Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 13. September 2022

Autor:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis **I**

Codeverzeichnis **II**

Tabellenverzeichnis **III**

Definitionsverzeichnis **IV**

Grammatikverzeichnis **V**

1 Implementierung **1**
1.1 Lexikalische Analyse . . . 3
1.1.1 Konkrete Grammatik für die Lexikalische Analyse . . . 3
1.1.2 Codebeispiel . . . 5
1.2 Syntaktische Analyse . . . 6
1.2.1 Umsetzung von Präzedenz und Assoziativität . . . 6
1.2.2 Konkrete Grammatik für die Syntaktische Analyse . . . 11
1.2.3 Ableitungsbaum Generierung . . . 13
1.2.3.1 Codebeispiel . . . 14
1.2.3.2 Ausgabe des Ableitunsgbaumes . . . 15
1.2.4 Ableitungsbaum Vereinfachung . . . 15
1.2.4.1 Codebeispiel . . . 17
1.2.5 Generierung des Abstrakten Syntaxbaumes . . . 18
1.2.5.1 PicoC-Knoten . . . 20
1.2.5.2 RETI-Knoten . . . 25
1.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung . . . 27
1.2.5.4 Abstrakte Grammatik . . . 29
1.2.5.5 Codebeispiel . . . 30
1.2.5.6 Ausgabe des Abstrakten Syntaxbaumes . . . 31
1.3 Code Generierung . . . 32
1.3.1 Passes . . . 33
1.3.1.1 PicoC-Shrink Pass . . . 33
1.3.1.1.1 Abstrakte Grammatik . . . 34
1.3.1.1.2 Codebeispiel . . . 35
1.3.1.2 PicoC-Blocks Pass . . . 37
1.3.1.2.1 Abstrakte Grammatik . . . 37
1.3.1.2.2 Codebeispiel . . . 39
1.3.1.3 PicoC-ANF Pass . . . 40
1.3.1.3.1 Abstrakte Grammatik . . . 42
1.3.1.3.2 Codebeispiel . . . 44
1.3.1.4 RETI-Blocks Pass . . . 46
1.3.1.4.1 Abstrakte Grammatik . . . 46
1.3.1.4.2 Codebeispiel . . . 47
1.3.1.5 RETI-Patch Pass . . . 50
1.3.1.5.1 Abstrakte Grammatik . . . 50
1.3.1.5.2 Codebeispiel . . . 51
1.3.1.6 RETI Pass . . . 54

1.3.1.6.1 Konkrete und Abstrakte Grammatik 54
1.3.1.6.2 Codebeispiel 56

Literatur **A**

Abbildungsverzeichnis

1.1 Ableitungsbäume zu den beiden Ableitungen. 8
1.2 Ableitungsbaum nach Parsen eines Ausdrucks. 16
1.3 Ableitungsbaum nach Vereinfachung. 17
1.4 Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen. 19
1.5 Generierung eines Abstrakten Syntaxbaumes mit Umdrehen. 19
1.6 Kompiliervorgang Kurzform. 32
1.7 Architektur mit allen Passes ausgeschrieben. 33

Codeverzeichnis

1.1 PicoC-Code des Codebeispiels. 6
1.2 Tokens für das Codebeispiel. 6
1.3 Ableitungsbaum nach Ableitungsbaum Generierung. 14
1.4 Ableitungsbaum nach Ableitungsbaum Vereinfachung. 18
1.5 Aus einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum. 31
1.6 PicoC Code für Codebespiel. 36
1.7 Abstrakter Syntaxbaum für Codebespiel. 37
1.8 PicoC-Blocks Pass für Codebespiel. 40
1.9 Symboltabelle für Codebespiel. 44
1.10 PicoC-ANF Pass für Codebespiel. 46
1.11 RETI-Blocks Pass für Codebespiel. 50
1.12 RETI-Patch Pass für Codebespiel. 54
1.13 RETI Pass für Codebespiel. 58

Tabellenverzeichnis

1.1 Präzedenzregeln von PicoC. 7
1.2 Zuordnung der Bezeichnungen von Produktionsregeln zu Operatoren. 9
1.3 PicoC-Knoten Teil 1. 21
1.4 PicoC-Knoten Teil 2. 22
1.5 PicoC-Knoten Teil 3. 23
1.6 PicoC-Knoten Teil 4. 24
1.7 RETI-Knoten. 26
1.8 Kompositionen von PicoC-Knoten und RETI-Knoten mit besonderer Bedeutung. 28
1.9 Attribute eines Symboltabelleneintrags. 41

Definitionsverzeichnis

1.1 Metasyntax 1
1.2 Metasprache 1
1.3 Erweiterte Backus-Naur-Form (EBNF) 1
1.4 Dialekt der Erweiterten Backus-Naur-Form aus Lark 2
1.5 Abstrakte Syntaxform (ASF) 3
1.6 Earley Parser 13
1.7 Entarteter Baum 15
1.8 Symboltabelle 42

Grammatikverzeichnis

1.1.1 Konkrete Grammatik der Sprache L_{PicoC} für die Lexikalische Analyse in EBNF 5
1.2.1 Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet 7
1.2.2 Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet 8
1.2.3 Beispiel für eine unäre rechtsassoziative Produktion in EBNF 9
1.2.4 Beispiel für eine unäre linksassoziative Produktion in EBNF 9
1.2.5 Beispiel für eine binäre linksassoziative Produktion in EBNF 10
1.2.6 Beispiel für eine binäre linksassoziative Produktion ohne Linksrekursion in EBNF 10
1.2.7 Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet 11
1.2.8 Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 1 . . . 12
1.2.9 Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2 . . . 13
1.2.10 Abstrakte Grammatik der Sprache L_{PiocC} in ASF 30
1.3.1 Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF 35
1.3.2 Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF 38
1.3.3 Abstrakte Grammatik der Sprache L_{PiocC_ANF} in ASF 43
1.3.4 Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF 47
1.3.5 Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF 51
1.3.6 Abstrakte Grammatik der Sprache L_{RETI} in ASF 55
1.3.7 Konkrete Grammatik der Sprache L_{RETI} für die Lexikalische Analyse in EBNF 55
1.3.8 Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF 56

1 Implementierung

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

Für das Parsen[1] des Programmes in der **Konkreten Syntax** der Sprache L_{PicoC} wird das **Lark Parsing Toolkit**[2] [3] verwendet. Das **Lark Parsing Toolkit** ist eine Bibliothek, die es ermöglicht mittels einer in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** (Definition 1.3 bzw. für den Dialekt von Lark Definition 1.4) spezifizierten Konkreten Grammatik ein Programm in **Konkreter 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 **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 **Kontextfreie Grammatiken** darzustellen.*

*Am grundlegensten lässt sich die **Erweiterte Backus-Naur-Form** in **Kürze** wie folgt beschreiben:*[b] [c]

- ***Terminalsymbole** werden in **Anführungszeichen** "" geschrieben (z.B. `"term"`).*
- ***Nicht-Terminalsymbole** werden normal hingeschrieben (z.B. `non-term`).*
- ***Leerzeichen** ␣ dienen zur visuellen **Abtrennung** von **Grammatiksymbolen**[d].*

*Weitere Details sind in der **Spezifikation** des **Standards** unter Link[e] zu finden. Allerdings werden in der Praxis, wie z.B. in Lark oft eigene **abgewandelte Notationen** wie in Definition 1.4 verwendet.*

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

[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] Also von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.
[e] `https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.

Definition 1.4: Dialekt der Erweiterten Backus-Naur-Form aus Lark

*Das **Lark Parsing Toolkit** verwendet eine **eigene Notation** für die **Erweiterte Backus-Naur-Form** (Definition 1.3), die sich teilweise in einzelnen Aspekten von der Syntax aus dem **Standard** unterscheidet und unter Link[a] dokumentiert ist.*

*Wichtige **Unterschiede** dieses Dialekts sind hierbei z.B.:*

- *für die Darstellung von **Optionaler Wiederholung** wird der aus **regulären Ausdrücken** bekannte `*`-Quantor zusammen mit optionalen **runden Klammern** `()` verwendet (z.B. `()*`).[b] Die Verwendung des `*`-Quantors kann wie in Umformung 1.0.1 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= b*\} \quad \Rightarrow \quad \{a ::= b\_tmp,\ \ b\_tmp ::= b\ \ b\_tmp \ \mid\ \varepsilon\} \tag{1.0.1}$$

- *für die Darstellung von **mindestents 1-Mal Wiederholung** wird der ebenfalls aus **regulären Ausdrücken** bekannte `+`-Operator zusammen mit optionalen **runden Klammern** `()` verwendet (z.B. `()+`). Die Verwendung des `+`-Quantors kann wie in Umformung 1.0.2 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= b+\} \quad \Rightarrow \quad \{a ::= b\ \ b\_tmp,\ \ b\_tmp ::= b\ \ b\_tmp \ \mid\ \varepsilon\} \tag{1.0.2}$$

- *für alle **ASCII-Symbole** zwischen z.B. ␣ und ∼ als **Alternative** aufgeschrieben kann auch die Abkürzung "␣".."∼" verwendet werden. Die Verwendung dieser Schreibweise kann wie in Umformung 1.0.3 zu sehen ist auch wieder zu normaler **Erweiterter Backus-Naur-Form** umgeschrieben werden.*

$$\{a ::= "ascii1"..."ascii2"\} \quad \Rightarrow \quad \{a ::= "ascii1" \ \mid\ \ldots\ \mid\ "ascii2"\} \tag{1.0.3}$$

*Um bei einer **Produktion** auszudrücken, wozu die **linke Seite** abgeleitet werden kann, wird das `::=`-Symbol verwendet. Dieses Symbol wird als **„kann abgeleitet werden zu“** gelesen.*

[a] `https://lark-parser.readthedocs.io/en/latest/grammar.html`.
[b] Der `*`-Quantor bedeutet im Gegensatz zum `+`-Quantor auch **keinmal** wiederholen.

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 **Konkreten Grammatik** einer Sprache einen **Parser** in einer vorher bestimmten Programmiersprache generieren, anstatt wie das **Lark Parsing Toolkit** bei Angabe einer **Konkreten Grammatik** direkt ein Programm in dieser Konkreten Grammatik **parsen** und einen **Ableitungsbaum** dafür generieren zu können. Lark besitzt des Weiteren eine **sehr gute Dokumentation** *Welcome to Lark's documentation! — Lark documentation*.

Neben den Konkreten Grammatiken, die aufgrund der Verwendung des **Lark Parsing Toolkit** in einem **eigenen Dialekt** der **Erweiterten Back-Naur-Form** spezifiziert sind, werden in den folgenden Unter-

[4] *ANTLR*.

kapiteln die **Abstrakten Grammatiken**, welche spezifzieren, welche Kompositionen für die **Abstrakten Syntaxbäume** der verschiedenden **Passes** erlaubt sind in einer bewusst anderen Notation aufgeschrieben. Diese Notation hat allerdings Ähnlichkeit mit dem Dialekt der **Erweiterten Backus-Naur-Form** aus dem **Lark Parsing Toolkit**.

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. Hierdurch würde die Abstrakten Grammatiken nur **unnötig verkompliziert**, wenn man die **Erweiterte Backus-Naur-Form** verwenden würde. Es gibt leider **keine** Standardnotation für **Abstrakte Grammatiken**, die sich deutlich durchgesetzt hat, daher wird für Abstrakte Grammatiken eine eigene **Abstrakte Syntaxform Notation** (Definition 1.5) verwendet. Des Weiteren trägt das Verwenden einer **unterschiedlichen Notation** für **Konkrete** und **Abstrakte Syntax** auch dazu bei, dass man beide direkter voneinander unterscheiden kann.

Definition 1.5: Abstrakte Syntaxform (ASF)

*Ist eine eigene **Metasyntax** für **Abstrakte Grammatiken**, die für diese Bachelorarbeit definiert wurde. Sie unterscheidet sich vom **Dialekt** der **Backus-Naur-Form** des **Lark Parsing Toolkit** (Definition 1.4) nur durch:*

- ***Terminalsymbole** müssen nicht von "" engeschlossen sein, da die **Knoten** in der **Abstrakten Syntax** sowieso schon **klar auszumachen** sind und von anderen Symbolen der **Metasprache** leicht zu unterscheiden sind (z.B. `Node(<non-term>, <non-term>)`).*
- *dafür müssen allerdings **Nicht-Terminalsymbole** von <>-Klammern eingeschlossen sein (z.B. `<non-term>`).*

Letztendlich geht es nur darum, dass aufgrund der Verwendung des **Lark Parsing Toolkit** die **Konkrete Grammatik** in einem eigenen Dialekt der **Erweiterter Backus-Naur-Form** angegeben sein muss und für das Implementieren der Passes die **Abstrakte Grammatik** 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 **Konkrete Grammatik** zu definieren, die den Teil der **Konkreten Syntax** beschreibt, der für die **Lexikalische Analyse** wichtig ist. Diese Konkrete Grammatik wird dann vom **Lark Parsing Toolkit** dazu verwendet ein Programm in **Konkreter Syntax** zu lexen und daraus Tokens für die **Syntaktische Analyse** zu erstellen, wie es im Unterkapitel **??** erläutert ist.

