Albert Ludwigs Universität Freiburg

Technische Fakultät

# PicoC-Compiler

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

Bachelorarbeit

---

*Abgabedatum:* 13. September 2022

*Author:*
Jürgen Mattheis

*Gutachter:*
Prof. Dr. Scholl

*Betreung:*
M.Sc. Seufert

---

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

**ERKLÄRUNG**

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

# Danksagungen

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

Wie mein Betreuer M.Sc. Tobias Seufert und wahrscheinlich auch mein Gutachter Prof. Dr. Scholll im Verlauf dieser Bachelorarbeit und des vorangegangenen Bachelorprojektes gemerkt haben, bin ich keine Person, die irgendwelche Dinge gerne so macht wie es üblich ist, ich schreibe meine Danksagung nicht auf eine bestimmte Weise, nur weil sich irgendwann mal etabliert hat wie eine Danksagung üblicherweise aussieht. Ich halte nicht viel von 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.

Ich halte es eher so, dass wenn mir wirklich etwas am Dank gegenüber Personen liegt, ich mir wirklich den Aufwand mache einen Text zu schreiben in dem ich diesen zum Ausdruck bringe, im anderen Fall kann man sich bei mir auf die typischen Standardfloskeln einstellen. Bei dieser Bachelorarbeit kann ich nur auf ersteres Zurückgreifen. 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 sehr respektvoll, 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 tuen, wie es die Anforderungen verlangen und nichts darüberhinaus tuen, 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 tuen, auch wenn es für sie keine Vorteile hat. Tobias 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[1], er hat sich bei der Korrektur dieser Schrifftlichen Ausarbeitung sogar die Mühe gemacht bei den einzelnen Problemstellen längere Textkommentare zu verfassen und das trotz dessen, dass meine Bachelorarbeit recht Umfangreich zu lesen ist[2] 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 meinen Betreuer Tobias will ich einen Kommentar zu meinem Gutachter Prof. Dr. Scholl anschließen. Ich hab während meines Bachelorprojekts und meiner Bachelorarbeit wahrscheinlich einen ziemlich eigensinnigen Eindruck gemacht, bei der Weise, wie ich bestimmte Dinge umsetzen wollte. 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.

[1] Wofür ich mich auch nochmal Entschuldigen will.
[2] Wobei er sich kein einziges Mal in geringster Weise entnervt darüber gezeigt hat.

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 Michel Giehl sich netterweise zur Verfügung gestellt hat. 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[3] zu finden.

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, was man auch selten im Studium erlebt, dass dem 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 ziemlich viel unerwartete Zeit zum Implementieren gebraucht haben. Mit den von Prof. Dr. Scholl festgelegten Funktionalitäten für die Bachelorarbeit ist der Zeitplan ziemlich perfekt aufgegangen. Mit meinen ambitionierten Plänen wäre es bei der Bachelorarbeit dagegeben wohl nichts geworden. Man hat daran gemerkt, dass Prof. Dr. Scholl das Wohlergehen der Studenten wichtig ist.

Mir hat die Implementierung des PicoC-Compilers tatsächlich ziemlich viel Spaß gemacht, da Compilerbau auch in mein perönliches Interessengebiet fällt. Das Aufschreiben dieser Schrifftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht[4]. 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, die sonst ziemlich wilkürlich erscheinen würde, 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 großartigen Bachelorarbeit und des vorangegangenen Bachelorprojektes und Michel Giehl für das Integrieren des PicoC-Compilers in seinen RETI-Emulator.

[3] https://github.com/michel-giehl/Reti-Emulator.

[4] Da es recht stressig ist im Kopf zu behalten, was man schon erklärt hat und wo noch eine Verständnislücke vorliegen könnte.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Codeverzeichnis

# Tabellenverzeichnis

# Definitionsverzeichnis

# Grammatikverzeichnis

# 1 Motivation

Als Programmierer kommt man nicht drumherum einen **Compiler** zu nutzen, er ist geradezu **essentiel** für den Beruf oder das Hobby des Programmierens. Selbst in der Programmiersprachen $L_{Python}$, welche als **interpretierte** Sprache bekannt ist, wird das in der Programmiersprache $L_{Python}$ geschriebene Programm vorher zu **Bytecode** kompiliert, bevor dieser von der **Python Virtual Machine (PVM)** interpretiert wird.

Compiler, wie der **GCC**[1] oder **Clang**[2] werden üblicherweise über eine **Commandline-Schnittstelle** verwendet, welche es für den Benutzer **unkompliziert** macht ein Programm, dass in der Programmiersprache geschrieben ist, die der Compiler kompiliert[3] zu **Maschinencode** zu kompilieren.

Meist funktioniert das über schlichtes und einfaches **Angeben der Datei**, die das Programm enthält, welches kompiliert werden soll, z.B. im Fall des **GCC** über `> gcc program.c -o machine_code`[4]. Als Ergebnis erhält man im Fall des **GCC** die mit der Option `-o` selbst benannte Datei `machine_code`, welche dann zumindest unter **Unix** über `> ./machine_code` **ausgeführt** werden kann, wenn das **Ausführungsrecht** gesetzt ist. Das gesamte gerade erläuterte Vorgehen ist in Abbildung 1.1 veranschaulicht.

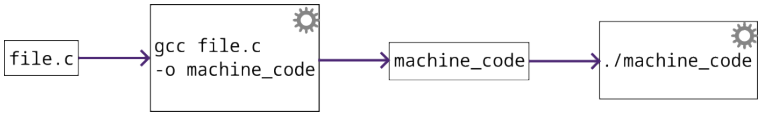



**Abbildung 1.1:** *Schritte zum Ausführen eines Programmes mit dem GCC.*

Der ganze Kompiliervorgang kann, wie er in Abbildung 1.2 dargestellt ist zu einer Box abstrahiert werden. Der Benutzer gibt ein **Programm** in der Sprache des Compilers rein und erhält **Maschinencode**, den er dann im besten Fall in eine andere Box hineingeben kann, welche die passende **Maschine** oder den passenden **Interpreter** in Form einer **Virtuellen Maschine** repräsentiert, der bzw. die den **Maschinencode** ausführen kann.

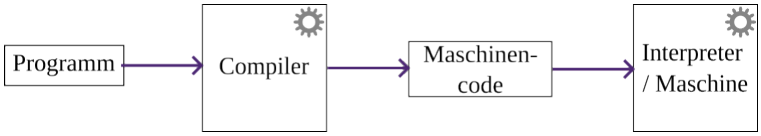



**Abbildung 1.2:** *Stark vereinfachte Schritte zum Ausführen eines Programmes.*

[1] *GCC, the GNU Compiler Collection - GNU Project*.
[2] *clang: C++ Compiler*.
[3] Im Fall des **GCC** und **Clang** ist es die Programmiersprache $L_C$.
[4] Bei **mehreren Dateien** ist das ganze allerdings etwas komplizierter, weil der **GCC** beim Angeben aller `.c`-Dateien nacheinander `gcc program_1.c ... program_n.c` nicht darauf achtet doppelten Code zu entfernen. Beim **GCC** muss am besten mittels einer **Makefile** dafür gesorgt werden, dass jede Datei einzeln zu **Objectcode** (Definition 3.3) kompiliert wird. Das Kompilieren zu **Objectcode** geht mittels des Befehls `gcc -c program_1.c ... program_n.c` und alle **Objectdateien** können am Ende mittels des **Linkers** mit dem Befehl `gcc -o machine_code program_1.o ... program_n.o` zusammen gelinkt werden.

Der Programmierer muss für das Vorgehen in Abbildung 1.2 **nichts** über die **Theoretischen Grundlagen des Compilerbau** wissen, noch wie der Compiler **intern** umgesetzt ist. In dieser Bachelorarbeit soll diese **Compilerbox** allerdings geöffnet werden und anhand eines eigenen im Vergleich zum **GCC** im Funktionsumfang **reduzierten Compilers** gezeigt werden, wie so ein Compiler **unter der Haube** stark vereinfacht funktionieren könnte.

Die konkrete **Aufgabe** besteht darin einen sogenannten **PicoC-Compiler** zu implementieren, der die **Programmiersprache** $L_{PicoC}$, welche eine Untermenge der Sprache $L_C$ ist[5] in eine zu **Lernzwecken** prädestinierte, **unkompliziert** gehaltene **Maschinensprache** $L_{RETI}$ kompilieren kann. Im Unterkapitel 1.1 wird näher auf die **RETI-Architektur** eingegangen, die der Sprache $L_{RETI}$ zu Grunde liegt und im Unterkapitel 1.2 wird näher auf die die Sprache $L_{PicoC}$ eingegangen, welche der **PicoC-Compiler** zur eben erwähnten Sprache $L_{RETI}$ kompilieren soll.

## 1.1 RETI-Architektur

Die **RETI-Architektur** ist eine zu Lernzwecken für die Vorlesungen P. D. C. Scholl, „Betriebssysteme“ und P. D. C. Scholl, „Technische Informatik“ entwickelte **32-Bit** Architektur, die sich vor allem durch ihre einfache Zugänglichkeit kennzeichnet und deren **Maschinensprache** $L_{RETI}$ als Zielsprache des **PicoC-Compilers** hergenommen wurde. In der Vorlesung P. D. C. Scholl, „Technische Informatik“ wird die **grundlegende RETI-Architektur** erklärt und in der Vorlesung P. D. C. Scholl, „Betriebssysteme“ wird diese Architektur erweitert, sodass diese mehr darauf angepasst ist, dass auch komplexere Kontrukte, wie ein **Betriebssystem**, **Interrupts**, **Prozesse**, **Funktionen** usw. auf nicht zu komplizierte Weise implementiert werden können.

Um den den **PicoC-Compiler** zu **testen** war es notwendig einen **RETI-Interpreter** zu implementieren, der genau die Variante der **RETI-Achitektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ **simuliert**.

**Anmerkung**

In dieser **Bachelorarbeit** wird **im Folgenden** bei der **Maschinensprache** $L_{RETI}$ immer von der Variante, welche durch die **RETI-Architektur** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ umgesetzt ist ausgegangen.

Die **Register** der **RETI-Architektur** werden in Tabelle 1.1 aufgezählt und erläutert. Die **Maschinenbefehle** und **Datenpfade** der **RETI-Architektur** sind im Kapitel Appendix dokumentiert, da diese **nicht explizit** zum **Verständnis** der späteren Kapitel notwendig sind, aber zum **vollständigen Verständnis** notwendig sind, um die später auftauchenden **RETI-Befehle** usw. **zu verstehen**. Der **Aufbau** der **Maschinensprache** $L_{RETI}$ ist durch Grammatik **??** und Grammatik **??** zusammengenommen beschrieben. Für genauere **Implementierungsdetails** ist allerdings auf die Vorlesungen P. D. C. Scholl, „Technische Informatik“ und P. D. C. Scholl, „Betriebssysteme“ zu verweisen.

[5] Die der **GCC** kompilieren kann.

| Register Kürzel | Register Ausgeschrieben | Aufgabe |
|---|---|---|
| PC | Program Counter | Zeigt auf den Maschinenbefehl, der als nächstes ausgeführt werden soll. |
| ACC | Accumulator | Für Operanden von Operationen oder für temporäre Werte. |
| IN1 | Indexregister 1 | Hat dieselbe Aufgabe wie das ACC-Register. |
| IN2 | Indexregister 2 | Hat dieselbe Aufgabe wie das ACC-Register. |
| SP | Stackpointer | Zeigt immer auf die erste freie Speicherzelle am Ende des Stacks, wo als nächstes Speicher allokiert werden kann. |
| BAF | Begin Aktive Funktion | Zeigt auf den Beginn des Stackframes der aktuell aktiven Funktion. |
| CS | Codesegment | Zeigt auf den Beginn des Codesegments. Die letzten 10 Bits werden verwendet, um 22 Bit Immediates aufzufüllen. Kann dadurch dazu verwendet werden, festzulegen welcher der 3 Peripheriegeräte[a] in der Memory Map[b] angesprochen werden soll. |
| DS | Datensegment | Zeigt auf den Beginn des Datensegments. |

[a] EPROM, UART und SRAM.
[b] Da die Memory Map zum Verständnis der Bachelorarbeit nicht wichtig ist, wird diese nicht mehr als nötig im weiteren Verlauf erläutert.

**Tabelle 1.1:** *Präzedenzregeln von PicoC.*

Die RETI-Architektur ermöglicht bei der Ausführung von RETI-Programmen Prozesse zu nutzen. In Abbildung 1.3 ist der Aufbau eines Prozesses im Hauptspeicher der RETI-Architektur zu sehen. Das RETI-Programm nutzt dabei den Stack für temporäre Zwischenergebnisse von Berechnungen und zum Anlegen der Stackframes von Funktionen, welche die Lokalen Variablen und Parameter einer Funktion speichern. Das SP- und BAF-Register erfüllen dabei ihre zugeteilten Aufgaben für den Stack.

Der Abschnitt für die Globalen Statischen Daten ist allgemein dazu da Daten zu beherbergen, die für den Rest der Programmausführung global zugänglich sein sollen, aber auch für die Lokalen Variablen der main-Funktion. Das DS-Register markiert den Anfang des Datensegments und damit auch den Anfang der Globalen Statischen Daten und kann als relativer Orientierungspunkt beim Zugriff und Abspeichern Globaler Statischer Daten dienen. Das CS-Register wird als relativer Orientierungspunkt genutzt, an dem die Ausführung von RETI-Programmen startet und zur Bestimmung der relativen Startadresse, an welcher der RETI-Code einer bestimmten Funktion anfängt.

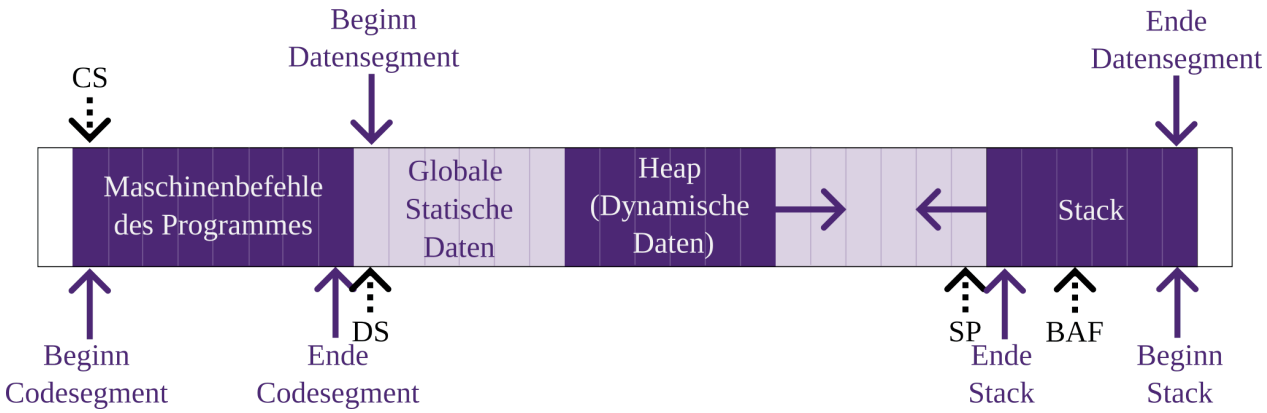



**Abbildung 1.3:** *Speicherorganisation.*

Die RETI-Architektur nutzt 3 verschiedene **Peripheriegeräte**, **EPROM**, **UART** und **SRAM**, die über eine **Memory Map**[6] den über die **Datenpfade** der RETI-Architektur 3.1 ansprechbaren **Adressraum** von $2^{32}$ Adressen[7] unter sich aufteilen.

Die **Ausführung** eines Programmes **startet** auf die einfachste Weise, indem es von einem **Startprogramm** im **EPROM**[8] aufgerufen wird. Der **EPROM** wird beim Start einer **RETI-CPU** als **erstes** aufgerufen, da nach der **Memory Map** der erste **Adressraum** von 0 bis $2^{30} - 1$ dem **EPROM** zugeordnet ist und das `PC`-Register **initial** den Wert 0 hat, also als **erstes** das Programm ausgeführt wird, welches an **Adresse** 0 im EPROM anfängt.

Die **UART**[9] ist eine **elektronische Schaltung** mit je nach Umsetzung mehr oder weniger **Pins**, wobei es allerdings immer einen `RX`- und einen `TX`-Pin gibt, für jeweils Empfangen[10] und Versenden[11] von Daten gibt. Jeder der **Pins** wird dabei mit einer anderen **Adresse** von $2^3$ verschiedenen Adressen angsprochen und jeweils 8-Bit können nach den Datenpfaden 3.1 auf einmal über einen Pin in ein **Register** der **UART** geschrieben werden, um **versandt** zu werden oder von einem Pin **empfangen** werden. Die **UART** kann z.B. genutzt werden, um Daten an einen sehr einfach gehaltenen **Monitor** zu senden, der diese dann anzeigt.

An letzter Stelle muss der **SRAM**[12] erwähnt werden, bei dem es sich um den **Hauptspeicher** der **RETI-CPU** handelt. Der **Zugriff auf den Hauptspeicher** ist deutlich schneller als z.B. auf ein **externes Speichermedium**, aber **langsamer** als der **Zugriff auf Register**.

## 1.2 Die Sprache PicoC

Die Sprache $L_{PicoC}$ ist eine Untermenge der Sprache $L_C$, welche

- **Einzeilige Kommentare** `//` and **Mehrzeilige Kommentare** `/*` and `*/`.
- die **Primitiven Datentypen** `int`, `char` und `void`.
- die **Abgeleiteten Datentypen Felder** (z.B. `int ar[3]`), **Verbunde** (z.B. `struct st {int attr1; attr2;}`) und **Zeiger** (z.B. `int *pntr`).
- `if(cond){ }`- / `else{ }`-**Statements**[13].
- `while(cond){ }`- und `do while(cond){ };`-**Statements**.
- **Arithmetische Ausdrücke**, welche mithilfe der **binären Operatoren** `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^` und **unären Operatoren** `-`, $\sim$ umgesetzt sind.
- **Logische Ausdrücke**, welche mithilfe der **Relationen** `==`, `!=`, `<`, `>`, `<=`, `>=` und **Logischer Verknüpfungen** `!`, `&&`, `||` umgesetzt sind.
- **Zuweisungen**, die mit dem **Zuweisungsoperator** `=` umgesetzt sind.
- **Funktionsdeklaration** (z.B. `int fun(int arg1[3], struct st arg2);`), **Funktionsdefinition** (z.B. `int fun(int arg1[3], struct st arg2){}`) und **Funktionsaufrufe** (z.B. `fun(ar, st_var)`).

[6] Da die **Memory Map** zum Verständnis der Bachelorarbeit **nicht wichtig** ist, sondern nur bei der Umsetzung des **RETI-Interpreters**, wird diese **nicht näher erläutert** als notwendig.
[7] Von 0 bis $2^{32} - 1$.
[8] Kurz für **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory.
[9] Kurz für **U**niversal **A**synchronous **R**eceiver **T**ransmitter.
[10] Engl. **R**eceiving, daher das `R`.
[11] Engl. **T**ransmission, daher das `T`.
[12] Kurz für **S**tatic **r**andom-**a**ccess **m**emory.
[13] Was die Kombination von `if` und `else`, nämlich `else if(cond){ }` miteinschließt.

beinhaltet. Die ausgegrauten • wurden bereits für das **Bachelorprojekt** umgesetzt und mussten für die **Bachelorarbeit** nur an die **neue Architektur** angepasst werden.

Der **Aufbau** der **Programmiersprache** $L_C$ ist durch Grammatik **??** und Grammatik **??** zusammengenommen beschrieben.

## 1.3 Eigenheiten der Sprachen C und PicoC

Einige **Eigenheiten** der Programmiersprache $L_C$, die genauso ein Teil der Programmiersprache $L_{PicoC}$ sind, da $L_{PicoC}$ eine Untermenge von $L_C$ ist und welche in der Implementierung des **PicoC-Compilers** in Kapitel **??** noch eine **wichtige Rolle** spielen werden im Folgenden genauer erläutert. Im Folgenden wird immer von der Programmiersprache $L_{PicoC}$ gesprochen, da es in dieser Bachelorarbeit um diese geht und die folgenden Beispiele für die Ausgaben alle mithilfe des **PicoC-Compilers** und **RETI-Interpreters** **kompiliert** bzw **ausgeführt** wurden, aber selbiges gilt genauso für $L_C$ aus bereits erläutertem Grund.

Bei der Programmiersprache $L_{PicoC}$ handelt es sich im eine **imperative** (Definition 1.1), **strukturierte** (Definition 1.2) und **prozedurale Programmiersprache** (Definition 1.3). Aufgrund dessen, dass es sich bei beiden um **Imperative Programmiersprachen** handelt ist es wichtig bei der Implementierung die **Reihenfolge** zu beachten und aufgrund dessen, dass es sich bei beiden um **Strukturierte** und **Prozedurale Programmiersprachen** handelt, ist es eine gute Methode bei der Implementierung auf **Blöcke**[14] zu setzen zwischen denen **hin und her** gesprungen werden kann und welche in den einzelnen Implementierungsschritten die **notwendige Datenstruktur** darstellen um **Auswahl** zwischen Codestücken, **Wiederholung** von Codestücken und **Sprünge** zu Blöcken mit entsprechend zu bestimmten **Bezeichnern** passenden **Labeln** umzusetzen.

**Definition 1.1: Imperative Programmierung**

*Wenn ein Programm aus einer **Folge von Befehlen** besteht, **deren Reihenfolge** auch bestimmt in welcher **Reihenfolge** diese **Befehle** auf einer **Maschine** ausgeführt werden.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

**Definition 1.2: Strukturierte Programmierung**

*Wenn ein Programm anstelle von z.B. `goto label`-Statements **Kontrollstruturen**, wie z.B. `if(cond) { } else { }`, `while(cond) { }` usw. verwendet, welche dem Programmcode **mehr Struktur** geben, weil die **Auswahl** zwischen Statements und die **Wiederholung** von Statements eine **klare und eindeutige Struktur** hat, welche bei Umsetzung mit einem `goto label`-Statement **nicht** so **eindeutig erkennbar** wäre und auch **nicht** umbedingt immer **gleich aufgebaut** wäre.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

**Definition 1.3: Prozedurale Programmierung**

*Programme werden z.B. mittels **Funktionen** in **überschaubare Unterprogramme** bzw. **Prozeduren** aufgeteilt, die **aufrufbar** sind. Dies **vermeidet** einerseits **redundanten Code**, indem Code **wiederverwendbar** gemacht wird und andererseits erlaubt es z.B. Codestücke nach ihren Aufgaben zu **abstrahieren**, den Codestücken wird eine **Aufgabe zugeteilt**, sie werden zu **Unterprogrammen** gemacht und fortan über einen **Bezeichner aufgerufen**, was den Code deutlich **überschaubarer** macht. da man die Aufgabe eines Codestücks nun nur noch mit seinem **Bezeichner assozieren** muss.*[a]

[14] Werden später im Kapitel **??** genauer erklärt.

[a] Thiemann, „Einführung in die Programmierung“.

