

---

ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

# PicoC-Compiler

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

BACHELORARBEIT

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

*Author:*  
Jürgen Mattheis

*Gutachter:*  
Prof. Dr. Scholl

*Betreuung:*  
M.Sc. Seufert

---

Eine Bachelorarbeit am Lehrstuhl für  
Betriebssysteme

---

---

---

## **ERKLÄRUNG**

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

---

---

# Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	V
Grammatikverzeichnis	VI
<b>1 Einführung</b>	<b>1</b>
1.1 Compiler und Interpreter	1
1.1.1 T-Diagramme	4
1.2 Formale Sprachen	6
1.2.1 Ableitungen	9
1.2.2 Mehrdeutige Grammatiken	12
1.2.3 Präzidenz und Assoziativität	13
1.3 Lexikalische Analyse	14
1.4 Syntaktische Analyse	17
1.5 Code Generierung	23
1.5.1 Monadische Normalform	24
1.5.2 A-Normalform	25
1.5.3 Ausgabe des Maschinencodes	27
1.6 Fehlermeldungen	28
1.6.1 Kategorien von Fehlermeldungen	28
<b>2 Ergebnisse und Ausblick</b>	<b>29</b>
2.1 Funktionsumfang	29
2.1.1 Kommandozeilenoptionen	29
2.1.2 Shell-Mode	32
2.1.3 Show-Mode	33
2.2 Qualitätssicherung	35
2.3 Erweiterungsideen	39
<b>Literatur</b>	<b>A</b>

---

---

---

# Abbildungsverzeichnis

1.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	6
1.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	6
1.3	Veranschaulichung von Linksassoziativität und Rechtsassoziativität . . . . .	13
1.4	Veranschaulichung von Präzidenz . . . . .	14
1.5	Veranschaulichung der Lexikalischen Analyse . . . . .	17
1.6	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum. . . . .	21
1.7	Veranschaulichung der Syntaktischen Analyse . . . . .	22
1.8	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten . . . . .	25
1.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen . . . . .	27
2.1	Show-Mode in der Verwendung . . . . .	35

---

---

# Codeverzeichnis

2.1	Shellaufruf und die Befehle <code>compile</code> und <code>quit</code> . . . . .	32
2.2	Shell-Mode und der Befehl <code>most_used</code> . . . . .	33
2.3	Typischer Test . . . . .	37
2.4	Testdurchlauf . . . . .	39
2.5	Beispiel für Tail Call . . . . .	41

---

---

# Tabellenverzeichnis

2.1	Kommandozeilenoptionen . . . . .	31
2.2	Makefileoptionen . . . . .	34
2.3	Testkategorien . . . . .	36

---

---

# Definitionsverzeichnis

1.1	Interpreter . . . . .	1
1.2	Compiler . . . . .	1
1.3	Maschiensprache . . . . .	2
1.4	Assemblersprache (bzw. engl. Assembly Language) . . . . .	2
1.5	Assembler . . . . .	3
1.6	Objectcode . . . . .	3
1.7	Linker . . . . .	3
1.8	Immediate . . . . .	3
1.9	Transpiler (bzw. Source-to-source Compiler) . . . . .	4
1.10	Cross-Compiler . . . . .	4
1.11	T-Diagram Programm . . . . .	4
1.12	T-Diagram Übersetzer (bzw. eng. Translator) . . . . .	5
1.13	T-Diagram Interpreter . . . . .	5
1.14	T-Diagram Maschine . . . . .	5
1.15	Symbol . . . . .	6
1.16	Alphabet . . . . .	7
1.17	Wort . . . . .	7
1.18	Formale Sprache . . . . .	7
1.19	Syntax . . . . .	7
1.20	Semantik . . . . .	7
1.21	Formale Grammatik . . . . .	7
1.22	Chromsky Hierarchie . . . . .	8
1.23	Reguläre Grammatik . . . . .	9
1.24	Kontextfreie Grammatik . . . . .	9
1.25	Wortproblem . . . . .	9
1.26	1-Schritt-Ableitungsrelation . . . . .	10
1.27	Ableitungsrelation . . . . .	10
1.28	Links- und Rechtsableitungableitung . . . . .	10
1.29	Linksrekursive Grammatiken . . . . .	10
1.30	Formaler Ableitungsbaum . . . . .	11
1.31	Mehrdeutige Grammatik . . . . .	12
1.32	LL(k)-Grammatik . . . . .	12
1.33	Assoziativität . . . . .	13
1.34	Präzidenz . . . . .	13
1.35	Pipe-Filter Architekturpattern . . . . .	14
1.36	Pattern . . . . .	14
1.37	Lexeme . . . . .	14
1.38	Lexer (bzw. Scanner oder auch Tokenizer) . . . . .	15
1.39	Bezeichner (bzw. Identifier) . . . . .	15
1.40	Literal . . . . .	16
1.41	Konkrete Syntax . . . . .	17
1.42	Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree) . . . . .	17
1.43	Parser . . . . .	17
1.44	Recognizer (bzw. Erkennen) . . . . .	18
1.45	Rekursiver Abstieg . . . . .	19
1.46	Transformer . . . . .	20
1.47	Visitor . . . . .	20

---

---

1.48	Abstrakte Syntax . . . . .	20
1.49	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST) . . . . .	20
1.50	Pass . . . . .	23
1.51	Reiner Ausdruck (bzw. engl. pure expression) . . . . .	24
1.52	Unreiner Ausdruck . . . . .	24
1.53	Monadische Normalform (bzw. engl. monadic normal form) . . . . .	24
1.54	Location . . . . .	25
1.55	Atomarer Ausdruck . . . . .	26
1.56	Komplexer Ausdruck . . . . .	26
1.57	A-Normalform (ANF) . . . . .	26
1.58	Fehlermeldung . . . . .	28

---



---

---

# Grammatikverzeichnis

1.1	Produktionen des Ableitungsbaums . . . . .	11
-----	--	----

# 1 Einführung

## 1.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 1.2) und eines **Interpreters** (Definition 1.1), 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**<sup>1</sup> und von **Tests** die **Beziehungen** in 1.2.1 zu belegen (siehe Subkapitel 2.2).

### Definition 1.1: Interpreter

*Interpretiert die **Instructions** bzw. **Statements** eines Programmes  $P$  direkt.*

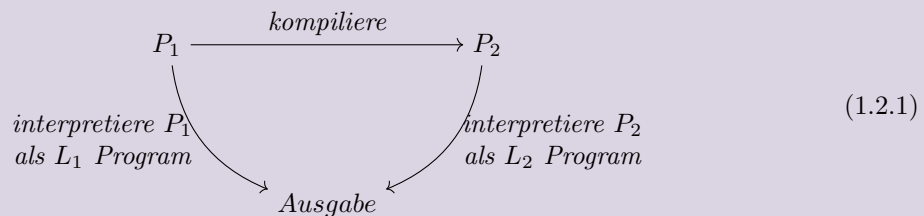
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstrakter Syntaxbaum** (Definition 1.49) und führt je nach Komposition der **Nodes** des Abstrakter Syntaxbaum, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

### Definition 1.2: 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 Program  $P_2$  übersetzt wird, dass bei beiden Programmen, wenn sie von **Interpretern** ihrer jeweiligen Sprachen  $L_1$  und  $L_2$  **interpretiert** werden, die gleiche **Ausgabe** rauskommt, wie es in Diagramm 1.2.1 dargestellt ist. Also beide Programme  $P_1$  und  $P_2$  die gleiche **Semantik** (Definition 1.20) haben und sich nur **syntaktisch** (Definition 1.19) durch die Sprachen  $L_1$  und  $L_2$ , in denen sie geschrieben stehen unterscheiden.<sup>a</sup>



<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

<sup>1</sup>Sammlung von Compilern für Linux bzw. GNU-Linux, steht für **GNU Compiler Collection**

Im Folgenden wird ein voll ausgeschriebenener **Compiler** als  $C_{i.w.k.min}^{o-j}$  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. Fall 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  $C_{w.k}^{o-j}$  geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition ??) kann man das als  $C_{min}$  schreiben.

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschinenenncode**, der in **Maschinensprache** (Definition 1.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 1.9) oder **Cross-Compiler** (Definition 1.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 1.4) voneinander zu unterscheiden.

### Definition 1.3: 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**, genauer gesagt 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 designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.<sup>a,b</sup>*

<sup>a</sup>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 1.8) haben.

<sup>b</sup>P. D. C. Scholl, „Betriebssysteme“.

### Definition 1.4: Assemblersprache (bzw. engl. Assembly Language)

*Eine sehr **hardwarenahe** Programmiersprache, deren **Instructions** eine starke Entsprechung zu bestimmten Maschinenbefehlen bzw. Folgen von Maschinenbefehlen<sup>a</sup> haben. Viele **Instructions** 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<sup>b</sup> der Instructions und drumherum<sup>c,d</sup>.*

<sup>a</sup>Instructions der Assemblersprache, die mehreren Maschinenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

<sup>b</sup>Z.B. erlaubt die Assemblersprache des **GCC** für die **X86\_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 mitcodiert werden.

<sup>c</sup>Z.B. sind im X86\_64 Assembler die Instructions 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.

<sup>d</sup>P. D. P. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 1.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenenncode** bzw. **Objectcode** (Definition 1.6) 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 den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschinenenncode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 1.7) zu Maschienncode zusammengesetzt wird ausführbar

ist.

#### Definition 1.5: Assembler

Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschinencode** bzw. **Objectcode** in **binärer Repräsentation**, der in **Maschiensprache** geschrieben ist.<sup>a</sup>

<sup>a</sup>P. D. P. Scholl, „Einführung in Embedded Systems“.

#### Definition 1.6: 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.<sup>a</sup>

<sup>a</sup>P. D. P. Scholl, „Einführung in Embedded Systems“.

#### Definition 1.7: Linker

Programm, das **Objektcode** aus mehreren Objektdateien zu ausführbarem **Maschinencode** in eine ausführbare Datei oder Bibliotheksdatei **linkt**, sodass unter anderem kein vermeidbarer **doppelter Code** darin vorkommt.<sup>a</sup>

<sup>a</sup>P. D. P. Scholl, „Einführung in Embedded Systems“.

Der **Maschinencode**, denn 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 ausgeschriebenem RETI-Operationen, RETI-Registern und Immediates (Definition 1.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschinencode, denn 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<sup>2</sup>.

#### Definition 1.8: 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, **beschränkter** ist als bei sonstigen Werten innerhalb des Hauptspeichers, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.<sup>a</sup>

<sup>a</sup>Ljohhuh, *What is an immediate value?*

<sup>2</sup>Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschinencode in z.B. **UTF-8 Codierung** ausführen kann, wäre dagegen unnötig kompliziert und aufwändig, da Hardware **binär** arbeitet und man dieser daher lieber direkt die binär codierten 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.

**Definition 1.9: Transpiler (bzw. Source-to-source Compiler)**

Kompiliert zwischen Sprachen, die ungefähr auf dem *gleichen* Level an *Abstraktion* arbeiten<sup>ab</sup>

<sup>a</sup>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**.

<sup>b</sup>Thiemann, „Compilerbau“.

**Definition 1.10: 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 *Maschinen Sprachen*  $B_1$  und  $B_2$  haben.<sup>ab</sup>

<sup>a</sup>Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler**  $C_{PicoC}^{Python}$ .

<sup>b</sup>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 **Maschinen Sprache**  $B_2$  einer Zielmaschine  $M_2$  läuft.<sup>3</sup>

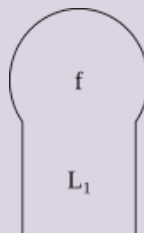
**1.1.1 T-Diagramme**

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus dem Paper 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 Programm (Definition 1.11), einen Übersetzer (Definition 1.12), einen Interpreter (Definition 1.13) und eine Maschine (Definition 1.14) zusammen.

**Definition 1.11: T-Diagram Programm**

Repräsentiert ein *Programm*, dass in der *Sprache*  $L_1$  geschrieben ist und die *Funktion*  $f$  berechnet.<sup>a</sup>



<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

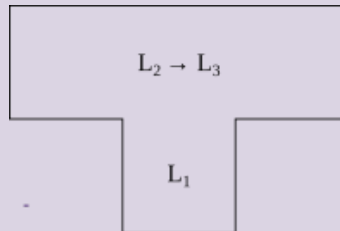
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein  $L$  dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 1.11 also reichen einfach eine 1 hinzuschreiben.

