx & \#Bef_{vor\ Zielbl.} = \#Bef_{vor\ akt.\ Bl.} \end{cases} \tag{0.0.5}$$

wobei:

- $\#Bef_{vor\ Zielbl.} \hat{=}$ **Anzahl** Befehle vor dem **Zielblock** zu dem gesprungen werden soll. Es handelt sich hierbei um ein versteckes Attribut `instrs_before` eines jeden **Blockes** `Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`.
- $\#Bef_{vor\ akt.\ Bl.}$ und idx haben die **gleiche Bedeutung**, wie in der Formel 0.0.4.
- $idx \hat{=}$ **relativer Index** des Befehls `JUMP <distance>` selbst im **aktuellen Block**.

In Abbildung 2 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 0.0.5 abgedeckt sind[c].

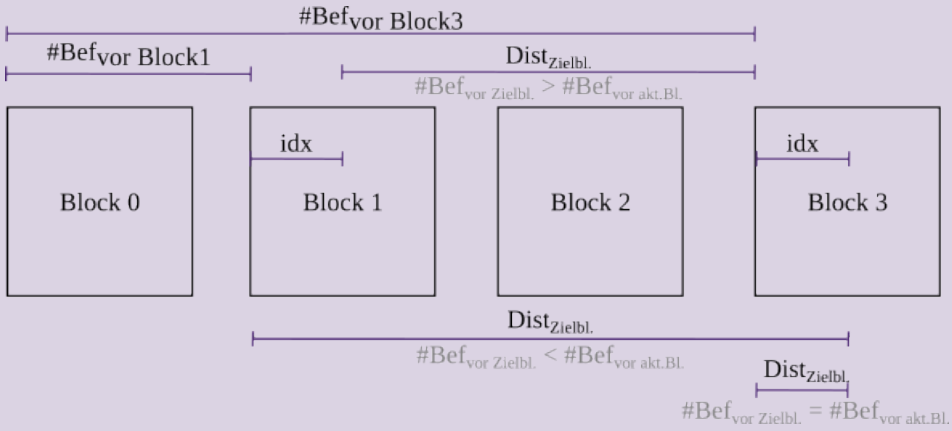


Abbildung 2: *Veranschaulichung der Dinstanzberechnung*

[a] Diese **Distanz** kann auch **negativ** werden.
[b] Im **RETI-Pass** gibt es **keine** Blöcke mehr.
[c] Die **Konstellationen** $\#Bef_{vor\ Zielbl.} > \#Bef_{vor\ akt.\ Bl.}$, $\#Bef_{vor\ Zielbl.} < \#Bef_{vor\ akt.\ Bl.}$ sind im Fall $\#Bef_{vor\ Zielbl.} != \#Bef_{vor\ akt.\ Bl.}$ 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'), 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 0.65: *RETI-Pass für Funktionsaufruf ohne Rückgabewert.*

0.0.5.3.1 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 0.66 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 0.66: *PicoC-Code für Funktionsaufruf mit Rückgabewert.*

Im **Abstrakten Syntaxbaum** in Code 0.67 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(Writeable(),IntType('int'),Name('var')),Call(Name('fun_with_return_value'),[]))`.

```
File
  Name './example_fun_call_with_return_value.ast',
  [
    FunDef
      IntType 'int',
      Name 'fun_with_return_value',
      [],
      [
        Return(BinOp(Num('21'), Mul('*'), Num('2')))
      ],
    FunDef
      VoidType 'void',
      Name 'fun_no_return_value',
      [],
      [
        Return(Empty())
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')),
        ↪  Call(Name('fun_with_return_value'), []))
        Exp(Call(Name('fun_no_return_value'), []))
      ]
  ]
```

Code 0.67: *Abstrakter Syntaxbaum für Funktionsaufruf mit Rückgabewert.*

Im **PicoC-ANF Pass** in Code 0.68 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 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 **Rücksprungadresse** in das `PC`-Register geladen[42], um wieder zur **aufrufenden Funktion** zurückzuspringen.

[42] Die **Rücksprungadresse** wurde zuvor durch den `NewStackframe()`-Knoten (siehe Unterkapitel 0.0.5.3 für Zusammenhang) eine Speicherzelle nach der Speicherzelle auf die das `BAF`-Register zeigt im **Stackframe** gespeichert.

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[43], 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 **aufrufende Funktion**, deren **Stackframe** nun wieder der aktuelle ist 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 notwendidg, 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 wissen, 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 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.