1.1.1 Konkrete Grammatik für die Lexikalische Analyse

In der Konkreten Grammatik 1.1.1 für die **Lexikalische Analyse** stehen **großgeschriebene** Nicht-Terminalsymbole entweder für einen **Tokentyp** oder einen **Teil der Beschreibung** des Aufbaus der zum **Tokentyp** gehörenden möglichen **Tokenswerte**. Zum Beispiel handelt es sich bei dem **großgeschriebenen** Nicht-Terminalsymbol `NUM` um einen **Tokentyp**, dessen zugeordnete **mögliche Tokenwerte** durch die **Produktion** `NUM ::= "0" | DIG_NO_0 DIG_WITH_0*` beschrieben werden. Diese **Produktionen** beschreiben, wie ein möglicher **Tokenwert**, in diesem Fall eine **Zahl** aufgebaut sein kann.

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

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

In Lark steht eine Zahl `.<number>`, die an ein **Nicht-Terminalsymbol** angehängt ist (z.B. `NONTERM.<number>`), dass auf der linken Seite des `::=`-Symbols einer Produktion steht für die **Priorität** der Produktion dieses **Nicht-Terminalsymbols**. Es wird immer die Produktion mit der **höchste** Priorität, also der höchsten Zahl `<number>` **zuerst** genommen.

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, damit das Wort `int` den **Tokentyp** `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 **Tokentyp** zuordnet, auch wenn das Wort eigentlich von einer anderen Produktion erkannt werden sollte. Ansonsten würden aus `int_var` die Tokens `Token('INT_DT', 'int'), Token('NAME', '_var')` generiert, anstatt dem Token`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[5] 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 Konkreten Grammatik zu lexen, **zu komplex**, um sie an dieser Stelle allgemein erklären zu können.

Der **Basic Lexer** verhält sich allerdings grundlegend so, wie es im Unterkapitel **??** erklärt wurde, nur berücksichtigt der **Basic Lexer** ebenfalls **Priortiäten**, sodass für den aktuellen Index[6] im Eingabeprogramm zuerst alle Produktionen der **höchsten Priorität** geprüft werden. Sobald eine dieser Produktionen ein **Lexeme** an dem aktuellen Index im Eingabeprogramm ableiten kann, wird hieraus direkt ein **Token** mit dem entsprechenden **Tokentyp** dieser Produktion und dem abgeleiteten **Tokenwert** erstellt. Weitere Produktionen werden **nicht** mehr geprüft. Ansonsten werden alle Produktionen der **nächstniedrigeren** Priorität geprüft usw.

[5] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/lexer.py.

[6] Ein **Lexer** bewegt sich über das Eingabeprogramm und erstellt, wenn ein **Lexeme** sich in der **Konkreten Grammatik** ableiten lässt ein **Token** und bewegt sich danach um die **Länge des Lexemes** viele Indices weiter.

COMMENT	::=	"//" /[^\n]*/ | "/*" /(. | \n)*?/ "*/"	*L_Comment*
RETI_COMMENT.2	::=	"//""␣"?"#"/[^\n]*/	
DIG_NO_0	::=	"1" | "2" | "3" | "4" | "5" | "6"	*L_Arith_Bit*
	|	"7" | "8" | "9"	
DIG_WITH_0	::=	"0" | *DIG_NO_0*	
NUM	::=	"0" | *DIG_NO_0 DIG_WITH_0*∗	
CHAR	::=	"␣".."∼"	
FILENAME	::=	*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	::=	"%"	
prec1_op	::=	*MUL_DEREF_PNTR* | *DIV* | *MOD*	
ADD	::=	"+"	
SUB_MINUS	::=	"−"	
prec2_op	::=	*ADD* | *SUB_MINUS*	
L_SHIFT	::=	"<<"	
R_SHIFT	::=	">>"	
shift_op	::=	*L_SHIFT* | *R_SHIFT*	
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: *Konkrete Grammatik 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 **Abstrakten Syntaxbaumes** 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** generierten **Tokens** 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 **Konkreter 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 **Abstrakten Syntaxbaum** in **Abstrakter Syntax** zu generieren.

1.2.1 Umsetzung von Präzedenz und Assoziativität

In diesem Unterkapitel wird eine ähnliche **Erklärweise**, wie in dem Buch Nystrom, *Parsing Expressions · Crafting Interpreters* verwendet. Die Programmiersprache L_{PicoC} hat dieselben **Präzedenzregeln** implementiert, wie die Programmiersprache L_C. Die **Präzedenzregeln** sind von der Webseite *C Operator Precedence - cppreference.com* übernommen. Die **Präzedenzregeln** der verschiedenen **Operatoren** der Programmiersprache L_{PicoC} sind in Tabelle 1.1 aufgelistet.

Präzedenzstufe	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`	Bitweise Linksshift und Rechtsshift	
6	`a<b a<=b` `a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer Gleich	
7	`a==b a!=b`	Gleichheit und Ungleichheit	
8	`a&b`	Bitweise UND	
9	`a^b`	Bitweise XOR (exclusive or)	
10	`a|b`	Bitweise ODER (inclusive or)	
11	`a&&b`	Logiches UND	
12	`a||b`	Logisches ODER	
13	`a=b`	Zuweisung	Rechts, dann links ←

Tabelle 1.1: *Präzedenzregeln von PicoC.*

Würde man diese **Operatoren** ohne Beachtung von **Präzedenzreglen** (Definiton **??**) und **Assoziativität** (Definition **??**) in eine Konkrete Grammatik verarbeiten wollen, so könnte eine Konkrete Grammatik $G = \langle N, \Sigma, P, exp \rangle$ 1.2.1 dabei rauskommen.

$prim\_exp$	::=	$name \mid NUM \mid$ "'" $CHAR$ "'" $\mid$ "(" exp ")"	*L_Arith_Bit*
	$\mid$	exp "[" exp "]" $\mid$ exp "." $name \mid name$ "(" fun_args ")"	*+L_Logic*
fun_args	::=	$[exp(",")\ exp)*]$	*+ L_Pntr*
un_op	::=	"−" $\mid$ "∼" $\mid$ "!" $\mid$ "∗" $\mid$ "&"	*+ L_Array*
un_exp	::=	$un\_op\ exp$	*+ L_Struct*
bin_op	::=	"∗" $\mid$ "/" $\mid$ "%" $\mid$ "+" $\mid$ "−" $\mid$ "&" $\mid$ "∧" $\mid$ "|"	*+ L_Fun*
	$\mid$	"<<" $\mid$ ">>" $\mid$ "<" $\mid$ "<=" $\mid$ ">" $\mid$ ">="	
	$\mid$	"!=" $\mid$ "==" $\mid$ "&&" $\mid$ "||" $\mid$ "="	
bin_exp	::=	$exp\ bin\_op\ exp$	
exp	::=	$prim\_exp \mid un\_exp \mid bin\_exp$	

Grammatik 1.2.1: *Undurchdachte Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz nicht beachtet*

Die Konkrete Grammatik 1.2.1 ist allerdings **mehrdeutig** (Definition **??**), d.h. verschiedene **Linksableitungen** in der Konkreten 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. Ab dem Moment, wo der Trick klar ist, wird das **Ableiten** mit der $\Rightarrow^*$-Relation **beschleunigt**.

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

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

Die beiden **abgeleiteten Wörter** sind **gleich**, allerdings sind die **Ableitungsbäume unterschiedlich**, wie in Abbildung 1.1 zu sehen ist. Da hier nur ein **Konzept** vermittelt werden soll, entsprechen die beiden **Ableitungsbäume** in Abbildung 1.1 nicht **1-zu-1** den Ableitungen 1.2.1 und 1.2.2, sondern sind **vereinfacht**.

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 **Klammern** setzen, um die **Präzedenz** 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. Es ist wichtig die **Präzedenzregeln** und die **Assoziativität** von Operatoren beim Erstellen der Konkreten Grammatik miteinzubeziehen, da das Ergebnis des **gleichen Ausdrucks** sich bei **unterschiedlicher Klammerung unterscheiden** kann.

Hierzu wird nun Tabelle 1.1 betrachtet. Für jede **Präzedenzstufe** in der Tabelle 1.1 wird eine eigene Produktion erstellt, wie es in Grammatik 1.2.2 dargestellt ist. Zudem braucht es eine **Produktion** `prim_exp` für die **„höchste“ Präzedenzstufe**, welche **Literale**, wie `'c'`, `5` oder `var` und geklammerte Ausdrücke wie `(3 && 14)` abdeckt.

prim_exp	::=	...	*L_Arith_Bit + L_Array*
post_exp	::=	...	*+ L_Pntr + L_Struct*
un_exp	::=	...	*+ L_Fun*
*arith_prec*1	::=	...	
*arith_prec*2	::=	...	
arith_shift	::=	...	
arith_and	::=	...	
arith_xor	::=	...	
arith_or	::=	...	
rel_exp	::=	...	*L_Logic*
eq_exp	::=	...	
logic_and	::=	...	
logic_or	::=	...	
assign_stmt	::=	...	*L_Assign*

Grammatik 1.2.2: *Erster Schritt zu einer durchdachten Konkreten Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, die Operatorpräzidenz beachtet*

Einige **Bezeichnungen** von **Nicht-Terminalsymbolen** auf der **linken Seite** des `::=`-Symbols in Grammatik 1.2.2 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_shift	a<<b a>>b
arith_and	a&b
arith_xor	a^b
arith_or	a|b
rel_exp	a<b a<=b a>b a>=b
eq_exp	a==b 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 **Präzedenzstufe** die ist, für welche die jeweilige Produktion verantwortlich ist oder deren Präzedenzstufe **höher** ist. Z.B. soll `un_op` sowohl den Ausdruck `-(3 * 14)` als auch einfach nur `(3 * 14)`[7] erkennen können, aber nicht `3 * 14` ohne Klammern, da dieser Ausdruck eine **geringe Präzedenz** hat. Des Weiteren muss bei Produktionen für Ausdrücke mit **Operatoren** unterschieden werden, ob die Operatoren **linksassoziativ** oder **rechtsassoziativ**, **unär**, **binär** usw. sind.

Im Folgenden werden **Produktionen** für alle **relevanten Fälle** von verschiedenen Kombinationen von **Präzedenzen** und **Assoziativitäten** erklärt. 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 beim Ableiten auch **terminieren** kann und es auch möglich ist, auschließlich einen Ausdruck **höherer Präzedenz** (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 in EBNF*

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 Präzedenz** (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 in EBNF*

[7] **Geklammerte Ausdrücke** werden nämlich von `prim_exp` erkannt, welches eine höhere **Präzedenzstufe** hat.

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 Präzedenzstufe 4 in Folge erkannt werden können[8] (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzedenzstufe 4 haben). Die Alternative `arith_prec1` auf der **rechten Seite** ermöglicht es, dass zwischen den Operationen der Präzedenzstufe 4 auch Operationen der Präzedenzstufe 3 auftauchen können (z.B. `3 + 1 / 4 - 1`, wobei `-` und `+` beide Präzedenzstufe 4 haben und `/` Präzedenzstufe 3). Mit der Alternative `arith_prec1` ist es möglich, dass ausschließlich ein Ausdruck **höherer Präzedenz** 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 binäre linksassoziative Produktion in EBNF*

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 binäre linksassoziative Produktion ohne Linksrekursion in EBNF*

Die von der Grammatik 1.2.6 erkannten Ausdrücke sind dieselben, wie für die Grammatik 1.2.5, allerdings ist die Grammatik 1.2.6 **flach** gehalten und ruft sich **nicht** selber auf, sondern nutzt den in der EBNF (Definition 1.3) definierten `*`-Operator, um **mehrere Operationen** der Präzedenzstufe 4 in Folge erkennen zu können (z.B. `3 + 1 - 4`, wobei `-` und `+` beide Präzedenzstufe 4 haben).

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

[a] Zu diesen Parsern zählt der **Earley Parser**, der im **PicoC-Compiler** verwendet wird glücklicherweise **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()`). Mithilfe dieser Paradigmen lässt sich die **Konkrete Grammatik** 1.2.7 definieren.

[8] Bezogen auf Tabelle 1.1.

*prec*1_*op*	::=	"∗" | "/" | "%"	*L_Misc*
*prec*2_*op*	::=	"+" | "−"	
shift_op	::=	"<<" | ">>"	
rel_op	::=	"<" | "<=" | ">" | ">="	
eq_op	::=	"==" | "!="	
fun_args	::=	[*logic_or*("," *logic_or*)∗]	
prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith_Bit*
post_exp	::=	*post_exp*"["*logic_or*"]" | *post_exp*"."*name* | *name*"("*fun_args*")"	+ *L_Array*
	|	*prim_exp*	+ *L_Pntr*
un_exp	::=	*un_op un_exp* | *post_exp*	+ *L_Struct*
*arith_prec*1	::=	*arith_prec*1 *prec*1_*op un_exp* | *un_exp*	+ *L_Fun*
*arith_prec*2	::=	*arith_prec*2 *prec*2_*op arith_prec*1 | *arith_prec*1	
arith_shift	::=	*arith_shift shift_op arith_prec*2 | *arith_prec*2	
arith_and	::=	*arith_and* "&" *arith_shift* | *arith_shift*	
arith_xor	::=	*arith_xor* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_xor* | *arith_xor*	
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*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	*L_Assign*

Grammatik 1.2.7: *Durchdachte Konkrete Grammatik der Sprache L_{PicoC} in EBNF, die Operatorpräzidenz beachtet*

1.2.2 Konkrete Grammatik für die Syntaktische Analyse

Die gesamte Konkrete Grammatik 1.2.8 ergibt sich wenn man die Konkrete Grammatik 1.2.7 um die **restliche Syntax** der Sprache L_{PicoC} erweitert, die sich nach einem **ähnlichen Prinzip** wie in Unterkapitel 1.2.1 erläutert ergibt.