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

**Definition 1.12: T-Diagramm 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**<sup>a</sup> die **Beziehungen** in 1.2.1.<sup>b</sup>

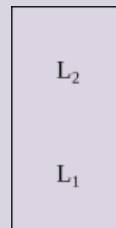


<sup>a</sup>Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

<sup>b</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 1.13: T-Diagramm Interpreter**

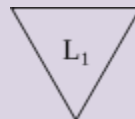
Repräsentiert einen **Interpreter**, der in der **Sprache**  $L_1$  geschrieben ist und **Programme** in der **Sprache**  $L_2$  interpretiert.<sup>a</sup>



<sup>a</sup>Earley und Sturgis, „A formalism for translator interactions“.

**Definition 1.14: T-Diagramm Maschine**

Repräsentiert eine **Maschine**, welche ein **Programm** in **Maschinensprache**  $L_1$  ausführt.<sup>a,b</sup>



<sup>a</sup>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).

<sup>b</sup>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 Adjazents** für **Interpretation** und **horizontale Adjazents** für **Übersetzung** steht.

Sowohl **horizontale** als auch **vertikale Adjazents** lassen sich, wie man in den Abbildungen 1.1 und 1.2 erkennen kann zusammenfassen.

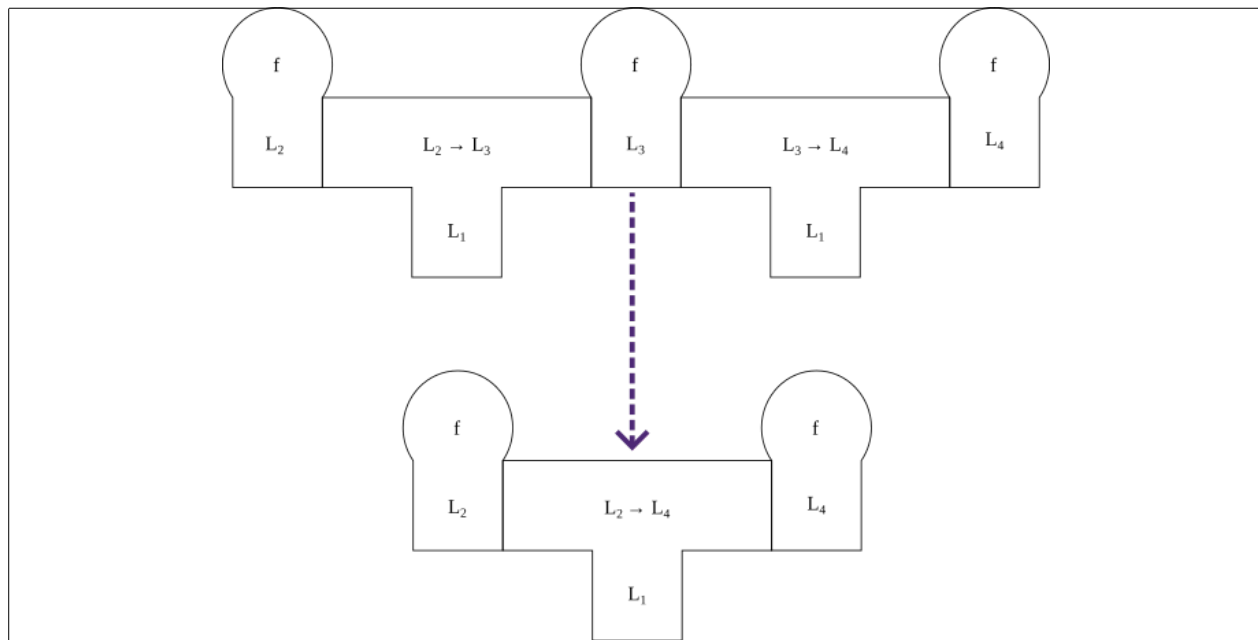


Abbildung 1.1: Horizontale Übersetzungszwischenschritte zusammenfassen

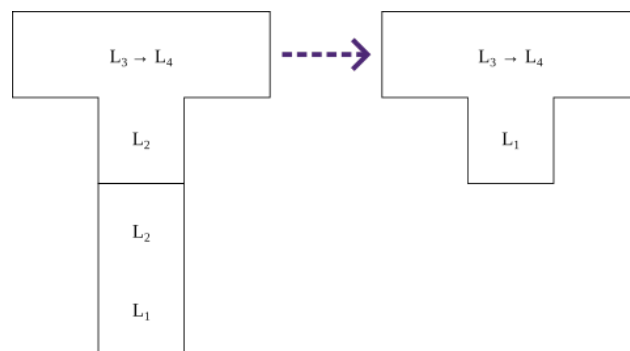


Abbildung 1.2: Vertikale Interpretierungszwischenschritte zusammenfassen

## 1.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 1.18) zu tun, 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** (Definition 1.15), **Alphabet** (Definition 1.16), **Wort** (Definition 1.17) usw. beinhaltet vorher eingeführt zu haben.

### Definition 1.15: Symbol

„Ein Symbol ist ein **Element** eines **Alphabets**  $\Sigma$ .“<sup>a</sup>

<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 1.16: Alphabet**

„Ein Alphabet ist eine **endliche, nicht-leere** Menge aus **Symbolen** (Definition 1.15).“<sup>a</sup>

<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 1.17: Wort**

„Ein Wort  $w = a_1 \dots a_n \in \Sigma^*$  ist eine **endliche Folge** von **Symbolen** aus einem **Alphabet**  $\Sigma$ .“<sup>a</sup>

<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 1.18: Formale Sprache**

„Eine **Formale Sprache** ist eine Menge von **Wörtern** (Definition 1.17) über dem **Alphabet**  $\Sigma$  (Definition 1.16).“<sup>a</sup>

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.

<sup>a</sup>Nebel, „Theoretische Informatik“.

Bei der Übersetzung eines Programmes von einer Sprache  $L_1$  zur Sprache  $L_2$  muss die **Semantik** (Definition 1.20) **gleich** bleiben. Beide Sprachen  $L_1$  und  $L_2$  haben eine **Grammatik** (Definition 1.21), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 1.19) haben.

**Definition 1.19: Syntax**

Die **Syntax** bezeichnet alles was mit dem **Aufbau** von **Formalen Sprachen** zu tun hat. Die **Grammatik** einer Sprache, aber auch die in **Natürlicher Sprache** ausgedrückten Regeln, welche den Aufbau von Wörtern einer Formalen Sprache beschreiben werden als **Syntax** bezeichnet. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben<sup>a, b</sup>

<sup>a</sup>Z.B. die **Konkrete** und **Abstrakte Syntax**, die später eingeführt werden.

<sup>b</sup>Thiemann, „Einführung in die Programmierung“.

**Definition 1.20: Semantik**

Die **Semantik** bezeichnet alles was mit der **Bedeutung** von **Formalen Sprachen** zu tun hat.<sup>a</sup>

<sup>a</sup>Thiemann, „Einführung in die Programmierung“.

**Definition 1.21: Formale Grammatik**

„Eine **Formale Grammatik** beschreibt wie **Wörter** einer **Sprache** abgeleitet werden können.“<sup>a</sup>

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:

- $N \hat{=}$  **Nicht-Terminalsymbole**.



- $\Sigma \hat{=} \textbf{Terminalsymbole}$ , wobei  $N \cap \Sigma = \emptyset^{b,c}$ .
- $P \hat{=} \text{Menge von Produktionsregeln}$   $w \rightarrow v$ , wobei  $w, v \in (N \cup \Sigma)^* \wedge w \notin \Sigma^*$ .<sup>d,e</sup>
- $S \hat{=} \textbf{Startsymbol}$ , wobei  $S \in N$ .

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

<sup>a</sup>Nebel, „Theoretische Informatik“.

<sup>b</sup>Weil mit ihnen **terminiert** wird.

<sup>c</sup>Kann auch als **Alphabet** bezeichnet werden.

<sup>d</sup> $w$  muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.

<sup>e</sup>Bzw.  $w, v \in V^* \wedge w \notin \Sigma^*$ .

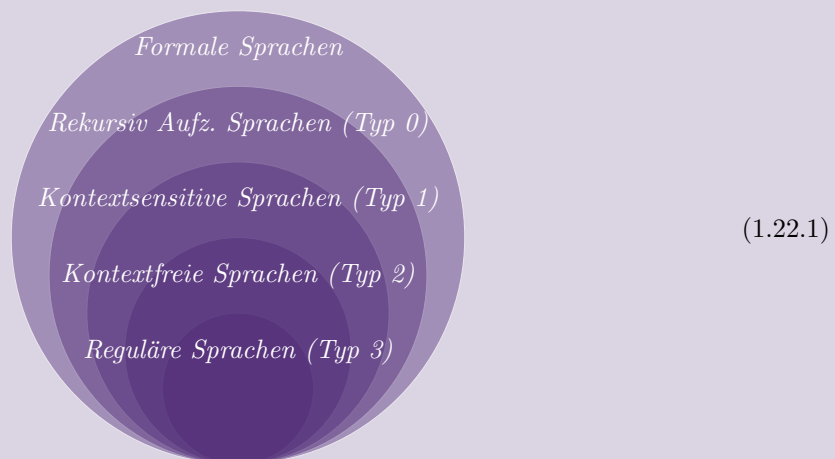
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 1.22) einteilen.

### Definition 1.22: 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 **Entscheidungsprobleme**, die in dieser Klasse **entscheidbar** bzw. **unentscheidbar** sind usw.

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

Zwischen den Sprachmengen **benachbarter Klassen** in Abbildung 1.22.1 besteht eine **echte Teilmen-genbeziehung**:  $L_3 \subset L_2 \subset L_1 \subset L_0$ . Jede **Reguläre Sprache** ist auch eine **Kontextfreie Sprache**, aber nicht jede **Kontextfreie Sprache** ist auch eine **Reguläre Sprache**.<sup>a</sup>



<sup>a</sup>Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 1.23) und **Kontextfreien Grammatiken** (Definition 1.24) beschrieben werden.

**Definition 1.23: Reguläre Grammatik**

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

$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \varepsilon \quad (1.23.1)$$

haben, wobei  $A, B$  **Nicht-Terminalsymbole** sind und  $c$  ein **Terminalsymbol** ist<sup>a,b</sup>.“<sup>c</sup>

<sup>a</sup>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.

<sup>b</sup>Dadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.

<sup>c</sup>Nebel, „Theoretische Informatik“.

**Definition 1.24: Kontextfreie Grammatik**

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

$$A \rightarrow v \quad (1.24.1)$$

haben, wobei  $A$  ein **Nicht-Terminalsymbol** ist und  $v$  ein beliebige Folge von **Grammatiksymbolen**<sup>a</sup> ist.“<sup>b</sup>

<sup>a</sup>Also eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

<sup>b</sup>Nebel, „Theoretische Informatik“.

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des **Wortproblems** (Definition 1.25). 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 1.25: Wortproblem**

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

Das Wortproblem kann durch die folgende **Indikatorfunktion**<sup>c</sup> zusammengefasst werden:

$$\mathbb{1}_L : \Sigma^* \rightarrow \{0, 1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \quad (1.25.1)$$

<sup>a</sup>Bzw. „ja“ oder „nein“ usw., es muss nicht umgedingt 1 oder 0 sein.

<sup>b</sup>Nebel, „Theoretische Informatik“.

<sup>c</sup>Auch **Charakteristische Funktion** genannt.

**1.2.1 Ableitungen**

Um sicher zu wissen, ob ein Compiler ein **Programm**<sup>4</sup> kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprache** des Compilers **abzuleiten**. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 1.26) und der normalen **Ableitungsrelation** (Definition 1.27) unterscheiden.

<sup>4</sup>Bzw. **Wort**.

**Definition 1.26: 1-Schritt-Ableitungsrelation**

„Eine **binäre Relation**  $\Rightarrow$  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 unterscheiden.

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

<sup>a</sup>Nebel, „Theoretische Informatik“.

**Definition 1.27: 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**  $u = w_1 \Rightarrow \dots \Rightarrow w_n = v$ , wobei  $n \geq 1$  und  $w_1, \dots, w_n \in (N \cup \Sigma)^*$ .“<sup>a</sup>

<sup>a</sup>Nebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**<sup>5</sup> kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 1.28) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 1.4 relevant.

