
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

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

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

0.0.1	Umsetzung von Pointern	9
0.0.1.1	Referenzierung	9
0.0.1.2	Dereferenzierung durch Zugriff auf Arrayindex ersetzen	11
0.0.2	Umsetzung von Arrays	12
0.0.2.1	Initialisierung von Arrays	12
0.0.2.2	Zugriff auf einen Arrayindex	17
0.0.2.3	Zuweisung an Arrayindex	22
0.0.3	Umsetzung von Structs	25
0.0.3.1	Deklaration und Definition von Structtypen	25
0.0.3.2	Initialisierung von Structs	27
0.0.3.3	Zugriff auf Structattribut	30
0.0.3.4	Zuweisung an Structattribut	34
0.0.4	Umsetzung des Zugriffs auf Derived datatypes im Allgemeinen	36
0.0.4.1	Übersicht	36
0.0.4.2	Anfangsteil	39
0.0.4.3	Mittelteil	41
0.0.4.4	Schluss teil	45
0.0.5	Umsetzung von Funktionen	49
0.0.5.1	Mehrere Funktionen	49
0.0.5.1.1	Sprung zur Main Funktion	52
0.0.5.2	Funktionsdeklaration und -definition und Umsetzung von Scopes	54
0.0.5.3	Funktionsaufruf	57
0.0.5.3.1	Ohne Rückgabewert	57
0.0.5.3.2	Mit Rückgabewert	59
0.0.5.3.3	Umsetzung von Call by Sharing für Arrays	62
0.0.5.3.4	Umsetzung von Call by Value für Structs	64
0.1	Fehlermeldungen	66
0.1.1	Error Handler	66
0.1.2	Arten von Fehlermeldungen	66
0.1.2.1	Syntaxfehler	66
0.1.2.2	Laufzeitfehler	66

Abbildungsverzeichnis

1	Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes	37
---	---------------------------------------------------------------------------	----

Codeverzeichnis

0.1	PicoC-Code für Pointer Referenzierung	9
0.2	Abstract Syntax Tree für Pointer Referenzierung	9
0.3	Symboltabelle für Pointer Referenzierung	10
0.4	PicoC-Mon Pass für Pointer Referenzierung	10
0.5	RETI-Blocks Pass für Pointer Referenzierung	11
0.6	PicoC-Code für Pointer Dereferenzierung	11
0.7	Abstract Syntax Tree für Pointer Dereferenzierung	12
0.8	PicoC-Shrink Pass für Pointer Dereferenzierung	12
0.9	PicoC-Code für Array Initialisierung	13
0.10	Abstract Syntax Tree für Array Initialisierung	13
0.11	Symboltabelle für Array Initialisierung	14
0.12	PicoC-Mon Pass für Array Initialisierung	15
0.13	RETI-Blocks Pass für Array Initialisierung	17
0.14	PicoC-Code für Zugriff auf einen Arrayindex	17
0.15	Abstract Syntax Tree für Zugriff auf einen Arrayindex	18
0.16	PicoC-Mon Pass für Zugriff auf einen Arrayindex	20
0.17	RETI-Blocks Pass für Zugriff auf einen Arrayindex	21
0.18	PicoC-Code für Zuweisung an Arrayindex	22
0.19	Abstract Syntax Tree für Zuweisung an Arrayindex	22
0.20	PicoC-Mon Pass für Zuweisung an Arrayindex	23
0.21	RETI-Blocks Pass für Zuweisung an Arrayindex	24
0.22	PicoC-Code für die Deklaration eines Structtyps	25
0.23	Abstract Syntax Tree für die Deklaration eines Structtyps	25
0.24	Symboltabelle für die Deklaration eines Structtyps	27
0.25	PicoC-Code für Initialisierung von Structs	27
0.26	Abstract Syntax Tree für Initialisierung von Structs	28
0.27	PicoC-Mon Pass für Initialisierung von Structs	29
0.28	RETI-Blocks Pass für Initialisierung von Structs	30
0.29	PicoC-Code für Zugriff auf Structattribut	30
0.30	Abstract Syntax Tree für Zugriff auf Structattribut	30
0.31	PicoC-Mon Pass für Zugriff auf Structattribut	33
0.32	RETI-Blocks Pass für Zugriff auf Structattribut	34
0.33	PicoC-Code für Zuweisung an Structattribut	34
0.34	Abstract Syntax Tree für Zuweisung an Structattribut	34
0.35	PicoC-Mon Pass für Zuweisung an Structattribut	35
0.36	RETI-Blocks Pass für Zuweisung an Structattribut	36
0.37	PicoC-Code für den Anfangsteil	39
0.38	Abstract Syntax Tree für den Anfangsteil	40
0.39	PicoC-Mon Pass für den Anfangsteil	41
0.40	RETI-Blocks Pass für den Anfangsteil	41
0.41	PicoC-Code für den Mittelteil	42
0.42	Abstract Syntax Tree für den Mittelteil	42
0.43	PicoC-Mon Pass für den Mittelteil	43
0.44	RETI-Blocks Pass für den Mittelteil	44
0.45	PicoC-Code für den Schlussteil	45
0.46	Abstract Syntax Tree für den Schlussteil	46
0.47	PicoC-Mon Pass für den Schlussteil	46

0.48 RETI-Blocks Pass für den Schlussteil	48
0.49 PicoC-Code für 3 Funktionen	49
0.50 Abstract Syntax Tree für 3 Funktionen	50
0.51 PicoC-Blocks Pass für 3 Funktionen	51
0.52 PicoC-Mon Pass für 3 Funktionen	51
0.53 RETI-Blocks Pass für 3 Funktionen	52
0.54 PicoC-Code für Funktionen, wobei die main Funktion nicht die erste Funktion ist	52
0.55 RETI-Blocks Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	53
0.56 RETI-Patch Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	54
0.57 RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist	54
0.58 PicoC-Code für Funktionen, wobei eine Funktion vorher deklariert werden muss	55
0.59 Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss	57
0.60 PicoC-Code für Funktionsaufruf ohne Rückgabewert	57
0.61 PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert	58
0.62 RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert	59
0.63 RETI-Pass für Funktionsaufruf ohne Rückgabewert	59
0.64 PicoC-Code für Funktionsaufruf mit Rückgabewert	59
0.65 PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert	60
0.66 RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert	61
0.67 RETI-Pass für Funktionsaufruf mit Rückgabewert	61
0.68 PicoC-Code für Call by Sharing für Arrays	62
0.69 PicoC-Mon Pass für Call by Sharing für Arrays	62
0.70 Symboltabelle für Call by Sharing für Arrays	63
0.71 RETI-Block Pass für Call by Sharing für Arrays	64
0.72 PicoC-Code für Call by Value für Structs	64
0.73 PicoC-Mon Pass für Call by Value für Structs	65
0.74 RETI-Block Pass für Call by Value für Structs	66

Tabellenverzeichnis

Definitionsverzeichnis

0.1	Entarteter Baum	32
0.2	Funktionsprototyp	55
0.3	Scope (bzw. Sichtbarkeitsbereich)	56

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(Writable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
11              ↪ Ref(Name('var')))
12      ]
13  ]
```

Code 0.2: Abstract Syntax Tree für Pointer Referenzierung

Bevor man einem **Pointer** 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`-Attribut 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()
```

```

11     },
12     Symbol
13     {
14         type qualifier:      Writeable()
15         datatype:           IntType('int')
16         name:               Name('var@main')
17         value or address:    Num('0')
18         position:           Pos(Num('2'), Num('6'))
19         size:               Num('1')
20     },
21     Symbol
22     {
23         type qualifier:      Writeable()
24         datatype:           PtrDecl(Num('1'), IntType('int'))
25         name:               Name('ptr@main')
26         value or address:    Num('1')
27         position:           Pos(Num('3'), Num('7'))
28         size:               Num('1')
29     }
30 ]

```

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.

```

1 File
2   Name './example_ptr_ref.picoc_mon',
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('ptr'), 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-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.

