Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 Umsetzung von Pointern

0.0.1.1 Referenzierung

Die **Referenzierung** (z.B. `&var`) 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 Pointer Referenzierung

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstract Syntax Tree** in Code 0.2 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Pointer `*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: Abstract Syntax Tree für Pointer Referenzierung

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

> Die **Größe** eines Pointers (z.B. eines Pointers auf ein Array von `int`: `pntr = int *pntr[3]`), die ihm `size`-Feld der **Symboltabelle** eingetragen ist, ist dabei immer: `size(pntr) = 1`.

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                FunDecl(VoidType('void'), Name('main'), [])
7          name:                    Name('main')
8          value or address:        Empty()
9          position:                Pos(Num('1'), Num('5'))
10         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 Pointer Referenzierung

Im **PicoC-Mon 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')))` ist, sind noch weitere Anweisungen zwischen den Zeilen `11` und `12` nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp)` nach dem Schema in Subkapitel 0.0.4.3 kümmern.

```
File
  Name './example_pntr_ref.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Assign(Name('var'), Num('42'))
        Exp(Num('42'))
        Assign(Global(Num('0')), Stack(Num('1')))
        // Assign(Name('pntr'), Ref(Name('var')))
        Ref(Global(Num('0')))
        Assign(Global(Num('1')), Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 0.4: PicoC-Mon Pass für Pointer Referenzierung

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 entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_pntr_ref.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('pntr'), Ref(Name('var')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

Code 0.5: RETI-Blocks Pass für Pointer Referenzierung

0.0.1.2 Dereferenzierung durch Zugriff auf Arrayindex ersetzen

Die **Dereferenzierung** (z.B. `*var`) wird im Folgenden anhand des Beispiels in Code 0.6 erklärt.

```
void main() {
  int var = 42;
  int *pntr = &var;
  *pntr;
}
```

Code 0.6: PicoC-Code für Pointer Dereferenzierung

Der Container-Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstract Syntax Tree** in Code 0.7 eine **Dereferenzierung** `*var`. Es gibt herbei **zwei** Fälle. Bei der Anwendung von **Pointer Arithmetik**, wie z.B. `*(var + 2 - 1)` übersetzt sich diese zu `Deref(Name('var'), BinOp(Num('2'), Sub(), BinOp(Num('1'))))`. Bei einer normalen **Dereferenzierung**, wie z.B. `*var`, übersetzt sich diese zu `Deref(Name('var'), Num('0'))`.

```
File
  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: Abstract Syntax Tree für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 0.8 wird ein Trick angewandet, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Der Trick besteht darin, dass der **Dereferenzoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Arrayindex** (z.B. `var[1]`) verhält[1]. Damit sparrt man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Arrayindex** (z.B. `var[1]`) übernehmen lassen.

```
1 File
2   Name './example_pntr_deref.picoc_shrink',
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(Subscr(Name('pntr'), Num('0')))
12      ]
13  ]
```

Code 0.8: PicoC-Shrink Pass für Pointer Dereferenzierung

0.0.2 Umsetzung von Arrays

0.0.2.1 Initialisierung von Arrays

Die **Initialisierung** eines **Arrays** (z.B. `int ar[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 0.9 erklärt.

[1] In der Sprache L_C gibt es einen Unterschied bei der Initialisierung bei z.B. `int *var = "string"` und z.B. `int var[1] = "string"`, der allerdings nichts mit den beiden Operatoren zu tuen hat, sondern mit der **Initialisierung**, bei der die Sprache L_C verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Arrayindex**, vor den Bezeichner schreibt (z.B. `var[1]`), obwohl es ein **Derived Datatype** ist.

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

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

Code 0.9: PicoC-Code für Array Initialisierung

Die **Initialisierung** eines **Arrays** `int ar[2][1] = {{3+1}, {4}}` wird im **Abstract Syntax Tree** in Code 0.10 mithilfe der Komposition `Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')), Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])]))` dargestellt.

```
File
  Name './example_array_init.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
        ↪  Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
      ],
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
        Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
        ↪  Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')])]))
      ]
  ]
```

Code 0.10: Abstract Syntax Tree für Array Initialisierung

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

Bei Variablen auf dem **Stackframe** wird ein Array **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Arrays genommen. Dies macht den **Zugriff auf einen Arrayindex** in Subkapitel 0.0.2.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf einen Arrayindex** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die entgegengesetzte Richtung wächst, verglichen mit den **Globalen**

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

Statischen Daten[3].

Das **Größe** des Arrays `datatype ar`$[\mathtt{dim}_1]\dots[\mathtt{dim}_k]$, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Arrays multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Arrayelemente**: $\mathtt{size}(\mathtt{datatype}(\mathtt{ar})) = \left(\prod_{j=1}^{n} \mathtt{dim}_j\right) \cdot \mathtt{size}(\mathtt{datatype})$[a].

[a] Die **Funktion** `type` ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die **Funktion** `size` nur bei einem **Datentyp** als **Funktionsargument** die **Größe dieses Datentyps** als **Zielwert** liefert

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

Code 0.11: Symboltabelle für Array Initialisierung

[3] Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

Im **PiocC-Mon Pass** in Code 0.12 werden zuerst die **Logischen Ausdrücke** in den Blättern des Teilbaums, der beim **Array-Initializers Container-Knoten** `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` anfängt nach dem **Depth-First-Search** Schema, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben[4].

Im finalen Schritt muss zwischen **Globalen Statischen Daten** bei der `main`-Funktion und **Stackframe** bei der Funktion `fun` unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))`, die in Tabelle **??** genauer beschrieben ist, versetzt in der selben Reihenfolge zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.

Der **Trick** ist hier, dass egal wieviele Dimensionen und was für einen Datentyp das **Array** hat, man letztendlich immer das gesamte Array erwischt, wenn man einfach die **Größe des Arrays** viele **Speicherzellen** mit z.B. der **Komposition** `Assign(Global(Num('0')), Stack(Num('2')))` verschiebt.

