

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

## PicoC-Compiler

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

BACHELORARBEIT

*Abgabedatum:* 28<sup>th</sup> April 2022

*Author:*  
Jürgen Mattheis

*Gutachter:*  
Prof. Dr. Scholl

*Betreuung:*  
M.Sc. Seufert

---

Eine Bachelorarbeit am Lehrstuhl für  
Betriebssysteme

---

---

---

## **ERKLÄRUNG**

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

---

---

# Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	IV
Grammatikverzeichnis	V
<b>1 Implementierung</b>	<b>1</b>
1.1 Lexikalische Analyse	1
1.1.1 Konkrete Syntax für die Lexikalische Analyse	1
1.1.2 Codebeispiel	3
1.2 Syntaktische Analyse	4
1.2.1 Umsetzung von Präzidenz und Assoziativität	4
1.2.2 Konkrete Syntax für die Syntaktische Analyse	9
1.2.3 Derivation Tree Generierung	11
1.2.3.1 Codebeispiel	11
1.2.3.2 Ausgabe des Derivation Tree	12
1.2.4 Derivation Tree Vereinfachung	13
1.2.4.1 Codebeispiel	14
1.2.5 Abstrakt Syntax Tree Generierung	15
1.2.5.1 PicoC-Knoten	17
1.2.5.2 RETI-Knoten	22
1.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung	23
1.2.5.4 Abstrakte Syntax	25
1.2.5.5 Codebeispiel	26
1.2.5.6 Ausgabe des Abstract Syntax Tree	27
1.3 Code Generierung	27
1.3.1 Passes	29
1.3.1.1 PicoC-Shrink Pass	30
1.3.1.1.1 Aufgabe	30
1.3.1.1.2 Abstrakte Syntax	30
1.3.1.1.3 Codebeispiel	31
1.3.1.2 PicoC-Blocks Pass	33
1.3.1.2.1 Aufgabe	33
1.3.1.2.2 Abstrakte Syntax	33
1.3.1.2.3 Codebeispiel	35
1.3.1.3 PicoC-ANF Pass	36
1.3.1.3.1 Aufgabe	36
1.3.1.3.2 Abstrakte Syntax	37
1.3.1.3.3 Codebeispiel	39
1.3.1.4 RETI-Blocks Pass	40
1.3.1.4.1 Aufgabe	40
1.3.1.4.2 Abstrakte Syntax	40
1.3.1.4.3 Codebeispiel	41

---

1.3.1.5	RETI-Patch Pass . . . . .	44
1.3.1.5.1	Aufgabe . . . . .	44
1.3.1.5.2	Abstrakte Syntax . . . . .	44
1.3.1.5.3	Codebeispiel . . . . .	45
1.3.1.6	RETI Pass . . . . .	48
1.3.1.6.1	Aufgabe . . . . .	48
1.3.1.6.2	Konkrete und Abstrakte Syntax . . . . .	48
1.3.1.6.3	Codebeispiel . . . . .	50

<b>Literatur</b>	<b>A</b>
------------------	----------

---

---

# Abbildungsverzeichnis

1.1	Ableitungsbäume zu den beiden Ableitungen . . . . .	6
1.2	Derivation Tree nach Parsen eines Ausdrucks . . . . .	13
1.3	Derivation Tree nach Vereinfachung . . . . .	14
1.4	Abstract Syntax Tree Generierung ohne Umdrehen . . . . .	16
1.5	Abstract Syntax Tree Generierung mit Umdrehen . . . . .	16
1.6	Cross-Compiler Kompiliervorgang ausgeschrieben . . . . .	28
1.7	Cross-Compiler Kompiliervorgang Kurzform . . . . .	28
1.8	Architektur mit allen Passes ausgeschrieben . . . . .	29

---

---

# Codeverzeichnis

1.1	PicoC-Code des Codebeispiels . . . . .	4
1.2	Tokens für das Codebeispiel . . . . .	4
1.3	Derivation Tree nach Derivation Tree Generierung . . . . .	12
1.4	Derivation Tree nach Derivation Tree Vereinfachung . . . . .	15
1.5	Aus vereinfachtem Derivation Tree generierter Abstract Syntax Tree . . . . .	26
1.6	PicoC Code für Codebeispiel . . . . .	32
1.7	Abstract Syntax Tree für Codebeispiel . . . . .	33
1.8	PicoC-Blocks Pass für Codebeispiel . . . . .	36
1.9	PicoC-ANF Pass für Codebeispiel . . . . .	40
1.10	RETI-Blocks Pass für Codebeispiel . . . . .	44
1.11	RETI-Patch Pass für Codebeispiel . . . . .	48
1.12	RETI Pass für Codebeispiel . . . . .	52

---

---

# Tabellenverzeichnis

1.1	Präzidenzregeln von PicoC . . . . .	5
1.2	Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren . . . . .	7
1.3	PicoC-Knoten Teil 1 . . . . .	18
1.4	PicoC-Knoten Teil 2 . . . . .	19
1.5	PicoC-Knoten Teil 3 . . . . .	20
1.6	PicoC-Knoten Teil 4 . . . . .	21
1.7	RETI-Knoten . . . . .	23
1.8	Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung . . . . .	24

---

---

# Definitionsverzeichnis

1.1	(BNF)	1
1.2	(EBNF)	1
1.3	Earley Parser	11
1.4	Label	22
1.5	Token-Knoten	22
1.6	Container-Knoten	22
1.7	Symboltabelle	37



---

---

# Grammatikverzeichnis

1.1.1 Grammatik der Konkreten Syntax der Sprache $L_{PicoC}$ für die Lexikalische Analyse in EBNF	3
1.2.1 Undurchdachte Konkrete Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF	5
1.2.2 Durchdachte Konkrete Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF . .	6
1.2.3 Beispiel für eine unäre rechtsassoziative Produktion . . . . .	7
1.2.4 Beispiel für eine unäre linksassoziative Produktion . . . . .	7
1.2.5 Beispiel für eine linksassoziative Produktion . . . . .	8
1.2.6 Beispiel für eine linksassoziative Produktion . . . . .	8
1.2.7 Durchdachte Konkrete Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF . .	9
1.2.8 Grammatik der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 1 . . . . .	10
1.2.9 Grammatik der Konkreten Syntax der Sprache $L_{PicoC}$ für die Syntaktische Analyse in EBNF, Teil 2 . . . . .	11
1.2.10 Abstrakte Syntax der Sprache $L_{PioC}$ . . . . .	25
1.3.1 Abstrakte Syntax der Sprache $L_{PioC\_Shrink}$ . . . . .	31
1.3.2 Abstrakte Syntax der Sprache $L_{PioC\_Blocks}$ . . . . .	34
1.3.3 Abstrakte Syntax der Sprache $L_{PioC\_ANF}$ . . . . .	38
1.3.4 Abstrakte Syntax der Sprache $L_{RETI\_Blocks}$ . . . . .	41
1.3.5 Abstrakte Syntax der Sprache $L_{RETI\_Patch}$ . . . . .	45
1.3.6 Konkrete Syntax der Sprache $L_{RETI}$ für die Lexikalische Analyse in EBNF . . . . .	49
1.3.7 Konkrete Syntax der Sprache $L_{RETI}$ für die Syntaktische Analyse in EBNF . . . . .	49
1.3.8 Abstrakte Syntax der Sprache $L_{RETI}$ . . . . .	49

# 1 Implementierung

In diesem Kapitel wird, nachdem im Kapitel ?? die nötigen **theoretischen Grundlagen** des **Compilerbau** vermittelt wurden, nun auf die **Implementierung** des **PicoC-Compilers** eingegangen. Aufgeteilt in die selben Kategorien **Lexikalische Analyse** 1.1, **Syntaktische Analyse** 1.2 und **Code Generierung** 1.3, wie in Kapitel ??, werden in den folgenden Unterkapiteln die einzelnen **Zwischenschritte** vom einem **Programm** in der **Konkreten Syntax** der Sprache  $L_{PicoC}$  hin zum einem Programm mit derselben **Semantik** in der **Konkreten Syntax** der Sprache  $L_{RETI}$  erklärt<sup>1</sup>.

In den folgenden Unterkapiteln werden Grammatiken für die Beschreibung der verschiedenen **Konkreten** und **Abstrakten Syntaxen**<sup>2</sup> verwendet, in welche das ursprüngliche Programm in der Konkreten Syntax der Sprache  $L_{PicoC}$

## Definition 1.1: (BNF)

## Definition 1.2: (EBNF)

## 1.1 Lexikalische Analyse

Für die **Lexikalische Analyse** ist es nur notwendig eine Grammatik zu definieren, die den Teil der **Konkreten Syntax** beschreibt, der die **verschiedenen Pattern** für die verschiedenen Token der Sprache  $L_{PicoC}$  beschreibt, also den Teil der für die **Lexikalische Analyse** wichtig ist. Diese Grammatik wird dann vom **Lark Parsing Toolkit**<sup>3</sup> dazu verwendet ein Programm in **Konkreter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel ?? erläutert ist.

### 1.1.1 Konkrete Syntax für die Lexikalische Analyse

In der Grammatik 1.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokennamen** oder einen **Teil der Beschreibung** eines **Tokennamen**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol NUM um einen **Tokennamen**, der durch die **Produktion**  $NUM ::= "0" \mid DIG\_NO\_0 \mid DIG\_WITH\_0^*$  beschrieben wird und beschreibt, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann. Das ist daran festzumachen, dass das Nicht-Terminalsymbol NUM in keiner anderen Produktion vorkommt, die auf der **linken Seite** des „**kann abgeleitet werden zu**“-**Symbols**  $::=$  ebenfalls ein **großgeschriebenen** Nicht-Terminalsymbol hat. Dagegen dient das

<sup>1</sup>Also **PicoC-Code** wird zu **RETI-Code** übersetzt

<sup>2</sup>Das **Plural** von Syntax ist **Syntaxen**, wie über ?? verifiziert werden kann.

<sup>3</sup>*Lark - a parsing toolkit for Python.*

**großgeschriebene** Nicht-Terminalsymbol `DIG_NO_0` aus der Produktion `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` nur zu Beschreibung von `NUM`.

Die in der Grammatik 1.1.1 definierten **Nicht-Terminalsymbole** können in der Grammatik 1.2.8 der **Konkreten Syntax** für die **Syntaktischen Analyse** verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl `NUM` stehen darf.

Die in der Konkreten Syntax vereinzelt **kleingeschriebenen** Nicht-Terminalsymbole, wie `name` haben nur den Zweck mehrere **Tokennamen**, wie `NAME | INT_NAME | CHAR_NAME` unter einem Überbegriff zu sammeln.

In Lark steht eine Zahl `.ZAHL`, die an ein **Nicht-Terminalsymbol** angehängt ist, dass auf der linken Seite des „**kann abgeleitet werden zu**“-Symbols `::=` einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. Es gibt den Fall, dass ein Wort von mehreren Produktionen erkannt wird, z.B. wird das Wort `int` sowohl von der Produktion `NAME`, als auch von der Produktion `INT_DT` erkannt. Daher ist es notwendig für `INT_DT` eine **Priorität** `INT_DT.2` zu setzen<sup>4</sup>, damit das Wort `int` den **Tokennamen** `INT_DT` zugewiesen bekommt und nicht `NAME`.

Allerdings muss für den Fall, dass `int` der **Präfix** eines Wortes ist, z.B. `int_var` noch die Produktion `INT_NAME.3` definiert werden, da der im **Lark Parsing Toolkit** verwendete **Basic Lexer** sobald ein Wort von einer Produktion erkannt wird, diesem direkt einen Tokennamen zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. In diesem Fall würden aus `int_var` die Token `Token('INT_DT', 'int')`, `Token('NAME', '_var')` generiert, anstatt `Token(NAME, 'int_var')`. Daher muss die Produktion `INT_NAME.3` eingeführt werden, die immer zuerst geprüft wird. Wenn es sich nur um das Wort `int` handelt, wird zuerst die Produktion `INT_NAME.3` geprüft, es stellt sich heraus, dass `int` von der Produktion `INT_NAME.3` nicht erkannt wird, daher wird als nächstes `INT_DT.2` geprüft, welches `int` erkennt.

Der **Basic Lexer** des **Lark Parsing Toolkit** funktioniert grundlegend so wie es im Unterkapitel ?? erklärt wurde, allerdings berücksichtigt der **Basic Lexer** ebenfalls **Prioritäten**, sodass für den aktuellen Index im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Wort** an dem aktuellen Index im Eingabeprogramm erkennt, bekommt es direkt den **Tokenwert** dieser Produktion zugewiesen, weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

<sup>4</sup>Es wird immer die **höchste** Priorität **zuerst** genommen.

<i>COMMENT</i>	::=	"//"/[ $\backslash$ n]*/"/*"/( $\cdot$   $\backslash$ n)*?/"*/"	<i>L_Comment</i>
<i>RETI_COMMENT.2</i>	::=	"//"" $\backslash$ ?"#"/[ $\backslash$ n]*/	
<i>DIG_NO_0</i>	::=	"1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"	<i>L_Arith</i>
<i>DIG_WITH_0</i>	::=	"0"   <i>DIG_NO_0</i>	
<i>NUM</i>	::=	"0"   <i>DIG_NO_0 DIG_WITH_0*</i>	
<i>ASCII_CHAR</i>	::=	" $\backslash$ " $\cdot$ " $\cdot$ " $\sim$ "	
<i>CHAR</i>	::=	"/'" <i>ASCII_CHAR</i> "'"	
<i>FILENAME</i>	::=	<i>ASCII_CHAR</i> + ".picoc"	
<i>LETTER</i>	::=	"a" $\cdot$ " $\cdot$ "z"   "A" $\cdot$ " $\cdot$ "Z"	
<i>NAME</i>	::=	( <i>LETTER</i>   " $\backslash$ ") ( <i>LETTER</i>   <i>DIG_WITH_0</i>   " $\backslash$ ")*	
<i>name</i>	::=	<i>NAME</i>   <i>INT_NAME</i>   <i>CHAR_NAME</i>   <i>VOID_NAME</i>	
<i>LOGIC_NOT</i>	::=	"!"	
<i>NOT</i>	::=	" $\sim$ "	
<i>REF_AND</i>	::=	"&"	
<i>un_op</i>	::=	<i>SUB_MINUS</i>   <i>LOGIC_NOT</i>   <i>NOT</i>   <i>MUL_DEREF_PNTR</i>   <i>REF_AND</i>	
<i>MUL_DEREF_PNTR</i>	::=	"*"	
<i>DIV</i>	::=	"/"	
<i>MOD</i>	::=	"%"	
<i>prec1_op</i>	::=	<i>MUL_DEREF_PNTR</i>   <i>DIV</i>   <i>MOD</i>	
<i>ADD</i>	::=	"+"	
<i>SUB_MINUS</i>	::=	"-"	
<i>prec2_op</i>	::=	<i>ADD</i>   <i>SUB_MINUS</i>	
<i>LT</i>	::=	"<"	<i>L_Logic</i>
<i>LTE</i>	::=	"<="	
<i>GT</i>	::=	">"	
<i>GTE</i>	::=	">="	
<i>rel_op</i>	::=	<i>LT</i>   <i>LTE</i>   <i>GT</i>   <i>GTE</i>	
<i>EQ</i>	::=	"=="	
<i>NEQ</i>	::=	"!="	
<i>eq_op</i>	::=	<i>EQ</i>   <i>NEQ</i>	
<i>INT_DT.2</i>	::=	"int"	<i>L_Assign_Alloc</i>
<i>INT_NAME.3</i>	::=	"int" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   " $\backslash$ ")+	
<i>CHAR_DT.2</i>	::=	"char"	
<i>CHAR_NAME.3</i>	::=	"char" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   " $\backslash$ ")+	
<i>VOID_DT.2</i>	::=	"void"	
<i>VOID_NAME.3</i>	::=	"void" ( <i>LETTER</i>   <i>DIG_WITH_0</i>   " $\backslash$ ")+	
<i>prim_dt</i>	::=	<i>INT_DT</i>   <i>CHAR_DT</i>   <i>VOID_DT</i>	

**Grammar 1.1.1:** Grammatik der Konkreten Syntax der Sprache  $L_{Picoc}$  für die Lexikalische Analyse in EBNF

### 1.1.2 Codebeispiel

In den folgenden Unterkapiteln wird das Beispiel in Code 1.1 dazu verwendet die Konstruktion eines **Abstract Syntax Trees** in seinen einzelnen **Zwischenschritten** zu erläutern.

```

1 struct st {int *(*attr)[4][5];};
2
3 void main() {
4     struct st *(*var[3][2]);
5 }

```

Code 1.1: PicoC-Code des Codebeispiels

Die vom **Basic Lexer** des **Lark Parsing Toolkit** erkannten **Token** sind Code 1.2 zu sehen.

```

1 [Token('FILENAME', './verbose_dt_simple_ast_gen_array_decl_and_alloc.picoc'), Token('STRUCT',
  ↳ 'struct'), Token('NAME', 'st'), Token('LBRACE', '{'), Token('INT_DT', 'int'),
  ↳ Token('MUL_DEREF_PNTR', '*'), Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'),
  ↳ Token('NAME', 'attr'), Token('RPAR', ')'), Token('LSQB', '['), Token('NUM', '4'),
  ↳ Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '5'), Token('RSQB', ']'),
  ↳ Token('SEMICOLON', ';'), Token('RBRACE', '}'), Token('SEMICOLON', ';'), Token('VOID_DT',
  ↳ 'void'), Token('NAME', 'main'), Token('LPAR', '('), Token('RPAR', ')'), Token('LBRACE',
  ↳ '{'), Token('STRUCT', 'struct'), Token('NAME', 'st'), Token('MUL_DEREF_PNTR', '*'),
  ↳ Token('LPAR', '('), Token('MUL_DEREF_PNTR', '*'), Token('NAME', 'var'), Token('LSQB',
  ↳ '['), Token('NUM', '3'), Token('RSQB', ']'), Token('LSQB', '['), Token('NUM', '2'),
  ↳ Token('RSQB', ']'), Token('RPAR', ')'), Token('SEMICOLON', ';'), Token('RBRACE', '}')]

```

Code 1.2: Tokens für das Codebeispiel

## 1.2 Syntaktische Analyse

In der **Syntaktischen Analyse** ist es die Aufgabe des **Parsers** aus einem Programm in **Konkreter Syntax** unter Verwendung der **Tokens** aus der **Lexikalischen Analyse** eine **Derivation Tree** zu generieren. Es ist danach die Aufgabe möglicher **Visitors** und die Aufgabe des **Transformers** aus diesem **Derivation Tree** einen **Abstract Syntax Tree** in **Abstrakter Syntax** zu generieren.

### 1.2.1 Umsetzung von Präzedenz und Assoziativität

Die Programmiersprache  $L_{PicoC}$  hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache  $L_C$ <sup>5</sup>. Die **Präzidenzregeln** der Programmiersprache  $L_{PicoC}$  sind in Tabelle 1.1 aufgelistet.

<sup>5</sup>C Operator Precedence - [cppreference.com](http://cppreference.com).

