Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

asdf

asdf

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

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 **Konkretten Syntax** der Sprache L_{PicoC} hin zum einem Programm mit derselben **Semantik** in der **Konkretten Syntax** der Sprache L_{RETI} erklärt.

Für das Parsen[1] des Programmes in der **Konkretten Syntax** der Sprache L_{PicoC} wird das **Lark Parsing Toolkit**[2] [3] verwendet. Das **Lark Parsing Toolkit** ist eine Bibliothek, die es ermöglicht mittels eines in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** (Definition 1.3 bzw. für den Dialekt von Lark Definition 1.4) spezifizierten Grammatik der **Konkretten Syntax** ein Programm in ebendieser **Konkretten Syntax** zu parsen und daraus einen **Ableitungsbaum** für die kompilerintere Weiterverarbeitung zu generieren.

Definition 1.1: Metasyntax

*Steht für den **Aufbau** einer **Metasprache** (Definition 1.2), der durch eine **Grammatik** oder **Natürliche Sprache** beschrieben werden kann.*

Definition 1.2: Metasprache

*Eine Metasprache ist eine **Sprache**, die dazu genutzt wird **andere Sprachen** zu **beschreiben**[a].*

[a] Das „**Meta**“ drückt allgemein aus, dass sich etwas auf einer **höheren Ebene** befindet. Um über die Ebene sprechen zu können, in der man sich **selbst befindet**, muss man von einer **höheren, außenstehenden Ebene** darüber reden.

Definition 1.3: Erweiterte Backus-Naur-Form (EBNF)

*Die Erweiterte Backus-Naur-Form[a] ist eine **Metasyntax** (Definition 1.1) die dazu verwendet wird **Kontextfreier Grammatiken** darzustellen.*[b] [c]

*Die Erweiterte Backus-Naur-Form ist zwar **standartisiert** und die Spezifikation des Standards kann unter Link[d] aufgefunden werden, allerdings werden in der Praxis, wie z.B. in Lark oft **eigene Notationen** verwendet.*

[a] Der Name kommt daher, dass es eine **Erweiterung** der **Backus-Naur-Form** ist, die hier allerdings **nicht** weiter erläutert wird.
[b] Nebel, „Theoretische Informatik“.
[c] *Grammar Reference — Lark documentation*.
[d] `https://standards.iso.org/ittf/PubliclyAvailableStandards/`.

[1] Wobei beim **Parsen** auch das **Lexen** inbegriffen ist.
[2] *Lark - a parsing toolkit for Python*.
[3] Shinan, *lark*.

Definition 1.4: Dialekt der EBNF aus Lark

*Das **Lark Parsing Toolkit** verwendet eine **eigene Notation** für die **Erweiterte Backus-Naur-Form**, die sich teilweise in einzelnen Aspekten von der Syntax aus dem **Standard** unterscheidet und unter Link[a] dokumentiert ist.*

*Ein für die Grammatiken des PicoC-Compilers wichtiger **Unterschied** ist z.B., dass dieser Dialekt anstelle von **geschweiften Klammern** {} für die Darstellung von **Wiederholung**, den aus **regulären Ausdrücken** bekannten ∗-Quantor optional zusammen mit **runden Klammern** `()` verwendet: `()`∗.[b]*

[a] `https://lark-parser.readthedocs.io/en/latest/grammar.html`.
[b] Bzw. kann der ∗-Quantor auch **keinmal** wiederholen bedeuteten.

Das **Lark Parsing Toolkit** wurde vor allem deswegen gewählt, weil es sehr **einfach in der Verwendung** ist. Andere derartige Tools, wie z.B. **ANTLR**[4] sind **Parser Generatoren**, die zur **Konkretten Syntax** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkretten Syntax** direkt ein Programm in dieser Konkretten Syntax **parsen** und einen **Ableitungsbaum** dafür generieren zu können.

Eine möglichst **geringe Laufzeit** durch Verwenden der effizientesten Algorithmen zu erreichen war keine der Hauptzielsetzungen für den **PicoC-Compiler**, da der **PicoC-Compiler** vor allem als **Lerntool** konzipiert ist, mit dem Studenten lernen können, wie der **Kompiliervorgang** von der Programmiersprache L_{PicoC} zur Maschinensprache L_{RETI} funktioniert. Eine ausführliche Diskussion zur Priorisierung Laufziet wurde in Unterkapitel **??** geführt. Lark besitzt des Weiteren eine **sehr gute Dokumentation** *Welcome to Lark's documentation! — Lark documentation*, sodass anderen Studenten, die den **PicoC-Compiler** vielleicht in ihr Projekt einbinden wollen, unkompliziert **Erweiterungen** für den **PicoC-Compiler** schreiben können.

Neben den **Konkretten Syntaxen**[5], die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterter Back-Naur-Form** spezifiziert sind, werden in den folgenden Unterkapiteln die **Abstrakte Syntaxen**, welche spezifzieren, welche Kompositionen für die **Abstrakter Syntaxbaums** der verschiedenden **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben, die allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit** hat.

Die Notation für die **Abstrakte Syntax** unterscheidet sich bewusst von der **Erweiterten Backus-Naur-Form**, da in der Abstrakten Syntax **Kompositionen von Knoten** beschrieben werden, die **klar auszumachen** sind, wodurch es die Grammatik nur **unnötig verkomplizieren** würde, wenn man die **Erweiterte Backus-Naur-Form** verwenden würde. Es gibt leider **keine** Standardnotation für die **Abstrakte Syntax**, die sich deutlich durchgesetzt hat, daher wird für die Abstrakte Syntaxen eine eigene **Abstract Syntax Form Notation** (Definition 1.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrette** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 1.5: Abstrakte Syntax Form (ASF)

*Die Abstrakte Syntax Form ist eine eigene **Metasyntax** für die **Grammatiken** von **Abstrakten Syntaxen**, die für diese Bachelorarbeit definiert wurde und sich von dem **Dialekt** der **Backus-Naur-Form** des **Lark Parsing Toolkit** nur dadurch unterschiedet, dass **Terminalsymbole** nicht von `""` engeschlossen sein müssen, da die **Knoten** in der **Abstrakten Syntax**, sowieso schon **klar auszumachen** sind und von anderen Symbolen der **Metasprache** leicht zu unterschiden sind.*

Letztendlich geht es allerdings nur darum, dass aufgrund der Verwendung des **Lark Parsing Toolkit** die **Konkrette Syntax** in einem eigenen Dialekt der **Erweiterter Backus-Naur-Form** angegeben sein muss

[4] *ANTLR*.
[5] Der **Plural** von Syntax ist **Syntaxen**, wie es in Quelle *Syntax* verifiziert werden kann.

und für das Implementieren der Passes die **Abstrakte Syntax** für den **Programmierer** möglichst **einfach verständlich** sein sollte, weshalb sich die **Abstrake Syntax Form** gut dafür eignet.

1.1 Lexikalische Analyse

Für die **Lexikalische Analyse** ist es nur notwendig eine Grammatik zu definieren, die den Teil der **Konkretten 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** dazu verwendet ein Programm in **Konkretter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel **??** erläutert ist.

1.1.1 Konkrette 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" | DIG_NO_0 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 **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 **Konkretten Syntax** für die **Syntaktischen Analayse** verwendet werden, um z.B. zu beschreiben, in welchem Kontext z.B. eine Zahl `NUM` stehen darf.

Die in der Konkrette 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[6], 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.

Die Implementierung des **Basic Lexer** aus dem **Lark Parsing Toolkit** ist unter Link[7] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten und ist aufgrund dessen, dass sie in der Lage ist nach einer spezifizierten Grammatik zu lexen, **zu komplex**, um sie an dieser Stelle allgemein

[6] Es wird immer die **höchste** Priorität **zuerst** genommen.
[7] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.

erklären zu können.

Der **Basic Lexer** verhält sich allerdings grundlegend so, wie es im Unterkapitel **??** erklärt wurde, allerdinds berücksichtigt der **Basic Lexer** ebenfalls **Priortiäten**, 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.