```

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 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.

```

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

```

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 hierbei **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'))`.

```

1 File
2   Name './example_pntr_deref.ast',

```

```

3  [
4      FunDef
5          VoidType 'void',
6          Name 'main',
7          [],
8          [
9              Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10             Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11                 ↪ Ref(Name('var')))
12             Exp(Deref(Name('ptr'), Num('0'))))
13         ]
14     ]

```

Code 0.7: Abstract Syntax Tree für Pointer Dereferenzierung

Im **PicoC-Shrink Pass** in Code 0.8 wird ein Trick angewendet, bei dem jeder Knoten `Deref(Name('ptr'), Num('0'))` einfach durch den Knoten `Subscr(Name('ptr'), 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]`) verhält¹. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Dereferenzierung** (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_ptr_deref.picoc_shrink',
3   [
4       FunDef
5           VoidType 'void',
6           Name 'main',
7           [],
8           [
9               Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
10              Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('ptr')),
11                  ↪ Ref(Name('var')))
12              Exp(Subscr(Name('ptr'), Num('0'))))
13          ]
14      ]

```

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.

¹In 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 tun 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.

```

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

```

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.

```

1 File
2   Name './example_array_init.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
10            ↪ Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
11            ↪ Array([Num('4')]))])
12       ],
13     FunDef
14       VoidType 'void',
15       Name 'fun',
16       [],
17       [
18         Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('2')], IntType('int')),
19            ↪ Name('ar')), Array([Array([Num('3'), Num('4')]), Array([Num('5'), Num('6')]))])
20       ]
21   ]

```

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 kann². 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**

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

Statischen Daten³.

Das **Größe** des Arrays datatype `ar[dim1]...[dimk]`, die ihm **size**-Attribut 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**: $\text{size}(\text{datatype}(\text{ar})) = \left(\prod_{j=1}^n \text{dim}_j\right) \cdot \text{size}(\text{datatype})^a$.

^aDie **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

```

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()
11    },
12    Symbol
13    {
14      type qualifier:      Writeable()
15      datatype:            ArrayDecl([Num('2'), Num('1')], IntType('int'))
16      name:                 Name('ar@main')
17      value or address:    Num('0')
18      position:            Pos(Num('2'), Num('6'))
19      size:                 Num('2')
20    },
21    Symbol
22    {
23      type qualifier:      Empty()
24      datatype:            FunDecl(VoidType('void'), Name('fun'), [])
25      name:                 Name('fun')
26      value or address:    Empty()
27      position:            Pos(Num('5'), Num('5'))
28      size:                 Empty()
29    },
30    Symbol
31    {
32      type qualifier:      Writeable()
33      datatype:            ArrayDecl([Num('2'), Num('2')], IntType('int'))
34      name:                 Name('ar@fun')
35      value or address:    Num('3')
36      position:            Pos(Num('6'), Num('6'))
37      size:                 Num('4')
38    }
39  ]

```

Code 0.11: Symboltabelle für Array Initialisierung

³Wenn 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 **Pioco-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** geschrieben⁴.

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.

```

1 File
2   Name './example_array_init.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
8         ↪   Array([Num('4')])])])
9         Exp(Num('3'))
10        Exp(Num('1'))
11        Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
12        Exp(Num('4'))
13        Assign(Global(Num('0')), Stack(Num('2')))
14      ],
15    Block
16      Name 'fun.0',
17      [
18        // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
19        ↪   Num('6')])])])
20        Exp(Num('3'))
21        Exp(Num('4'))
22        Exp(Num('5'))
23        Exp(Num('6'))
24        Assign(Stackframe(Num('3')), Stack(Num('4')))
25      ],
26    ]

```

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')))`,

⁴Da 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.

```

1 File
2   Name './example_array_init.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Array([BinOp(Num('3')), Add('+'), Num('1'))]),
8         ↪   Array([Num('4')]))])
9         # Exp(Num('3'))
10        SUBI SP 1;
11        LOADI ACC 3;
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        # Exp(Num('4'))
24        SUBI SP 1;
25        LOADI ACC 4;
26        STOREIN SP ACC 1;
27        # Assign(Global(Num('0')), Stack(Num('2')))
28        LOADIN SP ACC 1;
29        STOREIN DS ACC 1;
30        LOADIN SP ACC 2;
31        STOREIN DS ACC 0;
32        ADDI SP 2;
33        # Return(Empty())
34        LOADIN BAF PC -1;
35      ],
36    Block
37      Name 'fun.0',
38      [
39        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
40        ↪   Num('6')]))])
41        # Exp(Num('3'))
42        SUBI SP 1;
43        LOADI ACC 3;
44        STOREIN SP ACC 1;
45        # Exp(Num('4'))
46        SUBI SP 1;
47        LOADI ACC 4;
48        STOREIN SP ACC 1;
49        # Exp(Num('5'))
50        SUBI SP 1;
51        LOADI ACC 5;
52        STOREIN SP ACC 1;
53        # 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(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
10              ↪ Array([Num('42')]))
11       ]
12     ]

```

```

11     ],
12     FunDef
13     VoidType 'void',
14     Name 'fun',
15     [],
16     [
17         Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
18             ↪ Array([Num('1'), Num('2'), Num('3')]))
19         Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
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 Funktion `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 Zugriffen 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 Zugriffen 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 verstecktes 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 Arrayindex** `var[1]` miteinander zu kombinieren, was in Subkapitel 0.0.4.3 allgemein erklärt ist.

Im letzten Schritt, dem **Schlusssteil 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 **Schlusssteil** `Exp(Stack(Num('1')))` auf eine andere Weise verarbeitet (siehe Subkapitel 0.0.4.4). Der **Unterdatentyp** ist dabei ein verstecktes 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 `datatype ar[dim1]...[dimn]` abgespeichert ist, kann mittels der Formel 0.0.1:

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

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 `idxi` in der Formel auftaucht.

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

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** `M[ref(ar[idx1]...[idxn])]` der Speicherzelle an der finalen **Adresse** `ref(ar[idx1]...[idxn])`.

^aScholl, „Betriebssysteme“.

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

```

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

```

```

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

```

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')), Stack(Num('1'))))` 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        # Ref(Global(Num('0')))
17        SUBI SP 1;
18        LOADI IN1 0;
19        ADD IN1 DS;
20        STOREIN SP IN1 1;
21        # Exp(Num('0'))
22        SUBI SP 1;
23        LOADI ACC 0;
24        STOREIN SP ACC 1;
25        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
26        LOADIN SP IN1 2;
27        LOADIN SP IN2 1;
28        MULTI IN2 1;
29        ADD IN1 IN2;
30        ADDI SP 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    Block
40      Name 'fun.0',

```

```

41      [
42          # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
43          # Exp(Num('1'))
44          SUBI SP 1;
45          LOADI ACC 1;
46          STOREIN SP ACC 1;
47          # Exp(Num('2'))
48          SUBI SP 1;
49          LOADI ACC 2;
50          STOREIN SP ACC 1;
51          # Exp(Num('3'))
52          SUBI SP 1;
53          LOADI ACC 3;
54          STOREIN SP ACC 1;
55          # Assign(Stackframe(Num('2')), Stack(Num('3')))
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          ADDI SP 3;
63          # Ref(Stackframe(Num('2')))
64          SUBI SP 1;
65          MOVE BAF IN1;
66          SUBI IN1 4;
67          STOREIN SP IN1 1;
68          # Exp(Num('1'))
69          SUBI SP 1;
70          LOADI ACC 1;
71          STOREIN SP ACC 1;
72          # Exp(Num('1'))
73          SUBI SP 1;
74          LOADI ACC 1;
75          STOREIN SP ACC 1;
76          # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
77          LOADIN SP ACC 2;
78          LOADIN SP IN2 1;
79          ADD ACC IN2;
80          STOREIN SP ACC 2;
81          ADDI SP 1;
82          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
83          LOADIN SP IN1 2;
84          LOADIN SP IN2 1;
85          MULTI IN2 1;
86          ADD IN1 IN2;
87          ADDI SP 1;
88          STOREIN SP IN1 1;
89          # Exp(Stack(Num('1')))
90          LOADIN SP IN1 1;
91          LOADIN IN1 ACC 0;
92          STOREIN SP ACC 1;
93          # Return(Empty())
94          LOADIN BAF PC -1;
95      ]
96  ]

```

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(Writable(), 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 Kompositionen, 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 berechnet, 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.