Später in der Entwicklung des **PicoC-Compilers** wurde die **Konkrete Grammatik** an die **aktuellste kostenlos auffindbare Version** der echten Konkreten Grammatik der Sprache L_C, zusammengesetzt aus einer Grammatik für die **Syntaktische Analyse** *ANSI C grammar (Yacc)* und **Lexikalische Analyse** *ANSI C grammar (Lex)* angepasst[9]. Auf diese Weise konnte sicherer gewährleistet werden kann, dass der **PicoC-Compiler** sich genauso verhält, wie geläufige Compiler der Programmiersprache L_C[10].

In der Konkreten Grammatik 1.2.8 für die **Syntaktische Analyse** werden einige der **Nicht-Terminalsymbole** bzw. **Tokentypen** aus der Konkreten Grammatik 1.1.1 für die **Lexikalischen Analyse** verwendet, wie z.B. `NUM`. Es werde aber auch Produktionen, wie `name` verwendet, die mehrere **Tokentypen** unter einem Überbegriff zusammenfassen.

Terminalsymbole, wie `;` oder `&&` gehören eigentlich zur **Lexikalischen Analyse**, jedoch erlaubt das **Lark Parsing Toolkit**, um die Konkrete Grammatik leichter lesbar zu machen einige **Terminalsymbole** einfach direkt in die Konkrete Grammatik 1.2.8 für die **Syntaktische Analyse** zu schreiben. Der **Tokentyp** für diese Terminalsymbole wird in diesem Fall vom **Lark Parsing Toolkit** bestimmt, welches einige sehr häufig verwendete **Terminalsymbole**, wie z.B. `;` oder `&&` bereits einen eigenen **Tokentyp** zugewiesen hat.

[9] An der für die Programmiersprache L_{PicoC} relevanten **Syntax** hat sich allerdings über die Jahre nichts wichtiges verändert, wie die Konkreten Grammatiken für die **Syntaktische Analyse** *ANSI C grammar (Yacc) old* und **Lexikalische Analyse** *ANSI C grammar (Lex) old* aus dem Jahre 1985 zeigen.

[10] Wobei z.B. die Compiler **GCC** (*GCC, the GNU Compiler Collection - GNU Project*) und **Clang** (*clang: C++ Compiler*) zu nennen wären.

Diese **Terminalsymbole** werden aber weiterhin vom **Basic Lexer** als Teil der **Lexikalischen Analyse** generiert.

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith_Bit + 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*""("")"	*L_Arith_Bit*
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_shift	::=	*arith_shift shift_op arith_prec*2 | *arith_prec*2	
arith_and	::=	*arith_and* "&" *arith_shift* | *arith_shift*	
arith_xor	::=	*arith_xor* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_xor* | *arith_xor*	
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: *Konkrete Grammatik 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: *Konkrete Grammatik der Sprache L_{PicoC} für die Syntaktische Analyse in EBNF, Teil 2*

Anmerkung

In der Konkreten 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 Konkrete Grammatik 1.2.8 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** daraus zu generieren.

Definition 1.6: Earley Parser

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

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

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

1.2.3.1 Codebeispiel

Der **Ableitungsbaum**, der mithilfe des **Earley Parsers** und der **Tokens** der **Lexikalischen Analyse** in Code 1.2 generiert wurde, ist in Code 1.3 zu sehen. Das Beispiel aus Code 1.1 wird hier **fortgeführt**. 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 Ableitunsgbaumes

Die Ausgabe des **Ableitungsbaumes** wird komplett vom **Lark Parsing Toolkit** übernommen. Für die **Inneren Knoten** werden die **Nicht-Terminalsymbole**, welche in der Konkreten Grammatik 1.2.8 den **linken Seiten** des `::=`-Symbols[11] entsprechen hergenommen und die **Blätter** sind **Terminalsymbole**, genauso, wie es in der Definition **??** eines **Ableitungsbaumes** auch schon definiert ist. Die **Konkrete Grammatik** 1.2.8 des **PicoC-Compilers** erlaubt es 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.

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 **Datentyp** einer Variable in der Programmiersprache L_C und somit auch der Programmiersprache L_{PicoC} korrekt bestimmen zu können die **Spiralregel** *Clockwise/Spiral Rule* in der Implementeirung des **PicoC-Compilers** umgesetzt werden muss. Dies ist allerdings nicht alleinig möglich, indem man die entsprechenden **Produktionen** in der Konkreten Grammatik 1.2.8 auf eine spezielle Weise passend spezifiziert. Der **PicoC-Compiler** soll in der Lage sein den Ausdruck `int (*ar[3])[2]` als **„Feld der Mächtigkeit** 3 **von Zeigern auf Felder der Mächtigkeit** 2 **von Integern“** erkennen zu können.

Was man erhalten will, ist ein **entarteter Baum** (Definition 1.7) 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])`.

Definition 1.7: Entarteter Baum

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

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

*Der Baum entspricht also einer **verketteten Liste**.*[c]

[a] Der **Außengrad** ist die Anzahl **ausgehender Kanten**.
[b] Der **Innengrad** ist die Anzahl **eingehener Kanten**.
[c] *Bäume*.

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.

[11] *Grammar: The language of languages (BNF, EBNF, ABNF and more)*.

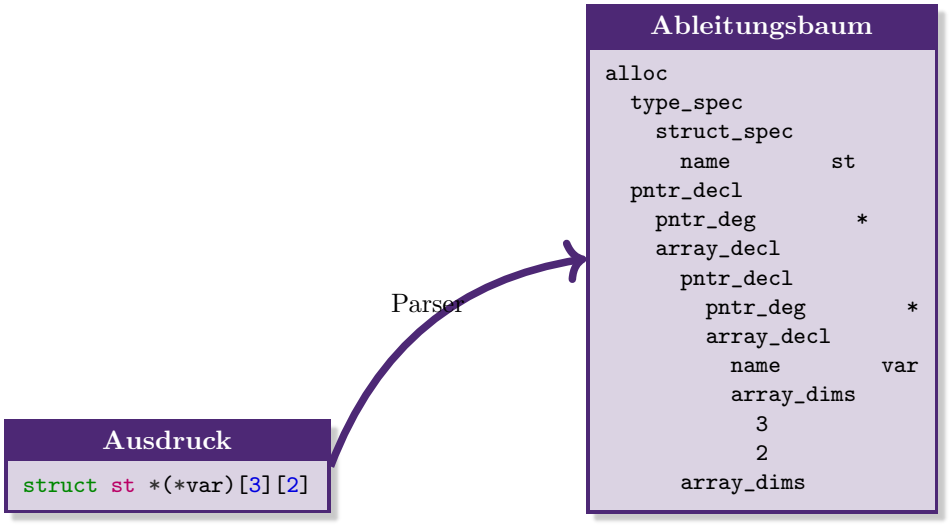

Abbildung 1.2: *Ableitungsbaum nach Parsen eines Ausdrucks.*

Dieser **Ableitungsbaum** für den Ausdruck `struct st *(*var[3][2])` hat allerdings einen Aufbau welcher durch die **Syntax** der **Zeigerdeklaratoren** `pntr_decl(num, datatype)` und **Felddeklaratoren** `array_decl(datatype, nums)` bestimmt ist, die **spiralähnlich** ist. Man würde allerdings gerne einen **entarteten Baum** erhalten, bei dem der Datentyp z.B. immer im **zweiten Attribut** weitergeht, anstatt abwechselnd im **zweiten** und **ersten**, wie beim **Zeigerdeklarator** `pntr_decl(num, datatype)` und **Felddeklarator** `array_decl(datatype, nums)`. Daher wird bei allen **Felddeklaratoren** `array_decl(datatype, nums)` immer das **erste Attribut** `datatype` mit dem **zweiten Attribut** `nums` getauscht.

Des Weiteren befindet sich in der **Mitte** der **Spirale**, die der **Ableitungsbaum** bildet der **Name der Variable** `name(var)` und nicht der **innerste Datentyp** `struct st`. Das liegt daran, dass der **Ableitungsbaum** einfach nur die **kompilerinterne Darstellung**, die durch das Parsen eines **Ausdrucks** in **Konkreter Syntax** (z.B. `struct st *(*var[3][2])`) **generiert** wird darstellt. Der Knoten für den **Bezeichner der Variable** `name(var)` wird daher mithilfe eines **Visitors** (Definition **??**) mit dem Knoten für den **innersten Datentyp** `struct st` getauscht.

Eine Änderung, die eher der **Ästhetik** dient, ist zusätzlich den Teilbaum, der den **Datentyp** darstellt mit dem Knoten `name(var)` zu **tauschen**. Das hat den Grund, dass der **Datentyp** üblicherweise **vor** dem **Bezeichner** der Variable steht: `datatype identifier`[12] und im vorherigen Schritt `name(var)` **vor** den **Datentyp** getauscht wurde. In Abbildung 1.3 ist zu sehen, wie der **Ableitungsbaum** aus Abbildung 1.2 mithilfe eines **Visitors vereinfacht** wird, sodass er die gerade erläuterten Ansprüche erfüllt.

Die Implementierung des **Visitors** aus dem **Lark Parsing Toolkit** ist unter Link[13] 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.

[12] Wie z.B. bei `int var`.
[13] https://github.com/lark-parser/lark/blob/d03f32be7f418dc21cfa45acc458e67fe0580f60/lark/visitors.py.