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

Grammatik 1.1.1: *Grammatik der Konkretten 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 **Abstrakter Syntaxbaums** in seinen einzelnen **Zwischenschritten** zu erläutern.

```c
struct st {int *(*attr)[4][5];};

void main() {
  struct st *(*var[3][2]);
}
```

Code 1.1: *PicoC-Code des Codebeispiels*

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

```
[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 **Konkretter Syntax** unter Verwendung der **Tokens** aus der **Lexikalischen Analyse** einen **Ableitungsbaum** zu generieren. Es ist danach die Aufgabe möglicher **Visitors** und die Aufgabe des **Transformers** aus diesem **Ableitungsbaum** einen **Abstrakter Syntaxbaum** in **Abstrakter Syntax** zu generieren.

1.2.1 Umsetzung von Präzidenz und Assoziativität

Die Programmiersprache L_{PicoC} hat dieselben **Präzidenzregeln** implementiert, wie die Programmiersprache L_C[8]. Die **Präzidenzregeln** der Programmiersprache L_{PicoC} sind in Tabelle 1.1 aufgelistet.

[8] *C Operator Precedence - cppreference.com*.

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

Tabelle 1.1: *Präzidenzregeln von PicoC*

Würde man diese **Operatoren** ohne Beachtung von **Präzidenzreglen** (Definiton **??**) 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*")"	*L_Arith +*
un_op	::=	"−" | "∼" | "!" | "∗" | "&"	
un_exp	::=	*un_op exp*	
bin_op	::=	"∗" | "/" | "%" | "+" | "−" | "&" | "∧" | "|"	
		| "<" | "<=" | ">" | ">=" | "!=" | "==" | "&&" | "||"	
bin_exp	::=	*exp bin_op exp*	
exp	::=	*prim_exp* | *un_exp* | *bin_exp*	

Grammatik 1.2.1: *Undurchdachte Konkrette 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.

$$\begin{aligned} \texttt{exp} &\Rightarrow \texttt{bin\_exp} \Rightarrow \texttt{exp bin\_op exp} \Rightarrow \texttt{bin\_exp bin\_op exp} \\ &\Rightarrow \texttt{exp bin\_op exp bin\_op exp} \Rightarrow^* \texttt{3 * 1 \&\& 4} \end{aligned} \tag{1.2.1}$$

$$\begin{aligned} &\texttt{exp} \Rightarrow \texttt{bin\_exp} \Rightarrow \texttt{exp bin\_op exp} \Rightarrow \texttt{prim\_exp bin\_op exp} \Rightarrow \texttt{NUM bin\_op exp} \\ &\quad \Rightarrow \texttt{3 bin\_op exp} \Rightarrow \texttt{3 * exp} \Rightarrow \texttt{3 * bin\_exp} \Rightarrow \texttt{3 * exp bin\_exp exp} \Rightarrow^* \texttt{3 * 1 \&\& 4} \end{aligned} \tag{1.2.2}$$

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 darsgestellt 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.

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

Grammatik 1.2.2: *Durchdachte Konkrette 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)`[9] 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` `~a`, `*a` und `&a` ist die **Alternative** `un_op un_exp` dafür zuständig, dass diese unären Operatoren **rechtsassoziativ** geschachtelt werden können (z.B. `!-~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.

$$un\_exp \quad ::= \quad un\_op\ un\_exp \quad | \quad post\_exp$$

Grammatik 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 auschließlich einen Ausdruck der **höchsten Präzidenz** (z.B. `42`) zu haben.

$$post\_exp \quad ::= \quad post\_exp\text{"["}logic\_or\text{"]"} \quad | \quad post\_exp\text{"."}name \quad | \quad name\text{"("}fun\_args\text{")"} \quad | \quad prim\_exp$$

Grammatik 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[10] (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzidenzstufe 4

[9] **Geklammerte Ausdrücke** werden nämlich von `prim_exp` erkannt, welches eine höhere **Präzidenzstufe** hat.
[10] Bezogen auf Tabelle 1.1.

haben). Das **Nicht-Terminalsymbol** `arith_prec1` auf der **rechten Seite** ermöglicht es, dass zwischen den 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 auschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B. `1 / 4`).

$$arith\_prec2 \quad ::= \quad arith\_prec2 \;\; prec2\_op \;\; arith\_prec1 \quad | \quad arith\_prec1$$

Grammatik 1.2.5: *Beispiel für eine linksassoziative Produktion*

Anmerkung

Manche **Parser**[a] 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.

$$arith\_prec2 \quad ::= \quad arith\_prec1 \;\; (prec2\_op \;\; arith\_prec1)*$$

Grammatik 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 Produktion1.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 1.3) 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 auschließlich ein Ausdruck **höherer Präzidenz** erkannt wird (z.B. `1 / 4`).

[a] 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.

$prec1\_op$	::=	$"*" \mid "/" \mid "\%"$	*L_Misc*
$prec2\_op$	::=	$"+" \mid "-"$	
rel_op	::=	$"<" \mid "<=" \mid ">" \mid ">="$	
eq_op	::=	$"==" \mid "!="$	
fun_args	::=	$[logic\_or(",", logic\_or)*]$	
$prim\_exp$	::=	$name \mid NUM \mid CHAR \mid "("logic\_or")"$	*L_Arith*
$post\_exp$	::=	$post\_exp"["logic\_or"]" \mid post\_exp"."name \mid name"("fun\_args")"$	*+ L_Array*
	$\mid$	$prim\_exp$	*+ L_Pntr*
un_exp	::=	$un\_op\ un\_exp \mid post\_exp$	*+ L_Struct*
$arith\_prec1$	::=	$arith\_prec1\ prec1\_op\ un\_exp \mid un\_exp$	*+ L_Fun*
$arith\_prec2$	::=	$arith\_prec2\ prec2\_op\ arith\_prec1 \mid arith\_prec1$	
$arith\_and$	::=	$arith\_and\ "\&"\ arith\_prec2 \mid arith\_prec2$	
$arith\_oplus$	::=	$arith\_oplus\ "\wedge"\ arith\_and \mid arith\_and$	
$arith\_or$	::=	$arith\_or\ "|"\ arith\_oplus \mid arith\_oplus$	
rel_exp	::=	$rel\_exp\ rel\_op\ arith\_or \mid arith\_or$	*L_Logic*
eq_exp	::=	$eq\_exp\ eq\_op\ rel\_exp \mid rel\_exp$	
$logic\_and$	::=	$logic\_and\ "\&\&"\ eq\_exp \mid eq\_exp$	
$logic\_or$	::=	$logic\_or\ "||"\ logic\_and \mid logic\_and$	
$assign\_stmt$	::=	$un\_exp\ "="\ logic\_or";"$	*L_Assign*

Grammatik 1.2.7: *Durchdachte Konkrette Syntax für Operatorpräzidenz in EBNF*

1.2.2 Konkrette Syntax für die Syntaktische Analyse

Die gesamte Grammatik 1.2.8, welche die **Konkrette 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 **Konkrette Syntax** an die **aktuellste konstenlos auffindbare Version** der echten Grammatik *ANSI C grammar (Yacc)* der Sprache L_C angepasst[11], 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**[12] und **Clang**[13] zu nennen wären.

In der Grammatik 1.2.8, welche die **Konkrette Syntax** der Sprache L_{PicoC} für die **Syntaktische Analyse** beschreibt, werden einige der **Tokennamen** aus der Grammatik 1.1.1 der **Konkretten 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 **Konkretten 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.

[11] 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 (Lex)* und **Lexikalische Analyse** **noauthor'ansi'nodate-2** aus dem Jahre 1985 zeigen.
[12] *GCC, the GNU Compiler Collection - GNU Project*.
[13] *clang: C++ Compiler*.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith + L_Array*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*+ L_Pntr + L_Struct*