```
1 File
2   Name './example_array_assignment.picoc_mon',
```

⁵Ist in Tabelle ?? genauer beschrieben ist

```

3  [
4    Block
5      Name 'main.0',
6      [
7        // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
8        Exp(Num('42'))
9        Ref(Global(Num('0')))
10       Exp(Num('2'))
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-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.

```

1 File
2   Name './example_array_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Subscr(Name('ar'), Num('2')), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Ref(Global(Num('0')))
13        SUBI SP 1;
14        LOADI IN1 0;
15        ADD IN1 DS;
16        STOREIN SP IN1 1;
17        # Exp(Num('2'))
18        SUBI SP 1;
19        LOADI ACC 2;
20        STOREIN SP ACC 1;
21        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22        LOADIN SP IN1 2;
23        LOADIN SP IN2 1;
24        MULTI IN2 1;
25        ADD IN1 IN2;
26        ADDI SP 1;
27        STOREIN SP IN1 1;
28        # Assign(Stack(Num('1')), Stack(Num('2')))
29        LOADIN SP IN1 1;
30        LOADIN SP ACC 2;
31        ADDI SP 2;
32        STOREIN IN1 ACC 0;
33        # Return(Empty())
34        LOADIN BAF PC -1;

```


35

]

36

]

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 Attribute dieses Symbols `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Attribut des Symbols 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 Attribute `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt werden. Die Attribute `type_qualifier`, `datatype` und `name` werden z.B. bei Name('ar@st') mithilfe der Attribute von `Alloc(Writable(), 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 `datatype`-Attribut enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structtyp**⁶ und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Die **Größe** einer Variable `st_var`, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st {datatype1 attr1; ... datatypen attrn;}`^a definiert ist (`struct st st_var;`), berechnet sich dabei aus der Summe der **Größen** der einzelnen **Datentypen** `datatype1 ... datatypen` der **Attribute** `attr1, ... attrn` des **Structtyps**: $\text{size}(\text{st}) = \sum_{i=1}^n \text{size}(\text{datatype}_i)$.

^aHier 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`.

```

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    {
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(Writable(), IntType('int'),
25      ↪ Name('len'))Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')),
26      ↪ Name('ar'))])
27      name:                Name('st')
28      value_or_address:    [Name('len@st'), Name('ar@st')]
29      position:            Pos(Num('1'), Num('7'))
30      size:                Num('3')

```

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

```

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 Structtyps

0.0.3.2 Initialisierung von Structs

Die **Initialisierung eines Structs** 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 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.

```

1 File
2   Name './example_struct_init.ast',
3   [
4     StructDecl
5       Name 'st1',
6       [
7         Alloc(Writeable(), ArrayDecl([Num('2')], PtrDecl(Num('1'), IntType('int'))),
7         ↪ Name('attr'))
8     ],

```

```

9      StructDecl
10      Name 'st2',
11      [
12          Alloc(Writable(), IntType('int'), Name('attr1'))
13          Alloc(Writable(), StructSpec(Name('st1')), Name('attr2'))
14      ],
15      FunDef
16      VoidType 'void',
17      Name 'main',
18      [],
19      [
20          Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
21          Assign(Alloc(Writable(), StructSpec(Name('st2')), Name('st')),
22              ↳ Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
23              ↳ Struct([Assign(Name('attr'), Array([Array([Ref(Name('var')),
24              ↳ Ref(Name('var'))]))]))]))))
25      ]
26  ]

```

Code 0.26: Abstract Syntax Tree für Initialisierung von Structs

Im **PicoC-Mon Pass** in Code 0.27 wird die **Komposition** `Assign(Alloc(Writable(), 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 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 **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 Fälle, man hat es auf der **rechten Seite** entweder mit einem **Struct-Initializer**, einem **Array-Initializer** oder einem **Logischen Ausdruck** zu tun. Bei **Array-** und **Struct-Initializer** 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.

```

1 File
2   Name './example_struct_init.picoc_mon',
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('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        SUBI SP 1;
18        LOADIN DS ACC 0;
19        STOREIN SP ACC 1;
20        # Ref(Global(Num('0')))
21        SUBI SP 1;
22        LOADI IN1 0;
23        ADD IN1 DS;
24        STOREIN SP IN1 1;
25        # Ref(Global(Num('0')))
26        SUBI SP 1;
27        LOADI IN1 0;
28        ADD IN1 DS;
29        STOREIN SP IN1 1;
30        # Assign(Global(Num('1')), Stack(Num('3')))
31        LOADIN SP ACC 1;
32        STOREIN DS ACC 3;
33        LOADIN SP ACC 2;

```

```

34     STOREIN DS ACC 2;
35     LOADIN SP ACC 3;
36     STOREIN DS ACC 1;
37     ADDI SP 3;
38     # Return(Empty())
39     LOADIN BAF PC -1;
40 ]
41 ]

```

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(Writable(), IntType('int'), Name('x'))
8         Alloc(Writable(), IntType('int'), Name('y'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Assign(Alloc(Writable(), StructSpec(Name('pos')), Name('st')),
16              ↳ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
17        Exp(Attr(Name('st'), Name('y')))
18      ]
19    ]

```

Code 0.30: Abstract Syntax Tree für Zugriff auf Structattribut

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

Die Komposition $\text{Exp}(\text{Attr}(\text{Name}('st'), \text{Name}('y')))$ 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 **Schlusssteil** 0.0.4.4 aufteilen lassen. In diesem Fall sind es $\text{Ref}(\text{Global}(\text{Num}('0')))$ (**Anfangsteil**), $\text{Ref}(\text{Attr}(\text{Stack}(\text{Num}('1')), \text{Name}('y')))$ (**Mittelteil**) und $\text{Exp}(\text{Stack}(\text{Num}('1')))$ (**Schlusssteil**). Der **Anfangsteil** und **Schlusssteil** sind genau gleich, wie in Subkapitel 0.0.2.2.

Nur für den **Mittelteil** wird eine andere Komposition $\text{Ref}(\text{Attr}(\text{Stack}(\text{Num}('1')), \text{Name}('y')))$ gebraucht. Diese Komposition $\text{Ref}(\text{Attr}(\text{Stack}(\text{Num}('1')), \text{Name}('y')))$ 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 $\text{Ref}(\text{Subscr}(\text{Stack}(\text{Num}('2')), \text{Stack}(\text{Num}('1'))))$ 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 $\text{Ref}(\text{Attr}(\text{Stack}(\text{Num}('1')), \text{Name}('y')))$, nämlich $\text{Name}('y')$. 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** $\text{Ref}(\text{exp}, \text{datatype})$ angehängt.

Sei datatype_i ein **Knoten** eines **entarteten Baumes** (siehe Definition 0.1 und Abbildung 0.0.2), dessen Wurzel datatype_1 ist. Dabei steht i für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen** $\text{ref}(\text{datatype}_i)$ von Speicherbereichen $\text{ref}(\text{datatype}_i) \dots \text{ref}(\text{datatype}_i) + \text{size}(\text{datatype}_i)$ im **Hauptspeicher** zuordnen, wobei gilt, dass $\text{ref}(\text{datatype}_i) \leq \text{ref}(\text{datatype}_{i+1}) < \text{ref}(\text{datatype}_i) + \text{size}(\text{datatype}_i)$.^{ab}

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

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



Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen $(\text{datatype}_{1,\text{idx}_1}, \dots, \text{datatype}_{n,\text{idx}_n})$, die das Resultat einer Aneinandereiung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen

datatype_i ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 0.0.3:

$$\text{ref}(\text{datatype}_{1,\text{idx}_1}, \dots, \text{datatype}_{n,\text{idx}_n}) = \text{ref}(\text{datatype}_1) + \sum_{i=1}^{n-1} \sum_{k=1}^{\text{idx}_i-1} \text{size}(\text{datatype}_{i,k}) \quad (0.0.3)$$

berechnet werden.^c

Dabei darf nur der letzte Knoten datatype_n vom Datentyp **Pointer** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, der nicht der **letzte Datentyp** datatype_n in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp datatype_1 bis direkt zum Datentyp **Pointer** geht $\text{datatype}_{\text{pntr}}$ und die **zweite Adressberechnung** einen Datentyp nach dem Datentyp **Pointer** anfängt $\text{datatype}_{\text{pntr}+1}$ und bis zum letzten Datentyp datatype_n geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** $\text{ref}(\text{datatype}_1)$ des Summanden aus der Formel 0.0.3 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse $M[\text{ref}(\text{datatype}_1, \dots, \text{datatype}_{\text{pntr}})]$ gesetzt werden.