In die Knoten `Global('0')` und `Stackframe('3')` wurde hierbei die **Startadresse** des jeweiligen Arrays geschrieben, sodass man nach dem **PicoC-Mon Pass** nie mehr Variablen in der **Symboltabelle** nachsehen muss und gleich weiß, ob sie in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

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

Code 0.12: PicoC-Mon Pass für Array Initialisierung

Im **RETI-Blocks Pass** in Code 0.13 werden die **Kompositionen** `Exp(exp)` und `Assign(Global(Num('0'))`,

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

`Stack(Num('2'))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
        ↪  Array([Num('4')])]))
        # Exp(Num('3'))
        SUBI SP 1;
        LOADI ACC 3;
        STOREIN SP ACC 1;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
        LOADIN SP ACC 2;
        LOADIN SP IN2 1;
        ADD ACC IN2;
        STOREIN SP ACC 2;
        ADDI SP 1;
        # Exp(Num('4'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('2')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # Return(Empty())
        LOADIN BAF PC -1;
      ],
    Block
      Name 'fun.0',
      [
        # // 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'))
        SUBI SP 1;
        LOADI ACC 4;
        STOREIN SP ACC 1;
        # Exp(Num('5'))
        SUBI SP 1;
        LOADI ACC 5;
        STOREIN SP ACC 1;
        # Exp(Num('6'))
```

```
52          SUBI SP 1;
53          LOADI ACC 6;
54          STOREIN SP ACC 1;
55          # Assign(Stackframe(Num('3')), Stack(Num('4')))
56          LOADIN SP ACC 1;
57          STOREIN BAF ACC -2;
58          LOADIN SP ACC 2;
59          STOREIN BAF ACC -3;
60          LOADIN SP ACC 3;
61          STOREIN BAF ACC -4;
62          LOADIN SP ACC 4;
63          STOREIN BAF ACC -5;
64          ADDI SP 4;
65          # Return(Empty())
66          LOADIN BAF PC -1;
67        ]
68    ]
```

Code 0.13: RETI-Blocks Pass für Array Initialisierung

0.0.2.2 Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 0.14 erklärt.

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

Code 0.14: PicoC-Code für Zugriff auf einen Arrayindex

Der **Zugriff auf einen Arrayindex** `ar[0]` wird im **Abstract Syntax Tree** in Code 0.15 mithilfe des **Container-Knotens** `Subscr(Name('ar'), Num('0'))` dargestellt.

```
1 File
2   Name './example_array_access.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
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 'fun',
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: Abstract Syntax Tree für Zugriff auf einen Arrayindex

Im **PicoC-Mon Pass** in Code 0.16 wird vom **Container-Knoten** `Subscr(Name('ar'), Num('0'))` zuerst im **Anfangsteil** 0.0.4.2 die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktionm `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil** 0.0.4.3 wird die **Adresse** ab der das **Arrayelement** des Arrays auf das Zugegriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Arrays, in dem dieses **Arrayelement** liegt auf den **Stack** gelegt. Da ein **Index** auf den Zugegriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `ar[1 + var]` bestimmt sein kann, indem auch **Variablen** vorkommen können, 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, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle **??** dokumentiert.

Zur **Adressberechnung** ist es notwendig auf die **Dimensionen** (z.B. `[Num('3')]`) des Arrays, auf dessen **Arrayelement** zugegriffen wird, zugreifen zu können. Daher ist der **Arraydatentyp** (z.B. `ArrayDecl([Num('3')], IntType('int'))`) dem **Container-Knoten** `Ref(exp, datatype)` als versteckes Attribut `datatype` angehängt. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon Pass** dem **Container-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 und es muss auch möglich sein, z.B. einen **Attributzugriff** `var.attr` und eine **Zugriff auf einen Arryindex** `var[1]` miteinander zu kombinieren, was in Subkapitel 0.0.4.3 allgemein erklärt ist.

Im letzten Schritt, dem **Schlussteil** 0.0.4.4 wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde, nun auf den **Stack** geschrieben, wobei dieser die **Adresse** auf dem Stack ersetzt, die es zum Finden des **Index** brauchte. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. Je nachdem, welchen **Datentyp** die Variable `ar` hat und auf welchen **Unterdatentyp** folglich im **Kontext** zuletzt zugegriffen wird, abhängig davon wird der **Schlussteil** `Exp(Stack(Num('1')))` auf eine andere Weise verarbeitet (siehe Subkapitel 0.0.4.4). Der **Unterdatentyp** ist dabei ein versteckes Attribut des `Exp(Stack(Num('1')))`-Knoten.

Der einzige **Unterschied**, je nachdem, ob der **Zugriff auf einen Arrayindex** (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, bei dem unterschiedliche **RETI-Instructions** für eine Variable, die in den **Globalen Statischen Daten** liegt und eine Variable, die auf dem **Stackframe** liegt erzeugt werden müssen.

Die Berechnung der **Adresse**, ab der ein **Arrayelement** eines Arrays $\texttt{datatype ar}[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist, kann mittels der Formel 0.0.1:

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

aus der Betriebssysteme Vorlesung[a] berechnet werden[b].

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden `ref(ar)` in der Formel.

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

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

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** $\texttt{M}[\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n])]$ der Speicherzelle an der finalen **Adresse** $\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n])$.

[a] Scholl, „Betriebssysteme“.
[b] `ref(exp)` steht dabei für die Berechnung der **Adresse** von `exp`, wobei `exp` z.B. `ar[3][2]` sein könnte.

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

```
26          Ref(Stackframe(Num('2')))
27          Exp(Num('1'))
28          Exp(Num('1'))
29          Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
30          Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
31          Exp(Stack(Num('1')))
32          Return(Empty())
33        ]
34    ]
```

Code 0.16: PicoC-Mon Pass für Zugriff auf einen Arrayindex

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

```
1  File
2    Name './example_array_access.reti_blocks',
3    [
4      Block
5        Name 'main.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(Global(Num('0')), Stack(Num('1')))
13         LOADIN SP ACC 1;
14         STOREIN DS ACC 0;
15         ADDI SP 1;
16         # // Exp(Subscr(Name('ar'), Num('0')))
17         # Ref(Global(Num('0')))
18         SUBI SP 1;
19         LOADI IN1 0;
20         ADD IN1 DS;
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 'fun.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(Stackframe(Num('2')), Stack(Num('3')))
57          LOADIN SP ACC 1;
58          STOREIN BAF ACC -2;
59          LOADIN SP ACC 2;
60          STOREIN BAF ACC -3;
61          LOADIN SP ACC 3;
62          STOREIN BAF ACC -4;
63          ADDI SP 3;
64          # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65          # Ref(Stackframe(Num('2')))
66          SUBI SP 1;
67          MOVE BAF IN1;
68          SUBI IN1 4;
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 Arrayindex

0.0.2.3 Zuweisung an Arrayindex

Die **Zuweisung** eines Wertes an einen **Arrayindex** (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[2] = 42;
4 }
```

Code 0.18: PicoC-Code für Zuweisung an Arrayindex

Im **Abstract Syntax Tree** in Code 0.19 wird eine **Zuweisung** an einen **Arrayindex** `ar[2] = 42;` durch die Komposition `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

```
1  File
2    Name './example_array_assignment.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
10         Assign(Subscr(Name('ar'), Num('2')), Num('42'))
11       ]
12   ]
```

Code 0.19: Abstract Syntax Tree für Zuweisung an Arrayindex

Im **PicoC-Mon Pass** in Code 0.20 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=`, bzw. des **Container-Knotens** der diesen darstellt ausgewertet: `Exp(Num('42'))`.