In $L_C$ ist die Bestimmung des **Datentyps** einer Variable etwas **komplizierter** als in manch anderen Programmiersprachen. Der Grund liegt darin, dass die eckigen `[<i>]`-Klammern zur Festlegung der **Mächtigkeit** eines Feldes **hinter** der **Variable** stehen: `<remaining-datatype><var>[<i>]`, während andere Programmiersprachen die eckigen `[<i>]`-Klammern **vor** die Variable schreiben `<remaining-datatype>[<i>]<var>`.

Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, ist es **schwieriger** den **Datentyp abzulesen**, als auch ein **Programm zu implementieren** was diesen erkennt. Damit ein Programmierer den **Datentyp ablesen** kann, kann dieser die **Spiralregel** verwenden, die unter der Webseite *Clockwise/Spiral Rule* nachgelesen werden kann. Werden die eckigen `[<i>]`-Klammern **hinter** die Variable geschrieben, wirken diese zum verwechseln ähnlich zum `<var>[<i>]`-Operator für den **Zugriff auf den Index eines Feldes**. Wenn ein Ausdruck geschrieben wird, wie `int ar[1] = {42}` wird, ist dieser vom Ausdruck `var[0] = 42` nur durch den **Kontext** um `var[1]` bzw. `var[0]` rum zu unterscheiden.

In Code 1.1 ist ein Beispiel zu sehen, indem die Variable `complex_var` den **Datentyp „Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2 von Zeigern auf Felder der Mächtigkeit 2 von Verbunden vom Typ `st`“** hat. Ein Vorteil die eckigen `[<i>]`-Klammern **hinter** die Variable zu schreiben ist in der **markierten Zeile** in Code 1.1 zu sehen. Will man auf ein **Element dieses Datentyps** zugreifen `(*complex_var[0][1])[1].attr`, so ist der Ausdruck fast genau **gleich aufgebaut**, wie der Ausdruck für den **Datentyp** `struct st (*complex_var[1][2])[2]`. Die **Ausgabe** des Beispiels in Code 1.1 ist in Code 1.2 zu sehen.

```
1 struct st {int attr;};
2
3 void main() {
4   struct st st_var[2] = {{.attr=314}, {.attr=42}};
5   struct st (*complex_var[1][2])[2] = {{&st_var, &st_var}};
6   print((*complex_var[0][1])[1].attr);
7 }
```

**Code 1.1:** *Beispiel für Spiralregel.*

```
1 42
```

**Code 1.2:** *Ausgabe von Beispiel für Spiralregel.*

In $L_C$ ist die **Ausführbarkeit einer Operation** oder **wie** diese Operation ausgeführt wird davon abhängig, was für einen **Datentyp** die Variable in diesem **Kontext** der Operation hat. In dem Beispiel in Code 1.3 wird in Zeile 2 ein **„Feld der Mächtigkeit 1 von Feldern der Mächtigkeit 2“** und Zeile 3 ein **„Zeiger auf Felder der Mächtigkeit 2“** erstellt. In den **markierten Zeilen** wird zweimal in Folge die **gleiche Operation** `<var>[0][1]` ausgeführt, allerdings hat die Operation aufgrund der **unterschiedlichen Datentypen** der Variablen einen **unterschiedlichen Effekt**.

In Zeile 4 wird ein **normaler Zugriff** auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt und in Zeile 5 wird allerdings erst dem **Zeiger** `int (*pntr)[2] = &ar[0];` **gefolgt** und dann ein Zugriff auf den **zweiten Eintrag** im **ersten Eintrag** des Felds `int ar[1][2] = {{314, 42}}` durchgeführt. **Beide Operationen** haben, wie in Code 1.4 zu sehen ist die **gleiche Ausgabe**.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(ar[0][1]);
5   print(pntr[0][1]);
6 }
```

**Code 1.3:** *Beispiel für unterschiedliche Ausführung.*

```
1 42 42
```

**Code 1.4:** *Ausgabe des Beispiels für unterschiedliche Ausführung.*

Eine weitere interessante Eigenheit, die tätsächlich nur in der **Untermenge** von $L_C$, die $L_{PicoC}$ darstellt gültig ist[15], ist dass die Operationen `<var>[0][1]` und `*(*(<var>+0)+1)` aus Code 1.3 und Code 1.5 komplett **austauschbar** sind. Die Ausgabe in Code 1.4 ist folglich **identisch** zur Ausgabe in Code 1.6.

```
1 void main() {
2   int ar[1][2] = {{314, 42}};
3   int (*pntr)[2] = &ar[0];
4   print(*(*(ar+0)+1));
5   print(*(*(pntr+0)+1));
6 }
```

**Code 1.5:** *Beispiel mit Dereferenzierungsoperator.*

```
1 42 42
```

**Code 1.6:** *Ausgabe des Beispiels mit Dereferenzierungsoperator.*

In der Programmiersprache $L_{PicoC}$ werden alle **Argumente** bei einem Funktionsaufruf nach der **Call By Value**-Strategie (Definition 1.4) übergeben. Ein Beispiel hierfür ist in Code 1.7 zu sehen. Hierbei wird ein **Verbund** `struct st copyable_ar = {.ar={314, 314}};`[16] an die Funktion `fun` übergeben. Hierzu wird der **Verbund** in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopiert** und an den Parameter `fun` gebunden.

Wie an der Ausgabe in Code 1.7 zu sehen ist hat die **Zuweisung** `copyable_ar.ar[1] = 42` an den Parameter `struct st copyable_ar` in der **aufgerufenen** Funktion `fun` keinen Einfluss auf die übergebene **lokale Variable** `copyable_ar` der **aufrufenden** Funktion. Der Eintrag an Index 1 im Feld bleibt bei `314`.

[15] In der Sprache $L_C$ gibt es einen Unterschied bei der Initialisierung bei z.B. `int *var = "string"` und z.B. `int var[1] = "string"`, der allerdings nichts mit den beiden Operatoren zu tuen hat, sondern mit der **Initialisierung**, bei der die Sprache $L_C$ verwirrenderweise die eckigen Klammern `[]` genauso, wie beim **Operator für den Zugriff auf einen Arrayindex**, vor den Bezeichner schreibt (z.B. `var[1]`), obwohl es ein **Derived Datatype** ist.

[16] **Später** wird darauf eingegangen, warum der Verbund den **Bezeichner** `copyable_ar` erhalten hat.

**Definition 1.4: Call by Value**

*Es wird eine **Kopie** des **Ergebnisses eines Ausdrucks**, welcher ein **Argument eines Funktionsaufrufes** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.*

*Das hat zur Folge, dass bei **Übergabe** einer Variable als **Argument** an eine Funktion, diese Variable bei **Änderungen** am entsprechenden **Parameter** der **aufgerufenen** Funktion in der **aufrufenden** Funktion **unverändert** bleibt.*[a]

[a] Bast, „Programmieren in C“.

```
1  struct st {int ar[2];};
2
3  int fun(struct st copyable_ar) {
4    copyable_ar.ar[1] = 42;
5  }
6
7  void main() {
8    struct st copyable_ar = {.ar={314, 314}};
9    print(copyable_ar.ar[1]);
10   fun(copyable_ar);
11   print(copyable_ar.ar[1]);
12 }
```

**Code 1.7:** *Beispiel dafür, dass Struct kopiert wird.*

```
1 314 314
```

**Code 1.8:** *Ausgabe von Beispiel, dass Struct kopiert wird.*

In der Programmiersprache $L_{PicoC}$ gibt es **kein Call by Reference** (Definition 1.5), allerdings kann der **Effekt** von Call by Reference mittels **Zeigern simuliert** werden, wie es in Code 1.11 bei der Funktion `fun_declared_before` und dem Parameter `int *param` zu sehen ist. Genau dieser **Trick** wird bei **Feldern** verwendet, um **nicht** das gesamte Feld bei einem Funktionsaufruf in den **Stackframe** der **aufgerufenen** Funktion `fun` **kopieren** zu müssen.

Wie im Beispiel in Code 1.9 zu sehen ist, wird in der markierten Zeile ein Feld `int ar[2] = {314, 314}` an die Funktion `fun` übergeben. Wie in der **Ausgabe** in Code 1.10 zu sehen ist, hat sich der Eintrag an Index 1 im Feld nach dem **Funktionsuaufruf** zu `42` geändert. Wird ein Feld direkt als Ausdruck `ar` ohne z.B. die eckigen `[]`-Klammern für einen **Indexzugriff** hingeschrieben wird die **Adresse** des Felds verwendet und **nicht** z.B. der **erste Eintrag** des Felds.

Eine Möglichkeit ein **Feld** als **Kopie** und **nicht** als **Referenz** zu übergeben ist es, wie in Code 1.7 das **Feld** als **Attribut** eines **Verbundes** zu übergeben, wie bei der Variable `copyable_ar`.

**Definition 1.5: Call by Reference**

*Es wird eine **implizite Referenz** einer Variable, welche ein **Argument eines Funktionsaufrufes** darstellt an den entsprechenden **Parameter** der **aufgerufenen** Funktion gebunden.*

***Implizit** meint hier, dass der Benutzer einer Programmiersprache mit Call by Reference **nicht mitbekommt**, dass er das **Argument selbst verändert** und **keine lokale Kopie des Arguments**.*[a]

[a] Bast, „Programmieren in C“.

```
1  int fun(int ar[2]) {
2    ar[1] = 42;
3  }
4
5  void main() {
6    int ar[2] = {314, 314};
7    print(ar[1]);
8    fun(ar);
9    print(ar[1]);
10 }
```

**Code 1.9:** *Beispiel dafür, dass Zeiger auf Feld übergeben wird.*

```
1 314 42
```

**Code 1.10:** *Ausgabe von Beispiel dafür, dass Zeiger auf Feld übergeben wird.*

Ein Programm in der Programmiersprache $L_{PicoC}$ wird von **oben-nach-unten** ausgewertet. Ein Problem tritt auf, wenn z.B. eine Funktion verwendet werden soll, die aber erst **unter** dem entsprechenden Funktionsaufruf **definiert** (Definition 1.8) wird. Es ist wichtig, dass der **Prototyp** (Definition 1.6) einer Funktion vorher durch die **Funktionsdefinition** bekannt ist, damit überprüft werden kann, ob die beim Funktionsaufruf **übergebenen Argumente** den gleichen **Datentyp** haben, wie die **Parameter** des **Prototyps** und ob die **Anzahl Argumente** mit der **Anzahl Parameter** des **Prototyps** übereinstimmt.

Allerdings lassen sich die Funktionen **nicht** immer so anordnen, dass jede in einem Funktionsaufruf referenzierte Funktion vorher definiert sein kann. Aus diesem Grund ist es möglich den **Prototyp** einer Funktion vorher zu **deklarieren** (Definition 1.7), wie es in den **markierten Zeile** im Beispiel in Code 1.11 zu sehen ist. Die **Ausgabe** des Beispiels ist in Code 1.12 zu sehen.

**Definition 1.6: Funktionsprototyp**

***Deklaration** einer Funktion, welche den **Funktionsbezeichner**, die **Datentypen** der einzelnen **Funktionsparameter**, die **Parametereihenfolge** und den **Rückgabewert** einer Funktion spezifiziert. Es ist **nicht** möglich zwei Funktiondeklarationen mit dem **gleichen** Funktionsbezeichner zu haben.*[a][b]

[a] Der **Funktionsprototyp** ist von der **Funktionsignatur** zu unterschieden, die in Programmiersprache wie `C++` und `Java` für die **Auflösung** von **Überladung** bei z.B. **Methoden** verwendet wird und sich in manchen Sprachen für den **Rückgabewert** interessiert und in manchen nicht, je nach Umsetzung. In solchen Sprachen ist es möglich mehrere **Methoden** oder **Funktionen** mit dem **gleichen** Bezeichner zu haben, solange sie sich durch die **Datentpyen** von **Parametern**, die **Parameterreihenfolge**, manchmal auch **Sichtbarkeitsbereiche** und **Klassentpyen** usw. unterschieden.

[b] *What is the difference between function prototype and function signature?*

**Definition 1.7: Deklaration**

*Der **Datentyp** bzw. **Prototyp** einer **Variablen** bzw. **Funktion**, sowie der **Bezeichner** dieser **Variable** bzw. **Funktion** wird dem **Compiler** mitgeteilt.*[abc]

[a] Über das **Schlüsselwort** `extern` lassen sich in der Programiersprache $L_C$ Veriablen **deklarieren**, **ohne** sie zu **definieren**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. D. P. Scholl, „Einführung in Embedded Systems“.

**Definition 1.8: Definition**

*Dem **Compiler** wird mitgeteilt, dass zu einem **bestimmten Zeitpunkt** in der Programmausführung **Speicher** für eine **Variable angelegt** werden soll und **wo**[a] dieser angelegt werden soll. Eine Funktion ist definiert ihr eine **relative Anfangsadresse** im **Hauptspeicher** zugewiesen werden kann, aber welcher die **Maschinenbefehle** für diese Funktion abgespeichert werden können.*[bc]

[a] Im Fall des PicoC-Compilers in den **Globalen Statischen Daten** oder auf dem **Stack**.
[b] *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*.
[c] P. D. P. Scholl, „Einführung in Embedded Systems“.

```
1  void fun_declared_before(int *param);
2
3  int fun_defined(int param) {
4    return param + 10;
5  }
6
7  void main() {
8    int res = fun_defined(22);
9    fun_declared_before(&res);
10   print(res);
11 }
12
13 void fun_declared_before(int *param) {
14   *param = *param + 10;
15 }
```

**Code 1.11:** *Beispiel für Deklaration und Definition.*

```
1 42
```

**Code 1.12:** *Ausgabe von Beispiel für Deklaration und Definition.*

In $L_{PicoC}$ lässt sich eine **definierte Variable** nur innerhalb ihres **Sichtbarkeitsbereichs** (Definition 1.9) verwenden. **Lokale Variablen** und **Parameter** lassen sich nur innerhalb der **Funktion** in welcher sie deklariert bzw. definiert wurden verwenden. Der **Sichtbarkeitsbereich** von **Lokalen Variablen** und **Parametern** erstreckt sich herbei von der **öffnenden** {-Klammer bis zur **schließenden** }-Klammer der **Funktionsdefinition**, in welcher sie definiert wurden.

Verschiedene **Sichtbarkeitsbereiche** können dabei **identische** Bezeichner besitzen. Im Beispiel in Code 1.13 kommt der markierte **Bezeichner** `local_var` in 2 verschiedenen **Sichtbarkeitsbereichen** vor, doch bezeichnet er 2 **unterschiedliche Variablen**. Der **Parameter** `param` und die **Lokale Variable** `local_var` dürfen **nicht**

den **gleichen Bezeichner** haben, da sie sich im gleichen **Sichtbarkeitsbereich** der Funktion `fun_scope` befinden. Die **Ausgabe** des Beispiels in Code 1.13 ist in Code 1.14 zu sehen.

**Definition 1.9: Sichtbarkeitsbereich (bzw. engl. Scope)**

***Bereich** in einem Programm, in dem eine Variable **sichtbar** ist und **verwendet** werden kann.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

```
1  int fun_scope(int param) {
2    int local_var = 2;
3    print(param);
4    print(local_var);
5  }
6
7  void main() {
8    int local_var = 4;
9    fun_scope(local_var);
10 }
```

**Code 1.13:** *Beispiel für Sichtbarkeitsbereichs.*