Die Formel 0.0.3 stellt dabei eine **Verallgemeinerung** der Formel 0.0.1 dar, die für alle möglichen Aneinanderreihungen 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 $\text{Ref}(\text{Global}(\text{num}))$ bzw. $\text{Ref}(\text{Stackframe}(\text{num}))$ repräsentiert dabei den Summanden $\text{ref}(\text{datatype}_1)$ in der Formel.

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

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

^aEs ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.

^b $\text{ref}(\text{datatype})$ steht dabei für das Schreiben der **Startadresse**, die dem **Datentyp** datatype zugeordnet ist auf den **Stack**.

^cDie **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** datatype_i , die vor dem **Element** / **Attribut** $\text{datatype}_{i,\text{idx}_i}$ liegen.

^dStructattribute haben **unterschiedliche** Größen.

Definition 0.1: 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

^aBäume.

```
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'),
8          ↪ Num('2'))]))
9        Exp(Num('4'))
10       Exp(Num('2'))
11       Assign(Global(Num('0')), Stack(Num('2')))
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-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.

```

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'),
8           ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # Assign(Global(Num('0')), Stack(Num('2')))
18        LOADIN SP ACC 1;
19        STOREIN DS ACC 1;
20        LOADIN SP ACC 2;
21        STOREIN DS ACC 0;
22        ADDI SP 2;
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')))

```

```

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

```

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 **Abstract 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(Writable(), IntType('int'), Name('x'))
8         Alloc(Writable(), IntType('int'), Name('y'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Assign(Alloc(Writable(), StructSpec(Name('pos')), Name('st')),
16              ↳ Struct([Assign(Name('x'), Num('4')), Assign(Name('y'), Num('2'))]))
17        Assign(Attr(Name('st'), Name('y')), Num('42'))
18      ]
19    ]

```

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 ansonsten 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'),
8           ↪ Num('2'))]))
9         Exp(Num('4'))
10        Exp(Num('2'))
11        Assign(Global(Num('0')), Stack(Num('2')))
12        // Assign(Attr(Name('st'), Name('y')), Num('42'))
13        Exp(Num('42'))
14        Ref(Global(Num('0')))
15        Ref(Attr(Stack(Num('1')), Name('y')))
16        Assign(Stack(Num('1')), Stack(Num('2')))
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.

```

1 File
2   Name './example_struct_attr_assignment.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('st'), Struct([Assign(Name('x'), Num('4')), Assign(Name('y'),
8           ↪ Num('2'))]))
9         # Exp(Num('4'))
10        SUBI SP 1;
11        LOADI ACC 4;
12        STOREIN SP ACC 1;
13        # Exp(Num('2'))
14        SUBI SP 1;
15        LOADI ACC 2;
16        STOREIN SP ACC 1;
17        # 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     # // 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 Structattribut

0.0.4 Umsetzung des Zugriffs auf Derived datatypes 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 **Schlusssteil** darin erkennen.

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 Datentypen **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** berechnet die **Startadresse** eines gewünschten **Pointerelements**, **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 **Schlusssteil** bleiben unverändert.

Die **Zugriffsoperation** muss dabei zum **Datentyp** im versteckten Attribut `datatype` passen, ansonsten gibt

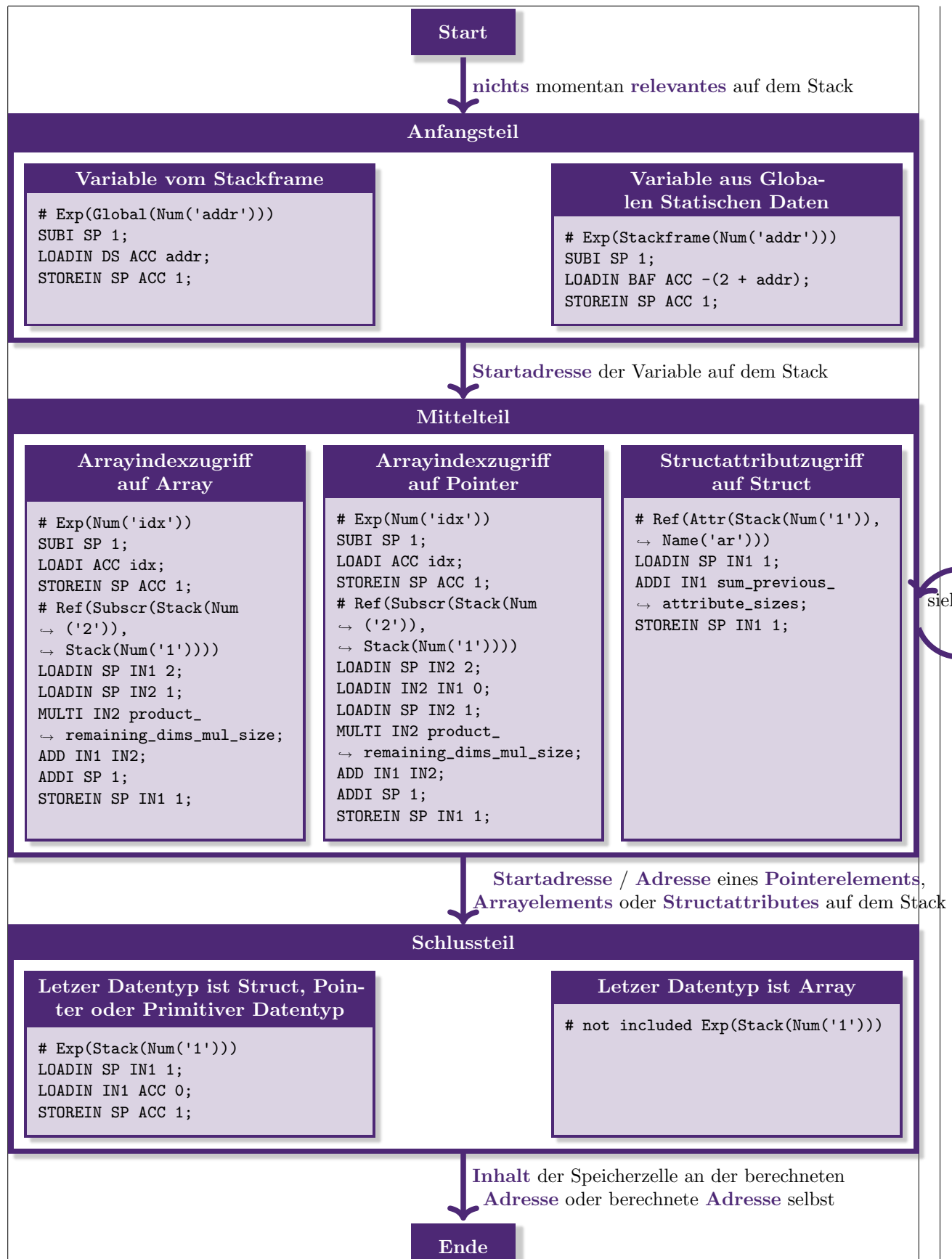


Abbildung 1: Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

es eine **DatatypeMismatch-Fehlermeldung**. Ein **Zugriff auf ein Arrayindex** `Subscr(exp1, exp2)` 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 **Pointerelement**, **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 versteckten 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 **Schlusssteil** wird durch die Komposition `Exp(Stack(Num('1')), datatype)` dargestellt. Je nachdem, ob das 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_{PicoC} die Eigenheit, dass wenn auf ein gesamtes **Array** zugegriffen wird⁸, 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.

⁸Und nicht auf ein **Element** des Arrays.

⁸**Startadresse** / **Adresse** eines **Pointerelements**, **Arrayelements** oder **Structattributes** auf dem **Stack**.

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.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil 0.0.4.2**, **Mittelteil 0.0.4.3** und **Schlusssteil 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

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;
7 }
8
9 void fun() {
10    struct ar_with_len st_ar[3];
11    int (*complex_var)[3];
12    &complex_var;
13 }
```

Code 0.37: PicoC-Code für den Anfangsteil

Im **Abstract Syntax Tree** in Code 0.38 wird die **Referenzierung** `&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 Syntax 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 **Fallunterscheidung** aufstellen, um von `Exp(Ref(Name('complex_var')))` das `Exp()` zu entfernen.