Danach ist das Vorgehen, bzw. sind die Kompostionen, die dieses darauffolgende 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 Subkapitel 0.0.2.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird berechet, wie in Subkapitel 0.0.2.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`[5] die Zuweisung `=` des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Arrayindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

[5] Ist in Tabelle ?? genauer beschrieben ist

```
File
  Name './example_array_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
        Exp(Num('42'))
        Ref(Global(Num('0')))
        Exp(Num('2'))
        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        Assign(Stack(Num('1')), Stack(Num('2')))
        Return(Empty())
      ]
  ]
```

Code 0.20: PicoC-Mon Pass für Zuweisung an Arrayindex

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

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Exp(Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar')))
        # // Assign(Subscr(Name('ar'), Num('2')), 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('2'))
        SUBI SP 1;
        LOADI ACC 2;
        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')))
```

```
30          LOADIN SP IN1 1;
31          LOADIN SP ACC 2;
32          ADDI SP 2;
33          STOREIN IN1 ACC 0;
34          # Return(Empty())
35          LOADIN BAF PC -1;
36      ]
37  ]
```

Code 0.21: RETI-Blocks Pass für Zuweisung an Arrayindex

0.0.3 Umsetzung von Structs

0.0.3.1 Deklaration und Definition von Structtypen

Die **Deklaration** eines neuen **Structtyps** (z.B. `struct st {int len; int ar[2];}`) und die **Definition** einer Variable mit diesem **Structtyp** (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 Structtyps

Bevor irgendwas definiert werden kann, muss erstmal ein **Structtyp** deklariert werden. Im **Abstract Syntax Tree** in Code 0.24 wird die **Deklaration eines Structtyps** `struct st {int len; int ar[2];}` durch die Komposition `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 **Structtyp** `struct st st_var;` wird durch die Komposition `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: Abstract Syntax Tree für die Deklaration eines Structtyps

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 0.24 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Felder dieses Eintrags `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Feld die Attribute des **Structtyps** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Structtyp** `st` die **Attribute** des Structtyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Structtyps**

zwei Attribute nicht gleich benannt werden können, aber dafür zwei **unterschiedliche Structtypen** ihre Attribute gleich benennen können.

Jedes der **Attribute** `[Name('len@st'), Name('ar@st')]` erhält auch einen eigenen Eintrag in der **Symboltabelle**, wobei die Felder `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt werden. Die Felder `type_qualifier`, `datatype` und `name` werden z.B. bei `Name('ar@st')` mithilfe der Attribute von `Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` belegt.

Für die **Definition** einer Variable `st_var@main` mit diesem **Structtyp** `st` wird ein Eintrag in der **Symboltabelle** angelegt. Das `datatyp`-Feld enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structyp**[6] und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Die **Größe** einer Variable `st_var`, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st` $\{$`datatype`$_1$ `attr`$_1$; ... `datatype`$_n$ `attr`$_n$;$\}$;[a] definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** `datatype`$_1$ ... `datatpye`$_n$ der **Attribute** `attr`$_1$, ... `attr`$_n$ des **Structtyps**: $\texttt{size(st)} = \sum_{i=1}^{n} \texttt{size(datatype}_i\texttt{)}$.

[a] Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache L_{PicoC} nicht die Fragwürdige Designentscheidung, auch die eckigen Klammern `[]` für die Definition eines Arrays **vor** die Variable zu schreiben von L_C übernommen. Es wird so getann, als würde der komplette **Datentyp** immer **hinter** der Variable stehen: `datatype var`.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               IntType('int')
        name:                   Name('len@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('15'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               ArrayDecl([Num('2')], IntType('int'))
        name:                   Name('ar@st')
        value or address:       Empty()
        position:               Pos(Num('1'), Num('24'))
        size:                   Num('2')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'),
        ↪  Name('len'))Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')),
        ↪  Name('ar'))])
        name:                   Name('st')
        value or address:       [Name('len@st'), Name('ar@st')]
        position:               Pos(Num('1'), Num('7'))
        size:                   Num('3')
```

[6] Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Structtyp** einnimmt.

```
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('5'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               StructSpec(Name('st'))
        name:                   Name('st_var@main')
        value or address:       Num('0')
        position:               Pos(Num('4'), Num('12'))
        size:                   Num('3')
      }
  ]
```

Code 0.24: Symboltabelle für die Deklaration eines Structtyps

0.0.3.2 Initialisierung von Structs

Die **Initialisierung eines Structs** wird im Folgenden mithilfe des Beispiels in Code 0.25 erklärt.

```
struct st1 {int *attr[2];};

struct st2 {int attr1; struct st1 attr2;};

void main() {
  int var = 42;
  struct st2 st = {.attr1=var, .attr2={.attr={{&var, &var}}}};
}
```

Code 0.25: PicoC-Code für Initialisierung von Structs

Im **Abstract Syntax Tree** in Code 0.26 wird die **Initialisierung eines Structs** `struct st1 st = {.attr1=var, .attr2={.attr={{&var, &var}}}}` mithilfe der **Komposition** `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([Array([Ref(Name('var')),
        ↪  Ref(Name('var'))])])])])])
      ]
  ]
```

Code 0.26: Abstract Syntax Tree für Initialisierung von Structs

Im **PicoC-Mon Pass** in Code 0.27 wird die **Komposition** `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Arrays** in Subkapitel 0.0.2.1 daher wird um keine Wiederholung zu betreiben auf Subkapitel 0.0.2.1 verwiesen. Um das ganze interressanter zu gestalten wurde das Beispiel in Code 0.25 so gewählt, dass sich daran eine komplexere, mehrstufige Initialisierung mit **verschiedenen** Datentypen erklären lässt.

Der **Struct-Initializer** Teilbaum `Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])])])])`, der beim **Struct-Initializer Container-Knoten** anfängt, wird auf dieselbe Weise nach dem **Depth-First-Search** Prinzip von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Arrays** in Subkapitel 0.0.2.1 bereits erklärt wurde.

Beim **Iterieren** über den **Teilbaum**, muss beim **Struct-Initializer** nur beachtet werden, dass bei den `Assign(lhs, exp)`-Knoten, über welche die **Attributzuweisung** dargestellt wird (z.B. `Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')), Ref(Name('var'))])])]))`) der Teilbaum beim rechten `exp` Attribut weitergeht.

Im Allgemeinen gibt es beim **Initialisieren** eines **Arrays** oder **Structs** im Teilbaum auf der **rechten Seite**, der beim jeweiligen obersten **Initializer** anfängt immer nur 3 Fällte, man hat es auf der **rechten** Seite entweder mit einem **Struct-Initialiser**, einem **Array-Initialiser** oder einem **Logischen Ausdruck** zu tuen. Bei **Array-** und **Struct-Initialisier** wird einfach über diese nach dem **Depth-First-Search** Schema von **links-nach-rechts** iteriert und die Ergebnisse der **Logischen Ausdrücken** in den **Blättern** auf den **Stack** gespeichert. Der Fall, dass ein **Logischer Ausdruck** vorliegt erübrigt sich damit.

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
```

```
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'),
          ↪  Array([Array([Ref(Name('var')), Ref(Name('var'))])])])])]))
11        Exp(Global(Num('0')))
12        Ref(Global(Num('0')))
13        Ref(Global(Num('0')))
14        Assign(Global(Num('1')), Stack(Num('3')))
15        Return(Empty())
16      ]
17    ]
```