Präzidenzstufe	Operatoren	Beschreibung	Assoziativität
1	<code>a()</code>	Funktionsaufruf	Links, dann rechts $\rightarrow$
	<code>a[]</code>	Indezzugriff	
	<code>a.b</code>	Attributzugriff	
2	<code>-a</code>	Unäres Minus	Rechts, dann links $\leftarrow$
	<code>!a ~a</code>	Logisches NOT und Bitweise NOT	
	<code>*a &amp;a</code>	Dereferenz und Referenz, auch Adresse-von	
3	<code>a*b a/b a%b</code>	Multiplikation, Division und Modulo	Links, dann rechts $\rightarrow$
4	<code>a+b a-b</code>	Addition und Subtraktion	
5	<code>a&lt;b a&lt;=b a&gt;b a&gt;=b</code>	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	<code>a==b a!=b</code>	Gleichheit und Ungleichheit	
7	<code>a&amp;b</code>	Bitweise UND	
8	<code>a^b</code>	Bitweise XOR (exclusive or)	
9	<code>a b</code>	Bitweise ODER (inclusive or)	
10	<code>a&amp;&amp;b</code>	Logisches UND	
11	<code>a  b</code>	Logisches ODER	Rechts, dann links $\leftarrow$
12	<code>a=b</code>	Zuweisung	

Tabelle 1.1: Präzidenzregeln von *PicoC*

Würde man diese **Operatoren** ohne Beachtung von **Präzidenzregeln** (Definition ??) und **Assoziativität** (Definition ??) in eine Grammatik verarbeiten wollen, so könnte eine Grammatik, wie Grammatik 1.2.1 dabei rauskommen.

```

prim_exp ::= name | NUM | CHAR | "("exp")"
un_op    ::= "-" | "~" | "!" | "*" | "&"
un_exp   ::= un_op exp
bin_op   ::= "*" | "/" | "%" | "+" | "-" | "&" | "^" | "|"
          | "<" | "<=" | ">" | ">=" | "!=" | "==" | "&&" | "||"
bin_exp  ::= exp bin_op exp
exp      ::= prim_exp | un_exp | bin_exp

```

*L<sub>Arith</sub>* +

**Grammar 1.2.1:** Undurchdachte Konkrete Syntax der Sprache  $L_{PicoC}$  für die Syntaktische Analyse in EBNF

Die Grammatik 1.2.1 ist allerdings **mehrdeutig**, d.h. verschiedene **Linksableitungen** in der Grammatik können zum selben **Wort** abgeleitet werden. Z.B. kann das Wort `3 * 1 && 4` sowohl über die **Linksableitung 1.2.1** als auch über die **Linksableitung 1.2.2** abgeleitet werden.

$$\begin{aligned}
 \text{exp} &\Rightarrow \text{bin\_exp} \Rightarrow \text{exp bin\_op exp} \Rightarrow \text{bin\_exp bin\_op exp} & (1.2.1) \\
 &\Rightarrow \text{exp bin\_op exp bin\_op exp} \Rightarrow^* 3 * 1 \&\& 4
 \end{aligned}$$

$$\begin{aligned}
 \text{exp} &\Rightarrow \text{bin\_exp} \Rightarrow \text{exp bin\_op exp} \Rightarrow \text{prim\_exp bin\_op exp} \Rightarrow \text{NUM bin\_op exp} & (1.2.2) \\
 &\Rightarrow 3 \text{ bin\_op exp} \Rightarrow 3 * \text{exp} \Rightarrow 3 * \text{bin\_exp} \Rightarrow 3 * \text{exp bin\_exp} \Rightarrow^* 3 * 1 \&\& 4
 \end{aligned}$$

Beide **Wörter** sind **gleich**, allerdings sind die **Ableitungsbäume unterschiedlich**, wie in Abbildung 1.1 zu sehen ist.



Abbildung 1.1: Ableitungsbäume zu den beiden Ableitungen

Der **linke Baum** entspricht Ableitung 1.2.1 und der **rechte Baum** entspricht Ableitung 1.2.2. Würde man in den Ausdrücken, die von diesen Bäumen dargestellt sind in **Klammern** setzen, um die **Präzidenz** sichtbar zu machen, so würde Ableitung 1.2.1 die Klammerung  $(3 * 1) \&\& 4$  haben und die Ableitung 1.2.2 die Klammerung  $3 * (1 \&\& 4)$  haben.

Aus diesem Grund ist es wichtig die **Präzidenzregeln** und die **Assoziativität** der Operatoren beim Erstellen der Grammatik miteinzubeziehen. Hierzu wird nun Tabelle 1.1 betrachtet. Für jede **Präzidenzstufe** in der Tabelle 1.1 wird eine eigene Regel erstellt werden, wie es in Grammatik 1.2.2 dargestellt ist. Zudem braucht es eine **Produktion** `prim_exp` für die höchste **Präzidenzstufe**, welche **Literale**, wie 'c', 5 oder `var` und geklammerte Ausdrücke wie  $(3 \&\& 14)$  abdeckt.

<code>prim_exp</code>	<code>::=</code>	<code>... L_Arith + L_Array</code>
<code>post_exp</code>	<code>::=</code>	<code>... + L_Pntr + L_Struct</code>
<code>un_exp</code>	<code>::=</code>	<code>... + L_Fun</code>
<code>arith_prec1</code>	<code>::=</code>	<code>...</code>
<code>arith_prec2</code>	<code>::=</code>	<code>...</code>
<code>arith_and</code>	<code>::=</code>	<code>...</code>
<code>arith_oplus</code>	<code>::=</code>	<code>...</code>
<code>arith_or</code>	<code>::=</code>	<code>...</code>
<code>rel_exp</code>	<code>::=</code>	<code>... L_Logic</code>
<code>eq_exp</code>	<code>::=</code>	<code>...</code>
<code>logic_and</code>	<code>::=</code>	<code>...</code>
<code>logic_or</code>	<code>::=</code>	<code>...</code>
<code>assign_stmt</code>	<code>::=</code>	<code>... L_Assign</code>

**Grammar 1.2.2:** Durchdachte Konkrete Syntax der Sprache  $L_{PiCoC}$  für die Syntaktische Analyse in EBNF

Einigen **Bezeichnungen** der **Produktionen** sind in Tabelle 1.2 ihren jeweiligen **Operatoren** zugeordnet für welche sie zuständig sind.

Bezeichnung der Produktionsregel	Operatoren
post_exp	a() a[] a.b
un_exp	-a !a ~a *a &a
arith_prec1	a*b a/b a%b
arith_prec2	a+b a-b
arith_and	a<b a<=b a>b a>=b
arith_oplus	a==b a!=b
arith_or	a&b
rel_exp	a~b
eq_exp	a b
logic_and	a&&b
logic_or	a  b
assign	a=b

**Tabelle 1.2:** Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren

Als nächstes müssen die einzelnen **Produktionen** entsprechend der **Ausdrücke** für die sie zuständig sind definiert werden. Jede der **Produktionen** soll nur Ausdrücke **erkennen** können, deren **Präzidenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzidenzstufe **höher** ist. Z.B. soll **un\_op** sowohl den Ausdruck  $-(3 * 14)$  als auch einfach nur  $(3 * 14)$ <sup>6</sup> erkennen können, aber nicht  $3 * 14$  ohne Klammern, da dieser Ausdruck eine **geringe Präzidenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

Bei z.B. der Produktion **un\_exp** in 1.2.3 für die **rechtsassoziativen unären Operatoren**  $-a$ ,  $!a$ ,  $\sim a$ ,  $*a$  und  $\&a$  ist die **Alternative** **un\_op un\_exp** dafür zuständig, dass diese unären Operatoren **rechtsassoziativ** geschachtelt werden können (z.B.  $! \sim 42$ ). Die Alternative **post\_exp** ist dafür zuständig, dass die Produktion auch **terminieren** kann und es auch möglich ist ausschließlich einen Ausdruck **höherer Präzidenz** (z.B. 42) zu haben.

$$un\_exp ::= un\_op\ un\_exp \mid post\_exp$$

**Grammar 1.2.3:** Beispiel für eine unäre rechtsassoziative Produktion

Bei z.B. der Produktion **post\_exp** in 1.2.4 für die **linksassoziativen unären Operatoren**  $a()$ ,  $a[]$  und  $a.b$  sind die Alternativen **post\_exp["logic\_or"]** und **post\_exp"."name** dafür zuständig, dass diese unären Operatoren **linksassoziativ** geschachtelt werden können (z.B.  $ar[3][1].car[4]$ ). Die Alternative **name("fun\_args")** ist für einen **einzelnen Funktionsaufruf** zuständig. Die Alternative **prim\_exp** ist dafür zuständig, dass die Produktion nicht nur bei **name("fun\_args")** **terminieren** kann und es auch möglich ist ausschließlich einen Ausdruck der **höchsten Präzidenz** (z.B. 42) zu haben.

$$post\_exp ::= post\_exp["logic\_or"] \mid post\_exp"."name \mid name("fun\_args") \mid prim\_exp$$

**Grammar 1.2.4:** Beispiel für eine unäre linksassoziative Produktion

Bei z.B. der Produktion **prec2\_exp** in 1.2.5 für die **binären linksassoziativen Operatoren**  $a+b$  und  $a-b$  ist die **Alternative** **arith\_prec2 prec2\_op arith\_prec1** dafür zuständig, dass **mehrere** Operationen der Präzidenzstufe 4 in Folge erkannt werden können<sup>7</sup> (z.B.  $3 + 1 - 4$ , wobei  $-$  und  $+$  beide Präzidenzstufe 4 haben). Das **Nicht-Terminalsymbol** **arith\_prec1** auf der **rechten Seite** ermöglicht es, dass zwischen den

<sup>6</sup>Geklammerte Ausdrücke werden nämlich von **prim\_exp** erkannt, welches eine höhere **Präzidenzstufe** hat.

<sup>7</sup>Bezogen auf Tabelle 1.1.



Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B.  $3 + 1 / 4 - 1$ , wobei  $-$  und  $+$  beide Präzidenzstufe 4 haben und  $/$  Präzidenzstufe 3). Mit der Alternative `arith_prec1` ist es möglich, dass ausschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B.  $1 / 4$ ).

$$\text{arith\_prec2} ::= \text{arith\_prec2 } \text{prec2\_op } \text{arith\_prec1} \mid \text{arith\_prec1}$$

**Grammar 1.2.5:** *Beispiel für eine linksassoziative Produktion*

Manche **Parser**<sup>a</sup> haben allerdings ein Problem mit **Linksrekursion** (Definition ??), wie sie z.B. in der Produktion 1.2.5 vorliegt. Dieses Problem lässt sich allerdings einfach lösen, indem man die Produktion 1.2.5 zur Produktion 1.2.6 umschreibt.

$$\text{arith\_prec2} ::= \text{arith\_prec1 } (\text{prec2\_op } \text{arith\_prec1})^*$$

**Grammar 1.2.6:** *Beispiel für eine linksassoziative Produktion*

Die von Produktion 1.2.6 erkannten Ausdrücke sind dieselben, wie für die Produktion 1.2.5, allerdings ist die Produktion 1.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 1.2) definierten  $*$ -Operator, um mehrere Operationen der Präzidenzstufe 4 in Folge erkennen zu können (z.B.  $3 + 1 - 4$ , wobei  $-$  und  $+$  beide Präzidenzstufe 4 haben).

Das **Nicht-Terminalsymbol** `arith_prec1` erlaubt es, dass **zwischen** der Folge von Operationen der Präzidenzstufe 4 auch Operationen der Präzidenzstufe 3 auftauchen können (z.B.  $3 + 1 / 4 - 1$ , wobei  $-$  und  $+$  beide Präzidenzstufe 4 haben und  $/$  Präzidenzstufe 3). Da der in der EBNF definierte  $*$ -Operator auch bedeutet, dass das Teilpattern auf das er sich bezieht **kein einziges mal** vorkommen kann, ist es mit dem **linken Nicht-Terminalsymbol** `arith_prec1` möglich, dass ausschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B.  $1 / 4$ ).

<sup>a</sup>Darunter zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird **nicht**.

Alle **Operatoren** der Sprache  $L_{PicoC}$  sind also entweder **binär** und **linksassoziativ** (z.B.  $a*b$ ,  $a-b$ ,  $a>=b$  oder  $a\&\&b$ ), **unär** und **rechtsassoziativ** (z.B.  $\&a$  oder  $!a$ ) oder **unär** und **linksassoziativ** (z.B.  $a[]$  oder  $a()$ ). Somit ergibt sich die Grammatik 1.2.7.

<i>prec1_op</i>	::=	"*"   "/"   "%"	<i>L_Misc</i>
<i>prec2_op</i>	::=	"+"   "-"	
<i>rel_op</i>	::=	"<"   "<="   ">"   ">="	
<i>eq_op</i>	::=	"=="   "!="	
<i>fun_args</i>	::=	[ <i>logic_or</i> ("," <i>logic_or</i> )*]	
<i>prim_exp</i>	::=	<i>name</i>   <i>NUM</i>   <i>CHAR</i>   "(" <i>logic_or</i> ")"	<i>L_Arith</i>
<i>post_exp</i>	::=	<i>post_exp</i> [" <i>logic_or</i> "]   <i>post_exp</i> ." <i>name</i> "   <i>name</i> (" <i>fun_args</i> ")"	+ <i>L_Array</i>
		<i>prim_exp</i>	+ <i>L_Pntr</i>
<i>un_exp</i>	::=	<i>un_op</i> <i>un_exp</i>   <i>post_exp</i>	+ <i>L_Struct</i>
<i>arith_prec1</i>	::=	<i>arith_prec1</i> <i>prec1_op</i> <i>un_exp</i>   <i>un_exp</i>	+ <i>L_Fun</i>
<i>arith_prec2</i>	::=	<i>arith_prec2</i> <i>prec2_op</i> <i>arith_prec1</i>   <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i>   <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i>   <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i>   <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp</i> <i>rel_op</i> <i>arith_or</i>   <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp</i> <i>eq_op</i> <i>rel_exp</i>   <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i>   <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> "  " <i>logic_and</i>   <i>logic_and</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	<i>L_Assign</i>

**Grammar 1.2.7:** Durchdachte Konkrete Syntax der Sprache  $L_{PicoC}$  für die Syntaktische Analyse in EBNF

## 1.2.2 Konkrete Syntax für die Syntaktische Analyse

Die gesamte Grammatik 1.2.8, welche die **Konkrete Syntax** der Sprache  $L_{PicoC}$  für die **Syntaktische Analyse** beschreibt ergibt sich wenn man die Grammatik 1.2.7 um die **restliche Syntax** der Sprache  $L_{PicoC}$  erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 1.2.7 erläutert ergibt.

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrete Syntax** an die **aktuellste konsistentlos auffindbare Version** der echten Grammatik *ANSI C grammar (Yacc)* der Sprache  $L_C$  angepasst<sup>8</sup>, damit es sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache  $L_C$ , wobei z.B. die Compiler **GCC**<sup>9</sup> und **Clang**<sup>10</sup> zu nennen wären.

In der Grammatik 1.2.8, welche die **Konkrete Syntax** der Sprache  $L_{PicoC}$  für die **Syntaktische Analyse** beschreibt, werden einige der **Tokennamen** aus der Grammatik 1.1.1 der **Konkreten Syntax** für die **Lexikalischen Analyse** verwendet, wie z.B. *NUM* aber auch *name*, welches eine Produktion ist, die mehrere **Tokennamen** unter einem Überbegriff zusammenfasst.

Terminalsymbole, wie ; oder && gehören eigentlich zur **Lexikalischen Analyse**, jedoch erlaubt das **Lark Parsing Toolkit** um die Grammatik leichter lesbar zu machen einige **Terminalsymbole** einfach direkt in die Grammatik 1.2.8 der **Konkreten Syntax** für die **Syntaktische Analyse** zu schreiben. Der **Tokenname** für diese Terminalsymbole wird in diesem Fall vom **Lark Parsing Toolkit** bestimmt, welches einige sehr häufige verwendete **Terminalsymbole**, wie ; oder && bereits einen **Tokennamen** zugewiesen hat.

<sup>8</sup>An der für die Programmiersprache  $L_{PicoC}$  relevanten **Syntax** hat sich allerdings über die Jahre nichts verändert, wie die Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Yacc)* und **Lexikalische Analyse** *ANSI C grammar (Lex)* aus dem Jahre 1985 zeigen.

<sup>9</sup>*GCC, the GNU Compiler Collection - GNU Project.*

<sup>10</sup>*clang: C++ Compiler.*

<i>prim_exp</i>	::=	<i>name</i>   <i>NUM</i>   <i>CHAR</i>   "(" <i>logic_or</i> ")"	<i>L_Arith</i> + <i>L_Array</i>
<i>post_exp</i>	::=	<i>array_subscr</i>   <i>struct_attr</i>   <i>fun_call</i>	+ <i>L_Pntr</i> + <i>L_Struct</i>
		<i>input_exp</i>   <i>print_exp</i>   <i>prim_exp</i>	+ <i>L_Fun</i>
<i>un_exp</i>	::=	<i>un_op un_exp</i>   <i>post_exp</i>	
<i>input_exp</i>	::=	"input" "(" ")"	
<i>print_exp</i>	::=	"print" "(" <i>logic_or</i> ")"	
<i>arith_prec1</i>	::=	<i>arith_prec1 prec1_op un_exp</i>   <i>un_exp</i>	
<i>arith_prec2</i>	::=	<i>arith_prec2 prec2_op arith_prec1</i>   <i>arith_prec1</i>	
<i>arith_and</i>	::=	<i>arith_and</i> "&" <i>arith_prec2</i>   <i>arith_prec2</i>	
<i>arith_oplus</i>	::=	<i>arith_oplus</i> "^" <i>arith_and</i>   <i>arith_and</i>	
<i>arith_or</i>	::=	<i>arith_or</i> " " <i>arith_oplus</i>   <i>arith_oplus</i>	
<i>rel_exp</i>	::=	<i>rel_exp rel_op arith_or</i>   <i>arith_or</i>	<i>L_Logic</i>
<i>eq_exp</i>	::=	<i>eq_exp eq_op rel_exp</i>   <i>rel_exp</i>	
<i>logic_and</i>	::=	<i>logic_and</i> "&&" <i>eq_exp</i>   <i>eq_exp</i>	
<i>logic_or</i>	::=	<i>logic_or</i> "  " <i>logic_and</i>   <i>logic_and</i>	
<i>type_spec</i>	::=	<i>prim_dt</i>   <i>struct_spec</i>	<i>L_Assign_Alloc</i>
<i>alloc</i>	::=	<i>type_spec pntr_decl</i>	
<i>assign_stmt</i>	::=	<i>un_exp</i> "=" <i>logic_or</i> ";"	
<i>initializer</i>	::=	<i>logic_or</i>   <i>array_init</i>   <i>struct_init</i>	
<i>init_stmt</i>	::=	<i>alloc</i> "=" <i>initializer</i> ";"	
<i>const_init_stmt</i>	::=	"const" <i>type_spec name</i> "=" <i>NUM</i> ";"	
<i>pntr_deg</i>	::=	"*"	<i>L_Pntr</i>
<i>pntr_decl</i>	::=	<i>pntr_deg array_decl</i>   <i>array_decl</i>	
<i>array_dims</i>	::=	("[" <i>NUM</i> "]")*	<i>L_Array</i>
<i>array_decl</i>	::=	<i>name array_dims</i>   "(" <i>pntr_decl</i> ")" <i>array_dims</i>	
<i>array_init</i>	::=	"{" <i>initializer</i> "," <i>initializer</i> "*" }	
<i>array_subscr</i>	::=	<i>post_exp</i> "[" <i>logic_or</i> "]"	
<i>struct_spec</i>	::=	"struct" <i>name</i>	<i>L_Struct</i>
<i>struct_params</i>	::=	( <i>alloc</i> ";" )+	
<i>struct_decl</i>	::=	"struct" <i>name</i> "{" <i>struct_params</i> "}"	
<i>struct_init</i>	::=	"{" <i>name</i> "=" <i>initializer</i>	
		( "," <i>name</i> "=" <i>initializer</i> )" }	
<i>struct_attr</i>	::=	<i>post_exp</i> "." <i>name</i>	
<i>if_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_If_Else</i>
<i>if_else_stmt</i>	::=	"if" "(" <i>logic_or</i> ")" <i>exec_part</i> "else" <i>exec_part</i>	
<i>while_stmt</i>	::=	"while" "(" <i>logic_or</i> ")" <i>exec_part</i>	<i>L_Loop</i>
<i>do_while_stmt</i>	::=	"do" <i>exec_part</i> "while" "(" <i>logic_or</i> ")" ";"	

**Grammar 1.2.8:** Grammatik der Konkreten Syntax der Sprache  $L_{PicoC}$  für die Syntaktische Analyse in EBNF, Teil 1

<code>decl_exp_stmt</code>	<code>::= alloc";"</code>	<i>L_Stmt</i>
<code>decl_direct_stmt</code>	<code>::= assign_stmt   init_stmt   const_init_stmt</code>	
<code>decl_part</code>	<code>::= decl_exp_stmt   decl_direct_stmt   RETI_COMMENT</code>	
<code>compound_stmt</code>	<code>::= "{" exec_part * "}"</code>	
<code>exec_exp_stmt</code>	<code>::= logic_or";"</code>	
<code>exec_direct_stmt</code>	<code>::= if_stmt   if_else_stmt   while_stmt   do_while_stmt</code>	
	<code>  assign_stmt   fun_return_stmt</code>	
<code>exec_part</code>	<code>::= compound_stmt   exec_exp_stmt   exec_direct_stmt</code>	
	<code>  RETI_COMMENT</code>	
<code>decl_exec_stmts</code>	<code>::= decl_part * exec_part*</code>	
<code>fun_args</code>	<code>::= [logic_or(" ", logic_or)*]</code>	<i>L_Fun</i>
<code>fun_call</code>	<code>::= name("fun_args")</code>	
<code>fun_return_stmt</code>	<code>::= "return" [logic_or];"</code>	
<code>fun_params</code>	<code>::= [alloc(" ", alloc)*]</code>	
<code>fun_decl</code>	<code>::= type_spec ptr_deg name("fun_params")</code>	
<code>fun_def</code>	<code>::= type_spec ptr_deg name("fun_params") "{" decl_exec_stmts "}"</code>	
<code>decl_def</code>	<code>::= (struct_decl   fun_decl);"   fun_def</code>	<i>L_File</i>
<code>decls_defs</code>	<code>::= decl_def*</code>	
<code>file</code>	<code>::= FILENAME decls_defs</code>	

**Grammar 1.2.9:** Grammatik der Konkreten Syntax der Sprache  $L_{PicoC}$  für die Syntaktische Analyse in EBNF, Teil 2

In der Grammatik 1.2.8 sind alle **Grammatiksymbole** ausgegraut, die das **Bachelorprojekt** betreffen. Alle nicht ausgegrauten **Grammatiksymbole** wurden für die Implementierung der **neuen Funktionalitäten**, welche die **Bachelorarbeit** betreffen hinzugefügt.

