

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

*Abgabedatum:* 28<sup>th</sup> 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 Structs . . . . .	8
0.0.1.1	Deklaration und Definition von Structtypen . . . . .	8
0.0.1.2	Initialisierung von Structs . . . . .	10
0.0.1.3	Zugriff auf Structattribut . . . . .	13
0.0.1.4	Zuweisung an Structattribut . . . . .	17
0.0.2	Umsetzung des Zugriffs auf Derived locations im Allgemeinen . . . . .	19
0.0.2.1	Übersicht . . . . .	19
0.0.2.2	Anfangsteil . . . . .	22
0.0.2.3	Mittelteil . . . . .	25
0.0.2.4	Schluss teil . . . . .	28

---

---

# Abbildungsverzeichnis

1	Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes . . . . .	20
---	---------------------------------------------------------------------------	----

---

---

# Codeverzeichnis

0.1	PicoC-Code für die Deklaration eines Structtyps . . . . .	8
0.2	Abstract Syntax Tree für die Deklaration eines Structtyps . . . . .	8
0.3	Symboltabelle für die Deklaration eines Structtyps . . . . .	10
0.4	PicoC-Code für Initialisierung von Structs . . . . .	10
0.5	Abstract Syntax Tree für Initialisierung von Structs . . . . .	11
0.6	PicoC-Mon Pass für Initialisierung von Structs . . . . .	12
0.7	RETI-Blocks Pass für Initialisierung von Structs . . . . .	13
0.8	PicoC-Code für Zugriff auf Structattribut . . . . .	13
0.9	Abstract Syntax Tree für Zugriff auf Structattribut . . . . .	13
0.10	PicoC-Mon Pass für Zugriff auf Structattribut . . . . .	16
0.11	RETI-Blocks Pass für Zugriff auf Structattribut . . . . .	17
0.12	PicoC-Code für Zuweisung an Structattribut . . . . .	17
0.13	Abstract Syntax Tree für Zuweisung an Structattribut . . . . .	18
0.14	PicoC-Mon Pass für Zuweisung an Structattribut . . . . .	18
0.15	RETI-Blocks Pass für Zuweisung an Structattribut . . . . .	19
0.16	PicoC-Code für den Anfangsteil . . . . .	23
0.17	Abstract Syntax Tree für den Anfangsteil . . . . .	23
0.18	PicoC-Mon Pass für den Anfangsteil . . . . .	24
0.19	RETI-Blocks Pass für den Anfangsteil . . . . .	25
0.20	PicoC-Code für den Mittelteil . . . . .	25
0.21	Abstract Syntax Tree für den Mittelteil . . . . .	26
0.22	PicoC-Mon Pass für den Mittelteil . . . . .	27
0.23	RETI-Blocks Pass für den Mittelteil . . . . .	28
0.24	PicoC-Code für den Schlussteil . . . . .	29
0.25	Abstract Syntax Tree für den Schlussteil . . . . .	29
0.26	PicoC-Mon Pass für den Schlussteil . . . . .	30
0.27	RETI-Blocks Pass für den Schlussteil . . . . .	32