Code 0.27: PicoC-Mon Pass für Initialisierung von Structs

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

```
1 File
2   Name './example_struct_init.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('st'), Struct([Assign(Name('attr1'), Name('var')),
          ↪  Assign(Name('attr2'), Struct([Assign(Name('attr'),
          ↪  Array([Array([Ref(Name('var')), Ref(Name('var'))])])])])]))
17        # Exp(Global(Num('0')))
18        SUBI SP 1;
19        LOADIN DS ACC 0;
20        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 Structs

0.0.3.3 Zugriff auf Structattribut

Der **Zugriff auf ein Structattribut** (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 Structattribut

Im **Abstract Syntax Tree** in Code 0.30 wird der **Zugriff auf ein Structattribut** `st.y` mithilfe der **Komposition** `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: Abstract Syntax Tree für Zugriff auf Structattribut

Im **PicoC-Mon Pass** in Code 0.31 wird die Komposition `Exp(Attr(Name('st'), Name('y')))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Exp(Subscr(Name('ar'), Num('0')))` in Subkapitel 0.0.2.2 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Subkapitel 0.0.2.2 verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Subkapitel 0.0.2.2 durch Kompositionen ersetzt, die sich in **Anfangsteil** 0.0.4.2, **Mittelteil** 0.0.4.3 und **Schlussteil** 0.0.4.4 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 Subkapitel 0.0.2.2.

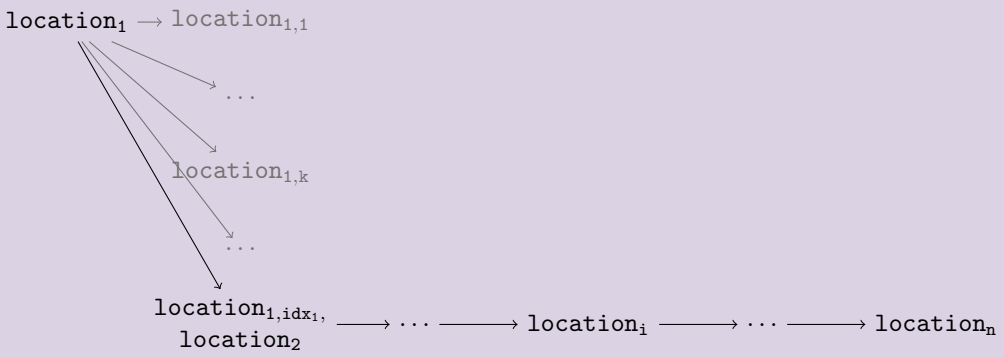
Nur für den **Mittelteil** wird eine andere Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` gebraucht. Diese Komposition `Ref(Attr(Stack(Num('1')), Name('y')))` erfüllt die Aufgabe die **Adresse**, ab der das **Attribut** auf das zugegriffen wird anfängt zu berechnen. Dabei wurde die **Anfangsadresse** des **Structs** indem dieses Attribut liegt bereits vorher auf den **Stack** gelegt.

Im Gegensatz zur Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` beim **Zugriff auf einen Arrayindex** in Subkapitel 0.0.2.2, muss hier vorher nichts anderes als die **Anfangsadresse** des **Structs** auf dem **Stack** liegen. Das **Structattribut** auf welches zugegriffen wird steht bereits in der Komposition `Ref(Attr(Stack(Num('1')), Name('y')))`, nämlich `Name('y')`. Den **Structtyp**, dem dieses Attribut gehört, kann man aus dem versteckten Attribut `datatype` herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im **PiocC-Mon Pass** dem **Container-Knoten** `Ref(exp, datatype)` angehängt.

Sei $\texttt{location}_\texttt{i}$ ein Knotes eines **entarteten Baumes** (siehe Definition 0.2 und Abbildung 0.0.2), dessen Wurzel $\texttt{location}_1$ ist. Dabei steht $\texttt{i}$ für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\texttt{ref}(\texttt{location}_\texttt{i})$ von Speicherbereichen $\texttt{ref}(\texttt{location}_\texttt{i}) \ \ldots \ \texttt{ref}(\texttt{location}_\texttt{i}) + \texttt{size}(\texttt{location}_\texttt{i})$ im Hauptspeicher zuordnen, wobei gilt, dass $\texttt{ref}(\texttt{location}_\texttt{i}) \leq \texttt{ref}(\texttt{location}_{\texttt{i}+1}) < \texttt{ref}(\texttt{location}_\texttt{i}) + \texttt{size}(\texttt{location}_\texttt{i})$.[ab]

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

Sei $\texttt{location}_{\texttt{i,idx}_\texttt{i}}$ ein beliebiges **Element / Attribut** des Datentyps $\texttt{location}_\texttt{i}$, sodass gilt: $\texttt{location}_{\texttt{i,idx}_\texttt{i}} = \texttt{location}_{\texttt{i}+1}$.



(0.0.2)

Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen $(\texttt{location}_{\texttt{1,idx}_1}, \ \ldots, \ \texttt{location}_{\texttt{n,idx}_\texttt{n}})$, die das Resultat einer Aneinandereihung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen

$\texttt{location}_\texttt{i}$ ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 0.0.3:

$$\texttt{ref}(\texttt{location}_{1,\texttt{idx}_1},\ \ldots,\ \texttt{location}_{\texttt{n},\texttt{idx}_\texttt{n}}) = \texttt{ref}(\texttt{location}_1) + \sum_{i=1}^{n-1} \sum_{k=1}^{idx_i - 1} \text{size}(\texttt{location}_{\texttt{i,k}}) \quad (0.0.3)$$

berechnet werden.[c]

Dabei darf nur der letzte Knoten $\texttt{location}_\texttt{n}$ den Datentyp **Pointer** haben. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, der nicht der **letzte Datentyp** $\texttt{location}_\texttt{n}$ in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp $\texttt{location}_1$ bis direkt zum Dantentyp Pointer geht $\texttt{location}_\texttt{pntr}$ und die **zweite Adressberechnung** einen Dantentyp nach dem Datentyp Pointer anfängt $\texttt{datatpye}_{\texttt{pntr}+1}$ und bis zum letzten Datenyp $\texttt{location}_\texttt{n}$ geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\texttt{ref}(\texttt{location}_1)$ des Summanden aus der Formel 0.0.3 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $\texttt{M}\left[\texttt{reflocation}_1 \ldots \texttt{location}_\texttt{pntr}\right]$ gesetzt werden.

Die Formel 0.0.3 stellt dabei eine **Verallgemeinerung** der Formel 0.0.1 dar, die für alle möglichen Aneinandereihungen von Zugriffen auf **Pointerelemente**, **Arrayelementen** und **Structattribute** 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 den gleichen Datentyp haben[d].

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden $\texttt{ref}(\texttt{location}_1)$ in der Formel.

Die Komposition `Exp(Attr(Stack(Num('1')), name))` repräsentiert dabei einen Summanden $\sum_{k=1}^{idx_i - 1} \text{size}(\texttt{location}_{\texttt{i,k}})$ in der Formel.

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

[a] Es ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.
[b] `ref(location)` steht dabei für das Schreiben der **Adresse** von `location` auf den Stack.
[c] Die **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Pointerelmente**, **Arrayelmente** oder **Structattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** $\texttt{location}_\texttt{i}$, die vor dem **Element** / **Attribut** $\texttt{location}_{\texttt{i,idx}_\texttt{i}}$ liegen.
[d] Structattribute haben **unterschiedliche** Größen.

Definition 0.1: Location

*Kollektiver Begriff für **Variablen**, **Attribute** bzw. **Elemente** von Variablen bestimmter Datentypen, **Speicherbereiche auf dem Stack**, die **temporäre Zwischenergebnisse** speichern und **Register**.*

*Im Grunde genommen alles, was mit einem **Programm zu tuen** hat und irgendwo **gespeichert** ist.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 0.2: Entarteter Baum

*Baum bei dem jeder Knoten **maximal** eine ausgehende Kante hat, also maximal **Außengrad** 1.*

Oder alternativ: *Baum beim dem jeder Knoten des Baumes **maximal** eine eingehende Kante hat, also maximal **Innengrad** 1.*

*Der Baum entspricht also einer **verketteten Liste**.*[a]

[a] *Bäume*.