obwohl das `Exp()` in den darauffolgenden **Passes** so oder so herausgefiltert wird. Daher wurde darauf verzichtet den Code ohne triftigen Grund **komplexer** zu machen.

```

1 File
2   Name './example_derived_dts_introduction_part.ast',
3   [
4     StructDecl
5       Name 'ar_with_len',
6       [
7         Alloc(Writable(), IntType('int'), Name('len'))
8         Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [],
14      [
15        Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
16              ↪ Name('st_ar')))
17        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
18              ↪ IntType('int'))), Name('complex_var')))
19        Exp(Ref(Name('complex_var')))
20      ],
21    FunDef
22      VoidType 'void',
23      Name 'fun',
24      [],
25      [
26        Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
27              ↪ Name('st_ar')))
28        Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
29              ↪ Name('complex_var')))
30        Exp(Ref(Name('complex_var')))
31      ]
32    ]
33  ]

```

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.

```

1 File
2   Name './example_derived_dts_introduction_part.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         Ref(Global(Num('9')))
8         Return(Empty())
9       ],
10    Block
11      Name 'fun.0',

```

```

12     [
13         Ref(Stackframe(Num('9')))
14         Return(Empty())
15     ]
16 ]

```

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.

```

1 File
2   Name './example_derived_dts_introduction_part.reti_blocks',
3   [
4       Block
5         Name 'main.1',
6         [
7             # Ref(Global(Num('9')))
8             SUBI SP 1;
9             LOADI IN1 9;
10            ADD IN1 DS;
11            STOREIN SP IN1 1;
12            # Return(Empty())
13            LOADIN BAF PC -1;
14        ],
15        Block
16          Name 'fun.0',
17          [
18              # Ref(Stackframe(Num('9')))
19              SUBI SP 1;
20              MOVE BAF IN1;
21              SUBI IN1 11;
22              STOREIN SP IN1 1;
23              # Return(Empty())
24              LOADIN BAF PC -1;
25          ]
26    ]

```

Code 0.40: RETI-Blocks Pass für den Anfangsteil

0.0.4.3 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinandereiung von Zugriffen auf **Pointer-elemente**, **Arrayelemente** oder **Structattribute** 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 **Abstract Syntax Tree** in Code 0.42 wird die Aneinandererihung von Zugriffen auf **Po-
interelemente**, **Arrayelemente** und **Structattribute** (*complex_var.ar)[2-2] durch die Kom-
position `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(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
8           ↪ Name('ar'))
9       ],
10    FunDef
11      VoidType 'void',
12      Name 'main',
13      [
14        Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
15          ↪ Array([Num('42')]))
16        Assign(Alloc(Writable(), StructSpec(Name('st')), Name('complex_var')),
17          ↪ Struct([Assign(Name('ar'), Ref(Name('var')))]))
18        Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
19          ↪ Sub('-', Num('2')))))
20      ]
21  ]

```

Code 0.42: Abstract Syntax Tree für den Mittelteil

Im **PicoC-Mon Pass** in Code 0.43 wird die Komposition `Exp(Subscr(Deref(Attr(Name('complex_var'),
Name('ar')), Num('0')), BinOp(Num('2'), Sub('-', Num('2')))))` durch die Kompositionen
`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 `Subscr(exp1, exp2)` wird dieser Container-Knoten einfach dem `exp` Attribut
des `Ref(exp)`-Container Knoten zugewiesen und die **Indexberechnung** für `exp2` davorgezogen (in die-
sem Fall dargestellt durch `Exp(Num('2'))` und `Exp(BinOp(Stack(Num('2')), Sub('-', Stack(Num('1'))))`)
und über `Stack(Num('1'))` auf das Ergebnis der **Indexberechnung** auf dem **Stack** zugegriffen:
`Exp(BinOp(Stack(Num('2')), Sub('-', Stack(Num('1'))))`.

```

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         Ref(Global(Num('1')))
14         Ref(Attr(Stack(Num('1')), Name('ar')))
15         Exp(Num('0'))
16         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
17         Exp(Num('2'))
18         Exp(Num('2'))
19         Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
20         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
21         Exp(Stack(Num('1')))
22         Return(Empty())
23     ]
24 ]

```

Code 0.43: PicoC-Mon Pass für den Mittelteil

Im **RETI-Blocks Pass** in Code 0.44 werden die Kompositionen `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 entsprechenden **RETI-Knoten** ersetzt. Bei der Generierung des **RETI-Code** muss auch das versteckte Attribut `datatype` im `Ref(exp, datatype)`-Container-Knoten berücksichtigt werden, was in Unterkapitel 0.0.4.1 zusammen mit der Abbildung 1 bereits erklärt wurde.

```

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     # Ref(Global(Num('1')))
27     SUBI SP 1;
28     LOADI IN1 1;
29     ADD IN1 DS;
30     STOREIN SP IN1 1;
31     # Ref(Attr(Stack(Num('1')), Name('ar')))
32     LOADIN SP IN1 1;
33     ADDI IN1 0;
34     STOREIN SP IN1 1;
35     # Exp(Num('0'))
36     SUBI SP 1;
37     LOADI ACC 0;
38     STOREIN SP ACC 1;
39     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
40     LOADIN SP IN2 2;
41     LOADIN IN2 IN1 0;
42     LOADIN SP IN2 1;
43     MULTI IN2 1;
44     ADD IN1 IN2;
45     ADDI SP 1;
46     STOREIN SP IN1 1;
47     # Exp(Num('2'))
48     SUBI SP 1;
49     LOADI ACC 2;
50     STOREIN SP ACC 1;
51     # Exp(Num('2'))
52     SUBI SP 1;
53     LOADI ACC 2;
54     STOREIN SP ACC 1;
55     # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
56     LOADIN SP ACC 2;
57     LOADIN SP IN2 1;
58     SUB ACC IN2;
59     STOREIN SP ACC 2;
60     ADDI SP 1;
61     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
62     LOADIN SP IN1 2;
63     LOADIN SP IN2 1;
64     MULTI IN2 1;
65     ADD IN1 IN2;
66     ADDI SP 1;
67     STOREIN SP IN1 1;
68     # Exp(Stack(Num('1')))
69     LOADIN SP IN1 1;
70     LOADIN IN1 ACC 0;
71     STOREIN SP ACC 1;
72     # Return(Empty())
73     LOADIN BAF PC -1;
74 ]
75 ]

```

Code 0.44: RETI-Blocks Pass für den Mittelteil

0.0.4.4 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 0.0.4.2 und **Mittelteil** 0.0.4.3 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird⁹, 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

Das Generieren des **Abstract Syntax Tree** in Code 0.46 verläuft wie üblich.

```

1 File
2   Name './example_derived_dts_final_part.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc(Writable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8       ],
9     FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14        Exp(Alloc(Writable(), ArrayDecl([Num('1')], Num('2')), IntType('int')),
15          ↪ Name('complex_var1'))
16        Exp(Alloc(Writable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
17          ↪ Name('complex_var2'))
18        Assign(Alloc(Writable(), IntType('int'), Name('var')), Num('42'))
19        Assign(Alloc(Writable(), PtrDecl(Num('1'), IntType('int')), Name('pntr1')),
20          ↪ Ref(Name('var')))
21        Assign(Alloc(Writable(), PtrDecl(Num('2'), IntType('int')), Name('complex_var3')),
22          ↪ Ref(Name('pntr1')))
23        Exp(Subscr(Name('complex_var1'), Num('0')))
24        Exp(Subscr(Name('complex_var2'), Num('0')))
25        Exp(Deref(Name('complex_var3'), Num('0')))
26      ]
27    ]
28  ]
```

⁹Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

Code 0.46: Abstract Syntax Tree für den Schlussteil

Im **PicoC-Mon Pass** in Code 0.47 wird das eben angesprochene auf den **Stack** speichern des **Inhalts** der berechneten **Adresse** mit der Komposition `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        Ref(Global(Num('0')))
17        Exp(Num('0'))
18        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
19        Exp(Stack(Num('1')))
20        Ref(Global(Num('2')))
21        Exp(Num('0'))
22        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23        Exp(Stack(Num('1')))
24        Ref(Global(Num('6')))
25        Exp(Num('0'))
26        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27        Exp(Stack(Num('1')))
28        Return(Empty())
29      ]
30    ]