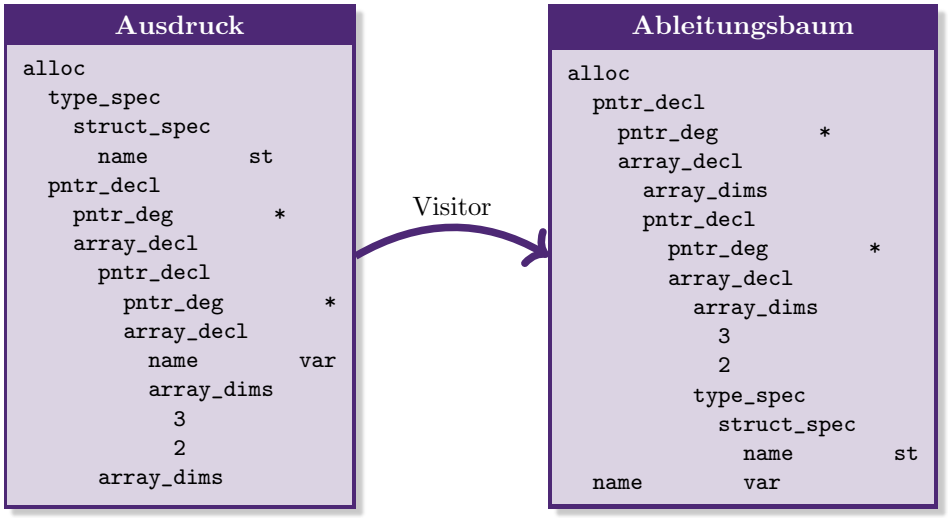

Abbildung 1.3: *Ableitungsbaum nach Vereinfachung.*

1.2.4.1 Codebeispiel

In Code 1.4 ist der **Ableitungsbaum** aus Code 1.3 nach der **Vereinfachung** mithilfe eines **Visitors** zu sehen. Das Beispiel aus Code 1.1 wird hier **fortgeführt**.

```
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 Generierung des Abstrakten Syntaxbaumes

Nachdem der **Ableitungsbaum** in Unterkapitel 1.2.4 vereinfacht wurde, ist der **vereinfachte Ableitungsbaum** in Code 1.4 nun dazu geeignet, um mit einem **Transformer** (Definition **??**) einen **Abstrakten Syntaxbaum** aus ihm zu generieren. Würde man den **vereinfachten Ableitungsbaum** des Ausdrucks `struct st *(*var[3][2])` auf die übliche Weise in einen entsprechenden **Abstrakten 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[14] 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 **rechts** in Abbildung 1.4 den Datentyp darstellt, würde man von **oben-nach-unten**[15] als **„Zeiger auf einen Zeiger auf ein Feld der Mächtigkeit 3 von Feldern der Mächtigkeit 2 von Verbunden des Typs `st`“** lesen. Man würde es also bis auf die **Dimensionen der Felder**, bei denen es freisteht, wie man sie liest genau **anders herum** lesen, 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 Zeigern auf einen Zeiger auf einen Verbund vom Typ `st`“** ist.

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

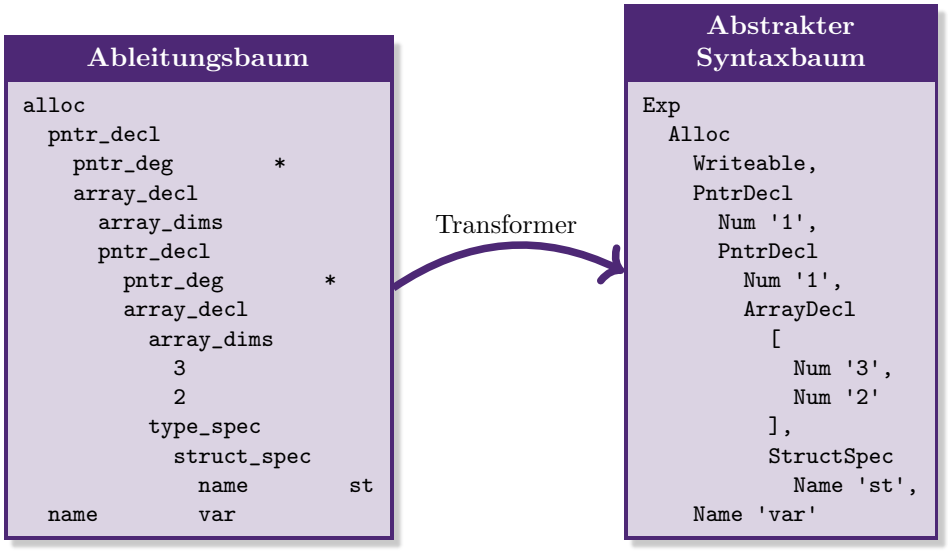

Abbildung 1.4: *Generierung eines Abstrakten Syntaxbaumes ohne Umdrehen.*

Der **Abstrakte Syntaxbaum** rechts in Abbildung 1.4 ist für die Weiterverarbeitung also ungeeignet, denn für die **Adressberechnung** bei einer Aneinanderreihung von Zugriffen auf **Zeigerelemente**, **Feldelemente** oder **Verbundsattribute**[16] will man den Datentyp in **umgekehrter Reihenfolge**. Aus diesem Grund muss der **Transformer** bei der Konstruktion des **Abstrakten Syntaxbaumes** zusätzlich dafür sorgen, dass jeder **Teilbaum**, der für einen **vollständigen Datentyp** steht **umgedreht** wird. Auf diese Weise kommt ein **Abstrakter Syntaxbaum** mit **richtig rum gedrehtem Datentyp**, wie **rechts** in Abbildung 1.5 zustande, der für die Weiterverarbeitung geeignet ist.

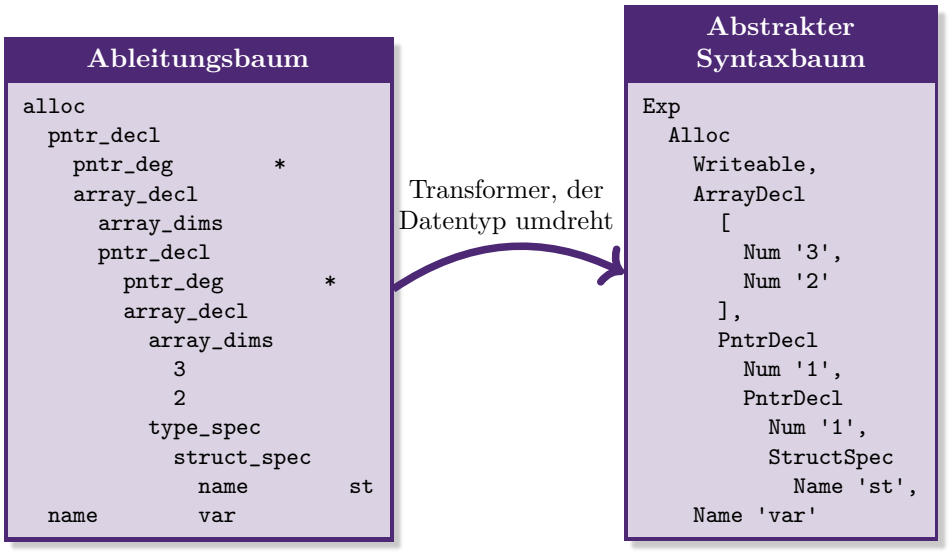

Abbildung 1.5: *Generierung eines Abstrakten Syntaxbaumes mit Umdrehen.*

Die Weiterverarbeitung des **Abstrakten Syntaxbaumes** geschieht mithilfe von **Passes**, welche im Unterkapitel 1.3 genauer beschrieben werden. Da die Knoten des **Abstrakten Syntaxbaumes** anders als beim

[16] Welche in Unterkapitel **??** genauer erläutert wird

Ableitungsbaum nicht die gleichen Bezeichnungen haben, wie **Nicht-Terminalsymbole** der Konkreten Grammatik, 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**. Diese Knoten kommen später im Unterkapitel 1.3 in den unterschiedlichen von den **Passes** umgeformten **Abstrakten Syntaxbäumen** vor.

Des Weiteren gibt die Abstrakte Grammatik 1.2.10 in Unterkapitel 1.2.5.4 Aufschluss darüber welche **Kompositionen von PicoC-Knoten** neben den bereits in Tabelle 1.8 definierten Kompositionen mit Bedeutung insgesamt überhaupt **möglich** sind.

1.2.5.1 PicoC-Knoten

Bei den **PicoC-Knoten** handelt es sich um Knoten, die wenn man die Programmiersprache L_{PicoC} auf das herunterbricht, was wirklich entscheidend ist ein **abstraktes Konstrukt** darstellen, welches z.B. ein **Container** für andere Knoten sein kann oder einen der ursprünglichen **Token darstellt**. Diese **Abstrakten Konstrukte** sollen allerdings immer noch etwas aus der Programmiersprache 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[17]. Alle **PicoC-Knoten**, die in den von den verschiedenen Passes generierten **Abstrakten Syntaxbäumen** vorkommen sind in Tabelle 1.3 mit einem **Beschreibungstext** dokumentiert.

[17] 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(str)`	Repräsentiert einen **Bezeichner** (z.B. `my_fun, my_var usw.`). Das Attribut `str` ist eine Zeichenkette, welche beliebige **Groß-** und **Kleinbuchstaben** (`a - z, A - Z`), **Zahlen** (`0 - 9`) und den **Unterstrich** (`_`) enthalten kann. An **erster Stelle** darf allerdings **keine Zahl** stehen.
`Num(str)`	Eine **Zahl** (z.B. `42, -3` usw.). Hierbei ist das Attribut `str` eine Zeichenkette, welche **beliebige Ziffern** (`0 - 9`) enthalten kann. Es darf nur **nicht** eine `0` am Anfang stehen und danach **weitere Ziffern** folgen. Der **Wert**, welcher durch die Zeichenkette dargestellt wird, darf **nicht größer** als $2^{32} - 1$ sein.
`Char(str)`	Ein **Zeichen** (`'c', '*'` usw.). Das Attribut `str` ist ein Zeichnen der **ASCII-Zeichenkodierung**.
`Minus(), Not(), DerefOp(), RefOp(), LogicNot()`	Die **unären Operatoren** `un_op`: `-a, ~a, *a, &a !a`.
`Add(), Sub(), Mul(), Div(), Mod(), LShift(), RShift(), Xor(), And(), Or(), LogicAnd(), LogicOr()`	Die **binären Operatoren** `bin_op`: `a + b, a - b, a * b, a / b, a % b, a << b, a >> b, a ^ b, a & b, a | b, a && b, a || 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 **Basisdatentypen**: `int`, `char`, `void`. Um eine **intuitive Bezeichnung** zu haben werden sie in der **Abstrakten Grammatik** einfach nur als **Datentypen** `datatype` eingeordnet werden.
`Placeholder()`	**Platzhalter** für einen Knoten, der diesen später **ersetzt**.
`BinOp(exp, bin_op, exp)`	Container für eine **binäre Operation**: `<exp1> <bin_op> <exp2>`.
`UnOp(un_op, exp)`	Container für eine **unäre Operation**: `<un_op> <exp>`.
`Atom(exp, rel, exp)`	Container für eine **binäre Relation**: `<exp1> <rel> <exp2>`
`ToBool(exp)`	Container für einen **Arithmetischen oder Bitweise Ausdruck**, wie z.B. `1 + 3` oder einfach nur `3`. Aufgrund des **Kontext** in dem sich dieser Ausdruck befindet, wird bei einem Ergebnis $x > 0$ auf 1 abgebildet und bei $x = 0$ auf 0.
`Alloc(type_qual, datatype, name, local_var_or_param)`	Container für eine **Allokation** `<type_qual> <datatype> <name>` mit den Attributen `type_qual`, `datatype` und `name`, die alle Informationen für einen Eintrag in der **Symboltabelle** enthalten. Zudem besitzt er ein versteckte Attribut `local_var_or_param`, dass die Information trägt, ob es sich bei der **Variable** um eine **Lokale Variable** oder einen **Parameter** handelt.
`Assign(exp1, exp2)`	Container für eine **Zuweisung** `exp1 = exp2`, wobei `exp1` z.B. ein `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` sein kann und `exp2` ein beliebiger **Logischer Ausdruck** sein kann.

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** geschrieben werden soll. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` einen **Datentyp** transportiert, der für den **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Stack(num)`	Holt sich z.B. für eine Berechnung einen zuvor auf den **Stack** geschriebenen Wert vom **Stack**, der `num` Speicherzellen relativ zum `SP`-Register steht.
`Stackframe(num)`	Holt sich den **Wert** einer **Variable**, eines **Verbundsattributs**, eines **Feldelements** etc., der `num` + 2 Speicherzellen relativ zum `BAF`-Register steht.
`Global(num)`	Holt sich den **Wert** einer **Variable**, eines **Verbundsattributs**, eines **Feldelements** etc., der `num` Speicherzellen relativ zum `DS`-Register steht.
`StackMalloc(num)`	Steht für das **Allokieren** von `num` Speicherzellen auf dem **Stack**.
`PntrDecl(num, datatype)`	Container, der für den **Zeigerdatentyp** steht: `<prim_dt> *<var>`. Hierbei gibt das **Attribut** `num` die **Anzahl zusammengefasster Zeiger** an und `datatype` ist der **Datentyp**, auf den der letzte dieser **Zeiger** zeigt.
`Ref(exp, datatype, error_data)`	Steht für die Anwendung des **Referenz-Operators** `&<var>`, der die **Adresse** einer **Location** (Definition **??**) auf den Stack schreiben soll, die über `exp` bestimmt ist. Zudem besitzt er 2 versteckte Attribute, wobei `datatype` einen **Datentyp** transportiert, der für den **RETI Blocks Pass** wichtig ist und `error_data` für **Fehlermeldungen** wichtig ist.
`Deref(exp1, exp2)`	Container für den **Indexzugriff** auf einen **Feld-** oder **Zeigerdatentyp**: `<var>[<i>]`. Hierbei ist `exp1` ein angehängtes weiteres `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` und `exp2` ist der **Index** auf den zugegriffen werden soll.
`ArrayDecl(nums, datatype)`	Container, der für den **Felddatentyp** steht: `<prim_dt> <var>[<i>]`. Hierbei ist das **Attribut** `nums` eine Liste von `Num('x')`, welche die **Dimensionen** des Felds angibt und `datatype` ist ein **Unterdatentyp** (Definition **??**).
`Array(exps, datatype)`	Container für den **Initialisierer** eines **Feldes**, z.B. `{{1, 2}, {3, 4}}`, dessen Attribut `exps` weitere Initialisierer für ein **Feld**, weitere Initialisierer für einen **Verbund** oder **Logische Ausdrücke** beinhalten kann. Des Weiteren besitzt es 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 **Feld-** oder **Zeigerdatentyp**: `<var>[<i>]`. Hierbei ist `exp1` ein angehängtes weiteres `Subscr(exp1, exp2)`, `Deref(exp1, exp2)`, `Attr(exp, name)` oder `Name('var')` und `exp2` ist der **Index** auf den zugegriffen werden soll.
`StructSpec(name)`	Container für die **Spezifikation** eines **Verbundstyps**: `struct <name>`. Hierbei legt das **Attribut** `name` fest, welchen selbst definierten **Verbundstyp** dieser Knoten spezifiziert.
`Attr(exp, name)`	Container für den **Zugriff** auf ein **Verbundsattribut**: `<var>.<attr>`. Hierbei kann `exp` eine angehängte weitere `Subscr(exp1, exp2)`, `Deref(exp1, exp2)` oder `Attr(exp, name)` Operation oder ein `Name('var')` sein. Das Attribut `name` ist das **Verbundsattribut** auf das zugegriffen werden soll.

Tabelle 1.4: *PicoC-Knoten Teil 2.*

PiocC-Knoten	Beschreibung
`Struct(assigns, datatype)`	Container für den **Initialisierer** eines **Verbundes**, z.B {.<attr1>={1,2},.<attr2>={3,4}}, dessen Attribut `assigns` eine Liste von `Assign(exp1, exp2)` ist, mit der Zuordnung eines **Attributezeichners** zu einem **Initialisierer** für ein **Feld** oder einen **Initialisierer** für einen **Verbund** oder 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 Verbundstyps**, z.B. `struct <var> {<datatype> <attr1>; <datatype> <attr2>;};`. Hierbei ist `name` der **Bezeichner** des **Verbundstyps** und `allocs` eine Liste von Bezeichnern der **Verbundsattribute** mit dazugehörigem **Datentyp**, wofür sich der Knoten `Alloc(type_qual, datatype, name)` sehr gut als Container eignet.
`If(exp, stmts)`	Container für eine **If-Anweisung** `if(<exp>) { <stmts> }` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`IfElse(exp, stmts1, stmts2)`	Container für eine **If-Else Anweisung** `if(<exp>) { <stmts1> } else { <stmts2> }` inklusive **Bedingung** `exp` und 2 **Branches** `stmts1` und `stmts2`, die zwei Alternativen Darstellen in denen jeweils **Listen von Anweisungen** stehen.
`While(exp, stmts)`	Container für eine **While-Anweisung** `while(<exp>) { <stmts> }` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`DoWhile(exp, stmts)`	Container für eine **Do-While-Anweisung** `do { <stmts> } while(<exp>);` inklusive **Bedingung** `exp` und einem **Branch** `stmts`, indem eine **Liste von Anweisungen** steht.
`Call(name, exps)`	Container für einen **Funktionsaufruf** `fun_name(exps)`, wobei `name` der **Bezeichner** der Funktion `fun_name` ist, die aufgerufen werden soll und `exps` eine **Liste von Argumenten** ist, die an diese Funktion übergeben werden soll.
`Return(exp)`	Container für eine **Return-Anweisung** `return <exp>`, wobei das **Attribut** `exp` einen **Logischen Ausdruck** darstellt, dessen Ergebnis von der **Return-Anweisung** zurückgegeben wird.