```
File
  Name './example_struct_attr_access.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')))
        // Exp(Attr(Name('st'), Name('y')))
        Ref(Global(Num('0')))
        Ref(Attr(Stack(Num('1')), Name('y')))
        Exp(Stack(Num('1')))
        Return(Empty())
      ]
  ]
```

Code 0.31: PicoC-Mon Pass für Zugriff auf Structattribut

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

```
File
  Name './example_struct_attr_access.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')))
```

```
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 Structattribut

0.0.3.4 Zuweisung an Structattribut

Die **Zuweisung an ein Structattribut** (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 Structattribut

Im **Abstact Syntax Tree** wird eine **Zuweisung an ein Structattribut** (z.B. `st.y = 42`) durch die Komposition `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: Abstract Syntax Tree für Zuweisung an Structattribut

Im **PicoC-Mon Pass** in Code 0.35 wird die Komposition `Assign(Attr(Name('st'), Name('y')), Num('42'))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Arrayelement** `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` in Subkapitel 0.0.2.3 darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsnten auf das Unterkapitel 0.0.2.3 verwiesen.

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

```
1 File
2   Name './example_struct_attr_assignment.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        // Assign(Attr(Name('st'), Name('y')), Num('42'))
12        Exp(Num('42'))
13        Ref(Global(Num('0')))
14        Ref(Attr(Stack(Num('1')), Name('y')))
15        Assign(Stack(Num('1')), Stack(Num('2')))
16        Return(Empty())
17      ]
18  ]
```

Code 0.35: PicoC-Mon Pass für Zuweisung an Structattribut

Im **RETI-Blocks Pass** in Code 0.36 werden die **Kompositionen** `Exp(Num('42'))`, `Ref(Global(Num('0')))`, `Ref(Attr(Stack(Num('1')), Name('y')))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre 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;
        STOREIN DS ACC 1;
        LOADIN SP ACC 2;
        STOREIN DS ACC 0;
        ADDI SP 2;
        # // Assign(Attr(Name('st'), Name('y')), 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;
        # Ref(Attr(Stack(Num('1')), Name('y')))
        LOADIN SP IN1 1;
        ADDI IN1 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 0.36: RETI-Blocks Pass für Zuweisung an Structattribut

0.0.4 Umsetzung des Zugriffs auf Derived locations im Allgemeinen

0.0.4.1 Übersicht

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 beschriebenen Datentypen (**Pointer**, **Array** und **Struct**) sehr ähnlich abläuft. Es lässt sich ein allgemeines Vorgehen, bestehend aus einem **Anfangsteil**, **Mittelteil** und **Schlussteil** darin erkennen.