```

Code 0.47: PicoC-Mon Pass für den Schlussteil

Im **RETI-Blocks Pass** in Code 0.48 wird die Komposition `Exp(Stack(Num('1')))` durch ihre entsprechenden **RETI-Knoten** ersetzt, wenn das versteckte Attribut `datatype` im `Exp(exp,datatype)`-Container-Knoten kein **Array** `ArrayDecl(nums, datatype)` enthält, ansonsten ist bei einem **Array** die **Adresse** auf dem **Stack** bereits das gewünschte Ergebnis. Genauer wurde in Unterkapitel 0.0.4.1 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     # Ref(Global(Num('0')))
37     SUBI SP 1;
38     LOADI IN1 0;
39     ADD IN1 DS;
40     STOREIN SP IN1 1;
41     # Exp(Num('0'))
42     SUBI SP 1;
43     LOADI ACC 0;
44     STOREIN SP ACC 1;
45     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
46     LOADIN SP IN1 2;
47     LOADIN SP IN2 1;
48     MULTI IN2 2;
49     ADD IN1 IN2;
50     ADDI SP 1;
51     STOREIN SP IN1 1;
52     # // not included Exp(Stack(Num('1')))
53     # Ref(Global(Num('2')))
54     SUBI SP 1;
55     LOADI IN1 2;
56     ADD IN1 DS;
57     STOREIN SP IN1 1;
58     # Exp(Num('0'))
59     SUBI SP 1;
60     LOADI ACC 0;
61     STOREIN SP ACC 1;
62     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63     LOADIN SP IN1 2;
64     LOADIN SP IN2 1;
```



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

Code 0.48: RETI-Blocks Pass für den Schlussteil

0.0.5 Umsetzung von Funktionen

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** kompiliert werden. Das Beispiel ist so gewählt, dass es möglichst **isoliert** von weiterem möglicherweise störendem Code ist.

```

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

```

Code 0.49: PicoC-Code für 3 Funktionen

Im **Abstract Syntax Tree** in Code 0.50 wird eine **Funktion**, wie z.B. `void fun(int param){ return param; }` mit der Komposition `FunDef(IntType(), Name('fun'), [Alloc(Writeable(), IntType()), Name('fun')])`, `[Return(Exp(Name('param')))]` dargestellt. Die einzelnen **Attribute** dieses Container-Knoten sind in Tabelle ?? erklärt.

```

1 File
2   Name './verbose_3_funs.ast',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Return
10          Empty
11      ],
12     FunDef
13       VoidType 'void',
14       Name 'fun1',
15       [],
16       [],
17     FunDef
18       IntType 'int',
19       Name 'fun2',
20       [],
21       [
22         Return
23           Num '1'
24       ]
25   ]

```

Code 0.50: Abstract Syntax Tree für 3 Funktionen

Im **PicoC-Blocks Pass** in Code 0.51 werden die **Statements** der Funktion in **Blöcke** `Block(name, stmts_instrs)` aufgeteilt. Dabei bekommt ein Block `Block(name, stmts_instrs)`, der die Statements der Funktion vom **Anfang** bis zum **Ende** oder bis zum Auftauchen eines `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` oder `DoWhile(exp, stmts)`¹⁰ beinhaltet den **Bezeichner** bzw. den `Name(str)`-Token-Knoten der Funktion an sein **Label** bzw. an sein `name`-Attribut zugewiesen. Dem **Bezeichner** wird vor der Zuweisung allerdings noch eine **Nummer** angehängt `<name>.<nummer>`¹¹.

Es werden parallel dazu neue Zuordnungen im **Dictionary** `fun_name_to_block_name` hinzugefügt. Das **Dictionary** ordnet einem **Funktionsnamen** den **Blocknamen** des Blockes, der das erste **Statement** der Funktion enthält und dessen **Bezeichner** `<name>.<nummer>` bis auf die angehängte **Nummer** identisch zu dem der Funktion ist zu¹². Diese Zuordnung ist nötig, da **Blöcke** noch eine **Nummer** an ihren Bezeichner `<name>.<nummer>` angehängt haben.

```

1 File
2   Name './verbose_3_funs.picoc_blocks',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
9         Block
10          Name 'main.2',
11          [
12            Return(Empty())
13          ]
14      ],
15     FunDef
16       VoidType 'void',
17       Name 'fun1',
18       [],
19       [
20         Block
21          Name 'fun1.1',
22          []
23      ],
24     FunDef
25       IntType 'int',
26       Name 'fun2',
27       [],
28       [
29         Block
30          Name 'fun2.0',
31          [
32            Return(Num('1'))
33          ]
34      ]
35   ]

```

¹⁰Eine Erklärung dazu ist in Unterkapitel ?? zu finden.

¹¹Der **Grund** dafür kann im Unterkapitel ?? nachgelesen werden.

¹²Das ist der **Block**, über den im **obigen letzten Paragraph** gesprochen wurde.

Code 0.51: PicoC-Blocks Pass für 3 Funktionen

Im **PicoC-Mon Pass** in Code 0.52 werden die `FunDef(datatype, name, allocs, stmts)`-Container-Knoten komplett aufgelöst, sodass sich im `File(name, decls_defs_blocks)`-Container-Knoten nur noch Blöcke befinden.

```

1 File
2   Name './verbose_3_funs.picoc_mon',
3   [
4     Block
5       Name 'main.2',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'fun1.1',
11      [
12        Return(Empty())
13      ],
14     Block
15      Name 'fun2.0',
16      [
17        // Return(Num('1'))
18        Exp(Num('1'))
19        Return(Stack(Num('1')))
20      ]
21   ]

```

Code 0.52: PicoC-Mon Pass für 3 Funktionen

Nach dem **RETI Pass** in Code 0.53 gibt es nur noch **RETI-Instructions**, die Blöcke wurden entfernt und die **RETI-Instructions** in diesen Blöcken wurden genauso zusammengefügt, wie die Blöcke angeordnet waren. Ohne die **Kommentare** könnte man die Funktionen nicht mehr direkt ausmachen, denn die **Kommentare** enthalten die **Labelbezeichner** `<name>.<nummer>` der Blöcke, die in diesem Beispiel immer zugleich bis auf die Nummer, dem **Namen** der jeweiligen **Funktion** entsprechen.

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

```

1 # // Block(Name('start.3'), [])
2 # // Exp(GoTo(Name('main.2'))))
3 # // not included Exp(GoTo(Name('main.2'))))
4 # // Block(Name('main.2'), [])
5 # Return(Empty())
6 LOADIN BAF PC -1;
7 # // Block(Name('fun1.1'), [])
8 # Return(Empty())

```

```

9 LOADIN BAF PC -1;
10 # // Block(Name('fun2.0'), [])
11 # // Return(Num('1'))
12 # Exp(Num('1'))
13 SUBI SP 1;
14 LOADI ACC 1;
15 STOREIN SP ACC 1;
16 # Return(Stack(Num('1'))))
17 LOADIN SP ACC 1;
18 ADDI SP 1;
19 LOADIN BAF PC -1;

```

Code 0.53: 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** des Programmes immer ganz vorne im **RETI-Code** beginnt. Man musste sich daher keine Gedanken darum machen, wie man die **Ausführung**, die von der `main`-Funktion ausgeht überhaupt startet.

Im Beispiel in Code 0.54 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 ist es, die main-Funktion einfach nach **vorne** zu schieben, damit diese als **erstes** ausgeführt wird. Im File(name, decls_defs)-Container-Knoten muss dazu im decls_defs-Attribut, welches eine **Liste von Funktionen** ist, die main-Funktion an Index 0 geschoben werden.

Eine andere Möglichkeit und die Möglichkeit für die sich in der **Implementierung** des **PicoC-Compilers** entschieden wurde, ist es, wenn die main-Funktion nicht die erste auftauchende Funktion ist, einen start.<number>-Block als ersten Block einzufügen, der einen GoTo(Name('main.<number>'))-Container-Knoten enthält, der im **RETI Pass 0.57** in einen Sprung zur main-Funktion übersetzt wird.

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

Das **Einsetzen** des start.<number>-Blockes erfolgt im **RETI-Patch Pass** in Code 0.56, da der **RETI-Patch**-Pass der Pass ist, der für das **Ausbessern** (engl. to patch) zuständig ist, wenn z.B. in manchen Fällen die main-Funktion nicht die erste Funktion ist.

