

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

---

*Abgabedatum:* 13. September 2022

*Autor:*

Jürgen Mattheis

*Gutachter:*

Prof. Dr. Scholl

*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.

# Danksagungen

Bevor der Inhalt dieser Schriftlichen Ausarbeitung der Bachelorarbeit anfängt, will ich einigen Personen noch meinen Dank aussprechen.

Ich schreibe die folgenden Danksagungen nicht auf eine bestimmte Weise, wie es sich vielleicht etabliert haben sollte Danksagungen zu schreiben und verwende auch keine künstlichen Floskeln, wie „mein aufrichtigster Dank“ oder „aus tiefstem Herzen“, sondern drücke im Folgenden die Dinge nur so aus, wie ich sie auch wirklich meine.

Estmal, ich hatte selten im Studium das Gefühl irgendwo Kunde zu sein, aber bei dieser Bachelorarbeit und dem vorangegangenen Bachelorprojekt hatte ich genau diese Gefühl, obwohl die Verhältnisse eigentlich genau umgekehrt sein sollten. Die Umgang mit mir wahr echt unglaublich nett und unbürokratisch, was ich als keine Selbstverständlichkeit ansehe und sehr wertgeschätzt habe.

An erster Stelle will ich zu meinem Betreuer M.Sc. Tobias Seufert kommen, der netterweise auch bereits die Betreuung meines Bachelorprojektes übernommen hatte. Wie auch während des Bachelorprojektes, haben wir uns auch bei den Meetings während der Bachelorarbeit hervorragend verstanden. Dabei ging die Freundlichkeit und das Engagement seitens Tobias weit über das hinaus, was man bereits als eine gute Betreuung bezeichnen würde.

Es gibt verschiedene Typen von Menschen, es gibt Leute, die nur genauso viel tun, wie es die Anforderungen verlangen und nichts darüberhinaus tun, wenn es nicht einen eigenen Vorteil für sie hat und es gibt Personen, die sich für nichts zu Schade sind und dies aus einer Philanthropie oder Leidenschaft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias<sup>1</sup> konnte ich während der langen Zeit, die er mein Bachelorprojekt und dann meine Bachelorarbeit betreut hat eindeutig als letzteren Typ Mensch einordnen.

Er war sich nie zu Schade für meine vielen Fragen während der Meetings, auch wenn ich meine Zeit ziemlich oft überzogen habe<sup>2</sup>, er hat sich bei der Korrektur dieser Schriftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere, wirklich hilfreiche Textkommentare zu verfassen und obendrauf auch noch Tippfehler usw. angemerkt und war sich nicht zu Schade die Rolle des Nachrichtenübermittlers zwischen mir und Prof. Dr. Scholl zu übernehmen. All dies war absolut keine Selbstverständlichkeit, vor allem wenn ich die Betreuung anderer Studenten, die ich kenne mit der vergleiche, die mir zu Teil wurde.

An den Kommentar zu meinem Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Wofür ich meinem Gutachter Prof. Dr. Scholl sehr dankbar bin, ist, dass er meine damals sehr ambitionierten Ideen für mögliche Funktionalitäten, die ich in den PicoC-Compiler für die Bachelorarbeit implementierten wollte runtergeschraubt hat. Man erlebt es äußerst selten im Studium, dass Studenten freiwillig weniger Arbeit gegeben wird.

Bei den für die Bachelorarbeit zu implementierenden Funktionalitäten gab es bei der Implementierung viele unerwartete kleine Details, die ich vorher garnicht bedacht hatte, die in ihrer Masse unerwartet viel Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan jedoch ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dageben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

---

<sup>1</sup>Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

<sup>2</sup>Wofür ich mich auch nochmal Entschuldigen will.

seinem eigenen Nachteil<sup>3 4</sup> weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholl im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, kann ich schon manchmal ziemlich eigensinnigen sein, bei der Weise, wie ich bestimmte Dinge umsetzen will. Ich habe es sehr geschätzt, dass mir das durchgehen gelassen wurde. Es ist, wie ich die Universitätswelt als Student erlebe bei Arbeitsvorgaben keine Selbstverständlichkeit, dass dem Studenten überhaupt die Freiheit und das Vertrauen gegeben wird diese auf seine eigenen Weise umzusetzen.

Vor allem, da mein eigenes Vorgehen größtenteils Vorteile für mich hatte, da ich auf diese Weise am meisten über Compilerbau gelernt hab und eher Nachteile für Prof. Dr. Scholl, da mein eigenes Vorgehen entsprechend mehr Zeit brauchte und ich daher als Bachelorarbeit keinen dazu passenden RETI-Emulator mit Graphischer Anzeige implementieren konnte, da die restlichen Funktionalitäten des PicoC-Compilers noch implementiert werden mussten.