---

---

# Tabellenverzeichnis

---

---

# Definitionsverzeichnis

0.1	Location . . . . .	15
0.2	Entarteter Baum . . . . .	16

---

---

# Grammatikverzeichnis



## 0.0.1 Umsetzung von Structs

### 0.0.1.1 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 0.1 erläutert.

```

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

Code 0.1: PicoC-Code für die Deklaration eines Structtyps

Bevor irgendwas definiert werden kann, muss erstmal ein **Structtyp** deklariert werden. Im **Abstract Syntax Tree** in Code 0.3 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 0.2: Abstract Syntax Tree für die Deklaration eines Structtyps

Für den **Structtyp** selbst wird in der **Symboltabelle**, die in Code 0.3 dargestellt ist ein Eintrag mit dem **Schlüssel** `st` erstellt. Die Felder dieses Eintrags `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt, allerdings sind in dem `value_address`-Feld die Attribute des **Structtyps** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Structtyp** `st` die **Attribute** des Structtyps in der **Symboltabelle** nachschlagen kann. Die Schlüssel der **Attribute** haben einen **Suffix** `@st` angehängt, der eine Art **Scope** innerhalb des **Structtyps** für seine Attribut darstellt. Es gilt foglich, dass **innerhalb** eines **Structtyps**

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

Jedes der **Attribute** [Name('len@st'), Name('ar@st')] erhält auch einen eigenen Eintrag in der **Symboltabelle**, wobei die Felder `type_qualifier`, `datatype`, `name`, `value_address`, `position` und `size` wie üblich belegt werden. Die Felder `type_qualifier`, `datatype` und `name` werden z.B. bei Name('ar@st') mithilfe der Attribute von `Alloc(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`-Feld enthält dabei den Namen des **Structtyps** als Komposition `StructSpec(Name('st'))`, wodurch jederzeit alle wichtigen Informationen zu diesem **Structtyp**<sup>1</sup> und seinen **Attributen** in der **Symboltabelle** nachgeschlagen werden können.

Die **Größe** einer Variable `st_var`, die ihm `size`-Feld des **Symboltabelleneintrags** eingetragen ist und mit dem **Structtyp** `struct st {datatype1 attr1; ... datatypen attrn;}`<sup>a</sup> 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**:  $size(st) = \sum_{i=1}^n size(datatype_i)$ .

<sup>a</sup>Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache *L<sub>PicoC</sub>* nicht die fragwürdige Designentscheidung, auch die eckigen Klammern `[]` für die Definition eines Arrays **vor** die Variable zu schreiben von *L<sub>C</sub>* 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')

```

<sup>1</sup>Wie 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 0.3: Symboltabelle für die Deklaration eines Structtyps

### 0.0.1.2 Initialisierung von Structs

Die **Initialisierung eines Structs** wird im Folgenden mithilfe des Beispiels in Code 0.4 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 0.4: PicoC-Code für Initialisierung von Structs

Im **Abstract Syntax Tree** in Code 0.5 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 0.5: Abstract Syntax Tree für Initialisierung von Structs

Im **PicoC-Mon Pass** in Code 0.6 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 ?? daher wird um keine Wiederholung zu betreiben auf Subkapitel ?? verwiesen. Um das ganze interessanter zu gestalten wurde das Beispiel in Code 0.4 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 ?? 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 0.6: PicoC-Mon Pass für Initialisierung von Structs

Im **RETI-Blocks Pass** in Code 0.7 werden die **Kompositionen** `Exp(exp)`, `Ref(exp)` und `Assign(Global(Num('1')), Stack(Num('3')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```

1 File
2   Name './example_struct_init.reti_blocks',
3   [
4     Block
5       Name 'main.0',
6       [
7         # // Assign(Name('var'), Num('42'))
8         # Exp(Num('42'))
9         SUBI SP 1;
10        LOADI ACC 42;
11        STOREIN SP ACC 1;
12        # Assign(Global(Num('0')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN DS ACC 0;
15        ADDI SP 1;
16        # // Assign(Name('st'), Struct([Assign(Name('attr1'), Name('var')),
    ↪ Assign(Name('attr2'), Struct([Assign(Name('attr'),
    ↪ Array([Array([Ref(Name('var')), Ref(Name('var'))]))]))]))
17        # Exp(Global(Num('0')))
18        SUBI SP 1;
19        LOADIN DS ACC 0;
20        STOREIN SP ACC 1;
21        # Ref(Global(Num('0')))
22        SUBI SP 1;
23        LOADI IN1 0;
24        ADD IN1 DS;
25        STOREIN SP IN1 1;
26        # Ref(Global(Num('0')))
27        SUBI SP 1;
28        LOADI IN1 0;
29        ADD IN1 DS;
30        STOREIN SP IN1 1;
31        # Assign(Global(Num('1')), Stack(Num('3')))
32        LOADIN SP ACC 1;
33        STOREIN DS ACC 3;

```