**Definition 1.28: Links- und Rechtsableitung**

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

Mit diesem Vorgehen kann man jedes ableitbare Wort generieren, denn dieses Vorgehen entspricht **Tiefensuche von links-nach-rechts**.“<sup>a</sup>

<sup>a</sup>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 1.29) ist<sup>6</sup>.

**Definition 1.29: 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.<sup>a</sup>

<sup>a</sup>Parsing Expressions · Crafting Interpreters.

<sup>5</sup>Bzw. **Wort**.

<sup>6</sup>Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (siehe Unterkapitel 1.2.2) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 1.30) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 1.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 1.42) als eine **compilerinterne Datenstruktur** umgesetzt werden.

### Definition 1.30: Formaler Ableitungsbaum

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

Das Adjektiv „**formal**“ kann dabei weggelassen werden, wenn der **Kontext** in dem 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 **Datenstruktur** zur einfachen Weiterverarbeitung umsetzt.

Den Knoten dieses Baumes sind **Grammatiksymbole**  $V = N \cup \Sigma \cup \varepsilon$  (Definition 1.21) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole**  $N$  und die **Blätter** sind entweder **Terminalsymbole**  $\Sigma$  oder das **leere Wort**  $\varepsilon$ .<sup>b</sup>

<sup>a</sup>Z.B. **Programmcode**.

<sup>b</sup>Nebel, „Theoretische Informatik“.

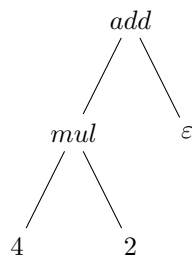
In Abbildung 1.30.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 1.30.1 nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition ??) angegebenen **Produktionen** 1.1 einer ansonsten nicht näher spezifizierten **Grammatik**  $G = \langle N, \Sigma, P, add \rangle$  ergibt.

$DIG\_NO\_0$	$::=$	"1"		"2"		"3"		"4"		"5"		"6"	$L\_Lex$
		"7"		"8"		"9"							
$DIG\_WITH\_0$	$::=$	"0"		$DIG\_NO\_0$									
$NUM$	$::=$	"0"		$DIG\_NO\_0$	$DIG\_WITH\_0^*$								
$add$	$::=$	$add$	"+"	$mul$		$mul$							$L\_Parse$
$mul$	$::=$	$mul$	"*"	$NUM$		$NUM$							

**Grammar 1.1:** *Produktionen des Ableitungsbaums*

$$add \Rightarrow mul \Rightarrow mul \text{ "*" } NUM \Rightarrow NUM \text{ "*" } NUM \Rightarrow 4 \text{ "*" } NUM \Rightarrow 4 \text{ "*" } 2 \quad (1.30.1)$$

Bei Ableitungsbäumen gibt es **keine** einheitliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 1.30.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort**  $\varepsilon$  hinzuzufügen.



(1.30.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.30.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen<sup>7</sup>.



(1.30.3)

## 1.2.2 Mehrdeutige Grammatiken

Für einen Compiler ist es notwendig, dass die **Grammatik**, welche die **Konkrete Syntax** beschreibt keine **Mehrdeutige Grammatik** (Definition 1.31) ist, denn sonst können unter anderem die **Präsenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel ?? an einem Beispiel demonstriert wird.

### Definition 1.31: Mehrdeutige Grammatik

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

<sup>a</sup>Alternativ, wenn es für  $w$  **mehrere** unterschiedliche **Linksableitungen** gibt.

<sup>b</sup>Nebel, „Theoretische Informatik“.

Bei manchen **Ansätzen** für das **Parsen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 1.32) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 1.45) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschaft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist<sup>8</sup>.

### Definition 1.32: LL(k)-Grammatik

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

<sup>a</sup>Das wird auch als **Lookahead** von  $k$  bezeichnet.

<sup>7</sup>Diese Option wurde beim **PicoC-Compiler** gewählt.

<sup>8</sup>Mehr **Erklärung** hierzu findet sich im Unterkapitel 1.4.

<sup>b</sup>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.

<sup>c</sup>Nebel, „Theoretische Informatik“.

### 1.2.3 Präzidenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Grammatik** für eine **Konkrete Syntax** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 1.33) und **Präzidenz** (Definition 1.34) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel ?? genauer erklärt.

#### Definition 1.33: 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.<sup>a</sup>

<sup>a</sup>Parsing Expressions · Crafting Interpreters.

Bei **Assoziativität** ist z.B. der **Multiplikationsoperator**  $*$  ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator**  $=$  ein Beispiel für einen **rechtsassoziativen** Operator. Dies ist in Abbildung 1.3 mithilfe von Klammern  $()$  veranschaulicht.

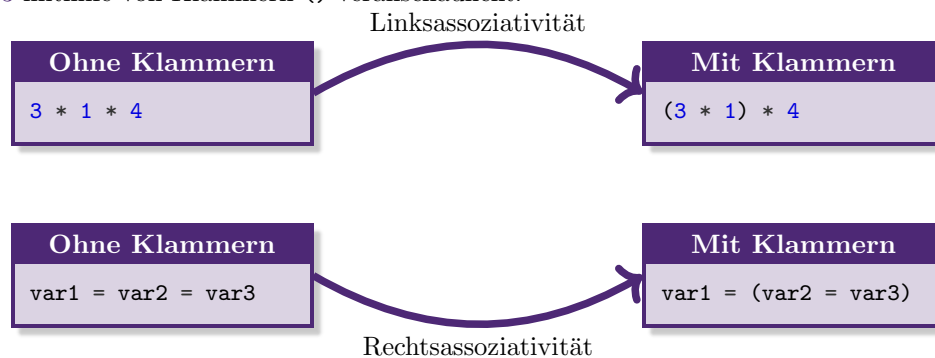


Abbildung 1.3: Veranschaulichung von Linksassoziativität und Rechtsassoziativität

#### Definition 1.34: Präzidenz

„Bestimmt, welcher Operator **zuerst** in einem Ausdruck, der eine Mischung **verschiedener** Operatoren enthält, ausgewertet wird. Operatoren mit einer **höheren Präzidenz**, werden **vor** Operatoren mit **niedrigerer Präzidenz** ausgewertet.“<sup>a</sup>

<sup>a</sup>Parsing Expressions · Crafting Interpreters.

Bei **Präzidenz** ist die Mischung der Operatoren für **Subtraktion**  $-$  und für **Multiplikation**  $*$  ein Beispiel für den Einfluss von Präzidenz. Dies ist in Abbildung 1.4 mithilfe der Klammern  $()$  veranschaulicht. Im Beispiel in Abbildung 1.4 ist bei den beiden **Subtraktionsoperatoren**  $-$  nacheinander und dem darauffolgenden **Multiplikationsoperator**  $*$  sowohl **Assoziativität** als auch **Präzidenz** im Spiel.



Abbildung 1.4: Veranschaulichung von Präzedenz

## 1.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architektur-patterns** (Definition 1.35) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 1.36) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

### Definition 1.35: Pipe-Filter Architekturpattern

Ist ein **Architekturpattern**, 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.<sup>a,b</sup>



<sup>a</sup>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 Anlehnung 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.

<sup>b</sup>Westphal, „Softwaretechnik“.

Diese Folgen endlicher Symbole werden auch **Lexeme** (Definition 1.37) genannt.

### Definition 1.36: 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<sup>a</sup>, die für die Beschreibung eines **Tokens**  $T$  zuständig sind.<sup>b</sup>

<sup>a</sup>Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

<sup>b</sup>Thiemann, „Compilerbau“.

### Definition 1.37: Lexeme

Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token**  $T$  einer **Sprache**  $L_{Lex}$  matched.<sup>a</sup>

<sup>a</sup>Thiemann, „Compilerbau“.



Diese **Lexeme** werden vom **Lexer** (Definition 1.38) im **Inputstring** identifiziert 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 1.38) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

#### Definition 1.38: Lexer (bzw. Scanner oder auch Tokenizer)

Ein **Lexer** ist eine *partielle Funktion*  $lex : \Sigma^* \rightarrow (N \times W)^*$ , 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*  $L_{Lex}$  ableiten lässt bzw. einem der *Pattern* der Sprache  $L_{Lex}$  entspricht.<sup>a</sup>

<sup>a</sup>Thieman, „Compilerbau“.

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

Um Verwirrung verzubäuen ist es wichtig folgende Unterscheidung hervorzuheben:

Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktische 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 1.39) von **Variablen, Konstanten und Funktionen** die Symbole<sup>a</sup>.

<sup>a</sup>Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel ?? **Symboltabelle** genannt wird.

#### Definition 1.39: Bezeichner (bzw. Identifier)

**Tokenwert**, der eine Konstante, Variable, Funktion usw. innerhalb ihres **Scopes eindeutig** benennt.<sup>a,b</sup>

<sup>a</sup>Außer wenn z.B. bei Funktionen die Programmiersprache das **Überladen** erlaubt usw. In diesem Fall wird die **Signatur** der Funktion als weiteres Unterscheidungsmerkmal hinzugenommen, damit es eindeutig ist.

<sup>b</sup>Thieman, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>9</sup> und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtigen Zeichen das leere Wort  $\epsilon$  zuordnet. Das ist auch im Sinne der Definition, denn  $\epsilon \in (N \times W)^*$  ist immer der Fall beim **Kleene Stern Operator**  $*$ . 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 **Überbegriffe** bzw. **Tokennamen** für

<sup>9</sup>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 Programmiersprache, die man zur Implementierung des Lexers nutzt wegabstrahiert.



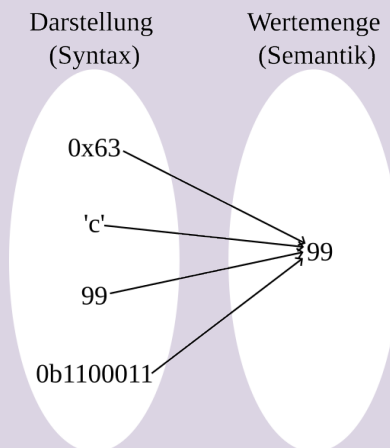
beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. NAME und NUM<sup>10</sup>, bzw. wenn man sich nicht Kurzformen sucht IDENTIFIER und NUMBER. Für **Lexeme**, wie if oder } sind die **Tokennamen** bzw. Überbegriffe genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich IF und RBRACE.

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 1.40) dargestellt werden, einmal als ASCII-Zeichen 'c', dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als 99<sup>11</sup>. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Die **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<sup>12</sup>, sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik ?? liefert den Beweis, dass die Sprache  $L_{PicoC\_Lex}$  des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie fast die Definition 1.23 erfüllt. Einzig die Produktion  $CHAR ::= \text{"\"ASCII\_CHAR\""}'$  sieht problematisch aus, kann allerdings auch als  $\{CHAR ::= \text{"\"CHAR2\""}, CHAR2 ::= ASCII\_CHAR\}$  **regulär** ausgedrückt werden<sup>13</sup>. Somit existiert eine **reguläre Grammatik**, welche die **Sprache**  $L_{PicoC\_Lex}$  beschreibt und damit ist die **Sprache**  $L_{PicoC\_Lex}$  **regulär**.

#### Definition 1.40: Literal

Eine von möglicherweise vielen weiteren **Darstellungsformen** (als **Zeichenkette**) für ein und denselben **Wert** eines **Datentyps**.<sup>a</sup>



<sup>a</sup>Thiemann, „Einführung in die Programmierung“.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 1.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

<sup>10</sup>Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Nodes haben will, damit unter anderem **mehr Code** in eine Zeile passt.

<sup>11</sup>Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen **0b1100011** und **0x63** darzustellen.

<sup>12</sup>Man nennt das auch einem **Lookahead** von 1

<sup>13</sup>Eine derartige Regel würde nur Probleme bereiten, wenn sich aus **ASCII\_CHAR** **beliebig breite** Wörter ableiten lassen.