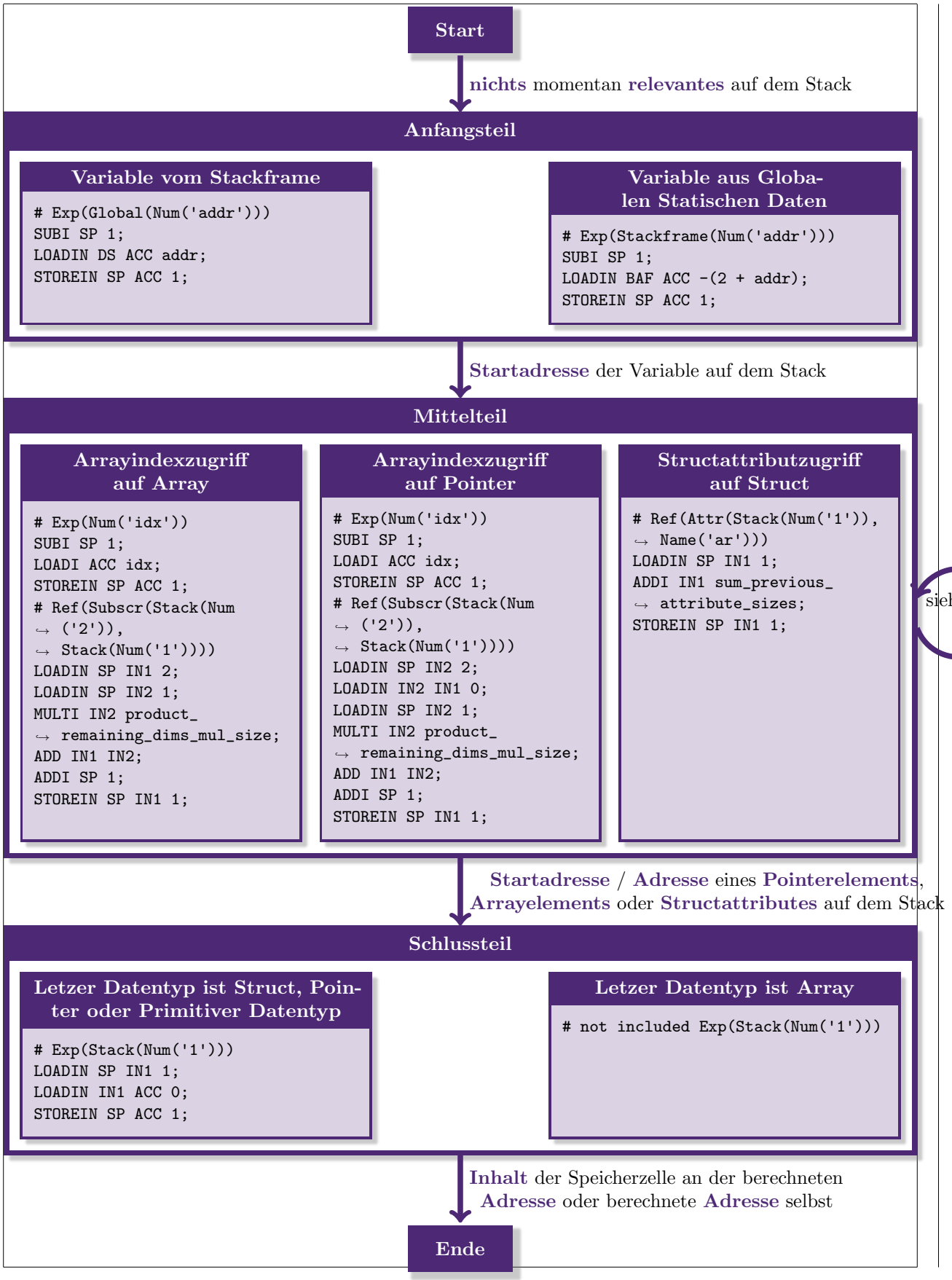


Abbildung 1: Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

Dieses Vorgehen ist in Abbildung 1 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der Datenypen **Pointer**, **Array** und **Struct** gemischt sind (z.B. `(*st_var.ar)[0]`).

Dies ist möglich, indem im **Mittelteil**, je nachdem, ob das versteckte Attribut `datatype` des `Ref(exp, datatype)`-Container-Knotens ein `ArrayDecl(nums, datatype)`, ein `PntrDecl(num, datatype)` oder `StructSpec(name)` beinhaltet und die dazu passende **Zugriffsoperation** `Subscr(exp1, exp2)` oder `Attr(exp, name)` vorliegt, einen anderen **RETI-Code** generiert wird. Dieser **RETI-Code** berechet die **Startadresse** eines gewünschten **Pointerelmements**, **Arrayelements** oder **Structattributs**.

Würde man bei einem `Subscr(Name('var'), exp2)` den Datentyp der Variable `Name('var')` von `ArrayDecl(nums, IntType())` zu `PointerDecl(num, IntType())` ändern, müsste nur der **Mittelteil** ausgetauscht werden. **Anfangsteil** und **Schlussteil** bleiben unverändert.

Die **Zugriffsoperation** muss dabei zum **Datentyp** im versteckten Attribut `datatype` passen, ansonsten gibt es eine `DatatypeMismatch`-**Fehlermeldung**. Ein **Zugriff auf ein Arrayindex** `Subscr(exp1, epp2)` kann dabei mit den Datentypen **Array** `ArrayDecl(nums, datatype)` und **Pointer** `PntrDecl(num, datatype)` kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Pointer** `PntrDecl(num, datatype)` die **Adresse**, die auf dem **Stack** liegt auf eine Speicherzelle mit einer weiteren **Adresse** zeigt und das gewünschte Element erst zu finden ist, wenn man der letzteren **Adresse** folgt. Ein **Zugriff auf ein Structattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.

> Um Verwirrung vorzubeugen, wird hier vorausschauend nochmal darauf hingewiesen, dass eine **Dereferenzierung** in der Form `Deref(exp1, exp2)` nicht mehr existiert, denn wie in Unterkapitel 0.0.1 bereits erklärt wurde, wurde der **Container-Knoten** `Deref(exp1, exp2)` im **PicoC-Shrink Pass** durch `Subscr(exp1, exp2)` ersetzt. Das hatte den Zweck, **doppelten Code** zu vermeiden, da die **Dereferenzierung** und der **Zugriff auf ein Arrayelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Arrayindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

Das versteckte Attribut `datatype` beinhaltet den **Unterdatentyp**, in welchem der Zugriff auf ein **Pointerelment**, **Arrayelement** oder **Structattribut** erfolgt. Der **Unterdatentyp** ist dabei ein **Teilbaum** des Baumes, der vom gesamten **Datentyp** der **Variable** gebildet wird. Wobei man sich allerdings nur für den obersten **Container-Knoten** oder **Token-Knoten** in diesem **Unterdatentyp** interessiert und die möglicherweise unter diesem momentan betrachteten **Knoten** liegenden **Container-Knoten** und **Token-Knoten** in einem anderen `Ref(exp, versteckte Attribut)`-Container-Knoten dem versteckte Attribut zugeordnet sind. Das versteckte Attribut `datatype` enthält also die Information auf welchen **Unterdatentyp** im dem momentanen **Kontext** gerade zugegriffen wird.

Der **Anfangsteil**, der durch die Komposition `Ref(Name('var'))` repräsentiert wird, ist dafür zuständig die **Startadresse** der Variablen `Name('var')` auf den Stack zu schreiben und je nachdem, ob diese Variable in den **Globalen Statischen Daten** oder auf dem **Stackframe** liegt einen anderen **RETI-Code** zu generieren.

Der **Schlussteil** wird durch die Komposition `Exp(Stack(Num('1')), datatype)` dargestellt. Je nachdem, ob versteckte Attribut `datatype` ein `CharType()`, `IntType()`, `PntrDecl(num, datatype)` oder `StructType(name)` ist, wird ein entsprechender **RETI-Code** generiert, der die **Adresse**, die auf dem **Stack** liegt dazu nutzt, um den **Inhalt** der Speicherzelle an dieser **Adresse** auf den **Stack** zu schreiben. Dabei wird die Speicherzelle der **Adresse** mit dem **Inhalt** auf den sie selbst zeigt überschreiben. 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.

Arrays haben in der Sprache L_C und somit auch in L_{PiocC} die Eigenheit, dass wenn auf ein gesamtes **Array**

zugegriffen wird[8], 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 wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

Implementieren lässt sich dieses Vorgehen, indem beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)` Ausdrucks ein `Exp(Stack(Num('1')))` an die Spitze einer **Liste der generierten Ausdrücke** gesetzt wird und der Ausdruck selbst als `exp`-Attribut des `Ref(exp)`-Knotens gesetzt wird und hinter dem `Exp(Stack(Num('1')))`-Container-Knoten in der Liste eingefügt wird. Beim Antreffen eines `Ref(exp)` wird fast gleich vorgegangen, wie beim Antreffen eines `Subscr(exp1, exp2)` oder `Attr(exp, name)`, nur, dass kein `Exp(Stack(Num('1')))` vorne an die Spitze der **Liste der generierten Ausdrücke** gesetzt wird. Und ein `Ref(exp)` bei dem `exp` direkt ein `Name(str)` ist, wird dieser einfach direkt durch `Ref(Global(num))` bzw. `Ref(Stackframe(num))` ersetzt.