```

34     LOADIN SP ACC 2;
35     STOREIN DS ACC 2;
36     LOADIN SP ACC 3;
37     STOREIN DS ACC 1;
38     ADDI SP 3;
39     # Return(Empty())
40     LOADIN BAF PC -1;
41 ]
42 ]

```

Code 0.7: RETI-Blocks Pass für Initialisierung von Structs

### 0.0.1.3 Zugriff auf Structattribut

Der **Zugriff auf ein Structattribut** (z.B. `st.y`) wird im Folgenden mithilfe des Beispiels in Code 0.8 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 0.8: PicoC-Code für Zugriff auf Structattribut

Im **Abstract Syntax Tree** in Code 0.9 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 0.9: Abstract Syntax Tree für Zugriff auf Structattribut

Im **PicoC-Mon Pass** in Code 0.10 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 ?? darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonstn auf das Subkapitel ?? verwiesen.

Die Komposition  $\text{Exp}(\text{Attr}(\text{Name}('st'), \text{Name}('y')))$  wird genauso, wie in Subkapitel ?? durch Kompositionen ersetzt, die sich in **Anfangsteil 0.0.2.2**, **Mittelteil 0.0.2.3** und **Schlusssteil 0.0.2.4** 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 ??.

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.

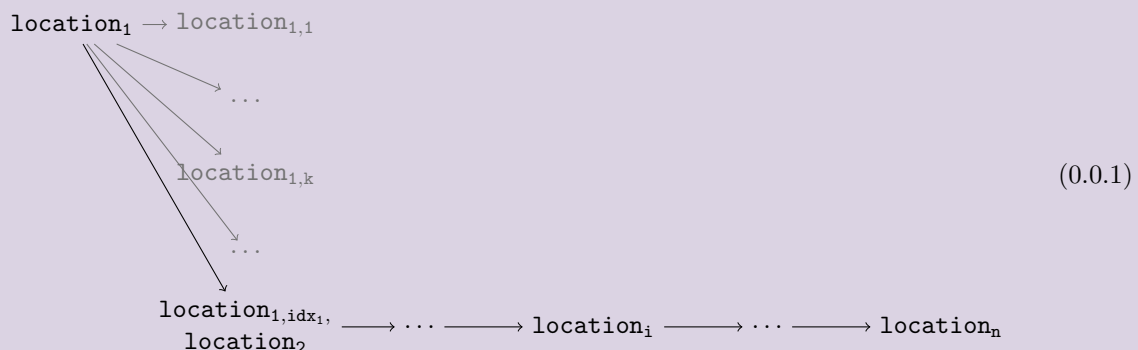
Im Gegensatz zur Komposition  $\text{Ref}(\text{Subscr}(\text{Stack}(\text{Num}('2')), \text{Stack}(\text{Num}('1'))))$  beim **Zugriff auf einen Arrayindex** in Subkapitel ??, muss hier vorher nichts anderes als die **Anfangsadresse** des **Structs** auf dem **Stack** liegen. Das **Structattribut** 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 Kompilervorgangs im **PiocC-Mon Pass** dem **Container-Knoten**  $\text{Ref}(\text{exp}, \text{datatype})$  angehängt.

Sei  $\text{location}_i$  eine **Location** (Definition 0.1) und ein **Knoten** eines entarteten Baumes (siehe Definition 0.2 und Abbildung 0.0.1), dessen Wurzel  $\text{location}_i$  ist. Dabei steht  $i$  für eine **Ebene** des entarteten Baumes. Die Knoten des entarteten Baumes lassen sich **Startadressen**  $\text{ref}(\text{location}_i)$  von Speicherbereichen  $\text{ref}(\text{location}_i) \dots \text{ref}(\text{location}_i) + \text{size}(\text{location}_i)$  im **Hauptspeicher** zuordnen, wobei gilt, dass  $\text{ref}(\text{location}_i) \leq \text{ref}(\text{location}_{i+1}) < \text{ref}(\text{location}_i) + \text{size}(\text{location}_i)$ .<sup>ab</sup>

Jeder **Location**  $\text{location}_i$  ist dabei ein **Dantentyp**  $\text{datatype}_i$  zugeordnet.

Sei  $\text{location}_{i,k}$  eine **Location** der ein beliebiges **Element / Attribut** des **Datentyps**  $\text{datatype}_i$  der **Location**  $\text{location}_i$  zugeordnet ist. Dabei gilt:  $\text{ref}(\text{location}_{i,k}) < \text{ref}(\text{location}_{i,k+1})$ .

Sei  $\text{location}_{i,\text{idx}_i}$  die **Location** eines beliebigen **Elementes / Attributes** des **Datentyps**  $\text{datatype}_i$  der **Location**  $\text{location}_i$ , sodass gilt:  $\text{location}_{i,\text{idx}_i} = \text{location}_{i+1}$ .



Die Berechnung der **Adresse** für eine beliebige Folge verschiedener Datentypen