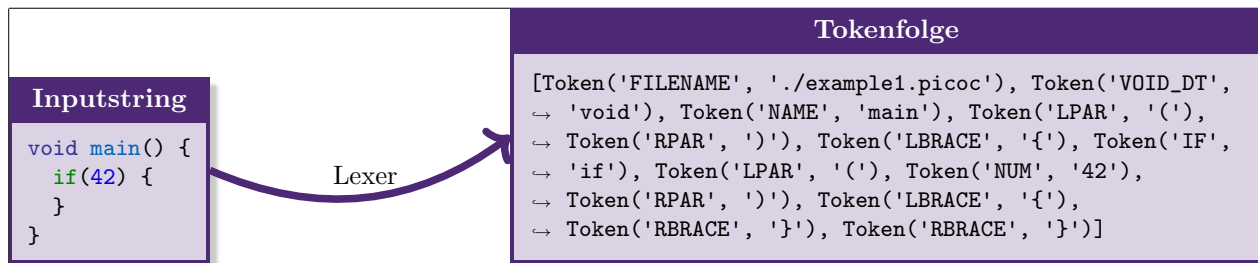


Abbildung 1.5: Veranschaulichung der Lexikalischen Analyse

## 1.4 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.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 1.41) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 1.43), ein **Ableitungsbaum** (Definition 1.42) generiert, der als Zwischenstufe hin zum einem **Abstrakter Syntaxbaum** (Definition 1.49) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaums** und dann erst des **Abstrakten Syntaxbaumes**.

### Definition 1.41: Konkrete Syntax

*Steht für alles, was mit dem **Aufbau von Ableitungsbäumen** zu tun hat, also z.B. was für **Ableitungen** mit den **Grammatiken**  $G_{Lex}$  und  $G_{Parse}$  zusammengenommen möglich sind.*

*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.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

### Definition 1.42: Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)

*Compilerinterne **Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 1.30) eines in **Konkreter Syntax** geschriebenen Programmes.*

*Die **Konkrete Syntax** nach der sich der **Ableitungsbaum** richtet wird optimalerweise immer so definiert, dass sich möglichst einfach ein **Abstrakter Syntaxbaum** daraus konstruieren lässt.<sup>a</sup>*

<sup>a</sup>JSON parser - Tutorial — Lark documentation.

### Definition 1.43: Parser

*Ein **Parser** ist ein Programm, dass aus einem Inputstring, der in **Konkreter Syntax** geschrieben ist, eine compilerinterne Darstellung, den **Ableitungsbaum** generiert, was auch als **Parsen** bezeichnet*

wird<sup>a, b</sup>.

<sup>a</sup>Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 1.43 verwendet.

<sup>b</sup>*JSON parser - Tutorial — Lark documentation.*

An dieser Stelle könnte möglicherweise eine Verwirrung entstehen, 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 verschiedenen Insekten entdeckt, dem Nachschlagen in einem Insektenlexikon 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<sup>a</sup>.

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 Insektenbeugnungen in einer für die **Weiterverarbeitung tauglichen Form**<sup>b</sup>.

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

<sup>a</sup>Das würde z.B. der Rolle eines **Semikolon** ; in der Sprache *LPicoC* entsprechen.

<sup>b</sup>Z.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beeinflussen sich gegenseitig.

Die vom **Lexer** im Inputstring 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 1.5 wieder relevant.

Ein **Parser** ist genauer gesagt ein erweiterter **Recognizer** (Definition 1.44), denn ein Parser löst das **Wortproblem** (Definition 1.25) für die **Sprache**, die durch die **Konkrete Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Ableitungsbaum**.

#### Definition 1.44: Recognizer (bzw. Erkenner)

*Entspricht dem Maschinenmodell eines **Automaten**. Im Bezug auf Compilerbau entspricht der **Recognizer** einem **Kellerautomaten**, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Recognizer** erkennt, ob ein Inputstring bzw. **Wort** sich mit den Produktionen der **Konkrete Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Syntax** beschrieben wird oder nicht.<sup>a, b</sup>*

<sup>a</sup>Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

<sup>b</sup>Thiemann, „Compilerbau“.

Für das **Parsen** gibt es grundsätzlich **zwei** 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 **Derivation Tree** mit dem **Startsymbol** der **Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man **Terminalsymbole** hat, die sich zum gewünschten **Inputstring** abgeleitet haben oder sich herausstellt, dass dieser nicht abgeleitet werden kann.<sup>a</sup>

Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist, weil der **Eingabewert** bzw. der **Inputstring** 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 1.45).

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 1.29) allerdings nicht möglich, ohne die Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der **Grammatik** entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

**Rekursiver Abstieg** kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 1.32) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewünschte Inputstring 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 der **Inputstring** sich **nicht** mit der verwendeten Grammatik ableiten lässt.<sup>b</sup>

Wenn man eine **LL(k)** Grammatik hat, kann man auf **Backtracking verzichten** und es reicht einfach nur immer **k Token** im Inputstring **vorauszuschauen**. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.<sup>c</sup>

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkreten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.<sup>d</sup>
- **Chart Parser:** Es wird **Dynamische Programmierung** verwendet und partielle Zwischenergebnisse 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.<sup>e</sup>

<sup>a</sup>What is Top-Down Parsing?

<sup>b</sup>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 *Lark - a parsing toolkit for Python*) ersetzt wurde.

<sup>c</sup>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.

<sup>d</sup>What is Bottom-up Parsing?

<sup>e</sup>Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie.

#### Definition 1.45: 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.

Der **Abstrakter Syntaxbaum** wird mithilfe von **Transformern** (Definition 1.46) und **Visitors** (Definition 1.47) 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 1.48).

#### Definition 1.46: Transformer

Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Ableitungsbaum** besucht und beim Antreffen eines bestimmten Knoten des **Derivation Tree** einen entsprechenden Knoten des **Abstrakter Syntaxbaum** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstrakter Syntaxbaum** konstruiert.<sup>a</sup>

<sup>a</sup>Transformers & Visitors — Lark documentation.

#### Definition 1.47: Visitor

Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Ableitungsbaum** besucht und in Bezug zu Compilerbau, beim Antreffen eines bestimmten **Knoten** des **Derivation Tree**, diesen **in-place** mit anderen Knoten **tauscht** oder **manipuliert**, um den **Derivation Tree** für die weitere Verarbeitung durch z.B. einen Transformer zu vereinfachen.<sup>a,b</sup>

<sup>a</sup>Kann theoretisch auch zur Konstruktion eines **Abstrakter Syntaxbaum** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakter Syntaxbaum** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.

<sup>b</sup>Transformers & Visitors — Lark documentation.

#### Definition 1.48: Abstrakte Syntax

Steht für alles, was mit dem **Aufbau** von **Abstrakten Syntaxbäumen** zu tun hat, also z.B. was für Arten von **Kompositionen** mit den **Knoten** eines **Abstrakten Syntaxbaums** möglich sind.

Ein **Abstract Syntax Tree**, der zur Kompilierung eines Wortes<sup>a</sup> generiert wurde, ist nach einer **Abstrakter Syntax** konstruiert.

Jene Produktionen, die in der **Konkreten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht. Dadurch sind die **Kompositionen**, welche die Knoten im **Abstract Syntax Tree** bilden können **syntaktisch** meist näher zur Syntax von **Maschinenbefehlen**.<sup>b</sup>

<sup>a</sup>Z.B. **Programmcode**.

<sup>b</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

#### Definition 1.49: 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.

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

Die **Abstrakte Syntax** nach der sich der **Abstrakte Syntaxbaum** richtet wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten<sup>a</sup> möglichst **einfach weiterverarbeitet** werden kann.

<sup>a</sup>Die verschiedenen **Passes**.

In Abbildung 1.6 wird das Beispiel aus Unterkapitel 1.2.1 fortgeführt, welches den **Arithmetischen Ausdruck**  $4 * 2$  in Bezug auf die Grammatik 1.1, welche die **höhere Präzidenz** der **Multiplikation**  $*$  berücksichtigt in einem **Ableitungsbaum** darstellt. In Abbildung 1.6 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**. Das geschieht bezogen auf das Beispiel aus Unterkapitel 1.2.1, indem jegliche Knoten, die im **Ableitungsbaum** nur existieren, weil die Grammatik so umgesetzt ist, dass es nur **einen** einzigen möglichen **Ableitungsbaum** geben kann **wegabstrahiert** werden.



**Abbildung 1.6:** 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, ist in Abbildung 1.7 die Syntaktische mit dem Beispiel aus Subkapitel 1.3 fortgeführt.

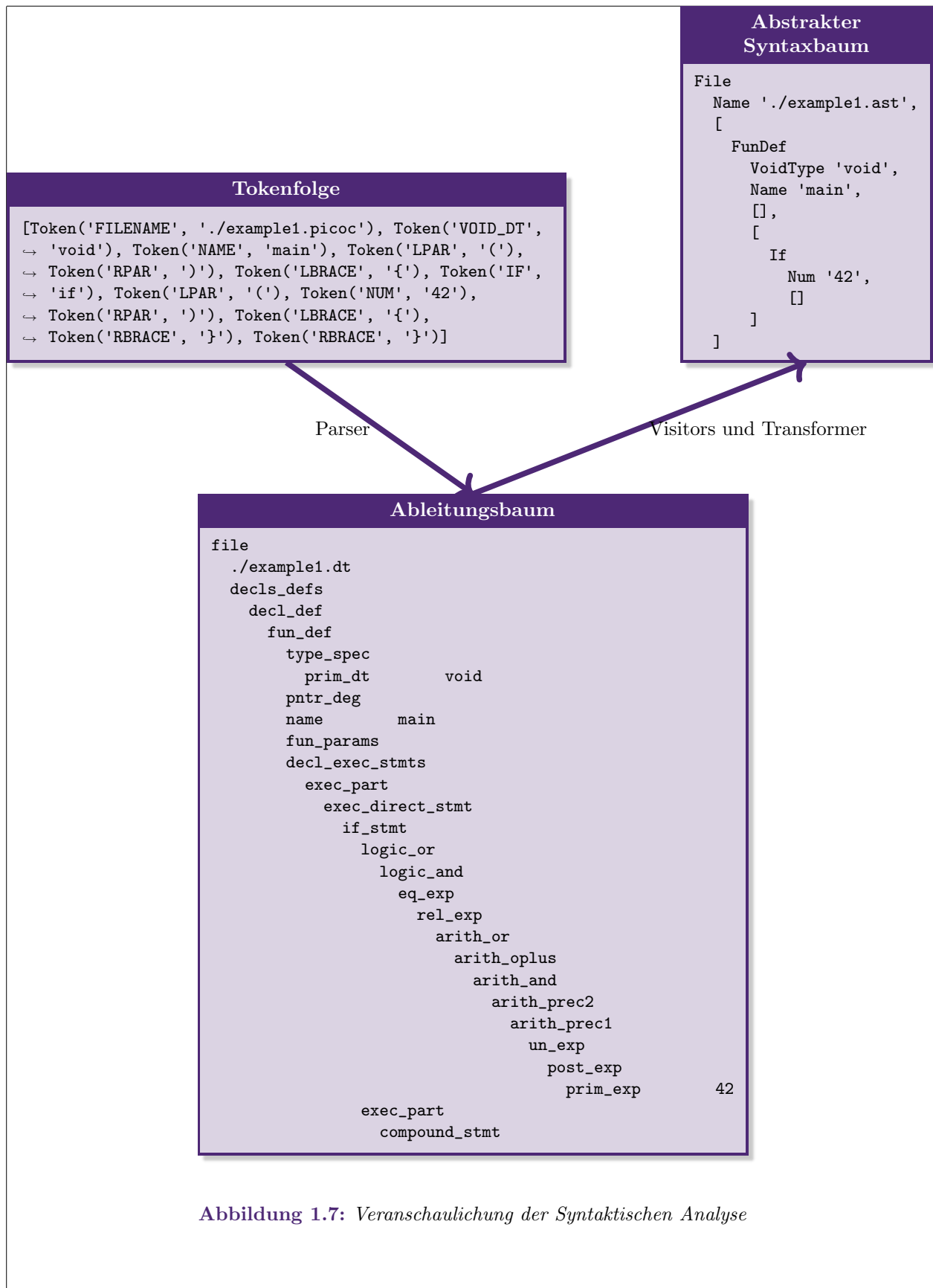


Abbildung 1.7: Veranschaulichung der Syntaktischen Analyse



## 1.5 Code Generierung

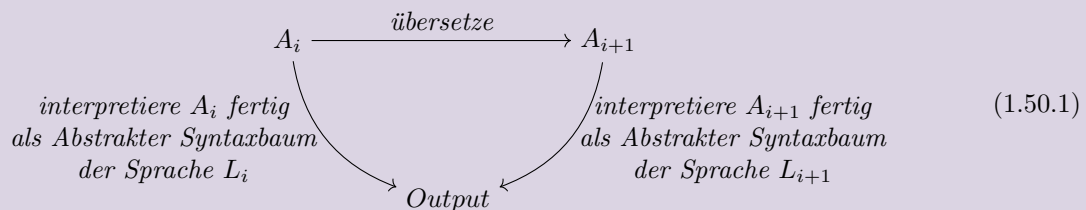
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** einer Sprache  $L_1$  in den **Abstrakter Syntaxbaum** einer Sprache  $L_2$  umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 1.50) nennt. So wie es auch schon mit dem **Derivation Tree** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakter Syntaxbaum** konstruiert hatte. Aus dem Derivation konnte, dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

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 Syntax**, 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 **Abstrakt Syntax Tree** überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die **Abstrakt Syntax** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Redeart** wurde aus der **Quelle** G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

### Definition 1.50: 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.<sup>ab</sup>*

*Für jeden **Pass** und für einen beliebigen **Abstrakten Syntaxbaum**  $A_i$  gilt ähnlich, wie bei einem **vollständigen Compiler** in 1.50.1, dass:*



*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.<sup>cd</sup>*

<sup>a</sup>Ein **Pass** kann mit einem **Transpiler** 1.9 (Definition 1.9) 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.