### 1.2.3 Derivation Tree Generierung

Die in Unterkapitel 1.2.2 definierte **Konkrete Syntax**, die von der Grammatik 1.2.8 beschrieben wird lässt sich mithilfe des **Earley Parsers** (Definition 1.3) von Lark dazu verwenden Code, der in der Sprache  $L_{PicoC}$  geschrieben ist zu parsen um einen **Derivation Tree** zu generieren.

#### Definition 1.3: Earley Parser

#### 1.2.3.1 Codebeispiel

Der **Derivation Tree**, der mithilfe des **Earley Parsers** und der **Token** der **Lexikalischen Analyse** aus dem Beispiel in Code 1.1 generiert wurde, ist in Code 1.3 zu sehen. Im Code 1.3 wurden einige Zeilen **markiert**, die später in Unterkapitel 1.2.4.1 zum Vergleich wichtig sind.

```

1 file
2   ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt
3   decls_defs
4     decl_def
5       struct_decl
6         name      st

```

```

7      struct_params
8      alloc
9      type_spec
10     prim_dt      int
11     pntr_decl
12     pntr_deg      *
13     array_decl
14     pntr_decl
15     pntr_deg      *
16     array_decl
17     name      attr
18     array_dims
19     array_dims
20     4
21     5
22 decl_def
23 fun_def
24     type_spec
25     prim_dt      void
26     pntr_deg
27     name      main
28     fun_params
29     decl_exec_stmts
30     decl_part
31     decl_exp_stmt
32     alloc
33     type_spec
34     struct_spec
35     name      st
36     pntr_decl
37     pntr_deg      *
38     array_decl
39     pntr_decl
40     pntr_deg      *
41     array_decl
42     name      var
43     array_dims
44     3
45     2
46     array_dims

```

Code 1.3: Derivation Tree nach Derivation Tree Generierung

### 1.2.3.2 Ausgabe des Derivation Tree

Die Ausgabe des **Derivation Tree** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Grammatik den **linken Seiten** des „kann abgeleitet werden zu“-Symbols  $::=$ <sup>11</sup> entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition ?? eines **Derivation Tree** auch schon definiert ist. Die **EBNF-Grammatik 1.2.8** des **PicoC-Compilers** erlaubt es allerdings auch, dass in einem **Blatt** garnichts  $\epsilon$  steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("NUM")*` gibt, in denen auch das **leere Wort**  $\epsilon$  abgeleitet werden kann.

Der **Abstract Syntax Tree** ist bewusst so gewählt, dass er sich optisch vom **Derivation Tree** unterscheidet.

<sup>11</sup>*Grammar: The language of languages (BNF, EBNF, ABNF and more).*

indem die Bezeichner der **Knoten** in **UpperCamelCase** geschrieben sind, im Gegensatz zum **Derivation Tree**, dessen **Innere Knoten** im **snake\_case** geschrieben sind<sup>12</sup>, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des „**kann abgeleitet werden zu**“-Symbols **::=**.

### 1.2.4 Derivation Tree Vereinfachung

Der **Derivation Tree** in Code 1.3, dessen Generierung in Unterkapitel 1.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Transformers** ein **Abstract Syntax Tree** generiert werden kann. Das Problem ist, dass um den **Datentyp** einer Variable in der Programmiersprache  $L_C$  und somit auch die Programmiersprache  $L_{PicoC}$  korrekt bestimmen zu können, wie z.B. ein „**Array der Mächtigkeit 3 von Pointern auf Arrays der Mächtigkeit 2 von Integeren**“ `int (*ar[3])[2]` die **Spiralregel**<sup>13</sup> in der Implementierung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Grammatik 1.2.8 der **Konkreten Syntax** auf eine spezielle Weise passend spezifiziert.

Was man erhalten will, ist ein **entarteter Baum** von **PicoC-Knoten**, an dem man den **Datentyp** direkt ablesen kann, indem man sich einfach über den **entarteten Baum** bewegt, wie z.B. `PntrDecl(Num('1'),ArrayDecl([Num('3'),Num('2')],PntrDecl(Num('1'),StructSpec(Name('st')))))` für den Ausdruck `struct st *(*var[3][2])`.

Es sind hierbei mehrere Probleme zu lösen. Hat man den Ausdruck `struct st *(*var[3][2])` wird dieser zu einem **Derivation Tree**, wie er in Abbildung 1.2 zu sehen ist.

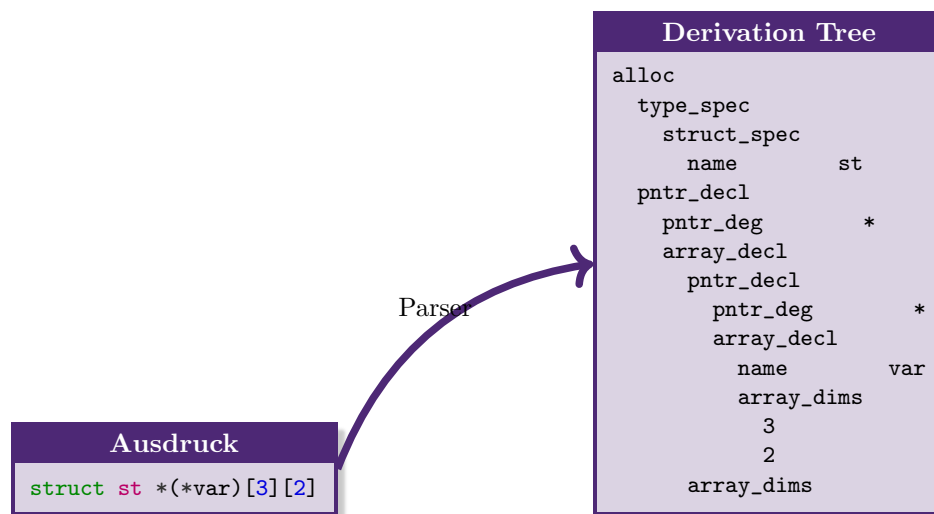


Abbildung 1.2: Derivation Tree nach Parsen eines Ausdrucks

Dieser **Derivation Tree** für den Ausdruck `struct st *(*var[3][2])` hat allerdings einen Aufbau welcher durch die **Syntax** der **Pointerdeklaratoren** `pnter_decl(num, datatype)` und **Arraydeklaratoren** `array_decl(datatype, nums)` bestimmt ist, die **spiralähnlich** ist. Man würde allerdings gerne einen **entarteten Baum** erhalten, bei dem der Datentyp immer im **zweiten Attribut** weitergeht, anstatt abwechselnd im **zweiten** und **ersten**, wie beim **Pointerdeklarator** `pnter_decl(num, datatype)` und **Arraydeklarator** `array_decl(datatype, nums)`. Daher muss beim **ArrayDeclarator** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht werden.

<sup>12</sup>Naming convention (programming).

<sup>13</sup>Clockwise/Spiral Rule.

Des Weiteren befindet sich in der **Mitte** dieser **Spirale**, die der **Derivation Tree** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`, da der **Derivation Tree** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Programms** in **Konkreter Syntax** (z.B. `struct st *(*var[3][2])`) **generiert** wird darstellt. Der **Name der Variable** `name(var)` sollte daher mit dem **innersten Datentyp** `struct st` ausgetauscht werden.

In Abbildung 1.3 ist daher zu sehen, wie der **Derivation Tree** aus Abbildung 1.2 mithilfe eines **Visitors** (Definition ??) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

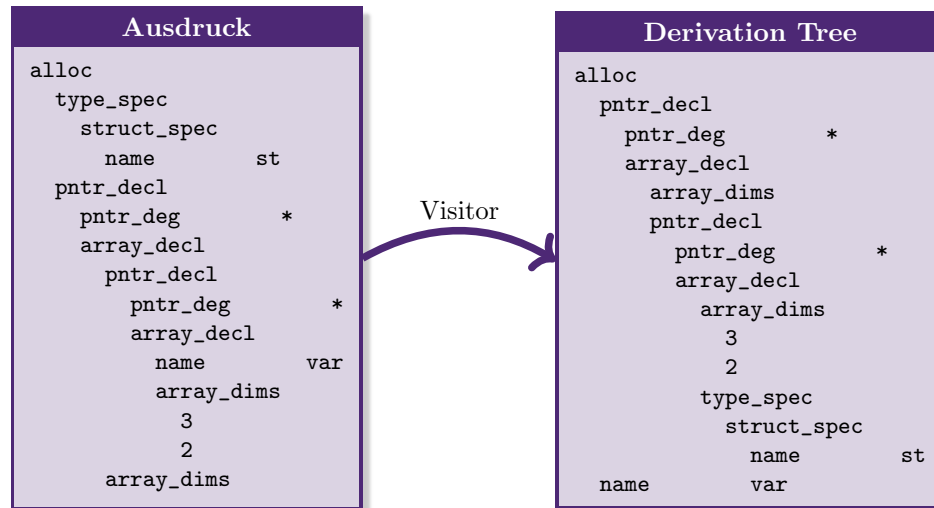


Abbildung 1.3: Derivation Tree nach Vereinfachung

#### 1.2.4.1 Codebeispiel

In Code 1.4 ist der **Derivation Tree** aus Code 1.3 nach der **Vereinfachung** mithilfe eines **Visitors** zu sehen.

```

1 file
2 ./verbose_dt_simple_ast_gen_array_decl_and_alloc.dt_simple
3 decls_defs
4   decl_def
5     struct_decl
6       name      st
7       struct_params
8         alloc
9           ptr_decl
10            ptr_deg   *
11            array_decl
12              array_dims
13                4
14                5
15            ptr_decl
16              ptr_deg   *
17              array_decl
18                array_dims
19                type_spec
20                  prim_dt      int
  
```

```

21      name      attr
22  decl_def
23    fun_def
24      type_spec
25        prim_dt      void
26      ptr_deg
27      name      main
28      fun_params
29      decl_exec_stmts
30      decl_part
31      decl_exp_stmt
32      alloc
33        ptr_decl
34          ptr_deg      *
35          array_decl
36            array_dims
37            ptr_decl
38              ptr_deg      *
39              array_decl
40                array_dims
41                  3
42                  2
43                type_spec
44                  struct_spec
45                    name      st
46      name      var

```

Code 1.4: Derivation Tree nach Derivation Tree Vereinfachung

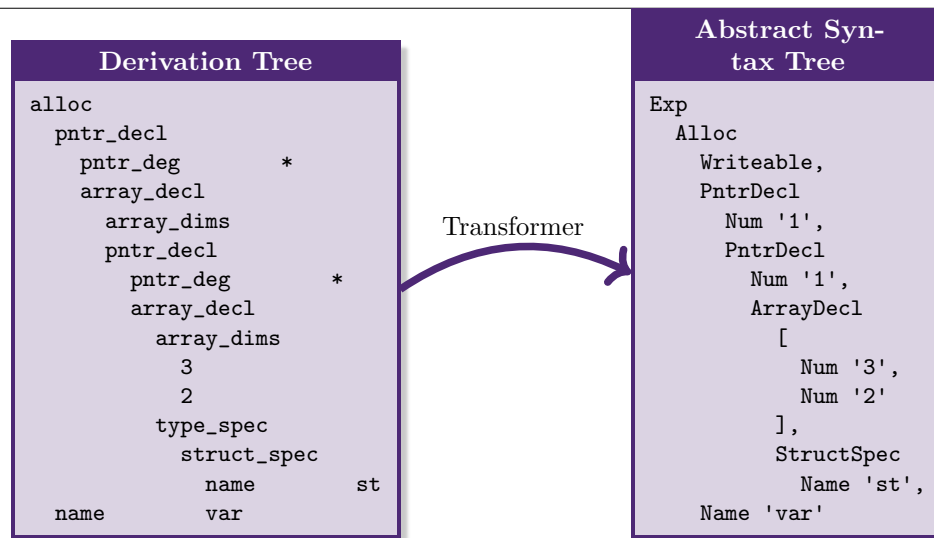
### 1.2.5 Abstrakt Syntax Tree Generierung

Nachdem der **Derivation Tree** in Unterkapitel 1.2.4 vereinfacht wurde, ist der **vereinfachte Derivation Tree** in Code 1.4 nun dazu geeignet, um mit einem **Transformer** (Definition ??) einen **Abstract Syntax Tree** aus ihm zu generieren. Würde man den **vereinfachten Derivation Tree** des Ausdrucks `struct st *(*var[3][2])` auf passende Weise in einen **Abstract Syntax Tree** umwandeln, so würde dabei ein **Abstract Syntax Tree** wie in Abbildung 1.4 rauskommen.

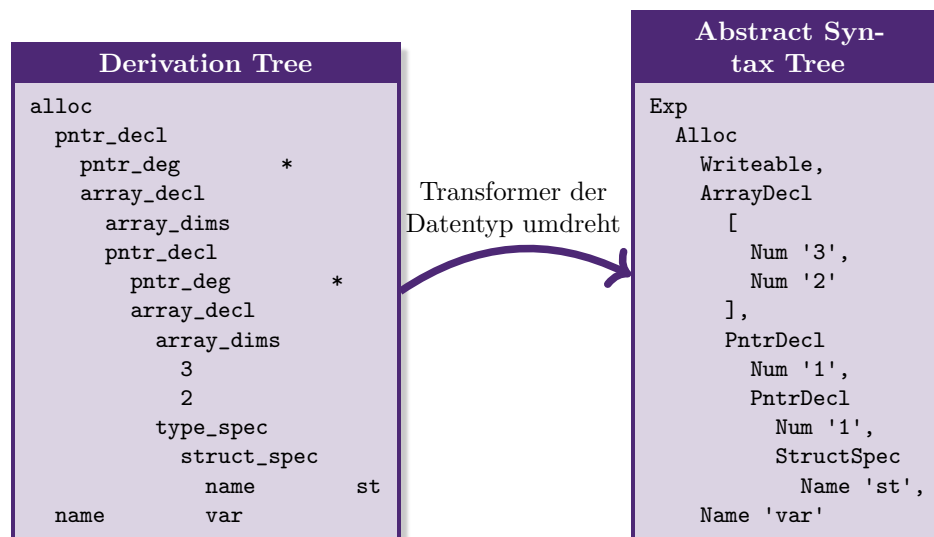
Den Teilbaum, der den Datentyp darstellt würde man von **oben-nach-unten**<sup>14</sup> als „**Pointer auf einen Pointer auf ein Array der Mächtigkeit 2, 3 von Structs des Typs st**“ lesen, also genau anders herum, als man den Ausdruck `struct st *(*var[3][2])` mit der **Spiralregel** lesen würde. Bei der **Spiralregel** fängt man beim Ausdruck `struct st *(*var[3][2])` bei der Variable `var` an und arbeitet sich dann auf „**Spiralbahnen**“, von **innen-nach-außen** durch den Ausdruck, um herauszufinden, dass dieser Datentyp ein „**Array der Mächtigkeit 3, 2 von Pointern auf einen Pointer auf einen Struct vom Typ st**“ ist.

<sup>14</sup>In der Informatik wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**, bzw. in diesem Beispiel von **links-nach-rechts**.



Abbildung 1.4: *Abstract Syntax Tree Generierung ohne Umdrehen*

Dieser **Abstract Syntax Tree** ist für die Weiterverarbeitung ungeeignet, denn für die **Adressberechnung** für eine Aneinanderreihung von Zugriffen auf **Pointerelemente**, **Arrayelemente** oder **Structattribute**, welche in Unterkapitel ?? genauer erläutert wird, will man den Datentyp in **umgekehrter Reihenfolge**. Aus diesem Grund muss der **Transformer** bei der Konstruktion des **Abstract Syntax Tree** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstract Syntax Tree** mit **richtig rum gedrehtem Datentyp**, wie in Abbildung 1.5 zustande, der für die Weiterverarbeitung geeignet ist.

Abbildung 1.5: *Abstract Syntax Tree Generierung mit Umdrehen*

Die Weiterverarbeitung des **Abstract Syntax Trees** geschieht mithilfe von **Passes**, welche im Unterkapitel ?? genauer beschrieben werden. Da die Knoten des **Abstract Syntax Tree** anders als beim **Derivation Tree** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Grammatik der **Konkreten Syntax**

ist es in den folgenden Unterkapiteln 1.2.5.1, 1.2.5.2 und 1.2.5.3 notwendig die **Bedeutung** der einzelnen **PicoC-Knoten**, **RETI-Knoten** und bestimmter **Kompositionen** dieser Knoten zu **dokumentieren**, die alle in den unterschiedlichen von den **Passes** umgeformten **Abstract Syntax Trees** vorkommen.

Des Weiteren gibt die **Abstrakte Syntax** die durch die Grammatik 1.2.1 in Unterkapitel 1.2.5.4 beschrieben wird aufschluss darüber welche **Kompositionen von PicoC-Knoten**, neben den bereits in Tabelle 1.2.10 definierten Kompositionen mit Bedeutung insgesamt überhaupt **möglich** sind.

### 1.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache  $L_{PicoC}$  darstellen. Für die **PicoC-Knoten** wurden möglichst **kurze** und **leicht** verständliche Bezeichner gewählt, da auf diese Weise bei der Implementierung der einzelnen Passes möglichst **viel Code in eine Zeile** passt und dieser Code auch durch leicht verständliche Bezeichner von Knoten **intuitiv verständlich** sein sollte<sup>15</sup>. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstract Syntax Trees** vorkommen sind in Tabelle 1.3 mit einem **Beschreibungstext** dokumentiert.

<sup>15</sup>Z.B. steht der **PicoC-Knoten** `Name(str)` für einen **Bezeichner**. Anstatt diesen Knoten in englisch `Identifizier(str)` zu nennen, wurde dieser als `Name(str)` gewählt, da `Name(str)` **kürzer** ist und **intuitiver verständlich**.