(datatype<sub>1,idx<sub>1</sub></sub>, ..., datatype<sub>n,idx<sub>n</sub></sub>), die das Resultat einer Aneinandereiung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente** und **Structattribute** unterschiedlicher Datentypen datatype<sub>i</sub> ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 0.0.2:

$$\text{ref}(\text{datatype}_{1,\text{idx}_1}, \dots, \text{datatype}_{n,\text{idx}_n}) = \text{ref}(\text{location}_1) + \sum_{i=1}^{n-1} \sum_{k=1}^{\text{idx}_i-1} \text{size}(\text{datatype}_{i,k}) \quad (0.0.2)$$

berechnet werden.<sup>c</sup>

Dabei darf nur der letzte Knoten location<sub>n</sub> den Datentyp **Pointer** haben. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Pointer**, der nicht der **letzte Datentyp** location<sub>n</sub> in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp location<sub>1</sub> bis direkt zum Datentyp **Pointer** geht location<sub>pntnr</sub> und die **zweite Adressberechnung** einen Datentyp nach dem Datentyp **Pointer** anfängt datatype<sub>pntnr+1</sub> und bis zum letzten Datentyp location<sub>n</sub> geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse** ref(location<sub>1</sub>) des Summanden aus der Formel 0.0.2 auf den Inhalt der Speicherzelle an der gerade in der **zweiten Adressberechnung** berechneten Adresse M[reflocation<sub>1</sub>...location<sub>pntnr</sub>] gesetzt werden.

Die Formel 0.0.2 stellt dabei eine **Verallgemeinerung** der Formel ?? dar, die für alle möglichen Aneinandereiungen 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 ??, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben<sup>d</sup>.

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden ref(location<sub>1</sub>) in der Formel.

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

Die Komposition `Exp(Stack(Num('1')))` repräsentiert dabei das Lesen des **Inhalts** M[ref(location<sub>1,idx<sub>1</sub></sub>, ..., location<sub>n,idx<sub>n</sub></sub>)] der Speicherzelle an der finalen **Adresse** ref(location<sub>1,idx<sub>1</sub></sub>, ..., location<sub>n,idx<sub>n</sub></sub>).

<sup>a</sup>Es ist ein Baum, der **nur** die **Datentypen** als Knoten enthält, auf die **zugegriffen** wird.

<sup>b</sup>ref(location) steht dabei für das Schreiben der **Adresse** von location auf den Stack.

<sup>c</sup>Die **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Pointerelemente**, **Arrayelemente** oder **Structattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps** location<sub>i</sub>, die vor dem **Element** / **Attribut** location<sub>1,idx<sub>1</sub></sub> liegen.

<sup>d</sup>Structattribute haben **unterschiedliche** Größen.

### Definition 0.1: Location

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

*Im Grunde genommen alles, was mit einem **Programm zu tun** hat und irgendwo **gespeichert** ist.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.



**Definition 0.2: Entarteter Baum**

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

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

Der Baum entspricht also einer *verketteten Liste*.<sup>a</sup>

<sup>a</sup>Bäume.