		| *input_exp* | *print_exp* | *prim_exp*	*+ L_Fun*
un_exp	::=	*un_op un_exp* | *post_exp*	
input_exp	::=	"*input*""("")"	
print_exp	::=	"*print*""("*logic_or*")"	
*arith_prec*1	::=	*arith_prec*1 *prec*1*_op un_exp* | *un_exp*	
*arith_prec*2	::=	*arith_prec*2 *prec*2*_op arith_prec*1 | *arith_prec*1	
arith_and	::=	*arith_and* "&" *arith_prec*2 | *arith_prec*2	
arith_oplus	::=	*arith_oplus* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_oplus* | *arith_oplus*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_op rel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
type_spec	::=	*prim_dt* | *struct_spec*	*L_Assign_Alloc*
alloc	::=	*type_spec pntr_decl*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	
initializer	::=	*logic_or* | *array_init* | *struct_init*	
init_stmt	::=	*alloc* "=" *initializer*";"	
const_init_stmt	::=	"*const*" *type_spec name* "=" *NUM*";"	
pntr_deg	::=	"*"*	*L_Pntr*
pntr_decl	::=	*pntr_deg array_decl* | *array_decl*	
array_dims	::=	("["*NUM*"]")*	*L_Array*
array_decl	::=	*name array_dims* | "("*pntr_decl*")"*array_dims*	
array_init	::=	"{"*initializer*("," *initializer*) * "}"	
array_subscr	::=	*post_exp*"["*logic_or*"]"	
struct_spec	::=	"*struct*" *name*	*L_Struct*
struct_params	::=	(*alloc*";")+	
struct_decl	::=	"*struct*" *name* "{"*struct_params*"}"	
struct_init	::=	"{""."*name*"="*initializer*	
		("," "."name"="initializer)*"}"	
struct_attr	::=	*post_exp*"."*name*	
if_stmt	::=	"*if*""("*logic_or*")" *exec_part*	*L_If_Else*
if_else_stmt	::=	"*if*""("*logic_or*")" *exec_part* "*else*" *exec_part*	
while_stmt	::=	"*while*""("*logic_or*")" *exec_part*	*L_Loop*
do_while_stmt	::=	"*do*" *exec_part* "*while*""("*logic_or*")"";"	

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

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

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

Anmerkung

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 Ableitungsbaum Generierung

Die in Unterkapitel 1.2.2 definierte **Konkrette Syntax**, die von der Grammatik 1.2.8 beschrieben wird lässt sich mithilfe des **Earley Parsers** (Definition 1.6) von Lark dazu verwenden Code, der in der Sprache L_{PicoC} geschrieben ist zu parsen um einen **Ableitungsbaum** zu generieren.

Definition 1.6: Earley Parser

*Ist ein Algorithmus für das **Parsen** von **Wörtern** einer **Kontextfreien Sprache**, der ein **Chart Parser** ist, welcher einen mittels **Dynamischer Programmierung** und dem **Top-Down Ansatz** arbeitenden **Earley Recognizer** (Defintion 1.7) nutzt, um einen **Ableitungsbaum** zu konstruieren.*

*Zur **Konstruktion des Ableitungsbaumes** muss dafür gesorgt werden, dass der **Earley Recognizer** bei der **Vervollständigungsoperation** Zeiger auf den **vorherigen Zustand** hinzugefügt, um durch **Rückwärtsverfolgen** dieser **Zeiger** die **Ableitung** wieder **nachvollziehen** zu können und so einen **Ableitungsbaum** konstruieren zu können.*[a]

[a] Earley, „An efiicient context-free parsing“.

Definition 1.7: Earley Recognizer

*Ist ein **Recognizer**, der für **alle Kontextfreien Sprachen** das **Wortproblem** entscheiden kann und dies mittels **Dynamischer Programmierung** mit dem **Top-Down Ansatz** umsetzt.*[a]

***Eingabe** und **Ausgabe** des Algorithmus sind:*

- ***Eingabe:** **Eingabewort** w und **Grammatik** $G_{Parse} = \langle N, \Sigma, P, S \rangle$*
- ***Ausgabe:** 0 wenn $w \notin L(G_{Parse})$[b] und 1 wenn $w \in L(G_{Parse})$*

*Bevor dieser **Algorithmus** erklärt wird müssen noch einige **Symbole** und **Notationen** erklärt werden:*

- *α, β, γ stellen eine **beliebige Folge** von **Grammatiksymbolen**[c] dar*
- *A und B stellen **Nicht-Terminalsymbole** dar*
- *a stellt ein **Terminalsymbol** dar*
- ***Earley's Punktnotation:** $A ::= \alpha \bullet \beta$ stellt eine **Produktion**, in der α **bereits geparst** wurde und β **noch geparst** werden muss*
- *Die **Indexierung** ist informell ausgedrückt so umgesetzt, dass die Indices **zwischen Tokennamen** liegen, also **Index** 0 **vor** dem ersten **Tokennamen** verortet ist, **Index** 1 **nach** dem ersten **Tokennamen** verortet ist und **Index** n **nach** dem **letzten Tokennamen** verortet ist*

*und davor müssen noch einige **Begriffe definiert** werden:*

- ***Zustandsmenge:** Für jeden der $n+1$ **Indices** j wird eine **Zustandsmenge** $Z(j)$ generiert*
- ***Zustand einer Zustandsmenge:** Ist ein **Tupel** $(A ::= \alpha \bullet \beta, i)$, wobei $A ::= \alpha \bullet \beta$ die **aktuelle Produktion** ist, die bis **Punkt** $\bullet$ geparst wurde und i der **Index** ist, ab welchem der Versuch der Erkennung eines **Teilworts** des **Eingabeworts** mithilfe dieser **Produktion begann***

*Der **Ablauf** des Algorithmus ist wie folgt:*

1. ***initialisiere** $Z(0)$ mit der **Produktion**, welches das **Startsymbol** S auf der **linken Seite** des **„kann abgeleitet werden zu"-Symbols** $::=$ hat*
2. *es werden in der **aktuellen Zustandsmenge** $Z(j)$ die folgenden **Operationen ausgeführt**:*
   - ***Voraussage:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet B\gamma, i)$ hat, wird für jede **Produktion** $(B ::= \beta)$ in der Grammatik, die ein B auf der **linken Seite** des **„kann abgeleitet werden zu"-Symbols** $::=$ hat ein **Zustand** $(B ::= \bullet\beta, j)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt*
   - ***Überprüfung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(A ::= \alpha \bullet a\gamma, i)$ hat wird der **Zustand** $(A ::= \alpha a \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j+1)$ hinzugefügt*
   - ***Vervollständigung:** Für jeden **Zustand** in der **Zustandsmenge** $Z(j)$, der die Form $(B ::= \beta\bullet, i)$ hat werden alle **Zustände** in $Z(i)$ gesucht, welche die Form $(A ::= \alpha \bullet B\gamma, i)$ haben und es wird der **Zustand** $(A ::= \alpha B \bullet \gamma, i)$ zur **Zustandsmenge** $Z(j)$ hinzugefügt*

   bis:

   - *der **Zustand** $(A ::= \beta\bullet, 0)$ in der **Zustandsmenge** $Z(n)$ auftaucht, wobei A das **Startsym-***

bol S *ist* $\Rightarrow w \in L(G_{Parse})$

- ***keine Zustände*** *mehr hinzugefügt werden können* $\Rightarrow w \notin L(G_{Parse})$

[a]Earley, „An efiicient context-free parsing“.
[b]$L(G_{Parse})$ ist die **Sprache**, welche durch die **Grammatik** G_{Parse} beschrieben wird.
[c]Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

1.2.3.1 Codebeispiel

Der **Ableitungsbaum**, 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: *Ableitungsbaum nach Ableitungsbaum Generierung*

1.2.3.2 Ausgabe des Ableitunsgbaums