Glücklicherweise gibt es aber doch noch einen passenden RETI-Emulator, der den PicoC-Compiler über seine Kommandozeilenargumente aufruft, um ein PicoC-Programm visuell auf einer RETI-CPU auszuführen. Für dessen Implementierung hat sich Michel Giehl netterweise zur Verfügung gestellt. Daher Danke auch an Michel Giehl, dass er sich mit meinem PicoC-Compiler auseinandergesetzt hat und diesen in seinen RETI-Emulator integriert hat, sodass am Ende durch unsere beiden Arbeiten ein anschauliches Lerntool für die kommenden Studentengenerationen entstehen konnte. Vor allem da er auch mir darin vertrauen musste, dass ich mit meinem PicoC-Compiler nicht irgendeinen Mist baue. Der RETI-Emulator von Michel Giehl ist unter [Link](#)<sup>5</sup> zu finden.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein persönliches Interessengebiet fällt<sup>6</sup>. Das Aufschreiben dieser Schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht<sup>7</sup>. Wobei ich allerdings sagen muss, dass ich eine große Erleichterung verspüre das ganze Wissen über Compilerbau mal aufgeschrieben zu haben, damit ich mir keine Sorgen machen muss dieses ziemlich nützliche Wissen irgendwann wieder zu vergessen. Es hilft einem auch als Programmierer ungemein weiter zu wissen, wie ein Compiler unter der Haube funktioniert, da man sich so viel besser merken, wie eine bestimmte Funktionalität einer Programmiersprache zu verwenden ist. Manch eine Funktionalität einer Programmiersprache kann in der Verwendung ziemlich willkürlich erscheinen, wenn man die technische Umsetzung dahinter im Compiler nicht kennt.

Ich wollte mich daher auch noch dafür Bedanken, dass mir ein so ergiebiges und interessantes Thema als Bachelorarbeit vorgeschlagen wurde und vor allem, dass auch das Vertrauen in mich gesteckt wurde, dass ich am Ende auch einen funktionsfähigen, sauber programmierten und gut durchdachten Compiler implementiere.

Zum Schluss nochmal ein abschließendes Danke an meinen Betreuer M.Sc Seufert und meinen Gutachter Prof. Dr. Scholl für die Betreuung und Bereitstellung dieser interessanten Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

---

<sup>3</sup>Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

<sup>4</sup>Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes 🤔.

<sup>5</sup><https://github.com/michel-giehl/Reti-Emulator>.

<sup>6</sup>Womit nicht alle Studenten so viel Glück haben.

<sup>7</sup>Dieses ständige überlegen, wo man möglicherweise eine Erklärücke hat, ob man nicht was wichtiges ausgelassen hat usw.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>I</b>
<b>Codeverzeichnis</b>	<b>II</b>
<b>Tabellenverzeichnis</b>	<b>III</b>
<b>Definitionsverzeichnis</b>	<b>IV</b>
<b>Grammatikverzeichnis</b>	<b>V</b>
0.0.1 Umsetzung von Verbunden . . . . .	1
0.0.1.1 Deklaration von Verbundtypen und Definition von Verbunden . . . . .	1
0.0.1.2 Initialisierung von Verbunden . . . . .	3
0.0.1.3 Zugriff auf Verbundsattribut . . . . .	6
0.0.1.4 Zuweisung an Verbundsattribut . . . . .	10
0.0.2 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen . . . . .	12
0.0.2.1 Anfangsteil . . . . .	15
0.0.2.2 Mittelteil . . . . .	18
0.0.2.3 Schlussteil . . . . .	21
<b>Literatur</b>	<b>A</b>

# Abbildungsverzeichnis

1	Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen. . . . .	13
---	--	----

# Codeverzeichnis

0.1	PicoC-Code für die Deklaration eines Verbundstyps. . . . .	1
0.2	Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps. . . . .	1
0.3	Symboltabelle für die Deklaration eines Verbundstyps. . . . .	3
0.4	PicoC-Code für Initialisierung von Verbunden. . . . .	3
0.5	Abstrakter Syntaxbaum für Initialisierung von Verbunden. . . . .	4
0.6	PicoC-ANF Pass für Initialisierung von Verbunden. . . . .	5
0.7	RETI-Blocks Pass für Initialisierung von Verbunden. . . . .	6
0.8	PicoC-Code für Zugriff auf Verbundsattribut. . . . .	6
0.9	Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut. . . . .	7
0.10	PicoC-ANF Pass für Zugriff auf Verbundsattribut. . . . .	9
0.11	RETI-Blocks Pass für Zugriff auf Verbundsattribut. . . . .	10
0.12	PicoC-Code für Zuweisung an Verbundsattribut. . . . .	10
0.13	Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut. . . . .	11
0.14	PicoC-ANF Pass für Zuweisung an Verbundsattribut. . . . .	11
0.15	RETI-Blocks Pass für Zuweisung an Verbundsattribut. . . . .	12
0.16	PicoC-Code für den Anfangsteil. . . . .	16
0.17	Abstrakter Syntaxbaum für den Anfangsteil. . . . .	17
0.18	PicoC-ANF Pass für den Anfangsteil. . . . .	17
0.19	RETI-Blocks Pass für den Anfangsteil. . . . .	18
0.20	PicoC-Code für den Mittelteil. . . . .	18
0.21	Abstrakter Syntaxbaum für den Mittelteil. . . . .	19
0.22	PicoC-ANF Pass für den Mittelteil. . . . .	19
0.23	RETI-Blocks Pass für den Mittelteil. . . . .	21
0.24	PicoC-Code für den Schlussteil. . . . .	21
0.25	Abstrakter Syntaxbaum für den Schlussteil. . . . .	22
0.26	PicoC-ANF Pass für den Schlussteil. . . . .	23
0.27	RETI-Blocks Pass für den Schlussteil. . . . .	25