`FunDecl(datatype, name, allocs)`	Container für eine **Funktionsdeklaration** `<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. Der Knoten `Alloc(type_spec, datatype, name)` dient dabei als Container für die **Parameter** in `allocs`.

Tabelle 1.5: *PicoC-Knoten Teil 3.*

PiocC-Knoten	Beschreibung
`FunDef(datatype, name, allocs, stmts_blocks)`	Container für eine **Funktionsdefinition** `<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 und `stmts_blocks` eine Liste von **Statemetns** bzw. **Blöcken** ist, welche diese Funktion beinhaltet. Der Knoten `Alloc(type_spec, datatype, name)` dient dabei als Container für die **Parameter** in `allocs`.
`NewStackframe(fun_name, goto_after_call)`	**Erstellt** einen neuen **Stackframe** und speichert den Wert des `BAF`-Registers der **aufrufenden Funktion** und die **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 enthält das Attribut `goto_after_call` ein `GoTo(Name('addr@next_instr'))`, welches später durch die **Adresse** des Befehls, der direkt auf den **Sprungbefehl** 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`-Registers **vor** der Wiederherstellung.
`File(name, decls_defs_blocks)`	Container der eine **Datei** repräsentiert, wobei `name` der **Dateiname** der Datei ist 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 **Anweisungen**, wobei das Attribut `name` der Bezeichner des **Labels** (Definition **??**) des Blocks ist und `stmts_instrs` eine **Liste von Anweisungen** oder **Befehlen** ist. Zudem besitzt er noch 4 versteckte Attribute, wobei `instrs_before` die Zahl der **Befehle vor** diesem **Block** zählt, `num_instrs` die Zahl der **Befehle 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 einen Sprung zu einem anderen **Block** durch Angabe des **Bezeichners** des **Labels** des Blocks, wobei das Attribut `name` der **Bezeichner** des Labels des Blocks ist, zu dem Gesprungen werden soll.
`SingleLineComment(prefix, content)`	Container für einen **Kommentar** (`//`, `/* <comment> */`), den der Compiler selbst während des **Kompiliervorgangs** erstellt, damit die **Zwischenschritte** der Kompilierung und auch der **finale RETI-Code** bei Betrachtung ausgegebener **Abstrakter Syntaxbäume** der verschiedenen Passes **leichter verständlich** sind.
`RETIComment(str)`	Container für einen **Kommentar** im Code der Form: `// # comment`, der im **RETI-Intepreter** später sichtbar ist und zur Orientierung genutzt werden kann. So ein **Kommentar** wäre allerdings in einer tatsächlichen Implementierung einer RETI-CPU **nicht umsetzbar** und eine Umsetzung wäre auch nicht sinnvoll. Der **Kommentar** ist im Attribut `str`, welches jeder Knoten besitzt gespeichert.

Tabelle 1.6: *PicoC-Knoten Teil 4.*

Anmerkung

Die ausgegrauten Attribute der PicoC-Knoten sind versteckte Attribute, die **nicht** direkt bei der Erstellung der **PicoC-Knoten** mit einem Wert **initialisiert** werden. Diese Attribute bekommen im **Verlauf der Kompilierung** beim Durchlaufen der verschiedenen Passes etwas zugewiesen, um im weiteren Kompiliervorgang **Informationen** zu transportieren. Das sind **Informationen**, die später im Kompiliervorgang **nicht** mehr so **leicht zugänglich** sind, wie zu dem Zeitpunkt, zu dem sie zugewiesen werden.

Jeder **Knoten** hat darüberhinaus auch noch 2 **Attribute** `value` und `position`. Das **Attribut** `value` entspricht bei einem **Blatt** dem **Tokenwert** des Tokens welches es ersetzt. Bei **Inneren Knoten** ist das Attribut `value` hingegen unbesetzt. Das **Attribut** `position` wird für **Fehlermeldungen** gebraucht.

1.2.5.2 RETI-Knoten

Bei den **RETI-Knoten** handelt es sich um Knoten, die irgendeinen einen **Bestandteil eines Befehls** 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 **Abstrakten Syntaxbäumen** vorkommen sind in Tabelle 1.2.5.1 mit einem **Beschreibungstext** dokumentiert.

RETI-Knoten	Beschreibung
`Program(name, instrs)`	Container der ein Programm repräsentiert: `<name> <instrs>`. Hierbei ist `name` der **Name** des **Programms**, welches ausgeführt werden soll und `instrs` ist eine **Liste von Befehlen**.
`Instr(op, args)`	Container für einen **Befehl**: `<op> <args>`. Hierbei ist `op` eine **Operation** und `args` eine **Liste von Argumenten** für diese Operation.
`Jump(rel, im_goto)`	Container für einen **Sprungbefehl**: `JUMP<rel> <im>`. Hierbei ist `rel` eine **Relation** und `im_goto` ist ein **Immediate Value** `Im(str)` für die **Anzahl an Speicherzellen**, um die relativ zum **Sprungbefehl** gesprungen werden soll. In einigen Fällen ist `im` ein `GoTo(Name('block.xyz'))`, das später im **RETI-Patch Pass** durch einen passenden **Immediate Value** ersetzt wird.
`Int(num)`	Container für den **Aufruf einer Interrupt-Service-Routine**: `INT <im>`. Hierbei ist `num` die **Interrruptvektornummer** (IVN) für die passende Adresse in der **Interruptvektortabelle**, in der die Adresse der **Interrupt-Service-Routine** (ISR) steht, die man aufrufen will.
`Call(name, reg)`	Container für einen **Prozeduraufruf**: `CALL <name> <reg>`. Hierbei ist `name` der **Bezeichner** der Prozedur, die aufgerufen werden soll und `reg` ist ein **Register**, das als **Argument** an die Prozedur dient. Diese **Operation** ist in der Betriebssysteme Vorlesung[a] **nicht deklariert**, sondern wurde hinzugefügt, um unkompliziert ein `CALL PRINT ACC` oder `CALL INPUT ACC` im RETI-Interpreter simulieren zu können.
`Name(str)`	Bezeichner für eine **Prozedur**, z.B. `PRINT`, `INPUT` oder den **Programnamen**, z.B. `PROGRAMNAME`. Hierbei ist `str` eine **Zeichenkette** für welche das gleich gilt, wie für das `str` Attribut des **PicoC-Knoten** `Name(str)`. Dieses **Argument** ist in der Betriebssysteme Vorlesung[a] **nicht deklariert**, sondern wurde hinzugefügt, um Bezeichner, wie `PRINT`, `INPUT` oder `PROGRAMNAME` schreiben zu können.
`Reg(reg)`	Container für ein **Register**. Hierbei ist `reg` das **Register** auf welches zugegriffen werden soll.
`Im(str)`	Ein **Immediate Value** (z.B. `42`, `-3` usw.). Hierbei ist das Attribut `str` eine Zeichenkette, welche **beliebige Ziffern** (`0 - 9`) enthalten kann. Es darf nur **nicht** eine `0` am Anfang stehen und danach **weitere Ziffern** folgen. Der **Wert**, welcher durch die Zeichenkette dargestellt wird, darf **nicht kleiner** als -2^{21} oder **größer** als $2^{21} - 1$ sein.
`Add(), Sub(), Mult(), Div(), Mod(), Xor(), Or(), And()`	**Compute-Memory** oder **Compute-Register** Operationen: `ADD, SUB, MULT, DIV, OPLUS, OR, AND`.
`Addi(), Subi(), Multi(), Divi(), Modi(), Xori(), 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** `rel`: `<, <=, >, >=, ==, !=, _NOP`.
`Rti()`	**R**eturn-**F**rom-**I**nterrupt Operation: `RTI`.
`Pc(), In1(), In2(), Acc(), Sp(), Baf(), Cs(), Ds()`	**Register** `reg`: `PC, IN1, IN2, ACC, SP, BAF, CS, DS`.

[a] Scholl, „Betriebssysteme“

Tabelle 1.7: *RETI-Knoten.*

1.2.5.3 Kompositionen von Knoten mit besonderer Bedeutung

In Tabelle 1.8 sind jegliche **Kompositionen** von **PicoC-Knoten** und **RETI-Knoten** aufgelistet, die eine **besondere Bedeutung** haben.

Komposition	Beschreibung
`Ref(Global(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum `DS`-Register steht auf den **Stack**.
`Ref(Stackframe(Num('addr')))`	Speichert **Adresse** der Speicherzelle, die $\texttt{Num}('\texttt{addr}') + 2$ Speicherzellen **relativ** zum `BAF`-Register steht auf den **Stack**.
`Ref(Subscr(Stack(Num('addr1')), Stack(Num('addr2'))), datatype)`	Berechnet die nächste **Adresse** aus der **Adresse**, die an Speicherzelle `Stack(Num('addr1'))` steht und dem **Index**, der an Speicherzelle `Stack(Num('addr2'))` steht und speichert diese auf den **Stack**. Die Berechnung ist abhängig davon, ob der **Datentyp** im versteckten Attribut `datatype` ein `ArrayDecl(datatype)` oder `PntrDecl(datatype)` ist.
`Ref(Attr(Stack(Num('addr1')), Name('attr')), datatype)`	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 `Name('st')` des **Verbundstyps** in `StructSpec(Name('st'))` notwendig mit dem diese Berechnung durchgeführt wird. Dieser **Verbundstyp** ist im versteckten Attribut `datatype` zu finden. Dabei muss dieser **Datentyp** im versteckten Attribut `datatype` ein `StructSpec(name)` sein, da diese Berechnung nur bei einem **Verbund** durchgeführt werden kann.
`Assign(Stack(Num('size'))), Global(Num('addr')))`	Schreibt `Num('size')` viele **Speicherzellen**, die `Num('addr')` viele Speicherzellen **relativ** zum `DS`-Register stehen, versetzt genauso auf den **Stack**.
`Assign(Stack(Num('size')), Stackframe(Num('addr')))`	Schreibt `Num('size')` viele **Speicherzellen**, die $\texttt{Num}('\texttt{addr}') + 2$ viele Speicherzellen **relativ** zum `BAF`-Register stehen, versetzt genauso auf den **Stack**.
`Assign(Stack(Num('addr1')), Stack(Num('addr2')))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr2')` Speicherzellen **relativ** zum `SP`-Register steht an der **Adresse** in der Speicherzelle, die `Num('addr1')` Speicherzellen **relativ** zum `SP`-Register steht.
`Assign(Global(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt zu den **Globalen Statischen Daten** ab einer `Num('addr')` viele Speicherzellen **relativ** zum `DS`-Register liegenden Adresse.
`Assign(Stackframe(Num('addr')), Stack(Num('size')))`	Schreibt `Num('size')` viele Speicherzellen, die auf dem **Stack** stehen, versetzt in den **Stackframe** der momentan aktiven Funktion ab einer `Num('addr')` viele Speicherzellen **relativ** zum `BAF`-Register liegenden Adresse.
`Exp(Global(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die `Num('addr')` Speicherzellen **relativ** zum `DS`-Register steht auf den **Stack**.
`Exp(Stackframe(Num('addr'))`	Speichert **Inhalt** der Speicherzelle, die $\texttt{Num}('\texttt{addr}') + 2$ Speicherzellen **relativ** zum `BAF`-Register steht auf den **Stack**.
`Exp(Stack(Num('addr')))`	Speichert **Inhalt** der Speicherzelle an der **Adresse**, die in der Speicherzelle gespeichert ist, die `Num('addr')` viele Speicherzellen **relativ** zum `SP`-Register liegt auf den **Stack**.
`Exp(Reg(reg))`	Schreibt den aktuellen Wert des **Registers** `reg` auf den **Stack**.
`Instr(Loadi(), [Reg(reg), GoTo(Name('addr@next_instr'))])`	Lädt in das `reg`-Register die **Adresse** des Befehls, der 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 Kompositionen **ausgelassen**, bei denen einfach nur ein `exp`i, i ::= ”1” | ”2” | ε durch `Stack(Num('`x`'))`, $x \in \mathbb{N}$ ersetzt wurde[a].

Zudem sind auch jegliche Kombinationen **ausgelassen**, bei denen nur ein **Ausdruck** an ein `Exp(exp)` bzw. `Ref(exp)` drangehängt wurde[b].

[a] Wie z.B. bei `BinOp(Stack(Num('2')), Add(), Stack(Num('1')))`.
[b] Wie z.B. bei `Exp(Num('42'))`.

1.2.5.4 Abstrakte Grammatik

Die **Abstrakte Grammatik** der Sprache L_{PicoC} ist in Grammatik 1.2.10 dargestellt.

stmt	::=	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
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	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	|	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
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	::=	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	|	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle stmt\rangle*)$	
file	::=	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 1.2.10: *Abstrakte Grammatik der Sprache L_{PiocC} in ASF*

1.2.5.5 Codebeispiel

In Code 1.5 ist der **Abstrakte Syntaxbaum** zu sehen, der aus dem **vereinfachten Ableitungsbaum** aus Code 1.4 mithilfe eines **Transformers** (Definition **??**) generiert wurde. Das Beispiel aus Code 1.1 wird hier **fortgeführt**.

