Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 13. September 2022

Autor:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

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

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

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

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

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

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

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

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

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

seinem eigenen Nachteil[3] [4] weniger Arbeit aufgebrummt hat empfand ich als ich eine äußerst nette Geste, die ich sehr geschätzt habe.

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

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

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

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

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

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

[3] Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.
[4] Vielleicht finde ich ja noch im nächsten Semester während des Betriebssysteme Tutorats noch etwas Zeit einige weitere Features einzubauen oder möglicherweise im Rahmen eines Masterprojektes ☺.
[5] https://github.com/michel-giehl/Reti-Emulator.
[6] Womit nicht alle Studenten so viel Glück haben.
[7] Dieses ständige überlegen, wo man möglicherweise eine Erklärlücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

0.0.1 Umsetzung von Zeigern

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

0.0.1.1 Referenzierung

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

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

Code 0.1: *PicoC-Code für Zeigerreferenzierung.*

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

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

Code 0.2: *Abstrakter Syntaxbaum für Zeigerreferenzierung.*

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

Anmerkung

Die **Anzahl Speicherzellen**, die ein Zeiger[a] (z.B. eines Zeigers auf ein Feld von Integern: `pntr = int *pntr[3]`) belegt ist dabei immer: $size(type(\texttt{pntr})) = 1$ *Speicherzelle*.[bcd]

[a] Die im `size`-Attribut der **Symboltabelle** eingetragen ist.
[b] Eine **Speicherzelle** ist in der **RETI-Architektur**, wie in Unterkapitel **??** erklärt 4 *Byte* breit.
[c] Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.

[8] Später mehr dazu.

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

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

Code 0.3: *Symboltabelle für Zeigerreferenzierung.*

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

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

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

```
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('pntr'), Ref(Name('var')))
11        Ref(Global(Num('0')))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        Return(Empty())
14      ]
15  ]
```

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

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

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

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

0.0.1.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

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

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

Code 0.6: *PicoC-Code für Zeigerdereferenzierung.*

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

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

Code 0.7: *Abstrakter Syntaxbaum für Zeigerdereferenzierung.*

Im **PicoC-Shrink Pass** in Code 0.8 wird ein **Trick** angewandet, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Die Bedeutung des letzteren Knoten wurde in Unterkapitel **??** erklärt. Der Trick besteht darin, dass der **Dereferenzierungsoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Feldindex** (z.B. `var[1]`) verhält, wie es bereits im Unterkapitel **??** erläutert wurde. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Feldindex** (z.B. `var[1]`) übernehmen lassen.

```
1 File
2   Name './example_pntr_deref.picoc_shrink',
3   [
4     FunDef
5       VoidType 'void',
6       Name 'main',
7       [],
8       [
```

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

```
9         Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
10        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
          ↪  Ref(Name('var')))
11        Exp(Subscr(Name('pntr'), Num('0')))
12      ]
13    ]
```

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

0.0.2 Umsetzung von Feldern

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

0.0.2.1 Initialisierung eines Feldes

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

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

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

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

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

```
16          Assign(Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')),
            ↪  Name('ar')), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
            ↪  Array([Num('5')])]))
17      ]
18  ]
```

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

Bei der **Initialisierung** eines **Feldes** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[11]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 0.11 dargestellt ist.

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

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

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

Anmerkung

Die **Anzahl Speicherzellen**, die ein Feld[a] `datatype ar`$[\texttt{dim}_1]\dots[\texttt{dim}_k]$ belegt berechnet sich aus der **Mächtigkeit** der einzelnen **Dimensionen** des Feldes, multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Feldelemente**: $size(type(\texttt{ar})) = \left(\prod_{j=1}^{n} \texttt{dim}_j\right) \cdot size(\texttt{datatype})$.[b][c]

[a] Die im `size`-Attribut des **Symboltabelleneintrags** eingetragen ist.
[b] Die Funktion *size* berechnet die **Anzahl Speicherzellen**, die ein Datentyp belegt.
[c] Die **Funktion** *type* ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die Funktion *size* als **Definitionsmenge** Datentypen hat.

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

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

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

Im **PiocC-ANF Pass** in Code 0.12 werden zuerst die Knoten für die **Logischen Ausdrücke** in den **Blättern** des Teilbaumes, dessen Wurzel der **Feld-Initializer**-Knoten `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('5')])])` ist ausgewertet. Die Auswertung geschieht hierbei nach dem Prinzip der **Tiefensuche**, von **links-nach-rechts**. Bei dieser Auswertung werden diese Knoten für die **Logischen Ausdrücke** durch Knoten erstetzt, welche das Ergebnis dieser Ausdrücke auf den **Stack** schreiben[12].