```
1 4 2
```

**Code 1.14:** *Ausgabe von Beispiel für Sichtbarkeitsbereichs.*

## 1.4 Gesetzte Schwerpunkte

Ein **Schwerpunkt** dieser Bachelorarbeit ist es in **erster Linie** bei der Kompilierung der Programmiersprache $L_{PicoC}$ in die Maschinensprache $L_{RETI}$ die **Syntax** und **Semantik** der Sprache $L_C$ identisch nachzuahmen. Der **PicoC-Compiler** soll die Sprache $L_{PicoC}$ im Vergleich zu z.B. dem **GCC**[17] ohne merklichen Unterschied[18] komplieren können.

In **zweiter Linie** soll dabei möglichst immer so Vorgegangen werden, wie es die **RETI-Codeschnipsel** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ vorgeben. Allerdings sollten diese bei **Inkonsistenzen** bezüglich der durch sie selbst vorgegebenen **Paradigmen** und anderen **Umstimmigkeiten** angepasst werden, da der **erstere Schwerpunkt** überwiegt.

Des Weiteren ist die **Laufzeit** bei Compilern zwar vor allem in der Industrie **nicht unwichtig**, aber bei **Compilern**, verglichen mit **Interpretern** weniger zu gewichten, da ein Compiler bei einem fertig implementierten Programm nur **einmal** Maschinencode generieren muss und dieser Maschinencode danach fortan ausgeführt wird. Beim einem **Compiler** ist daher eher zu priorisieren, dass der kompilierte **Maschinencode** möglichst **effizient** ist.

[17] Da die Sprache $L_{PicoC}$ eine **Untermenge** von $L_C$ ist, kann der **GCC** $L_{PicoC}$ ebenfalls kompilieren, allerdings **nicht** in die gewünschte Maschinensprache $L_{RETI}$.

[18] Natürlich mit **Ausnahme** der sich unterscheidenden **Maschinensprachen** zu welchen kompiliert wird und der unterschiedlichen **Commandline-Optionen** und **Fehlermeldungen**.

## 1.5 Über diese Arbeit

Der Quellcode des **PicoC-Compilers** ist **öffentlich** unter Link[19] zu finden. In der Datei `README.md` (siehe Abbildung 1.4) ist unter **„Getting Started“** ein kleines **Einführungstutorial** verlinkt. Unter **„Usage“** ist eine **Dokumentation** über die verschiedenen **Command-line Optionen** und verschiedene **Funktionalitäten der Shell** verlinkt. Deneben finden sich noch weitere Links zu möglicherweise interessanten Dokumenten. Der **letzte Commit** vor der Abgabe der **Bachelorarbeit** ist unter Link[20] zu finden.

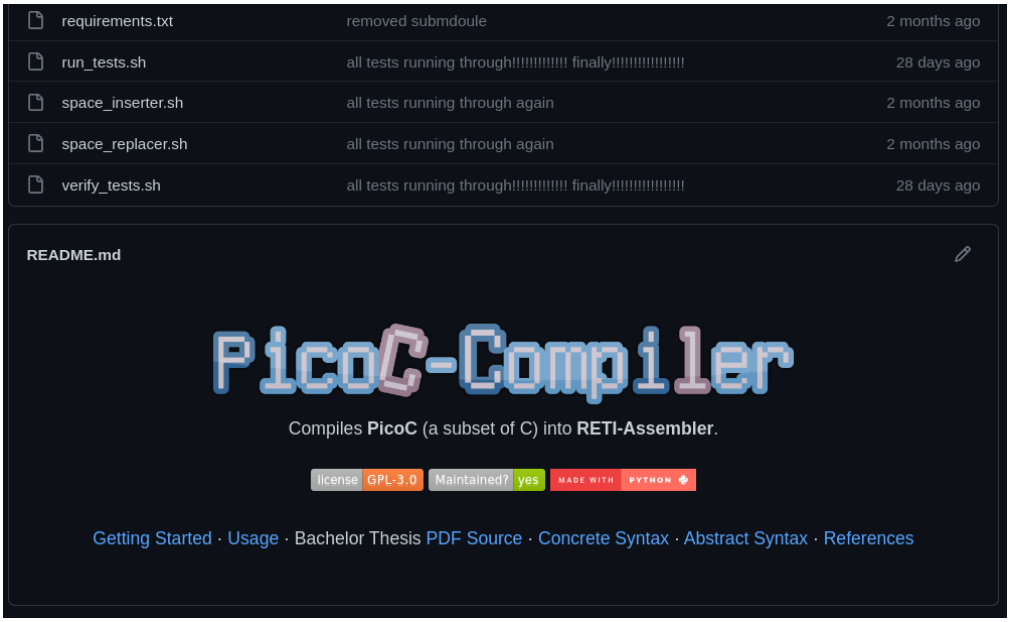



**Abbildung 1.4:** *README.md im Github Repository der Bachelorarbeit.*

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit wurde ebenfalls **veröffentlicht**, falls Studenten, die den **PicoC-Compiler** in Zukunft nutzen sich in der Tiefe dafür interessieren, wie dieser unter der Haube funktioniert. Die **Schrifftliche Ausarbeitung** dieser Bachelorarbeit ist als **PDF** unter Link[21] zu finden. Die **PDF** der Schrifftliche Ausarbeitung der Bachleorararbeit wird aus dem **Latexquellcode**, welcher unter Link[22] veröffentlicht ist automatisch mithife der **Github Action** Nemec, *copy_file_to_another_repo_action* und der **Makefile** Ueda, *Makefile for LaTeX* generiert.

Alle verwendeten **Latex Bibltiotheken** sind unter Link[23] zu finden[24]. Die Grafiken, die nicht mittels der Tikz Bibltiothek in Latex erstellt wurden, wurden mithilfe des Vectorgraphikeditors **Inkscape**[25] erstellt. Falls Interesse besteht **Grafiken** aus der Schrifftlichen Ausarbeitung der Bachelorarbeit zu verwenden, so sind diese zusammen mit den `.svg`-Dateien von **Inkscape** im Ordner `/figures` zu finden.

Alle weitere **verwendete Software**, wie verwendete **Python Bibltiotheken**, **Vim/Neovim Plugins**, **Tmux Plugins** usw. sind in der `README.md` unter **„References“** bzw. direkt unter Link[26] zu finden.

[19] https://github.com/matthejue/PicoC-Compiler.
[20] https://github.com/matthejue/PicoC-Compiler/tree/bcafedffa9ff3075372554b14f1a1d369af68971.
[21] https://github.com/matthejue/Bachelorarbeit_out/blob/main/Main.pdf.
[22] https://github.com/matthejue/Bachelorarbeit.
[23] https://github.com/matthejue/Bachelorarbeit/blob/master/content/Packete_und_Deklarationen.tex.
[24] Jede einzelne verwendete Latex **Bibliothek** einzeln anzugeben wäre allerdings etwas zu aufwendig.
[25] Developers, *Draw Freely — Inkscape*.
[26] https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/references.md.

Um die verschiedenen **Aspekte** dieser Schrifftlichen Ausarbeitung der Bachelorarbeit besser erklären zu können, werden **Codebeispiele** verwendet. In diesem Kapitel Motivation werden Codebeispiele zur **Anschauung** verwendet und mithilfe des in den **PicoC-Compiler** integrierten **RETI-Interpreters Ausgaben** erzeugt, die in dieses Dokument **eingelesen** wurden. In Kapitel **??** werden kleine repräsentative **PicoC-Programme** in wichtigen **Zwischenstadien der Kompilierung** gezeigt[27].

Die **Codebeispiele** wurden alle mit dem **PicoC-Compiler** kompiliert und danach **nicht** mehr **verändert**, also genauso, wie der **PicoC-Compiler** sie kompiliert aus den Dateien in dieses Dokument eingelesen. Alle hier zur Repräsentation verwendeten **PicoC-Programme** lassen sich unter dem Link[28] finden und mithilfe der im Ordner `/code_examples` beiliegenden `/Makefile` und dem Befehl `> make compile-all` genauso **kompilieren**, wie sie hier dargestellt sind[29].

### 1.5.1 Still der Schrifftlichen Ausarbeitung

In dieser **Schrifftliche Ausarbeitung der Bachelorarbeit** sind die manche **Wörter** für einen besseren Lesefluss **hervorgehoben**. Es ist so gedacht, dass die **Hervorgehobenen Wörter** beim Lesen sichtbare **Ankerpunkte** darstellen an denen sich **orientiert** werden kann, aber auch damit der **Inhalt** eines vorher gelesener **Paragraphs** nochmal durch Überfliegen der Hervorgehobenen Wörter in **Erinnerung gerufen** werden kann.

Bei den **Erklärungen** wurden darauf geachtet bei jeder der verwendeten **Methodiken** und jeder **Designentscheidung** die Frage zu klären, „**warum** etwas geanu so gemacht wurde und nicht anders“, denn wie es im Buch LeFever, *The Art of Explanation* auf eine deutlich ausführlichere Weise dargelegt wird, ist einer der **zentralen Fragen**, die ein Leser in erster Linie zum **wirklichen Verständnis** eines Themas beantwortet braucht[30] die Frage des **„warum“**.

Zum **Verweis auf Quellen** an denen sich z.B. bei der Formulierung von **Definitionen** orientiert wurde, wurden um den **Lesefluss** nicht zu stören **Fußnoten**[31] verwendet. Die meisten Definitionen wurden in **eigenen Worten** formuliert, damit die Definitionen selbst zueinander **konsistent** sind, wie auch das in Ihnen verwendete **Vokabular**. Wurde eine Definition **wörtlich** aus einer Quelle übernomnen, so wurde die Definition oder der entsprechende Teil in **„Anführungszeichen“** gesetzt. Beim Verweis auf Quellen **außerhalb** einer **Definitionsbox**, wurde allerdings meistens, sofern die **Quelle** wirklich **relevant** war auf das **Zitieren über Fußnoten** verzichtet.

### 1.5.2 Aufbau der Schrifftlichen Arbeit

Die **Schrifftliche Ausarbeitung** der Bachelorarbeit ist in 4 Kapitel unterteilt: Motivation, Einführung, **??** und **??**.

Im momentanen Kapitel Motivation, wurde ein kurzer **Einstieg** in das Thema **Compilerbau** gegeben und die **zentrale Aufgabenstellung** der Bachelorarbeit erläutert, sowie auf **Schwerpunkte** und kleinere **Teilprobleme**, die eines **besonderen Fokus** bedürfen eingegangen.

Im Kapitel Einführung werden die notwendigen **Theoretischen Grundlagen** eingeführt, die zum Verständnis des Kapitels **Implementierung** notwendig sind. Das Kapitel soll darüberhinaus aber auch einen **Überblick** über das Thema Compilerbau geben, sodass nicht nur ein Grundverständnis für das eine **spezifische**

[27] Also die verschiedenen in den **Passes** generierten **Abstrakten Syntaxbäume**, sofern der **Pass** für den gezeigten Aspekt relevant ist.
[28] `https://github.com/matthejue/Bachelorarbeit/tree/master/code_examples`.
[29] Es wurde zu diesem Zweck die **Command-line Option** `-t`, `--thesis` erstellt, die bestimmte Kommentare **herausfiltert**, damit die generierten **Abstrakten Syntaxbäume** in den verschiedenen Zwischenstufen der Kompilierung **nicht** zu **überfüllt** mit Kommentaren sind.
[30] Vor allem **Anfang**, wo der Leser **wenig** über das Thema **weiß**.
[31] Das ist ein **Beispiel** für eine **Fußnote**.

**Vorgehen**, welches zur Implementierung des **PicoC-Compiler** verwendet wurde vermittelt wird, sondern auch ein **Vergleich** zu **anderen Vorgehensweisen** möglich ist. Die **Theoretischen Grundlagen** umfassen die wichtigsten Definitionen in Bezug zu Compilern und den verschiedenen **Phasen der Kompilierung**, welche durch die Unterkapitel **Lexikalische Analyse**, **Syntaktische Analyse** und **Code Generierung** repräsentiert sind.

Des Weiteren wurden für **T-Diagramme** und **Formale Sprachen** eigene Unterkapitel erstellt. Für **T-Diagramme** wurde ein eigenes Unterkapitel erstellt, da sie häufig in der Schrifftlichen Ausarbeitung verwendet werden und die **T-Diagramm Notation** nicht allgemein bekannt ist. Für **Formale Sprachen** wurde ein eigenes Unterkapitel erstellt, da für den Gutachter Prof. Dr. Scholl das Thema **Formale Sprachen** eher **fachfremd** ist, aber dieses Thema einige zentrale und wichtige Fachbegriffe besitzt, bei denen es wichtig ist die genaue **Definition** zu haben. Generell wurde im Kapitel **Einführung** versucht an Erklärungen nicht zu sparren, damit aufgrund dessen, dass das Thema eher fachfremd für Prof. Dr. Scholl ist für das Kapitel **Implementierung** keine wichtigen Aspekte unverständlich bleiben.

Im Kapitel **??** werden die einzelnen Aspekte der Implementierung des **PicoC-Compilers**, unterteilt in die verschiedenen **Phasen der Kompilierung** nach dennen das Kapitel **Einführung** ebenfalls unterteilt ist erklärt. Dadurch, dass Kapitel **Implementierung** und Kapitel **Einführung** eine **ähliche Kapiteleinteilung** haben, ist es besonders einfach zwischen beiden hin und her zu wechseln. Wenn z.B. eine Definition im Kapitel **Einführung** gesucht wird, die zum Verständis eines Aspekts in Kapitel **Implemenentierung** notwendig ist, so kann aufgrund der ähnlichen **Kapiteleinteilung** die entsprechende Definition analog im Kapitel **Einleitung** gefunden werden.

Im Kapitel **??** wird ein **Überblick** über die **wichtigsten Funktionalitäten** des PicoC-Compilers gegeben, indem anhand **kleiner Anleitungen** gezeigt wird wie man diese verwendet. Des Weiteren wird darauf eingegangen, wie die **Qualitätsicherung** für den **PicoC-Compiler** umgesetzt wurde, also wie gewährleistet wird, dass der **PicoC-Compiler** funktioniert. Zum Schluss wird noch auf **weitere Erweiterungsideen** eingegangen, die auch interessant zu implementieren wären.

Im Kapitel Appendix werden einige **Definitionen** und **Themen** angesprochen, die bei **Interesse** zur **weiteren Vertiefung** da sind und **unabhänging** von den anderen Kapiteln sind. Diese **Themen** und **Defintionen** sind dazu da den Bogen von der **spezifischen** Implementierung des **PicoC-Compilers** wieder zum **allgemeinen Vorgehen** bei der Implementierung eines Compilers zu schlagen. Diese **Themen** und **Definitionen** passen nicht ins Kapitel **??**, da diese selbst **nichts** mit der Implementierung des **PicoC-Compilers** zu tuen haben und auch nichts ins **Kapitel** Einführung, da dieses nur **Theoretische Grundlagen** erklärt, die für das Kapitel **Implementierung** wichtig sind.

Generell wurde in der Schrifftlichen Ausarbeitung immer versucht **Parallelen** zu Implementierung **echter** Compiler zu ziehen. Der Zweck des **PicoC-Compilers** ist es primär ein **Lerntool** zu sein, weshalb Methoden, wie **Liveness Analyse** (Definition 3.9) usw., die in **echten** Compilern zur Anwendung kommen **nicht umgesetzt** wurden, da sich an die **vorgegebenen Paradigmen** aus der Vorlesung P. D. C. Scholl, „Betriebssysteme“ gehalten werden sollte.

# 2 Einführung

## 2.1 Compiler und Interpreter

Die wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** $L_{PicoC}$ in die **RETI-Sprache** $L_{RETI}$ das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[1] und von **Tests** die **Beziehungen** in 2.2.1 zu belegen (siehe Subkapitel **??**).

**Definition 2.1: Interpreter**

***Interpretiert*** *die* ***Befehle***[a] *oder* ***Statements*** *eines Programmes P direkt.*

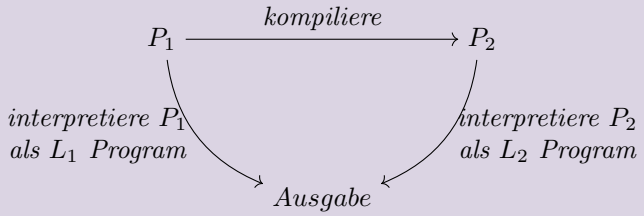
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen* ***Sub-Bäumen*** *des* ***Abstrakten Syntaxbaumes*** *(wird später eingeführt unter Definition 2.44) und führt je nach Komposition der* ***Knoten*** *des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[b]

[a] **Maschinensprache** kann **genauso** interpretiert werden, wie auch eine **Programmiersprache**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.2: Compiler**

***Kompiliert*** *ein* ***beliebiges*** *Program $P_1$, welches in einer Sprache $L_1$ geschrieben ist, in ein Program $P_2$, welches in einer Sprache $L_2$ geschrieben ist.*

*Wobei* ***Kompilieren*** *meint, dass ein beliebiges Program $P_1$ in der Sprache $L_1$ so in die Sprache $L_2$ zu einem Programm $P_2$ übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen $L_1$ und $L_2$* ***interpretiert*** *werden, sie die gleiche* ***Ausgabe*** *haben, wie es in Diagramm 2.2.1 dargestellt ist. Also beide Programme $P_1$ und $P_2$ die gleiche* ***Semantik*** *(Definition 2.15) haben und sich nur* ***syntaktisch*** *(Definition 2.14) durch die Sprachen $L_1$ und $L_2$, in denen sie geschrieben stehen unterscheiden.*[a]




(2.2.1)

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

[1] Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **G**NU **C**ompiler **C**ollection

**Anmerkung**

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C^{o\_j}_{i\_w\_k\_min}$ geschrieben, wobei $C_w$ die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschinensprache $L_{B_i}$ einer Maschine $M_i$ kompiliert. Falls die Notwendigkeit besteht, die **Maschine** $M_i$ anzugeben, zu dessen **Maschinensprache** $L_{B_i}$ der Compiler kompiliert, wird das als $C_i$ geschrieben. Falls die Notwendigkeit besteht die **Sprache** $L_o$ anzugeben, in der der Compiler selbst geschrieben ist, wird das als $C^o$ geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert ($L_{w\_k}$) oder in der er selbst geschrieben ist ($L_{o\_j}$) anzugeben, wird das als $C^{o\_j}_{w\_k}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 3.18) kann man das als $C_{min}$ schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, das in einer **Programmiersprache** geschrieben ist zu **Maschinencode**, der in **Maschinensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 3.5) oder **Cross-Compiler** (Definition 2.5). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 3.1) voneinander zu unterscheiden.

**Definition 2.3: Maschinensprache**

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauergesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen konstanten** Anzahl von Fetch- und Execute Phasen im **Komplexeren Fall**. Die Maschinenbefehle sind meist so entworfen, dass sie sich innerhalb bestimmter **Wortbreiten**, die **Zweierpotenzen** sind kodieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.*[a] [b]

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.4) haben.

[b] P. D. C. Scholl, „Betriebssysteme“.

Der **Maschinencode**, den ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschinenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschine, die binär arbeitet verständlich sein sollen und nicht für den Programmierer.

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschinencode, der die Maschinenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.4) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, den der PicoC-Compiler generiert, in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung[2].

**Definition 2.4: Immediate**

*__Konstanter Wert__, der als __Teil__ eines __Maschinenbefehls__ gespeichert ist und dessen __Wertebereich__ dementsprechend auch durch die die Anzahl an Bits, die ihm innerhalb dieses __Maschinenbefehls__ zur Verfügung gestellt sind beschränkt ist. Der Wertebereich ist __beschränkter__ als bei sonstigen Werten*

[2] Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschinencode in z.B. **UTF-8 Kodierung** ausführen kann, wäre dagegen unnötig kompliziert und aufwändig, da Hardware **binär** arbeitet und man dieser daher lieber direkt die binär kodierten Maschinenbefehle übergibt, anstatt z.B. eine unnötig **platzverbrauchenden** UTF-8 Codierung zu verwenden, die nur in sehr vielen Schritten einen Befehl verarbeiten kann, da die Register und Speicherzellen des Hauptspeichers üblicherweise nur **32- bzw. 64-Bit Breite** haben.

*innerhalb des Hauptspeichers, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.*[a]

[a] Ljohhuh, *What is an immediate value?*

**Definition 2.5: Cross-Compiler**

*Kompiliert auf einer **Maschine** $M_1$ ein Program, dass in einer **Sprache** $L_w$ geschrieben ist für eine **andere Maschine** $M_2$, wobei beide Maschinen $M_1$ und $M_2$ unterschiedliche **Maschinensprachen** $B_1$ und $B_2$ haben.*[a][b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** $C^{Python}_{PicoC}$.
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Ein **Cross-Compiler** ist entweder notwendig, wenn eine Zielmaschine $M_2$ nicht ausreichend **Rechenleistung** hat, um ein Programm in der Wunschsprache $L_w$ selbst **zeitnah** zu kompilieren oder wenn noch kein Compiler $C_w$ für die **Wunschsprache** $L_w$ und andere Programmiersprachen $L_o$, in denen man Programmieren wollen würde existiert, der unter der **Maschinensprache** $B_2$ einer Zielmaschine $M_2$ läuft.[3]

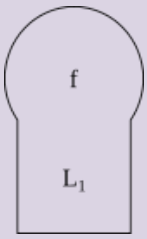
### 2.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus der Wissenschaftlichen Publikation J. Earley und Sturgis, „A formalism for translator interactions“ entnommen ist besonders gut, da diese optimal darauf **zugeschnitten** sind die Eigenheiten von Compilern in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Program (Definition 2.6), einen Übersetzer (Definition 2.7), einen Interpreter (Definition 2.8) und eine Maschine (Definition 2.9) zusammen.

**Definition 2.6: T-Diagram Programm**

*Repräsentiert ein **Programm**, dass in der **Sprache** $L_1$ geschrieben ist und die **Funktion** $f$ berechnet.*[a]




[a] J. Earley und Sturgis, „A formalism for translator interactions“.

**Anmerkung**

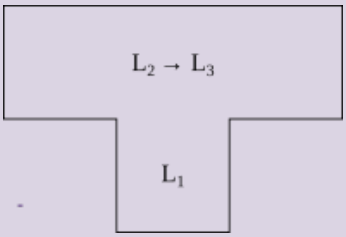
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein $L$ dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.6 also reichen einfach eine 1 hinzuschreiben.

[3] Die an vielen Universitäten und Schulen eingesetzen programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren Microcontroller eine **C-ähnliche Sprache** in die Maschinensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst nicht genug Rechenleistung besitzt, um ein Programm selbst zeitnah zu kompilieren.

**Definition 2.7: T-Diagram Übersetzer (bzw. eng. Translator)**

*Repräsentiert einen **Übersetzer**, der in der **Sprache** $L_1$ geschrieben ist und **Programme** von der **Sprache** $L_2$ in die **Sprache** $L_3$ kompiliert.*

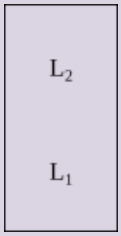
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 2.2.1.*[b]




[a] Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

**Definition 2.8: T-Diagram Interpreter**

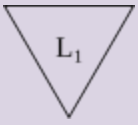
*Repräsentiert einen **Interpreter**, der in der **Sprache** $L_1$ geschrieben ist und **Programme** in der **Sprache** $L_2$ interpretiert.*[a]




[a] J. Earley und Sturgis, „A formalism for translator interactions“.

**Definition 2.9: T-Diagram Maschine**

*Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache** $L_1$ ausführt.*[a b]




[a] Wenn die Maschine **Programme** in einer höheren Sprache als **Maschinensprache** ausführt, ist es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschine**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).
[b] J. Earley und Sturgis, „A formalism for translator interactions“.

Aus den verschiedenen **Blöcken** lassen sich **Kompositionen** bilden, indem man sie **adjazent** zueinander platziert. Allgemein lässt sich grob sagen, dass **vertikale Adjazenz** für **Interpretation** und **horinzontale Adjazenz** für **Übersetzung** steht.

Sowohl **horinzontale** als auch **vertikale Adjazenz** lassen sich, wie man in den Abbildungen 2.1 und 2.2 erkennen kann zusammenfassen.

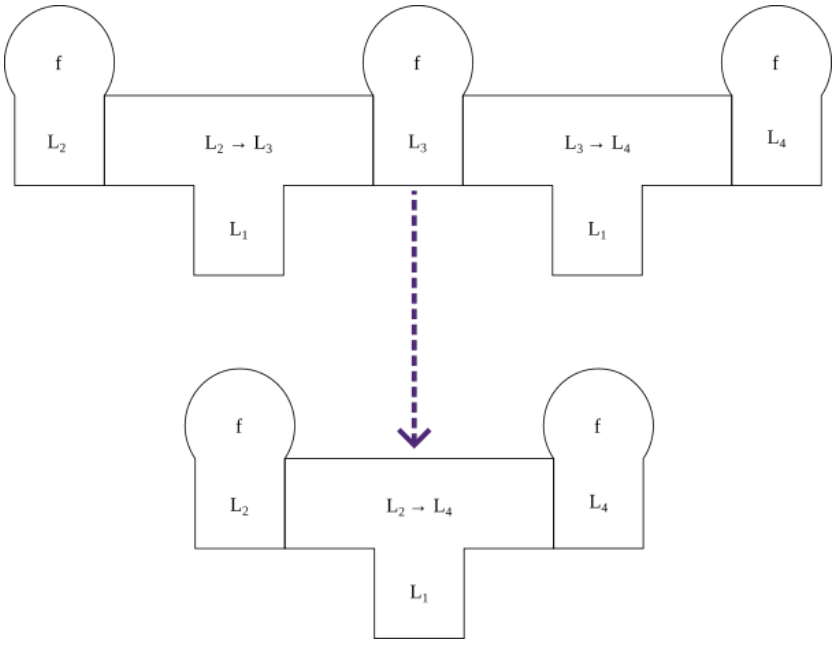



**Abbildung 2.1:** *Horinzontale Übersetzungszwischenschritte zusammenfassen.*

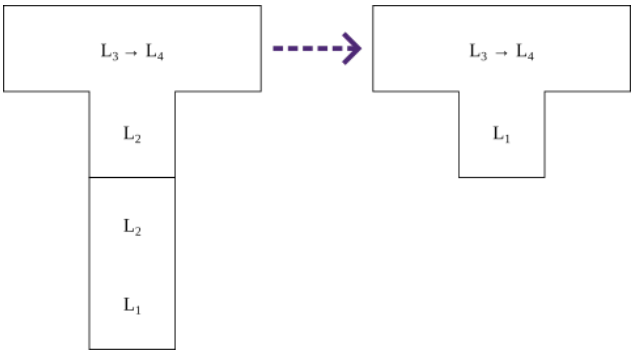



**Abbildung 2.2:** *Vertikale Interpretierungszwischenschritte zusammenfassen.*

## 2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.13) zu tuen, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der Sprache $L_1$ in eine Sprache $L_2$ ist. Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen**, was die Begriffe **Symbol** (Definiton 2.10), **Alphabet** (Definition 2.11), **Wort** (Definition 2.12) beinhaltet vorher eingeführt zu haben.

**Definition 2.10: Symbol**

*„Ein Symbol ist ein **Element** eines **Alphabets** $\Sigma$.“*[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.11: Alphabet**

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 2.10).“*[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.12: Wort**

*„Ein Wort $w = a_1...a_n \in \Sigma^*$ ist eine **endliche Folge** von **Symbolen** aus einem **Alphabet** $\Sigma$.*

*Es gibt es die **Konkatenation** $w_1w_2 = a_1 \dots a_nb_1 \dots b_n$ von **Wörtern** $w_1 = a_1 \dots a_n$ und $w_2 = b_1 \dots b_n$ und die **Länge** eines **Wortes** $|w|$.*

*Ein wichtiges Wort ist das **leere Wort** $\varepsilon$ für das gilt: $|\varepsilon| = 0$ und $\forall w \in \Sigma^* : \varepsilon w = w\varepsilon = w$. Es handelt sich bei $\varepsilon$ also um das **Neutrale Element** bei der **Konkatenation** von **Wörtern**.*