Es wird solange dem jeweiligen `exp1` des `Subscr(exp1, exp2)`-Knoten, dem `exp` des `Attr(exp, name)`-Knoten oder dem `exp` des `Ref(exp)`-Knoten gefolgt und der jeweilige **Container-Knoten** selbst als `exp` des `Ref(exp)`-Knoten eingesetzt und hinten in die **Liste der generierten Ausdrücke** eingefügt, bis man bei einem `Name(name)` ankommt. Der `Name(name)`-Knoten wird zu einem `Ref(Global(num))` oder `Ref(Stackframe(num))` umgewandelt und ebenfalls ganz hinten in die **Liste der generierten Ausdrücke** eingefügt. Wenn man dem `exp` Attribut eines `Ref(exp)`-Knoten folgt, wird allerdings kein `Ref(exp)` in die **Liste der generierten Ausdrücke** eingefügt, sondern das `datatype`-Attribut des zuletzt eingefügten `Ref(exp, datatype)` manipuliert, sodass dessen `datatype` in ein `ArrayDecl([Num('1')], datatype)` eingebettet ist und so ein auf das `Ref(exp)` folgendes `Deref(exp1, exp2)` oder `Subscr(exp1, exp2)` direkt behandelt wird.

Parallel wird eine Liste der `Ref(exp)`-Knoten geführt, deren versteckte Attribute `datatype` und `error_data` die entsprechenden Informationen zugewiesen bekommen müssen. Sobald man beim `Name(name)`-Knoten angekommen ist und mithilfe dieses in der **Symboltabelle** den **Dantentyp** der Variable nachsehen kann, wird der **Datentyp** der Variable nun ebenfalls, wie die Ausdrücke `Subscr(exp1, exp2)` und `Attr(exp, name)` schrittweise durchiteriert und dem jeweils nächsten `datatype`-Attribut gefolgt werden. Das **Iterieren** über den **Datentyp** wird solange durchgeführt, bis alle `Ref(exp)`-Knoten ihren im jeweiligen **Kontext** vorliegenden **Datentyp** in ihrem `datatype`-Attribut zugewiesen bekommen haben. Alles andere führt zu einer **Fehlermeldung**, für die das versteckte Attribut `error_data` genutzt wird.[a]

[a] Man kann diese Implementierung gut mit dem **Auseinanderrollen** und **Wieder-Einrollen** einer **Spirale** vergleichen.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil** 0.0.4.2, **Mittelteil** 0.0.4.3 und **Schlussteil** 0.0.4.4 bei der Kompilierung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente**, **Structattribute** bei gemischten Ausdrücken, wie `(*st_first.ar)[0];` einzeln isoliert betrachtet und erläutert.

0.0.4.2 Anfangsteil für Globale Statische Daten und Stackframe

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

```
1 struct ar_with_len {int len; int ar[2];};
2
3 void main() {
4   struct ar_with_len st_ar[3];
5   int *(*complex_var)[3];
6   &complex_var;
```

[8] Und nicht auf ein **Element** des Arrays.
[8] **Startadresse / Adresse** eines **Pointerelements**, **Arrayelements** oder **Structattributes** auf dem Stack.

```
}

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

Code 0.37: PicoC-Code für den Anfangsteil

Im **Abstract Syntax Tree** in Code 0.38 wird die **Refererenzierung** `&complex_var` mit der Komposition `Exp(Ref(Name('complex_var')))` dargestellt. Üblicherweise wird aber einfach nur `Ref(Name('complex_var')` geschrieben, aber da beim Erstellen des **Abstract Syntx Tree** jeder **Logischer Ausdruck** in ein `Exp(exp)` eingebettet wird, ist das `Ref(Name('complex_var')` in ein `Exp()` eingebettet. Man müsste an vielen Stellen eine gesonderte **Fallunterschiedung** aufstellen, um von `Exp(Ref(Name('complex_var')))` das `Exp()` zu entfernen, obwohl das `Exp()` in den darauffolgenden **Passes** so oder so herausgefiltet wird. 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',
      [],
      [
        Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('complex_var')))
        Exp(Ref(Name('complex_var')))
      ]
  ]
```

Code 0.38: Abstract Syntax Tree für den Anfangsteil

Im **PicoC-Mon Pass** in Code 0.39 wird die Komposition `Exp(Ref(Name('complex_var')))` durch die Komposition `Ref(Global(Num('9')))` bzw. `Ref(Stackframe(Num('9')))` ersetzt, je nachdem, ob die Variable `Name('complex_var')` in den **Globalen Statischen Daten** oder auf dem **Stack** liegt.

```
File
  Name './example_derived_dts_introduction_part.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        // 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')))
        Ref(Global(Num('9')))
        Return(Empty())
      ],
    Block
      Name 'fun.0',
      [
        // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
        ↪  Name('st_ar')))
        // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
        ↪  Name('complex_var')))
        // Exp(Ref(Name('complex_var')))
        Ref(Stackframe(Num('9')))
        Return(Empty())
      ]
  ]
```

Code 0.39: PicoC-Mon Pass für den Anfangsteil

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

```
File
  Name './example_derived_dts_introduction_part.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // 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')))
        # Ref(Global(Num('9')))
        SUBI SP 1;
        LOADI IN1 9;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Return(Empty())
```

```
16        LOADIN BAF PC -1;
17      ],
18    Block
19      Name 'fun.0',
20      [
21        # // Exp(Alloc(Writeable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
          ↪  Name('st_ar')))
22        # // Exp(Alloc(Writeable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
          ↪  IntType('int'))), Name('complex_var')))
23        # // Exp(Ref(Name('complex_var')))
24        # Ref(Stackframe(Num('9')))
25        SUBI SP 1;
26        MOVE BAF IN1;
27        SUBI IN1 11;
28        STOREIN SP IN1 1;
29        # Return(Empty())
30        LOADIN BAF PC -1;
31      ]
32  ]
```

Code 0.40: RETI-Blocks Pass für den Anfangsteil

0.0.4.3 Mittelteil für die verschiedenen Derived Datatypes

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinandereiheung von Zugriffen auf **Pointerelemente**, **Arrayelemente** oder **Structattribute** berechnet wird (z.B. `(*complex_var.ar)[0]`), 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)[0];
7 }
```

Code 0.41: PicoC-Code für den Mittelteil