```
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 einem vereinfachtem Ableitungsbaum generierter Abstrakter Syntaxbaum.*

1.2.5.6 Ausgabe des Abstrakten Syntaxbaumes

Ein **Teilbaum** eines **Abstrakten Syntaxbaumes** kann entweder in der **Konkreten Syntax** der Sprache, für dessen Kompilierung er generiert wurde oder in der **Abstrakten Syntax**, die beschreibt, wie der Abstrakte Syntaxbaum selbst aufgebaut sein darf ausgegeben werden.

Das Ausgeben eines **Abstrakten Syntaxbaumes** wird im **PicoC-Compiler** über die **Magische Methode** `__repr__()`[18] der Programmiersprache L_{Python} umgesetzt. Sobald ein **PicoC-Knoten** oder **RETI-Knoten** ausgegeben werden soll, gibt seine Magische Methode `__repr__()` eine nach der **Abstrakten** oder **Konkreten**

[18] Spezielle Methode, die immer aufgerufen wird, wenn das **Objekt**, dass in Besitz dieser Methode ist als **Zeichenkette** mittels `print()` oder zur **Repräsentation** ausgegeben werden soll.

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. Der gesamte **Abstrakte Syntaxbaum** wird durchlaufen und die Magischen `__repr__()`-Methoden der verschiedenen Knoten aufgerufen, die immer jeweils die `__repr__()`-Methoden ihrer Kinder aufrufen und die zurückgegebenen **Textrepräsentationen** passend **zusammenfügen** und selbst **zurückgeben**.

Beim **PicoC-Compiler** sind **Abstrakte** und **Konkrete Syntax** miteinander gemischt. Für **PicoC-Knoten** wird die **Abstrakte Syntax** verwendet, da Passes schließlich auf **Abstrakten Syntaxbäumen** operieren. Bei **RETI-Knoten** wird die **Konkrete Syntax** verwendet, da **Maschinenbefehle** in **Konkreter Syntax** schließlich das **Endprodukt** des Kompiliervorgangs sein sollen.

Da die **Konkrette Syntax** von **RETI-Knoten** sehr simpel ist, macht es kaum einen Unterschied in der Erkennbarkeit, bis auf fehlende gescheifte Klammern `()` usw., ob man die **RETI-Knoten** in **Abstrakter** oder **Konkreter Syntax** schreibt. Daher werden die **RETI-Knoten** einfach immer direkt in **Konkreter Syntax** ausgegeben. Auf diese Weise muss nicht beim **letzten Pass** daran gedacht werden am Ende die **Konkrete**, statt der **Abstrakten Syntax** für die **RETI-Knoten** auszugeben.

Die Ausgabe des **Abstrakten Syntaxbaums** ist bewusst so gewählt, dass sie sich optisch von der des **Ableitungsbaums** unterscheidet, indem die Bezeichner der **Knoten** in **UpperCamelCase**[19] geschrieben sind. Das steht im Gegensatz zum **Ableitungsbaum**, dessen **Innere Knoten** im **snake_case** geschrieben sind, da sie die **Nicht-Terminalsymbole** auf den **linken Seiten** des `::=`-Symbols in der **Konkreten Grammatik** 1.2.8 darstellen, welche in **snake_case** geschrieben sind.

1.3 Code Generierung

Nach der Generierung eines **Abstrakten Syntaxbaums** als Ergebnis der **Lexikalischen** und **Syntaktischen Analyse** in Unterkapitel 1.2.5, wird in diesem Kapitel auf Basis der verschiedenen **Kompositionen** von Knoten im **Abstrakten Syntaxbaum** das gewünschte Endprodukt des **PicoC-Compilers**, der **RETI-Code** generiert.

Man steht nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** der Sprache L_{PicoC}, der durch die **Abstrakte Grammatik** 1.2.10 spezifiziert ist in einen **semantisch** gleichen **Abstrakten Syntaxbaum** der Sprache L_{RETI} umzuformen, der durch die **Abstrakte Grammatik** 1.3.6 spezifiziert ist. In **T-Diagramm-Notation** (siehe Unterkapitel **??**) lässt sich das darstellen, wie in Abbildung 1.6.

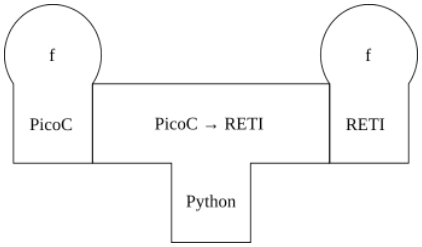

Abbildung 1.6: *Kompiliervorgang Kurzform.*

Das gerade angesprochene Problem lässt sich, wie in Unterkapitel **??** bereits beschrieben **vereinfachen**, indem man das Problem in mehrere **Passes** (Definition **??**) aufteilt, die jeweils ein **überschaubares Teilproblem** lösen. Man nähert sich Schrittweise immer mehr der **Syntax** der Sprache L_{RETI} an.

[19] *Naming convention (programming).*

In Abbildung 1.7 ist das **T-Diagramm** aus Abbildung 1.6 detailierter dargestellt. Das T-Diagramm gibt einen 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 **Abstrakten Syntaxbaum** des vorherigen Passes oder der Syntaktischen Analyse, um einen eigenen **Abstrakten Syntaxbaum** in seiner eigenen **Sprache** zu generieren.

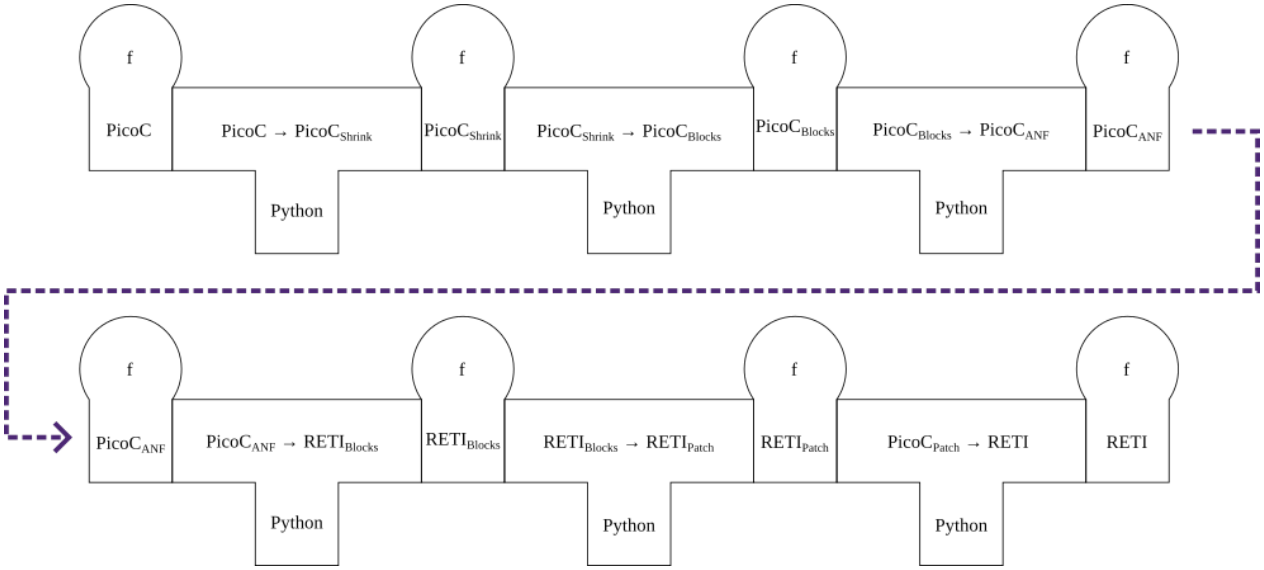

Abbildung 1.7: *Architektur mit allen Passes ausgeschrieben.*

Im Unterkapitel 1.3.1 werden die unterschiedlichen **Passes** des PicoC-Compilers erklärt. In den darauffolgenden Unterkapiteln **??**, **??**, **??** und **??** zu **Zeigern**, **Feldern**, **Verbunden** und **Funktionen** werden einzelne **Aspekte**, die Thema dieser **Bachelorarbeit** sind **genauer betrachtet** und erklärt, die im Unterkapitel 1.3.1 nicht 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.

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[20] der **Bachelorarbeit** wichtig.

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

1.3.1.1 PicoC-Shrink Pass

Die 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 **Zeigerarithmetik** `*(pntr + i)` in der Untermenge der Sprache L_C, welche die Sprache L_{PicoC} darstellt die **gleiche Semantik** hat, wie der **Operator** für den **Zugriff** auf den **Index**

[20] In kurz: **Zeiger**, **Felder**, **Verbunde** und **Funktionen**.

eines **Feldes** `ar[i]`[21].

Daher wandelt der **PicoC-Shrink Pass** alle Verwendungen des **Knoten** `Deref(exp, i)` im jeweiligen **Abstrakten 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 den **Zugriff auf einen Feldindex** `var[i]` übernehmen.

1.3.1.1.1 Abstrakte Grammatik

Die **Abstrakte Grammatik** 1.3.1 der Sprache L_{PicoC_Shrink} ist fast **identisch** mit der **Abstrakten Grammatik** 1.2.10 der Sprache L_{PiocC}, nach welcher der **erste** Abstrakte Syntaxbaum in der **Syntaktischen Analyse** generiert wurde. Der einzige **Unterschied** liegt darin, dass es den Knoten `Deref(exp1, exp2)` in der Abstrakten Grammatik 1.3.1 **nicht** mehr gibt. Das liegt daran, dass dieser Pass alle **Vorkommnisse** des Knoten `Deref(exp1, exp2)` durch den Knoten `Subscr(exp1, exp2)` auswechselt.

[21] Wobei `*pntr` und `pntr[0]` einander entsprechen.

stmt	::=	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	$\mid$	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	$\mid$	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	$\mid$	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	$\mid$	$Call(Name('print'), \langle exp\rangle)$	
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	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Deref(\langle exp\rangle, \langle exp\rangle) \mid Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	$\mid$	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
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	::=	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	$\mid$	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$	
		$\quad Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle stmt\rangle*)$	
file	::=	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 1.3.1: *Abstrakte Grammatik der Sprache L_{PiocC_Shrink} in ASF*

Anmerkung

Alles ausgegraute bedeutet, es hat sich im Vergleich zur letzten Abstrakten Grammatik **nichts** geändert. Alles rot markierte bedeutet, es wurde **entfernt** oder **abgeändert**. Alle normal in `schwarz` geschriebenen Knoten sind **neu** hinzugefügt. Das gilt genauso für **alle** folgenden **Grammatiken**.

1.3.1.1.2 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“, welches diesen Algorithmus allerdings **rekursiv** implementiert.

Die **rekursive** Implementierung des Algorithmus wäre allerdings **kein** gutes **Anschauungsbeispiel**, das 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`-Anweisungen, wären mit der **rekursiven** Implementierung aus der Vorlesung **nicht** veranschaulicht gewesen. Daher wurde die **rekursive Implementierung** aus der Vorlesung zu einem **iterativen** Algorithmus 1.6 umgeschrieben, um unter anderem auch `if-` und `while`-Statements zu enthalten.

Beide Varianten des **Algorithmus** wurden zum **Testen** des PicoC-Compilers verwendet und sind als Tests im Ordner `/tests` unter Link[22], 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 im jeweiligen Unterkapitel beschrieben wird und werden nicht im Detail erläutert. Viele Details der Passes werden später in den Unterkapiteln **??**, **??**, **??** und **??** zu **Zeigern**, **Feldern**, **Verbunden** und **Funktionen** mit eigenen **Codebeispielen** genauer erklärt und alle sonstigen Details sind dem **Bachelorprojekt** zuzuordnen, dessen Aspekte in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit **nicht** im Detail erläutert werden.

```
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 **Abstrakten Syntaxbaum**, der in der **Syntaktischen Analyse** generiert wurde.

```
1  File
2    Name './example_faculty_it.ast',
3    [
4      FunDef
5        IntType 'int',
```

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

```
6       Name 'faculty',
7       [
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 **Abstrakten Syntaxbaum** in Code 1.7, da das Codebeispiel keine **Dereferenzierung** `*pntr` enthält. Es wurde auf ein weiteres Codebeispiel für diesen **Pass** verzichtet, da din diesem das gleiche zu sehen wäre, wie in Codebeispiel 1.7.

1.3.1.2 PicoC-Blocks Pass

Die Aufgabe des **PicoC-Blocks Pass** 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.