*Bei $\Sigma^*$ handelt es sich um **Kleenesche Hülle** eines **Alphabets** $\Sigma$, es ist die **Sprache aller Wörter**, welche durch beliebige **Konkatenation** von **Symbolen** aus dem **Alphabet** $\Sigma$ gebildet werden können, wobei $\varepsilon \in \Sigma^*$. Dies ist die **größte Sprache** über $\Sigma$ und **jede Sprache** über $\Sigma$ ist eine **Teilmenge** davon.“*[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.13: Formale Sprache**

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.12) über dem **Alphabet** $\Sigma$ (Definition 2.11).“*[a]

*Das Adjektiv **„formal“** kann dabei weggelassen werden, wenn der **Kontext** indem die **Sprache** verwendet wird **eindeutig** ist, da man das Adjektiv **„formal“** nur verwendet um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Sprache** herauszustellen.*

[a] Nebel, „Theoretische Informatik“.

Bei der Übersetzung eines Programmes von einer Sprache $L_1$ zur Sprache $L_2$ muss die **Semantik** (Definition 2.15) **gleich** bleiben. Beide Sprachen $L_1$ und $L_2$ haben eine **Grammatik** (Definition 2.16), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 2.14) haben.

**Definition 2.14: Syntax**

*Die Syntax bezeichnet alles was mit dem **Aufbau** von **Formalen Sprachen** zu tuen hat. Die **Grammatik** einer Sprache, aber auch die in **Natürlicher Sprache** ausgedrückten Regeln, welche den Aufbau von Wörtern einer Formalen Sprache beschreiben, beschreiben die Syntax dieser Sprache. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben*[a].[b]

[a] Z.B. die **Konkrete** und **Abstrakte Syntax**, die später eingeführt werden.
[b] Thiemann, „Einführung in die Programmierung“.

**Definition 2.15: Semantik**

*Die Semantik bezeichnet alles was mit der **Bedeutung** von **Formalen Sprachen** zu tuen hat.*[a]

[a] Thiemann, „Einführung in die Programmierung“.

**Definition 2.16: Formale Grammatik**

*„Eine Formale Grammatik beschriebt wie **Wörter** einer **Sprache** abgeleitet werden können.“*[a]

*Das Adjektiv **„formal“** kann dabei weggelassen werden, wenn der **Kontext** indem die **Grammatik** verwendet wird **eindeutig** ist, da man das Adjektiv **„formal“** nur verwendet um den Unterschied zum im normalen Sprachgebrauch verwendeten Begriff einer **Grammatik** herauszustellen.*

*Eine Grammatik wird durch das Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt, wobei:*

- $N \mathrel{\hat{=}}$ ***Nicht-Terminalsymbole**.*
- $\Sigma \mathrel{\hat{=}}$ ***Terminalsymbole**, wobei $N \cap \Sigma = \emptyset$*[b c]*.*
- $P \mathrel{\hat{=}}$ *Menge von **Produktionsregeln** $w \to v$, wobei $w, v \in (N \cup \Sigma)^* \wedge w \notin \Sigma^*$.*[d e]*.*
- $S \mathrel{\hat{=}}$ ***Startsymbol**, wobei $S \in N$.*

*Zusätzlich ist es praktisch **Nicht-Terminalsymbole** $N$, **Terminalsymbole** $\Sigma$ und das **leere Wort** $\varepsilon$ allgemein als Menge der **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ zu definieren.*

[a] Nebel, „Theoretische Informatik“.
[b] Weil mit ihnen **terminiert** wird.
[c] Kann auch als **Alphabet** bezeichnet werden.
[d] $w$ muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.
[e] Bzw. $w, v \in V^* \wedge w \notin \Sigma^*$.

Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 2.17) einteilen.

**Definition 2.17: Chromsky Hierarchie**

*Die Chromsky Hierarchie ist eine Hierarchie in der **Formale Sprachen** nach der **Komplexität** ihrer **Formalen Grammatiken** in verschiedene **Klassen** unterteilt werden. Jede dieser Klassen hat verschiedene **Eigenschaften**, wie **Entscheidungeprobleme**, die in dieser Klasse **entscheidbar** bzw. **unentscheidbar** sind usw.*

*Eine Sprache $L_i$ ist in der **Chromsky Hierarchie** vom Typ $i \in \{0, \ldots, 3\}$, falls sie von einer Grammatik dieses Typs $i$ erzeugt wird.*

*Zwischen den Sprachmengen **benachbarter Klassen** in Abbildung 2.17.1 besteht eine **echte Teilmengenbeziehung**: $L_3 \subset L_2 \subset L_1 \subset L_0$. Jede **Reguläre Sprache** ist auch eine **Kontextfreie Sprache**, aber nicht jede **Kontextfreie Sprache** ist auch eine **Reguläre Sprache**.*[a]


(2.17.1)

[a] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 2.18) und **Kontextfreien Grammatiken** (Definition 2.19) beschrieben werden.

**Definition 2.18: Reguläre Grammatik**

*"Ist eine Grammatik für die gilt, dass **alle Produktionen** eine der Formen:*

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{2.18.1}$$

*haben, wobei $A, B$ **Nicht-Terminalsymbole** sind und $c$ ein **Terminalsymbol** ist[a b].”[c]*

[a] Diese Definition einer **Regulären Grammatik** ist **rechtsregulär**, es ist auch möglich diese Definition **linksregulär** zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.
[b] Dadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.
[c] Nebel, „Theoretische Informatik“.

**Definition 2.19: Kontextfreie Grammatik**

*"Ist eine Grammatik für die gilt, dass **alle Produktionen** die Form:*

$$A \to v \tag{2.19.1}$$

*haben, wobei $A$ ein **Nicht-Terminalsymbol** ist und $v$ ein beliebige Folge von **Grammatiksymbolen**[a] ist.”[b]*

[a] Also eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.
[b] Nebel, „Theoretische Informatik“.

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des **Wortproblems** (Definition 2.20). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise immer **entscheidbar**. Wenn das Programm ein **Wort** der **Sprache** ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es **kein Wort** der **Sprache**, die der Compiler kompiliert, wird eine **Fehlermeldung** ausgegeben.

**Definition 2.20: Wortproblem**

*Ein Entscheidungeproblem, bei dem man zu einem* ***Wort*** $w \in \Sigma^*$ *und einer* ***Sprache*** $L$ *als* ***Eingabe*** 1 *oder* 0[a] ***ausgibt****, je nachdem, ob dieses Wort* $w$ *Teil der Sprache* $L$ *ist* $w \in L$ *oder nicht* $w \notin L$.[b]

*Das Wortproblem kann durch die folgende* ***Indikatorfunktion***[c] *zusammengefasst werden:*

$$\mathbb{1}_L : \Sigma^* \to \{0,1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \tag{2.20.1}$$

[a] Bzw. **„ja“** oder **„nein“** usw., es muss nicht umgedingt 1 oder 0 sein.
[b] Nebel, „Theoretische Informatik“.
[c] Auch **Charakteristische Funktion** genannt.

### 2.2.1 Ableitungen

Um sicher zu wissen, ob ein Compiler ein **Programm**[4] kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprache** des Compilers **abzuleiten**. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 2.21) und der normalen **Ableitungsrelation** (Definition 2.22) unterschieden.

**Definition 2.21: 1-Schritt-Ableitungsrelation**

*„Eine* ***binäre Relattion*** $\Rightarrow$ *zwischen Wörtern aus* $(N \cup \Sigma)^*$*, die alle möglichen Wörter* $(N \cup \Sigma)^*$ *in Relation zueinander setzt, die sich nur durch das* ***einmalige*** *Anwenden einer Produktionsregel voneinander unterschieden.*

*Es gilt* $u \Rightarrow v$ ***genau dann wenn*** $u = w_1 x w_2$, $v = w_1 y w_2$ ***und*** *es eine Regel* $x \to y \in P$ *gibt, wobei* $w_1, w_2, x, y \in (N \cup \Sigma)^*$ *“*[a]

[a] Nebel, „Theoretische Informatik“.

**Definition 2.22: Ableitungsrelation**

*„Eine* ***binäre Relation*** $\Rightarrow^*$*, welche der* ***reflexive, transitive Abschluss*** *der* ***1-Schritt-Ableitungsrelation*** $\Rightarrow$ *ist. Auf der* ***rechten*** *Seite der Ableitungsrelation* $\Rightarrow^*$ *steht also ein Wort aus* $(N \cup \Sigma)^*$*, welches durch* ***beliebig häufiges*** *Anwenden von Produktionsregeln entsteht.*

*Es gilt* $u \Rightarrow^* v$ ***genau dann wenn*** $u = w_1 \Rightarrow \ldots \Rightarrow w_n = v$*, wobei* $n \geq 1$ *und* $w_1, \ldots, w_n \in (N \cup \Sigma)^*$*. “*[a]

[a] Nebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**[5] kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 2.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 relevant.

**Definition 2.23: Links- und Rechtsableitungableitung**

*„In jedem* ***Ableitungsschritt*** *wird bei* ***Typ-3- und Typ-2-Grammatiken*** *auf das am* ***weitesten links (Linksableitung)*** *bzw.* ***rechts (Rechtsableitung)*** *stehende* ***Nicht-Terminalsymbol*** *eine Produktionsregel angewandt, bei Typ-1- und Typ-0-Grammatiken ist es statt einem* ***Nicht-Terminalsymbol***

[4] Bzw. **Wort**.
[5] Bzw. **Wort**.

*die **linke** Seite einer Produktion.*

*Mit diesem Vorgehen kann man jedes ableitbare Wort generieren, denn dieses Vorgehen entspricht **Tiefensuche** von **links-nach-rechts**.“*[a]

[a] Nebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 2.24) ist[6].

**Definition 2.24: Linksrekursive Grammatiken**

*Eine **Grammatik** ist **linksrekursiv**, wenn sie ein **Nicht-Terminalsymbol** enthält, dass **linksrekursiv** ist.*

*Ein **Nicht-Terminalsymbol** ist **linksrekursiv**, wenn das **linkeste Symbol** in einer seiner **Produktionen** es selbst ist oder zu sich selbst gemacht werden kann durch eine Folge von Ableitungen:*

$$A \Rightarrow^* Aa,$$

*wobei $a$ eine beliebige Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen** ist.*[a]

[a] *Parsing Expressions · Crafting Interpreters*.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (Definition 2.26) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 2.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 2.37) als eine **compilerinterne Datenstruktur** umgesetzt werden.

**Definition 2.25: Formaler Ableitungsbaum**

*Ist ein Baum, in dem die Syntax eines **Wortes**[a] nach den **Produktionen** der zugehörigen Grammatik, die angewendet werden mussten um das Wort abzuleiten **hierarchisch** zergliedert dargestellt wird.*

*Das Adjektiv **„formal“** kann dabei weggelassen werden, wenn der **Kontext** indem der **Ableitungsbaum** verwendet wird **eindeutig** ist, da man das Adjektiv **„formal“** nur verwendet um den Unterschied zum **compilerinternen Ableitungsbaum** herauszustellen, der den Formalen Ableitungsbaum als **Datentstruktur** zur einfachen Weiterverarbeitung umsetzt.*

*Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 2.16) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** $N$ und die **Blätter** sind entweder **Terminalsymbole** $\Sigma$ oder das **leere Wort** $\varepsilon$.*[b]

[a] Z.B. **Programmcode**.
[b] Nebel, „Theoretische Informatik“.

In Abbildung 2.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 2.25.1 nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition **??**) angegebenen **Produktionen** 2.1 einer ansonsten nicht näher spezifizierten **Grammatik** $G = \langle N, \Sigma, P, add \rangle$ ergibt.

[6] Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

| | | | |
|---|---|---|---|
| *DIG_NO_0* | ::= | "1" \| "2" \| "3" \| "4" \| "5" \| "6" | *L_Lex* |
| | | \| "7" \| "8" \| "9" | |
| *DIG_WITH_0* | ::= | "0" \| *DIG_NO_0* | |
| *NUM* | ::= | "0" \| *DIG_NO_0* *DIG_WITH_0*∗ | |
| *add* | ::= | *add* "+" *mul* \| *mul* | *L_Parse* |
| *mul* | ::= | *mul* "∗" *NUM* \| *NUM* | |

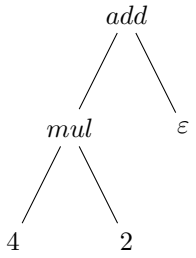
**Grammatik 2.1:** *Produktionen für Ableitungsbaum in EBNF*

**Anmerkung**

Werden die **Produktionen** einer Grammatik in z.B. **EBNF** angegeben, wie in Grammatik **??**, wird die Angabe dieser Produktionen auch oft als **Grammatik** bezeichnet, obwohl Grammatiken eigentlich durch ein Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt sind.

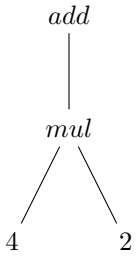
$$add \Rightarrow mul \Rightarrow mul\ "*"\ NUM \Rightarrow NUM\ "*"\ NUM \Rightarrow 4\ "*"\ NUM \Rightarrow 4\ "*"\ 2 \tag{2.25.1}$$

Bei Ableitungsbäumen gibt es **keine** einheutliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 2.25.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort** $\varepsilon$ hinzuzufügen.




(2.25.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 2.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[7].




(2.25.3)

Für einen Compiler ist es notwendig, dass die **Konkrete Grammatik** keine **Mehrdeutige Grammatik** (Definition 2.26) ist, denn sonst können unter anderem die **Präzedenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel **??** an einem Beispiel demonstriert wird.

[7] Diese Option wurde beim **PicoC-Compiler** gewählt.

**Definition 2.26: Mehrdeutige Grammatik**

*„Eine Grammatik ist **mehrdeutig**, wenn es ein Wort $w \in L(G)$ gibt, das mehrere **Ableitungsbäume** zulässt“.*[a][b]

[a] Alternativ, wenn es für $w$ **mehrere** unterschiedliche **Linksableitungen** gibt.
[b] Nebel, „Theoretische Informatik“.

### 2.2.2 Präzedenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Konkreten Grammatik** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 2.27) und **Präzedenz** (Definition 2.28) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel **??** genauer erklärt.

**Definition 2.27: Assoziativität**

*„Bestimmt, welcher Operator aus einer Reihe **gleicher** Operatoren **zuerst** ausgewertet wird.“*

*Es wird grundsätzlich zwischen **linksassoziativen** Operatoren, bei denen der **linke Operator** vor dem **rechten Operator** ausgewertet wird und **rechtsassoziativen** Operatoren, bei denen es genau anders rum ist unterschieden.*[a]

[a] *Parsing Expressions · Crafting Interpreters.*

Bei **Assoziativität** ist z.B. der **Multitplikationsoperator** `*` ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator** `=` ein Beispiel für einen **rechtsassoziativen** Operator. Dies ist in Abbildung 2.3 mithilfe von Klammern `()` veranschaulicht.

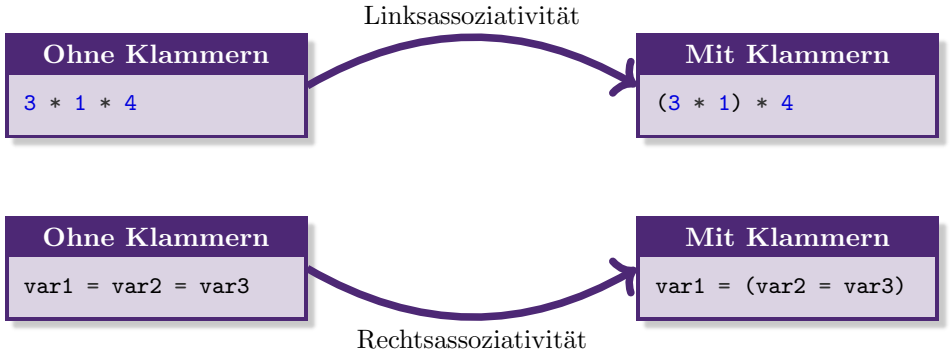



**Abbildung 2.3:** *Veranschaulichung von Linksassoziativität und Rechtsassoziativität.*

**Definition 2.28: Präzedenz**

*„Bestimmt, welcher Operator **zuerst** in einem Ausdruck, der eine Mischung **verschiedener** Operatoren enthält, ausgewertet wird. Operatoren mit einer **höheren Präzedenz**, werden **vor** Operatoren mit **niedrigerer Präzedenz** ausgewertet.“*[a]

[a] *Parsing Expressions · Crafting Interpreters.*

Bei **Präzedenz** ist die Mischung der Operatoren für **Subraktion** `'-'` und für **Multiplikation** `*` ein Beispiel für den Einfluss von Präzedenz. Dies ist in Abbildung 2.4 mithilfe der Klammern `()` veranschaulicht. Im Beispiel in Abbildung 2.4 ist bei den beiden **Subtraktionsoperatoren** `'-'` nacheinander und dem darauffolgenden **Multitplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzedenz** im Spiel.


**Abbildung 2.4:** *Veranschaulichung von Präzedenz.*

## 2.3 Lexikalische Analyse

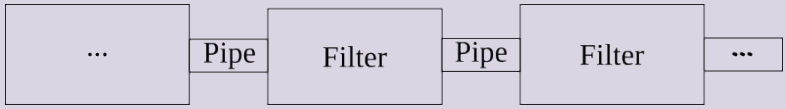
Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.29) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt in einem Eingabewort[8] endliche Folgen Symbolen[9] zu finden, die durch bestimmte **Pattern** (Definition 2.30) erkannt werden, die durch eine **reguläre Grammatik** spezifiziert sind. Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.31) genannt.

**Definition 2.29: Pipe-Filter Architekturpattern**

*Ist ein **Archikteturpattern**, welches aus **Pipes** und **Filtern** besteht, wobei der **Ausgang** eines **Filters** der **Eingang** des durch eine **Pipe** verbundenen adjazenten nächsten **Filters** ist, falls es einen gibt.*

*Ein **Filter** stellt einen Schritt dar, indem eine Eingabe **weiterverarbeitet** wird und **weitergereicht** wird. Bei der **Weiterverarbeitung** können Teile der Eingabe **entfernt**, **hinzugefügt** oder **vollständig ersetzt** werden.*

*Eine **Pipe** stellt ein **Bindeglied** zwischen zwei **Filtern** dar.*[a][b]




[a] Das ein **Bindeglied** eine eigene Bezeichnung erhält, bedeutet allerdings nicht, dass es eine eigene wichtige **Aufgabe** erfüllt. Wie bei vielen **Pattern**, soll mit dem Namen des **Pattern**, in diesem Fall durch das **Pipe** die Anlehung an z.B. die **Pipes aus Unix**, z.B. `cat /proc/bus/input/devices | less` zum Ausdruck gebracht werden. Und so banal es klingt, sollen manche Bezeichnungen von Pattern auch einfach nur gut klingen.
[b] Westphal, „Softwaretechnik“.

**Definition 2.30: Pattern**

***Beschreibung** aller möglichen **Lexeme**, die eine Menge $\mathbb{P}_T$ bilden und einem bestimmten **Token** $T$ zugeordnet werden. Die Menge $\mathbb{P}_T$ ist eine möglicherweise unendliche Menge von **Wörtern**, die sich mit den Produktionen einer **regulären Grammatik** $G_{Lex}$ einer **regulären Sprache** $L_{Lex}$ beschreiben lassen [a], die für die Beschreibung eines **Tokens** $T$ zuständig sind.*[b]

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] Thiemann, „Compilerbau“.

**Definition 2.31: Lexeme**

*Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches von einem **Pattern** für eines der **Token** $T$ einer **Sprache** $L_{Lex}$ erkannt wird.*[a]

[8] Z.B. dem Inhalt einer Datei, welche in **UTF-8** kodiert ist.
[9] Also **Teilwörter** des **Eingabeworts**.

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.32) im **Eingabewort** identifziert und **Tokens** $T$ zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** (Definition 2.32) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Ein **Lexeme** ist dabei nicht immer das gleiche wie der **Tokenwert**, denn z.B. im Fall von $L_{PicoC}$ kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.34) dargestellt werden, einmal als ASCII-Zeichen `'c'`, das dann als Tokenwert den entsprechenden Wertes aus der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[10]. Zu einem **Lexeme** wie z.B. `'c'` wäre das entsprechende **Token** dazu `Token('CHAR', '99')`. Bei einem **Lexeme** wie z.B. `'99'` wäre das entsprechende **Token** `Token('NUM', '99')`, bei dem das **Lexeme** mit dem **Tokenwert** übereinstimmt. Der **Tokenwert** ist der letztendlich verwendete **Wert** an sich, unabhängig von der Darstellungsform.

**Definition 2.32: Lexer (bzw. Scanner oder auch Tokenizer)**

*Ein **Lexer** ist eine **partielle** Funktion* $lex : \Sigma^* \rightharpoonup (N \times W)^*$*, welche ein **Wort** bzw. **Lexeme** aus* $\Sigma^*$ *auf ein **Token** $T$ mit einem **Tokennamen** $N$ und einem **Tokenwert** $W$ abbildet, falls dieses **Wort** sich unter der **regulären Grammatik** $G_{Lex}$, der **regulären Sprache** $L_{Lex}$ abbleiten lässt bzw. einem der **Pattern** der Sprache $L_{Lex}$ entspricht.*[a]

[a] Thiemann, „Compilerbau“.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die von keinem **Pattern** eines **Tokens** der Sprache $L_{Lex}$ erkannt werden. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** ausgegeben.

**Anmerkung**

Um Verwirrung verzubeugen ist es wichtig folgende Unterscheidung hervorzuheben:

Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktischen Analyse** und der **Code Generierung**, auf diesen verschiedenen Ebenen unterschiedliche Konzepte als Symbole bezeichnet.

In der Lexikalischen Analyse sind einzelne **Zeichen eines Zeichensatzes** die Symbole.

In der Syntaktischen Analyse sind die **Tokennamen** die Symbole.

In der Code Generierung sind die **Bezeichner** (Definition 2.33) von **Variablen, Konstanten und Funktionen** die Symbole[a].

[a] Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel **??** **Symboltabelle** genannt wird.

**Definition 2.33: Bezeichner (bzw. Identifier)**

***Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres Sichtbarkeitsbereichs **eindeutig** benennt.*[a][b]

[a] Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist.

[10] Die Programmiersprache $L_{Python}$ erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

[b] Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[11] und Tabs `\t` aus dem Eingabewort herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort $\epsilon$ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in (N \times W)^*$ ist immer der Fall bei der **Kleeneschen Hülle** $\Sigma^*$, wobei $\Sigma = N \times W$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert**, ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie `42`, `314` oder `12`. Die **Tokennamen** sind **Überbegriffe** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen.

Für die zuvor als Beispiel genannten **Bezeichner** und **Zahlen** wären z.B. `NAME`[12] und `NUM`[13] passende **Tokennamen**[14], bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für **Lexeme**, wie `if` oder `}` sind die **Tokennamen** genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

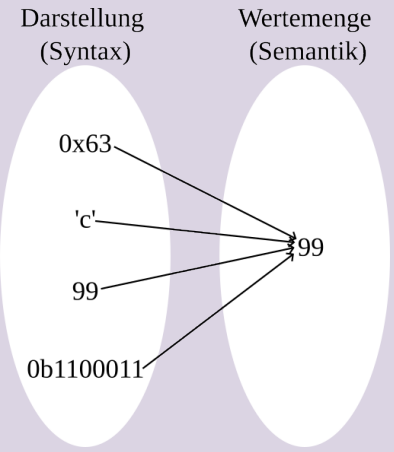
Die **Konkrete Grammatik** $G_{Lex}$, die zur Beschreibung der Token $T$ der Sprache $L_{Lex}$ verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[15], sich **nichts merkt**, also unabhängig davon, **was** für Symbole und **wie oft** bestimmte Symbole **davor** aufgetaucht sind funktioniert. Auch für den PicoC-Compiler lässt sich aus der Grammatik **??** schlussfolgern, dass die **Sprache** des **PicoC-Compilers** für die **Lexikalische Analyse** $L_{PicoC\_Lex}$ **regulär** ist, da alle ihre **Produktionen** die Definition 2.18 erfüllen.

Produktionen mit **Alternative**, wie z.B. $DIG\_WITH\_0 ::= \text{"0"} \mid DIG\_NO\_0$ sind **unproblematisch**, denn sie können immer auch als $\{DIG\_WITH\_0 ::= \text{"0"},\ DIG\_WITH\_0 ::= DIG\_NO\_0\}$ ausgedrückt werden und die Schreibweise "␣".."∼" in Grammatik **??** ist eine **Abkürzung** dafür alle ASCII-Symbole zwischen ␣ und ∼ als **Alternative** aufzuschreiben, womit diese **Alternativen** wie gerade gezeigt umgeformt werden können, um ebenfalls **regulär** zu sein. Somit existiert mit der Grammatik **??** eine **reguläre Grammatik**, welche die **Sprache** $L_{PicoC\_Lex}$ beschreibt und damit ist die **Sprache** $L_{PicoC\_Lex}$ nach der **Chromsky Hierarchie** (Definition 2.17) **regulär**.

**Definition 2.34: Literal**

*Eine von möglicherweise vielen weiteren **Darstellungsformen** (als **Zeichenkette**) für ein und denselben **Wert** eines **Datentyps**.*[a]

[11] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.
[12] Für z.B. `my_fun`, `my_var` und `my_const`.
[13] Für z.B. `42`, `314` und `12`.
[14] Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Knoten haben will, damit unter anderem **mehr Code** in eine Zeile passt.
[15] Man nennt das auch einem **Lookahead** von 1




[a]Thiemann, „Einführung in die Programmierung“.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

**Eingabewort**