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

<sup>c</sup>**Interpretieren** geht immer von einem Programm in **Konkreter Syntax** aus, wobei der **Abstrakte Syntaxbaum** ein **Zwischenschritt** bei der **Interpretierung** ist.

<sup>d</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

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



### 1.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tun, welche **Unreine Ausdrücke** (Definition 1.52) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 1.51) 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 1.51: 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.<sup>a,b</sup>*

<sup>a</sup>Sondern z.B. **intern** etwas am **Kompilierprozess** ändert.

<sup>b</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

#### Definition 1.52: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definition 1.53).

#### Definition 1.53: Monadische Normalform (bzw. engl. monadic normal form)

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

*Eine **Konkrete Syntax** ist in **Monadischer Normalform**, wenn sie **reine Ausdrücke** und **unreine Ausdrücke nicht** miteinander **mischt**, sondern voneinander **trennt**.<sup>a</sup>*

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

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 1.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**<sup>14</sup> aufgeschrieben wurden.

In der Abbildung 1.8 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.

<sup>14</sup>Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

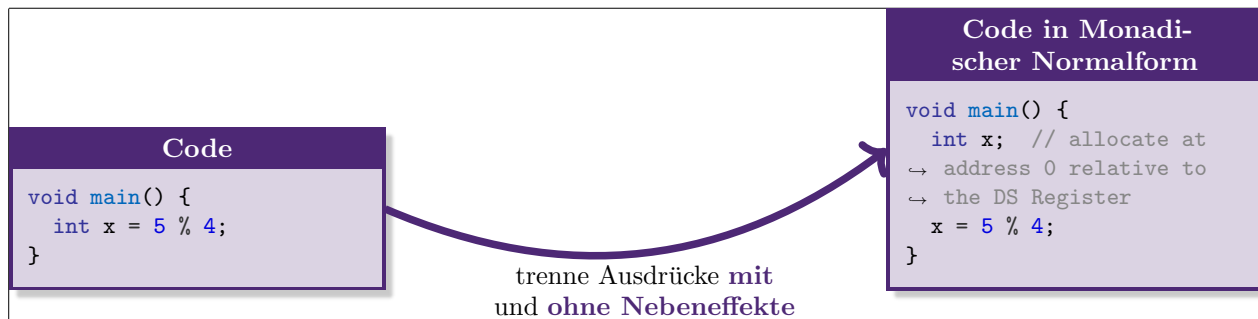


Abbildung 1.8: Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakter Syntaxbaum** der **Syntax** von **Maschinenbefehlen** anzunähern, 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.

### 1.5.2 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 1.56) 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 1.54) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statement**, 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 1.55) 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 1.54: 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 tun** hat und irgendwo **gespeichert** ist oder als **Speicherort** dient.<sup>a</sup>*

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 1.57). Wenn eine **Konkrete Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 1.57), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 1.55).

**Definition 1.55: 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  $L_{\text{PicoC}}$  entweder eine **Variable** `var`, eine **Zahl** `12`, ein **ASCII-Zeichen** `'c'` oder ein **Zugriff auf eine Location**, wie z.B. `stack(1)`.<sup>a</sup>

<sup>a</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 1.56: Komplexer Ausdruck**

Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist, wie z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`.<sup>ab</sup>

<sup>a</sup>`int var` ist eine **Allokation**.

<sup>b</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

**Definition 1.57: A-Normalform (ANF)**

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

Eine **Konkrete Syntax** 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 Syntax** ist in **A-Normalform**, wenn die **Konkrete Syntax** für welche sie definiert wurde in **A-Normalform** ist.<sup>abc</sup>

<sup>a</sup>A-Normalization: *Why and How (with code)*.

<sup>b</sup>Bolingbroke und Peyton Jones, „Types are calling conventions“.

<sup>c</sup>G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 1.9 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**<sup>15</sup> aufgeschrieben wurden.

Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** inklusive **Liveness Analysis**, um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden.<sup>16</sup>

Aus diesem Grund verwendet das Beispiel in Abbildung 1.9 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 **Abstrakter 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 Ausdruck** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl 4**, **Variable** `var` und **ASCII-Zeichen** `'c'` nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

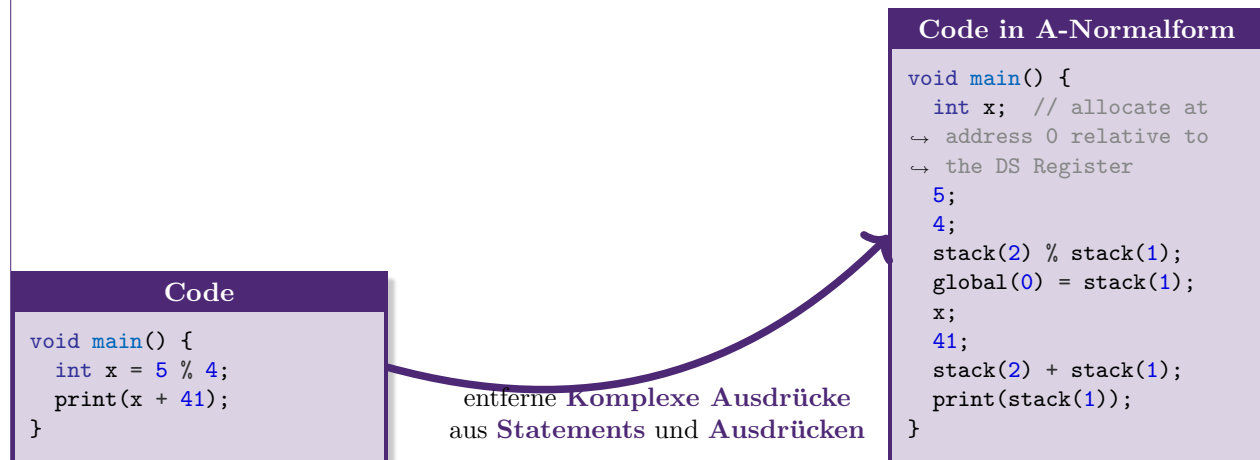
Im Fall, dass **Register** für z.B. **temporäre Zwischenergebnisse** genutzt werden und der **Maschinen-**

<sup>15</sup>Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

<sup>16</sup>Die in diesem **Paragraph** erwähnten **Begriffe** werden **nicht** genauer 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 verrechnen<sup>17</sup>, 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<sup>18</sup>. 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 verrechnet 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 1.9:** Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen

Ein solcher **Pass** hat vor allem in erster Linie die Aufgabe den **Abstrakt Syntax Tree** der **Syntax** von **Maschinenbefehlen** besonders dadurch anzunähern, dass er auf der Ebene der Konkreten Syntax 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.

### 1.5.3 Ausgabe des Maschienenencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakter Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt<sup>19</sup>. Der Weg von **Abstrakter Syntax** zu **Konkreter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax**

<sup>17</sup>Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.

<sup>18</sup>Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. **MULT ACC IN2**.

<sup>19</sup>Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenslesbaren Repräsentation** ausgegeben.

zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden musste.

Jeder **Knoten** des **Abstrakter Syntaxbaums** 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 **Depth-First-Search** Schema der gesamte **Abstract Sybtax Tree** 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 **zurückgeben**.

## 1.6 Fehlermeldungen

### Definition 1.58: Fehlermeldung

*Benachrichtigung beliebiger Form, die darüber informiert, dass:*

1. Ein Program beim **Kompilieren** von der **Konkreten Syntax** abweicht, also der **Inputstring** sich nicht mit der Konrekten Syntax **ableiten** lässt oder auf etwas **zugegriffen** werden soll, was noch **nicht** deklariert oder definiert wurde.
2. Beim Ausführen eine **verbotene** Operation ausgeführt wurde.<sup>a</sup>

<sup>a</sup>Errors in C/C++ - GeeksforGeeks.

### 1.6.1 Kategorien von Fehlermeldungen

---

---

# 2 Ergebnisse und Ausblick

Zum Schluss soll ein **Überblick** über das gegeben werden, was im Kapitel ?? implementiert wurde. Im Unterkapitel 2.1 wird darauf eingegangen ob die **versprochenen Funktionalitäten** des **PicoC-Compilers** aus Kapitel ?? alle implementiert werden konnten und daraufhin mithilfe **kurzer Anleitungen** ein grober Einblick gegeben, wie auf diese Funktionalitäten Zugriffen werden kann, aber auch auf Funktionalitäten **anderer mitimplementierter Tools**. Im Unterkapitel 2.2 wird aufgezeigt, was zur **Qualitätssicherung** implementiert wurde, um zu gewährleisten, dass der **PicoC-Compiler** die Kompilierung der **Programmiersprache  $L_{PicoC}$**  in **Syntax** und **Semantik identisch** zur entsprechenden **Untermenge** der Programmiersprache  $L_C$  umsetzt. Als allerletztes wird im Unterkapitel 2.3 ein Ausblick gegeben, wie der PicoC-Compiler **erweitert** werden könnte.

## 2.1 Funktionsumfang

In Kapitel ?? konnten **alle** Funktionalitäten, die in Kapitel ?? erläutert wurden implementiert werden. Während der **Funktionsumfang** des **PicoC-Compiler** zum Stand des **Bachelorprojektes** noch sehr beschränkt war und einzige eine **Strukturierte Programmierung** mit `if(cond){}else{}`, `while(cond){}` usw. erlaubte und komplexere Programme nur mit **viel Aufwand** und **unübersichtlichen Spaghetticode** implementierbar waren, erlaubt es der **PicoC-Compiler** nachdem er in der **Bachelorarbeit** um **Felder**, **Zeiger**, **Verbunde** und **Funktionen** erweitert wurde mittels der **Funktionen** eine **Prozedurale Programmierung** umzusetzen, in welcher das Programm in **Teilaufgaben** aufgeteilt werden kann, was zusammen mit der Möglichkeit **Felder**, **Zeiger** und **Verbunde** zu verwenden zur einem **geordneteren, intuitiv verständlicheren** und **übersichtlicheren** Code beiträgt.

Bei der Implementierung des **PicoC-Compilers** wurden verschiedene **Kommandozeilenoptionen** und **Modes** implementiert. Diese werden in den folgenden Kapiteln 2.1.1, 2.1.2 und 2.1.3 mithilfe kurzer **Anleitungen** erklärt.

Die kurzen **Anleitungen** in dieser **Schriftlichen Ausarbeitung** der Bachelorarbeit sollen nur zu einem **schnellen, grundlegenden Verständnis** der Verwendung des **PicoC-Compilers** und seiner **Kommandozeilenoptionen** und **Befehle** beihelfen, sowie zum Verständnis der **weiteren implementierten Tools**. Alle weiteren **Kommandozeilenoptionen** und **Befehle** sind für die Verwendung des PicoC-Compilers **unwichtig** und erweisen sich nur in **speziellen Situationen** als nützlich, weshalb für diese auf die **ausführlichere Dokumentation** unter **Link**<sup>1</sup> verwiesen wird.