1.3.1.2.1 Abstrakte Grammatik

Zur Umsetzung dieses Passes ist es notwendig die **Abstrakte Grammatik** 1.3.1 der Sprache L_{PicoC_Shrink} um die Knoten zu erweitern, die am Anfang dieses Unterkapitels 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 Anweisungen** `stmt` mehr. Das resultiert in der **Abstrakten Grammatik** 1.3.2 der Sprache L_{PicoC_Blocks}.

stmt	::=	$SingleLineComment(\langle str\rangle, \langle str\rangle) \mid RETIComment()$	*L_Comment*
un_op	::=	$Minus() \mid Not()$	*L_Arith_Bit*
bin_op	::=	$Add() \mid Sub() \mid Mul() \mid Div() \mid Mod()$	
	|	$Oplus() \mid And() \mid Or()$	
exp	::=	$Name(\langle str\rangle) \mid Num(\langle str\rangle) \mid Char(\langle str\rangle)$	
	|	$BinOp(\langle exp\rangle, \langle bin\_op\rangle, \langle exp\rangle)$	
	|	$UnOp(\langle un\_op\rangle, \langle exp\rangle) \mid Call(Name('input'), Empty())$	
	|	$Call(Name('print'), \langle exp\rangle)$	
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	::=	$Alloc(\langle type\_qual\rangle, \langle datatype\rangle, Name(\langle str\rangle))$	
stmt	::=	$Assign(\langle exp\rangle, \langle exp\rangle)$	
datatype	::=	$PntrDecl(Num(\langle str\rangle), \langle datatype\rangle)$	*L_Pntr*
exp	::=	$Ref(\langle exp\rangle)$	
datatype	::=	$ArrayDecl(Num(\langle str\rangle)+, \langle datatype\rangle)$	*L_Array*
exp	::=	$Subscr(\langle exp\rangle, \langle exp\rangle) \mid Array(\langle exp\rangle+)$	
datatype	::=	$StructSpec(Name(\langle str\rangle))$	*L_Struct*
exp	::=	$Attr(\langle exp\rangle, Name(\langle str\rangle))$	
	|	$Struct(Assign(Name(\langle str\rangle), \langle exp\rangle)+)$	
decl_def	::=	$StructDecl(Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))+)$	
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	::=	$Call(Name(\langle str\rangle), \langle exp\rangle*)$	*L_Fun*
stmt	::=	$Return(\langle exp\rangle)$	
decl_def	::=	$FunDecl(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*)$	
	|	$FunDef(\langle datatype\rangle, Name(\langle str\rangle),$ $Alloc(Writeable(), \langle datatype\rangle, Name(\langle str\rangle))*, \langle block\rangle*)$	
block	::=	$Block(Name(\langle str\rangle), \langle stmt\rangle*)$	*L_Blocks*
stmt	::=	$GoTo(Name(\langle str\rangle))$	
file	::=	$File(Name(\langle str\rangle), \langle decl\_def\rangle*)$	*L_File*

Grammatik 1.3.2: *Abstrakte Grammatik der Sprache L_{PiocC_Blocks} in ASF*

Anmerkung

Eine **Abstrakte Grammatik** soll im Gegensatz zu einer **Konkreten Grammatik** für den **Programmierer**, der einen darauf aufbauenden Compiler implementiert **einfach verständlich** sein und stellt daher eine **Obermenge** aller tatsächlich möglichen **Kompositionen** von **Knoten** dar[a].

[a] D.h. auch wenn dort `exp` als **Attribut** steht, kann dort **nicht** jeder Knoten, der sich aus dem **Nicht-Terminalsymbol** `exp` ergibt auch wirklich eingesetzt werden.

1.3.1.2.2 Codebeispiel

In Code 1.8 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 1.7 mithilfe des **PicoC-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel in Code 1.6 aus Unterkapitel 1.3.1.1 weitergeführt. Es wurden nun eigene **Blöcke** für die Funktion `faculty` und die `main`-Funktion erstellt, in denen die jeweils **ersten** Anweisungen der jeweiligen Funktionen bis zur **letzten** Anweisung 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
          Name 'while_branch.4',
          [
            // If(Atom(Name('n'), Eq('=='), Num('1')), []),
            IfElse
              Atom(Name('n'), Eq('=='), Num('1')),
              [
                GoTo(Name('if.3'))
              ],
              [
                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

Die Aufgabe des **PicoC-ANF Pass** ist es den **Abstrakten Syntaxbaum** der Sprache L_{PicoC_Blocks} in die **Abstrakte Grammatik** der Sprache L_{PicoC_ANF} umzuformen, welche in **A-Normalform** (Definition **??**) und damit auch in **Monadischer Normalform** (Definition **??**) ist. Um Redundanz zu vermeiden, wird zur Erklärung der **A-Normalform** auf Unterkapitel **??** verwiesen und zur Erklärung der **Monadischen Normalform** auf Unterkapitel **??** verwiesen.

Zudem wird eine **Symboltabelle** (Definition 1.8) verwendet. In der Symboltabelle werden **Symboltabelleneintrag** erstellt, deren **Attribute** Informationen transportieren. Die **Verwendungszwecke** der **Attribute** eines Symboltabelleneintrag werden in Tabelle 1.9 erklärt.

Bezeichnung des Attributs	Verwendung
`type_qualifier`	**Type Qualifier**, wie `Const()` und `Writeable()` für eine **nicht beschreibbare Konstante** (z.B. `const int var = 42`) und **beschreibbare Variable** bei Nicht-Angabe von `const` (z.B. `int var`).
`datatype`	**Datentyp**, **Funktionsprototyp** (Definition **??**).
`name`	**Bezeichner** von **Konstanten**, **Variablen**, **Funktionen**, **Datentyp** usw. Bei **Variablen**, **Konstanten** und **Verbundsattributen** wird ein **Suffix** `@scope` angehängt, der die Zugehörigkeit zum **Sichtbarkeitsbereich** von z.B. einer bestimmten **Funktion** oder einem bestimmten **Verbundstyp** darstellt.
`value_or_adress`	**Wert** einer **Konstanten**. **Adresse** einer **Variablen** oder **Funktion**. Bei der **Deklaration** eines **Verbundstyps** werden seine **Attribute** hier als Liste abgespeichert.
`position`	**Position** des **Lexemes** mit **Zeilennummer** und **Spaltennummer** innerhalb der Textdatei eines Programms.

Tabelle 1.9: *Attribute eines Symboltabelleneintrags.*

In der **Symboltabelle** wird beim Anlegen eines **neuen Eintrags** für eine Variable zunächst eine **Adresse** an das `value or address`-Attribut dieses **Eintrags** 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** der momentan **aktiven Funktion**. Einer der beiden **Counter** wird nach **Anlegen eines Eintrags** entsprechend der **Größe** der angelegten Variable **hochgezählt**.

Kommt im **Programmcode** an einer späteren Stelle eine **definierte** Variable `Name('var')` vor, so wird mit der **Konkatenation** des Bezeichners der **Variable** und des Bezeichners des momentanen **Sichtbarkeitsbereichts** `var@scope`[23] als **Schlüssel** in der **Symboltabelle** der entsprechende **Symboltabelleneintrag** der Variable `var` nachgeschlagen. Anstelle des `Name(str)`-Knotens der Variable wird ein `Global(num)`- bzw. `Stackframe(num)`-Knoten eingefügt, dessen `num`-Attribut die **Adresse** im `value or address`-Attribut des **Symboltabelleneintrags** zugewiesen bekommt.

Ob der `Global(num)`- oder der `Stackframe(num)`-Knoten für die Ersetzung verwendet wird, entscheidet sich anhand des **Sichtbarkeitsbereichs** `scope`. Für den **Sichtbarkeitsbereich** der `main`-Funktion wird der `Global(num)`-Knoten verwendet. Für den **Sichtbarkeitsbereich** jeder anderen Funktion wird der `Stackframe(num)`-Knoten verwendet.

Darüberhinaus gibt es den **Sichtbarkeitsbereich** `global!`, dessen Variablen und Konstanten **überall sichtbar** sind und der für **globale Variablen** verwendet wird. Der **Sichtbarkeitsbereich** `global!` hat die **geringste Priorität** aller Sichtbarkeitsbereiche. Das bedeutet, dass, wenn im **Sichtbarkeitsbereich einer Funktion** und im **Sichtarkeitsbereich** `global!` zweimal der **gleiche Bezeichner** vorkommt, dem Sichtbarkeitsbereich der Funktion **Vorrang** gelassen wird. Für den **Sichtbarkeitsbereich** `global!` wird der `Global(num)`-Knoten verwendet.

Das Symbol `'!'` im **Suffix** `global!` und das Symbol `'@'` als Trennzeichen in `var@scope` wurden aus einem bestimmten Grund verwendet, nämlich, weil kein **Bezeichner** die Symbole `'@'` und `!` jemals selbst enthalten kann. Die **Produktionen** für einen Bezeichner in der **Konkretten Grammatik** $G_{Lex} \uplus G_{Parse}$ (siehe 1.1.1 und 1.2.10) lassen beide Symbole `@` und `!` nicht zu. Damit ist es ausgeschlossen, dass es zu **Problemen** kommt, falls ein Benutzer des **PicoC-Compilers** zufällig auf die Idee kommt eine Funktion auf eine unpassende Weise zu benennen[24].

[23] Die Umsetzung von **Sichtbarkeitsbereichen** wird in Unterkapitel **??** genauer beschrieben.
[24] Z.B. `var@fun2` oder `global` als **Funktionsname**.

Definition 1.8: Symboltabelle

*Eine meist über ein **Assoziatives Feld** umgesetzte **Datenstruktur**, die notwendig ist, um das Konzept von **Variablen** und **Konstanten** 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, welche später nicht mehr so einfach zugänglich sind, wie zu dem Zeitpunkt, zu dem sie in einen **Symboltabelleneintrag** gespeichert werden.*

*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 der **Code Generierung** werden **Bezeichner** als **Symbole** bezeichnet.

1.3.1.3.1 Abstrakte Grammatik

Zur Umsetzung dieses **Passes** ist es notwendig die **Abstrakte Grammatik** 1.3.2 der Sprache L_{PicoC_Blocks} 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** enthalten können. Wie es bereits im Unterkapitel **??** erklärt wurde, ist beim **PicoC-Compiler** der Knoten `Stack(Num(str))` der einzige **Atomare Knoten**.

Des Weiteren werden auch **Funktionen** und **Funktionsaufrufe** aufgelöst, sodass die **Blöcke** `Block(Name(str), stmt*)` nun direkt im `decls_defs_blocks`-Attribut des `File(Name(str), decls_defs_blocks*)`-Knoten liegen usw.[25]. Die **Symboltabelle** ist ebenfalls als **Abstrakter Syntaxbaum** umgesetzt, wofür in der **Abstrakten Grammatik** 1.3.3 der Sprache L_{PicoC_ANF} neue Knoten eingeführt wurden. Das ganze resultiert in der **Abstrakten Grammatik** 1.3.3 der Sprache L_{PicoC_ANF}.

[25] Im Unterkapitel **??** wird das **Auflösen von Funktionen** genauer erklärt.

stmt	::=	*SingleLineComment*(⟨*str*⟩, ⟨*str*⟩) | *RETIComment*()	*L_Comment*
un_op	::=	*Minus*() | *Not*()	*L_Arith_Bit*
bin_op	::=	*Add*() | *Sub*() | *Mul*() | *Div*() | *Mod*()	
	|	*Oplus*() | *And*() | *Or*()	
exp	::=	*Name*(⟨*str*⟩) | *Num*(⟨*str*⟩) | *Char*(⟨*str*⟩) | *Global*(*Num*(⟨*str*⟩))	
	|	*Stackframe*(*Num*(⟨*str*⟩)) | *Stack*(*Num*(⟨*str*⟩))	
	|	*BinOp*(*Stack*(*Num*(⟨*str*⟩)), ⟨*bin_op*⟩, *Stack*(*Num*(⟨*str*⟩)))	
	|	*UnOp*(⟨*un_op*⟩, *Stack*(*Num*(⟨*str*⟩))) | *Call*(*Name*('*input*'), *Empty*())	
	|	*Call*(*Name*('*print*'), ⟨*exp*⟩)	
	|	*Exp*(⟨*exp*⟩)	
un_op	::=	*LogicNot*()	*L_Logic*
rel	::=	*Eq*() | *NEq*() | *Lt*() | *LtE*() | *Gt*() | *GtE*()	
bin_op	::=	*LogicAnd*() | *LogicOr*()	
exp	::=	*Atom*(*Stack*(*Num*(⟨*str*⟩)), ⟨*rel*⟩, *Stack*(*Num*(⟨*str*⟩)))	
	|	*ToBool*(*Stack*(*Num*(⟨*str*⟩)))	
type_qual	::=	*Const*() | *Writeable*()	*L_Assign_Alloc*
datatype	::=	*IntType*() | *CharType*() | *VoidType*()	
exp	::=	*Alloc*(⟨*type_qual*⟩, ⟨*datatype*⟩, *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*⟩), ⟨*datatype*⟩)	*L_Pntr*
	|	*Ref*(*Global*(⟨*str*⟩)) | *Ref*(*Stackframe*(⟨*str*⟩))	
	|	*Ref*(*Subscr*(⟨*exp*⟩, ⟨*exp*⟩ | *Ref*(*Attr*(⟨*exp*⟩, *Name*(⟨*str*⟩)))))	
datatype	::=	*ArrayDecl*(*Num*(⟨*str*⟩)+, ⟨*datatype*⟩)	*L_Array*
exp	::=	*Subscr*(⟨*exp*⟩, *Stack*(*Num*(⟨*str*⟩))) | *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	::=	*IfElse*(*Stack*(*Num*(⟨*str*⟩)), ⟨*stmt*⟩∗, ⟨*stmt*⟩∗)	*L_If_Else*
exp	::=	*Call*(*Name*(⟨*str*⟩), ⟨*exp*⟩∗)	*L_Fun*
stmt	::=	*StackMalloc*(*Num*(⟨*str*⟩)) | *NewStackframe*(*Name*(⟨*str*⟩), *GoTo*(⟨*str*⟩))	
	|	*Exp*(*GoTo*(*Name*(⟨*str*⟩))) | *RemoveStackframe*()	
	|	*Return*(*Empty*()) | *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*⟩), ⟨*block*⟩∗)	*L_File*
symbol_table	::=	*SymbolTable*(⟨*symbol*⟩∗)	*L_Symbol_Table*
symbol	::=	*Symbol*(⟨*type_qual*⟩, ⟨*datatype*⟩, ⟨*name*⟩, ⟨*val*⟩, ⟨*pos*⟩, ⟨*size*⟩)	
type_qual	::=	*Empty*()	
datatype	::=	*BuiltIn*() | *SelfDefined*()	
name	::=	*Name*(⟨*str*⟩)	
val	::=	*Num*(⟨*str*⟩) | *Empty*()	
pos	::=	*Pos*(*Num*(⟨*str*⟩), *Num*(⟨*str*⟩)) | *Empty*()	
size	::=	*Num*(⟨*str*⟩) | *Empty*()	