# Tabellenverzeichnis



# Definitionsverzeichnis

0.1 Entarteter Baum . . . . . 9

# Grammatikverzeichnis

## 0.0.1 Umsetzung von Verbunden

Bei Verbunden wird in diesem Unterkapitel zunächst geklärt, wie die **Deklaration von Verbundstypen** umgesetzt ist. Ist ein **Verbundstyp** deklariert, kann damit einhergehend ein **Verbund** mit diesem Verbundstyp **definiert** werden. Die Umsetzung von beidem wird in Unterkapitel 0.0.1.1 erläutert. Des Weiteren ist die Umsetzung der **Initalisierung eines Verbundes** 0.0.1.2, des **Zugriffs auf ein Verbundsattribut** 0.0.1.3 und der **Zuweisung an ein Verbundsattribut** 0.0.1.4 zu klären.

### 0.0.1.1 Deklaration von Verbundstypen und Definition von Verbunden

Die Umsetzung der **Deklaration** (Definition ??) eines neuen **Verbundstyps** (z.B. `struct st {int len; int ar[2];}`) und der **Definition** (Definition ??) eines **Verbundes** mit diesem **Verbundstyp** (z.B. `struct st st_var;`) wird im Folgenden anhand des Beispiels in Code 0.1 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 Verbundstyps.

Bevor ein Verbund definiert werden kann, muss erstmal ein **Verbundstyp** deklariert werden. Im **Abstrakten Syntaxbaum** in Code 0.3 wird die **Deklaration eines Verbundstyps** `struct st {int len; int ar[2];}` durch die Knoten `StructDecl(Name('st'), [Alloc(Writeable(), IntType('int'), Name('len')) Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('ar'))])` dargestellt.

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

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

**Code 0.2:** Abstrakter Syntaxbaum für die Deklaration eines Verbundstyps.

Für den **Verbundstyp** selbst und seine **Verbundsattribute** werden in der **Symboltabelle**, die in Code 0.3 dargestellt ist Symboltabelleneinträge mit den **Schlüsseln** `st`, `len@st` und `ar@st` erstellt. Die Schlüssel der

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

Die Attribute<sup>8</sup> der Symboltabelleneinträge für die **Verbundsattribute** sind genauso belegt wie bei **üblichen Variablen**. Die Attribute des Symboltabelleneintrags für den **Verbundstyp** `type_qualifier`, `datatype`, `name`, `position` und `size` sind wie üblich belegt. In dem `value_address`-Attribut des Symboltabelleneintrags für den **Verbundstyp** sind die **Verbundsattribute** `[Name('len@st'), Name('ar@st')]` aufgelistet, sodass man über den **Verbundstyp** `st` als Schlüssel die **Verbundsattribute** des Verbundstyps in der **Symboltabelle** nachschlagen kann.

Für die **Definition** einer Variable `st_var@main` mit diesem **Verbundstyp** `st` wird ein **Symboltabelleneintrag** in der **Symboltabelle** angelegt. Das `datatype`-Attribut dieses **Symboltabelleneintrags** enthält dabei den Namen des **Verbundstyps** als `StructSpec(Name('st'))`. Dadurch können jederzeit alle wichtigen Informationen zu diesem **Verbundstyp**<sup>9</sup> und seinen **Verbundsattributen** in der **Symboltabelle** nachgeschlagen werden.

#### Anmerkung

Die **Anzahl Speicherzellen** die eine Variable `st_var` belegt<sup>a</sup>, die mit dem **Verbundstyp** `struct st {datatype1 attr1; ... datatypen attrn;}b definiert ist (struct st st_var;), berechnet sich aus der Summe der Anzahl Speicherzellen, welche die einzelnen Datentypen datatype1 ... datatypen der Verbundsattribute attr1, ... attrn des Verbundstyps belegen:  $size(st) = \sum_{i=1}^n size(datatype_i)$ .c`

<sup>a</sup>Die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.

<sup>b</sup>Hier wird es der Einfachheit halber so dargestellt, als hätte die Programmiersprache *L<sub>PicoC</sub>* **nicht** die manchmal etwas unpraktische Designentscheidung, auch die eckigen Klammern `[]` bei der Definition eines Feldes **hinter** die Variable zu schreiben von *L<sub>C</sub>* übernommen. Es wird so getan, als würde der komplette **Datentyp** immer **vor** der Variable stehen: `datatype var`.

<sup>c</sup>Die Funktion `size` berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.

```

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'))

```

<sup>8</sup>Die über einen **Bezeichner** selektierbaren Elemente eines **Symboltabelleneintrags** und eines **Verbunds** heißen bei beiden **Attribute**.