PiocC-Knoten	Beschreibung
Name(val)	Ein <b>Bezeichner</b> , z.B. <code>my_fun</code> , <code>my_var</code> usw. , aber da es keine gute Kurzform für <code>Identifizier()</code> (englisches Wort für Bezeichner) gibt, wurde dieser Knoten <code>Name()</code> genannt.
Num(val)	Eine <b>Zahl</b> , z.B. 42, -3 usw.
Char(val)	Ein <b>Zeichen</b> der <b>ASCII-Zeichenkodierung</b> , z.B. <code>'c'</code> , <code>'*'</code> usw.
Minus(), Not(), DerefOp(), RefOp(), LogicNot()	Die <b>unären Operatoren</b> <code>un_op</code> : <code>-a</code> , <code>~a</code> , <code>*a</code> , <code>&amp;a !a</code> .
Add(), Sub(), Mul(), Div(), Mod(), Oplus(), And(), Or(), LogicAnd(), LogicOr()	Die <b>binären Operatoren</b> <code>bin_op</code> : <code>a + b</code> , <code>a - b</code> , <code>a * b</code> , <code>a / b</code> , <code>a % b</code> , <code>a ^ b</code> , <code>a &amp; b</code> , <code>a   b</code> , <code>a &amp;&amp; b</code> , <code>a    b</code> .
Eq(), NEq(), Lt(), LtE(), Gt(), GtE()	Die <b>Relationen</b> <code>rel</code> : <code>a == b</code> , <code>a != b</code> , <code>a &lt; b</code> , <code>a &lt;= b</code> , <code>a &gt; b</code> , <code>a &gt;= b</code> .
Const(), Writeable()	Die <b>Type Qualifier</b> <code>type_qual</code> : <code>const</code> , was für ein <b>nicht beschreibbare Konstante</b> steht und das <b>nicht</b> Angeben von <code>const</code> , was für einen <b>beschreibbare</b> Variable steht.
IntType(), CharType(), VoidType()	Die <b>Type Specifier</b> für <b>Primitiven Datentypen</b> , die in der Abstrakten Syntax, um eine intuitive Bezeichnung zu haben einfach nur unter <b>Datentypen</b> <code>datatype</code> eingeordnet werden: <code>int</code> , <code>char</code> , <code>void</code> .
Placeholder()	<b>Platzhalter</b> für einen Knoten, der diesen später <b>ersetzt</b> .
BinOp(exp, bin_op, exp)	Container für eine <b>binäre Operation</b> mit 2 Expressions: <code>&lt;exp1&gt; &lt;bin_op&gt; &lt;exp2&gt;</code>
UnOp(un_op, exp)	Container für eine <b>unäre Operation</b> mit einer Expression: <code>&lt;un_op&gt; &lt;exp&gt;</code> .
Exit(num)	Container für einen <b>Exit Code</b> , der vor der Beendigung in das ACC Register geschrieben wird und steht für die <b>Beendigung</b> des laufenden Programmes.
Atom(exp, rel, exp)	Container für eine <b>binäre Relation</b> mit 2 Expressions: <code>&lt;exp1&gt; &lt;rel&gt; &lt;exp2&gt;</code>
ToBool(exp)	Container für einen <b>Arithmetischen Ausdruck</b> , wie z.B. <code>1 + 3</code> oder einfach nur <code>3</code> , der nicht nur 1 oder 0 als Ergebnis haben kann und daher bei einem Ergebnis <code>x &gt; 1</code> auf 1 abgebildet wird.
Alloc(type_qual, datatype, name, local_var_or_param)	<b>Container</b> für eine <b>Allokation</b> <code>&lt;type_qual&gt; &lt;datatype&gt; &lt;name&gt;</code> mit den notwendigen Knoten <code>type_qual</code> , <code>datatype</code> und <code>name</code> , die alle für einen Eintrag in der <b>Symboltabelle</b> notwendigen Informationen enthalten. Zudem besitzt er ein verstecktes Attribut <code>local_var_or_param</code> , dass die Information trägt, ob es sich bei der <b>Variable</b> um eine <b>Lokale Variable</b> oder einen <b>Parameter</b> handelt.
Assign(lhs, exp)	Container für eine <b>Zuweisung</b> , wobei <code>lhs</code> ein <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> , <code>Attr(exp, name)</code> oder <code>Name('var')</code> sein kann und <code>exp</code> ein beliebiger <b>Logischer Ausdruck</b> sein kann: <code>lhs = exp</code> .

Tabelle 1.3: PicoC-Knoten Teil 1

PiocC-Knoten	Beschreibung
Exp(exp, datatype, error_data)	Container für einen <b>beliebigen Ausdruck</b> , dessen Ergebnis auf den <b>Stack</b> soll. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Stack(num)	Container, der für das <b>temporäre</b> Ergebnis einer Berechnung, das <b>num</b> Speicherzellen relativ zum <b>Stackpointer Register SP</b> steht.
Stackframe(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Begin-Aktive-Funktion Register BAF</b> steht.
Global(num)	Container, der für eine Variable steht, die <b>num</b> Speicherzellen relativ zum <b>Datensegment Register DS</b> steht.
StackMalloc(num)	Container, der für das <b>Allokieren</b> von <b>num</b> Speicherzellen auf dem <b>Stack</b> steht.
PntrDecl(num, datatype)	Container, der für den <b>Pointerdatatype</b> steht: <code>&lt;prim_dt&gt; *&lt;var&gt;</code> , wobei das <b>Attribut</b> <b>num</b> die <b>Anzahl zusammengefasster Pointer</b> angibt und <b>datatype</b> der Datentyp ist, auf den der oder die <b>Pointer</b> zeigen.
Ref(exp, datatype, error_data)	Container, der für die Anwendung des <b>Referenz-Operators</b> <code>&amp;&lt;var&gt;</code> steht und die <b>Adresse</b> einer <b>Location</b> (Definition ??) auf den Stack schreiben soll, die über <b>exp</b> eingegrenzt wird. Zudem besitzt er 2 versteckte Attribute, wobei <b>datatype</b> im <b>RETI Blocks Pass</b> wichtig ist und <b>error_data</b> für <b>Fehlermeldungen</b> wichtig ist.
Deref(lhs, exp)	Container für den <b>Indezzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> , <code>Attr(exp, name)</code> oder ein <code>Name('var')</code> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
ArrayDecl(nums, datatype)	Container, der für den <b>Arraydatatype</b> steht: <code>&lt;prim_dt&gt; &lt;var&gt;[&lt;i&gt;]</code> , wobei das <b>Attribut</b> <b>nums</b> eine Liste von <code>Num('x')</code> ist, die die <b>Dimensionen</b> des Arrays angibt und <b>datatype</b> der Datentyp ist, der über das Anwenden von <code>Subscript()</code> auf das Array zugreifbar ist.
Array(exps, datatype)	Container für den <b>Initializer</b> eines <b>Arrays</b> , dessen Einträge <b>exps</b> weitere Initializer für eine <b>Array-Dimension</b> oder ein Initializer für ein <b>Struct</b> oder ein <b>Logischer Ausdruck</b> sein können, z.B. <code>{{1, 2}, {3, 4}}</code> . Des Weiteren besitzt er ein verstecktes Attribut <b>datatype</b> , welches für den <b>PicoC-ANF Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
Subscr(exp1, exp2)	Container für den <b>Indezzugriff</b> auf einen <b>Array-</b> oder <b>Pointerdatatype</b> : <code>&lt;var&gt;[&lt;i&gt;]</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> sein kann und <b>exp2</b> der Index ist auf den zugegriffen werden soll.
StructSpec(name)	Container für einen selbst definierten <b>Structdatatype</b> : <code>struct &lt;name&gt;</code> , wobei das <b>Attribut</b> <b>name</b> festlegt, welchen <b>selbst definierte</b> Structdatatype dieser Container-Knoten repräsentiert.
Attr(exp, name)	Container für den <b>Attributzugriff</b> auf einen <b>Structdatatype</b> : <code>&lt;var&gt;.&lt;attr&gt;</code> , wobei <b>exp1</b> eine angehängte weitere <code>Subscr(exp1, exp2)</code> , <code>Deref(exp1, exp2)</code> oder <code>Attr(exp, name)</code> Operation sein kann oder ein <code>Name('var')</code> sein kann und <b>name</b> das Attribut ist, auf das zugegriffen werden soll.

Tabelle 1.4: PicoC-Knoten Teil 2

PiocC-Knoten	Beschreibung
Struct(assigns, datatype)	Container für den <b>Initializer</b> eines <b>Structs</b> , z.B. {.<attr1>={1, 2}, .<attr2>={3, 4}}, dessen Eintrag assigns eine Liste von Assign(lhs, exp) ist mit einer Zuordnung eines <b>Attributezeichners</b> , zu einem weiteren Initializer für eine <b>Array-Dimension</b> oder zu einem Initializer für ein <b>Struct</b> oder zu einem <b>Logischen Ausdruck</b> . Des Weiteren besitzt er ein verstecktes Attribut datatype, welches für den <b>PicoC-ANF Pass</b> Informationen transportiert, die für <b>Fehlermeldungen</b> wichtig sind.
StructDecl(name, allocs)	Container für die Deklaration eines <b>selbstdefinierten Structdatentyps</b> , z.B. struct <var> {<datatype> <attr1>; <datatype> <attr2>;}, wobei name der <b>Bezeichner</b> des Structdatentyps ist und allocs eine Liste von Bezeichnern der <b>Attribute</b> des Structdatentyps mit dazugehörigem <b>Datentyp</b> , wofür sich der <b>Container-Knoten</b> Alloc(type_qual, datatype, name) sehr gut als <b>Container</b> eignet.
If(exp, stmts)	Container für ein <b>If Statement</b> if(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
IfElse(exp, stmts1, stmts2)	Container für ein <b>If-Else Statement</b> if(<exp>) { <stmts2> } else { <stmts2> } inklusive <b>Condition</b> exp und 2 <b>Branches</b> stmts1 und stmts2, die zwei Alternativen darstellen in denen jeweils <b>Listen von Statements</b> oder GoTo(Name('block.xyz'))'s stehen können.
While(exp, stmts)	Container für ein <b>While-Statement</b> while(<exp>) { <stmts> } inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
DoWhile(exp, stmts)	Container für ein <b>Do-While-Statement</b> do { <stmts> } while(<exp>); inklusive <b>Condition</b> exp und einem <b>Branch</b> stmts, indem eine <b>Liste von Statements</b> stehen kann oder ein einzelnes GoTo(Name('block.xyz')).
Call(name, exps)	Container für einen <b>Funktionsaufruf</b> : fun_name(exps), wobei name der <b>Bezeichner</b> der Funktion ist, die aufgerufen werden soll und exps eine <b>Liste von Argumenten</b> ist, die an die Funktion übergeben werden soll.
Return(exp)	Container für ein <b>Return-Statement</b> : return <exp>, wobei das <b>Attribut</b> exp einen <b>Logischen Ausdruck</b> darstellt, dessen Ergebnis vom <b>Return-Statement</b> zurückgegeben wird.
FunDecl(datatype, name, allocs)	Container für eine <b>Funktionsdeklaration</b> , z.B. <datatype> <fun_name>(<datatype> <param1>, <datatype> <param2>), wobei datatype der <b>Rückgabewert</b> der Funktion ist, name der <b>Bezeichner</b> der Funktion ist und allocs die <b>Parameter</b> der Funktion sind, wobei der <b>Container-Knoten</b> Alloc(type_spec, datatype, name) als Container für die Parameter dient.

Tabelle 1.5: PicoC-Knoten Teil 3

PiocC-Knoten	Beschreibung
FunDef(datatype, name, allocs, stmts.blocks)	Container für eine <b>Funktionsdefinition</b> , z.B. <datatype> <fun.name>(<datatype> <param>) {<stmts>}, wobei datatype der <b>Rückgabewert</b> der Funktion ist, name der <b>Bezeichner</b> der Funktion ist, allocs die <b>Parameter</b> der Funktion sind, wobei der <b>Container-Knoten</b> Alloc(type_spec, datatype, name) als Container für die Parameter dient und stmts.blocks eine Liste von <b>Statements</b> bzw. <b>Blöcken</b> ist, welche diese Funktion beinhaltet.
NewStackframe(fun_name, goto_after_call)	Container für die <b>Erstellung</b> eines neuen <b>Stackframes</b> und Speicherung des Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und der <b>Rücksprungadresse</b> nacheinander an den <b>Anfang</b> des neuen <b>Stackframes</b> . Das Attribut fun_name steht dabei für den Bezeichner der Funktion, für die ein neuer <b>Stackframe</b> erstellt werden soll. Das Attribut fun_name dient später dazu den <b>Block</b> dieser Funktion zu finden, weil dieser für den weiteren Kompilierungsvorgang wichtige Information in seinen versteckten Attributen gespeichert hat. Des Weiteren ist das Attribut goto_after_call ein GoTo(Name('addr@next_instr')), welches später durch die <b>Adresse</b> des Befehls, der direkt auf die <b>Jump Instruction</b> folgt, ersetzt wird.
RemoveStackframe()	Container für das <b>Entfernen</b> des aktuellen <b>Stackframes</b> durch das <b>Wiederherstellen</b> des im noch <b>aktuellen Stackframe</b> gespeicherten Werts des BAF-Registers der <b>aufgerufenen Funktion</b> und das Setzen des SP-Registers auf den Wert des BAF-Registers vor der Wiederherstellung.
File(name, decls_defs.blocks)	Container für alle <b>Funktionen</b> oder <b>Blöcke</b> , welche eine Datei als Ursprung haben, wobei name der <b>Dateiname</b> der Datei ist, die erstellt wird und decls_defs.blocks eine Liste von <b>Funktionen</b> bzw. <b>Blöcken</b> ist.
Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)	Container für <b>Statements</b> , der auch als <b>Block</b> bezeichnet wird, wobei das Attribut name der Bezeichner des <b>Labels</b> (Definition 1.4) des Blocks ist und stmts_instrs eine <b>Liste von Statements</b> oder <b>Instructions</b> . Zudem besitzt er noch 3 versteckte Attribute, wobei instrs_before die Zahl der <b>Instructions</b> vor diesem <b>Block</b> zählt, num_instrs die Zahl der Instructions ohne Kommentare in diesem Block zählt, param_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>Parameter</b> der Funktion belegt werden müssen und local_vars_size die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die <b>lokalen Variablen</b> der Funktion belegt werden müssen.
GoTo(name)	Container für ein <b>Goto</b> zu einem anderen <b>Block</b> , wobei das Attribut name der Bezeichner des <b>Labels</b> des Blocks ist zu dem Gesprungen werden soll.
SingleLineComment(prefix, content)	Container für einen <b>Kommentar</b> , den der Compiler selber während des <b>Kompilierungsvorgangs</b> erstellt, der im <b>RETI-Interpreter</b> selbst später <b>nicht</b> sichtbar sein wird, aber in den <b>Immediate-Dateien</b> , welche die <b>Abstract Syntax Trees</b> nach den verschiedenen <b>Passes</b> enthalten.
RETIComment(value)	Container für einen <b>Kommentar</b> im Code der Form: // # comment, der im <b>RETI-Interpreter</b> später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer <b>RETI-CPU nicht umsetzbar</b> ist und auch nicht sinnvoll wäre umzusetzen. Der <b>Kommentar</b> ist im Attribut <b>value</b> , welches jeder Knoten besitzt gespeichert.

Tabelle 1.6: PicoC-Knoten Teil 4

**Definition 1.4: Label**

Durch einen *Bezeichner eindeutig* zuordenbares *Sprungziel* im Programmcode.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

Die ausgegrauten Attribute der PicoC-Nodes sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Nodes** mit einem Wert **initialisiert** werden, sondern im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen bekommen, dass im weiteren Kompilierungsvorgang **Informationen** transportiert, die später im Kompilierungsvorgang nicht mehr so leicht zugänglich wären.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** **value** und **position**, wobei **value** bei einem **Token-Knoten** (Definition 1.5) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.6) unbesetzt ist. Das **Attribut position** wird später für Fehlermeldungen gebraucht.

**Definition 1.5: Token-Knoten**

Ersetzt ein **Token** bei der Generierung des **Abstract Syntax Tree**, damit der Zugriff auf Knoten des **Abstract Syntax Tree** möglichst **simpel** ist und keine vermeidbaren Fallunterscheidungen gemacht werden müssen.

**Token-Knoten** entsprechen im **Abstract Syntax Tree** **Blättern**.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.

**Definition 1.6: Container-Knoten**

Dient als **Container** für andere **Container-Knoten** und **Token-Knoten**. Die **Container-Knoten** werden optimalerweise immer so gewählt, dass sie **mehrere Produktionen der Konkreten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.<sup>a</sup>

**Container-Knoten** entsprechen im **Abstract Syntax Tree** **Inneren Knoten**.<sup>b</sup>

<sup>a</sup>Wie z.B. die verschiedenen **Arithmetischen Ausdrücke**, wie z.B. **1 % 3** und **Logischen Ausdrücke**, wie z.B. **1 && 2 < 3**, die einen gleichen Aufbau haben mit immer einer **Operation** in der Mitte haben und 2 **Operanden** auf beiden Seiten und sich unter dem Überbegriff **Binäre Operationen** zusammenfassen lassen.

<sup>b</sup>Thiemann, „Compilerbau“.

**1.2.5.2 RETI-Knoten**

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen **Ausdruck** aus der Sprache  $L_{RETI}$  darstellen. Für die **RETI-Knoten** wurden aus bereits in Unterkapitel 1.2.5.1 erläuterten Grund, genauso wie für die **RETI-Knoten** möglichst **kurze** und **leicht** verständliche Bezeichner gewählt. Alle **RETI-Knoten**, die in den von den verschiedenen Passes generierten **Abstract Syntax Trees** vorkommen sind in Tabelle 1.2.5.1 mit einem **Beschreibungstext** dokumentiert.



RETI-Knoten	Beschreibung
Program(name, instrs)	Container für alle <b>Instructions</b> : <name> <instrs>, wobei <b>name</b> der <b>Dateiname</b> der Datei ist, die erstellt wird und <b>instrs</b> eine <b>Liste von Instructions</b> ist.
Instr(op, args)	Container für eine <b>Instruction</b> : <op> <args>, wobei <b>op</b> eine <b>Operation</b> ist und <b>args</b> eine <b>Liste von Argumenten</b> für dieser Operation.
Jump(rel, im_goto)	Container für eine <b>Jump-Instruction</b> : JUMP<rel> <im>, wobei <b>rel</b> eine <b>Relation</b> ist und <b>im_goto</b> ein <b>Immediate Value</b> <b>Im(val)</b> für die <b>Anzahl an Speicherzellen</b> , um die relativ zur <b>Jump-Instruction</b> gesprungen werden soll oder ein <b>GoTo(Name('block.xyz'))</b> , das später im <b>RETI-Patch Pass</b> durch einen passenden <b>Immediate Value</b> ersetzt wird.
Int(num)	Container für einen <b>Interruptaufruf</b> : INT <im>, wobei <b>num</b> die <b>Interruptvektornummer</b> (IVN) für die passende Speicherzelle in der <b>Interruptvektortabelle</b> ist, in der die Adresse der <b>Interrupt-Service-Routine</b> (ISR) steht.
Call(name, reg)	Container für einen <b>Prozeduraufruf</b> : CALL <name> <reg>, wobei <b>name</b> der <b>Bezeichner</b> der Prozedur, die aufgerufen werden soll ist und <b>reg</b> ein <b>Register</b> ist, das als <b>Argument</b> an die Prozedur dient. Diese <b>Operation</b> ist in der Betriebssysteme Vorlesung <sup>a</sup> nicht deklariert, sondern wurde dazuerfunden, um unkompliziert ein CALL PRINT ACC oder CALL INPUT ACC im RETI-Interpreter simulieren zu können.
Name(val)	Bezeichner für eine <b>Prozedur</b> , z.B. PRINT oder INPUT oder den <b>Programmnamen</b> , z.B. PROGRAMNAME. Dieses <b>Argument</b> ist in der Betriebssysteme Vorlesung <sup>a</sup> nicht deklariert, sondern wurde dazuerfunden, um Bezeichner, wie PRINT, INPUT oder PROGRAMNAME schreiben zu können.
Reg(reg)	Container für ein <b>Register</b> .
Im(val)	Ein <b>Immediate Value</b> , z.B. 42, -3 usw.
Add(), Sub(), Mult(), Div(), Mod(), Oplus(), Or(), And()	<b>Compute-Memory</b> oder <b>Compute-Register</b> Operationen: ADD, SUB, MULT, DIV, OPLUS, OR, AND.
Addi(), Subi(), Multi(), Divi(), Modi(), Oplusi(), Ori(), Andi()	<b>Compute-Immediate</b> Operationen: ADDI, SUBI, MULTI, DIVI, MODI, OPLUSI, ORI, ANDI.
Load(), Loadin(), Loadi()	<b>Load</b> Operationen: LOAD, LOADIN, LOADI.
Store(), Storein(), Move()	<b>Store</b> Operationen: STORE, STOREIN, MOVE.
Lt(), LtE(), Gt(), GtE(), Eq(), NEq(), Always(), NOP()	<b>Relationen</b> : <, <=, >, >=, ==, !=, _NOP.
Rti()	<b>Return-From-Interrupt</b> Operation: RTI.
Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()	<b>Register</b> : PC, IN1, IN2, ACC, SP, BAF, CS, DS.

<sup>a</sup> Scholl, „Betriebssysteme“

Tabelle 1.7: RETI-Knoten

### 1.2.5.3 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

In Tabelle 1.8 sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben und nicht bereits in der **Abstrakten Syntax 1.2.8** enthalten sind.