Die Ausgabe des **Ableitungsbaums** 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** `::=`[14] entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition **??** eines **Ableitungsbaums** auch schon definiert ist. Die **EBNF-Grammatik** 1.2.8 des **PicoC-Compilers** erlaubt es allerdings auch, dass in einem **Blatt** garnichts ε steht, weil es z.B. **Produktionen**, wie `array_dims ::= ("["NUM"]")*` gibt, in denen auch das **leere Wort** ε abgeleitet werden kann.

Die Ausgabe des **Abstrakter Syntaxbaum** ist bewusst so gewählt, dass sie sich optisch vom **Ableitungsbaum** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase** geschrieben sind, im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, wie auch die **Nicht-Terminalsymbole** auf den **linken Seiten** des **„kann abgeleitet werden zu“-Symbols** `::=`.

1.2.4 Ableitungsbaum Vereinfachung

Der **Ableitungsbaum** in Code 1.3, dessen Generierung in Unterkapitel 1.2.3.1 besprochen wurde ist noch untauglich, damit aus ihm mittels eines **Tramsformers** ein **Abstrakter Syntaxbaum** generiert werden kann. Das Problem ist, dass um den den **Datentyp** einer Variable in der Programmiersprache L_C und somit auch die Programmiersprache L_{PicoC} korrekt bestimmen zu können, wie z.B. ein **„Feld der Mächtigkeit** 3 **von Pointern auf Felder der Mächtigkeit** 2 **von Integern“** `int (*ar[3])[2]` die **Spiralregel**[15] in der Implementeirung des **PicoC-Compilers** umgesetzt werden muss und das ist nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Grammatik 1.2.8 der **Konkretten 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 **Ableitungsbaum**, wie er in Abbildung 1.2 zu sehen ist.

[14] *Grammar: The language of languages (BNF, EBNF, ABNF and more).*
[15] *Clockwise/Spiral Rule.*

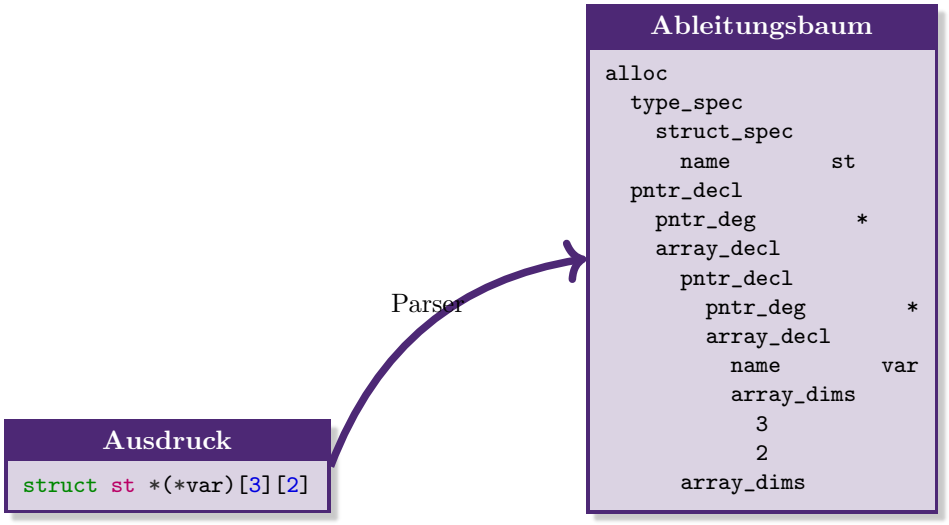

Abbildung 1.2: *Ableitungsbaum nach Parsen eines Ausdrucks*

Dieser **Ableitungsbaum** für den Ausdruck `struct st *(*var[3][2])` hat allerdings einen Aufbau welcher durch die **Syntax** der **Pointerdeklaratoren** `pntr_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** `pntr_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.

Des Weiteren befindet sich in der **Mitte** dieser **Spirale**, die der **Ableitungsbaum** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`, da der **Ableitungsbaum** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Programms** in **Konkretter 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 **Ableitungsbaum** aus Abbildung 1.2 mithilfe eines **Visitors** (Definition **??**) **vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des **Visitors** aus dem **Lark Parsing Toolkit** ist unter Link[16] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Visitor** verhält sich allerdings grundlegend so, wie es in Definition **??** erklärt wurde.

[16] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.

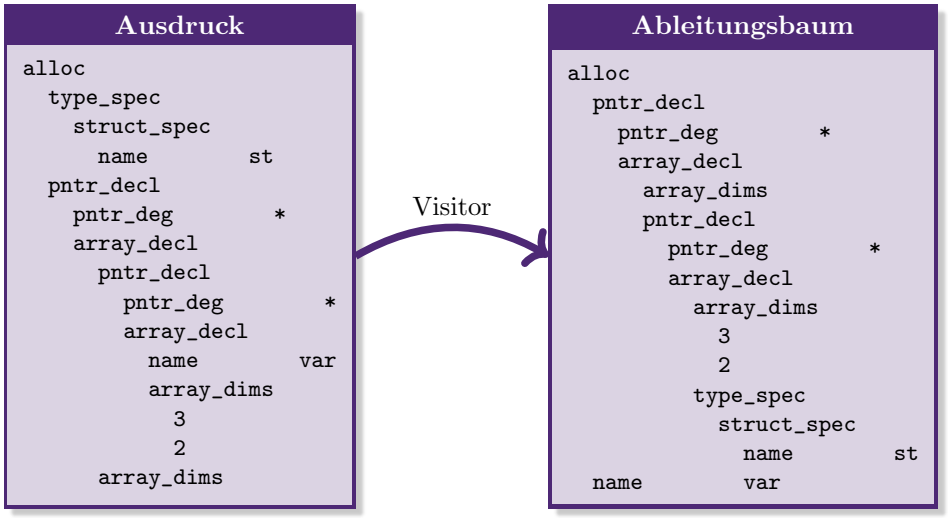

Abbildung 1.3: *Ableitungsbaum nach Vereinfachung*

1.2.4.1 Codebeispiel

In Code 1.4 ist der **Ableitungsbaum** 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              pntr_decl
10               pntr_deg        *
11               array_decl
12                 array_dims
13                   4
14                   5
15                 pntr_decl
16                   pntr_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         pntr_deg
27         name        main
28         fun_params
29         decl_exec_stmts
```

```
30          decl_part
31            decl_exp_stmt
32              alloc
33                pntr_decl
34                  pntr_deg        *
35                  array_decl
36                    array_dims
37                    pntr_decl
38                      pntr_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: *Ableitungsbaum nach Ableitungsbaum Vereinfachung*

1.2.5 Abstrakt Syntax Tree Generierung

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

Die Implementierung des **Transformers** aus dem **Lark Parsing Toolkit** ist unter Link[17] zu finden ist. Diese Implementierung ist allerdings **zu spezifisch** auf Lark zugeschnitten, um sie an dieser Stelle allgemein erklären zu können. Der **Transformer** verhält sich allerdings grundlegend so, wie es in Definition **??** erklärt wurde.

Den Teilbaum, der den Datentyp darstellt würde man von von **oben-nach-unten**[18] als **„Pointer auf einen Pointer auf ein Feld der Mächtigkeit 2 von Feldern der Mächtigkeit 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 **„Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Pointern auf einen Pointer auf einen Struct vom Typ `st`“** ist.

[17] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.
[18] In der Informatik wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**, bzw. in diesem Beispiel von **links-nach-rechts**.

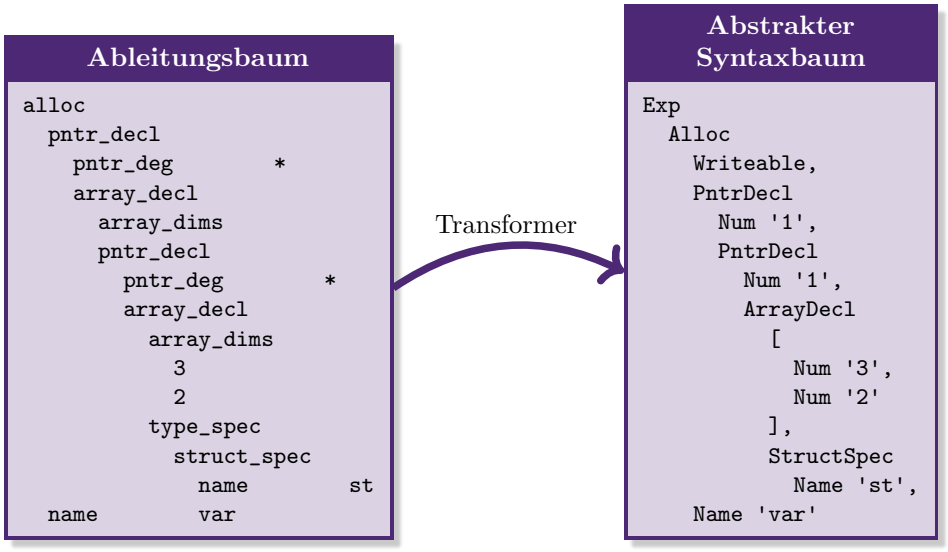

Abbildung 1.4: *Abstrakter Syntaxbaum Generierung ohne Umdrehen*

Dieser **Abstrakter Syntaxbaum** ist für die Weiterverarbeitung ungeeignet, denn für die **Adressberechnung** für eine Aneinandereihung 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 **Abstrakter Syntaxbaum** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie in Abbildung 1.5 zustande, der für die Weiterverarbeitung geeignet ist.

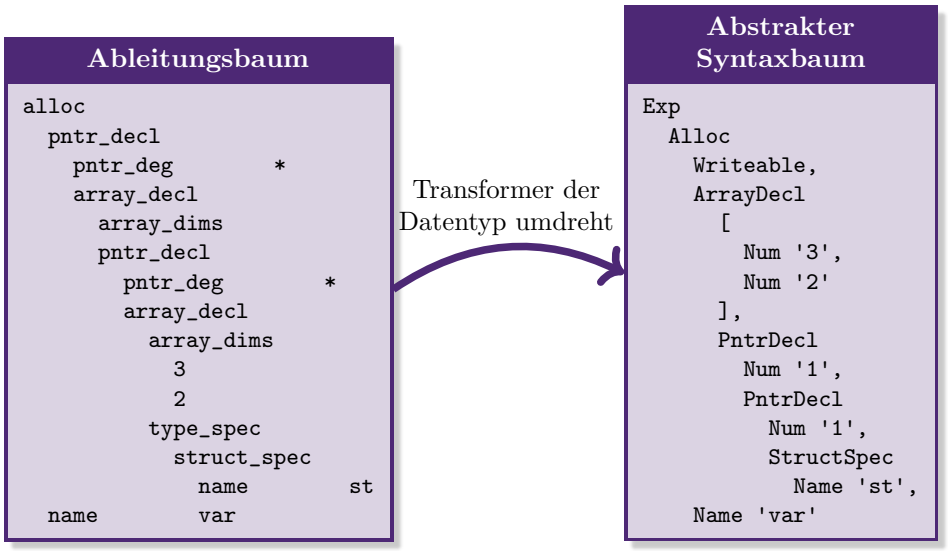

Abbildung 1.5: *Abstrakter Syntaxbaum Generierung mit Umdrehen*

Die Weiterverarbeitung des **Abstrakter Syntaxbaums** geschieht mithilfe von **Passes**, welche im Unterkapitel **??** genauer beschrieben werden. Da die Knoten des **Abstrakter Syntaxbaum** anders als beim **Ableitungsbaum** nicht die gleichen Bezeichnungen haben wie **Produktionen** der Grammatik der **Kon-**

kretten 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 **Abstrakter Syntaxbaums** 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[19]. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 1.3 mit einem **Bschreibungstext** dokumentiert.

[19] Z.B. steht der **PicoC-Knoten** `Name(str)` für einen **Bezeichner**. Anstatt diesen Knoten in englisch `Identifier(str)` zu nennen, wurde dieser als `Name(str)` gewählt, da `Name(str)` **kürzer** ist und **inuitiver verständlich**.

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

Tabelle 1.3: *PicoC-Knoten Teil 1*

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

Tabelle 1.4: *PicoC-Knoten Teil 2*

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

Tabelle 1.5: *PicoC-Knoten Teil 3*

PiocC-Knoten	Beschreibung
`FunDef(datatype, name, allocs, stmts_blocks)`	Container für eine **Funktionsdefinition**, z.B. `<datatype> <fun_name>(<datatype> <param>) {<stmts>}`, wobei `datatype` der **Rückgabewert** der Funktion ist, `name` der **Bezeichner** der Funktion ist, `allocs` die **Parameter** der Funktion sind, wobei der **Container-Knoten** `Alloc(type_spec, datatype, name)` als Cotainer für die Parameter dient und `stmts_blocks` eine Liste von **Statemetns** bzw. **Blöcken** ist, welche diese Funktion beinhaltet.