<sup>9</sup>Wie z.B. vor allem die **Größe** bzw. **Anzahl an Speicherzellen**, die dieser **Verbundstyp** einnimmt.

```

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'))])
25     name:             Name('st')
26     value or address:  [Name('len@st'), Name('ar@st')]
27     position:         Pos(Num('1'), Num('7'))
28     size:             Num('3')
29   },
30   Symbol
31   {
32     type qualifier:    Empty()
33     datatype:          FunDecl(VoidType('void'), Name('main'), [])
34     name:             Name('main')
35     value or address:  Empty()
36     position:         Pos(Num('3'), Num('5'))
37     size:             Empty()
38   },
39   Symbol
40   {
41     type qualifier:    Writable()
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 Verbundstyps.

### 0.0.1.2 Initialisierung von Verbunden

Die Umsetzung der **Initialisierung eines Verbundes** 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 Verbunden.

Im **Abstrakten Syntaxbaum** in Code 0.5 wird die **Initialisierung eines Verbundes** `struct st1 st = {.attr1=var, .attr2={.attr={&var, &var}}}` mithilfe der **Knoten** `Assign(Alloc(Writable(),`

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'))),
8         ↪ Name('attr'))
9       ],
10    StructDecl
11      Name 'st2',
12      [
13        Alloc(Writeable(), IntType('int'), Name('attr1'))
14        Alloc(Writeable(), StructSpec(Name('st1')), Name('attr2'))
15      ],
16    FunDef
17      VoidType 'void',
18      Name 'main',
19      [],
20      [
21        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
22        Assign(Alloc(Writeable(), StructSpec(Name('st2')), Name('st')),
23        ↪ Struct([Assign(Name('attr1'), Name('var')), Assign(Name('attr2'),
24        ↪ Struct([Assign(Name('attr'), Array([Ref(Name('var')), Ref(Name('var'))]))]))]))
25      ]
26    ]
27  ]

```

**Code 0.5:** Abstrakter Syntaxbaum für Initialisierung von Verbunden.

Im **PicoC-ANF Pass** in Code 0.6 wird `Assign(Alloc(Writeable(), StructSpec(Name('st1')), Name('st')), Struct(...))` auf fast dieselbe Weise ausgewertet, wie bei der **Initialisierung eines Feldes** in Unterkapitel ?? . Aus diesem Grund wird an dieser Stelle auf Unterkapitel ?? 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 **Verbund-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 **Verbund-Initializer** Knoten anfängt, wird auf dieselbe Weise nach dem Prinzip der **Tiefensuche** von **links-nach-rechts** ausgewertet, wie es bei der **Initialisierung eines Feldes** in Unterkapitel ?? bereits erklärt wurde.

Beim **Iterieren** über den **Teilbaum**, muss beim **Verbund-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 **Feldes** oder **Verbundes** 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 **Verbund-Initializer**, einem **Feld-Initializer** oder einem **Logischen Ausdruck** zu tun. Bei **Feld-** und **Verbund-Initialisier** wird einfach über diese nach dem Prinzip der **Tiefensuche** 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')),
11        ↪ Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var'))),
12        ↪ Ref(Name('var'))]))]))))
13        Exp(Global(Num('0')))
14        Ref(Global(Num('0')))
15        Ref(Global(Num('0')))
16        Assign(Global(Num('1')), Stack(Num('3')))
17        Return(Empty())
18      ]
19    ]

```

**Code 0.6:** *PicoC-ANF Pass für Initialisierung von Verbunden.*

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')),
17        ↪ Assign(Name('attr2'), Struct([Assign(Name('attr'), Array([Ref(Name('var'))),
18        ↪ Ref(Name('var'))]))]))))
19        # Exp(Global(Num('0')))
20        SUBI SP 1;
21        LOADIN DS ACC 0;
22        STOREIN SP ACC 1;
23        # Ref(Global(Num('0')))
24        SUBI SP 1;
25        LOADI IN1 0;
26        ADD IN1 DS;
27        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 Verbunden.*

### 0.0.1.3 Zugriff auf Verbundsattribut

Der **Zugriff auf ein Verbundsattribut** (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 Verbundsattribut.*

Im **Abstrakten Syntaxbaum** in Code 0.9 wird der **Zugriff auf ein Verbundsattribut** `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 ]

```

**Code 0.9:** Abstrakter Syntaxbaum für Zugriff auf Verbundsattribut.

Im **PicoC-ANF Pass** in Code 0.10 wird die Komposition `Exp(Attr(Name('st'), Name('y')))` auf ähnliche Weise ausgewertet, wie die Komposition, die einen **Zugriff auf ein Feldelement** `Exp(Subscr(Name('ar'), Num('0')))` in Unterkapitel ?? darstellt. Daher wird hier, um Wiederholung zu vermeiden nur auf wichtige Aspekte hingewiesen und ansonsten auf das Unterkapitel ?? verwiesen.

Die Komposition `Exp(Attr(Name('st'), Name('y')))` wird genauso, wie in Unterkapitel ?? durch Kompositionen ersetzt, die sich in **Anfangsteil 0.0.2.1**, **Mittelteil 0.0.2.2** und **Schlusssteil 0.0.2.3** aufteilen lassen. In diesem Fall sind es `Ref(Global(Num('0')))` (**Anfangsteil**), `Ref(Attr(Stack(Num('1')), Name('y')))` (**Mittelteil**) und `Exp(Stack(Num('1')))` (**Schlusssteil**). Der **Anfangsteil** und **Schlusssteil** sind genau gleich, wie in Unterkapitel ??.

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

Im Gegensatz zur Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` beim **Zugriff auf einen Feldindex** in Unterkapitel ??, muss hier vorher nichts anderes als die **Anfangsadresse** des **Verbundes** auf dem **Stack** liegen. Das **Verbundsattribut** auf welches zugegriffen wird steht bereits in der Komposition `Ref(Attr(Stack(Num('1')), Name('y')))`, nämlich `Name('y')`. Den **Verbundstyp**, dem dieses Attribut gehört, kann man aus dem versteckten Attribut `datatype` herauslesen. Das versteckte Attribut wird während des Kompiliervorgangs im **Piocc-ANF Pass** dem Knoten `Ref(exp, datatype)` angehängt.