Komposition	Beschreibung
Ref(Global(Num('addr')))	Speichert <b>Adresse</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Datensegment Register</b> DS steht auf den <b>Stack</b> .
Ref(Stackframe(Num('addr')))	Speichert <b>Adresse</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF steht auf den <b>Stack</b> .
Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))))	Berechnet die nächste <b>Adresse</b> aus der <b>Adresse</b> , die an Speicherzelle Stack(Num('addr1')) steht und dem <b>Subscript Index</b> , der an Speicherzelle Stack(Num('addr2')) steht und speichert diese auf den <b>Stack</b> . Die Berechnung ist abhängig davon ob der <b>Datentyp</b> ArrayDecl(datatype) oder PntrDecl(datatype) ist. Der <b>Datentyp</b> ist ein verstecktes Attribut von Ref(exp).
Ref(Attr(Stack(Num('addr1')), Name('attr')))	Berechnet die nächste <b>Adresse</b> aus der <b>Adresse</b> , die an Speicherzelle Stack(Num('addr1')) steht und dem <b>Attributnamen</b> Name('attr') und speichert diese auf den <b>Stack</b> . Zur Berechnung ist der Name des <b>Struct</b> in StructSpec(Name('st')) notwendig, dessen <b>Attribut</b> Name('attr') ist. StructSpec(Name('st')) ist ein verstecktes Attribut von Ref(exp).
Assign(Stack(Num('size')), Global(Num('addr')))	Schreibt Num('size') viele Speicherzellen, die ab Global(Num('addr')) relativ zum <b>Datensegment Register</b> DS stehen, versetzt genauso auf den <b>Stack</b> .
Assign(Stack(Num('size')), Stackframe(Num('addr')))	Schreibt Num('size') viele Speicherzellen, die ab Stackframe(Num('addr')) relativ zum <b>Begin-Aktive-Funktion Register</b> BAF stehen, versetzt genauso auf den <b>Stack</b> .
Exp(Global(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Datensegment Register</b> DS steht auf den <b>Stack</b> .
Exp(Stackframe(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF steht auf den <b>Stack</b> .
Exp(Stack(Num('addr')))	Speichert <b>Inhalt</b> der Speicherzelle, die Num('addr') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht auf den <b>Stack</b> .
Assign(Stack(Num('addr1')), Stack(Num('addr2')))	Speichert <b>Inhalt</b> der Speicherzelle Stack(Num('addr2')), die Num('addr2') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht an der <b>Adresse</b> in der Speicherzelle, die Num('addr1') Speicherzellen <b>relativ</b> zum <b>Stackpointer Register</b> SP steht.
Assign(Global(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem <b>Stack</b> stehen, versetzt genauso auf die Speicherzellen ab Num('addr') <b>relativ</b> zum <b>Datensegment Register</b> DS.
Assign(Stackframe(Num('addr')), Stack(Num('size')))	Schreibt Num('size') viele Speicherzellen, die auf dem <b>Stack</b> stehen, versetzt genauso auf die Speicherzellen ab Num('addr') <b>relativ</b> zum <b>Begin-Aktive-Funktion Register</b> BAF.
Exp(Reg(reg))	Schreibt den aktuellen Wert des <b>Registers</b> reg auf den <b>Stack</b> .
Instr(Loadi(), [Reg(Acc()), GoTo(Name('addr@next_instr'))])	Lädt in das Register ACC die <b>Adresse</b> der Instruction, die in diesem Kontext direkt nach dem Sprung zum Block einer anderen Funktion steht.

Tabelle 1.8: Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung

Um die obige Tabelle 1.8 nicht mit unnötig viel repetitiven Inhalt zu füllen, wurden die zahlreichen Kompositionen **ausgelassen**, bei denen einfach nur **exp** durch **Stack(Num('x'))**,  $x \in \mathbb{N}$  ersetzt wurde.

Zudem sind auch jegliche Kombinationen ausgelassen, bei denen einfach nur eine **Expression** an ein **Exp(exp)** bzw. **Ref(exp)** drangehängt wurde.

#### 1.2.5.4 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache  $L_{PicoC}$  wird durch die Grammatik 1.2.10 beschrieben.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i>   <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>BinOp(&lt;exp&gt;, &lt;bin_op&gt;, &lt;exp&gt;)</i>   <i>UnOp(&lt;un_op&gt;, &lt;exp&gt;)</i>   <i>Call(Name('input'), Empty())</i>   <i>Call(Name('print'), &lt;exp&gt;)</i>	
<i>stmt</i>	::=	<i>Exp(&lt;exp&gt;)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(&lt;exp&gt;, &lt;rel&gt;, &lt;exp&gt;)</i>   <i>ToBool(&lt;exp&gt;)</i>	
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(&lt;type_qual&gt;, &lt;datatype&gt;, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(&lt;exp&gt;, &lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), &lt;datatype&gt;)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref(&lt;exp&gt;, &lt;exp&gt;)</i>   <i>Ref(&lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, &lt;datatype&gt;)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(&lt;exp&gt;, &lt;exp&gt;)</i>   <i>Array(&lt;exp&gt;+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(&lt;exp&gt;, Name(str))</i>   <i>StructAssign(Name(str), &lt;exp&gt;+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>IfElse(&lt;exp&gt;, &lt;stmt&gt;*, &lt;stmt&gt;*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>DoWhile(&lt;exp&gt;, &lt;stmt&gt;*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), &lt;exp&gt;*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(&lt;exp&gt;)</i>	
<i>decl_def</i>	::=	<i>FunDecl(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))*)</i>   <i>FunDef(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))* &lt;stmt&gt;*)</i>	
<i>file</i>	::=	<i>File(Name(str), &lt;decl_def&gt;*)</i>	<i>L_File</i>

**Grammar 1.2.10:** Abstrakte Syntax der Sprache  $L_{PicoC}$

Man spricht hier von der „**Abstrakten Syntax der Sprache  $L_{PicoC}$** “ und meint hier mit der Sprache  $L_{PicoC}$  **nicht** die Sprache, welche durch die **Abstrakte Syntax** beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck die **Abstrakt Syntax** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakt Syntax** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Redeart** wurde aus der **Quelle** G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

### 1.2.5.5 Codebeispiel

In Code 1.5 ist der **Abstract Syntax Tree** zu sehen, der aus dem **vereinfachten Derivation Tree** aus Code 1.4 mithilfe eines **Transformers** generiert wurde.

```

1 File
2   Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
3   [
4     StructDecl
5       Name 'st',
6       [
7         Alloc
8           Writeable,
9           PtrDecl
10            Num '1',
11            ArrayDecl
12              [
13                Num '4',
14                Num '5'
15              ],
16            PtrDecl
17              Num '1',
18              IntType 'int',
19            Name 'attr'
20          ],
21      FunDef
22        VoidType 'void',
23        Name 'main',
24        [],
25        [
26          Exp
27            Alloc
28              Writeable,
29              ArrayDecl
30                [
31                  Num '3',
32                  Num '2'
33                ],
34              PtrDecl
35                Num '1',
36              PtrDecl
37                Num '1',
38              StructSpec
39                Name 'st',
40              Name 'var'
41            ]
42          ]

```

**Code 1.5:** Aus vereinfachtem Derivation Tree generierter Abstract Syntax Tree**1.2.5.6 Ausgabe des Abstract Syntax Tree**

Ein **Knoten** eines **Abstract Syntax Tree** kann entweder in der **Konkreter Syntax** der Sprache, für dessen Kompilierung er generiert wurde oder in der **Abstrakter Syntax**, die beschreibt, wie der Abstract Syntax Tree selbst aufgebaut sein darf ausgegeben werden.

Das Ausgeben eines **Abstract Syntax Trees** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`<sup>16</sup> der Programmiersprache **Python** umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkreten Syntax** aufgebaute **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **runden öffnenden** ( und **schließenden** ) **Klammern**, sowie **Kommas** ', ', **Semikolons** ; usw. zur Darstellung der **Hierarchie** und zur **Abtrennung** zurück. Dabei wird nach dem **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** durchlaufen und die Magische `__repr__()`-Methode der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methode ihrer Kinder aufrufen und die zurückgegebene **Textrepräsentation** passend **zusammenfügen** und selbst **zurückgeben**.

Im **PicoC-Compiler** wurden **Abstrakte** und **Konkrete Syntax** miteinander gemischt. Für **PicoC-Knoten** wurde die **Abstrakte Syntax** verwendet, da Passes schließlich auf **Abstract Syntax Trees** operieren. Bei **RETI-Knoten** wurde die **Konkrete Syntax** verwendet, da **Maschinenbefehle** in **Konkreter Syntax** schließlich das **Endprodukt** des Kompiliervorgangs sein sollen. Da die **Abstrakte Syntax** von **RETI-Knoten** so simpel ist, macht es kaum einen Unterschied in der Erkennbarkeit, bis auf fehlende geschiefte Klammern ( ) usw., ob man die **RETI-Knoten** in **Abstrakter** oder **Konkreter Syntax** schreibt. Daher kann man auch einfach gleich die **RETI-Knoten** in **Konkreter Syntax** ausgeben und muss nicht beim letzten **Pass** daran denken, am Ende die **Konkrete**, statt der **Abstrakten Syntax** für die **RETI-Knoten** auszugeben.

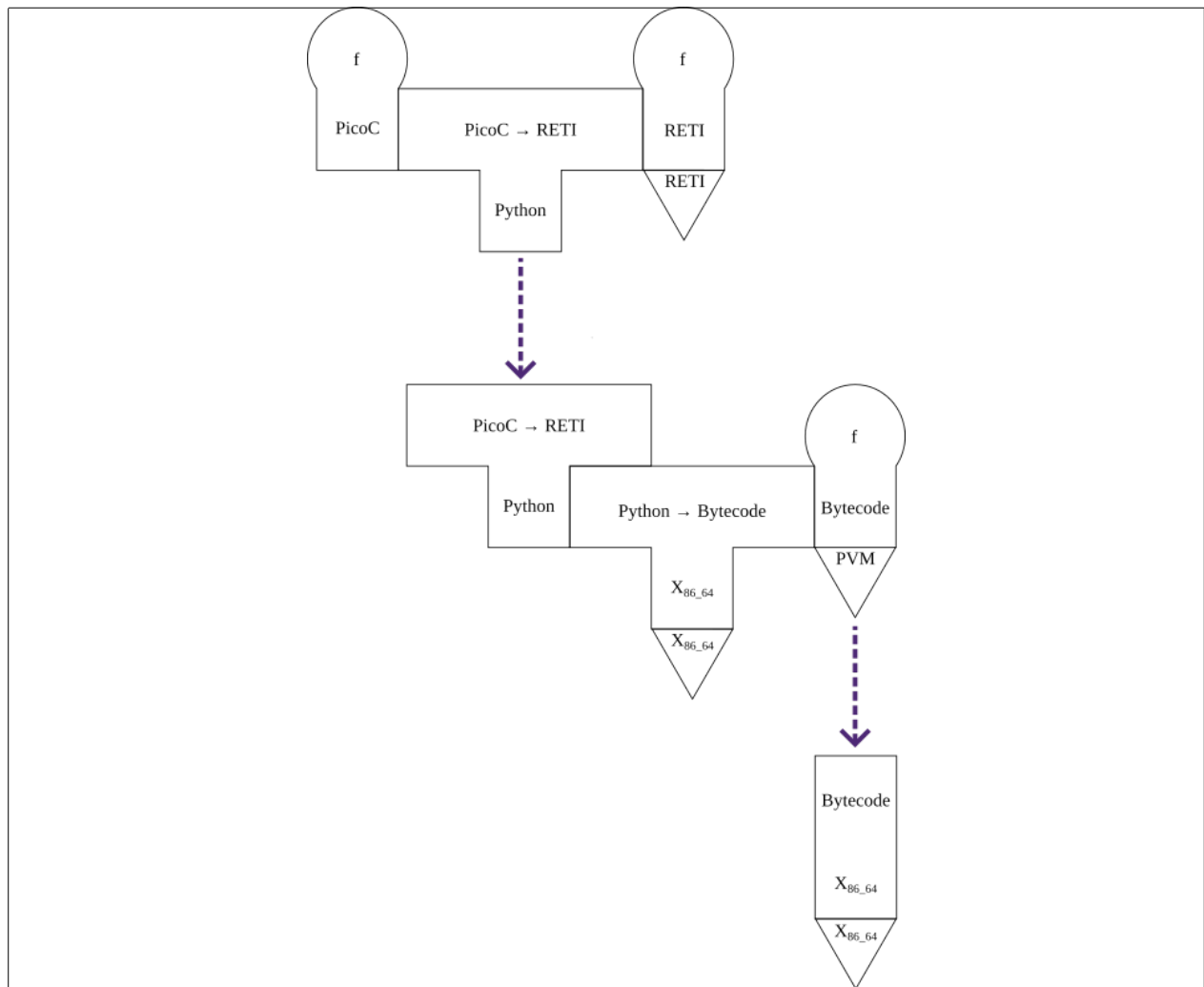
**1.3 Code Generierung**

Nach der Generierung eines **Abstract Syntax Tree** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel ??, wird in diesem Kapitel mit den verschiedenen **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** im **Abstract Syntax Tree** als Basis das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstract Syntax Tree** der Sprache  $L_{PicoC}$ , der durch die **Abstrakte Syntax** in Grammatik 1.2.10 spezifiziert ist in einen entsprechenden **Abstract Syntax Tree** der Sprache  $L_{RETI}$  umzuformen. Das ganze lässt sich, wie in Unterkapitel ?? bereits beschrieben vereinfachen, indem man dieses Problem in mehrere **Passes** (Definition ??) herunterbricht.

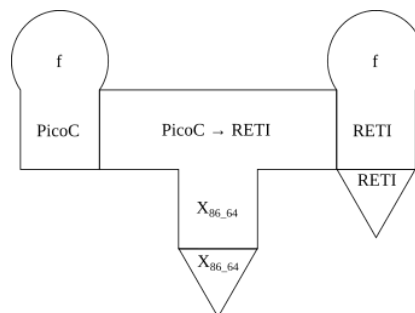
Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** (Definiton ??). Damit **RETI-Code** erzeugt werden kann, der auf der **RETI-Architektur** läuft, muss erst, wie im **T-Diagramm** (siehe Unterkapitel ??) in Abbildung 1.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer  $X_{86,64}$ -Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer  $X_{86,64}$ -Architektur laufen könnte. Und selbst dieses **T-Diagramm** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu  $X_{86,64}$  kompiliert wurde usw.

<sup>16</sup>Spezielle Methode, die immer aufgerufen wird, wenn das **Object**, dass in Besitz dieser Methode ist als **String** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.



**Abbildung 1.6:** *Cross-Compiler Kompilervorgang ausgeschrieben*

Dieses längliche **T-Diagramm** in Abbildung 1.6 lässt sich zusammenfassen, sodass man das **T-Diagramm** in Abbildung 1.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in X<sub>86\_64</sub>-Maschinensprache geschrieben ist.



**Abbildung 1.7:** *Cross-Compiler Kompilervorgang Kurzform*

Nachdem der Kompilierprozess des **PicoC-Compiler** im **vertikalen** nun genauer angesehen wurde, wird

der Kompilierprozess im Folgenden im **horizontalen**, auf der Ebene der verschiedenen **Passes** genauer betrachtet. Die Abbildung 1.8 gibt einen guten Überblick über alle **Passes** und wie diese in der **Pipe-Architektur** (Definition ??) des **PicoC-Compilers** aufeinanderfolgen. In der **Pipe-Architektur** nutzt der jeweils nächste **Pass** den generierten **Abstract Syntax Tree** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstract Syntax Tree** in seiner eigenen **Sprache** zu generieren.

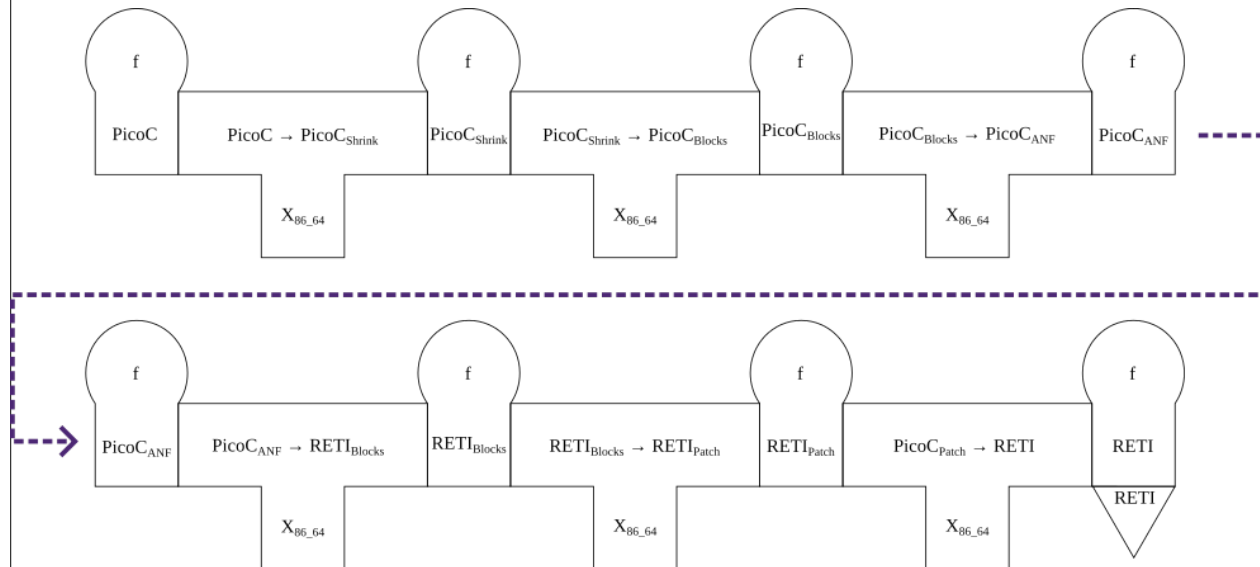


Abbildung 1.8: Architektur mit allen Passes ausgeschrieben

Im Unterkapitel 1.3.1 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 1.3.1 nicht ausreichend vertieft wurden. Viele der verwendeten **Ansätze** zur Lösung dieser Probleme basieren auf der Vorlesung Scholl, „Betriebssysteme“ und wurden in dieser Bachelorarbeit weiter ausgearbeitet, wo es nötig war, sodass diese mit dem **PicoC-Compiler** auch in der **Praxis** implementiert werden konnten.

Um die verschiedenen Aspekte besser erklären zu können, werden **Codebeispiele** verwendet, in welchen ein kleines repräsentatives **PicoC-Programm** für einen spezifischen Aspekt in wichtigen **Zwischenstadien der Kompilierung** gezeigt wird<sup>17</sup>. Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem **Link**<sup>18</sup> finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind<sup>19</sup>.

### 1.3.1 Passes

Im Folgenden werden die verschiedenen **Passes** des **PicoC-Compilers** für die Generierung von **RETI-Code** besprochen. Viele dieser **Passes** haben **Aufgaben**, die eher unter die Themenbereiche des **Bachelorprojekts** fallen. Allerdings ist das Verständnis der **Passes** auch für das Verständnis der verschiedenen Aspekte<sup>20</sup> der

<sup>17</sup>Also die verschiedenen in den **Passes** generierten **Abstract Syntax Trees**, sofern der **Pass** für den gezeigten Aspekt relevant ist.

<sup>18</sup>[https://github.com/matthejue/Bachelorarbeit/tree/master/code\\_examples](https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples).

<sup>19</sup>Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstract Syntax Trees** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.

<sup>20</sup>In kurz: **Pointer**, **Arrays**, **Structs** und **Funktionen**.

**Bachelorarbeit** wichtig.

Auf jedes Detail der einzelnen **Passes** wird in diesem Unterkapitel allerdings nicht eingegangen, da diese einerseits in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern, Arrays, Structs** und **Funktionen** im Detail erklärt sind und andererseits viele Aufgaben dieser **Passes** eher dem **Bachelorprojekt** zuzurechnen sind.

### 1.3.1.1 PicoC-Shrink Pass

#### 1.3.1.1.1 Aufgabe

Der Aufgabe des **PicoC-Shrink Pass** ist in Unterkapitel ?? ausführlich an einem Beispiel erklärt. Kurzgefasst hat der **PicoC-Shrink Pass** die Aufgabe, die Eigenheit auszunutzen, dass der **Dereferenzierungoperator** **\*pntr** und die damit einhergehende **Pointer Arithmetik** **\*(pntr + i)** sich in der Untermenge der Sprache  $L_C$ , welche die Sprache  $L_{PicoC}$  darstellt genau gleich verhält, wie der **Operator** für den **Zugriff** auf den **Index** eines **Arrays** **ar[i]**.

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** **Deref(exp, i)** im jeweiligen **Abstract Syntax Tree** in **Knoten** **Subscr(exp, i)** um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung vermeiden lassen. Man lässt die **Dereferenzierung** **\*(var + i)** einfach von den Routinen für einen **Zugriff auf einen Arrayindex** **var[i]** übernehmen.