`NewStackframe(fun_name, goto_after_call)`	Container für die **Erstellung** eines neuen **Stackframes** und Speicherung des Werts des `BAF`-Registers der **aufrufenden Funktion** und der **Rücksprungadresse** nacheinander an den **Anfang** des neuen **Stackframes**. Das Attribut `fun_name` stehte dabei für den Bezeichner der Funktion, für die ein neuer **Stackframe** erstellt werden soll. Das Attribut `fun_name` dient später dazu den **Block** dieser Funktion zu finden, weil dieser für den weiteren Kompiliervorang wichtige Information in seinen versteckte Attributen gespeichert hat. Des Weiteren ist das Attribut `goto_after_call` ein `GoTo(Name('addr@next_instr'))`, welches später durch die **Adresse** des Befehls, der direkt auf die **Jump Instruction** folgt, ersetzt wird.
`RemoveStackframe()`	Container für das **Entfernen** des aktuellen **Stackframes** durch das **Wiederherstellen** des im noch **aktuellen Stackframe** gespeicherten Werts des `BAF`-Registes der **aufrufenden Funktion** und das Setzen des `SP`-Registers auf den Wert des `BAF`-Registesr **vor** der Wiederherstellung.
`File(name, decls_defs_blocks)`	Container für alle **Funkionen** oder **Blöcke**, welche eine Datei als Ursprung haben, wobei `name` der **Dateiname** der Datei ist, die erstellt wird und `decls_defs_blocks` eine Liste von **Funktionen** bzw. **Blöcken** ist.
`Block(name, stmts_instrs, instrs_before, num_instrs, param_size, local_vars_size)`	Container für **Statements**, der auch als **Block** bezeichnet wird, wobei das Attribut `name` der Bezeichners des **Labels** (Definition 1.8) des Blocks ist und `stmts_instrs` eine **Liste von Statements** oder **Instructions**. Zudem besitzt er noch 3 versteckte Attribute, wobei `instrs_before` die Zahl der **Instructions** vor diesem **Block** zählt, `num_instrs` die Zahl der Instructions ohne Kommentare in diesem Block zählt, `param_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **Parameter** der Funktion belegt werden müssen und `local_vars_size` die voraussichtliche Anzahl an Speicherzellen aufaddiert, die für die **lokalen Variablen** der Funktion belegt werden müssen.
`GoTo(name)`	Container für ein `Goto` zu einem anderen **Block**, wobei das Attribut `name` der Bezeichner des **Labels** des Blocks ist zu dem Gesprungen werden soll.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar**, den der Compiler selber während des **Kompiliervorangs** erstellt, der im **RETI-Interpreter** selbst später **nicht** sichtbar sein wird, aber in den **Immediate**-Dateien, welche die **Abstrakter Syntaxbaums** nach den verschiedenen **Passes** enthalten.
`RETIComment(value)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** später sichtbar sein wird und zur Orientierung genutzt werden kann, allerdings in einer tatsächlichen Implementierung einer **RETI-CPU nicht umsetzbar** ist und auch nicht sinnvoll wäre umzusetzen. Der **Kommentar** ist im Attribut **value**, welches jeder Knoten besitzt gespeichert.

Tabelle 1.6: *PicoC-Knoten Teil 4*

Definition 1.8: Label

*Durch einen **Bezeichner eindeutig** zuordenbares **Sprungziel** im Programmcode.*[a]

[a]Thiemann, „Compilerbau“.

Anmerkung

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 Kompiliervorgang **Informationen** transportiert, die später im Kompiliervorgang nicht mehr so leicht zugänglich wären.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** `value` und `position`, wobei **value** bei einem **Token-Knoten** (Definition 1.9) dem **Tokenwert** des Tokens, welches es ersetzt entspricht und bei **Container-Knoten** (Definition 1.10) unbesetzt ist. Das **Attribut** `position` wird später für Fehlermeldungen gebraucht.

Definition 1.9: Token-Knoten

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

***Token-Knoten** entsprechen im Abstrakter Syntaxbaum **Blättern**.*[a]

[a]Thiemann, „Compilerbau“.

Definition 1.10: 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 **Konkretten Syntax** abdecken, die einen **gleichen Aufbau** haben und sich auch unter einem **Überbegriff** zusammenfassen lassen.*[a]

***Container-Knoten** entsprechen im Abstrakter Syntaxbaum **Inneren Knoten**.*[b]

[a]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.
[b]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 erläutertem 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 **Abstrakter Syntaxbaums** vorkommen sind in Tabelle 1.2.5.1 mit einem **Beschreibungstext** dokumentiert.

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

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

Anmerkung

Um die obige Tabelle 1.8 nicht mit unnötig viel repetetiven Inhalt zu füllen, wurden die zahlreichen Kompostionen **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.

$stmt$	$::=$	$SingleLineComment(str, str) \mid RETIComment()$	*L_Comment*
un_op	$::=$	$Minus() \mid Not()$	*L_Arith*
bin_op	$::=$	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	$::=$	$Name(str) \mid Num(str) \mid Char(str)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
$stmt$	$::=$	$Exp(\langle exp\rangle)$	
un_op	$::=$	$LogicNot()$	*L_Logic*
rel	$::=$	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	$::=$	$LogicAnd() \mid LogicOr()$	
exp	$::=$	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle) \mid ToBool(\langle exp\rangle)$	
$type\_qual$	$::=$	$Const() \mid Writeable()$	*L_Assign_Alloc*
$datatype$	$::=$	$IntType() \mid CharType() \mid VoidType()$	
exp	$::=$	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
$stmt$	$::=$	$Assign(\langle exp\rangle, \langle exp\rangle)$	
$datatype$	$::=$	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
exp	$::=$	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
$datatype$	$::=$	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	$::=$	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
$datatype$	$::=$	$StructSpec(Name(str))$	*L_Struct*
exp	$::=$	$Attr(\langle exp\rangle, Name(str))$	
	$\mid$	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
$decl\_def$	$::=$	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
$stmt$	$::=$	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	$\mid$	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
$stmt$	$::=$	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	$\mid$	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	$::=$	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
$stmt$	$::=$	$Return(\langle exp\rangle)$	
$decl\_def$	$::=$	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
$file$	$::=$	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

Grammatik 1.2.10: *Abstrakte Syntax der Sprache* L_{PiocC}

Anmerkung

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 **Abstrakter Syntaxbaum** zu sehen, der aus dem **vereinfachten Ableitungsbaum** aus Code 1.4 mithilfe eines **Transformers** generiert wurde.