#### Anmerkung 🔍

Sei  $\text{datatype}_i$  ein **Knoten** eines **entarteten Baumes** (siehe Definition 0.1 und Abbildung 0.0.1), dessen Wurzel  $\text{datatype}_1$  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  $\text{ref}(\text{datatype}_i) \leq \text{ref}(\text{datatype}_{i+1}) < \text{ref}(\text{datatype}_i) + \text{size}(\text{datatype}_i)$ .<sup>ab</sup>

Sei  $\text{datatype}_{i,k}$  ein beliebiges **Element** / **Attribut** des **Datentyps**  $\text{datatype}_i$ . 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**  $\text{datatype}_i$ , 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 **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** unterschiedlicher Datentypen  $\text{datatype}_i$  ist (z.B. `*complex_var.attr3[2]`), kann mittels der Formel 0.0.2:

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

berechnet werden.<sup>c d</sup>

Dabei darf nur der letzte Knoten  $\text{datatype}_n$  vom Datentyp **Zeiger** sein. Ist in einer Folge von **Datentypen** ein Knoten vom Datentyp **Zeiger**, der nicht der **letzte Datentyp**  $\text{datatype}_n$  in der Folge ist, so muss die **Adressberechnung** in 2 Adressberechnungen aufgeteilt werden, wobei die **erste Adressberechnung** vom ersten Datentyp  $\text{datatype}_1$  bis direkt zum Datentyp Zeiger geht  $\text{datatype}_{\text{pntr}}$  und die **zweite Adressberechnung** einen Datentyp nach dem Datentyp Zeiger anfängt  $\text{datatype}_{\text{pntr}+1}$  und bis zum letzten Datentyp  $\text{datatype}_n$  geht. Bei der **zweiten Adressberechnung** muss dabei die **Adresse**  $\text{ref}(\text{datatype}_1)$  des Summanden aus der Formel 0.0.2 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 0.0.2 stellt dabei eine **Verallgemeinerung** der Formel ?? dar, die für alle möglichen Aneinandereiungen von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** funktioniert (z.B. `(*complex_var.attr2)[3]`). Da die Formel **allgemein** sein muss, lässt sie sich nicht so elegant mit einem Produkt  $\prod$  schreiben, wie die Formel ??, da man nicht davon ausgehen kann, dass alle Elemente den gleichen Datentyp haben<sup>e</sup>.

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**  $\text{ref}(\text{datatype}_{1,\text{idx}_1}, \dots, \text{datatype}_{n,\text{idx}_n})$ .

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

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

<sup>c</sup>Die Funktion **size** berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.

<sup>d</sup>Die **äußere Schleife** iteriert nacheinander über die Folge von Datentypen, die aus den **Zugriffen** auf **Zeigerelmente**, **Feldelemente** oder **Verbundsattribute** resultiert. Die **innere Schleife** iteriert über alle **Elemente** oder **Attribute** des momentan betrachteten **Datentyps**  $\text{datatype}_i$ , die vor dem **Element** / **Attribut**  $\text{datatype}_{i,\text{idx}_i}$  liegen.

<sup>e</sup>Verbundsattribute 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 bei dem jeder Knoten des Baumes **maximal** eine eingehende Kante hat, also maximal **Innengrad**<sup>a</sup> 1.*

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

<sup>a</sup>Der **Innengrad** ist die Anzahl **eingehener Kanten**.

<sup>b</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      ]
17    ]

```

**Code 0.10:** *PicoC-ANF Pass für Zugriff auf Verbundsattribut.*

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'))

```

```

9      SUBI SP 1;
10     LOADI ACC 4;
11     STOREIN SP ACC 1;
12     # Exp(Num('2'))
13     SUBI SP 1;
14     LOADI ACC 2;
15     STOREIN SP ACC 1;
16     # Assign(Global(Num('0')), Stack(Num('2')))
17     LOADIN SP ACC 1;
18     STOREIN DS ACC 1;
19     LOADIN SP ACC 2;
20     STOREIN DS ACC 0;
21     ADDI SP 2;
22     # // Exp(Attr(Name('st'), Name('y')))
23     # Ref(Global(Num('0')))
24     SUBI SP 1;
25     LOADI IN1 0;
26     ADD IN1 DS;
27     STOREIN SP IN1 1;
28     # Ref(Attr(Stack(Num('1')), Name('y')))
29     LOADIN SP IN1 1;
30     ADDI IN1 1;
31     STOREIN SP IN1 1;
32     # Exp(Stack(Num('1')))
33     LOADIN SP IN1 1;
34     LOADIN IN1 ACC 0;
35     STOREIN SP ACC 1;
36     # Return(Empty())
37     LOADIN BAF PC -1;
38 ]
39 ]

```

**Code 0.11:** *RETI-Blocks Pass für Zugriff auf Verbundsattribut.*

#### 0.0.1.4 Zuweisung an Verbundsattribut

Die **Zuweisung an ein Verbundsattribut** (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 Verbundsattribut.*

Im **Abstract Syntax Tree** wird eine **Zuweisung an ein Verbundsattribut** (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 0.13:** Abstrakter Syntaxbaum für Zuweisung an Verbundsattribut.