```

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        // Exp(Attr(Name('st'), Name('y')))
13        Ref(Global(Num('0')))
14        Ref(Attr(Stack(Num('1')), Name('y')))
15        Exp(Stack(Num('1')))
16        Return(Empty())
17      ]
18    ]

```

Code 0.10: PicoC-Mon Pass für Zugriff auf Structattribut

Im **RETI-Blocks Pass** in Code 0.11 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')))

```

```

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

```

Code 0.11: RETI-Blocks Pass für Zugriff auf Structattribut

#### 0.0.1.4 Zuweisung an Structattribut

Die **Zuweisung an ein Structattribut** (z.B. `st.y = 42`) wird im Folgenden anhand des Beispiels in Code 0.12 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 0.12: 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(Writeable(), 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 0.13: Abstract Syntax Tree für Zuweisung an Structattribut

Im **PicoC-Mon Pass** in Code 0.14 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 ?? darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel ?? verwiesen.

Im Gegensatz zum Vorgehen in Unterkapitel ?? 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 0.0.1.3 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        Return(Empty())
18      ]
19    ]

```

Code 0.14: PicoC-Mon Pass für Zuweisung an Structattribut

Im **RETI-Blocks Pass** in Code 0.15 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.O',
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      # // Assign(Attr(Name('st'), Name('y')), Num('42'))
24      # Exp(Num('42'))
25      SUBI SP 1;
26      LOADI ACC 42;
27      STOREIN SP ACC 1;
28      # Ref(Global(Num('0')))
29      SUBI SP 1;
30      LOADI IN1 0;
31      ADD IN1 DS;
32      STOREIN SP IN1 1;
33      # Ref(Attr(Stack(Num('1')), Name('y')))
34      LOADIN SP IN1 1;
35      ADDI IN1 1;
36      STOREIN SP IN1 1;
37      # Assign(Stack(Num('1')), Stack(Num('2')))
38      LOADIN SP IN1 1;
39      LOADIN SP ACC 2;
40      ADDI SP 2;
41      STOREIN IN1 ACC 0;
42      # Return(Empty())
43      LOADIN BAF PC -1;
44    ]
45  ]

```

Code 0.15: RETI-Blocks Pass für Zuweisung an Structattribut

## 0.0.2 Umsetzung des Zugriffs auf Derived locations im Allgemeinen

### 0.0.2.1 Übersicht

In den Unterkapiteln ??, ?? und 0.0.1 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 **Schluss**teil darin erkennen.