```
File
  Name './verbose_dt_simple_ast_gen_array_decl_and_alloc.ast',
  [
    StructDecl
      Name 'st',
      [
        Alloc
          Writeable,
          PntrDecl
            Num '1',
            ArrayDecl
              [
                Num '4',
                Num '5'
              ],
              PntrDecl
                Num '1',
                IntType 'int',
          Name 'attr'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '3',
                Num '2'
              ],
              PntrDecl
                Num '1',
                PntrDecl
                  Num '1',
                  StructSpec
                    Name 'st',
            Name 'var'
      ]
  ]
```

Code 1.5: *Aus vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum*

1.2.5.6 Ausgabe des Abstrakter Syntaxbaum

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

Das Ausgeben eines **Abstrakter Syntaxbaums** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`[20] der Programmiersprache **Python** umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkretten 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 **Konkrette Syntax** miteinander gemischt. Für **PicoC-Knoten** wurde die **Abstrakte Syntax** verwendet, da Passes schließlich auf **Abstrakter Syntaxbaums** operieren. Bei **RETI-Knoten** wurde die **Konkrette Syntax** verwendet, da **Maschienenbefehle** in **Konkretter 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 gescheifte Klammern `()` usw., ob man die **RETI-Knoten** in **Abstrakter** oder **Konkretter** Syntax schreibt. Daher kann man auch einfach gleich die **RETI-Knoten** in **Konkretter Syntax** ausgeben und muss nicht beim letzten **Pass** daran denken, am Ende die **Konkrette**, statt der **Abstrakten Syntax** für die **RETI-Knoten** auszugeben.

1.3 Code Generierung

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

Man steht nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** der Sprache L_{PicoC}, der durch die **Abstrakte Syntax** in Grammatik 1.2.10 spezifiziert ist in einen entsprechenden **Abstrakter Syntaxbaum** 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-Diagram** (siehe Unterkapitel **??**) in Abbildung 1.6 zu sehen ist, der **Python-Code** des **PicoC-Compilers** mittels eines Compilers, der z.B. auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte zu **Bytecode** kompiliert werden. Dieser **Bytecode** wird dann von der **Python-Virtual-Machine** (PVM) interpretiert, welche wiederum auf einer $\texttt{X}_{86\_64}$-Architektur laufen könnte. Und selbst dieses **T-Diagram** könnte noch ausführlicher ausgedrückt werden, indem nachgeforscht wird, in welcher Sprache eigentlich die **Python-Virtual-Machine** geschrieben war, bevor sie zu $\texttt{X}_{86\_64}$ kompiliert wurde usw.

[20] 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.

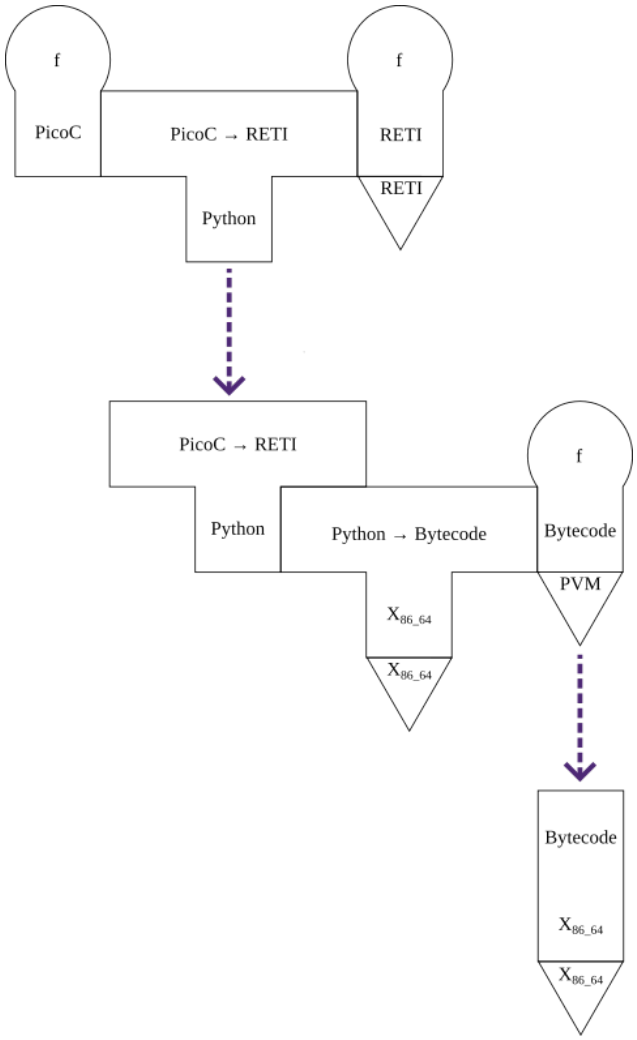

Abbildung 1.6: *Cross-Compiler Kompiliervorgang ausgeschrieben*

Dieses längliche **T-Diagram** in Abbildung 1.6 lässt sich zusammenfassen, sodass man das **T-Diagram** in Abbildung 1.7 erhält, in welcher direkt angegeben ist, dass der **PicoC-Compiler** in $\mathtt{X}_{86\_64}$-Maschienensprache geschrieben ist.

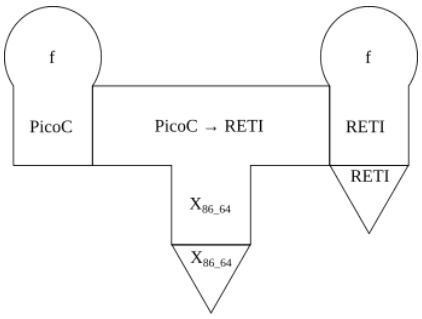

Abbildung 1.7: *Cross-Compiler Kompiliervorgang Kurzform*

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

Kompilierprozess im Folgenden im **horinzontalen**, 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 **Abstrakter Syntaxbaum** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstrakter Syntaxbaum** in seiner eigenen **Sprache** zu generieren.

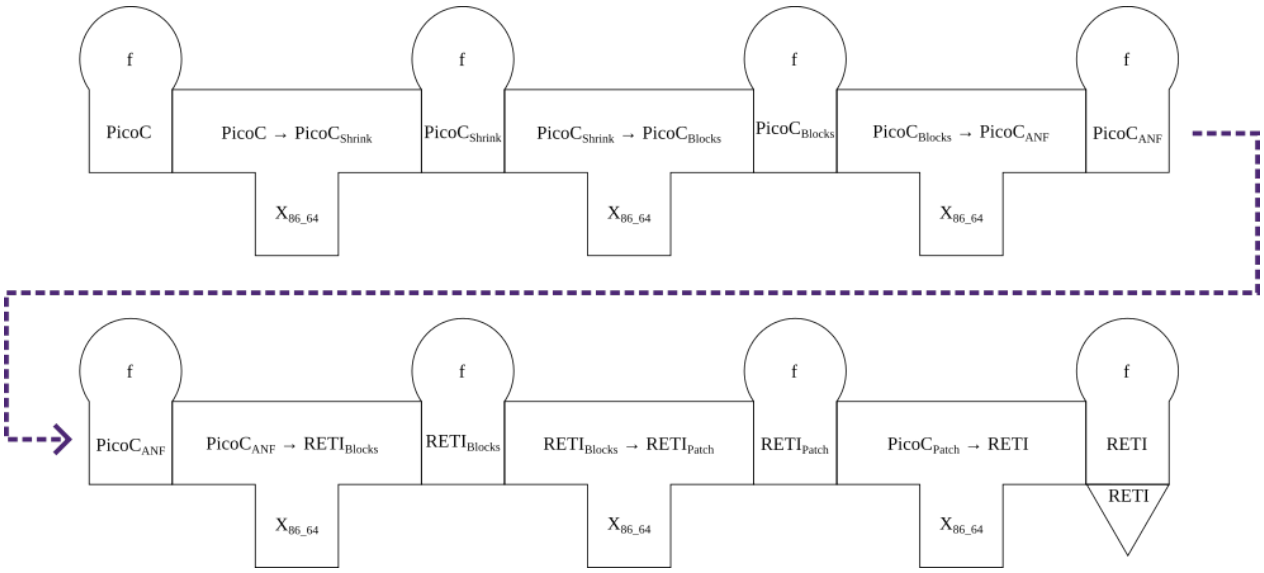

Abbildung 1.8: *Architektur mit allen Passes ausgeschrieben*

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 verwendenten **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[21]. 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[22] finden und mithilfe der im Ordner `/code_examples` beiliegenden `Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[23].

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 veschiedenen Aspekte[24] der

[21] Also die verschiedenen in den **Passes** generierten **Abstrakter Syntaxbaums**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[22] `https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples`.
[23] Es wurden zu diesem Zweck spezielle neue **Command-line Optionen** erstellt, die bestimmte Kommentare **herausfiltern** und manche Container-Knoten **einzeilig** machen, damit die generierten **Abstrakter Syntaxbaums** in den verscchiedenen Zwischenstufen der Kompilierung **nicht** zu langgestreckt und **überfüllt** mit Kommentaren sind.
[24] In kurz: **Pointer**, **Arrays**, **Strcuts** 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 **Abstrakter Syntaxbaum** in **Knoten** `Subscr(exp, i)` um, sodass sich dadurch viele vermeidbare **Fallunterscheidungen** und **doppelter Code** bei der Implementierung vermeiden lassen. Man lässt die **Derefenzierung** `*(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_{PiocC} in Tabelle 1.2.10, nach welcher der **erste** Abstrakter Syntaxbaum 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.

stmt	::=	$SingleLineComment(str, str) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(str) \mid Num(str) \mid Char(str)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
stmt	::=	$Exp(\langle exp\rangle)$	
un_op	::=	$LogicNot()$	*L_Logic*
rel	::=	$Eq() \mid NEq() \mid Lt() \mid LtE() \mid Gt() \mid GtE()$	
bin_op	::=	$LogicAnd() \mid LogicOr()$	
exp	::=	$Atom(\langle exp\rangle, \langle rel\rangle, \langle exp\rangle) \mid ToBool(\langle exp\rangle)$	
type_qual	::=	$Const() \mid Writeable()$	*L_Assign_Alloc*
datatype	::=	$IntType() \mid CharType() \mid VoidType()$	
exp	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(str))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(str), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(str)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(str))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(str))$	
	|	$Struct(Assign(Name(str), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))+)$	
stmt	::=	$If(\langle exp\rangle, \langle stmt\rangle*)$	*L_If_Else*
	|	$IfElse(\langle exp\rangle, \langle stmt\rangle*, \langle stmt\rangle*)$	
stmt	::=	$While(\langle exp\rangle, \langle stmt\rangle*)$	*L_Loop*
	|	$DoWhile(\langle exp\rangle, \langle stmt\rangle*)$	
exp	::=	$Call(Name(str), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*)$	
	|	$FunDef(\langle datatype\rangle, Name(str),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(str))*, \langle stmt\rangle*)$	
file	::=	$File(Name(str), \langle decl\_def\rangle*)$	*L_File*

Grammatik 1.3.1: *Abstrakte Syntax der Sprache* L_{PiocC_Shrink}

Anmerkung

Der rot markierte Knoten bedeutet, dass dieser im Vergleich zur voherigen **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** Algoirthmus ist allerdings **kein** gutes **Anschaungsbeispiel**, 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`-Statemtens zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter Link[25], 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 Codebespiel*

In Code 1.7 sieht man den **Abstrakter Syntaxbaum**, 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        [
```