Im **PicoC-ANF 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 Feldelement** `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` in Unterkapitel ?? 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** Knoten `Attr(Name('st'), Name('y'))` von `Assign(Attr(Name('st'), Name('y')), Num('42'))` wie in Unterkapitel 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-ANF Pass für Zuweisung an Verbundsattribut.

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.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        # // 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 Verbandsattribut.*

## 0.0.2 Umsetzung des Zugriffs auf Zusammengesetzte Datentypen im Allgemeinen

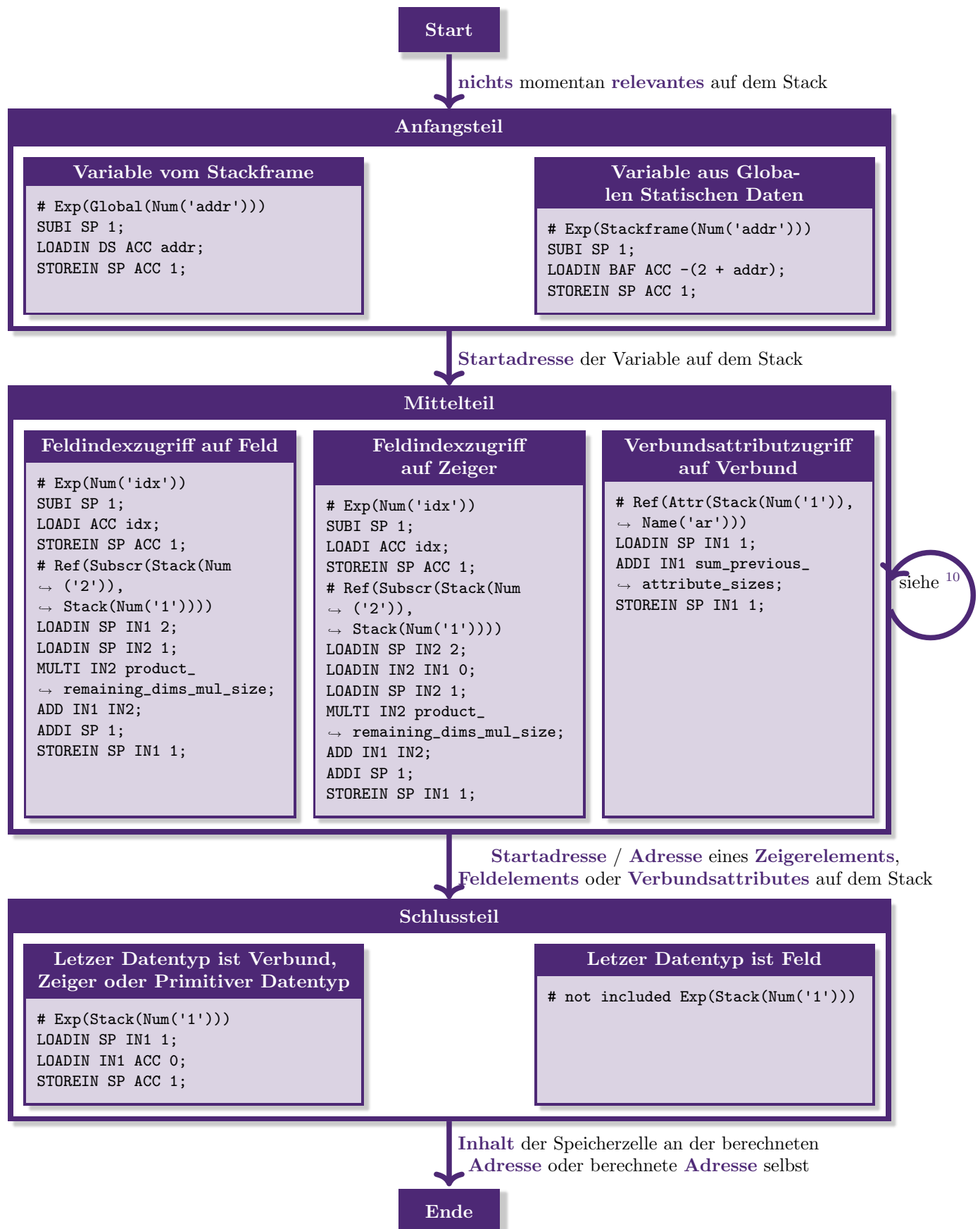


Abbildung 1: Allgemeine Veranschaulichung des Zugriffs auf Zusammengesetzte Datentypen.

In den Unterkapiteln ??, ?? und 0.0.1 fällt auf, dass der **Zugriff** auf **Elemente** / **Attribute** der in diesen Kapiteln beschriebenen Datentypen (**Zeiger**, **Feld** und **Verbund**) 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 **Zeiger**, **Feld** und **Verbund** 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)`-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 **Zeigerelements**, **Feldelements** oder **Verbundsattributs**.