#### 1.3.1.1.2 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache  $L_{PicoC\_Shrink}$  in Tabelle 1.3.1 ist fast **identisch** mit der **Abstrakten Syntax** der Sprache  $L_{PicoC}$  in Tabelle 1.2.10, nach welcher der **erste** Abstract Syntax Tree in der **Syntaktischen Analyse** generiert wurde. Der einzige **Unterschied** liegt darin, dass es den Knoten **Deref(exp, exp)** in Tabelle 1.3.1 **nicht** mehr gibt. Das liegt daran, dass dieser Pass alle **Vorkommnisse** des Knoten **Deref(exp, exp)** durch den Knoten **Subscr(exp, exp)** auswechselt, der ebenfalls bereits in der **Abstrakten Syntax** der Sprache  $L_{PicoC}$  definiert ist.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i>   <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>BinOp(&lt;exp&gt;, &lt;bin_op&gt;, &lt;exp&gt;)</i>   <i>UnOp(&lt;un_op&gt;, &lt;exp&gt;)</i>   <i>Call(Name('input'), Empty())</i>   <i>Call(Name('print'), &lt;exp&gt;)</i>	
<i>stmt</i>	::=	<i>Exp(&lt;exp&gt;)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(&lt;exp&gt;, &lt;rel&gt;, &lt;exp&gt;)</i>   <i>ToBool(&lt;exp&gt;)</i>	
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(&lt;type_qual&gt;, &lt;datatype&gt;, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(&lt;exp&gt;, &lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), &lt;datatype&gt;)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Deref(&lt;exp&gt;, &lt;exp&gt;)</i>   <i>Ref(&lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, &lt;datatype&gt;)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(&lt;exp&gt;, &lt;exp&gt;)</i>   <i>Array(&lt;exp&gt;+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(&lt;exp&gt;, Name(str))</i>   <i>StructAssign(Name(str), &lt;exp&gt;+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>IfElse(&lt;exp&gt;, &lt;stmt&gt;*, &lt;stmt&gt;*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>DoWhile(&lt;exp&gt;, &lt;stmt&gt;*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), &lt;exp&gt;*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(&lt;exp&gt;)</i>	
<i>decl_def</i>	::=	<i>FunDecl(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))*)</i>   <i>FunDef(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))**, &lt;stmt&gt;*)</i>	
<i>file</i>	::=	<i>File(Name(str), &lt;decl_def&gt;*)</i>	<i>L_File</i>

Grammar 1.3.1: Abstrakte Syntax der Sprache  $L_{Pioc\_Shrink}$ 

Der rot markierte Knoten bedeutet, dass dieser im Vergleich zur vorherigen **Abstrakten Syntax** nicht mehr da ist.

#### 1.3.1.1.3 Codebeispiel

In den nächsten Unterkapiteln wird das Beispiel in Code 1.6 zur **Anschauung** der verschiedenen **Passes** verwendet. Im Code 1.6 ist in der Funktion **faculty** ein **iterativer** Algorithmus implementiert, der die **Fakultät** eines übergebenen **Arguments** berechnet. Der Algorithmus basiert auf einem **Beispielprogramm**



aus der Vorlesung Scholl, „Betriebssysteme“, welcher in der Vorlesung allerdings **rekursiv** implementiert ist.

Dieser **rekursive** Algorithmus ist allerdings **kein** gutes **Anschauungsbeispiel**, dass viele der Aufgaben der verschiedenen **Passes** bei der Kompilierung veranschaulicht hätte. Viele Aufgaben der **Passes**, wie z.B. bei der Kompilierung von **if**-, **if-else**-, **while**- und **do-while**-Statements wären im Beispiel aus der Vorlesung **nicht** enthalten gewesen. Daher wurde das Beispiel aus der Vorlesung zu einem **iterativen** Algorithmus 1.6 umgeschrieben, um **if**- und **while**-Statemens zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter [Link<sup>21</sup>](#), unter den Testbezeichnungen `example_faculty_rec.picoc` und `example_faculty_it.picoc` zu finden.

Die Codebeispiele in diesem und den folgenden Unterkapiteln dienen allerdings nur als **Anschauung** des jeweiligen **Passes**, der in diesem Unterkapitel beschrieben wird und werden nicht im Detail erläutert, da viele Details der Passes später in den Unterkapiteln ??, ??, ?? und ?? zu **Pointern**, **Arrays**, **Structs** und **Funktionen** mit eigenen **Codebeispielen** erklärt werden und alle sonstigen Details dem **Bachelorprojekt** zuzurechnen sind.

```

1 // based on a example program from Christoph Scholl's Operating Systems lecture
2
3 int faculty(int n){
4     int res = 1;
5     while (1) {
6         if (n == 1) {
7             return res;
8         }
9         res = n * res;
10        n = n - 1;
11    }
12 }
13
14 void main() {
15     print(faculty(4));
16 }
```

**Code 1.6:** PicoC Code für Codebeispiel

In Code 1.7 sieht man den **Abstract Syntax Tree**, der in der **Syntaktischen Analyse** generiert wurde.

```

1 File
2   Name './example_faculty_it.ast',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
```

<sup>21</sup>[https://github.com/matthejue/PicoC-Compiler/tree/new\\_architecture/tests](https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests).

```

10  [
11    Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1')),
12    While
13      Num '1',
14      [
15        If
16          Atom(Name('n'), Eq('=='), Num('1')),
17          [
18            Return(Name('res'))
19          ]
20          Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
21          Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
22      ]
23  ],
24  FunDef
25    VoidType 'void',
26    Name 'main',
27    [],
28    [
29      Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
30    ]
31 ]

```

Code 1.7: Abstract Syntax Tree für Codebeispiel

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstract Syntax Tree** in Code 1.7, da das Codebeispiel keine **Dereferenzierung** enthält.

### 1.3.1.2 PicoC-Blocks Pass

#### 1.3.1.2.1 Aufgabe

Die Aufgabe des **PicoC-Blocks Passes** ist es die Knoten `If(exp, stmts)`, `IfElse(exp, stmts1, stmts2)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` mithilfe von `Block(name, stmts_instrs-)`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)-Knoten` umzusetzen. Der `IfElse(exp, stmts1, stmts2)-Knoten` wird zur Umsetzung der **Bedingung** verwendet und es wird, je nachdem, ob die Bedingung **wahr** oder **falsch** ist mithilfe der `GoTo(label)-Knoten` in einen von zwei **alternativen Branches** gesprungen oder ein **Branch** erneut aufgerufen usw.

#### 1.3.1.2.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache  $L_{PicoC\_Shrink}$  in Tabelle 1.3.1 um die Knoten zu erweitern, die im Unterkapitel 1.3.1.2.1 erwähnt wurden. Die Knoten `If(exp, stmts)`, `While(exp, stmts)` und `DoWhile(exp, stmts)` gibt es nicht mehr, da sie durch `Block(name, stmts_instrs-)`, `GoTo(label)-` und `IfElse(exp, stmts1, stmts2)-Knoten` ersetzt wurden. Die **Funktionsdefinition** `FunDef(<datatype>, Name(str), Alloc(Writable(), <datatype>, Name(str))* , <block>*)` ist nun ein Container für **Blöcke** `Block(Name(str), <stmt>*)` und keine Statements `stmt` mehr. Das resultiert in der **Abstrakten Syntax** der Sprache  $L_{PicoC\_Blocks}$  in Tabelle 1.3.2.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i>   <i>RETIComment()</i>	<i>L_Comment</i>
<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>	
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>BinOp(&lt;exp&gt;, &lt;bin_op&gt;, &lt;exp&gt;)</i>   <i>UnOp(&lt;un_op&gt;, &lt;exp&gt;)</i>   <i>Call(Name('input'), Empty())</i>   <i>Call(Name('print'), &lt;exp&gt;)</i>	
<i>stmt</i>	::=	<i>Exp(&lt;exp&gt;)</i>	
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>	
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>	
<i>exp</i>	::=	<i>Atom(&lt;exp&gt;, &lt;rel&gt;, &lt;exp&gt;)</i>   <i>ToBool(&lt;exp&gt;)</i>	
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>	
<i>exp</i>	::=	<i>Alloc(&lt;type_qual&gt;, &lt;datatype&gt;, Name(str))</i>	
<i>stmt</i>	::=	<i>Assign(&lt;exp&gt;, &lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>PntrDecl(Num(str), &lt;datatype&gt;)</i>	<i>L_Pntr</i>
<i>exp</i>	::=	<i>Ref(&lt;exp&gt;)</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, &lt;datatype&gt;)</i>	<i>L_Array</i>
<i>exp</i>	::=	<i>Subscr(&lt;exp&gt;, &lt;exp&gt;)</i>   <i>Array(&lt;exp&gt;+)</i>	
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>
<i>exp</i>	::=	<i>Attr(&lt;exp&gt;, Name(str))</i>   <i>StructAssign(Name(str), &lt;exp&gt;)+)</i>	
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))+)</i>	
<i>stmt</i>	::=	<i>If(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>IfElse(&lt;exp&gt;, &lt;stmt&gt;*, &lt;stmt&gt;*)</i>	<i>L_If_Else</i>
<i>stmt</i>	::=	<i>While(&lt;exp&gt;, &lt;stmt&gt;*)</i>   <i>DoWhile(&lt;exp&gt;, &lt;stmt&gt;*)</i>	<i>L_Loop</i>
<i>exp</i>	::=	<i>Call(Name(str), &lt;exp&gt;*)</i>	<i>L_Fun</i>
<i>stmt</i>	::=	<i>Return(&lt;exp&gt;)</i>	
<i>decl_def</i>	::=	<i>FunDecl(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))*)</i>   <i>FunDef(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))*</i> , <i>&lt;block&gt;*)</i>	
<i>block</i>	::=	<i>Block(Name(str), &lt;stmt&gt;*)</i>	<i>L_Blocks</i>
<i>stmt</i>	::=	<i>GoTo(Name(str))</i>	
<i>file</i>	::=	<i>File(Name(str), &lt;decl_def&gt;*)</i>	<i>L_File</i>

Grammar 1.3.2: Abstrakte Syntax der Sprache *L<sub>PiocC\_Blocks</sub>*

Alles **rot** markierte bedeutet, es wurde **entfernt** oder **abgeändert**. Alles **ausgegraut** bedeutet, es hat sich im Vergleich zur letzten Abstrakten Syntax **nichts** geändert. Alle normal in **schwarz** geschriebenen Knoten wurden **neu** hinzugefügt.

Die **Abstrakte Syntax** soll im Gegensatz zur **Konkreten Syntax** meist nur vom **Programmierer**

verstanden werden, der den Compiler implementiert und sollte daher vor allem **einfach verständlich** sein und stellt daher eine **Obermenge** aller tatsächlich möglichen **Kompositionen** von **Knoten** dar<sup>a</sup>.

Man bezeichnet hier die **Abstrakte Syntax** als „**Abstrakte Syntax der Sprache**  $L_{Picoc\_Blocks}$ “. Diese Sprache  $L_{Picoc\_Blocks}$  wird durch eine **Konkrete Syntax** beschrieben, die allerdings nicht weiter relevant ist, da in den **Passes** nur **Abstract Syntax Trees** umgeformt werden. Es ist hierbei nur wichtig zu wissen, dass die **Abstrakte Syntax** theoretisch zur Kompilierung der Sprache  $L_{Picoc\_Blocks}$  definiert ist, also die Sprache  $L_{PicoC\_Blocks}$  nicht die Sprache ist, die von der **Abstrakten Syntax** beschrieben ist.

<sup>a</sup>D.h. auch wenn dort **exp** als **Attribut** steht, kann dort **nicht** jeder Knoten, der sich aus der **Produktion** **exp** ergibt auch wirklich eingesetzt werden.

### 1.3.1.2.3 Codebeispiel

In Code 1.8 sieht man den **Abstract-Syntax-Tree** des **PiocoC-Blocks Passes** für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem nun eigene **Blöcke** für die Funktion **faculty** und die **main**-Funktion erstellt werden, in denen die **ersten** Statements der jeweiligen Funktionen bis zum **letzten** Statement oder bis zum ersten **Auftauchen** eines **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-Knoten** stehen. Je nachdem, ob ein **If(exp, stmts)-**, **IfElse(exp, stmts1, stmts2)-**, **While(exp, stmts)-** oder **DoWhile(exp, stmts)-Knoten** auftaucht, werden für die **Bedingung** und mögliche **Branches** eigene **Blöcke** erstellt.

```

1 File
2   Name './example_faculty_it.picoc_blocks',
3   [
4     FunDef
5       IntType 'int',
6       Name 'faculty',
7       [
8         Alloc(Writable(), IntType('int'), Name('n'))
9       ],
10      [
11        Block
12          Name 'faculty.6',
13          [
14            Assign(Alloc(Writable(), IntType('int'), Name('res')), Num('1'))
15            // While(Num('1'), [])
16            GoTo(Name('condition_check.5'))
17          ],
18        Block
19          Name 'condition_check.5',
20          [
21            IfElse
22              Num '1',
23              [
24                GoTo(Name('while_branch.4'))
25              ],
26              [
27                GoTo(Name('while_after.1'))
28              ]
29          ],
30        Block
31          Name 'while_branch.4',

```

```

32     [
33         // If(Atom(Name('n'), Eq('=='), Num('1')), [],),
34         IfElse
35             Atom(Name('n'), Eq('=='), Num('1')),
36             [
37                 GoTo(Name('if.3'))
38             ],
39             [
40                 GoTo(Name('if_else_after.2'))
41             ]
42     ],
43     Block
44         Name 'if.3',
45         [
46             Return(Name('res'))
47         ],
48     Block
49         Name 'if_else_after.2',
50         [
51             Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
52             Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
53             GoTo(Name('condition_check.5'))
54         ],
55     Block
56         Name 'while_after.1',
57         []
58 ],
59 FunDef
60     VoidType 'void',
61     Name 'main',
62     [],
63     [
64         Block
65             Name 'main.0',
66             [
67                 Exp(Call(Name('print'), [Call(Name('faculty'), [Num('4')])]))
68             ]
69     ]
70 ]

```

Code 1.8: PicoC-Blocks Pass für Codebeispiel

### 1.3.1.3 PicoC-ANF Pass

#### 1.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** ist es den **Abstract Syntax Tree** der Sprache  $L_{PicoC\_Blocks}$  in die **Abstrakte Syntax** der Sprache  $L_{PicoC\_ANF}$  umzuformen, welche in **A-Normalform** (Definition ??) und damit auch in **Monadischer Normalform** (Definition ??) ist. Um Wiederholung zu vermeiden wird zur Erklärung der **A-Normalform** auf Unterkapitel ?? verwiesen.

Zudem wird eine **Symboltabelle** (Definition 1.7) eingeführt. In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** zugewiesen, die dem Wert einer von zwei **Countern** `rel_global_addr` und `rel_stack_addr` entspricht. Der Counter `rel_global_addr` ist für Variablen in den **Globalen Statischen Daten** und der Counter `rel_stack_addr` ist für Variablen auf dem **Stackframe**. Einer der beiden **Counter** wird entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle diese Variable `Name('symbol')` vor, so wird mit dem **Symbol**<sup>22</sup> als Schlüssel in der **Symboltabelle** nachgeschlagen und anstelle des `Name(str)`-Knotens die in der **Symboltabelle** nachgeschlagene Adresse in einem `Global(Num('addr'))`- bzw. `Stackframe(Num('addr'))`-Knoten eingesetzt eingefügt. Ob der `Global(Num('addr'))`- oder der `Stackframe(Num('addr'))`-Knoten zum Einsatz kommt, entscheidet sich anhand des **Scopes** (z.B. `@scope`), der in der **Symboltabelle** an den **Bezeichner** drangehängt ist (z.B. `identifizier@scope`).<sup>23</sup>

#### Definition 1.7: Symboltabelle

*Eine über ein **Assoziatives Feld** umgesetzte **Datenstruktur**, die notwendig ist, um das Konzept einer **Variablen** in einer Sprache umzusetzen. Diese Datenstruktur ordnet jedem **Symbol**<sup>a</sup> einer **Variablen**, **Konstanten** oder **Funktion** aus einem **Programm**, Informationen, wie die **Adresse**, die **Position** im Programmcode oder den **Datentyp** zu.*

*Die **Symboltabelle** muss nur während des **Kompilervorgangs** im **Speicher** existieren, da die Einträge in der **Symboltabelle** beeinflussen, was für **Maschinencode** generiert wird und dadurch im **Maschinencode** bereits die richtigen **Adressen** usw. angesprochen werden und es die **Symboltabelle** selbst **nicht** mehr braucht.*

<sup>a</sup>In einer **Symboltabelle** werden **Bezeichner** als **Symbole** bezeichnet.

#### 1.3.1.3.2 Abstrakte Syntax

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Syntax** der Sprache  $L_{PicoC\_Blocks}$  in Tabelle 1.3.2 in die **A-Normalform** zu bringen. Darunter fällt es unter anderem, dafür zu sorgen, dass **Komplexe Knoten**, wie z.B. `BinOp(exp, bin_op, exp)` nur **Atomare Knoten**, wie z.B. `Stack(Num(str))` enthalten können. Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass u.a. die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `File(Name(str), block*)`-Knoten liegen usw., was in Unterkapitel ?? genauer erklärt wird. Die **Symboltabelle** ist ebenfalls als **Abstract Syntax Tree** umgesetzt, wofür in der **Abstrakten Syntax** der Sprache  $L_{PicoC\_ANF}$  in Grammatik 1.3.3 neue Knoten eingeführt werden.

Das ganze resultiert in der **Abstrakten Syntax** der Sprache  $L_{PicoC\_ANF}$  in Grammatik 1.3.3.

<sup>22</sup>Bzw. der **Bezeichner**

<sup>23</sup>Die Umsetzung von **Scopes** wird in Unterkapitel ?? genauer beschrieben.