[25] https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

```
8         Alloc(Writeable(), IntType('int'), Name('n'))
9       ],
10      [
11        Assign(Alloc(Writeable(), 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: *Abstrakter Syntaxbaum für Codebespiel*

Im **PicoC-Shrink-Pass** ändert sich nichts im Vergleich zum **Abstrakter Syntaxbaum** 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(lable)-` 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(lable)-` und `IfElse(exp, stmts1, stmts2)`-Knoten ersetzt wurden. Die **Funktionsdefinition** `FunDef(⟨datatype⟩, Name(str), Alloc(Writeable(), ⟨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.

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

Grammatik 1.3.2: *Abstrakte Syntax der Sprache* L_{PiocC_Blocks}

Anmerkung

Alles rot markierte bedeutet, es wurde **entfernt** oder **abgeändert**. Alles ausgegraute 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 **Konkretten 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[a].

Man bezeichnet hier die **Abstrakte Syntax** als **„Abstrakte Syntax der Sprache** L_{Picoc_Blocks}**“**. Diese Sprache L_{Picoc_Blocks} wird durch eine **Konkrette Syntax** beschrieben, die allerdings nicht weiter relevant ist, da in den **Passes** nur **Abstrakter Syntaxbaums** 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.

[a] 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 **PiocC-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.