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 Feldindex** `Subscr(exp1, exp2)` kann dabei mit den Datentypen **Feld** `ArrayDecl(nums, datatype)` und **Zeiger** `PntrDecl(num, datatype)` kombiniert werden. Allerdings benötigen beide Kombinationen unterschiedlichen **RETI-Code**. Das liegt daran, dass bei einem **Zeiger** `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 Verbundsattribut** `Attr(exp, name)` kann nur mit dem Datentyp **Struct** `StructSpec(name)` kombiniert werden.

#### Anmerkung 🔍

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 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 Feldelement** jeweils gegenseitig austauschbar sind. Der **Zugriff auf einen Feldindex** steht also gleichermaßen auch für eine **Dereferenzierung**.

Das versteckte Attribut `datatype` beinhaltet den **Unterdatentyp**, in welchem der Zugriff auf ein **Zeigerelement**, **Feldelement** oder **Verbundsattribut** 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 **Knoten** in diesem **Unterdatentyp** interessiert und die möglicherweise unter diesem momentan betrachteten **Knoten** liegenden Knoten in einem anderen `Ref(exp, versteckte Attribut)`-Knoten, dem jeweiligen versteckten Attribut `datatype` 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.

**Felder** haben in der Sprache  $L_C$  und somit auch in  $L_{PicoC}$  die Eigenheit, dass wenn auf ein gesamtes **Feld** zugegriffen wird<sup>11</sup>, die **Adresse** des ersten Elements ausgegeben wird und nicht der **Inhalt** der Speicherzelle des ersten Elements. Bei allen anderen in der Sprache  $L_{PicoC}$  implementierten Datentypen wird immer der **Inhalt** der Speicherzelle ausgegeben, die an der **Adresse** zu finden ist, die auf dem **Stack** liegt.

#### Anmerkung

**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')))`-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 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.

Im Folgenden werden anhand mehrerer Beispiele die einzelnen Abschnitte **Anfangsteil** 0.0.2.1, **Mittelteil** 0.0.2.2 und **Schlusssteil** 0.0.2.3 bei der Kompilierung von **Zugriffen** auf **Zeigerelemente**, **Feldelemente**, **Verbundsattribute** bei gemischten Ausdrücken, wie `(*st.first.ar)[0]`; einzeln isoliert betrachtet und erläutert.

#### 0.0.2.1 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() {
```

<sup>11</sup>Und nicht auf ein **Element** des Feldes.

<sup>11</sup>Startadresse / Adresse eines **Zeigerelements**, **Feldelements** oder **Verbundsattributes** auf dem Stack.

```

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

Im **Abstrakten Syntaxbaum** 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:** *Abstrakter Syntaxbaum für den Anfangsteil.*

Im **PicoC-ANF 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(Ref(Name('complex_var')))
8         Ref(Global(Num('9')))
9         Return(Empty())
10      ],
11     Block
12       Name 'fun.0',
13       [
14         // Exp(Ref(Name('complex_var')))
15         Ref(Stackframe(Num('9')))
16         Return(Empty())
17      ]
18   ]

```

**Code 0.18:** *PicoC-ANF 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(Ref(Name('complex_var')))
8         # Ref(Global(Num('9')))
9         SUBI SP 1;
10        LOADI IN1 9;
11        ADD IN1 DS;
12        STOREIN SP IN1 1;
13        # Return(Empty())
14        LOADIN BAF PC -1;
15      ],
16     Block
17       Name 'fun.0',
18       [
19         # // Exp(Ref(Name('complex_var')))
20         # Ref(Stackframe(Num('9')))
21        SUBI SP 1;

```

```

22     MOVE BAF IN1;
23     SUBI IN1 11;
24     STOREIN SP IN1 1;
25     # Return(Empty())
26     LOADIN BAF PC -1;
27 ]
28 ]

```

Code 0.19: *RETI-Blocks Pass für den Anfangsteil.*

### 0.0.2.2 Mittelteil

Der **Mittelteil**, bei dem die **Startadresse** / **Adresse** einer Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute** berechnet wird (z.B. `(*complex_var.ar)[2-2]`), wird im Folgenden mithilfe des Beispiels in Code 0.20 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 **Abstrakten Syntaxbaum** in Code 0.21 wird die Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** und **Verbundsattribute** `(*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(), PtrDecl(Num('1'), ArrayDecl([Num('1')], IntType('int'))),
9           ↪ Name('ar'))
8       ],
9     FunDef
10       VoidType 'void',
11       Name 'main',
12       [],
13       [
14         Assign(Alloc(Writable(), ArrayDecl([Num('1')], IntType('int')), Name('var')),
16           ↪ Array([Num('42')]))
15         Assign(Alloc(Writable(), StructSpec(Name('st')), Name('complex_var')),
17           ↪ Struct([Assign(Name('ar'), Ref(Name('var')))]))
16         Exp(Subscr(Deref(Attr(Name('complex_var'), Name('ar')), Num('0')), BinOp(Num('2'),
17           ↪ Sub('-', Num('2')))))

```

```

17     ]
18 ]

```

**Code 0.21:** *Abstrakter Syntaxbaum für den Mittelteil.*

Im **PicoC-ANF 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 Knoten einfach dem `exp` Attribut des `Ref(exp)`-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    ]
27  ]