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

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

bzw. auf den **Stackframe** geschrieben.[13]

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

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

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

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

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

Der **Trick** ist hier, dass egal wieviele **Dimensionen** und was für einen **grundlegenden Datentyp**[14] das Feld hat, man letztendlich immer das gesamte Feld erwischt, wenn man z.B. mit den **Knoten** `Assign(Global(Num('0')), Stack(Num('2')))` einfach so viele Speicherzellen rüberkopiert, wie das Feld **Speicherzellen** belegt.

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

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

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

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

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

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

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

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

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

0.0.2.2 Zugriff auf einen Feldindex

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

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

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

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

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

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

Im **PicoC-ANF Pass** in Code 0.16 wird zuerst das Schreiben der **Adresse** einer Variable `Name('ar')` des Knoten `Subscr(Name('ar'), Num('0'))` auf den **Stack** dargestellt. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Knoten `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktionm `fun` wird das durch die Knoten `Ref(Stackframe(Num('2')))` dargestellt. Diese Phase wird als **Anfangsteil ??** bezeichnet.

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

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

In Tabelle 4 ist das ganze **veranschaulicht**. In dem Auschnitt liegt die **Startadresse** $2^{31} + 67$ des Felds `int ar[3] = {1, 2, 3}` auf dem **Stack** und darüber wurde der **Wert des Index** `[1+1]` berechnet und auf dem Stack gespeichert (in rot markiert). Der Wert des Index wurde noch **nicht** auf auf die **Startadresse** des Felds **draufaddiert**.[15]

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

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

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

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

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

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

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

Tabelle 5: *Ausschnitt des Datensegments nach Schlussteil.*

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

Definition 0.1: Unterdatentyp

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

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

Anmerkung

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

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

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

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

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

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

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

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

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

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

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

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

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
      Name 'fun.1',
      [
        # // Assign(Name('ar'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
```

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

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

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

0.0.2.3 Zuweisung an Feldindex

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

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

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

Im **Abstrakten Syntaxbaum** in Code 0.19 wird eine **Zuweisung** an einen **Feldindex** `ar[2] = 42;` durch die Knoten `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

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

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

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

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

Danach ist das Vorgehen und die damit verbundenen Knoten, die dieses Vorgehen darstellen: `Ref(Global(Num('0')))`, `Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum **Anfangsteil** und **Mittelteil** aus dem vorherigen Unterkapitel 0.0.2.2. Die eben genannten Knoten stellen die Berechnung der **Adresse** des **Index**, dem das Ergebnis des Logischen Ausdrucks auf der rechten Seite des **Zuweisungsoperators** `=` zugewiesen wird dar. Dies ist in Tabelle 6 bis kurz **vor** der **Zuweisung** veranschaulicht. Als nächstes soll der Wert 42, der in rot markiert in die Speicherzelle an der **Adresse** $2^{31} + 68$ gespeichert werden.

Absolutadresse	Wert	Register
...	...	
$2^{31} + 64$	1	SP
$2^{31} + 65$	$2^{31} + 68$	
$2^{31} + 66$	42	
$2^{31} + 67$	1	
$2^{31} + 68$	2	
$2^{31} + 69$	3	
...	...	
...	...	BAF

Tabelle 6: *Ausschnitt des Datensegments vor Zuweisung.*

Zum Schluss stellen die Knoten `Assign(Stack(Num('1')), Stack(Num('2')))` die **Zuweisung** `stack(1) = stack(2)` des Ergebnisses des Ausdrucks auf der **rechten Seite der Zuweisung** zum **Feldindex** dar. Die **Adresse** des Feldindex wurde im Schritt davor berechnet. Die **Zuweisung** an den **Feldindex** ist in Tabelle 7 veranschaulicht.[16]

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

Absolutadresse	Wert	Register
...	...	
$2^{31}+64$	1	
$2^{31}+65$	$2^{31}+68$	
$2^{31}+66$	42	SP
$2^{31}+67$	1	
$2^{31}+68$	42	
$2^{31}+69$	3	
...	...	
...	...	BAF

Tabelle 7: *Ausschnitt des Datensegments nach Zuweisung.*

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

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

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

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
```

```
17          # Exp(Num('1'))
18          SUBI SP 1;
19          LOADI ACC 1;
20          STOREIN SP ACC 1;
21          # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
22          LOADIN SP IN1 2;
23          LOADIN SP IN2 1;
24          MULTI IN2 1;
25          ADD IN1 IN2;
26          ADDI SP 1;
27          STOREIN SP IN1 1;
28          # Assign(Stack(Num('1')), Stack(Num('2')))
29          LOADIN SP IN1 1;
30          LOADIN SP ACC 2;
31          ADDI SP 2;
32          STOREIN IN1 ACC 0;
33          # Return(Empty())
34          LOADIN BAF PC -1;
35      ]
36  ]
```

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

Literatur

Vorlesungen

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