```
void main() {
  if(42) {
  }
}
```

Lexer

**Tokenfolge**

```
[Token('FILENAME', './example1.picoc'), Token('VOID_DT',
↪ 'void'), Token('NAME', 'main'), Token('LPAR', '('),
↪ Token('RPAR', ')'), Token('LBRACE', '{'), Token('IF',
↪ 'if'), Token('LPAR', '('), Token('NUM', '42'),
↪ Token('RPAR', ')'), Token('LBRACE', '{'),
↪ Token('RBRACE', '}'), Token('RBRACE', '}')]
```

**Abbildung 2.5:** *Veranschaulichung der Lexikalischen Analyse.*

**Anmerkung**

Das Symbol ↪ zeigt im Code der Tokens in Abbildung 2.5 und in den folgenden Codes einen **Zeilenumbruch** an, wenn eine **Zeile zu lang** ist.

## 2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** $G_{Parse}$ notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern `'('` bzw. öffnende geschweifte Klammern `'{'` es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer `')'` bzw. schließende geschweifte Klammer `'}'` geschlossen wurden. Dies lässt sich nicht mehr mit einer **Regulären Grammatik** (Definition 2.18) beschreiben, sondern es braucht eine **Kontextfreie Grammatik** (Definition 2.19) hierfür, die es erlaubt zwischen **zwei Terminalsymbolen** ein **Nicht-Terminalsymbol** abzuleiten.

Für den PicoC-Compiler lässt sich aus der Grammatik **??** schlussfolgern, dass die **Sprache** des **PicoC-Compilers** für die **Syntaktische Analyse** $L_{PicoC\_Parse}$ **kontextfrei**, aber nicht mehr **regulär** ist, da **alle**

ihre **Produktionen** die Definition für **Kontextfreie Grammatiken** 2.19 erfüllen, aber **nicht** die Definition für **Reguläre Grammatiken** 2.18.

Dass die Grammatik **kontextfrei** ist lässt sich auch sehr leicht erkennen, weil **alle Produktionen** auf der **linken Seite** des ::=-Symbols immer nur ein **Nicht-Terminalsymbol** haben und auf der **rechten Seite** eine **beliebige Folge** von **Grammatiksymbolen**[16]. Dass diese Grammatik aber nicht **regulär** sein kann, lässt sich sehr einfach an z.B. der Produktion $if\_stmt ::= "if""("logic\_or")"\ \ exec\_part$ erkennen, bei der das **Nicht-Terminalsymbol** *logic_or* von den **Terminalsymbolen** für **öffnende Klammer** { und **schließende Klammer** } eingeschlossen sein muss, was mit einer **Regulären Grammatik** nicht ausgedrückt werden kann.

Somit existiert mit der Grammatik **??** eine **Kontextfreie Grammatik** und **nicht Reguläre Grammatik**, welche die **Sprache** $L_{PicoC\_Parse}$ beschreibt und damit ist die **Sprache** $L_{PicoC\_Parse}$ nach der **Chromsky Hierarchie** (Definition 2.17) **kontextfrei**, aber **nicht regulär**.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.35) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 2.38) ein **Ableitungsbaum** (Definition 2.37) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 2.44) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

**Definition 2.35: Konkrete Syntax**

*Steht für alles, was mit dem **Aufbau** von **Ableitungen** nach einer **Konkretten Grammatik** zu tuen hat.*

*Ein **Programm** in seiner **Textrepräsentation**, wie es in einer Textdatei nach den Produktionen der **Grammatiken** $G_{Lex}$ und $G_{Parse}$ abgeleitet steht, bevor man es kompiliert, ist in **Konkreter Syntax** aufgeschrieben.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Um einen kurzen Begriff für die **Grammatik**, welche die **Konkrete Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Konkrete Grammatik** (Definition 2.36) bezeichnet.

**Definition 2.36: Konkrete Grammatik**

*Grammatik, die eine **Konkrete Syntax** beschreibt.*

*In der **Konkreten Grammatik** entsprechen die **Terminalsymbole** den **Tokennamen**, der in der **Lexikalischen Analyse** generierten **Tokens**[a] und **Nicht-Terminalsymbole** entsprechen bei einem **Ableitungsbaum** den Stellen, wo ein **Teilbaum** eingehängt ist.*

[a] Wobei das **Lark Parsing Toolkit**, welches später bei der **Implementierung** verwendet wird eine spezielle **Metasyntax** zur Spezifikation von Grammatiken nutzt, bei der für bestimmten häufig genutzte **Terminalsymbolen** ein **Tokenwert** in die Grammatik geschrieben wird.

**Definition 2.37: Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)**

***Compilerinterne Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 2.25) eines in **Konkreter Syntax** geschriebenen Programmes.*

*Der **Formale Ableitungsbaum** visualiert dabei **Ableitungen** nach einer entsprechenden **Konkreten Grammatik hierarchisch** in einem Baum.*[a]

[16] Also eine **beliebige Folge** von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

[a] *JSON parser - Tutorial — Lark documentation*.

Die **Konkrete Grammatik** nach der **Ableitungsbaum** konstruiert ist, wird optimalerweise immer so definiert, dass sich möglichst **einfach** aus dem **Ableitungsbaum** ein **Abstrakter Syntaxbaum** konstruieren lässt.

**Definition 2.38: Parser**

*Ein **Parser** ist ein Programm, dass aus einem Eingabewort[a], welches in **Konkreter Syntax** geschrieben ist eine compilerinterne Datenstruktur, den **Ableitungsbaum** generiert, was auch als **Parsen** bezeichnet wird[b].*[c]

[a] Z.B. wiederum ein **Programm**.
[b] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.38 verwendet.
[c] *JSON parser - Tutorial — Lark documentation*.

**Anmerkung**

An dieser Stelle könnte möglicherweise eine Verwirrung enstehen, welche Rolle dann überhaupt ein **Lexer** hier spielt.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines **Parsers**. Der **Lexer** ist ausschließlich für die **Lexikalische Analyse** verantwortlich und entspricht z.B., wenn man bei einem Wanderausflug verschiedene Insekten entdeckt, dem Nachschlagen in einem Insekten**lexikon** und dem Aufschreiben, welchen Insekten man in welcher **Reihenfolge** begegnet ist. Zudem kann man bestimmte **Sehenswürdigkeiten** an denen man während des Ausflugs vorbeikommt ebenfalls festhalten, da es eine Rolle spielen kann in welchem örtlichen **Kontext** man den Insekten begegnet ist[a].

Der **Parser** vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich und entspricht, um auf das Beispiel zurückzukommen, dem Darstellen von **Beziehungen** zwischen den Insektenbegnungen in einer für die **Weiterverarbeitung tauglichen Form**[b].

In der Weiterverarbeitung kann der **Interpreter** das interpretieren und daraus bestimmte Schlüsse ziehen und ein **Compiler** könnte es vielleicht in eine für Menschen leichter entschüsselbare Sprache kompilieren.

[a] Das würde z.B. der Rolle eines **Semikolon** `;` in der Sprache $L_{PicoC}$ entsprechen.
[b] Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beinflussen sich gegenseitig und ihre Umwelt.

Die vom **Lexer** im Eingabewort identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung in der **Grammatik** $G_{Parse}$ entspricht. Dabei wird in der Grammatik $L_{Parse}$ nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und nicht, welchen konkreten Wert diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.39), denn ein Parser löst das **Wortproblem** (Definition 2.20) für die **Sprache**, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Ableitungsbaum**.

**Definition 2.39: Recognizer (bzw. Erkenner)**

*Entspricht einem **Kellerautomaten**[a], in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit den Produktionen der **Konkreten Grammatik** einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Grammatik** beschrieben wird oder nicht. Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** (Definition 2.20) bekannt.*[b c]

[a] **Automat** mit dem **Kontextfreie Grammatiken** erkannt werden.
[b] Der **deutsche** Begriff **„Erkenner“** scheint **nicht** wirklich **etabliert** und geläufig zu sein, daher wird im Folgenden der **englische** Begriff **„Recognizer“** verwendet.
[c] Thiemann, „Compilerbau“.

**Anmerkung**

Für das **Parsen** gibt es grundsätzlich **drei** verschiedene Ansätze:

- **Top-Down Parsing:** Der **Ableitungsbaum** wird von **oben-nach-unten** generiert, also von der **Wurzel** zu den **Blättern**. Dementsprechend fängt die Generierung des **Ableitungsbaumes** mit dem **Startsymbol** der **Konkreten Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man **Terminalsymbole** hat, die sich zum gewünschten **Eingabewort** abgeleitet haben oder sich herausstellt, dass dieses nicht abgeleitet werden kann.[a]

  Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist, weil das **Eingabewort** von **links nach rechts** eingelesen wird, was gut damit zusammenpasst, dass die **Linksableitung** die **Blätter** von **links-nach-rechts** generiert.

  Welche der **Produktionen** für ein **Nicht-Terminalsymbol** angewandt wird, wenn es mehrere **Alternativen** gibt, wird entweder durch **Backtracking** oder durch **Vorausschauen** gelöst.

  Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg** (Definition 3.6).

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.24) allerdings nicht möglich, ohne die Konkrete Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der Konkreten Grammatik entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

  **Rekursiver Abstieg** kann mit **Backtracking** verbunden werden, um auch Konkrete Grammatiken parsen zu können, die nicht **LL(k)** (Definition 3.7) sind. Dabei werden meist nach dem Prinzip der **Tiefensuche** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass das **Eingabewort** sich **nicht** mit der verwendeten Konkreten Grammatik ableiten lässt.[b]

  Wenn man eine **LL(k)**-Grammatik hat, kann man auf **Backtracking verzichten** und es reicht einfach nur immer $k$ **Token** im Eingabewort **vorauszuschauen**. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.[c]

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** einer **Konkreten Grammatik** rückwärts anzuwenden, bis man beim **Startsymbol** landet.[d]

- **Chart Parsing:** Es wird **Dynamische Programmierung** verwendet und **partielle Zwischen-**

**ergebnisse** werden in einer **Tabelle** (bzw. einem **Chart**) gespeichert und können **wiederverwendet** werden. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist[e]. **Chart Parser** können dabei **top-down** oder **bottom-up** Ansätze umsetzen. Da die **Implementierung** von **Chart Parsern** fundamental anders ist als bei **Top-Down** und **Bottom-Up Parsern**, wird diese **Kategorie** von Parsern nochmal **speziell unterschieden** und nicht gesagt, es sei ein **Top-Down Parser** oder **Bottom-Up Parser**, der **Dynamische Programmierung** verwendet.

[a] *What is Top-Down Parsing?*
[b] Diese Form von Parsing wurde im **PicoC-Compiler** implementiert, als dieser noch auf dem Stand des **Bachelorprojektes** war, bevor er durch den nicht selbst implementierten **Earley Parser** von **Lark** (siehe Webseite *Lark - a parsing toolkit for Python*) ersetzt wurde.
[c] Diese Art von Parser ist im **RETI-Interpreter** implementiert, da die **RETI-Sprache** eine besonders simple **LL(1) Grammatik** besitzt. Diese Art von **Parser** wird auch als **Predictive Parser** oder **LL(k) Recursive Descent Parser** bezeichnet, wobei **Recursive Descent** das englische Wort für **Rekursiven Abstieg** ist.
[d] *What is Bottom-up Parsing?*
[e] Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

Der **Abstrakte Syntaxbaum** wird mithilfe von **Transformern** (Definition 2.40) und **Visitors** (Definition 2.41) generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese ein Programm von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 2.42).

**Definition 2.40: Transformer**

*Ein Programm, das von **unten-nach-oben**[a] nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaum** besucht und beim Antreffen eines bestimmten Knoten des **Ableitungsbaumes** je nach Kontext einen entsprechenden Knoten des **Abstrakten Syntaxbaumes** erzeugt und diesen anstelle des Knotens des **Ableitungsbaumes** setzt und so Stück für Stück den **Abstrakten Syntaxbaum** konstruiert.*[b]

[a] In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.
[b] *Transformers & Visitors — Lark documentation*.

**Definition 2.41: Visitor**

*Ein Programm, das von **unten-nach-oben**[a], nach dem Prinzip der Breitensuche alle Knoten des **Ableitungsbaumes** besucht und beim Antreffen eines bestimmten **Knoten** des Ableitungsbaumes, diesen **in-place** mit anderen Knoten **tauscht** oder **manipuliert**, um den Ableitungbaum für die weitere Verarbeitung durch z.B. einen Transformer zu vereinfachen.*[b c]

[a] In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.
[b] Kann theoretisch auch zur Konstruktion eines **Abstrakten Syntaxbaumes** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakten Syntaxbaumes** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.
[c] *Transformers & Visitors — Lark documentation*.

**Definition 2.42: Abstrakte Syntax**

*Steht für alles, was mit dem **Aufbau** von **Abstrakten Syntaxbäumen** zu tuen hat.*

*Ein **Abstrakter Syntaxbaum**, der zur Kompilierung eines Wortes*[a] *generiert wurde befindet sich in **Abstrakter Syntax**.*[b]

[a] Z.B. **Programmcode**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Um einen kurzen Begriff für die **Grammatik**, welche die **Abstrakte Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 2.43) bezeichnet.

**Definition 2.43: Abstrakte Grammatik**

*Grammatik, die eine **Abstrakte Syntax** beschreibt, also beschreibt was für Arten von **Kompositionen** mit den **Knoten** eines **Abstrakten Syntaxbaumes** möglich sind.*

*Jene Produktionen, die in der **Konkreten Grammatik** für die Umsetzung von **Präzedenz** notwendig waren, sind in der **Abstrakten Grammatik** abgeflacht. Dadurch sind die **Kompositionen**, welche die Knoten im **Abstrakten Syntaxbaum** bilden können **syntaktisch** meist näher an der Syntax von **Maschinenbefehlen**.*

**Definition 2.44: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)**

*Ist ein **compilerinterne Datenstruktur**, welche eine **Abstraktion** eines dazugehörigen **Ableitungsbaumes** darstellt, in dessen Aufbau auch das Erfordernis eines **leichten Zugriffs** und einer **leichten Weiterverarbeitbarkeit** eingeflossen ist. Bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, soll man möglichst schnell die Fragen beantworten können, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.*

*Die Knoten des **Abstrakten Syntaxbaumes** enthalten dabei verschiedene **Attribute**, welche wichtigen Informationen für den Kompiliervorang und Fehlermeldungen enthalten.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Im **Abstrakten Syntaxbaum** können theoretisch auch die **Token** aus der **Lexikalischen Analyse** weiterverwendet werden, allerdings ist dies **nicht empfehlenswert**. Es ist zum empfehlen die **Token** durch eigene entsprechende Knoten umzusetzen, damit der **Zugriff** auf Knoten des Abstrakten Syntaxbaumes immer **einheitlich** erfolgen kann und auch, da manche **Token** des Abstrakten Syntaxbaum nocht **nicht optimal benannt** sind. Manche „Symbole“ werden in der **Lexikalischen Analyse** mehrfach verwendet, wie z.B. das Symbol `-` in $L_{PicoC}$, welches für die **binäre Subtraktionsoperation** als auch die **unäre Minusoperation** verwendet wurde. Der verwendete **Tokenname** dieses Symbols lautet im **PicoC-Compiler** `SUB_MINUS`. Da in der **Syntaktischen Analyse** beide Operationen nur in **bestimmten Kontexten** vorkommen, **lassen** sie sich **unterscheiden** und dementsprechend können für beide Operationen jeweils zwei **seperate Knoten** erstellt werden. Im Fall des **PicoC-Compilers** sind es die Knoten `Sub()` und `Minus()`.

Im Gegensatz zum **Formalen Ableitungsbaum**, ergibt es beim **Abstrakten Syntaxbaum** keinen Sinn zusätzlich einen **Formalen Abstrakten Syntaxbaum** zu unterschieden, da das Konzept eines **Abstrakten Syntaxbaumes** ohne eine Datenstruktur zu sein für sich allein gesehen keine Anwendung hat. Wenn von Abstrakten Syntaxbäumen die Rede ist, ist immer eine **Datenstruktur** gemeint.

Die **Abstrakte Grammatik** nach der ein **Abstrakter Syntaxbaum** konstruiert ist wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten[17] möglichst **einfach weiterverarbeitet** werden kann.

In Abbildung 2.6 wird das Beispiel aus Unterkapitel 2.2.1 fortgeführt, welches den **Arithmetischen Ausdruck** `4 * 2` in Bezug auf die Konkrete Grammatik 2.1, welche die **höhere Präzedenz** der **Multipikation** $*$ berücksichtigt in einem **Ableitungsbaum** darstellt. In Abbildung 2.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**. Das geschieht bezogen auf das Beispiel aus Unterkapitel 2.2.1, indem jegliche Knoten **wewgabstrahiert** werden, die im **Ableitungsbaum** nur existieren, weil die Konkrete Grammatik so umgesetzt ist, dass es nur **einen** einzigen möglichen **Ableitungsbaum** geben kann.

[17] Den verschiedenen **Passes**.


**Abbildung 2.6:** *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

Die **Baumdatenstruktur** des **Ableitungsbaumes** und **Abstrakten Syntaxbaumes** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Programmes ausführen muss möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche er ausführen muss.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, sind in Abbildung 2.7 die einzelnen **Zwischenschritte** von den Tokens der **Lexikalischen Analyse** zum **Abstrakten Syntaxbaum** anhand des fortgeführten Beispiels aus Subkapitel 2.3 veranschaulicht. In Abbildung 2.7 werden die Darstellungen des **Ableitungsbaumes** und des **Abstrakten Syntaxbaumes** verwendet, wie sie vom **PicoC-Compiler** ausgegeben werden. In der Darstellung des **PicoC-Compilers** stellen die verschiedenen **Einrückungen** die verschiedenen **Ebenen** dieser Bäume dar. Die Bäume **wachsen** von der **Wurzel** von **links-nach-rechts** zu den **Blättern**.

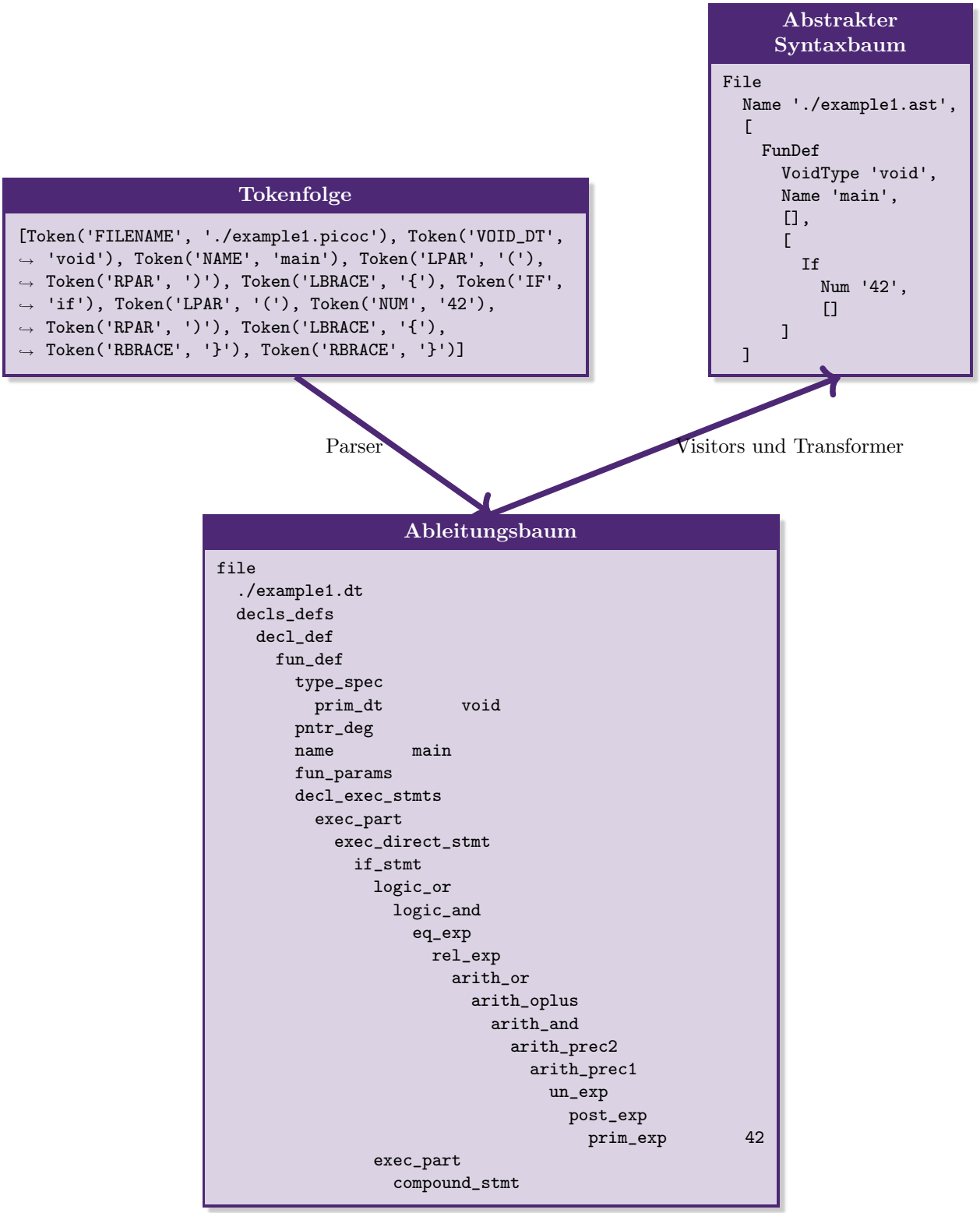



**Abbildung 2.7:** *Veranschaulichung der Syntaktischen Analyse.*

## 2.5 Code Generierung

In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** einer Sprache $L_1$ in den **Abstrakten Syntaxbaum** einer Sprache $L_2$ umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.45) nennt. So wie es auch schon mit dem **Ableitungsbaum** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakten Syntaxbaum** kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

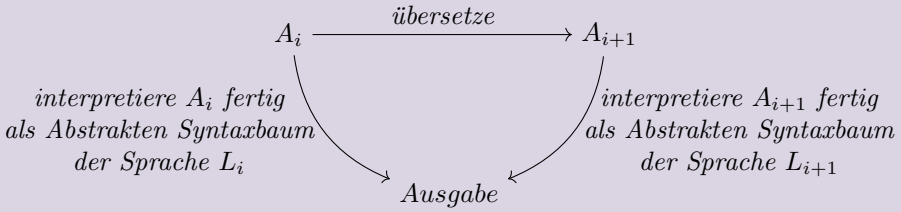
**Anmerkung**

Man spricht hier von dem **„Abstrakten Syntaxbaum einer Sprache $L_1$ (bzw. $L_2$)“** und meint hier mit der Sprache $L_1$ (bzw. $L_2$) **nicht** die Sprache, welche durch die **Abstrakte Grammatik**, nach welcher der **Abstrakte Syntaxbaum** abgeleitet ist beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll und zu deren Zweck der **Abstrakte Syntaxbaum** überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Grammatik** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Konvention** wurde aus dem Buch G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

**Definition 2.45: Pass**

***Einzelner Übersetzungsschritt*** *in einem* ***Kompiliervorgang*** *von einem beliebigen* ***Abstrakten Syntaxbaum*** $A_i$ *einer Sprache* $L_i$ *zu einem* ***Abstrakten Syntaxbaum*** $A_{i+1}$ *einer Sprache* $L_{i+1}$*, der meist* ***eine*** *bestimmte* ***Teilaufgabe*** *übernimmt, die sich mit keiner* ***Teilaufgabe*** *eines anderen* ***Passes*** *überschneidet und möglichst wenig* ***Ähnlichkeit*** *mit den* ***Teilaufgaben*** *anderer* ***Passes*** *haben sollte.*[a][b]

*Für* ***jeden Pass*** *und für einen* ***beliebigen*** *Abstrakten Syntaxbaum* $A_i$ *gilt ähnlich, wie bei einem* ***vollständigen Compiler*** *in 2.45.1, dass:*




(2.45.1)

*wobei man hier so tut, als gäbe es zwei* ***Interpreter*** *für die zwei Sprachen* $L_i$ *und* $L_{i+1}$*, welche den jeweiligen* ***Abstrakten Syntaxbaum*** $A_i$ *bzw.* $A_{i+1}$ *fertig interpretieren.*[c][d]

[a] Ein **Pass** kann mit einem **Transpiler** 3.5 (Definition 3.5) verglichen werden, da sich die zwei Sprachen $L_i$ und $L_{i+1}$ aufgrund der **Kleinschrittigkeit** meist auf einem ähnlichen **Abstraktionslevel** befinden. Der Unterschied ist allerdings, dass ein **Transpiler** zwei Programme, die in $L_i$ bzw. $L_{i+1}$ geschrieben sind kompiliert. Ein **Pass** ist dagegen immer **kleinschrittig** und operiert ausschließlich auf **Abstrakten Syntaxbäumen**, ohne Parsing usw.

[b] Der Begriff kommt aus dem **Englischen** von „passing over“, da der gesamte **Abstrakte Syntaxbaum** in einem **Pass** durchlaufen wird.

[c] **Interpretieren** geht immer von einem Programm in **Konkreter Syntax** aus, wobei der **Abstrakte Syntaxbaum** ein **Zwischenschritt** bei der **Interpretierung** ist.

[d] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Die von den **Passes** umgeformten **Abstrakten Syntaxbäume** sollten dabei mit jedem **Pass** der **Syntax** von **Maschinenbefehlen** immer ähnlicher werden, bis es schließlich nur noch Maschinenbefehle sind.

### 2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche **Unreine Ausdrücke** (Definition 2.47) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.46) und **Unreine Ausdrücke** voneinander **trennt**. Das wird erreicht, indem man aus den Unreinen Ausdrücken **vorangestellte Statements** macht, die man **vor** den jeweiligen reinen Ausdruck, mit dem sie **gemischt** waren stellt. Der Unreine Ausdruck muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, denn ein **Unreiner Ausdruck** hatte den **Reinen Ausdruck**, mit dem er gemischt war in irgendeinerweise beeinflussen könnte.

**Definition 2.46: Reiner Ausdruck (bzw. engl. pure expression)**

*Ein **Reiner Ausdruck** ist ein Ausdruck, der **rein** ist. Das bedeutet, dass dieser Ausdruck **keine Nebeneffekte** erzeugt. Ein **Nebeneffekt** ist eine **Bedeutung**, die ein Ausdruck hat, die sich **nicht** mit **RETI-Code** darstellen lässt.*[a][b]

[a] Sondern z.B. **intern** etwas am **Kompilierprozess** ändert.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.47: Unreiner Ausdruck**

*Ein **Unreiner Ausdruck** ist ein Ausdruck, der **kein Reiner Ausdruck** ist.*

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definiton 2.48).

**Definition 2.48: Monadische Normalform (bzw. engl. monadic normal form)**

*Ein **Statement** oder **Ausdruck** ist in **Monadischer Normalform**, wenn es oder er nach einer **Konkreten Grammatik** in **Monadischer Normalform** abgeleitet wurde.*

*Eine **Konkrete Grammatik** ist in **Monadischer Normalform**, wenn sie **reine Ausdrücke** und **unreine Ausdrücke nicht** miteinander **mischt**, sondern voneinander **trennt**.*[a]

*Eine **Abstrakte Grammatik** ist in **Monadischer Normalform**, wenn die **Konkrete Grammatik** für welche sie definiert wurde in **Monadischer Normalform** ist.*

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**[18] aufgeschrieben wurden.

In der Abbildung 2.8 ist der Ausdruck mit dem **Nebeneffekt** eine Variable zu **allokieren:** `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt, daher muss der **Unreine** Ausdruck als eigenständiges Statement **vorangestellt** werden.

[18] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.


**Abbildung 2.8:** *Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten.*

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in **RETI-Knoten** haben. Somit wird eine **Seperation** von Subbäumen, die keine Entsprechung in **RETI-Knoten** haben und denen, die eine haben bewerkstelligt wird. Ein **Reiner Ausdruck** ist **Maschinenbefehlen** ähnlicher als ein Ausdruck, indem ein **Reiner** und **Unreiner Ausdruck** gemischt sind. Somit sparrt man sich in der Implementierung **Fallunterscheidungen**, indem die **Reinen Ausdrücke** direkt in **RETI-Code** übersetzt werden können und **nicht** unterschieden werden muss, ob darin **Unreine Ausdrücke** vorkommen.

### 2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache** $L_1$ um eine **höhere Programmiersprache** und bei $L_2$ um **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.51) aus **Statements** und **Ausdrücken** entfernt. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Statements macht, in denen die **Komplexen Ausdrücke temporären Locations** zugewiesen werden (Definiton 2.49) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statemtent**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 2.50) enthalten.

Sollte es sich bei dem **Komplexen Ausdruck** um einen **Unreinen Ausdruck** handeln, welcher nur einen **Nebeneffekt** ausführt und sich nicht in **RETI-Befehle** übersetzt, so wird aus diesem ein **vorangestelltes Statement** gemacht, welches einfach nur den **Nebeneffekt** dieses **Unreinen Ausdrucks** ausführt.

**Definition 2.49: Location**

*Kollektiver Begriff für **Variablen**, **Attribute** bzw. **Elemente** von Variablen bestimmter Datentypen, **Speicherbereiche auf dem Stack**, die **temporäre Zwischenergebnisse** speichern und **Register**.*

*Im Grunde genommen alles, was mit einem **Programm zu tuen** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.52). Wenn eine **Konkrete Grammatik** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.52), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.50).

**Definition 2.50: Atomarer Ausdruck**

*Ein **Atomarer Ausdruck** ist ein Ausdruck, der ein **Reiner Ausdruck** ist und der in eine **Folge von RETI-Befehlen** übersetzt werden kann, die **atomar** ist, also **nicht** mehr weiter in kleinere Folgen von RETI-Befehlen **zerkleinert** werden kann, welche die **Übersetzung** eines anderen Ausdrucks sind.*

*Also z.B. im Fall der Sprache $L_{PicoC}$ entweder eine **Variable** `var`, eine **Zahl** `12`, ein **ASCII-Zeichen** `'c'` oder ein **Zugriff auf eine Location**, wie z.B. `stack(1)`.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.51: Komplexer Ausdruck**

*Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist, wie z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`.*[a][b]

[a] `int var` ist eine **Allokation**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 2.52: A-Normalform (ANF)**

*Ein **Statement** oder **Ausdruck** ist in **A-Normalform**, wenn es oder er nach einer **Konkreten Grammatik** in **A-Normalform** abgeleitet wurde.*

*Eine **Konkrete Grammatik** ist in **A-Normalform**, wenn sie in **Monadischer Normalform** ist und wenn alle **Komplexen Ausdrücke** nur **Atomare Ausdrücke** enthalten und einer **Location** zugewiesen sind.*

*Eine **Abstrakte Grammatik** ist in **A-Normalform**, wenn die **Konkrete Grammatik** für welche sie definiert wurde in **A-Normalform** ist.*[a][b][c]

[a] *A-Normalization: Why and How (with code)*.
[b] Bolingbroke und Peyton Jones, „Types are calling conventions“.
[c] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.9 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**[19] aufgeschrieben wurden.

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition 3.11) inklusive **Liveness Analysis** (Definition 3.9) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.[20]

Aus diesem Grund verwendet das Beispiel in Abbildung 2.9 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **Abstrakte Syntaxbaum** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdrück** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

Im Fall, dass **Register** für z.B. **temporäre Zwischenergebnisse** genutzt werden und der **Maschinen-**

[19] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[20] Die in diesem **Paragraph** erwähnten **Begriffe** werden nur grob erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

**befehlssatz** es erlaubt **zwei Register** miteinander zu verechnen[21], ist es möglich **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ als **atomar** zu definieren, da sie mit einem **Maschinenbefehl** verarbeitet werden können[22]. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das `SP`-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verechnet werden können.

Die Statements `4`, `x`, usw. für sich sind in diesem Fall **Statements**, bei denen ein **Komplexer Ausdruck** einer **Location**, in diesem Fall einer **Speicherzelle des Stack** zugewiesen wird, da `4`, `x` usw. in diesem Fall auch als **Komplexe Ausdrücke** zählen. Auf das Ergebnis dieser **Komplexen Ausdrücke** wird mittels `stack(2)` und `stack(1)` zugegriffen, um diese im **Komplexen Ausdruck** `stack(2) % stack(1)` miteinander zu verrechnen und wiederum einer Speicherzelle des Stack zuzuweisen.

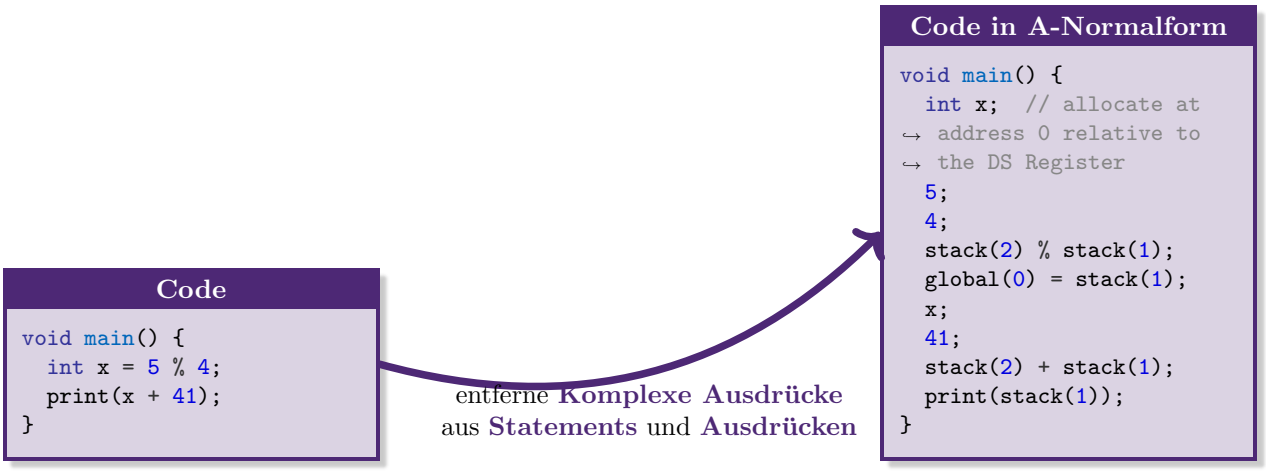



**Abbildung 2.9:** *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.*

Ein solcher **Pass** hat vor allem in erster Linie die Aufgabe den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** besonders dadurch anzunähren, dass er die Statements **weniger komplex** macht und diese dadurch den ziemlich **simplen Maschinenbefehlen** syntaktisch ähnlicher sind. Des Weiteren **vereinfacht** dieser Pass die **Implementierung** der nachfolgenden Passes enorm, da Statements z.B. nur noch die Form `global(rel_addr) = stack(1)` haben, die viel **einfacher verarbeitet** werden kann.

Alle weiteren denkbaren **Passes** sind zu **spezifisch** auf bestimmte **Statements** und **Ausdrücke** ausgelegt, als das sich zu diesen allgemein etwas mit einer **Theorie** dahinter sagen lässt. Alle **Passes**, die zur Implementierung des **PicoC-Compilers** geplant und ausgedacht wurden sind im Unterkapitel **??** definiert.

### 2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden ist es notwendig aus dem finalen **Abstrakten Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt. Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben. Der Weg von der **Abstrakten Syntax** zur **Konkreten Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax** zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden musste.

[21] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[22] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.

Jeder **Knoten** des **Abstrakten Syntaxbaumes** erhält dazu eine Methode, welche hier `to_string` genannt wird, die eine **Textrepräsentation** seiner selbst und all seiner Knoten mit an den richtigen Stellen passend gesetzten **Semikolons** ; usw. ausgibt. Dabei wird nach dem Prinzip der **Tiefensuche** der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die immer wiederum die Methode `to_string` ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgebeben**.

## 2.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**[23] eintritt:

1. der **Parser**[24] entscheidet das **Wortproblem** für ein **Eingabeprogramm**[25] mit `0`, also das **Eingabeprogramm** befolgt nicht die **Syntax** der **Sprache** des Compilers[26].

2. in den **Passes** tritt eine Fall ein, der **nicht** in der **Semantik** der Sprache des Compilers abgedeckt ist, z.B.:

   - eine **Variable** wird **verwendet**, obwohl sie noch **nicht deklariert** ist.

   - bei einem **Funktionsaufruf** werden **mehr** Argumente oder Argumente des **falschen Datentyps** übergeben, als in der **Funktionsdeklaration** oder **Funktionsdefinition** angegeben ist.

3. Während der **Laufzeit** des Compilers tritt ein Ereignis ein, das **nicht** durch die **Semantik** der Sprache des Compilers abgedeckt ist oder das **Betriebssystem** nicht erlaubt, z.B.:

   - eine **nicht erlaubte Operation**, wie **Division durch** 0 (z.B. `42 / 0`) soll ausgeführt werden.

   - **Segmentation Fault:** Wenn auf **Speicher zugegriffen** wird, der vom **Betriebssystem geschützt** ist.

oder während des des **Linkens** (Definition 3.4) etwas nicht zusammenpasst, wie z.B.:

- es gibt **keine** oder **mehr als eine** `main`-Funktion.

- eine Funktion, die in einer **Objektdatei** (Definition 3.3) benötigt wird, wird von **keiner** anderen oder **mehr als einer** Objektdatei bereitsgestellt.

wird eine **Fehlermeldung** (Definition 2.53) ausgegeben.

**Definition 2.53: Fehlermeldung**

***Benachrichtigung*** *beliebiger Form, die einen* ***Grund*** *angibt weshalb ein Programm* ***nicht weiter ausgeführt*** *werden kann*[a]*. Das* ***Ausgeben*** *einer Fehlermeldung kann dabei auf* ***verschiedene Weisen*** *erfolgen, wie z.B.*

- *über* `stdout` *oder* `stderr` *im einem* ***Terminal Emulator*** *oder* ***richtigen Terminal***[b]*.*

- *über eine* ***Dialogbox*** *in einer* ***Graphischen Benutzerfläche***[c] *oder* ***Zeichenorientierten Benutzerschnittstelle***[d]*.*

[23] *Errors in C/C++ - GeeksforGeeks*.
[24] Bzw. der **Recognizer** im Parser.
[25] Bzw. **Wort**.
[26] Bzw. das **Eingabeprogramm** lässt sich **nicht** mit der Konkreten Grammatik des Compilers **ableiten**.

- *in ein **Register** oder an eine **spezielle Adresse** des **Hauptspeichers** wird ein **Wert geschrieben**.*
- ***Logdatei**[e] auf einem **Speichermedium**.*

[a] Dieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler selbst kompiliert** hat.
[b] Nur unter **Linux**, **Windows** hat sowas nicht.
[c] In engl. **G**raphical **U**ser **I**nterface, kurz **GUI**.
[d] In engl. **T**ext-based **U**ser **I**nterface, kurz **TUI**.
[e] In engl. **log file**.

### 2.6.1 Umsetzung von Zeigern

#### 2.6.1.1 Referenzierung

**Referenzierung** (z.B. `&var`) ist eine **Operation** bei der ein **Zeiger** auf eine **Location** in Form der **Anfangsadresse** dieser Location als Ergebnis zurückgegeben wird. Die Implementierung der **Referenzierung** wird im Folgenden anhand des Beispiels in Code 2.1 erklärt.

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

**Code 2.1:** *PicoC-Code für Zeigerreferenzierung.*

Der Knoten `Ref(Name('var'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 2.2 eine **Referenzierung** `&var` und der Knoten `PntrDecl(Num('1'), IntType('int'))` repräsentiert einen Zeiger `*pntr`.

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

**Code 2.2:** *Abstrakter Syntaxbaum für Zeigerreferenzierung.*

Bevor man einem **Zeiger** eine **Adresse** (z.B. `&var`) zuweisen kann, muss dieser erstmal **definiert** sein. Dafür braucht es einen Eintrag in der **Symboltabelle** in Code 2.3.

**Anmerkung**

Die **Größe** eines Zeigers (z.B. eines Zeigers auf ein Feld von `int`: `pntr = int *pntr[3]`), die im `size`-Attribut der **Symboltabelle** eingetragen ist, ist dabei immer: `size(pntr)` = 1.