<i>stmt</i>	::=	<i>SingleLineComment(str, str)</i>   <i>RETIComment()</i>	<i>L_Comment</i>	
<i>un_op</i>	::=	<i>Minus()</i>   <i>Not()</i>	<i>L_Arith</i>	
<i>bin_op</i>	::=	<i>Add()</i>   <i>Sub()</i>   <i>Mul()</i>   <i>Div()</i>   <i>Mod()</i>   <i>Oplus()</i>   <i>And()</i>   <i>Or()</i>		
<i>exp</i>	::=	<i>Name(str)</i>   <i>Num(str)</i>   <i>Char(str)</i>   <i>Global(Num(str))</i>   <i>Stackframe(Num(str))</i>   <i>Stack(Num(str))</i>   <i>BinOp(Stack(Num(str)), &lt;bin_op&gt;, Stack(Num(str)))</i>   <i>UnOp(&lt;un_op&gt;, Stack(Num(str)))</i>   <i>Call(Name('input'), Empty())</i>   <i>Call(Name('print'), &lt;exp&gt;)</i>   <i>Exp(&lt;exp&gt;)</i>		
<i>un_op</i>	::=	<i>LogicNot()</i>	<i>L_Logic</i>	
<i>rel</i>	::=	<i>Eq()</i>   <i>NEq()</i>   <i>Lt()</i>   <i>LtE()</i>   <i>Gt()</i>   <i>GtE()</i>		
<i>bin_op</i>	::=	<i>LogicAnd()</i>   <i>LogicOr()</i>		
<i>exp</i>	::=	<i>Atom(Stack(Num(str)), &lt;rel&gt;, Stack(Num(str)))</i>   <i>ToBool(Stack(Num(str)))</i>		
<i>type_qual</i>	::=	<i>Const()</i>   <i>Writeable()</i>	<i>L_Assign_Alloc</i>	
<i>datatype</i>	::=	<i>IntType()</i>   <i>CharType()</i>   <i>VoidType()</i>		
<i>exp</i>	::=	<i>Alloc(&lt;type_qual&gt;, &lt;datatype&gt;, Name(str))</i>		
<i>stmt</i>	::=	<i>Assign(Global(Num(str)), Stack(Num(str)))</i>   <i>Assign(Stackframe(Num(str)), Stack(Num(str)))</i>   <i>Assign(Stack(Num(str)), Global(Num(str)))</i>   <i>Assign(Stack(Num(str)), Stackframe(Num(str)))</i>		
<i>datatype</i>	::=	<i>PntrDecl(Num(str), &lt;datatype&gt;)</i>   <i>Ref(Global(str))</i>   <i>Ref(Stackframe(str))</i>   <i>Ref(Subscr(&lt;exp&gt;, &lt;exp&gt;   Ref(Attr(&lt;exp&gt;, Name(str))))</i>	<i>L_Pntr</i>	
<i>datatype</i>	::=	<i>ArrayDecl(Num(str)+, &lt;datatype&gt;)</i>	<i>L_Array</i>	
<i>exp</i>	::=	<i>Subscr(&lt;exp&gt;, Stack(Num(str)))</i>   <i>Array(&lt;exp&gt;+)</i>		
<i>datatype</i>	::=	<i>StructSpec(Name(str))</i>	<i>L_Struct</i>	
<i>exp</i>	::=	<i>Attr(&lt;exp&gt;, Name(str))</i>   <i>Struct(Assign(Name(str), &lt;exp&gt;)+)</i>		
<i>decl_def</i>	::=	<i>StructDecl(Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))+)</i>		
<i>stmt</i>	::=	<i>IfElse(Stack(Num(str)), &lt;stmt&gt;*, &lt;stmt&gt;*)</i>	<i>L_If_Else</i>	
<i>exp</i>	::=	<i>Call(Name(str), &lt;exp&gt;*)</i>	<i>L_Fun</i>	
<i>stmt</i>	::=	<i>StackMalloc(Num(str))</i>   <i>NewStackframe(Name(str), GoTo(str))</i>   <i>Exp(GoTo(Name(str)))</i>   <i>RemoveStackframe()</i>   <i>Return(Empty())</i>   <i>Return(&lt;exp&gt;)</i>		
<i>decl_def</i>	::=	<i>FunDecl(&lt;datatype&gt;, Name(str))</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))*</i>   <i>FunDef(&lt;datatype&gt;, Name(str),</i> <i>Alloc(Writeable(), &lt;datatype&gt;, Name(str))* &lt;block&gt;*)</i>		
<i>block</i>	::=	<i>Block(Name(str), &lt;stmt&gt;*)</i>	<i>L_Blocks</i>	
<i>stmt</i>	::=	<i>GoTo(Name(str))</i>		
<i>file</i>	::=	<i>File(Name(str), &lt;block&gt;*)</i>	<i>L_File</i>	
<i>symbol_table</i>	::=	<i>SymbolTable(&lt;symbol&gt;*)</i>	<i>L_Symbol_Table</i>	
<i>symbol</i>	::=	<i>Symbol(&lt;type_qual&gt;, &lt;datatype&gt;, &lt;name&gt;, &lt;val&gt;, &lt;pos&gt;, &lt;size&gt;)</i>		
<i>type_qual</i>	::=	<i>Empty()</i>		
<i>datatype</i>	::=	<i>BuiltIn()</i>   <i>SelfDefined()</i>		
<i>name</i>	::=	<i>Name(str)</i>		
<i>val</i>	::=	<i>Num(str)</i>   <i>Empty()</i>		
<i>pos</i>	::=	<i>Pos(Num(str), Num(str))</i>   <i>Empty()</i>		
<i>size</i>	::=	<i>Num(str)</i>   <i>Empty()</i>		

### 1.3.1.3.3 Codebeispiel

In Code 1.9 sieht man den **Abstract-Syntax-Tree** des **PiocC-ANF Passes** für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem alle Statements und Ausdrücke in **A-Normalform** sind. Die **IfElse(exp, stmts, stmts)**-Knoten sind hier in **A-Normalform** gebracht worden, indem ihre **Komplexe Bedingung** vorgezogen wurde und das Ergebnis der **Komplexen Bedingung** einer **Location** zugewiesen ist und sie selbst das Ergebnis über den **Atomaren Ausdruck** `Stack(Num(str))` vom Stack lesen: `IfElse(Stack(Num(str)), stmts, stmts)`. **Funktionen** sind nur noch über die **Labels** von Blöcken zu erkennen, die den gleichen Bezeichner haben, wie die ursprüngliche Funktion und es lässt sich nur durch das **Nachverfolgen** der `GoTo(Name('label'))`-Knoten nachvollziehen, was ursprünglich zur Funktion gehörte.

```

1 File
2   Name './example_faculty_it.picoc_mon',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         // Assign(Name('res'), Num('1'))
8         Exp(Num('1'))
9         Assign(Stackframe(Num('1')), Stack(Num('1')))
10        // While(Num('1'), [])
11        Exp(GoTo(Name('condition_check.5')))
12      ],
13    Block
14      Name 'condition_check.5',
15      [
16        // IfElse(Num('1'), [], [])
17        Exp(Num('1')),
18        IfElse
19          Stack
20            Num '1',
21            [
22              GoTo(Name('while_branch.4'))
23            ],
24            [
25              GoTo(Name('while_after.1'))
26            ]
27        ],
28      Block
29        Name 'while_branch.4',
30        [
31          // If(Atom(Name('n'), Eq('=='), Num('1')), [])
32          // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
33          Exp(Stackframe(Num('0')))
34          Exp(Num('1'))
35          Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1')))),
36          IfElse
37            Stack
38              Num '1',
39              [
40                GoTo(Name('if.3'))
41              ],
42              [
43                GoTo(Name('if_else_after.2'))
44              ]
45        ],

```



```

46  Block
47      Name 'if.3',
48      [
49          // Return(Name('res'))
50          Exp(Stackframe(Num('1')))
51          Return(Stack(Num('1')))
52      ],
53  Block
54      Name 'if_else_after.2',
55      [
56          // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
57          Exp(Stackframe(Num('0')))
58          Exp(Stackframe(Num('1')))
59          Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
60          Assign(Stackframe(Num('1')), Stack(Num('1')))
61          // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
62          Exp(Stackframe(Num('0')))
63          Exp(Num('1'))
64          Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
65          Assign(Stackframe(Num('0')), Stack(Num('1')))
66          Exp(GoTo(Name('condition_check.5')))
67      ],
68  Block
69      Name 'while_after.1',
70      [
71          Return(Empty())
72      ],
73  Block
74      Name 'main.0',
75      [
76          StackMalloc(Num('2'))
77          Exp(Num('4'))
78          NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
79          Exp(GoTo(Name('faculty.6')))
80          RemoveStackframe()
81          Exp(ACC)
82          Exp(Call(Name('print'), [Stack(Num('1'))]))
83          Return(Empty())
84      ]
85  ]

```

Code 1.9: PicoC-ANF Pass für Codebespiel

#### 1.3.1.4 RETI-Blocks Pass

##### 1.3.1.4.1 Aufgabe

Die Aufgabe des **RETI-Blocks Passes** ist es die **Statements** in der **Blöcken**, die durch **PicoC-Knoten** im **Abstract Syntax Tree** der Sprache  $L_{PicoC\_ANF}$  dargestellt sind durch ihren entsprechenden **RETI-Knoten** zu ersetzen.

##### 1.3.1.4.2 Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache  $L_{RETI\_Blocks}$  in Grammatik 1.3.4 ist verglichen mit der **Abstrakten Syntax** der Sprache  $L_{PicoC\_ANF}$  in Grammatik 1.3.3 stark verändert, denn der Großteil der **PicoC-Knoten** wird in diesem Pass durch entsprechende **RETI-Knoten** ersetzt. Die einzigen verbleibenden **PicoC-Knoten**

sind  $\text{Exp}(\text{GoTo}(\text{str}))$ ,  $\text{Block}(\text{Name}(\text{str}), \langle \text{instr} \rangle^*)$  und  $\text{File}(\text{Name}(\text{str}), \langle \text{block} \rangle^*)$ , da das gesamte Konzept mit den **Blöcken** erst im **RETI-Pass** in Unterkapitel 1.3.8 aufgelöst wird.

$\text{reg}$	$::= \text{ACC}() \mid \text{IN1}() \mid \text{IN2}() \mid \text{PC}() \mid \text{SP}() \mid \text{BAF}()$	$L\_RETI$
	$\mid \text{CS}() \mid \text{DS}()$	
$\text{arg}$	$::= \text{Reg}(\langle \text{reg} \rangle) \mid \text{Num}(\text{str})$	
$\text{rel}$	$::= \text{Eq}() \mid \text{NEq}() \mid \text{Lt}() \mid \text{LtE}() \mid \text{Gt}() \mid \text{GtE}()$	
	$\mid \text{Always}() \mid \text{NOp}()$	
$\text{op}$	$::= \text{Add}() \mid \text{Addi}() \mid \text{Sub}() \mid \text{Subi}() \mid \text{Mult}() \mid \text{Multi}()$	
	$\mid \text{Div}() \mid \text{Divi}() \mid \text{Mod}() \mid \text{Modi}() \mid \text{Oplus}() \mid \text{Oplusi}()$	
	$\mid \text{Or}() \mid \text{Ori}() \mid \text{And}() \mid \text{Andi}()$	
	$\mid \text{Load}() \mid \text{Loadin}() \mid \text{Loadi}() \mid \text{Store}() \mid \text{Storein}() \mid \text{Move}()$	
$\text{instr}$	$::= \text{Instr}(\langle \text{op} \rangle, \langle \text{arg} \rangle^+) \mid \text{Jump}(\langle \text{rel} \rangle, \text{Num}(\text{str})) \mid \text{Int}(\text{Num}(\text{str}))$	
	$\mid \text{RTI}() \mid \text{Call}(\text{Name}(\text{'print'}), \langle \text{reg} \rangle) \mid \text{Call}(\text{Name}(\text{'input'}), \langle \text{reg} \rangle)$	
	$\mid \text{SingleLineComment}(\text{str}, \text{str})$	
	$\mid \text{Instr}(\text{Loadi}(), [\text{Reg}(\text{Acc}()), \text{GoTo}(\text{Name}(\text{str}))]) \mid \text{Jump}(\text{Eq}(), \text{GoTo}(\text{Name}(\text{str})))$	
$\text{instr}$	$::= \text{Exp}(\text{GoTo}(\text{str}))$	$L\_PicoC$
$\text{block}$	$::= \text{Block}(\text{Name}(\text{str}), \langle \text{instr} \rangle^*)$	
$\text{file}$	$::= \text{File}(\text{Name}(\text{str}), \langle \text{block} \rangle^*)$	

#### Grammar 1.3.4: Abstrakte Syntax der Sprache $L_{RETI\_Blocks}$

##### 1.3.1.4.3 Codebeispiel

In Code 1.10 sieht man den **Abstract-Syntax-Tree** des **RETI-Blocks Passes** für das aus Unterkapitel 1.6 weitergeführte Beispiel, indem die **Statements**, die durch entsprechende **PicoC-Knoten** im **Abstrakt Syntax Tree** der Sprache  $L_{PicoC\_ANF}$  in Grammatik 1.3.3 repräsentiert waren nun durch ihre entsprechenden **RETI-Knoten** ersetzt werden.

```

1 File
2   Name './example_faculty_it.reti_blocks',
3   [
4     Block
5       Name 'faculty.6',
6       [
7         # // Assign(Name('res'), Num('1'))
8         # Exp(Num('1'))
9         SUBI SP 1;
10        LOADI ACC 1;
11        STOREIN SP ACC 1;
12        # Assign(Stackframe(Num('1')), Stack(Num('1')))
13        LOADIN SP ACC 1;
14        STOREIN BAF ACC -3;
15        ADDI SP 1;
16        # // While(Num('1'), [])
17        # Exp(GoTo(Name('condition_check.5')))
18        Exp(GoTo(Name('condition_check.5')))
19      ],
20    Block
21      Name 'condition_check.5',
22      [
23        # // IfElse(Num('1'), [], [])
24        # Exp(Num('1'))

```

```

25     SUBI SP 1;
26     LOADI ACC 1;
27     STOREIN SP ACC 1;
28     # IfElse(Stack(Num('1')), [], [])
29     LOADIN SP ACC 1;
30     ADDI SP 1;
31     JUMP== GoTo(Name('while_after.1'));
32     Exp(GoTo(Name('while_branch.4')))
33 ],
34 Block
35     Name 'while_branch.4',
36     [
37         # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
38         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
39         # Exp(Stackframe(Num('0')))
40         SUBI SP 1;
41         LOADIN BAF ACC -2;
42         STOREIN SP ACC 1;
43         # Exp(Num('1'))
44         SUBI SP 1;
45         LOADI ACC 1;
46         STOREIN SP ACC 1;
47         LOADIN SP ACC 2;
48         LOADIN SP IN2 1;
49         SUB ACC IN2;
50         JUMP== 3;
51         LOADI ACC 0;
52         JUMP 2;
53         LOADI ACC 1;
54         STOREIN SP ACC 2;
55         ADDI SP 1;
56         # IfElse(Stack(Num('1')), [], [])
57         LOADIN SP ACC 1;
58         ADDI SP 1;
59         JUMP== GoTo(Name('if_else_after.2'));
60         Exp(GoTo(Name('if.3')))
61     ],
62 Block
63     Name 'if.3',
64     [
65         # // Return(Name('res'))
66         # Exp(Stackframe(Num('1')))
67         SUBI SP 1;
68         LOADIN BAF ACC -3;
69         STOREIN SP ACC 1;
70         # Return(Stack(Num('1')))
71         LOADIN SP ACC 1;
72         ADDI SP 1;
73         LOADIN BAF PC -1;
74     ],
75 Block
76     Name 'if_else_after.2',
77     [
78         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
79         # Exp(Stackframe(Num('0')))
80         SUBI SP 1;
81         LOADIN BAF ACC -2;

```

```

82     STOREIN SP ACC 1;
83     # Exp(Stackframe(Num('1')))
84     SUBI SP 1;
85     LOADIN BAF ACC -3;
86     STOREIN SP ACC 1;
87     # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
88     LOADIN SP ACC 2;
89     LOADIN SP IN2 1;
90     MULT ACC IN2;
91     STOREIN SP ACC 2;
92     ADDI SP 1;
93     # Assign(Stackframe(Num('1')), Stack(Num('1')))
94     LOADIN SP ACC 1;
95     STOREIN BAF ACC -3;
96     ADDI SP 1;
97     # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
98     # Exp(Stackframe(Num('0')))
99     SUBI SP 1;
100    LOADIN BAF ACC -2;
101    STOREIN SP ACC 1;
102    # Exp(Num('1'))
103    SUBI SP 1;
104    LOADI ACC 1;
105    STOREIN SP ACC 1;
106    # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
107    LOADIN SP ACC 2;
108    LOADIN SP IN2 1;
109    SUB ACC IN2;
110    STOREIN SP ACC 2;
111    ADDI SP 1;
112    # Assign(Stackframe(Num('0')), Stack(Num('1')))
113    LOADIN SP ACC 1;
114    STOREIN BAF ACC -2;
115    ADDI SP 1;
116    # Exp(GoTo(Name('condition_check.5')))
117    Exp(GoTo(Name('condition_check.5')))
118  ],
119  Block
120    Name 'while_after.1',
121    [
122      # Return(Empty())
123      LOADIN BAF PC -1;
124    ],
125  Block
126    Name 'main.0',
127    [
128      # StackMalloc(Num('2'))
129      SUBI SP 2;
130      # Exp(Num('4'))
131      SUBI SP 1;
132      LOADI ACC 4;
133      STOREIN SP ACC 1;
134      # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
135      MOVE BAF ACC;
136      ADDI SP 3;
137      MOVE SP BAF;
138      SUBI SP 4;

```

```

139     STOREIN BAF ACC 0;
140     LOADI ACC GoTo(Name('addr@next_instr'));
141     ADD ACC CS;
142     STOREIN BAF ACC -1;
143     # Exp(GoTo(Name('faculty.6')))
144     Exp(GoTo(Name('faculty.6')))
145     # RemoveStackframe()
146     MOVE BAF IN1;
147     LOADIN IN1 BAF 0;
148     MOVE IN1 SP;
149     # Exp(ACC)
150     SUBI SP 1;
151     STOREIN SP ACC 1;
152     LOADIN SP ACC 1;
153     ADDI SP 1;
154     CALL PRINT ACC;
155     # Return(Empty())
156     LOADIN BAF PC -1;
157 ]
158 ]

```

Code 1.10: RETI-Blocks Pass für Codebeispiel

Wenn der **Abstract Syntax Tree** ausgegeben wird, ist die Darstellung nicht ausschließlich in **Abstrakter Syntax**, da die **RETI-Knoten** aus bereits im Unterkapitel 1.2.5.6 vermitteltem Grund in **Konkreter Syntax** ausgegeben werden.

### 1.3.1.5 RETI-Patch Pass

#### 1.3.1.5.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist das **Ausbessern** (engl. to patch) des **Abstract Syntax Trees**, durch:

- das **Einfügen** eines **start.<nummer>-Blockes**, welcher ein **GoTo(Name('main'))** zur **main-Funktion** enthält, wenn in manchen Fällen die **main-Funktion** **nicht** die erste Funktion ist und daher am Anfang zur **main-Funktion** gesprungen werden muss.
- das **Entfernen** von **GoTo()**'s, deren Sprung nur **eine** Adresse weiterspringen würde.
- das **Voranstellen** von **RETI-Knoten**, die vor jeder **Division** **Instr(Div(), args)** prüfen, ob, nicht durch 0 geteilt wird.<sup>24</sup>
- das Überprüfen darauf, ob bestimmte **Immediates** **Im(str)** in Befehlen, wie z.B. **Jump(rel, Im(str))**, **Instr(Loadin(), [reg, reg, Im(str)])**, **Instr(Loadi(), [reg, Im(str)])** usw. **kleiner**  $-2^{21}$  oder **größer**  $2^{21} - 1$  sind. Im Fall dessen, dass es so ist, muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **bitweise Oder** berechnet werden. Im Fall, dessen, dass der **Immediate** allerdings **kleiner**  $-(2^{31})$  oder **größer**  $2^{31} - 1$  ist, wird eine Fehlermeldung **TooLargeLiteral** ausgegeben.

#### 1.3.1.5.2 Abstrakte Syntax

<sup>24</sup>Das fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

Die **Abstrakte Syntax** der Sprache  $L_{RETI\_Patch}$  in Grammatik 1.3.5 ist im Vergleich zur **Abstrakten Syntax** der Sprache  $L_{RETI\_Blocks}$  in Grammatik 1.3.4 kaum verändert. Es muss nur ein Knoten `Exit()` hinzugefügt werden, der im Falle einer **Division durch 0** die Ausführung des Programs beendet.

$reg$	$::= ACC() \mid IN1() \mid IN2() \mid PC() \mid SP() \mid BAF()$ $\mid CS() \mid DS()$	$L_{RETI}$
$arg$	$::= Reg(\langle reg \rangle) \mid Num(str)$	
$rel$	$::= Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$ $\mid Always() \mid NOP()$	
$op$	$::= Add() \mid Addi() \mid Sub() \mid Subi() \mid Mult() \mid Multi()$ $\mid Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi()$ $\mid Or() \mid Ori() \mid And() \mid Andi()$	
$instr$	$::= Instr(\langle op \rangle, \langle arg \rangle+) \mid Jump(\langle rel \rangle, Num(str)) \mid Int(Num(str))$ $\mid RTI() \mid Call(Name('print'), \langle reg \rangle) \mid Call(Name('input'), \langle reg \rangle)$ $\mid SingleLineComment(str, str)$ $\mid Instr(Loadi(), [Reg(Acc()), GoTo(Name(str))]) \mid Jump(Eq(), GoTo(Name(str)))$	
$instr$	$::= Exp(GoTo(str)) \mid Exit(Num(str))$	$L_{PicoC}$
$block$	$::= Block(Name(str), \langle instr \rangle^*)$	
$file$	$::= File(Name(str), \langle block \rangle^*)$	

### Grammar 1.3.5: Abstrakte Syntax der Sprache $L_{RETI\_Patch}$

#### 1.3.1.5.3 Codebeispiel

In Code 1.11 sieht man den **Abstract-Syntax-Tree** des **Pioco-Patch Passes** für das aus Unterkapitel 1.6 weitergeführte Beispiel. Durch den **RETI-Patch Pass** wurde hier ein `start.<number>-Block`<sup>25</sup> eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-**Statements** entfernt<sup>26</sup>, die nur einen Sprung um **eine** Position entsprachen hätten.

```

1 File
2   Name './example_faculty_it.reti_patch',
3   [
4     Block
5       Name 'start.7',
6       [
7         # // Exp(GoTo(Name('main.0')))
8         Exp(GoTo(Name('main.0')))
9       ],
10    Block
11      Name 'faculty.6',
12      [
13        # // Assign(Name('res'), Num('1'))
14        # Exp(Num('1'))
15        SUBI SP 1;
16        LOADI ACC 1;
17        STOREIN SP ACC 1;
18        # Assign(Stackframe(Num('1')), Stack(Num('1')))
19        LOADIN SP ACC 1;
20        STOREIN BAF ACC -3;

```

<sup>25</sup>Dieser **Block** wurde im Code 1.8 markiert.