Der Umgang mit einer **Funktion ohne Rückgabewert** wurde am Anfang dieses Unterkapitels 0.0.5.3 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())` hinzufügt 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        [
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       ],
```

[43] Als in Unterkapitel 0.0.5.3 **bisher** erklärt wurde.

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

Code 0.68: *PicoC-ANF Pass für Funktionsaufruf mit Rückgabewert.*

Im **RETI-Blocks Pass** in Code 0.69 werden die **PicoC-Knoten** `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 **semantisch** entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_with_return_value.reti_blocks',
  [
    Block
      Name 'fun_with_return_value.2',
      [
        # // Return(BinOp(Num('21'), Mul('*'), Num('2')))
        # Exp(Num('21'))
        SUBI SP 1;
        LOADI ACC 21;
        STOREIN SP ACC 1;
        # Exp(Num('2'))
        SUBI SP 1;
        LOADI ACC 2;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
        LOADIN SP ACC 2;
        LOADIN SP IN2 1;
        MULT ACC IN2;
        STOREIN SP ACC 2;
        ADDI SP 1;
        # Return(Stack(Num('1')))
        LOADIN SP ACC 1;
        ADDI SP 1;
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_no_return_value.1',
      [
        # 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 0.69: *RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert.*

0.0.5.3.2 Umsetzung der Übergabe eines Feldes

Die Eigenheit, dass bei der **Übergabe** eines **Felds** 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 0.70 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 0.70: *PicoC-Code für die Übergabe eines Feldes.*

Später im **PicoC-ANF Pass** muss im Fall dessen, dass der Datentyp, der an eine Funktion übergeben wird ein **Feld** `ArrayDecl(nums, datatype)` ist, auf spezielle Weise vorgegangen werden. Der **oberste Knoten** des Teilbaums, der den **Feld-Datentyp** `ArrayDecl(nums, datatype)` darstellt, muss zu einem **Zeiger** `PntrDecl(num, datatype)` umgewandelt werden und der Rest des Teilbaumes, der am `datatype`-Attribut hängt, muss an das `datatype`-Attribut des **Zeigers** `PntrDecl(num, datatype)` gehängt werden. Bei einem **Mehrdimensionalen Feld** fällt eine **Dimension** an den Zeiger weg und der Rest des Felds wird an das `datatype`-Attribut des **Zeigers** `PntrDecl(num, datatype)` gehängt.

Diese **Umwandlung** eines **Felds** zu einem **Zeiger** kann in der **Symboltabelle** in Code 0.71 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 `Num('1')`, da ein **Zeiger** nur eine **Speicherzelle** einnimmt.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                FunDecl(VoidType('void'), Name('fun_array_from_stackframe'),
           ↪  [Alloc(Writeable(), 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:          Writeable()
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(Writeable(), 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:         Writeable()
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:         Writeable()
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:         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 0.71: *Symboltabelle für die Übergabe eines Feldes.*

Im **PicoC-ANF Pass** in Code 0.72 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-Pass** durch unterschiedliche **RETI-Befehle** ersetzt. Hierbei stellen die Zahlen `'0'` bzw. `'6'` in den Knoten `Global(num)` bzw. `Stackframe(num)`, die aus der **Symboltabelle** entnommen sind die **relative Adressen** relativ zum `DS`-Register bzw. `SP`-Register dar. Die Zahl `'6'` ergibt sich dadurch, dass das Feld `local_var` die **Dimensionen** 2×3 hat und ein Feld von **Integern** ist, also $size(type(\texttt{local\_var})) = \left(\prod_{j=1}^{n} \texttt{dim}_\texttt{j}\right) \cdot size(int) = 2 \cdot 3 \cdot 1 = 6$ *Speicherzellen*.

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

Code 0.72: *PicoC-ANF Pass für die Übergabe eines Feldes.*

Im **RETI-Blocks Pass** in Code 0.73 werden **PicoC-Knoten** `Ref(Global(Num('0')))` und `Ref(Stackframe(Num('6')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_fun_call_by_sharing_array.reti_blocks',
  [
    Block
      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 0.73: *RETI-Block Pass für die Übergabe eines Feldes.*

0.0.5.3.3 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 0.74 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 0.74: *PicoC-Code für die Übergabe eines Verbundes.*

Im **PicoC-ANF Pass** in Code 0.75 werden zur **Übergabe der beiden Verbunde** `local_var@main` und `param@fun_array_from_global_data`, die beiden Verbunde 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[44]. 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.

Das wird durch eine Variable `argmode_on` implementiert, die auf `true` gesetzt wird, solange der **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 immer 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** `Ref(Global(num))` bzw. `Ref(Stackframe(num))` für die Ersetzung verwendet.[45]

[44] Wie es in Unterkapitel 0.0.5.3.2 erklärt wurde

[45] Die **Bedeutung** aller hier erwähnten **Knoten** und **Kompositionen** von Knoten wird in den Tabellen der Kapitel **??**, **??** und **??** erläutert.

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.

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

Code 0.75: *PicoC-ANF Pass für die Übergabe eines Verbundes.*

Im **RETI-Blocks Pass** in Code 0.76 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.

```
File
  Name './example_fun_call_by_value_struct.reti_blocks',
  [
    Block
      Name 'fun_struct_from_stackframe.2',
      [
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun_struct_from_global_data.1',
      [
```

```
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 0.76: *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 (besucht am 09. 07. 2022).