```
1 SymbolTable
2   [
3     Symbol
4       {
5         type qualifier:         Empty()
6         datatype:               FunDecl(VoidType('void'), Name('main'), [])
7         name:                   Name('main')
```

```
8         value or address:       Empty()
9         position:               Pos(Num('1'), Num('5'))
10        size:                   Empty()
11      },
12    Symbol
13      {
14        type qualifier:         Writeable()
15        datatype:               IntType('int')
16        name:                   Name('var@main')
17        value or address:       Num('0')
18        position:               Pos(Num('2'), Num('6'))
19        size:                   Num('1')
20      },
21    Symbol
22      {
23        type qualifier:         Writeable()
24        datatype:               PntrDecl(Num('1'), IntType('int'))
25        name:                   Name('pntr@main')
26        value or address:       Num('1')
27        position:               Pos(Num('3'), Num('7'))
28        size:                   Num('1')
29      }
30  ]
```

**Code 2.3:** *Symboltabelle für Zeigerreferenzierung.*

Im **PicoC-ANF Pass** in Code 2.4 wird der Knoten `Ref(Name('var'))` durch die Knoten `Ref(GlobalRead(Num('0')))` und `Assign(GlobalWrite(Num('1')), Tmp(Num('1')))` ersetzt, deren Bedeutung in Unterkapitel **??** erklärt wurde. Im Fall, dass in `Ref(exp)` das `exp` vielleicht nicht direkt ein `Name('var')` enthält und `exp` z.B. ein `Subscr(Attr(Name('var'), Name('attr')))` ist, sind noch **weitere Anweisungen** zwischen den Zeilen `11` und `12` nötig, die sich in diesem Beispiel um das Übersetzen von `Subscr(exp)` und `Attr(exp, name)` nach dem Schema in Subkapitel **??** kümmern.

```
1 File
2   Name './example_pntr_ref.picoc_mon',
3   [
4     Block
5       Name 'main.0',
6       [
7         // Assign(Name('var'), Num('42'))
8         Exp(Num('42'))
9         Assign(Global(Num('0')), Stack(Num('1')))
10        // Assign(Name('pntr'), Ref(Name('var')))
11        Ref(Global(Num('0')))
12        Assign(Global(Num('1')), Stack(Num('1')))
13        Return(Empty())
14      ]
15  ]
```

**Code 2.4:** *PicoC-ANF Pass für Zeigerreferenzierung.*

Im **RETI-Blocks Pass** in Code 2.5 werden die **PicoC-Knoten** `Ref(Global(Num('0')))` und

`Assign(Global(Num('1')), Stack(Num('1')))` durch ihre **semantisch** entsprechenden **RETI-Knoten** ersetzt, deren Bedeutungen im Appendix in Unterkapitel 2.6.2.3 erklärt sind.

```
File
  Name './example_pntr_ref.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Name('var'), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Assign(Name('pntr'), Ref(Name('var')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Assign(Global(Num('1')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 1;
        ADDI SP 1;
        # Return(Empty())
        LOADIN BAF PC -1;
      ]
  ]
```

**Code 2.5:** *RETI-Blocks Pass für Zeigerreferenzierung.*

#### 2.6.1.2 Dereferenzierung durch Zugriff auf Feldindex ersetzen

**Dereferenzierung** (z.B. `*var`) ist eine **Operation** bei der einem **Zeiger** zur Location hin **gefolgt** wird, auf welche dieser zeigt und das Ergebnis z.B. der **Inhalt** der ersten Speicherzelle der referenzierten Location ist. Die Implementierung von **Dereferenzierung** wird im Folgenden anhand des Beispiels in Code 2.6 erklärt.

```
void main() {
  int var = 42;
  int *pntr = &var;
  *pntr;
}
```

**Code 2.6:** *PicoC-Code für Zeigerdereferenzierung.*

Der Knoten `Deref(Name('var'), Num('0'))` repräsentiert im **Abstrakten Syntaxbaum** in Code 2.7 eine **Dereferenzierung** `*var`. Es gibt herbei **zwei** Fälle. Bei der Anwendung von **Zeigerarithmetik**, wie z.B.

`*(var + 2 - 1)` übersetzt sich diese zu `Deref(Name('var'), BinOp(Num('2'), Sub(), Num('1')))`. Bei einer normalen **Dereferenzierung**, wie z.B. `*var`, übersetzt sich diese zu `Deref(Name('var'), Num('0'))`[27].

```
File
  Name './example_pntr_deref.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Deref(Name('pntr'), Num('0')))
      ]
  ]
```

**Code 2.7:** *Abstrakter Syntaxbaum für Zeigerdereferenzierung.*

Im **PicoC-Shrink Pass** in Code 2.8 wird ein **Trick** angewandet, bei dem jeder Knoten `Deref(Name('pntr'), Num('0'))` einfach durch den Knoten `Subscr(Name('pntr'), Num('0'))` ersetzt wird. Die Bedeutung des letzteren Knoten wurde in Unterkapitel **??** erklärt. Der Trick besteht darin, dass der **Dereferenzierungsoperator** (z.B. `*(var + 1)`) sich identisch zum **Operator für den Zugriff auf einen Feldindex** (z.B. `var[1]`) verhält, wie es bereits im Unterkapitel 1.3 erläutert wurde. Damit spart man sich viele vermeidbare **Fallunterscheidungen** und **doppelten Code** und kann die **Derefenzierung** (z.B. `*(var + 1)`) einfach von den Routinen für einen **Zugriff auf einen Feldindex** (z.B. `var[1]`) übernehmen lassen.