### 2.1.1 Kommandozeilenoptionen

Will man einfach nur ein **Programm** `program.picoc` kompilieren ist das mit dem **PicoC-Compiler** genauso **unkompliziert** wie mit dem **GCC** durch einfaches **Angeben der Datei**, die kompiliert werden soll: `> picoc_compiler program.picoc`. Als Ergebnis des Kompiliervorgangs wird eine Datei `program.reti` mit dem entsprechenden **RETI-Code** erstellt, wobei für die **Benennung der Datei** einfach nur der

---

<sup>1</sup>[https://github.com/matthejue/PicoC-Compiler/blob/new\\_architecture/doc/help-page.txt](https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt)

**Basisname** der Datei `program` an eine neue **Dateiendung** `.reti` angehängt wird<sup>2</sup>.

Daneben gibt es allerdings auch die Möglichkeit **Kommandozeilenoptionen** `<cli-options>` in der Form `> picoc_compiler <cli-options> program.picoc` mitanzugeben, von denen die **wichtigsten** in Tabelle 2.1 erklärt sind. Alle weiteren **Kommandozeilenoptionen** können in der **Dokumentation** unter [Link](#) nachgelesen werden.

<sup>2</sup>Beim **GCC** wird bei **Nicht-Angabe** eines **Dateinamen** mit der `-o` Option dagegen eine Datei mit der festen Namen `a.out` erstellt.



Kommandozeilenoption	Beschreibung	Standardwert
<code>-i, --intermediate_stages</code>	Gibt <b>Zwischenschritte</b> der Kompilierung in Form der verschiedenen <b>Tokens</b> , <b>Ableitungsbäume</b> , <b>Abstrakten Syntaxbäume</b> der verschiedenen <b>Passes</b> in Dateien mit <b>entsprechenden Dateieindungen</b> aber gleichem <b>Basinamen</b> aus. Im <b>Shell-Mode</b> erfolgt <b>keine</b> Ausgabe in Dateien, sondern nur im <b>Terminal</b> .	<b>false</b> , most_used: true
<code>-p, --print</code>	Gibt alle <b>Dateiausgaben</b> auch im <b>Terminal</b> aus. Diese Option ist im <b>Shell-Mode</b> dauerhaft aktiviert.	<b>false</b> (true im <b>Shell-Mode</b> und für den most_used- Befehl)
<code>-v, --verbose</code>	Fügt den verschiedenen <b>Zwischenschritten der Kompilierung</b> , unter anderem auch dem finalen RETI-Code <b>Kommentare</b> hinzu, welche ein <b>Statement</b> oder <b>Befehl</b> aus einem <b>vorherigen Pass</b> beinhalten, der durch die darunterliegenden Statements oder Befehle <b>ersetzt</b> wurde. Wenn die <code>--run</code> -Option aktiviert ist, wird der <b>Zustand</b> der virtuellen RETI-CPU <b>vor</b> und <b>nach jedem Befehl</b> angezeigt.	<b>false</b>
<code>-vv, --double_verbose</code>	Hat <b>dieselben Effekte</b> , wie die <code>--verbose</code> -Option, aber bewirkt zusätzlich <b>weitere Effekte</b> . <b>PicoC-Knoten</b> erhalten bei der Ausgabe in den Abstrakten Syntaxbäumen zusätzliche <b>runde Klammern</b> , sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In <b>Fehlermeldungen</b> werden <b>mehr Tokens</b> angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der <code>--intermediate_stages</code> -Option werden in den dadurch ausgegebenen <b>Abstrakten Syntaxbäumen</b> ebenfalls <b>versteckte Attribute</b> , die <b>Informationen zu Datentypen</b> und für <b>Fehlermeldungen</b> beinhalten angezeigt.	<b>false</b>
<code>-h, --help</code>	Zeigt die <b>Dokumentation</b> , welche ebenfalls unter <b>Link</b> gefunden werden kann <b>im Terminal</b> an. Mit der <code>--color</code> -Option kann die <b>Dokumentation</b> mit <b>farblicher Hervorhebung</b> im Terminal angezeigt werden.	<b>false</b>
<code>-R, --run</code>	Führt die <b>RETI-Befehle</b> , die das Ergebnis der Kompilierung sind mit einer <b>virtuellen RETI-CPU</b> aus. Wenn die <code>--intermediate_stages</code> -Option aktiviert ist, wird eine Datei <code>&lt;basename&gt;.reti_states</code> erstellt, welche den <b>Zustand der RETI-CPU</b> nach dem <b>letzten</b> ausgeführten RETI-Befehl enthält. Wenn die <code>--verbose</code> - oder <code>--double_verbose</code> -Option aktiviert ist, wird der Zustand der RETI-CPU <b>vor</b> und <b>nach</b> jedem Befehl auch noch zusätzlich in die Datei <code>&lt;basename&gt;.reti_states</code> ausgegeben.	<b>false</b> , most_used: true
<code>-B, --process_begin</code>	Setzt die <b>relative Adresse</b> , wo der <b>Prozess</b> bzw. das <b>Codesegment</b> für das ausgeführte Programm beginnt.	3
<code>-D, --datasegment_size</code>	Setzt die Größe des <b>Datensegments</b> . Diese <b>Option</b> muss mit <b>Vorsicht</b> gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die <b>Globalen Statischen Daten</b> und der <b>Stack</b> miteinander kollidieren.	32

Tabelle 2.1: Kommandozeilenoptionen



Alle **kleingeschriebenen** Kommandozeilenoptionen, wie `-i`, `-p`, `-v` usw. betreffen dabei den **PicoC-Compiler** und alle **großgeschriebenen** Kommandozeilenoptionen, wie `-R`, `-B`, `-D` usw. betreffen den **RETI-Interpreter**.

### 2.1.2 Shell-Mode

Will man z.B. eine **Folge von Statements** in der Programmiersprache  $L_{PicoC}$  **schnell** kompilieren ohne eine Datei erstellen zu müssen, so kann der **PicoC-Compiler** im sogenannten **Shell-Mode** aufgerufen werden. Hierzu wird der PicoC-Compiler **ohne Argumente** `> picoc_compiler` aufgerufen, wie es in Code 2.1 zu sehen ist. Die angegebene **Folge von Statements** `<seq-of-stmts>` wird dabei automatisch in eine `main`-Funktion eingefügt: `void main(){<seq-of-stmts>}`.

Mit dem `> compile <cli-options> <filename>`-Befehl (oder der **Abkürzung** `cpl`) kann **PicoC-Code** zu **RETI-Code** kompiliert werden. Die Kommandozeilenoptionen `<cli-options>` sind dieselben, wie wenn der Compiler **direkt** mit Kommandozeilenoptionen aufgerufen wird. Die **wichtigsten** dieser **Kommandozeilenoptionen** sind in Tabelle 2.1 angegeben.

Mit dem Befehl `> quit` kann der **Shell-Mode** wieder **verlassen** werden.

```
> picoc_compiler
PicoC Shell. Enter `help` (shortcut `?`) to see the manual.
PicoC> cpl "6 * 7;";
----- RETI -----
SUBI SP 1;
LOADI ACC 6;
STOREIN SP ACC 1;
SUBI SP 1;
LOADI ACC 7;
STOREIN SP ACC 1;
LOADIN SP ACC 2;
LOADIN SP IN2 1;
MULT ACC IN2;
STOREIN SP ACC 2;
ADDI SP 1;
LOADIN BAF PC -1;

Compilation successfull

PicoC> quit
```

**Code 2.1:** Shellaufruf und die Befehle *compile* und *quit*

Wenn man möglichst alle nützlichen **Kommandozeilenoptionen** direkt aktiviert haben will, bei denen es **keinen** Grund gibt, sie nicht mitanzugeben, kann der Befehl `> most_used <cli-options> <filename>` (oder seine **Abkürzung** `mu`) genutzt werden, um diese Kommandozeilenoptionen mit dem `compile`-Befehl **nicht** jedes mal **selbst** Angeben zu müssen. In der Tabelle 2.1 sind in grau die Werte der einzelnen **Kommandozeilenoptionen** angegeben, die bei dem Befehl `most_used` gesetzt werden. In Code 2.2 ist der `most_used`-Befehl in seiner Verwendung zu sehen.






Dadurch, dass die `--intermediate_stages`- und die `--run`-Option beim `most_used`-Befehl aktiviert sind, werden die verschiedenen **Zwischenstufen** der Kompilierung, wie **Tokens**, **Derivation Tree** usw., sowie der **Zustand der RETI-CPU** nach der Ausführung des **letzten** Befehls angezeigt. Aus **Platzgründen** ist das meiste allerdings mit `'...'` ausgelassen.


```

PicoC> mu "int var = 42;";
----- Code -----
// stdin.picoc:
void main() {int var = 42;}
----- Tokens -----
...
----- Derivation Tree -----
...
----- Derivation Tree Simple -----
...
----- Abstract Syntax Tree -----
...
----- PicoC Shrink -----
...
----- PicoC Blocks -----
...
----- PicoC Mon -----
...
----- Symbol Table -----
...
----- RETI Blocks -----
...
----- RETI Patch -----
...
----- RETI -----
SUBI SP 1;
LOADI ACC 42;
STOREIN SP ACC 1;
LOADIN SP ACC 1;
STOREIN DS ACC 0;
ADDI SP 1;
LOADIN BAF PC -1;
----- RETI Run -----
...
Compilation successfull

```

Code 2.2: Shell-Mode und der Befehl *most\_used*

Im **Shell-Mode** kann der **Cursor** mit den  und  Pfeiltasten bewegt werden. In der **Befehlshistorie** kann sich mit den  und  Pfeiltasten **rückwärts** und **vorwärts** bewegt werden. Mit  kann ein Befehl **automatisch vervollständigt** werden.

Es gibt für den **Shell-Mode** noch **weitere Befehle**, wie `color_toggle`, `history` etc. und **kleinere Funktionalitäten** für die Shell, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wird allerdings auf die **Dokumentation** unter [Link](#) verwiesen, welche auch über den Befehl  angezeigt werden kann.

### 2.1.3 Show-Mode

Der **Show-Mode** ist ein Nebenprodukt der Implementierung des **PicoC-Compilers**. Dieser **Mode** wurde eigentlich nur implementiert, um beim **Testen** des PicoC-Compilers **Bugs** bei der Generierung des **RETI-Code** zu finden, indem im Terminal eine **virtuelle RETI-CPU** angezeigt wird, welches den **kompletten**

**Zustand** einer virtuell ausgeführten RETI mit allen **Registern**, **SRAM**, **UART**, **EPROM** und einigen **weiteren Informationen** anzeigt.

Allerdings bringt die Möglichkeit des **Show-Mode**, die **RETI-Befehle** des übersetzten Programmes in **Ausführung zu sehen** auch einen großen **Lerneffekt** mit sich, weshalb der **Show-Mode** noch **weiterentwickelt** wurde, sodass auch **Studenten** ihn auf unkomplizierte Weise nutzen können.

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen**, die z.B. unter `<more-options>` noch für die **Makefile** gesetzt werden können sind in Tabelle 2.2 aufgelistet.

Kommandozeilenoption	Beschreibung	Standardwert
FILEPATH	Pfad zur Datei, die im <b>Show-Mode</b> angezeigt werden soll	<code>()</code>
TESTNAME	Name des Tests. Alles andere als der <b>Basisname</b> , wie die <b>Dateiendung</b> wird abgeschnitten	<code>()</code>
EXTENSION	<b>Dateiendung</b> , die an TESTNAME angehängt werden soll zu <code>./tests/TESTNAME.EXTENSION</code>	<code>reti_states</code>
NUM_WINDOWS	<b>Anzahl Fenster</b> auf die ein Dateiinhalte <b>verteilt</b> werden soll	<code>5</code>
VERBOSE	Möglichkeit die <b>Kommandozeilenoption</b> <code>-v</code> oder <code>-vv</code> zu aktivieren für eine <b>ausführlichere Ausgabe</b>	<code>()</code>
DEBUG	Möglichkeit die <b>Kommandozeilenoption</b> <code>-d</code> zu aktivieren, um bei <code>make test-show TESTNAME=&lt;testname&gt;</code> den <b>Debugger</b> für den entsprechenden <b>Test</b> <code>&lt;testname&gt;</code> zu starten	<code>()</code>

**Tabelle 2.2:** Makefileoptionen

Alternativ kann der **Show-Mode** mit dem Befehl `make test-show TESTNAME=<testname> <more-options>` auch für einen der geschriebenen **Tests** im Ordner `/tests` gestartet werden. Der **Test** wird bei diesem Befehl **erst ausgeführt** und dann der **Show-Mode** gestartet.