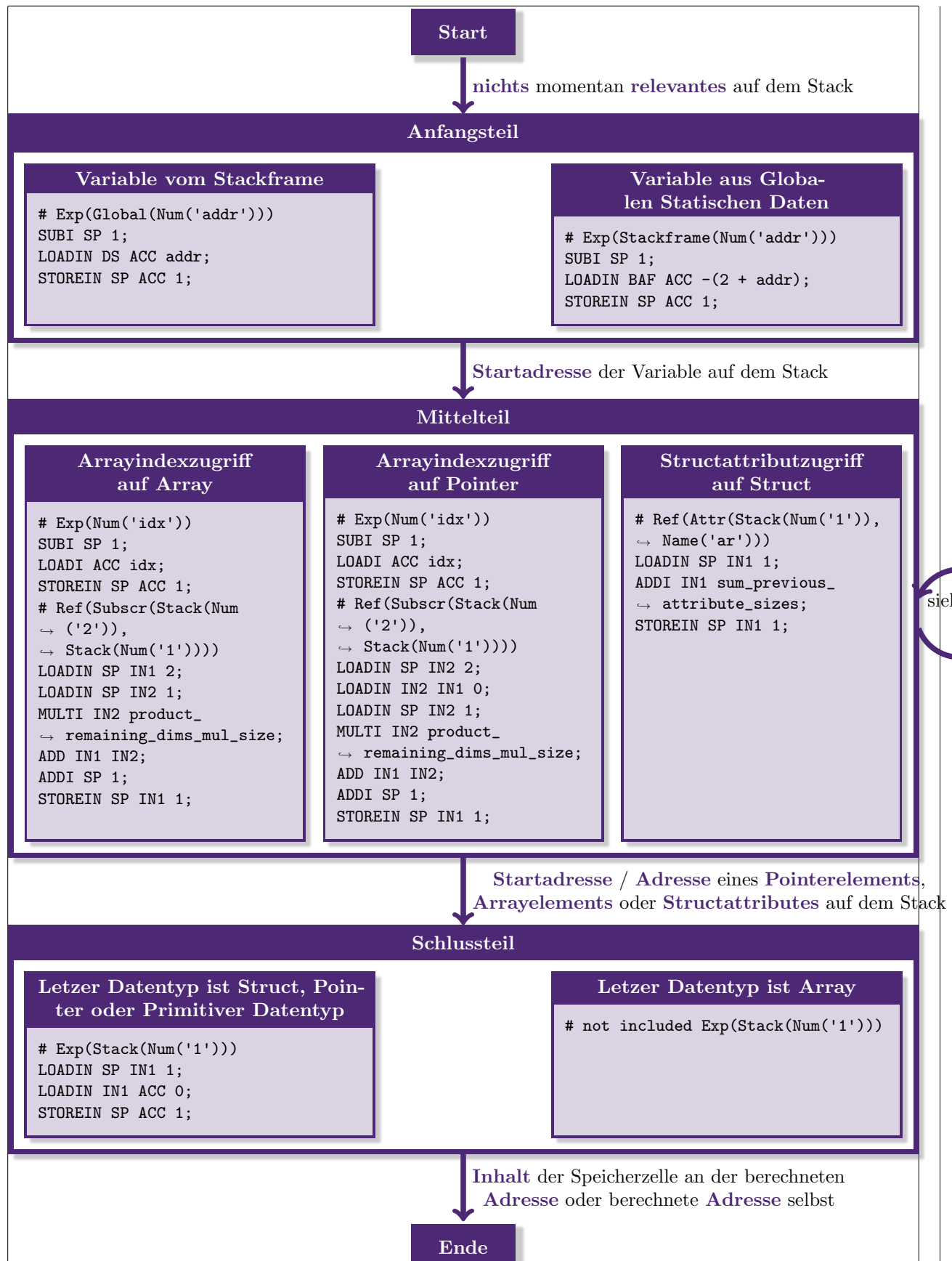


Abbildung 1: Allgemeine Veranschaulichung des Zugriffs auf Derived Datatypes

Dieses Vorgehen ist in Abbildung 1 veranschaulicht. Dieses Vorgehen erlaubt es auch gemischte Ausdrücke zu schreiben, in denen die verschiedenen **Zugriffsarten** für **Elemente** / **Attribute** der 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 **Zugriffsoption** `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 **Zugriffsoption** muss dabei zum **Datentyp** im versteckten Attribut `datatype` passen, ansonsten gibt 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 ?? 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 wird<sup>3</sup>, die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache *L<sub>PicoC</sub>* implementierten Datentypen wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

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

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

Parallel wird eine Liste der `Ref(exp)`-Knoten geführt, deren **versteckte Attribute** `datatype` und `error_data` die entsprechenden Informationen zugewiesen bekommen müssen. Sobald man beim `Name(name)`-Knoten angekommen ist und mithilfe dieses in der **Symboltabelle** den **Datentyp** 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.<sup>a</sup>

<sup>a</sup>Man kann diese Implementierung gut mit dem **Auseinanderrollen** und **Wieder-Einrollen** einer **Spirale** vergleichen.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil 0.0.2.2**, **Mittelteil 0.0.2.3** und **Schlusssteil 0.0.2.4** bei der Kompilierung von **Zugriffen** auf **Pointerelemente**, **Arrayelemente**, **Structattribute** bei gemischten Ausdrücken, wie `(*st.first.ar)[0]`; einzeln isoliert betrachtet und erläutert.

### 0.0.2.2 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 0.16 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;
```

<sup>3</sup>Und nicht auf ein **Element** des Arrays.

<sup>3</sup>**Startadresse** / **Adresse** eines **Pointerelements**, **Arrayelements** oder **Structattributes** auf dem Stack.

```

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

```

Code 0.16: PicoC-Code für den Anfangsteil

Im **Abstract Syntax Tree** in Code 0.17 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')], IntType('int'))),
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 0.17: Abstract Syntax Tree für den Anfangsteil



Im **PicoC-Mon Pass** in Code 0.18 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         // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
8         ↪ Name('st_ar'))
9         // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], PtrDecl(Num('1'),
10        ↪ IntType('int'))), Name('complex_var'))
11        // Exp(Ref(Name('complex_var'))
12        Ref(Global(Num('9')))
13        Return(Empty())
14      ],
15    Block
16      Name 'fun.0',
17      [
18        // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
19        ↪ Name('st_ar'))
20        // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int'))),
21        ↪ Name('complex_var'))
22        // Exp(Ref(Name('complex_var'))
23        Ref(Stackframe(Num('9')))
24        Return(Empty())
25      ]
26    ]
27  ]