```
File
  Name './example_pntr_deref.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), IntType('int'), Name('var')), Num('42'))
        Assign(Alloc(Writeable(), PntrDecl(Num('1'), IntType('int')), Name('pntr')),
        ↪  Ref(Name('var')))
        Exp(Subscr(Name('pntr'), Num('0')))
      ]
  ]
```

**Code 2.8:** *PicoC-Shrink Pass für Zeigerdereferenzierung.*

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

### 2.6.2 Umsetzung von Feldern

#### 2.6.2.1 Initialisierung eines Feldes

Die **Initialisierung** eines **Feldes** (z.B. `int ar[2][1] = {{3+1}, {4}}`) wird im Folgenden anhand des Beispiels in Code 2.9 erklärt.

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

**Code 2.9:** *PicoC-Code für die Initialisierung eines Feldes.*

Die **Initialisierung** eines **Feldes** `intar[2][1]={{3+1},{4}}` wird im **Abstrakten Syntaxbaum** in Code 2.10 mithilfe der Komposition `Assign(Alloc(Writeable(),ArrayDecl([Num('2'),Num('1')],IntType('int')),Name('ar')),Array([Array([BinOp(Num('3'),Add('+'),Num('1'))]),Array([Num('4')])]))` dargestellt.

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

**Code 2.10:** *Abstrakter Syntaxbaum für die Initialisierung eines Feldes.*

Bei der **Initialisierung** eines **Feldes** wird zuerst `Alloc(Writeable(), ArrayDecl([Num('2'), Num('1')], IntType('int')))` ausgewertet, da eine Variable zuerst definiert sein muss, bevor man sie verwenden kann[28]. Das **Definieren** der Variable `ar` erfolgt mittels der **Symboltabelle**, die in Code 2.11 dargestellt ist.

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

Bei Variablen auf dem **Stackframe** wird ein Feld **rückwärts** auf das Stackframe geschrieben und auch die **Adresse des ersten Elements** als Adresse des Feldes genommen. Dies macht den **Zugriff auf einen Feldindex** in Subkapitel 2.6.2.2 deutlich unkomplizierter, da man so nicht mehr zwischen **Stackframe** und **Globalen Statischen Daten** beim **Zugriff auf einen Feldindex** unterscheiden muss, da es Probleme macht, dass ein **Stackframe** in die entgegengesetzte Richtung wächst, verglichen mit den **Globalen Statischen Daten**[29].

**Anmerkung**

Das **Größe** des Feldes `datatype ar`$[\mathtt{dim}_1]\dots[\mathtt{dim}_k]$, die ihm `size`-Attribut des **Symboltabelleneintrags** eingetragen ist, berechnet sich dabei aus der **Mächtigkeit** der einzelnen **Dimensionen** des Feldes multipliziert mit der **Größe** des **grundlegenden Datentyps** der einzelnen **Feldelemente**: $\mathtt{size(datatype(ar))} = \left(\prod_{j=1}^{n} \mathtt{dim}_j\right) \cdot \mathtt{size(datatype)}$[a].

[a] Die **Funktion** `type` ordnet einer **Variable** ihren **Datentyp** zu. Das ist notwendig, weil die **Funktion** `size` nur bei einem **Datentyp** als **Funktionsargument** die **Größe dieses Datentyps** als **Zielwert** liefert

```
1  SymbolTable
2    [
3      Symbol
4        {
5          type qualifier:          Empty()
6          datatype:                FunDecl(VoidType('void'), Name('main'), [])
7          name:                    Name('main')
8          value or address:        Empty()
9          position:                Pos(Num('1'), Num('5'))
10         size:                    Empty()
11       },
12     Symbol
13       {
14         type qualifier:          Writeable()
15         datatype:                ArrayDecl([Num('2'), Num('1')], IntType('int'))
16         name:                    Name('ar@main')
17         value or address:        Num('0')
18         position:                Pos(Num('2'), Num('6'))
19         size:                    Num('2')
20       },
21     Symbol
22       {
23         type qualifier:          Empty()
24         datatype:                FunDecl(VoidType('void'), Name('fun'), [])
25         name:                    Name('fun')
26         value or address:        Empty()
27         position:                Pos(Num('5'), Num('5'))
28         size:                    Empty()
29       },
30     Symbol
31       {
32         type qualifier:          Writeable()
33         datatype:                ArrayDecl([Num('2'), Num('2')], IntType('int'))
34         name:                    Name('ar@fun')
35         value or address:        Num('3')
36         position:                Pos(Num('6'), Num('6'))
```

[29] Wenn man beim **GCC** *GCC, the GNU Compiler Collection - GNU Project* einen Stackframe mittels des **GDB** *GCC, the GNU Compiler Collection - GNU Project* beobachtet, sieht man, dass dieser es genauso macht.

```
37        size:                    Num('4')
38      }
39  ]
```

**Code 2.11:** *Symboltabelle für die Initialisierung eines Feldes.*

Im **PiocC-ANF Pass** in Code 2.12 werden zuerst die **Logischen Ausdrücke** in den Blättern des Teilbaumes, der beim **Feld-Initializers** Knoten `Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]), Array([Num('4')])])` anfängt nach dem Prinzip der **Tiefensuche**, von **links-nach-rechts** ausgewertet und auf den **Stack** geschrieben[30].

Im finalen Schritt muss zwischen **Globalen Statischen Daten** bei der `main`-Funktion und **Stackframe** bei der Funktion `fun` unterschieden werden. Die auf den Stack ausgewerteten Expressions werden mittels der Komposition `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))`, die in Tabelle **??** genauer beschrieben ist, versetzt in der selben Reihenfolge zu den **Globalen Statischen Daten** bzw. auf den **Stackframe** geschrieben.

Der **Trick** ist hier, dass egal wieviele Dimensionen und was für einen grundlegenden Datentyp[31] das **Feld** hat, man letztendlich immer das gesamte Feld erwischt, wenn man einfach die **Größe des Feldes** viele **Speicherzellen** mit z.B. der **Komposition** `Assign(Global(Num('0')), Stack(Num('2')))` verschiebt.

In die Knoten `Global('0')` und `Stackframe('3')` wurde hierbei die **Startadresse** des jeweiligen Feldes geschrieben, sodass man nach dem **PicoC-ANF Pass** nie mehr Variablen in der **Symboltabelle** nachsehen muss und gleich weiß, ob sie in Bezug zu den **Globalen Statischen Daten** oder dem **Stackframe** stehen.

```
1  File
2    Name './example_array_init.picoc_mon',
3    [
4      Block
5        Name 'main.1',
6        [
7          // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
           ↪  Array([Num('4')])]))
8          Exp(Num('3'))
9          Exp(Num('1'))
10         Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
11         Exp(Num('4'))
12         Assign(Global(Num('0')), Stack(Num('2')))
13         Return(Empty())
14       ],
15     Block
16       Name 'fun.0',
17       [
18         // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
           ↪  Num('6')])]))
19         Exp(Num('3'))
20         Exp(Num('4'))
21         Exp(Num('5'))
22         Exp(Num('6'))
```

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

[31] Z.B. ein Feld von **Verbunden**.

```
23        Assign(Stackframe(Num('3')), Stack(Num('4')))
24        Return(Empty())
25      ]
26  ]
```

**Code 2.12:** *PicoC-ANF Pass für die Initialisierung eines Feldes.*

Im **RETI-Blocks Pass** in Code 2.13 werden die **Kompositionen** `Exp(exp)` und `Assign(Global(Num('0')), Stack(Num('2')))` bzw. `Assign(Stackframe(Num('3')), Stack(Num('4')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
1 File
2   Name './example_array_init.reti_blocks',
3   [
4     Block
5       Name 'main.1',
6       [
7         # // Assign(Name('ar'), Array([Array([BinOp(Num('3'), Add('+'), Num('1'))]),
          ↪  Array([Num('4')])]))
8         # Exp(Num('3'))
9         SUBI SP 1;
10        LOADI ACC 3;
11        STOREIN SP ACC 1;
12        # Exp(Num('1'))
13        SUBI SP 1;
14        LOADI ACC 1;
15        STOREIN SP ACC 1;
16        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
17        LOADIN SP ACC 2;
18        LOADIN SP IN2 1;
19        ADD ACC IN2;
20        STOREIN SP ACC 2;
21        ADDI SP 1;
22        # Exp(Num('4'))
23        SUBI SP 1;
24        LOADI ACC 4;
25        STOREIN SP ACC 1;
26        # Assign(Global(Num('0')), Stack(Num('2')))
27        LOADIN SP ACC 1;
28        STOREIN DS ACC 1;
29        LOADIN SP ACC 2;
30        STOREIN DS ACC 0;
31        ADDI SP 2;
32        # Return(Empty())
33        LOADIN BAF PC -1;
34      ],
35    Block
36      Name 'fun.0',
37      [
38        # // Assign(Name('ar'), Array([Array([Num('3'), Num('4')]), Array([Num('5'),
          ↪  Num('6')])]))
39        # Exp(Num('3'))
40        SUBI SP 1;
41        LOADI ACC 3;
```

```
42          STOREIN SP ACC 1;
43          # Exp(Num('4'))
44          SUBI SP 1;
45          LOADI ACC 4;
46          STOREIN SP ACC 1;
47          # Exp(Num('5'))
48          SUBI SP 1;
49          LOADI ACC 5;
50          STOREIN SP ACC 1;
51          # Exp(Num('6'))
52          SUBI SP 1;
53          LOADI ACC 6;
54          STOREIN SP ACC 1;
55          # Assign(Stackframe(Num('3')), Stack(Num('4')))
56          LOADIN SP ACC 1;
57          STOREIN BAF ACC -2;
58          LOADIN SP ACC 2;
59          STOREIN BAF ACC -3;
60          LOADIN SP ACC 3;
61          STOREIN BAF ACC -4;
62          LOADIN SP ACC 4;
63          STOREIN BAF ACC -5;
64          ADDI SP 4;
65          # Return(Empty())
66          LOADIN BAF PC -1;
67        ]
68    ]
```

**Code 2.13:** *RETI-Blocks Pass für die Initialisierung eines Feldes.*

#### 2.6.2.2 Zugriff auf einen Feldindex

Der **Zugriff auf einen Feldindex** (z.B. `ar[0]`) wird im Folgenden anhand des Beispiels in Code 2.14 erklärt.

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

**Code 2.14:** *PicoC-Code für Zugriff auf einen Feldindex.*

Der **Zugriff auf einen Feldindex** `ar[0]` wird im **Abstrakten Syntaxbaum** in Code 2.15 mithilfe des Knotens `Subscr(Name('ar'), Num('0'))` dargestellt.

```
File
  Name './example_array_access.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign(Alloc(Writeable(), ArrayDecl([Num('1')], IntType('int')), Name('ar')),
        ↪  Array([Num('42')]))
        Exp(Subscr(Name('ar'), Num('0')))
      ],
    FunDef
      VoidType 'void',
      Name 'fun',
      [],
      [
        Assign(Alloc(Writeable(), ArrayDecl([Num('3')], IntType('int')), Name('ar')),
        ↪  Array([Num('1'), Num('2'), Num('3')]))
        Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
      ]
  ]
```

**Code 2.15:** *Abstrakter Syntaxbaum für Zugriff auf einen Feldindex.*

Im **PicoC-ANF Pass** in Code 2.16 wird vom Knoten `Subscr(Name('ar'), Num('0'))` zuerst im **Anfangsteil** **??** die **Adresse** der Variable `Name('ar')` auf den **Stack** geschrieben. Bei den **Globalen Statischen Daten** der `main`-Funktion wird das durch die Komposition `Ref(Global(Num('0')))` dargestellt und beim **Stackframe** der Funktionm `fun` wird das durch die Komposition `Ref(Stackframe(Num('2')))` dargestellt.

In nächsten Schritt, dem **Mittelteil** **??** wird die **Adresse** ab der das **Feldelement** des Feldes auf das Zugegriffen werden soll anfängt berechnet. Dabei wurde im **Anfangsteil** bereits die **Anfangsadresse** des Feldes, in dem dieses **Feldelement** liegt auf den **Stack** gelegt. Da ein **Index** auf den Zugegriffen werden soll auch durch das Ergebnis eines **komplexeren Ausdrucks**, z.B. `ar[1 + var]` bestimmt sein kann, indem auch **Variablen** vorkommen können, kann dieser nicht während des **Kompilierens** berechnet werden, sondern muss zur **Laufzeit** berechnet werden.

Daher muss zuerst der Wert des **Index**, dessen Adresse berechnet werden soll bestimmt werden, z.B. im einfachen Fall durch `Exp(Num('0'))` und dann muss die **Adresse des Index** berechnet werden, was durch die Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` dargestellt wird. Die Bedeutung der Komposition `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` ist in Tabelle **??** dokumentiert.

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

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

Im letzten Schritt, dem **Schlussteil ??** wird der **Inhalt** des **Index**, dessen **Adresse** in den vorherigen Schritten berechnet wurde, nun auf den **Stack** geschrieben, wobei dieser die **Adresse** auf dem Stack ersetzt, die es zum Finden des **Index** brauchte. Dies wird durch den Knoten `Exp(Stack(Num('1')))` dargestellt. Je nachdem, welchen **Datentyp** die Variable `ar` hat und auf welchen **Unterdatentyp** folglich im **Kontext** zuletzt zugegriffen wird, abhängig davon wird der **Schlussteil** `Exp(Stack(Num('1')))` auf eine andere Weise verarbeitet (siehe Subkapitel **??**). Der **Unterdatentyp** ist dabei ein versteckes Attribut des `Exp(Stack(Num('1')))`-Knoten.

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

**Anmerkung**

Die Berechnung der **Adresse**, ab der ein **Feldelement** eines Feldes $\texttt{datatype ar}[\texttt{dim}_1]\dots[\texttt{dim}_n]$ abgespeichert ist, kann mittels der Formel 2.6.1:

$$\texttt{ref}(\texttt{ar}[\texttt{idx}_1]\dots[\texttt{idx}_n]) = \texttt{ref}(\texttt{ar}) + \left(\sum_{i=1}^{n}\left(\prod_{j=i+1}^{n}\texttt{dim}_j\right)\cdot\texttt{idx}_i\right)\cdot \text{size}(\texttt{datatype}) \tag{2.6.1}$$

aus der Betriebssysteme Vorlesung P. D. C. Scholl, „Betriebssysteme“ berechnet werden[a].

Die Komposition `Ref(Global(num))` bzw. `Ref(Stackframe(num))` repräsentiert dabei den Summanden $\texttt{ref}(\texttt{ar})$ in der Formel.

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

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

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

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

```
1  File
2    Name './example_array_access.picoc_mon',
3    [
4      Block
5        Name 'main.1',
6        [
7          // Assign(Name('ar'), Array([Num('42')]))
8          Exp(Num('42'))
9          Assign(Global(Num('0')), Stack(Num('1')))
10         // Exp(Subscr(Name('ar'), Num('0')))
11         Ref(Global(Num('0')))
12         Exp(Num('0'))
13         Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
14         Exp(Stack(Num('1')))
15         Return(Empty())
16       ],
```

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

**Code 2.16:** *PicoC-ANF Pass für Zugriff auf einen Feldindex.*

Im **RETI-Blocks Pass** in Code 2.17 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')) und Stack(Num('1'))))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        # // Assign(Name('ar'), Array([Num('42')]))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Assign(Global(Num('0')), Stack(Num('1')))
        LOADIN SP ACC 1;
        STOREIN DS ACC 0;
        ADDI SP 1;
        # // Exp(Subscr(Name('ar'), Num('0')))
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Exp(Num('0'))
        SUBI SP 1;
        LOADI ACC 0;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
        LOADIN SP IN1 2;
        LOADIN SP IN2 1;
        MULTI IN2 1;
```

```
30        ADD IN1 IN2;
31        ADDI SP 1;
32        STOREIN SP IN1 1;
33        # Exp(Stack(Num('1')))
34        LOADIN SP IN1 1;
35        LOADIN IN1 ACC 0;
36        STOREIN SP ACC 1;
37        # Return(Empty())
38        LOADIN BAF PC -1;
39      ],
40    Block
41      Name 'fun.0',
42      [
43        # // Assign(Name('ar'), Array([Num('1'), Num('2'), Num('3')]))
44        # Exp(Num('1'))
45        SUBI SP 1;
46        LOADI ACC 1;
47        STOREIN SP ACC 1;
48        # Exp(Num('2'))
49        SUBI SP 1;
50        LOADI ACC 2;
51        STOREIN SP ACC 1;
52        # Exp(Num('3'))
53        SUBI SP 1;
54        LOADI ACC 3;
55        STOREIN SP ACC 1;
56        # Assign(Stackframe(Num('2')), Stack(Num('3')))
57        LOADIN SP ACC 1;
58        STOREIN BAF ACC -2;
59        LOADIN SP ACC 2;
60        STOREIN BAF ACC -3;
61        LOADIN SP ACC 3;
62        STOREIN BAF ACC -4;
63        ADDI SP 3;
64        # // Exp(Subscr(Name('ar'), BinOp(Num('1'), Add('+'), Num('1'))))
65        # Ref(Stackframe(Num('2')))
66        SUBI SP 1;
67        MOVE BAF IN1;
68        SUBI IN1 4;
69        STOREIN SP IN1 1;
70        # Exp(Num('1'))
71        SUBI SP 1;
72        LOADI ACC 1;
73        STOREIN SP ACC 1;
74        # Exp(Num('1'))
75        SUBI SP 1;
76        LOADI ACC 1;
77        STOREIN SP ACC 1;
78        # Exp(BinOp(Stack(Num('2')), Add('+'), Stack(Num('1'))))
79        LOADIN SP ACC 2;
80        LOADIN SP IN2 1;
81        ADD ACC IN2;
82        STOREIN SP ACC 2;
83        ADDI SP 1;
84        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
85        LOADIN SP IN1 2;
86        LOADIN SP IN2 1;
```

```
87          MULTI IN2 1;
88          ADD IN1 IN2;
89          ADDI SP 1;
90          STOREIN SP IN1 1;
91          # Exp(Stack(Num('1')))
92          LOADIN SP IN1 1;
93          LOADIN IN1 ACC 0;
94          STOREIN SP ACC 1;
95          # Return(Empty())
96          LOADIN BAF PC -1;
97        ]
98    ]
```

**Code 2.17:** *RETI-Blocks Pass für Zugriff auf einen Feldindex.*

#### 2.6.2.3 Zuweisung an Feldindex

Die **Zuweisung** eines Wertes an einen **Feldindex** (z.B. `ar[2] = 42;`) wird im Folgenden anhand des Beispiels in Code 2.18 erläutert.

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

**Code 2.18:** *PicoC-Code für Zuweisung an Feldindex.*

Im **Abstrakten Syntaxbaum** in Code 2.19 wird eine **Zuweisung** an einen **Feldindex** `ar[2] = 42;` durch die Komposition `Assign(Subscr(Name('ar'), Num('2')), Num('42'))` dargestellt.

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

**Code 2.19:** *Abstrakter Syntaxbaum für Zuweisung an Feldindex.*

Im **PicoC-ANF Pass** in Code 2.20 wird zuerst die **rechte** Seite des **rechtsassoziativen** Zuweisungsoperators `=`, bzw. des Knotens der diesen darstellt ausgewertet: `Exp(Num('42'))`.

Danach ist das Vorgehen, bzw. sind die Kompostionen, die dieses darauffolgende Vorgehen darstellen: `Ref(Global(Num('0')))`, `Exp(Num('2'))` und `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` identisch zum

**Anfangsteil** und **Mittelteil** aus dem vorherigen Subkapitel 2.6.2.2. Es wird die **Adresse** des **Index**, dem das Ergebnis der Ausdrucks auf der rechten Seite des **Zuweisungsoperators** = zugewiesen wird berechet, wie in Subkapitel 2.6.2.2.

Zum Schluss stellt die **Komposition** `Assign(Stack(Num('1')), Stack(Num('2')))`[32] die Zuweisung = des Ergebnisses des Ausdrucks auf der **rechten** Seite der Zuweisung zum **Feldindex**, dessen **Adresse** im Schritt danach berechnet wurde dar.

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

**Code 2.20:** *PicoC-ANF Pass für Zuweisung an Feldindex.*

Im **RETI-Blocks Pass** in Code 2.21 werden die **Kompositionen** `Ref(Global(Num('0')))`, `Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))` und `Assign(Stack(Num('1')), Stack(Num('2')))` durch ihre entsprechenden **RETI-Knoten** ersetzt.

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        # // Assign(Subscr(Name('ar'), Num('1')), Num('42'))
        # Exp(Num('42'))
        SUBI SP 1;
        LOADI ACC 42;
        STOREIN SP ACC 1;
        # Ref(Global(Num('0')))
        SUBI SP 1;
        LOADI IN1 0;
        ADD IN1 DS;
        STOREIN SP IN1 1;
        # Exp(Num('1'))
        SUBI SP 1;
        LOADI ACC 1;
        STOREIN SP ACC 1;
        # Ref(Subscr(Stack(Num('2')), Stack(Num('1'))))
```

[32] Ist in Tabelle ?? genauer beschrieben ist

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

**Code 2.21:** *RETI-Blocks Pass für Zuweisung an Feldindex.*

# Appendix

## RETI Architektur Details

| Typ | Modus | Befehl | Wirkung |
|---|---|---|---|
| 01 | 00 | LOAD D i | $D := M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$ |
| 01 | 01 | LOADIN S D i | $D := M(\langle S \rangle + i), \langle PC \rangle := \langle PC \rangle + 1$ |
| 01 | 11 | LOADI D i | $D := 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$, bei $D = PC$ wird der PC nicht inkrementiert |
| 10 | 00 | STORE S i | $M(\langle i \rangle) := S, \langle PC \rangle := \langle PC \rangle + 1$ |
| 10 | 01 | STOREIN D S i | $M(\langle D \rangle + i) := S, \langle PC \rangle := \langle PC \rangle + 1$ |
| 10 | 11 | MOVE S D | $D := S, \langle PC \rangle := \langle PC \rangle + 1$, Move: Bei $D = PC$ wird der PC nicht inkrementiert |

**Tabelle 3.1:** *Load und Store Befehle.*

| Typ | M | RO | F | Befehl | Wirkung |
|---|---|---|---|---|---|
| 00 | 0 | 0 | 000 | ADDI D i | $[D] := [D] + [i], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 001 | SUBI D i | $[D] := [D] - [i], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 010 | MULI D i | $[D] := [D] * [i], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 011 | DIVI D i | $[D] := [D] / [i], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 100 | MODI D i | $[D] := [D] \% [i], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 101 | OPLUSI D i | $[D] := [D] \oplus 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 110 | ORI D i | $[D] := [D] \vee 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 0 | 0 | 101 | ANDI D i | $[D] := [D] \wedge 0^{10}i, \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 000 | ADD D i | $[D] := [D] + [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 001 | SUB D i | $[D] := [D] - [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 010 | MUL D i | $[D] := [D] * [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 011 | DIV D i | $[D] := [D] / [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 100 | MOD D i | $[D] := [D] \% [M(\langle i \rangle)], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 101 | OPLUS D i | $D := D \oplus M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 110 | OR D i | $D := D \vee M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | 1 | 0 | 101 | AND D i | $D := D \wedge M(\langle i \rangle), \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 000 | ADD D S | $[D] := [D] + [S], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 001 | SUB D S | $[D] := [D] - [S], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 010 | MUL D S | $[D] := [D] * [S], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 011 | DIV D S | $[D] := [D] / [S], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 100 | MOD D S | $[D] := [D] \% [S], \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 101 | OPLUS D S | $D := D \oplus S, \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 110 | OR D S | $D := D \vee S, \langle PC \rangle := \langle PC \rangle + 1$ |
| 00 | $*$ | 1 | 101 | AND D S | $D := D \wedge S, \langle PC \rangle := \langle PC \rangle + 1$ |

**Tabelle 3.2:** *Compute Befehle.*

| Type | Condition | J | Befehl | Wirkung |
|---|---|---|---|---|
| 11 | 000 | 00 | NOP | $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 001 | 00 | JUMP$_>$i | Falls $[ACC] > 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 010 | 00 | JUMP$_=$i | Falls $[ACC] = 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 011 | 00 | JUMP$_\geq$i | Falls $[ACC] \geq 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 100 | 00 | JUMP$_<$i | Falls $[ACC] < 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 101 | 00 | JUMP$_\neq$i | Falls $[ACC] \neq 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ |
| 11 | 110 | 00 | JUMP$_\leq$i | Falls $[ACC] \leq 0$: $\langle PC\rangle := \langle PC\rangle + [i]$, sonst: $\langle PC\rangle := \langle PC\rangle + 1$ $\langle PC\rangle := \langle PC\rangle + [i]$ |
| 11 | 111 | 00 | JUMPi | $\langle PC\rangle := \langle PC\rangle + [i]$ |
| 11 | * | 01 | INT i | $\langle PC\rangle := IVT[i]$ Interrupt Nr.i wird Ausgeführt |
| 11 | * | 10 | RTI | **Rücksprungadresse** vom **Stack** entfernt, in PC geladen, Wechsel in **Usermodus** |

**Tabelle 3.3:** *Jump Befehle.*

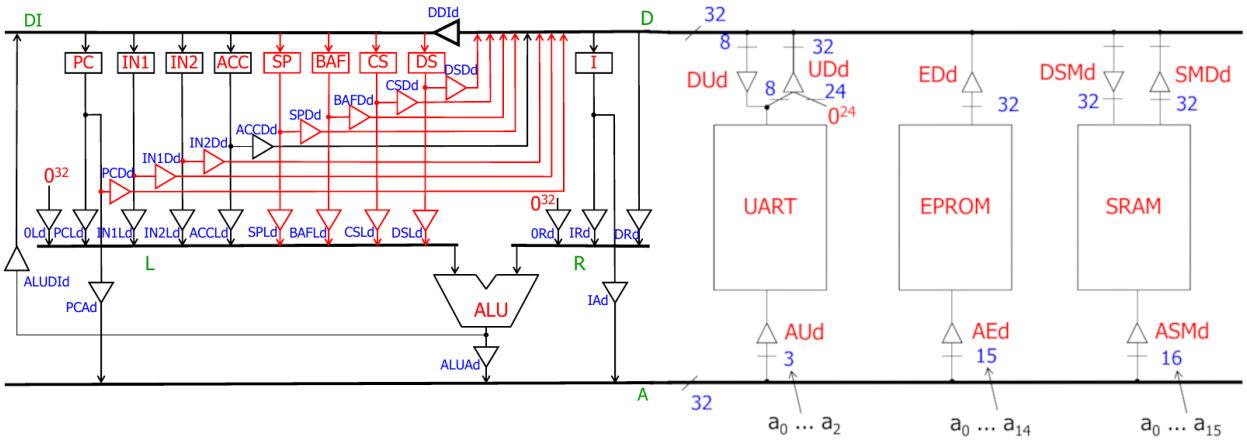



**Abbildung 3.1:** *Datenpfade der RETI-Architektur.*

## Sonstige Definitionen

Im Folgenden sind einige Definitionen aufgelistet, die zur **Erklärung** der Vorgehensweise zur Implementierung eines **üblichen Compilers** referenziert werden, aber **nichts** mit dem Vorgehen zur Implementierung des **PicoC-Compilers** zu tuen haben.