```
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')), Num('0')))
17      ]
18  ]
```

Code 0.42: Abstract Syntax Tree für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Array([Num('42')]))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('complex_var'), Struct([Assign(Name('ar'), Ref(Name('var')))]))
11        Ref(Global(Num('0')))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')), Num('0')))
14        Ref(Global(Num('1')))
15        Ref(Attr(Stack(Num('1')), Name('ar')))
16        Exp(Num('0'))
17        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        Return(Empty())
22      ]
23  ]
```

Code 0.43: PicoC-Mon Pass für den Mittelteil

```
1 File
2   Name './example_derived_dts_main_part.reti_blocks',
3   [
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')), Num('0')))
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('0'))
49        SUBI SP 1;
50        LOADI ACC 0;
51        STOREIN SP ACC 1;
52        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
53        LOADIN SP IN1 2;
54        LOADIN SP IN2 1;
55        MULTI IN2 1;
56        ADD IN1 IN2;
57        ADDI SP 1;
58        STOREIN SP IN1 1;
59        # Exp(Stack(Num('1')))
60        LOADIN SP IN1 1;
61        LOADIN IN1 ACC 0;
62        STOREIN SP ACC 1;
63        # Return(Empty())
64        LOADIN BAF PC -1;
65    ]
66 ]
```

Code 0.44: RETI-Blocks Pass für den Mittelteil

0.0.4.4 Schlussteil für die verschiedenen Derived Datatypes

```
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

```
1  File
2    Name './example_derived_dts_final_part.ast',
3    [
4      StructDecl
5        Name 'st',
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: Abstract Syntax Tree für den Schlussteil

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

```
4     Block
5       Name 'main.0',
6       [
7         // Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
          ↪  Name('complex_var1')))
8         // Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
          ↪  Name('complex_var2')))
9         // Assign(Name('var'), Num('42'))
10        Exp(Num('42'))
11        Assign(Global(Num('4')), Stack(Num('1')))
12        // Assign(Name('pntr1'), Ref(Name('var')))
13        Ref(Global(Num('4')))
14        Assign(Global(Num('5')), Stack(Num('1')))
15        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
16        Ref(Global(Num('5')))
17        Assign(Global(Num('6')), Stack(Num('1')))
18        // Exp(Subscr(Name('complex_var1'), Num('0')))
19        Ref(Global(Num('0')))
20        Exp(Num('0'))
21        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22        Exp(Stack(Num('1')))
23        // Exp(Subscr(Name('complex_var2'), Num('0')))
24        Ref(Global(Num('2')))
25        Exp(Num('0'))
26        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        Exp(Stack(Num('1')))
28        // Exp(Subscr(Name('complex_var3'), Num('0')))
29        Ref(Global(Num('6')))
30        Exp(Num('0'))
31        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
32        Exp(Stack(Num('1')))
33        Return(Empty())
34      ]
35  ]
```

Code 0.47: PicoC-Mon Pass für den Schlussteil

```
1 File
2   Name './example_derived_dts_final_part.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
          ↪  Name('complex_var1')))
8         # // Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
          ↪  Name('complex_var2')))
9         # // Assign(Name('var'), Num('42'))
10        # Exp(Num('42'))
11        SUBI SP 1;
12        LOADI ACC 42;
13        STOREIN SP ACC 1;
14        # Assign(Global(Num('4')), Stack(Num('1')))
15        LOADIN SP ACC 1;
```

```
16          STOREIN DS ACC 4;
17          ADDI SP 1;
18          # // Assign(Name('pntr1'), Ref(Name('var')))
19          # Ref(Global(Num('4')))
20          SUBI SP 1;
21          LOADI IN1 4;
22          ADD IN1 DS;
23          STOREIN SP IN1 1;
24          # Assign(Global(Num('5')), Stack(Num('1')))
25          LOADIN SP ACC 1;
26          STOREIN DS ACC 5;
27          ADDI SP 1;
28          # // Assign(Name('complex_var3'), Ref(Name('pntr1')))
29          # Ref(Global(Num('5')))
30          SUBI SP 1;
31          LOADI IN1 5;
32          ADD IN1 DS;
33          STOREIN SP IN1 1;
34          # Assign(Global(Num('6')), Stack(Num('1')))
35          LOADIN SP ACC 1;
36          STOREIN DS ACC 6;
37          ADDI SP 1;
38          # // Exp(Subscr(Name('complex_var1'), Num('0')))
39          # Ref(Global(Num('0')))
40          SUBI SP 1;
41          LOADI IN1 0;
42          ADD IN1 DS;
43          STOREIN SP IN1 1;
44          # Exp(Num('0'))
45          SUBI SP 1;
46          LOADI ACC 0;
47          STOREIN SP ACC 1;
48          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
49          LOADIN SP IN1 2;
50          LOADIN SP IN2 1;
51          MULTI IN2 2;
52          ADD IN1 IN2;
53          ADDI SP 1;
54          STOREIN SP IN1 1;
55          # // not included Exp(Stack(Num('1')))
56          # // Exp(Subscr(Name('complex_var2'), Num('0')))
57          # Ref(Global(Num('2')))
58          SUBI SP 1;
59          LOADI IN1 2;
60          ADD IN1 DS;
61          STOREIN SP IN1 1;
62          # Exp(Num('0'))
63          SUBI SP 1;
64          LOADI ACC 0;
65          STOREIN SP ACC 1;
66          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
67          LOADIN SP IN1 2;
68          LOADIN SP IN2 1;
69          MULTI IN2 2;
70          ADD IN1 IN2;
71          ADDI SP 1;
72          STOREIN SP IN1 1;
```

```
73          # Exp(Stack(Num('1')))
74          LOADIN SP IN1 1;
75          LOADIN IN1 ACC 0;
76          STOREIN SP ACC 1;
77          # // Exp(Subscr(Name('complex_var3'), Num('0')))
78          # Ref(Global(Num('6')))
79          SUBI SP 1;
80          LOADI IN1 6;
81          ADD IN1 DS;
82          STOREIN SP IN1 1;
83          # Exp(Num('0'))
84          SUBI SP 1;
85          LOADI ACC 0;
86          STOREIN SP ACC 1;
87          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
88          LOADIN SP IN2 2;
89          LOADIN IN2 IN1 0;
90          LOADIN SP IN2 1;
91          MULTI IN2 1;
92          ADD IN1 IN2;
93          ADDI SP 1;
94          STOREIN SP IN1 1;
95          # Exp(Stack(Num('1')))
96          LOADIN SP IN1 1;
97          LOADIN IN1 ACC 0;
98          STOREIN SP ACC 1;
99          # Return(Empty())
100         LOADIN BAF PC -1;
101     ]
102 ]
```

Code 0.48: RETI-Blocks Pass für den Schlussteil

Literatur

Online

- *Bäume*. URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17. 07. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 07. 2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).

Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).