```

Code 0.18: PicoC-Mon Pass für den Anfangsteil

Im **RETI-Blocks Pass** in Code 0.19 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         # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
8         ↪ Name('st_ar'))
9         # // Exp(Alloc(Writable(), PtrDecl(Num('1'), ArrayDecl([Num('3')],
10        ↪ PtrDecl(Num('1'), IntType('int'))), Name('complex_var'))
11        # // Exp(Ref(Name('complex_var'))
12        # Ref(Global(Num('9')))
13        SUBI SP 1;
14        LOADI IN1 9;
15        ADD IN1 DS;
16        STOREIN SP IN1 1;
17        # Return(Empty())

```

```

16     LOADIN BAF PC -1;
17 ],
18 Block
19   Name 'fun.0',
20   [
21     # // Exp(Alloc(Writable(), ArrayDecl([Num('3')], StructSpec(Name('ar_with_len'))),
22     ↪ Name('st_ar')))
23     # // Exp(Alloc(Writable(), PntrDecl(Num('1'), ArrayDecl([Num('3')],
24     ↪ IntType('int'))), Name('complex_var')))
25     # // Exp(Ref(Name('complex_var')))
26     # Ref(Stackframe(Num('9')))
27     SUBI SP 1;
28     MOVE BAF IN1;
29     SUBI IN1 11;
30     STOREIN SP IN1 1;
31     # Return(Empty())
32     LOADIN BAF PC -1;
33   ]
34 ]

```

Code 0.19: RETI-Blocks Pass für den Anfangsteil

### 0.0.2.3 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinanderreihung 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 0.20 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 0.20: PicoC-Code für den Mittelteil

Im **Abstract Syntax Tree** in Code 0.21 wird die Aneinanderreihung von Zugriffen auf **Pointer-elemente**, **Arrayelemente** und **Structattribute** `(*complex_var.ar)[2-2]` durch die Komposition `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(), PntrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
8         ↪ Name('ar'))

```

```

8      ],
9      FunDef
10      VoidType 'void',
11      Name 'main',
12      [],
13      [
14          Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
15              ↪ Array([Num('42')]))
16          Assign(Alloc(Writeable(), 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 0.21: Abstract Syntax Tree für den Mittelteil

Im **PicoC-Mon Pass** in Code 0.22 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 diesem 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        // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
14            ↪ BinOp(Num('2'), Sub('-', Num('2'))))
15        Ref(Global(Num('1')))
16        Ref(Attr(Stack(Num('1')), Name('ar')))
17        Exp(Num('0'))
18        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
19        Exp(Num('2'))
20        Exp(Num('2'))
21        Exp(BinOp(Stack(Num('2')), Sub('-', Stack(Num('1'))))
22        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
23        Exp(Stack(Num('1')))
24        Return(Empty())
25    ]
26 ]

```

## Code 0.22: PicoC-Mon Pass für den Mittelteil

Im **RETI-Blocks Pass** in Code 0.23 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 0.0.2.1 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        # // Exp(Subscr(Subscr(Attr(Name('complex_var'), Name('ar')), Num('0')),
27        ↪      BinOp(Num('2'), Sub('-'), Num('2'))))
28        # Ref(Global(Num('1')))
29        SUBI SP 1;
30        LOADI IN1 1;
31        ADD IN1 DS;
32        STOREIN SP IN1 1;
33        # Ref(Attr(Stack(Num('1')), Name('ar')))
34        LOADIN SP IN1 1;
35        ADDI IN1 0;
36        STOREIN SP IN1 1;
37        # Exp(Num('0'))
38        SUBI SP 1;
39        LOADI ACC 0;
40        STOREIN SP ACC 1;
41        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
42        LOADIN SP IN2 2;
43        LOADIN IN2 IN1 0;
44        LOADIN SP IN2 1;
45        MULTI IN2 1;

```