**Definition 3.1: Assemblersprache (bzw. engl. Assembly Language)**

*Eine sehr **hardwarenahe** Programmiersprache, deren **Befehle** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen[a] haben. Viele **Befehle** haben eine ähnliche übliche Struktur* `Operation <Operanden>`*, mit einer **Operation**, die einem **Opcode** eines Maschinenbefehls bezeichnet und keinen oder mehreren **Operanden**, wie die späteren Maschinenbefehle, denen sie entsprechen. Allerdings gibt es oftmals noch viel „syntaktischen Zucker“ innerhalb[b] der Befehle und*

*drumherum*[c].[d]

[a]Befehle der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Befehle** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.
[b]Z.B. erlaubt die Assemblersprache des **GCC** für die $X_{86\_64}$**-Architektur** für manche Operanden die Syntax `n(%r)`, die einen **Speicherzugriff** mit **Offset** *n* zur Adresse, die im **Register** `%r` steht durchführt, wobei z.B. die Klammern `()` usw. nur „syntaktischer Zucker“ sind und natürlich nicht mitkodiert werden.
[c]Z.B. sind im $X_{86\_64}$ Assembler die Befehle in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels `jmp <label>` gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.
[d]P. D. P. Scholl, „Einführung in Embedded Systems“.

**Anmerkung**

Ein **Assembler** (Definition 3.2) ist in üblichen Compilern in einer bestimmten Form meist schon integriert, da Compiler üblicherweise direkt **Maschinencode** bzw. **Objectcode** (Definition 3.3) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur die Ausgabe liefern, welche er in den allermeisten Fällen haben will, nämlich den **Maschinencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 3.4) zu Maschienencode zusammengesetzt wird ausführbar ist.

**Definition 3.2: Assembler**

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschinensprache** geschrieben ist.*[a]

[a]P. D. P. Scholl, „Einführung in Embedded Systems“.

**Definition 3.3: Objectcode**

*Bei Komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen wird häufig **Objectcode** erzeugt, der neben der Folge von Maschinenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.*[a]

[a]P. D. P. Scholl, „Einführung in Embedded Systems“.

**Definition 3.4: Linker**

*Programm, dass **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt** bzw. zusammenfügt, sodass unter anderem kein vermeidbarer **doppelter** Code darin vorkommt.*[a]

[a]P. D. P. Scholl, „Einführung in Embedded Systems“.

**Definition 3.5: Transpiler (bzw. Source-to-source Compiler)**

*Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten*[a][b]

[a]Die Programmiersprache **TypeScript** will als **Obermenge** von **JavaScript** die Sprachhe Javascript **erweitern** und gleichzeitig die **syntaktischen Mittel** von JavaScript unterstützen. Daher bietet es sich Typescript zu Javascript zu **transpilieren**.
[b]Thiemann, „Compilerbau“.

**Definition 3.6: Rekursiver Abstieg**

*Es wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die **Produktionen** dieses Nicht-Terminalsymbols umsetzt. **Prozeduren** rufen sich dabei wechselseitig gegenseitig entsprechend der Produktionsregeln auf, falls eine Produktionsregel ein entsprechendes **Nicht-Terminalsymbol** enthält.*

**Anmerkung**

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 3.7) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 3.6) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschafft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist[a].

[a] Mehr **Erklärung** hierzu findet sich im Unterkapitel 2.4.

**Definition 3.7: LL(k)-Grammatik**

*Eine Grammatik ist **LL(k)** für $k \in \mathbb{N}$, falls jeder Ableitungsschritt eindeutig durch die nächsten $k$ **Token** des **Eingabeworts** zu bestimmen ist[a]. Dabei steht **LL** für left-to-right und leftmost-derivation, da das **Eingabewort** von **links nach rechts** geparsed und immer **Linksableitungen** genommen werden müssen[b], damit die obige Bedingung mit den **nächsten** $k$ Symbolen gilt.[c]*

[a] Das wird auch als **Lookahead** von $k$ bezeichnet.
[b] Wobei sich das mit den **Linksableitungen** automatisch ergibt, wenn man das Eingabewort von **links-nach-rechts** parsed und jeder der nächsten $k$ **Ableitungsschritte** eindeutig sein soll.
[c] Nebel, „Theoretische Informatik“.

**Definition 3.8: Earley Recognizer**

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

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

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

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

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

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

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

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

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

   *bis:*
   - *der **Zustand** $(A ::= \beta\bullet, 0)$ in der **Zustandsmenge** $Z(n)$ auftaucht, wobei $A$ das **Startsymbol** $S$ ist $\Rightarrow w \in L(G_{Parse})$.*
   - ***keine Zustände** mehr hinzugefügt werden können $\Rightarrow w \notin L(G_{Parse})$.*

[a] Jay Earley, „An efiicient context-free parsing“.
[b] **Erklärweise** wurde von der Webseite *Earley parser* übernommen.
[c] *Earley Parser*.
[d] $L(G_{Parse})$ ist die **Sprache**, welche durch die **Konkrete Grammatik** $G_{Parse}$ beschrieben wird.
[e] Also eine Folge von **Terminalsymbolen** und **Nicht-Terminalsymbolen**.

**Definition 3.9: Liveness Analyse**

*Findet heraus, welche **Variablen** in welchen **Regionen** eines Programmes **verwendet** werden.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.10: Live Variable**

*Eine Location, deren momentaner Wert **später** im Programmablauf noch **verwendet** wird. Man sagt auch die Location ist **live**.*[a][b]

[a] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Live Variable**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.11: Graph Coloring**

*Problem bei dem den **Knoten** eines Graphen[a] **Zahlen**[b] zugewiesen werden sollen, sodass **keine** zwei **adjazente Knoten** die **gleiche Zahl** haben und **möglichst wenige** unterschiedliche Zahlen gebraucht werden.*[c d]

[a] In Bezug zu Compilerbau ein **Ungerichteter Graph**.
[b] Bzw. **Farben**.
[c] Es gibt leider **kein** allgemein verwendetes **deutsches** Wort für **Graph Coloring**.
[d] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.12: Interference Graph**

*Ein **ungerichteter Graph** mit **Locations** als **Knoten**, der eine **Kante** zwischen zwei Locations hat, wenn es sich bei beiden Locations **zu dem Zeitpunkt** um **Live Locations** handelt. In Bezug auf **Graph Coloring** bedeutet eine **Kante**, dass diese zwei Locations **nicht** die **gleiche Zahl**[a] zugewiesen bekommen dürfen.*[b]

[a] Bzw. **Farbe**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.13: Kontrollflussgraph**

***Gerichteter Graph**, der den Kontrollfluss eines Programmes beschreibt.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.14: Kontrollfluss**

*Die **Reihenfolge** in der z.B. **Statements**, **Funktionsaufrufe** usw. eines Programmes ausgewertet werden*[a].

[a] Man geht hier von einem **imperativen** Programm aus.

**Definition 3.15: Kontrollflussanalyse**

***Analyse** des **Kontrollflusses** (Defintion 3.14) eines **Programmes**, um herauszufinden zwischen welchen Teilen des Programms **Daten ausgetauscht** werden und welche **Abhängigkeiten** sich daraus ergeben.*

*Der **simpelste Ansatz** ist es in einen Kontrollflussgraph **iterativ** einen Algorithmus*[a] *anzuwenden, bis sich an den Werten der Knoten **nichts** mehr **ändert***[b].[c]

[a] Im Bezug zu Compilerbau die **Linveness Analayse**.
[b] Bis diese sich **stabilisiert** haben
[c] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 3.16: Two-Space Copying Collector**

*Ein **Garbabe Collector** bei dem der **Heap** in `FromSpace` und `ToSpace` unterteilt wird und bei **nicht ausreichendem** Speicherplatz auf dem **Heap** alle Variablen, die in Zukunft noch verwendet werden vom `FromSpace` zum `ToSpace` kopiert werden. Der aktuelle `ToSpace` wird danach zum neuen `FromSpace` und der aktuelle `FromSpace` wird danach zum neuen `ToSpace`.*[a]

[a] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

## Bootstrapping

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ (Defintion 3.17) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache $L_Python$, in der der momentane Compiler $C_{PicoC}$ für $L_{PicoC}$ implementiert ist und die Unabhängigkeit von einer **anderen Maschine**, die bisher immer für das Cross-Compiling notwendig war erreicht werden.

**Definition 3.17: Self-compiling Compiler**

*Compiler $C^w_w$, der in der Sprache $L_w$ **geschrieben** ist, die er **selbst** kompiliert. Also ein Compiler, der sich **selbst** kompilieren kann.*[a]

[a] J. Earley und Sturgis, „A formalism for translator interactions“.

Will man nun für eine Maschine $M_{RETI}$, auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 3.20) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ in der Sprache $L_{PicoC}$ geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**[1] reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auf der **Maschine** $M_{RETI}$ zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der mit der Maschinensprache $B_{RETI}$ läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ nicht selbst komplett in der Maschinensprache $B_{RETI}$ schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** $C^{Python}_{PicoC}$, den man bereits in der Programmiersprache $L_{Python}$ implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 3.19) darstellt, auf einer anderen Maschine $M_{other}$ dafür nutzt, damit dieser den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ für die Maschine $M_{RETI}$ kompiliert bzw. **bootstraped** und man den kompilierten **RETI-Maschiendencode** dann einfach von der Maschine $M_{other}$ auf die Maschine $M_{RETI}$ kopiert.[2]

[1] Beschreibt die Situation, wenn ein System sich selbst als **Abhängigkeit** hat, damit es überhaupt einen **Anfang** für dieses System geben kann. Dafür steht das Problem mit der **Henne** und dem **Ei** sinnbildlich, da hier die Frage ist, wie das ganze seinen Anfang genommen hat, da beides **zirkular** voneinander abhängt.

[2] Im Fall, dass auf der Maschine $M_{RETI}$ die Programmiersprache $L_{Python}$ bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auch mithife des **Cross-Compilers** $C^{Python}_{PicoC}$ als **externe Entität** und der Programmiersprache $L_{Python}$ auf der Maschine $M_{RETI}$ selbst kompiliert werden.

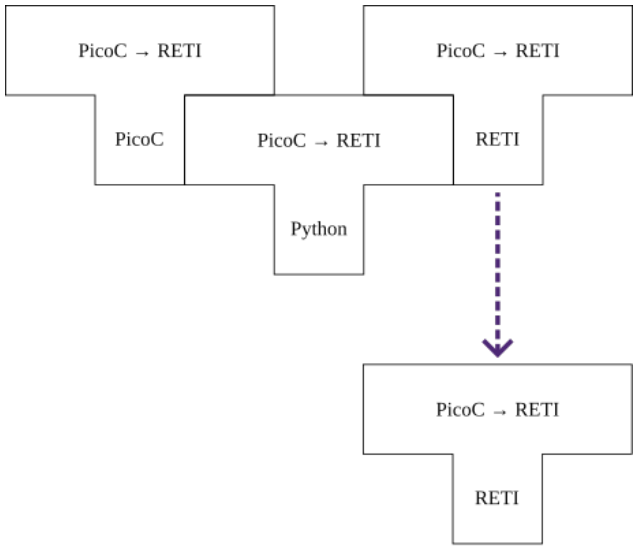



**Abbildung 3.2:** *Cross-Compiler als Bootstrap Compiler.*

**Anmerkung**

Einen ersten **minimalen Compiler** $C_{2\_w\_min}$ für eine Maschine $M_2$ und Wunschsprache $L_w$ kann man entweder mittels eines **externen Bootstrap Compilers** $C_w^o$ kompilieren[a] oder man schreibt ihn direkt in der **Maschinensprache** $B_2$ bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** $A_2$.

Die letzte Option wäre allerdings nur beim allerersten Compiler $C_{first}$ für eine allererste **abstraktere Programmiersprache** $L_{first}$ mit Schleifen, Verzweigungen usw. notwendig gewesen. Ansonsten hätte man immer eine Kette, die beim allersten Compiler $C_{first}$ anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

[a] In diesem Fall, dem **Cross-Compiler** $C_{PicoC}^{Python}$.

**Definition 3.18: Minimaler Compiler**

*Compiler $C_{w\_min}$, der nur die **notwendigsten Funktionalitäten** einer Wunschsprache $L_w$, wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** $C_w^w$ oder einer **ersten Version** $C_{w_i}^{w_i}$ des Self-compiling Compilers $C_w^w$ wichtig sind.*[a b]

[a] Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.
[b] Thiemann, „Compilerbau“.

**Definition 3.19: Boostrap Compiler**

*Compiler $C_w^o$, der es ermöglicht einen **Self-compiling Compiler** $C_w^w$ zu **boostrapen**, indem der Self-compiling Compiler $C_w^w$ mit dem **Bootstrap Compiler** $C_w^o$ **kompiliert** wird*[a]*. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** $C_w^w$ bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.*[b]

[a] Dabei kann es sich um einen **lokal** auf der Maschine selbst laufenden Compiler oder auch um einen **Cross-Compiler**

handeln.
[b] Thiemann, „Compilerbau“.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der einen **minimalen Compiler** (Definition 3.18) für eine Teilmenge der **Programmiersprache** C bzw. $L_C$ darstellt, könnte man auch noch weitere Teile der Programmiersprache $C$ bzw. $L_C$ für die Maschine $M_{RETI}$ mittels **Bootstrapping** implementieren.[3]

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine $M_{RETI}$ selbst, aufbauend auf diesem **minimalen Compiler** $C^{PicoC}_{RETI\_PicoC}$, wie in Subdefinition 3.20.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** $C_C$ weiterentwickelt.

**Definition 3.20: Bootstrapping**

*Wenn man einen **Self-compiling Compiler** $C^w_w$ einer Wunschsprache $L_w$ auf einer **Zielmaschine** $M$ zum laufen bringt[abcd]. Dabei ist die Art von **Bootstrapping** in 3.20.1 nochmal gesondert hervorzuheben:*

**3.20.1:** *Wenn man die **aktuelle Version** eines **Self-compiling Compilers** $C^{w_i}_{w_i}$ der Wunschsprache $L_{w_i}$ mithilfe von **früheren Versionen** seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ kompiliert wird und schafft es so **iterativ** immer umfangreichere Compiler zu bauen.[efg]*

[a] Z.B. mithilfe eines **Bootstrap Compilers**.
[b] Der Begriff hat seinen Ursprung in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügengeschichten des Freiherrn von Münchhausen** bekannten Redewendung „sich am eigenen Schopf aus dem Sumpf ziehen“entspricht.
[c] Hat man einmal einen solchen **Self-compiling Compiler** $C^w_w$ auf der Maschine $M$ zum laufen gebracht, so kann man den Compiler auf der Maschine $M$ weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache $L_o$, in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.
[d] Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute **Probe aufs Exempel** darstellen, dass der Compiler auch wirklich funktioniert.
[e] Es ist hierbei theoretisch nicht notwendig den **letzten** Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ für das Kompilieren des **neuen** Self-compiling Compilers $C^{w_i}_{w_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C^{w_{i-3}}_{w_{i-3}}$ auch bereits alle Funktionalitäten, die beim Schreiben des **Self-compiling Compilers** $C^w_w$ verwendet werden kompilieren kann.
[f] Der Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, welches bei Bedarf dann **Systemsoftware**, Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber. und **Anwendungssoftware**, Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt, lädt.
[g] J. Earley und Sturgis, „A formalism for translator interactions“.

[3] Natürlich könnte man aber auch einfach den **Cross-Compiler** $C^{Python}_{PicoC}$ um weitere Funktionalitäten von $L_C$ erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache $L_{Python}$.

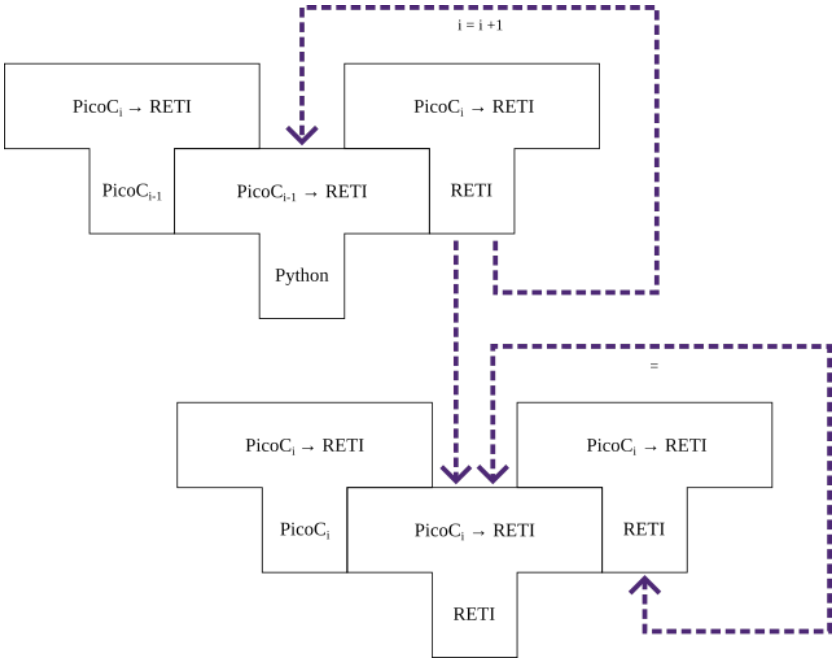



**Abbildung 3.3:** *Iteratives Bootstrapping.*

**Anmerkung**

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 3.20.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache $L_{w_i}$ geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 3.20.1 eine frühere Version $C_{w_{i-1}}^{w_{i-1}}$ nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

# Literatur

## Online


- *A-Normalization: Why and How (with code)*. URL: https://matt.might.net/articles/a-normalization/ (besucht am 23. 07. 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).
- Developers, Inkscape Website. *Draw Freely — Inkscape*. URL: https://inkscape.org/ (besucht am 03. 08. 2022).
- *Earley Parser*. URL: https://rahul.gopinath.org/post/2021/02/06/earley-parsing/ (besucht am 20. 06. 2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/errors-in-cc/ (besucht am 10. 05. 2022).
- *GCC, the GNU Compiler Collection - GNU Project*. URL: https://gcc.gnu.org/ (besucht am 13. 07. 2022).
- *JSON parser - Tutorial — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (besucht am 09. 07. 2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13. 04. 2022).
- *Parsing Expressions · Crafting Interpreters*. URL: https://www.craftinginterpreters.com/parsing-expressions.html (besucht am 09. 07. 2022).
- *Transformers & Visitors — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09. 07. 2022).
- *Variablen in C und C++, Deklaration und Definition — Coder-Welten.de*. URL: https://www.coder-welten.de/einstieg/variablen-in-c-3.html (besucht am 11. 08. 2022).
- *What is Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 2022).
- *What is the difference between function prototype and function signature?* SoloLearn. URL: https://www.sololearn.com/Discuss/171026/what-is-the-difference-between-function-prototype-and-function-signature/ (besucht am 18. 07. 2022).
- *What is Top-Down Parsing?* URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22. 06. 2022).

## Bücher


- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).
- LeFever, Lee. *The Art of Explanation: Making your Ideas, Products, and Services Easier to Understand*. 1. Aufl. Wiley, 20. Nov. 2012.


## Artikel


- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: 10.1145/355598.362740.
- 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


- Bast, Prof. Dr. Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020 (besucht am 09. 07. 2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html (besucht am 09. 07. 2022).
- Scholl, Prof. Dr. Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- — „Technische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 3. Aug. 2022. (Besucht am 03. 08. 2022).
- Scholl, Prof. Dr. Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://earth.informatik.uni-freiburg.de/uploads/es-2122/ (besucht am 09. 07. 2022).
- Thiemann, Prof. Dr. Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/ (besucht am 09. 07. 2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/ (besucht am 09. 07. 2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtvl (besucht am 19. 07. 2022).

## Sonstige Quellen


- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23. 07. 2022).
- *Earley parser*. In: *Wikipedia*. Page Version ID: 1090848932. 31. Mai 2022. URL: https://en.wikipedia.org/w/index.php?title=Earley_parser&oldid=1090848932 (besucht am 15. 08. 2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).
- Nemec, Devin. *copy_file_to_another_repo_action*. original-date: 2020-08-24T19:25:58Z. 27. Juli 2022. URL: https://github.com/dmnemec/copy_file_to_another_repo_action (besucht am 03. 08. 2022).
- Ueda, Takahiro. *Makefile for LaTeX*. original-date: 2018-07-06T15:01:24Z. 10. Mai 2022. URL: https://github.com/tueda/makefile4latex (besucht am 03. 08. 2022).