```

1 File
2     Name './verbose_3_funs_main.reti_patch',
3     [
4         Block
5             Name 'start.3',
6             [
7                 # // Exp(GoTo(Name('main.0')))
8                 Exp(GoTo(Name('main.0')))
9             ],
10        Block
11            Name 'fun1.2',
12            [
13                # Return(Empty())
14                LOADIN BAF PC -1;

```

```

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

```

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

Im **RETI Pass** in Code 0.57 wird das `GoTo(Name('main.<nummer>'))` durch den entsprechenden Sprung `JUMP <distanz_zur_main_funktion>` ersetzt und die Blöcke entfernt.

```

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

```

Code 0.57: RETI Pass für Funktionen, wobei die main Funktion nicht die erste Funktion ist

0.0.5.2 Funktionsdeklaration und -definition und Umsetzung von Scopes

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 **Fehler-**

meldungen ausgegeben werden können, wenn der **Prototyp** (Definition 0.2) der Funktion nicht mit den **Datentypen** der **Argumente** oder der **Anzahl Argumente** übereinstimmt, die beim **Funktionsaufruf** an die Funktion in einer **festen** Reihenfolge übergeben werden.

Die Deklaration einer Funktion kann explizit erfolgen (z.B. `int fun2(int var);`), wie in der im Beispiel in Code 0.58 **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** zur Funktion `fun2`, die allerdings erst nach der `main`-Funktion definiert ist. Daher ist eine **Funktionsdeklaration**, wie in der **markierten Zeile** 1 notwendig. Beim **Funktionsaufruf** zur Funktion `fun1` ist das **nicht** notwendig, da die Funktion vorher **definiert** wurde, wie in den **markierten Zeilen** 3-4 zu sehen ist.

Definition 0.2: Funktionsprototyp

*Deklaration einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parameterreihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktionendeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.^{a,b}*

^aDer **Funktionsprototyp** ist von der **FunktionsSignatur** zu unterscheiden, die in Programmiersprache wie C++ und Java für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentypen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Scopes** und **Klassentypen** usw. unterscheiden.

^bWhat is the difference between function prototype and function signature?

```

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

Die Variablen `var@main` und `var@fun2` der `main`-Funktion und der Funktion `fun2` haben unterschiedliche **Scopes** (Definition 0.3). Die **Scopes** der **Funktionen** werden mittels eines **Suffix** `"@<fun_name>"` umgesetzt, der an den **Bezeichner** `var` drangehängt wird: `var@<fun_name>`. Dieser **Suffix** wird geändert sobald beim **Top-Down**¹³ Durchiterieren über den **Abstract Syntax Tree** des aktuellen **Passes** ein **Funktionswechsel** eintritt und

¹³D.h. von der **Wurzel** zu den **Blättern** eines Baumes.

über die Statements der nächsten Funktion iteriert wird, für die der **Suffix** der neuen Funktion `FunDef(name, datatype, params, stmts)` angehängt wird, der aus dem `name`-Attribut entnommen wird.

Ein Grund, warum **Scopes** 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 **Dictionary** umgesetzten **Symboltabelle** eindeutig sind. Damit man einer Variable direkt den **Scope** ablesen kann in dem sie definiert wurde, ist der **Suffix** ebenfalls im `Name(str)`-Token-Knoten des `name`-Attributtes eines **Symbols** der Symboltabelle angehängt. Zur besseren Vorstellung ist dies 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 um einen **Parameter** handelt, ist nur innerhalb des **Codeblocks** {} der Funktion `fun2` **sichtbar**. Das ist dadurch umgesetzt, dass der **Suffix**, der bei jedem **Funktionswechsel** angepasst wird, auch beim Nachschlagen eines **Symbols** in der **Symboltabelle** an den **Bezeichner** der Variablen, die man nachschlagen will angehängt wird. Und da die Zuordnungen im **Dictionary** **eindeutig** sind, kann eine Variable nur in genau der Funktion nachgeschlagen werden, in der sie **definiert** wurde.

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

Definition 0.3: Scope (bzw. Sichtbarkeitsbereich)

*Bereich in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.*^a

^aThiemann, „Einführung in die Programmierung“.

```

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

```

```

25     name:                Name('main')
26     value or address:    Empty()
27     position:            Pos(Num('6'), Num('5'))
28     size:                Empty()
29 },
30 Symbol
31 {
32     type qualifier:      Writeable()
33     datatype:            IntType('int')
34     name:                Name('var@main')
35     value or address:    Num('0')
36     position:            Pos(Num('7'), Num('6'))
37     size:                Num('1')
38 },
39 Symbol
40 {
41     type qualifier:      Writeable()
42     datatype:            IntType('int')
43     name:                Name('var@fun2')
44     value or address:    Num('0')
45     position:            Pos(Num('12'), Num('13'))
46     size:                Num('1')
47 }
48 ]

```

Code 0.59: Symboltabelle für Funktionen, wobei eine Funktion vorher deklariert werden muss

0.0.5.3 Funktionsaufruf

0.0.5.3.1 Ohne Rückgabewert

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 z.B. noch kein **Rückgabewert** zurückgegeben wird¹⁴.

```

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

```

Code 0.60: PicoC-Code für Funktionsaufruf ohne Rückgabewert

¹⁴Was allerdings im nächsten Unterkapitel 0.0.5.3.2 erklärt wird.

```

1 File
2   Name './example_fun_call_no_return_value.picoc_mon',
3   [
4     Block
5       Name 'main.1',
6       [
7         StackMalloc(Num('2'))
8         Ref(Global(Num('0')))
9         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
10        Exp(GoTo(Name('stack_fun.0')))
11        RemoveStackframe()
12        Return(Empty())
13      ],
14    Block
15      Name 'stack_fun.0',
16      [
17        Return(Empty())
18      ]
19  ]

```

Code 0.61: PicoC-Mon Pass für Funktionsaufruf ohne Rückgabewert

```

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

```

```
33     [  
34     # Return(Empty())  
35     LOADIN BAF PC -1;  
36     ]  
37 ]
```

Code 0.62: RETI-Blocks Pass für Funktionsaufruf ohne Rückgabewert

```
1 # StackMalloc(Num('2'))  
2 SUBI SP 2;  
3 # Ref(Global(Num('0')))  
4 SUBI SP 1;  
5 LOADI IN1 0;  
6 ADD IN1 DS;  
7 STOREIN SP IN1 1;  
8 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))  
9 MOVE BAF ACC;  
10 ADDI SP 3;  
11 MOVE SP BAF;  
12 SUBI SP 4;  
13 STOREIN BAF ACC 0;  
14 LOADI ACC 14;  
15 ADD ACC CS;  
16 STOREIN BAF ACC -1;  
17 JUMP 5;  
18 # RemoveStackframe()  
19 MOVE BAF IN1;  
20 LOADIN IN1 BAF 0;  
21 MOVE IN1 SP;  
22 # Return(Empty())  
23 LOADIN BAF PC -1;  
24 # Return(Empty())  
25 LOADIN BAF PC -1;
```

Code 0.63: RETI-Pass für Funktionsaufruf ohne Rückgabewert

0.0.5.3.2 Mit Rückgabewert

```
1 void stack_fun() {  
2     return 42;  
3 }  
4  
5 void main() {  
6     int var = stack_fun();  
7 }
```

Code 0.64: PicoC-Code für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './example_fun_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         // Return(Num('42'))
8         Exp(Num('42'))
9         Return(Stack(Num('1')))
10      ],
11     Block
12       Name 'main.0',
13       [
14         // Assign(Name('var'), Call(Name('stack_fun'), []))
15         StackMalloc(Num('2'))
16         NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
17         Exp(GoTo(Name('stack_fun.1')))
18         RemoveStackframe()
19         Assign(Global(Num('0')), Stack(Num('1')))
20         Return(Empty())
21      ]
22   ]

```

Code 0.65: PicoC-Mon Pass für Funktionsaufruf mit Rückgabewert