```

**Code 0.22:** *PicoC-ANF 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)`-Knoten berücksichtigt werden, was in Unterkapitel 0.0.2 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.O',
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;
46        ADD IN1 IN2;
47        ADDI SP 1;
48        STOREIN SP IN1 1;
49        # Exp(Num('2'))
50        SUBI SP 1;
51        LOADI ACC 2;
52        STOREIN SP ACC 1;
53        # Exp(Num('2'))
54        SUBI SP 1;
55        LOADI ACC 2;
56        STOREIN SP ACC 1;
57        # 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.*

### 0.0.2.3 Schlussteil

Der **Schlussteil**, bei dem der **Inhalt** der Speicherzelle an der **Adresse**, die im **Anfangsteil** 0.0.2.1 und **Mittelteil** 0.0.2.2 auf dem **Stack** berechnet wurde, auf den **Stack** gespeichert wird<sup>12</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;
13 }

```

**Code 0.24:** *PicoC-Code für den Schlussteil.*

Das Generieren des **Abstrakten Syntaxbaumes** in Code 0.25 verläuft wie üblich.

```

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

```

<sup>12</sup>Und dabei die Speicherzelle der **Adresse** selbst überschreibt.

```

3  [
4    StructDecl
5      Name 'st',
6      [
7        Alloc(Writeable(), ArrayDecl([Num('2')], IntType('int')), Name('attr'))
8      ],
9    FunDef
10     VoidType 'void',
11     Name 'main',
12     [],
13     [
14       Exp(Alloc(Writeable(), ArrayDecl([Num('1'), Num('2')], IntType('int')),
15         ↪ Name('complex_var1')))
16       Exp(Alloc(Writeable(), ArrayDecl([Num('1')], StructSpec(Name('st'))),
17         ↪ Name('complex_var2')))
18       Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
19       Assign(Alloc(Writeable(), PtrDecl(Num('1'), IntType('int')), Name('pntr1')),
20         ↪ Ref(Name('var')))
21       Assign(Alloc(Writeable(), 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   ]

```

**Code 0.25:** Abstrakter Syntaxbaum für den Schlussteil.

Im **PicoC-ANF 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         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('4')), Stack(Num('1')))
10        // Assign(Name('pntr1'), Ref(Name('var')))
11        Ref(Global(Num('4')))
12        Assign(Global(Num('5')), Stack(Num('1')))
13        // Assign(Name('complex_var3'), Ref(Name('pntr1')))
14        Ref(Global(Num('5')))
15        Assign(Global(Num('6')), Stack(Num('1')))
16        // Exp(Subscr(Name('complex_var1'), Num('0')))
17        Ref(Global(Num('0')))
18        Exp(Num('0'))
19        Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
20        Exp(Stack(Num('1')))
21        // Exp(Subscr(Name('complex_var2'), Num('0')))
22        Ref(Global(Num('2')))
23        Exp(Num('0'))

```



```

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

```

**Code 0.26:** *PicoC-ANF 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)`-Knoten kein **Feld** `ArrayDecl(nums, datatype)` enthält, ansonsten ist bei einem **Feld** die **Adresse** auf dem **Stack** bereits das gewünschte Ergebnis. Genauer wurde in Unterkapitel 0.0.2 zusammen mit der Abbildung 1 bereits erklärt.

```

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

```

```

36      # // Exp(Subscr(Name('complex_var1'), Num('0')))
37      # Ref(Global(Num('0')))
38      SUBI SP 1;
39      LOADI IN1 0;
40      ADD IN1 DS;
41      STOREIN SP IN1 1;
42      # Exp(Num('0'))
43      SUBI SP 1;
44      LOADI ACC 0;
45      STOREIN SP ACC 1;
46      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
47      LOADIN SP IN1 2;
48      LOADIN SP IN2 1;
49      MULTI IN2 2;
50      ADD IN1 IN2;
51      ADDI SP 1;
52      STOREIN SP IN1 1;
53      # // not included Exp(Stack(Num('1')))
54      # // Exp(Subscr(Name('complex_var2'), Num('0')))
55      # Ref(Global(Num('2')))
56      SUBI SP 1;
57      LOADI IN1 2;
58      ADD IN1 DS;
59      STOREIN SP IN1 1;
60      # Exp(Num('0'))
61      SUBI SP 1;
62      LOADI ACC 0;
63      STOREIN SP ACC 1;
64      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
65      LOADIN SP IN1 2;
66      LOADIN SP IN2 1;
67      MULTI IN2 2;
68      ADD IN1 IN2;
69      ADDI SP 1;
70      STOREIN SP IN1 1;
71      # Exp(Stack(Num('1')))
72      LOADIN SP IN1 1;
73      LOADIN IN1 ACC 0;
74      STOREIN SP ACC 1;
75      # // Exp(Subscr(Name('complex_var3'), Num('0')))
76      # Ref(Global(Num('6')))
77      SUBI SP 1;
78      LOADI IN1 6;
79      ADD IN1 DS;
80      STOREIN SP IN1 1;
81      # Exp(Num('0'))
82      SUBI SP 1;
83      LOADI ACC 0;
84      STOREIN SP ACC 1;
85      # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
86      LOADIN SP IN2 2;
87      LOADIN IN2 IN1 0;
88      LOADIN SP IN2 1;
89      MULTI IN2 1;
90      ADD IN1 IN2;
91      ADDI SP 1;
92      STOREIN SP IN1 1;

```

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

**Code 0.27:** *RETI-Blocks Pass für den Schlussteil.*

# Literatur

## Online

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