Grammatik 1.3.3: *Abstrakte Grammatik der Sprache L_{PiocC_ANF} in ASF*

1.3.1.3.2 Codebeispiel

In Code 1.9 sieht man die als **Abstrakter Syntaxbaum** umgesetzte **Symboltabelle**, in der alle **Variablen** und **Funktionen** aus dem weitergeführten Beispiel in Code 1.6 aus Unterkapitel 1.3.1.1 einen Eintrag haben.

```
SymbolTable
  [
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(IntType('int'), Name('faculty'), [Alloc(Writeable(),
        ↪  IntType('int'), Name('n'))])
        name:                   Name('faculty')
        value or address:       Empty()
        position:               Pos(Num('3'), Num('4'))
        size:                   Empty()
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('n@faculty')
        value or address:       Num('0')
        position:               Pos(Num('3'), Num('16'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Writeable()
        datatype:               IntType('int')
        name:                   Name('res@faculty')
        value or address:       Num('1')
        position:               Pos(Num('4'), Num('6'))
        size:                   Num('1')
      },
    Symbol
      {
        type qualifier:         Empty()
        datatype:               FunDecl(VoidType('void'), Name('main'), [])
        name:                   Name('main')
        value or address:       Empty()
        position:               Pos(Num('14'), Num('5'))
        size:                   Empty()
      }
  ]
```

Code 1.9: *Symboltabelle für Codebespiel.*

In Code 1.10 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 1.8 mithilfe des **PicoC-ANF Pass** resultierenden **Abstrakten Syntaxbaum**. Alle Anweisungen und Ausdrücke sind in **A-Normalform**, z.B. sind die `IfElse(exp, stmts1, stmts2)`-Knoten dadurch in **A-Normalform**, dass ihre **Komplexen Ausdrücke** im `exp`-Attribut in ein `Exp(exp)`-Knoten eingesetzt und vorgezogen werden (z.B. `Exp(Atom(Stack(Num('2')), Eq('=='), Stack(Num('1'))))`). Die `Exp(exp)`-Knoten weisen die **Ergebnisse** der Komplexen Ausdrücke **Locations** zu, welche sich beim **PicoC-Compiler** auf dem **Stack** befinden. Die **Ergeb-**

nisse werden dann über **Atomare Ausdrücke** `Stack(Num(str))` vom Stack gelesen: `IfElse(Stack(Num(str)), stmts1, stmts2)`.

Funktionen sind nur noch über die `Name(str)`-Knoten von **Blöcken** zu erkennen, die den gleichen **Bezeichner** haben, wie die ursprüngliche **Funktion** und es lässt sich nur durch das **Nachverfolgen** bestimmter `GoTo(Name(str))`-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'))
          ]
      ],
    Block
      Name 'if.3',
      [
        // 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 1.10: *PicoC-ANF Pass für Codebespiel.*

1.3.1.4 RETI-Blocks Pass

Die Aufgabe des **RETI-Blocks Pass** ist es die **PicoC-Knoten** der **Anweisungen** in den **Blöcken** des **Abstrakten Syntaxbaums** der Sprache L_{PicoC_ANF} durch **semantisch** entsprechende **RETI-Knoten** zu ersetzen.

1.3.1.4.1 Abstrakte Grammatik

Die **Abstrakte Grammatik** 1.3.4 der Sprache L_{RETI_Blocks} ist verglichen mit der **Abstrakten Grammatik** 1.3.3 der Sprache L_{PicoC_ANF} stark verändert, denn der Großteil der **PicoC-Knoten** wird in diesem Pass durch **semantisch** 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 1.3.1.6 aufgelöst wird.

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

Grammatik 1.3.4: *Abstrakte Grammatik der Sprache L_{RETI_Blocks} in ASF*

1.3.1.4.2 Codebeispiel

In Code 1.11 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 1.10 mithilfe des **RETI-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. Die **Anweisungen**, die durch entsprechende **PicoC-Knoten** im **Abstrakten Syntaxbaum** der Sprache L_{PicoC_ANF} repräsentiert waren, werden nun durch **semantisch** entsprechende **RETI-Knoten** ersetzt.

```
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'))
        SUBI SP 1;
        LOADI ACC 1;
        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 1.11: *RETI-Blocks Pass für Codebespiel.*

Anmerkung

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

1.3.1.5 RETI-Patch Pass

Die Aufgabe des **RETI-Patch Pass** ist das **Ausbessern** (engl. to patch) des **Abstrakten Syntaxbaumes** der Sprache L_{RETI_Blocks} durch:

- das **Einfügen** eines `start.<number>`-Blocks, 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** vor jede **Division** `Instr(Div(), args)`, die prüfen, ob durch `0` geteilt wird. Ist es der Fall, dass durch 0 geteilt wird, dann wird in das `ACC`-Register der Fehlercode 1 geschrieben, der für die Fehlerart `DivisionByZero` steht und die Ausführung des Programmes wird **beendet**. Andernfalls **läuft** das Programm **weiter**.
- das Überprüfen darauf, ob bestimmte **Immediates** `Im(str)` in Befehlen, wie z.B. `Jump(rel, Im(str))`, `Instr(Loadin(), [reg1, reg2, Im(str)])`, `Instr(Loadi(), [reg, Im(str)])` usw. **kleiner** als -2^{21} oder **größer** als $2^{21} - 1$ sind. Im diesem Fall muss der **gewünschte Zahlenwert** durch **Bitshiften** und Anwenden von **Bitweise ODER** aus Zahlenwerten berechnet werden, die sich im Zahlenbereich zwischen -2^{21} und $2^{21} - 1$ befinden. Im Fall dessen, dass der **Immediate** allerdings **kleiner** als $-(2^{31})$ oder **größer** als $2^{31} - 1$ ist, wird eine Fehlermeldung `TooLargeLiteral` ausgegeben.[26]

1.3.1.5.1 Abstrakte Grammatik

[26] Einiges in diesem **Pass** fällt unter die Themenbereiche des **Bachelorprojekts** und wird daher **nicht** genauer erläutert.

Die **Abstrakte Grammatik** 1.3.5 der Sprache L_{RETI_Patch} ist im Vergleich zur **Abstrakten Grammatik** 1.3.4 der Sprache L_{RETI_Blocks} **unverändert**. Es sind **keine neuen Knoten** hinzugekommen und **keine** Knoten wurden **abgeändert** oder völlig **entfernt**.

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

Grammatik 1.3.5: *Abstrakte Grammatik der Sprache L_{RETI_Patch} in ASF*

1.3.1.5.2 Codebeispiel

In Code 1.12 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 1.11 mithilfe des **RETI-Patch Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. Durch den **RETI-Patch Pass** wurde hier ein `start.<nummer>`-Block[27] eingesetzt, da die `main`-Funktion **nicht** die **erste** Funktion ist. Des Weiteren wurden durch diesen Pass einzelne `GoTo(Name(str))`-Knoten **entfernt**, die nur einem Sprung um **eine Adresse** weiter entsprochen hätten[28].

```
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;
```

[27] Dieser `start.<nummer>`-Block wurde im Code 1.12 markiert.
[28] Diese **entfernten** `GoTo(Name(str))`-Knoten wurden in Code 1.12 markiert.

```
20        STOREIN BAF ACC -3;
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 1.12: *RETI-Patch Pass für Codebespiel.*

1.3.1.6 RETI Pass

Die Aufgabe des **RETI-Patch Pass** ist es die letzten verbliebenen **PicoC-Knoten** im **Abstrakten Syntaxbaum** zu entfernen bzw. zu ersetzen. Dementsprechend werden die **Blöcke** `Block(Name(str), instr*)` entfernt und die **Knoten** in diesen Blöcken werden genauso **zusammengefügt**, wie die Blöcke **angeordnet** waren.

Des Weiteren werden 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 einen **Immediate** `Im(str(distance_or_address))` mit passender **Adresse** oder **Distanz** oder einen **Sprungbefehl** mit passender **Distanz** `Jump(Always(), Im(str(distance)))` ersetzt. Die **Distanz-** und **Adressberechnung** wird in Unterkapitel **??** genauer erklärt.

1.3.1.6.1 Konkrete und Abstrakte Grammatik

Die **Abstrakte Grammatik** 1.3.6 der Sprache L_{RETI} hat im Vergleich zur **Abstrakten Grammatik** 1.3.5 der Sprache L_{RETI_Patch} nur noch ausschließlich **RETI-Knoten**, dementsprechend gibt es die Knoten `Instr(Loadi(), [Reg(Acc()), GoTo(Name(str))])`, `Jump(Eq(), GoTo(Name(str)))` nicht mehr. Des Weiteren gibt es **keine Blöcke** `Block(Name(str), instr*)` mehr, alle **RETI-Knoten** stehen nun in einem `Program(Name(str), instr*)`-Knoten.

Ausgegeben wird der finale Abstrakte Syntaxbaum in **Konkreter Syntax**, die durch die Konkreten Grammatiken 1.3.7 und 1.3.8 für jeweils die **Lexikalische** und **Syntaktische Analyse** $G_{RETI\_Lex} \uplus G_{RETI\_Parse}$ beschrieben wird.

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 1.3.6: *Abstrakte Grammatik der Sprache L_{RETI} in ASF*

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 1.3.7: *Konkrete Grammatik 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	::=	*(instr";")**	

Grammatik 1.3.8: *Konkrete Grammatik der Sprache L_{RETI} für die Syntaktische Analyse in EBNF*

1.3.1.6.2 Codebeispiel

In Code 1.13 sieht man den aus dem **Abstrakten Syntaxbaum** aus Code 1.12 mithilfe des **PicoC-Blocks Pass** resultierenden **Abstrakten Syntaxbaum**. Es wird das Beispiel aus Unterkapitel 1.6 weitergeführt. Es gibt **keine Blöcke** mehr und die **Knoten** in diesen Blöcken wurden **zusammengesetzt**, wie sie in den **Blöcken** angeordnet waren. Das Programm ist komplett in **RETI-Knoten** übersetzt, die allerdings in ihrer **Konkreten Syntax** ausgegeben werden.

Die letzten **Nicht-RETI-Befehle** oder **RETI-Befehle**, die **nicht** auschließlich aus **RETI-Knoten** bestanden `LOADI ACC GoTo(Name('addr@next_instr'))`, `Exp(GoTo(Name('main.0')))` und `JUMP== GoTo(Name('if_else_after.2'))`, wurden durch **RETI-Befehle** ersetzt, welche in Code 1.13 markiert wurden.

Der `Program(Name(str), instr)`-Knoten, der alle **RETI-Knoten** beinhaltet, gibt alleinig die **RETI-Knoten**, die er beinhaltet aus und fügt ansonsten **nichts** zur Ausgabe hinzu, wie z.B. den **Bezeichner des Programms** oder **Einrückung**. Hierdurch erzeugt der **Abstrakte Syntaxbaum**, wenn er in **Konkreter Syntax** in eine Datei ausgegeben wird direkt **RETI-Code** in **menschenlesbarer Repräsentation**.

```
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;
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 1.13: *RETI Pass für Codebespiel.*

Literatur

Online

- *ANSI C grammar (Lex)*. URL: https://www.quut.com/c/ANSI-C-grammar-l-2011.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Lex) old*. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Yacc)*. URL: https://www.quut.com/c/ANSI-C-grammar-y.html (besucht am 15. 08. 2022).
- *ANSI C grammar (Yacc) old*. URL: https://www.lysator.liu.se/c/ANSI-C-grammar-y.html (besucht am 15. 08. 2022).
- *ANTLR*. URL: https://www.antlr.org/ (besucht am 31. 07. 2022).
- *Bäume*. URL: https://www.stefan-marr.de/pages/informatik-abivorbereitung/baume/ (besucht am 17. 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

- Nystrom, Robert. *Parsing Expressions · Crafting Interpreters*. Genever Benning, 2021. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09. 07. 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, 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, 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).

Sonstige Quellen

- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).
- *Naming convention (programming)*. In: *Wikipedia*. Page Version ID: 1100066005. 24. Juli 2022. URL: https://en.wikipedia.org/w/index.php?title=Naming_convention_(programming)&oldid=1100066005 (besucht am 30. 07. 2022).
- Shinan, Erez. *lark: a modern parsing library*. Version 1.1.2. URL: https://github.com/lark-parser/lark (besucht am 31. 07. 2022).