<sup>26</sup>Diese **entfernten GoTo(Name(str))**'s wurden ebenfalls im Code 1.8 markiert.

```

21     ADDI SP 1;
22     # // While(Num('1'), [])
23     # Exp(GoTo(Name('condition_check.5')))
24     # // not included Exp(GoTo(Name('condition_check.5')))
25 ],
26 Block
27     Name 'condition_check.5',
28     [
29         # // IfElse(Num('1'), [], [])
30         # Exp(Num('1'))
31         SUBI SP 1;
32         LOADI ACC 1;
33         STOREIN SP ACC 1;
34         # IfElse(Stack(Num('1')), [], [])
35         LOADIN SP ACC 1;
36         ADDI SP 1;
37         JUMP== GoTo(Name('while_after.1'));
38         # // not included Exp(GoTo(Name('while_branch.4')))
39     ],
40 Block
41     Name 'while_branch.4',
42     [
43         # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
44         # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
45         # Exp(Stackframe(Num('0')))
46         SUBI SP 1;
47         LOADIN BAF ACC -2;
48         STOREIN SP ACC 1;
49         # Exp(Num('1'))
50         SUBI SP 1;
51         LOADI ACC 1;
52         STOREIN SP ACC 1;
53         LOADIN SP ACC 2;
54         LOADIN SP IN2 1;
55         SUB ACC IN2;
56         JUMP== 3;
57         LOADI ACC 0;
58         JUMP 2;
59         LOADI ACC 1;
60         STOREIN SP ACC 2;
61         ADDI SP 1;
62         # IfElse(Stack(Num('1')), [], [])
63         LOADIN SP ACC 1;
64         ADDI SP 1;
65         JUMP== GoTo(Name('if_else_after.2'));
66         # // not included Exp(GoTo(Name('if.3')))
67     ],
68 Block
69     Name 'if.3',
70     [
71         # // Return(Name('res'))
72         # Exp(Stackframe(Num('1')))
73         SUBI SP 1;
74         LOADIN BAF ACC -3;
75         STOREIN SP ACC 1;
76         # Return(Stack(Num('1')))
77         LOADIN SP ACC 1;

```

```

78     ADDI SP 1;
79     LOADIN BAF PC -1;
80 ],
81 Block
82     Name 'if_else_after.2',
83     [
84         # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
85         # Exp(Stackframe(Num('0')))
86         SUBI SP 1;
87         LOADIN BAF ACC -2;
88         STOREIN SP ACC 1;
89         # Exp(Stackframe(Num('1')))
90         SUBI SP 1;
91         LOADIN BAF ACC -3;
92         STOREIN SP ACC 1;
93         # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
94         LOADIN SP ACC 2;
95         LOADIN SP IN2 1;
96         MULT ACC IN2;
97         STOREIN SP ACC 2;
98         ADDI SP 1;
99         # Assign(Stackframe(Num('1')), Stack(Num('1')))
100        LOADIN SP ACC 1;
101        STOREIN BAF ACC -3;
102        ADDI SP 1;
103        # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
104        # Exp(Stackframe(Num('0')))
105        SUBI SP 1;
106        LOADIN BAF ACC -2;
107        STOREIN SP ACC 1;
108        # Exp(Num('1'))
109        SUBI SP 1;
110        LOADI ACC 1;
111        STOREIN SP ACC 1;
112        # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
113        LOADIN SP ACC 2;
114        LOADIN SP IN2 1;
115        SUB ACC IN2;
116        STOREIN SP ACC 2;
117        ADDI SP 1;
118        # Assign(Stackframe(Num('0')), Stack(Num('1')))
119        LOADIN SP ACC 1;
120        STOREIN BAF ACC -2;
121        ADDI SP 1;
122        # Exp(GoTo(Name('condition_check.5')))
123        Exp(GoTo(Name('condition_check.5')))
124    ],
125 Block
126     Name 'while_after.1',
127     [
128         # Return(Empty())
129         LOADIN BAF PC -1;
130     ],
131 Block
132     Name 'main.0',
133     [
134         # StackMalloc(Num('2'))

```



```

135     SUBI SP 2;
136     # Exp(Num('4'))
137     SUBI SP 1;
138     LOADI ACC 4;
139     STOREIN SP ACC 1;
140     # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr')))
141     MOVE BAF ACC;
142     ADDI SP 3;
143     MOVE SP BAF;
144     SUBI SP 4;
145     STOREIN BAF ACC 0;
146     LOADI ACC GoTo(Name('addr@next_instr'));
147     ADD ACC CS;
148     STOREIN BAF ACC -1;
149     # Exp(GoTo(Name('faculty.6')))
150     Exp(GoTo(Name('faculty.6')))
151     # RemoveStackframe()
152     MOVE BAF IN1;
153     LOADIN IN1 BAF 0;
154     MOVE IN1 SP;
155     # Exp(ACC)
156     SUBI SP 1;
157     STOREIN SP ACC 1;
158     LOADIN SP ACC 1;
159     ADDI SP 1;
160     CALL PRINT ACC;
161     # Return(Empty())
162     LOADIN BAF PC -1;
163 ]
164 ]

```

Code 1.11: RETI-Patch Pass für Codebeispiel

### 1.3.1.6 RETI Pass

#### 1.3.1.6.1 Aufgabe

Die Aufgabe des **RETI-Patch Passes** ist es die `GoTo(Name(str))`-Knoten in den den Knoten `Instr(Loadi(), [reg, GoTo(Name(str))])`, `Jump(Eq(), GoTo(Name(str)))` und `Exp(GoTo(Name(str)))` durch eine entsprechende **Adresse** zu ersetzen, die entsprechende **Distanz** oder einen entsprechenden **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))`. Die **Distanz-** und **Adressberechnung** wird in Unterkapitel ?? genauer mit **Formeln** erklärt.

#### 1.3.1.6.2 Konkrete und Abstrakte Syntax

Die **Abstrakte Syntax** der Sprache  $L_{RETI}$  in Grammatik 1.3.8 hat im Vergleich zur **Abstrakten Syntax** der Sprache  $L_{RETI.Patch}$  in Grammatik 1.3.5 nur noch ausschließlich **RETI-Knoten**. Alle **RETI-Knoten** stehen nun einem `Program(Name(str), instr)`-Knoten.

Ausgegeben wird der finale **Maschinencode** allerdings in **Konkreter Syntax**, die sich aus den Grammatiken 1.3.6 und 1.3.7 für jeweils die **Lexikalische** und **Syntaktische Analyse** zusammensetzt. Der Grund, warum die **Konkrete Syntax** der Sprache  $L_{RETI}$  auch nochmal in einen Teil für die **Lexikalische** und **Syntaktische Analyse** unterteilt ist, hat den Grund, dass für die Bachelorarbeit zum **Testen** des **PicoC-Compilers** ein **RETI-Interpreter** implementiert wurde, der den RETI-Code **lexen** und **parsen** muss, um ihn später **interpretieren** zu können.

<i>dig_no_0</i>	::=	"1"   "2"   "3"   "4"   "5"   "6"	<i>L_Program</i>
		"7"   "8"   "9"	
<i>dig_with_0</i>	::=	"0"   <i>dig_no_0</i>	
<i>num</i>	::=	"0"   <i>dig_no_0</i> <i>dig_with_0</i> *   "-" <i>dig_no_0</i> *	
<i>letter</i>	::=	"a"..."Z"	
<i>name</i>	::=	<i>letter</i> ( <i>letter</i>   <i>dig_with_0</i>   _)*	
<i>reg</i>	::=	"ACC"   "IN1"   "IN2"   "PC"   "SP"	
		"BAF"   "CS"   "DS"	
<i>arg</i>	::=	<i>reg</i>   <i>num</i>	
<i>rel</i>	::=	"=="   "!="   "<"   "<="   ">"	
		">="   "_NOP"	

**Grammar 1.3.6:** Konkrete Syntax der Sprache *L<sub>RETI</sub>* für die Lexikalische Analyse in EBNF

<i>instr</i>	::=	"ADD" <i>reg</i> <i>arg</i>   "ADDI" <i>reg</i> <i>num</i>   "SUB" <i>reg</i> <i>arg</i>	<i>L_Program</i>
		"SUBI" <i>reg</i> <i>num</i>   "MULT" <i>reg</i> <i>arg</i>   "MULTI" <i>reg</i> <i>num</i>	
		"DIV" <i>reg</i> <i>arg</i>   "DIVI" <i>reg</i> <i>num</i>   "MOD" <i>reg</i> <i>arg</i>	
		"MODI" <i>reg</i> <i>num</i>   "OPLUS" <i>reg</i> <i>arg</i>   "OPLUSI" <i>reg</i> <i>num</i>	
		"OR" <i>reg</i> <i>arg</i>   "ORI" <i>reg</i> <i>num</i>	
		"AND" <i>reg</i> <i>arg</i>   "ANDI" <i>reg</i> <i>num</i>	
		"LOAD" <i>reg</i> <i>num</i>   "LOADIN" <i>arg</i> <i>arg</i> <i>num</i>	
		"LOADI" <i>reg</i> <i>num</i>	
		"STORE" <i>reg</i> <i>num</i>   "STOREIN" <i>arg</i> <i>argnum</i>	
		"MOVE" <i>reg</i> <i>reg</i>	
		"JUMP" <i>rel</i> <i>num</i>   <i>INT</i> <i>num</i>   <i>RTI</i>	
		"CALL" "INPUT" <i>reg</i>   "CALL" "PRINT" <i>reg</i>	
<i>program</i>	::=	<i>name</i> ( <i>instr</i> ";" )*	

**Grammar 1.3.7:** Konkrete Syntax der Sprache *L<sub>RETI</sub>* für die Syntaktische Analyse in EBNF

<i>reg</i>	::=	<i>ACC</i> ()   <i>IN1</i> ()   <i>IN2</i> ()   <i>PC</i> ()   <i>SP</i> ()   <i>BAF</i> ()	<i>L_RETI</i>
		<i>CS</i> ()   <i>DS</i> ()	
<i>arg</i>	::=	<i>Reg</i> ( <i>&lt;reg&gt;</i> )   <i>Num</i> ( <i>str</i> )	
<i>rel</i>	::=	<i>Eq</i> ()   <i>NEq</i> ()   <i>Lt</i> ()   <i>LtE</i> ()   <i>Gt</i> ()   <i>GtE</i> ()	
		<i>Always</i> ()   <i>NOp</i> ()	
<i>op</i>	::=	<i>Add</i> ()   <i>Addi</i> ()   <i>Sub</i> ()   <i>Subi</i> ()   <i>Mult</i> ()   <i>Multi</i> ()	
		<i>Div</i> ()   <i>Divi</i> ()   <i>Mod</i> ()   <i>Modi</i> ()   <i>Oplus</i> ()   <i>Oplusi</i> ()	
		<i>Or</i> ()   <i>Ori</i> ()   <i>And</i> ()   <i>Andi</i> ()	
		<i>Load</i> ()   <i>Loadin</i> ()   <i>Loadi</i> ()   <i>Store</i> ()   <i>Storein</i> ()   <i>Move</i> ()	
<i>instr</i>	::=	<i>Instr</i> ( <i>&lt;op&gt;</i> , <i>&lt;arg&gt;</i> +)   <i>Jump</i> ( <i>&lt;rel&gt;</i> , <i>Num</i> ( <i>str</i> ))   <i>Int</i> ( <i>Num</i> ( <i>str</i> ))	
		<i>RTI</i> ()   <i>Call</i> ( <i>Name</i> ('print'), <i>&lt;reg&gt;</i> )   <i>Call</i> ( <i>Name</i> ('input'), <i>&lt;reg&gt;</i> )	
		<i>SingleLineComment</i> ( <i>str</i> , <i>str</i> )	
		<i>Instr</i> ( <i>Loadi</i> (), [ <i>Reg</i> ( <i>Acc</i> ()), <i>GoTo</i> ( <i>Name</i> ( <i>str</i> ))])   <i>Jump</i> ( <i>Eq</i> (), <i>GoTo</i> ( <i>Name</i> ( <i>str</i> )))	
<i>program</i>	::=	<i>Program</i> ( <i>Name</i> ( <i>str</i> ), <i>&lt;instr&gt;</i> *)	
<i>instr</i>	::=	<i>Exp</i> ( <i>GoTo</i> ( <i>str</i> ))   <i>Exit</i> ( <i>Num</i> ( <i>str</i> ))	<i>L_PicoC</i>
<i>block</i>	::=	<i>Block</i> ( <i>Name</i> ( <i>str</i> ), <i>&lt;instr&gt;</i> *)	
<i>file</i>	::=	<i>File</i> ( <i>Name</i> ( <i>str</i> ), <i>&lt;block&gt;</i> *)	

**Grammar 1.3.8:** Abstrakte Syntax der Sprache *L<sub>RETI</sub>*

### 1.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkreten Syntax** ausgegeben werden, wie in Code 1.12 zu sehen ist. Es gibt **keine Blöcke** mehr und die **RETI-Befehle** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** ausschließlich aus **RETI-Ausdrücken** bestehen<sup>27</sup>, die sich in den **Blöcken** befunden haben, wurden durch **RETI-Befehle** ersetzt.

Der **Program(Name(str), instr)**-Knoten, indem alle **RETI-Knoten** stehen gibt alleinig die **RETI-Knoten**, die er beinhaltet aus und fügt ansonsten nichts hinzu, wodurch der **Abstract Syntax Tree**, wenn er in eine Datei ausgegeben wird, direkt **RETI-Code** in **menschenlesbarer Repräsentation** erzeugt.

```

1 # // Exp(GoTo(Name('main.0')))
2 JUMP 67;
3 # // Assign(Name('res'), Num('1'))
4 # Exp(Num('1'))
5 SUBI SP 1;
6 LOADI ACC 1;
7 STOREIN SP ACC 1;
8 # Assign(Stackframe(Num('1')), Stack(Num('1')))
9 LOADIN SP ACC 1;
10 STOREIN BAF ACC -3;
11 ADDI SP 1;
12 # // While(Num('1'), [])
13 # Exp(GoTo(Name('condition_check.5')))
14 # // not included Exp(GoTo(Name('condition_check.5')))
15 # // IfElse(Num('1'), [], [])
16 # Exp(Num('1'))
17 SUBI SP 1;
18 LOADI ACC 1;
19 STOREIN SP ACC 1;
20 # IfElse(Stack(Num('1')), [], [])
21 LOADIN SP ACC 1;
22 ADDI SP 1;
23 JUMP== 54;
24 # // not included Exp(GoTo(Name('while_branch.4')))
25 # // If(Atom(Name('n'), Eq('=='), Num('1')), [])
26 # // IfElse(Atom(Name('n'), Eq('=='), Num('1')), [], [])
27 # Exp(Stackframe(Num('0')))
28 SUBI SP 1;
29 LOADIN BAF ACC -2;
30 STOREIN SP ACC 1;
31 # Exp(Num('1'))
32 SUBI SP 1;
33 LOADI ACC 1;
34 STOREIN SP ACC 1;
35 LOADIN SP ACC 2;
36 LOADIN SP IN2 1;
37 SUB ACC IN2;
38 JUMP== 3;
39 LOADI ACC 0;
40 JUMP 2;
41 LOADI ACC 1;
42 STOREIN SP ACC 2;

```

<sup>27</sup>Wie z.B. `LOADI ACC GoTo(Name('addr@next_instr')), Exp(GoTo(Name('main.0')))` und `JUMP== GoTo(Name('if_else_after.2'))`.

```

43 ADDI SP 1;
44 # IfElse(Stack(Num('1')), [], [])
45 LOADIN SP ACC 1;
46 ADDI SP 1;
47 JUMP== 7;
48 # // not included Exp(GoTo(Name('if.3')))
49 # // Return(Name('res'))
50 # Exp(Stackframe(Num('1')))
51 SUBI SP 1;
52 LOADIN BAF ACC -3;
53 STOREIN SP ACC 1;
54 # Return(Stack(Num('1')))
55 LOADIN SP ACC 1;
56 ADDI SP 1;
57 LOADIN BAF PC -1;
58 # // Assign(Name('res'), BinOp(Name('n'), Mul('*'), Name('res')))
59 # Exp(Stackframe(Num('0')))
60 SUBI SP 1;
61 LOADIN BAF ACC -2;
62 STOREIN SP ACC 1;
63 # Exp(Stackframe(Num('1')))
64 SUBI SP 1;
65 LOADIN BAF ACC -3;
66 STOREIN SP ACC 1;
67 # Exp(BinOp(Stack(Num('2')), Mul('*'), Stack(Num('1'))))
68 LOADIN SP ACC 2;
69 LOADIN SP IN2 1;
70 MULT ACC IN2;
71 STOREIN SP ACC 2;
72 ADDI SP 1;
73 # Assign(Stackframe(Num('1')), Stack(Num('1')))
74 LOADIN SP ACC 1;
75 STOREIN BAF ACC -3;
76 ADDI SP 1;
77 # // Assign(Name('n'), BinOp(Name('n'), Sub('-'), Num('1')))
78 # Exp(Stackframe(Num('0')))
79 SUBI SP 1;
80 LOADIN BAF ACC -2;
81 STOREIN SP ACC 1;
82 # Exp(Num('1'))
83 SUBI SP 1;
84 LOADI ACC 1;
85 STOREIN SP ACC 1;
86 # Exp(BinOp(Stack(Num('2')), Sub('-'), Stack(Num('1'))))
87 LOADIN SP ACC 2;
88 LOADIN SP IN2 1;
89 SUB ACC IN2;
90 STOREIN SP ACC 2;
91 ADDI SP 1;
92 # Assign(Stackframe(Num('0')), Stack(Num('1')))
93 LOADIN SP ACC 1;
94 STOREIN BAF ACC -2;
95 ADDI SP 1;
96 # Exp(GoTo(Name('condition_check.5')))
97 JUMP -58;
98 # Return(Empty())
99 LOADIN BAF PC -1;

```

```
100 # StackMalloc(Num('2'))
101 SUBI SP 2;
102 # Exp(Num('4'))
103 SUBI SP 1;
104 LOADI ACC 4;
105 STOREIN SP ACC 1;
106 # NewStackframe(Name('faculty'), GoTo(Name('addr@next_instr'))
107 MOVE BAF ACC;
108 ADDI SP 3;
109 MOVE SP BAF;
110 SUBI SP 4;
111 STOREIN BAF ACC 0;
112 LOADI ACC 80;
113 ADD ACC CS;
114 STOREIN BAF ACC -1;
115 # Exp(GoTo(Name('faculty.6'))
116 JUMP -78;
117 # RemoveStackframe()
118 MOVE BAF IN1;
119 LOADIN IN1 BAF 0;
120 MOVE IN1 SP;
121 # Exp(ACC)
122 SUBI SP 1;
123 STOREIN SP ACC 1;
124 LOADIN SP ACC 1;
125 ADDI SP 1;
126 CALL PRINT ACC;
127 # Return(Empty())
128 LOADIN BAF PC -1;
```

**Code 1.12:** *RETI Pass für Codebespiel*

---

---

# Literatur

## Online

- *ANSI C grammar (Lex)*. URL: <https://www.lysator.liu.se/c/ANSI-C-grammar-1.html> (besucht am 29.07.2022).
- *ANSI C grammar (Yacc)*. URL: <http://www.quut.com/c/ANSI-C-grammar-y.html> (besucht am 29.07.2022).
- *ANSI C grammar (Yacc)*. URL: <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html> (besucht am 29.07.2022).
- *C Operator Precedence* - *cppreference.com*. URL: [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence) (besucht am 27.04.2022).
- *clang: C++ Compiler*. URL: <http://clang.org/> (besucht am 29.07.2022).
- *Clockwise/Spiral Rule*. URL: <https://c-faq.com/decl/spiral.anderson.html> (besucht am 29.07.2022).
- *GCC, the GNU Compiler Collection* - *GNU Project*. URL: <https://gcc.gnu.org/> (besucht am 13.07.2022).
- *Grammar: The language of languages (BNF, EBNF, ABNF and more)*. URL: <https://matt.might.net/articles/grammars-bnf-ebnf/> (besucht am 30.07.2022).

## Bücher

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

## Vorlesungen

- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [https://abs.informatik.uni-freiburg.de/src/teach\\_main.php?id=157](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157) (besucht am 09.07.2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).

## Sonstige Quellen

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).

- *Naming convention (programming)*. In: *Wikipedia*. Page Version ID: 1100066005. 24. Juli 2022. URL: [https://en.wikipedia.org/w/index.php?title=Naming\\_convention\\_\(programming\)&oldid=1100066005](https://en.wikipedia.org/w/index.php?title=Naming_convention_(programming)&oldid=1100066005) (besucht am 30.07.2022).