```

1 File
2   Name './example_fun_call_with_return_value.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # // Return(Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Return(Stack(Num('1')))
13        LOADIN SP ACC 1;
14        ADDI SP 1;
15        LOADIN BAF PC -1;
16      ],
17     Block
18       Name 'main.0',
19       [
20        # // Assign(Name('var'), Call(Name('stack_fun'), []))
21        # StackMalloc(Num('2'))
22        SUBI SP 2;
23        # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
24        MOVE BAF ACC;
25        ADDI SP 2;
26        MOVE SP BAF;
27        SUBI SP 2;
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.1'))))
33      # RemoveStackframe()
34      MOVE BAF IN1;
35      LOADIN IN1 BAF 0;
36      MOVE IN1 SP;
37      # Assign(Global(Num('0')), Stack(Num('1'))))
38      LOADIN SP ACC 1;
39      STOREIN DS ACC 0;
40      ADDI SP 1;
41      # Return(Empty())
42      LOADIN BAF PC -1;
43  ]
44 ]

```

Code 0.66: RETI-Blocks Pass für Funktionsaufruf mit Rückgabewert

```

1 JUMP 7;
2 # // Return(Num('42'))
3 # Exp(Num('42'))
4 SUBI SP 1;
5 LOADI ACC 42;
6 STOREIN SP ACC 1;
7 # Return(Stack(Num('1'))))
8 LOADIN SP ACC 1;
9 ADDI SP 1;
10 LOADIN BAF PC -1;
11 # // Assign(Name('var'), Call(Name('stack_fun'), []))
12 # StackMalloc(Num('2'))
13 SUBI SP 2;
14 # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr'))))
15 MOVE BAF ACC;
16 ADDI SP 2;
17 MOVE SP BAF;
18 SUBI SP 2;
19 STOREIN BAF ACC 0;
20 LOADI ACC 17;
21 ADD ACC CS;
22 STOREIN BAF ACC -1;
23 JUMP -15;
24 # RemoveStackframe()
25 MOVE BAF IN1;
26 LOADIN IN1 BAF 0;
27 MOVE IN1 SP;
28 # Assign(Global(Num('0')), Stack(Num('1'))))
29 LOADIN SP ACC 1;
30 STOREIN DS ACC 0;
31 ADDI SP 1;
32 # Return(Empty())
33 LOADIN BAF PC -1;

```

Code 0.67: RETI-Pass für Funktionsaufruf mit Rückgabewert

0.0.5.3.3 Umsetzung von Call by Sharing für Arrays

```

1 void stack_fun(int (*param1)[3], int param2[2][3]) {
2 }
3
4 void main() {
5     int local_var1[2][3];
6     int local_var2[2][3];
7     stack_fun(local_var1, local_var2);
8 }

```

Code 0.68: PicoC-Code für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'main.0',
11      [
12        StackMalloc(Num('2'))
13        Ref(Global(Num('0')))
14        Ref(Global(Num('6')))
15        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
16        Exp(GoTo(Name('stack_fun.1')))
17        RemoveStackframe()
18        Return(Empty())
19      ]
20   ]

```

Code 0.69: PicoC-Mon Pass für Call by Sharing für Arrays

```

1 SymbolTable
2   [
3     Symbol
4     {
5       type qualifier:      Empty()
6       datatype:            FunDecl(VoidType('void'), Name('stack_fun'),
7         ↪ [Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
8         ↪ Name('param1')), Alloc(Writable(), ArrayDecl([Num('3')], IntType('int')),
9         ↪ Name('param2'))])
10      name:                 Name('stack_fun')
11      value or address:     Empty()
12      position:             Pos(Num('1'), Num('5'))
13      size:                 Empty()
14    },
15  ]

```

```

12 Symbol
13 {
14     type qualifier:      Writeable()
15     datatype:           PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
16     name:               Name('param1@stack_fun')
17     value or address:   Num('0')
18     position:           Pos(Num('1'), Num('21'))
19     size:               Num('1')
20 },
21 Symbol
22 {
23     type qualifier:      Writeable()
24     datatype:           PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
25     name:               Name('param2@stack_fun')
26     value or address:   Num('1')
27     position:           Pos(Num('1'), Num('37'))
28     size:               Num('1')
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('4'), Num('5'))
37     size:               Empty()
38 },
39 Symbol
40 {
41     type qualifier:      Writeable()
42     datatype:           ArrayDecl([Num('2'), Num('3')], IntType('int'))
43     name:               Name('local_var1@main')
44     value or address:   Num('0')
45     position:           Pos(Num('5'), Num('6'))
46     size:               Num('6')
47 },
48 Symbol
49 {
50     type qualifier:      Writeable()
51     datatype:           ArrayDecl([Num('2'), Num('3')], IntType('int'))
52     name:               Name('local_var2@main')
53     value or address:   Num('6')
54     position:           Pos(Num('6'), Num('6'))
55     size:               Num('6')
56 }
57 ]

```

Code 0.70: Symboltabelle für Call by Sharing für Arrays

```

1 File
2   Name './example_fun_call_by_sharing_array.reti_blocks',
3   [
4     Block
5     Name 'stack_fun.1',

```



```

6      [
7      # Return(Empty())
8      LOADIN BAF PC -1;
9      ],
10     Block
11     Name 'main.0',
12     [
13     # StackMalloc(Num('2'))
14     SUBI SP 2;
15     # Ref(Global(Num('0')))
16     SUBI SP 1;
17     LOADI IN1 0;
18     ADD IN1 DS;
19     STOREIN SP IN1 1;
20     # Ref(Global(Num('6')))
21     SUBI SP 1;
22     LOADI IN1 6;
23     ADD IN1 DS;
24     STOREIN SP IN1 1;
25     # NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
26     MOVE BAF ACC;
27     ADDI SP 4;
28     MOVE SP BAF;
29     SUBI SP 4;
30     STOREIN BAF ACC 0;
31     LOADI ACC GoTo(Name('addr@next_instr'));
32     ADD ACC CS;
33     STOREIN BAF ACC -1;
34     Exp(GoTo(Name('stack_fun.1')))
35     # RemoveStackframe()
36     MOVE BAF IN1;
37     LOADIN IN1 BAF 0;
38     MOVE IN1 SP;
39     # Return(Empty())
40     LOADIN BAF PC -1;
41     ]
42 ]

```

Code 0.71: RETI-Block Pass für Call by Sharing für Arrays

0.0.5.3.4 Umsetzung von Call by Value für Structs

```

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

```

Code 0.72: PicoC-Code für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Return(Empty())
8       ],
9     Block
10      Name 'main.0',
11      [
12        StackMalloc(Num('2'))
13        Assign(Stack(Num('3')), Global(Num('0')))
14        NewStackframe(Name('stack_fun'), GoTo(Name('addr@next_instr')))
15        Exp(GoTo(Name('stack_fun.1')))
16        RemoveStackframe()
17        Return(Empty())
18      ]
19  ]

```

Code 0.73: PicoC-Mon Pass für Call by Value für Structs

```

1 File
2   Name './example_fun_call_by_value_struct.reti_blocks',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         # Return(Empty())
8         LOADIN BAF PC -1;
9       ],
10    Block
11      Name 'main.0',
12      [
13        # StackMalloc(Num('2'))
14        SUBI SP 2;
15        # Assign(Stack(Num('3')), Global(Num('0')))
16        SUBI SP 3;
17        LOADIN DS ACC 0;
18        STOREIN SP ACC 1;
19        LOADIN DS ACC 1;
20        STOREIN SP ACC 2;
21        LOADIN DS ACC 2;
22        STOREIN SP ACC 3;
23        # NewStackframe(Name('stack_fun'), 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('stack_fun.1')))  
33     # RemoveStackframe()  
34     MOVE BAF IN1;  
35     LOADIN IN1 BAF 0;  
36     MOVE IN1 SP;  
37     # Return(Empty())  
38     LOADIN BAF PC -1;  
39 ]  
40 ]
```

Code 0.74: RETI-Block Pass für Call by Value für Structs

0.1 Fehlermeldungen

0.1.1 Error Handler

0.1.2 Arten von Fehlermeldungen

0.1.2.1 Syntaxfehler

0.1.2.2 Laufzeitfehler

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).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: <https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/> (besucht am 18.07.2022).

Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09.07.2022).
- Thiemann, Peter. „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).