```
File
  Name './example_faculty_it.picoc_blocks',
  [
    FunDef
      IntType 'int',
      Name 'faculty',
      [
        Alloc(Writeable(), IntType('int'), Name('n'))
      ],
      [
        Block
          Name 'faculty.6',
          [
            Assign(Alloc(Writeable(), IntType('int'), Name('res')), Num('1'))
            // While(Num('1'), [])
            GoTo(Name('condition_check.5'))
          ],
        Block
          Name 'condition_check.5',
          [
            IfElse
              Num '1',
              [
                GoTo(Name('while_branch.4'))
              ],
              [
                GoTo(Name('while_after.1'))
              ]
          ],
        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 Codebespiel*

1.3.1.3 PicoC-ANF Pass

1.3.1.3.1 Aufgabe

Die Aufgabe des **PicoC-ANF Passes** ist es den **Abstrakter Syntaxbaum** 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 1.11) 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**[26] 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. `identifier@scope`).[27]

Definition 1.11: 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**[a] 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 **Kompiliervorgangs** 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.*

[a] 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 **Abstrakter Syntaxbaum** 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.

[26] Bzw. der **Bezeichner**
[27] Die Umsetzung von **Scopes** wird in Unterkapitel **??** genauer beschrieben.

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

Grammatik 1.3.3: *Abstrakte Syntax der Sprache* L_{PiocC_ANF}

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 alls 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.

```
File
 Name './example_faculty_it.picoc_mon',
 [
   Block
     Name 'faculty.6',
     [
       // Assign(Name('res'), Num('1'))
       Exp(Num('1'))
       Assign(Stackframe(Num('1')), Stack(Num('1')))
       // While(Num('1'), [])
       Exp(GoTo(Name('condition_check.5')))
     ],
   Block
     Name 'condition_check.5',
     [
       // IfElse(Num('1'), [], [])
       Exp(Num('1')),
       IfElse
         Stack
           Num '1',
         [
           GoTo(Name('while_branch.4'))
         ],
         [
           GoTo(Name('while_after.1'))
         ]
     ],
   Block
     Name 'while_branch.4',
     [
       // If(Atom(Name('n'), Eq('==')), Num('1')), [])
       // IfElse(Atom(Name('n'), Eq('==')), Num('1')), [], [])
       Exp(Stackframe(Num('0')))
       Exp(Num('1'))
       Exp(Atom(Stack(Num('2')), Eq('==')), Stack(Num('1')))),
       IfElse
         Stack
           Num '1',
         [
           GoTo(Name('if.3'))
         ],
         [
           GoTo(Name('if_else_after.2'))
         ]
     ],
```

```
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 **Abstrakter Syntaxbaum** 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 `Exp(GoTo(str))`, `Block(Name(str), ⟨instr⟩*)` und `File(Name(str), ⟨block⟩*)`, da das gesamte Konzept mit den **Blöcken** erst im **RETI-Pass** in Unterkapitel 1.3.8 aufgelöst wird.

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

Grammatik 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 entsprechennden **RETI-Knoten** ersetzt werden.

```
File
  Name './example_faculty_it.reti_blocks',
  [
    Block
      Name 'faculty.6',
      [
        # // Assign(Name('res'), Num('1'))
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Assign(Stackframe(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN BAF ACC -3;
        ADDI SP 1;
        # // While(Num('1'), [])
        # Exp(GoTo(Name('condition_check.5')))
        Exp(GoTo(Name('condition_check.5')))
      ],
    Block
      Name 'condition_check.5',
      [
        # // IfElse(Num('1'), [], [])
        # 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 Codebespiel*

Anmerkung

Wenn der **Abstrakter Syntaxbaum** ausgegeben wird, ist die Darstellung nicht auschließlich in **Abstrakter Syntax**, da die **RETI-Knoten** aus bereits im Unterkapitel 1.2.5.6 vermitteltem Grund in **Konkretter Syntax** ausgeben 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 **Abstrakter Syntaxbaums**, 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.[28]
- 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

[28] 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	::=	$Add() \mid Addi() \mid Sub() \mid Subi() \mid Mult() \mid Multi() \mid Div() \mid Divi() \mid Mod() \mid Modi() \mid Oplus() \mid Oplusi() \mid Or() \mid Ori() \mid And() \mid Andi() \mid Load() \mid Loadin() \mid Loadi() \mid Store() \mid Storein() \mid Move()$	
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	::=	$File(Name(str), \langle block \rangle *)$	

Grammatik 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 **PiocC-Patch Passes** für das aus Unterkapitel 1.6 weitergeführte Beispiel. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>`-Block[29] eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-**Statements** entfernt[30], die nur einen Sprung um **eine** Position entsprochen 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;
```

[29] Dieser **Block** wurde im Code 1.8 markiert.
[30] 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 Codebespiel*

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 Konkrette 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 **Konkretter 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 **Konkrette 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.

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

Grammatik 1.3.6: *Konkrette Syntax der Sprache L_{RETI} für die Lexikalische Analyse in EBNF*

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

Grammatik 1.3.7: *Konkrette Syntax der Sprache L_{RETI} für die Syntaktische Analyse in EBNF*

reg	::=	*ACC*() | *IN*1() | *IN*2() | *PC*() | *SP*() | *BAF*()	*L_RETI*
		| *CS*() | *DS*()	
arg	::=	*Reg*(⟨*reg*⟩) | *Num*(*str*)	
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
		| *Always*() | *NOp*()	
op	::=	*Add*() | *Addi*() | *Sub*() | *Subi*() | *Mult*() | *Multi*()	
		| *Div*() | *Divi*() | *Mod*() | *Modi*() | *Oplus*() | *Oplusi*()	
		| *Or*() | *Ori*() | *And*() | *Andi*()	
		| *Load*() | *Loadin*() | *Loadi*() | *Store*() | *Storein*() | *Move*()	
instr	::=	*Instr*(⟨*op*⟩, ⟨*arg*⟩+) | *Jump*(⟨*rel*⟩, *Num*(*str*)) | *Int*(*Num*(*str*))	
		| *RTI*() | *Call*(*Name*('*print*'), ⟨*reg*⟩) | *Call*(*Name*('*input*'), ⟨*reg*⟩)	
		| *SingleLineComment*(*str*, *str*)	
		| *Instr*(*Loadi*(), [*Reg*(*Acc*()), *GoTo*(*Name*(*str*))]) | *Jump*(*Eq*(), *GoTo*(*Name*(*str*)))	
program	::=	*Program*(*Name*(*str*), ⟨*instr*⟩$*$)	
instr	::=	*Exp*(*GoTo*(*str*)) | *Exit*(*Num*(*str*))	*L_PicoC*
block	::=	*Block*(*Name*(*str*), ⟨*instr*⟩$*$)	
file	::=	*File*(*Name*(*str*), ⟨*block*⟩$*$)	

Grammatik 1.3.8: *Abstrakte Syntax der Sprache L_{RETI}*

1.3.1.6.3 Codebeispiel

Nach dem **RETI-Pass** ist das Programm komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkretten 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** auschließlich aus **RETI-Ausdrücken** bestehen[31], 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 **Abstrakter Syntaxbaum**, 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;
```

[31] 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-l.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).
- *ANTLR*. URL: https://www.antlr.org/ (besucht am 31. 07. 2022).
- *C Operator Precedence - cppreference.com*. URL: 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 Reference — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/grammar.html (besucht am 31. 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).
- *Welcome to Lark's documentation! — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/ (besucht am 31. 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).

Artikel

- Earley, Jay. „An efiicient context-free parsing“. In: 13 (1968). URL: https://web.archive.org/web/20040708052627/http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/p94-earley.pdf (besucht am 10. 08. 2022).

Vorlesungen

- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09. 07. 2022).
- Scholl, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- Thiemann, Prof. Dr. 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).
- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: https://github.com/lark-parser/lark (besucht am 31. 07. 2022).
- *Syntax*. In: *Wiktionary*. Page Version ID: 9196998. 7. Juni 2022. URL: https://de.wiktionary.org/w/index.php?title=Syntax&oldid=9196998 (besucht am 31. 07. 2022).