Der **Show-Mode** nutzt den Terminal Texteditor **Neovim**<sup>3</sup> um einen **Dateiinhalte** über mehrere **Fenster** verteilt anzuzeigen, so wie es in Abbildung 2.1 zu sehen ist. Für den **Show-Mode** wird eine eigene **Konfiguration für Neovim** verwendet, welche in der **Konfigurationsdatei** `/interpr_showcase.vim` spezifiziert ist.

Gedacht ist der **Show-Mode** vor allem dafür etwas ähnliches wie ein **RETI-Debugger** zu sein und wird daher standardmäßig bei **Nicht-Angabe** einer **EXTENSION** auf die Datei `<program>.reti_states` angewandt. Der **Show-Mode** kann aber auch dazu genutzt werden **andere Dateien**, welche verschiedene Zwischenschritte der Kompilierung darstellen anzuzeigen, indem **EXTENSION** auf eine andere **Dateiendung** gesetzt wird.

Im **Show-Mode** wird ein Trick angewandt, indem die verschiedenen **Zustände der RETI-CPU nicht zur Laufzeit** des **Show-Mode** berechnet werden, sondern schon berechnet wurden und nacheinander in die Datei `<program>.reti_states` ausgegeben wurden. Der **Show-Mode** macht nichts anderes, als immer an die Stelle zu springen, an welcher der nächste Zustand anfängt. Durch Drücken von `Tab` und `↑ -Tab` können auf diese Weise die **verschiedenen Zuständen der RETI-CPU vor** und **nach** der Ausführung eines Befehls **angezeigt** werden.

<sup>3</sup>Home - Neovim.

index: 43	00019 ADDI SP 1;	00057 LOADIN DS ACC 0;	00095 STOREIN SP ACC 2;	00133 0
instruction: ADDI SP 1;	00020 LOADIN SP ACC 1;	00058 STOREIN SP ACC 1;	00096 ADDI SP 1;	00134 0
ACC: 1	00021 STOREIN DS ACC 0;	00059 LOADIN SP ACC 1;	00097 LOADIN SP ACC 1;	00135 0
ACC_SIMPLE: 1	00022 ADDI SP 1;	00060 ADDI SP 1;	00098 STOREIN DS ACC 0;	00136 0
IN1: 0	00023 SUBI SP 1;	00061 CALL PRINT ACC;	00099 ADDI SP 1;	00137 0
IN1_SIMPLE: 0	00024 LOADIN DS ACC 0;	00062 SUBI SP 1;	00100 JUMP -32;	00138 0
IN2: 4	00025 STOREIN SP ACC 1;	00063 LOADI ACC 0;	00101 SUBI SP 1;	00139 0
IN2_SIMPLE: 4	00026 SUBI SP 1;	00064 STOREIN SP ACC 1;	00102 LOADIN DS ACC 0;	00140 0
PC: 2147483686	00027 LOADI ACC 4;	00065 LOADIN SP ACC 1;	00103 STOREIN SP ACC 1;	00141 0
PC_SIMPLE: 38	00028 STOREIN SP ACC 1;	00066 STOREIN DS ACC 0;	00104 LOADIN SP ACC 1;	00142 0
SP: 2147483792	00029 LOADIN SP ACC 2;	00067 ADDI SP 1;	00105 ADDI SP 1;	00143 0
SP_SIMPLE: 144	00030 STOREIN SP IN2 1;	00068 SUBI SP 1;	00106 JUMP== 7;	00144 4 <- SP
BAF: 2147483650	00031 SUB ACC IN2;	00069 LOADIN DS ACC 0;	00107 SUBI SP 1;	00145 1
BAF_SIMPLE: 2	00032 JUMP< 3;	00070 STOREIN SP ACC 1;	00108 LOADIN DS ACC 0;	UART:
CS: 2147483651	00033 LOADI ACC 0;	00071 SUBI SP 1;	00109 STOREIN SP ACC 1;	00000 0
CS_SIMPLE: 3	00034 JUMP 2;	00072 LOADI ACC 2;	00110 LOADIN SP ACC 1;	00001 0
DS: 2147483762	00035 LOADI ACC 1;	00073 STOREIN SP ACC 1;	00111 ADDI SP 1;	00002 0
DS_SIMPLE: 114	00036 STOREIN SP ACC 2;	00074 LOADIN SP ACC 2;	00112 CALL PRINT ACC;	00003 0
SRAM:	00037 ADDI SP 1;	00075 LOADIN SP IN2 1;	00113 LOADIN BAF PC -1;	EPROM:
00000 JUMP 0;	00038 LOADIN SP ACC 1; <- PC	00076 SUB ACC IN2;	00114 3 <- DS	00000 LOADI DS -2097152; <- IN1
00001 2147483648	00039 ADDI SP 1;	00077 JUMP< 3;	00115 0	00001 MULTI DS 1824; <- ACC
00002 0 <- BAF	00040 JUMP== 2;	00078 LOADI ACC 0;	00116 0	00002 MOVE DS SP;
00003 CALL INPUT ACC; <- CS	00041 JUMP -32;	00079 JUMP 2;	00117 0	00003 MOVE DS BAF;
00004 SUBI SP 1;	00042 SUBI SP 1;	00080 LOADI ACC 1;	00118 0	00004 MOVE DS CS; <- IN2
00005 STOREIN SP ACC 1;	00043 LOADIN DS ACC 0;	00081 STOREIN SP ACC 2;	00119 0	00005 ADDI SP 145;
00006 LOADIN SP ACC 1;	00044 STOREIN SP ACC 1;	00082 ADDI SP 1;	00120 0	00006 ADDI BAF 2;
00007 STOREIN DS ACC 0;	00045 SUBI SP 1;	00083 LOADIN SP ACC 1;	00121 0	00007 ADDI CS 3;
00008 ADDI SP 1;	00046 LOADI ACC 2;	00084 ADDI SP 1;	00122 0	00008 ADDI DS 114;
00009 SUBI SP 1;	00047 STOREIN SP ACC 1;	00085 JUMP== 16;	00123 0	00009 MOVE CS PC;
00010 LOADIN DS ACC 0;	00048 LOADIN SP ACC 2;	00086 SUBI SP 1;	00124 0	
00011 STOREIN SP ACC 1;	00049 LOADIN SP IN2 1;	00087 LOADIN DS ACC 0;	00125 0	
00012 SUBI SP 1;	00050 SUB ACC IN2;	00088 STOREIN SP ACC 1;	00126 0	index: 44
00013 LOADI ACC 1;	00051 STOREIN SP ACC 2;	00089 SUBI SP 1;	00127 0	instruction: LOADIN SP ACC 1;
00014 STOREIN SP ACC 1;	00052 ADDI SP 1;	00090 LOADI ACC 1;	00128 0	ACC: 1
00015 LOADIN SP ACC 2;	00053 LOADIN SP ACC 1;	00091 STOREIN SP ACC 1;	00129 0	ACC_SIMPLE: 1
00016 LOADIN SP IN2 1;	00054 ADDI SP 1;	00092 LOADIN SP ACC 2;	00130 0	IN1: 0
00017 ADD ACC IN2;	00055 JUMP== 13;	00093 LOADIN SP IN2 1;	00131 0	IN1_SIMPLE: 0
00018 STOREIN SP ACC 2;	00056 SUBI SP 1;	00094 ADD ACC IN2;	00132 0	IN2: 4
00019 ADDI SP 1;	00057 LOADIN DS ACC 0;	00095 STOREIN SP ACC 2;	00133 0	IN2_SIMPLE: 4
				PC: 2147483687

Abbildung 2.1: Show-Mode in der Verwendung

Zur **besseren Orientierung** wird für alle Register ebenfalls ein mit der Registerbezeichnung beschrifteter **Zeiger** <- REG an Adressen im **EPROM**, **UART** und **SRAM** angezeigt, je nachdem, ob der **Wert im Register** nach der **Memory Map** dem **Adressbereich** von **EPROM**, **UART** oder **SRAM** entspricht.

Durch Drücken von **Esc** oder **q** kann der **Show-Mode** wieder verlassen werden. Es gibt für den **Show-Mode** noch viele weitere **Tastenkürzel**, die sich in der ein oder anderen Situation als **nützlich** erweisen können. Für die **Erklärung** dieser wieder allerdings auf die **Dokumentation** unter [Link](#) verwiesen. Des Weiteren stehen durch die Nutzung des Terminal Texteditors **Neovim** auch alle **Funktionalitäten** dieses mächtigen Terminal Texteditors zur Verfügung, welche mittels der Eingabe von **:help** **nachgelesen** werden können oder mittels der Eingabe von **:Tutor** mithilfe einer kurzen **Einführungsanleitung** **erlernt** werden können.

## 2.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm 1.2.1 in Unterkapitel 1.1 genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 2.0.1 definieren. Ein **beliebiges** Testprogramm  $P_{PicoC}$  in der Sprache  $L_{PicoC}$  muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm  $P_{RETI}$  in der Sprache  $L_{RETI}$ , trotz der **unterschiedlichen Syntax**.

Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis **/tests** bzw. unter [Link](#)<sup>4</sup> zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 2.3.

<sup>4</sup>[https://github.com/matthejue/PicoC-Compiler/tree/new\\_architecture/tests](https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests).

Testkategorie	Beschreibung
basic	<b>Einfache</b> Tests, welche die <b>grundlegenden Funktionalitäten</b> des Compilers testen
advanced	Tests, die <b>Spezialfälle</b> und <b>Kombinationen</b> verschiedener Funktionalitäten des Compilers testen
hard	Tests, die <b>längere, komplexe Programme</b> testen, für welche die Funktionalitäten des Compilers in <b>perfekter Harmonie</b> miteinander funktionieren müssen
example	Tests, die <b>bekannte Algorithmen</b> darstellen und daher als gutes, <b>repräsentatives</b> Beispiel für die <b>Funktionsfähigkeit</b> des PicoC-Compilers dienen
error	Tests, die <b>Fehlermeldungen</b> testen. Für diese Tests wird <b>keine Verifikation</b> ausgeführt
exclude	Tests, für welche aufgrund vielfältiger Gründe <b>keine Verifikation</b> ausgeführt werden soll

Tabelle 2.3: Testkategorien

Dass die Programme in beiden Sprachen die **gleiche Semantik** haben, lässt sich mit einer **hohen Wahrscheinlichkeit** gewährleisten, wenn beide die **gleiche Ausgabe** haben und es sehr **unwahrscheinlich** ist zufällig bei der gewählten Eingabe die spezifische Ausgabe zu erhalten. Wenn **immer mehr Tests**, die alle einen unterschiedlichen Teil der Semantik der Sprache  $L_{PicoC}$  abdecken vorliegen, bei denen die jeweiligen Programme  $P_{PicoC}$  und  $P_{RETI}$  interpretiert die gleiche **Ausgabe** haben, dann kann mit **immer höherer Wahrscheinlichkeit** von einem **funktionierenden** Compiler ausgegangen werden.

Die Kante vom Testprogramm  $P_{PicoC}$  zur Ausgabe aus Diagramm 2.0.1 drückt aus, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat, in welcher der **Schreiber des Tests** die Rolle des entsprechenden **Interpreters**<sup>5</sup> aus Diagramm 1.2.1 übernimmt und die **erwartete Ausgabe** seiner eigenen Interpretation des **PicoC-Codes** anstelle von `<space_seperated_output>` hineinschreibt.

Ein Beispiel für einen **Test** ist in Code 2.3 zu sehen. Sobald die Tests mithilfe der `/Makefile` mit dem Befehl `> make test` ausgeführt werden, wird als erstes für **jeden** Test das Bashscript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert<sup>6</sup> und die entsprechenden Werte in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt.

Die Datei `<program>.in` enthält **Eingaben**, welche durch `input()`-Funktionsaufrufe eingelesen werden, die Datei `<program>.out_expected` enthält zu **erwartende Ausgaben** der `print(<exp>)`-Funktionsaufrufe, die später eingeführte Datei `<program>.out` enthält die **tatsächlichen Ausgaben** der `print(<exp>)`-Funktionsaufrufe bei der **Ausführung des Tests** und die Datei `<program>.datasegment_size` enthält die **Größe des Datensegments** für die Ausführung des entsprechenden Tests.

<sup>5</sup>Der die **Semantik** des Tests umsetzt.