```

45     ADD IN1 IN2;
46     ADDI SP 1;
47     STOREIN SP IN1 1;
48     # Exp(Num('2'))
49     SUBI SP 1;
50     LOADI ACC 2;
51     STOREIN SP ACC 1;
52     # Exp(Num('2'))
53     SUBI SP 1;
54     LOADI ACC 2;
55     STOREIN SP ACC 1;
56     # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
57     LOADIN SP ACC 2;
58     LOADIN SP IN2 1;
59     SUB ACC IN2;
60     STOREIN SP ACC 2;
61     ADDI SP 1;
62     # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
63     LOADIN SP IN1 2;
64     LOADIN SP IN2 1;
65     MULTI IN2 1;
66     ADD IN1 IN2;
67     ADDI SP 1;
68     STOREIN SP IN1 1;
69     # Exp(Stack(Num('1'))))
70     LOADIN SP IN1 1;
71     LOADIN IN1 ACC 0;
72     STOREIN SP ACC 1;
73     # Return(Empty())
74     LOADIN BAF PC -1;
75 ]
76 ]

```

Code 0.23: RETI-Blocks Pass für den Mittelteil

#### 0.0.2.4 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 0.0.2.2 und **Mittelteil** 0.0.2.3 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird<sup>4</sup>, wird im Folgenden mithilfe des Beispiels in Code 0.24 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;

```

<sup>4</sup>Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

```
13 }
```

### Code 0.24: PicoC-Code für den Schlussteil

Das Generieren des **Abstract Syntax Tree** in Code 0.25 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('ptr1')),
20          ↪ Ref(Name('var')))
21        Assign(Alloc(Writable(), PtrDecl(Num('2'), IntType('int')), Name('complex_var3')),
22          ↪ Ref(Name('ptr1')))
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  ]
```

### Code 0.25: Abstract Syntax Tree für den Schlussteil

Im **PicoC-Mon Pass** in Code 0.26 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         // Exp(Alloc(Writable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
8         ↪ Name('complex_var1')))
9         // Exp(Alloc(Writable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
10        ↪ Name('complex_var2')))
11        // Assign(Name('var'), Num('42'))
12        Exp(Num('42'))
13      ]
14    ]
15  ]
```

```

11     Assign(Global(Num('4')), Stack(Num('1')))
12     // Assign(Name('pntr1'), Ref(Name('var')))
13     Ref(Global(Num('4')))
14     Assign(Global(Num('5')), Stack(Num('1')))
15     // Assign(Name('complex_var3'), Ref(Name('pntr1')))
16     Ref(Global(Num('5')))
17     Assign(Global(Num('6')), Stack(Num('1')))
18     // Exp(Subscr(Name('complex_var1'), Num('0')))
19     Ref(Global(Num('0')))
20     Exp(Num('0'))
21     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22     Exp(Stack(Num('1')))
23     // Exp(Subscr(Name('complex_var2'), Num('0')))
24     Ref(Global(Num('2')))
25     Exp(Num('0'))
26     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
27     Exp(Stack(Num('1')))
28     // Exp(Subscr(Name('complex_var3'), Num('0')))
29     Ref(Global(Num('6')))
30     Exp(Num('0'))
31     Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
32     Exp(Stack(Num('1')))
33     Return(Empty())
34 ]
35 ]

```

Code 0.26: PicoC-Mon Pass für den Schlussteil

Im **RETI-Blocks Pass** in Code 0.27 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 0.0.2.1 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         # // Exp(Alloc(Writable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
8         ↪   Name('complex_var1')))
9         # // Exp(Alloc(Writable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
10        ↪   Name('complex_var2')))
11        # // Assign(Name('var'), Num('42'))
12        # Exp(Num('42'))
13        SUBI SP 1;
14        LOADI ACC 42;
15        STOREIN SP ACC 1;
16        # Assign(Global(Num('4')), Stack(Num('1')))
17        LOADIN SP ACC 1;
18        STOREIN DS ACC 4;
19        ADDI SP 1;
20        # // Assign(Name('pntr1'), Ref(Name('var')))

```

```

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

```



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

Code 0.27: RETI-Blocks Pass für den Schlussteil

---

---

# Literatur

## Online

- *Bäume*. URL: <https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/> (besucht am 17.07.2022).

## Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).