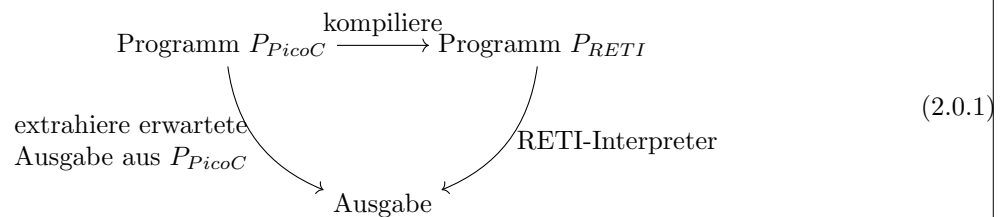
<sup>6</sup>Falls vorhanden.

```
// in:21 2 6 7
// expected:42 42
// datasegment:4

void main() {
    print(input() * input());
    print(input() * input());
}
```

Code 2.3: Typischer Test

Die Kante vom Programm  $P_{RETI}$  zur Ausgabe aus Abbildung 2.0.1 ist dadurch erfüllt, dass das Programm  $P_{RETI}$  vom **RETI-Interpreter** interpretiert wird und jedes mal beim Antreffen des **RETI-Befehls** CALL PRINT ACC der entsprechende **Inhalt** des ACC-Registers in die Datei `<program>.out` ausgegeben wird. Ein Test kann mit einer bestimmten Wahrscheinlichkeit die **Korrektheit** des **Teils der Semantik** der Sprache  $L_{PicoC}$ , die er abdeckt **verifizieren**, wenn der Inhalt von `<program>.out_expected` und `<program>.out` **identisch** ist.

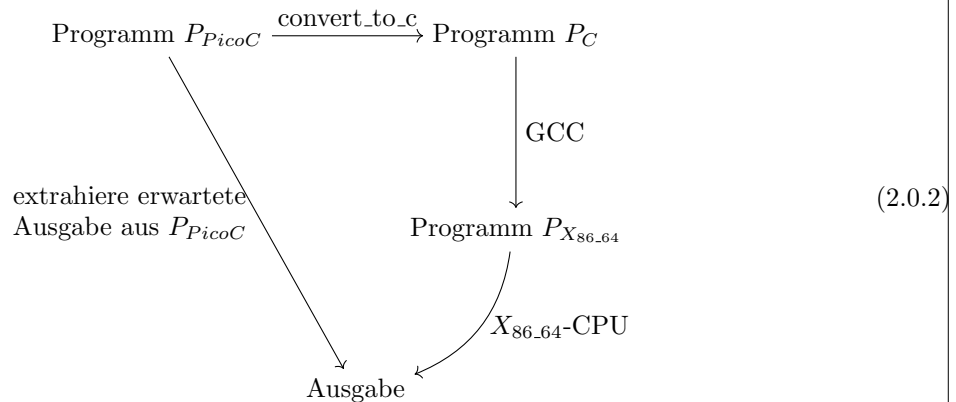


Allerdings gibt es bei dem Testverfahren, welches in Diagramm 2.0.1 dargestellt ist ein **Problem**, denn der **Schreiber** der Tests ist in diesem Fall die **gleiche Person**, die auch den **Compiler implementiert**. Wenn der **Schreiber** der Tests ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl im **Test** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von  $L_{PicoC}$  entspricht<sup>7</sup>.

Aus diesem Grund muss hier eine **weitere Maßnahme**, welche in Diagramm 2.0.2 dargestellt ist eingeführt werden, die gewährleistet, dass die **Ausgabe** in Diagramm 2.0.1 sich auf jeden Fall aus der **Semantik** der Sprache  $L_{PicoC}$ <sup>8</sup> ergibt. Das wird erreicht, indem wie in Diagramm 2.0.2 dargestellt ist, überprüft wird, ob die **Ausgabe** des Pfades von  $P_{PicoC}$  zur Ausgabe mit der **Ausgabe** des Pfades von  $P_C$  über  $P_{X_{86-64}}$  **identisch** ist.

<sup>7</sup>Welche ja identisch zu der von  $L_C$  sein sollte.

<sup>8</sup>Die eine **Untermenge** von  $L_C$  ist.



Das Programm  $P_C$  ergibt sich dabei aus dem Testprogramm  $P_{PicoC}$  durch **Ausführen** des Pythonscripts `/convert_to_c.py`, welches **später näher erläutert** wird. Mithilfe der `/Makefile` und dem Befehl `> make convert` lässt sich dieses Pythonscript auf **alle** Tests anwenden.

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante von  $P_C$  zu  $P_{X86_64}$ . Beim **GCC** handelt es sich um einen Compiler der Sprache  $L_C$ , der somit auch mit Ausnahme der `print()` und `input()`-Funktionen auch die Sprache  $L_{PicoC}$  kompilieren kann. Der **GCC** setzt aufgrund seiner bekanntermaßen **vielfachen Verwendung** auf der Welt und seinem **sehr langem Bestehen** seit 1987<sup>9</sup> <sup>10</sup> die **Semantik** der Sprache  $L_C$ , vor allem für die kleine Untermenge, welche  $L_{PicoC}$  darstellt mit sehr hoher Wahrscheinlichkeit **korrekt** um.

Durch das **Abgleichen** mit dem **GCC** in Diagramm 2.0.2 kann nun **sichergestellt** werden, dass die Tests **nicht** nur die Interpretation, die der Schreiber der Tests und Implementierer des **PicoC-Compilers** von der Semantik der Sprache  $L_{PicoC}$  hat **bestätigen**, sondern die tatsächliche **Einhaltung der Semantik** der Sprache  $L_{PicoC}$  testen.

Dazu durchläuft jeder Test, wie in Diagramm 2.0.2 dargestellt ist eine **Verifikation**, in der **verifiziert** wird, ob bei der Kompilierung des Testprogramms  $P_C$  mit dem **GCC** und Ausführung des hieraus generierten  $X_{86_64}$ -Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe // `expected:<space_seperated_output>` des Testschreibers ist. Erst dann ist ein Test **verifiziert**, d.h. man kann, wenn der Test **vernünftig definiert** ist mit **hoher Wahrscheinlichkeit** sagen<sup>11</sup>, dass wenn dieser Test für den **PicoC-Compiler** durchläuft, der **Teil der Semantik** der Sprache  $L_{PicoC}$ , den dieser Test testet vom PicoC-Compiler **korrekt umgesetzt** ist.

Für diese **Verifikation** ist das Bashscript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt wird. Beim Befehl `> make test` wird dieses Bashscript **vor** dem eigentlichen Testen<sup>12</sup> durchgeführt. In Code 2.4 ist ein Testdurchlauf mit `> make test` zu sehen. Wobei **Verified: 50/50** anzeigt, wieviele der Tests **verifizierbar** sind<sup>13</sup>, also beim **GCC** ohne Fehlermeldung durchlaufen, **Not verified:** die **nicht verifizierbaren** Tests angibt, **Running through: 88 / 88** anzeigt wieviele Tests mit dem **PicoC-Compiler** durchlaufen, **Not running through:** die **nicht** durchlaufenden Tests angibt, **Passed: 88 / 88** zeigt bei wievielen Tests die Ausgabe mit der erwarteten Ausgabe **identisch** ist, **Not passed:** die Tests anzeigt, bei denen das **nicht** der Fall ist.

<sup>9</sup>History - GCC Wiki.

<sup>10</sup>In der langen **Bestehenszeit** und bei der **vielen Verwendung** wurden die **allermeisten kritischen Bugs** wahrscheinlich schon gefunden.

<sup>11</sup>Es besteht allerdings immer eine **Chance**, dass die Ausgabe für den Test nur **zufällig** übereinstimmt. Diese Chance kann allerdings durch **vernünftige Definition** des Tests sehr **gering** gehalten werden.

<sup>12</sup>Prüfen, ob der interpretierte RETI-Code des PicoC-Compilers die **gleiche Ausgabe** hat, wie der Schreiber des Tests **erwartet**.

<sup>13</sup>Also **alle** Tests aus den **Kategorien basic, advanced, hard und example**.



```

> make test
=====
= ./tests/basic_array_init.picoc =
=====
...
=====
=          Verification          =
=====
./tests/basic_array_init.c
...
=====
=          Results              =
=====
Verified: 50 / 50
Not verified:
Running through: 88 / 88
Not running through:
Passed: 88 / 88
Not passed:

```

Code 2.4: Testdurchlauf

Der Befehl `make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` TESTNAME, VERBOSE und DEBUG aus Tabelle 2.2 kombinieren.

Das Pythonscript `/convert_to_c.py` ist notwendig, da  $L_{PicoC}$  sich bei den Funktionen `print()` und `input()` von der **Syntax** der Sprache  $L_C$  unterscheidet, bei der z.B. `printf("%d", 12)` anstelle von `print(12)` geschrieben werden muss. Für die Sprache  $L_{PicoC}$  erfüllen die Funktionen `print()` und `input()` allerdings nur den **Zweck**, dass sie zum **Testen des Compilers** gebraucht werden, um über die Funktion `input()` für eine bestimmte **Eingabe** die **Ausgabe** über die Funktion `print()` testen zu können. Aus diesem Grund ist es notwendig die **Syntax** dieser Funktionen in  $L_C$  zu übersetzen.

Die Funktion `print(<exp>)` wird vom Pythonscript `convert_to_c.py` zu `printf("%d", <exp>)` übersetzt. Zuvor muss über `#include<stdio.h>` die **Standard-Input-Output Bibliothek** `<stdio.h>` eingebunden werden. Bei der Funktion `input()` wurde **nicht** der aufwändige **Umweg** genommen die Funktion `input()` durch ihre entsprechende Funktion in der Sprache  $L_C$  zu ersetzen. Es geht viel direkter, indem **nacheinander** die `input()`-Funktionen durch entsprechende Eingaben aus der Datei `<program>.in` ersetzt werden. Man schreibt einfach **direkt** den Wert hin, den die `input()`-Funktionen normalerweise einlesen sollten.

## 2.3 Erweiterungsideen

Mit dem **Funktionsumfang** des **PicoC-Compilers**, der in Unterkapitel 2.2 erläutert wurde muss allerdings das Ende der Fahnenstange noch **nicht** erreicht sein. Weitere Ideen, die in den **PicoC-Compiler**<sup>14</sup> implementiert werden könnten, wären:

- **Tail Call:** Wenn ein Funktionsaufruf das **letzte** Statement in einem Funktionsblock ist, wird der Stackframe dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion zurückgekehrt werden muss<sup>15</sup>. Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden.

<sup>14</sup>Möglicherweise ja im Rahmen eines **Masterprojektes** 😊.

<sup>15</sup>Was der Grund ist, warum ein **Stackframe** überhaupt angelegt wird, damit später beim **Rücksprung** aus der **aufgerufenen Funktion** die Ausführung mit allen Variablen, wie **vor der Ausführung** fortgesetzt werden kann.



bevor der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die nur Tail Calls ausführt nur eine **konstante Menge** an **Speicherplatz** auf dem Stack verbraucht. In Code 2.5 sind **zwei Tail Calls** markiert.

- **Partial Evaluator:** Bei Ausdrücken wie `4 + input() - 2`, `input() * 1` oder `0 + input() * 2` können **Teilausdrücke** bereits **während** des **Kompilierens** zu `2 + input()`, `input()` und `input() * 2` berechnet werden. Die kann durch einen neuen **PicoC-Eval Pass** umgesetzt werden, der **vor** oder **nach** dem **PicoC-Shrink Pass** den Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache  $L_{PicoC\_Eval}$  umformt. In der Abstrakten Syntax der Sprache  $L_{PicoC\_Eval}$  sind **binäre Operationen** zwischen zwei `Num(str)`-PicoC-Knoten **nicht möglich**. Diese **Vorberechnung** kann auch auf **Konstanten** und **Variablen** ausgeweitet werden. Der **Vorteil** ist, dass hierdurch weniger **RETI-Code** produziert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.
- **Lazy Evaluation:** sdasdf
- **Garbage Collector:**
- **Array Länge vorne speichern:**
- **PicoPython:** super einfach ein PicoPython zu machen von der Syntax her durch auswechseln der Grammatik
- **Graph Coloring:** richtigen Compiler mit Graph Coloring machen, liveness analyse
- **Debugger:** Debugger Informationen rein machen
- **Linker:** Linker und weitere Dateien
- **Call by reference mit &:**
- **Objectorientierung:** Structs weiter ausbauen
- **external keyword:** Structs weiter ausbauen
- **#include von Bibliotheken, mehrere Dateien, Headerdateien .h:** sdasdf, print über Uart umsetzen
- **Richtiger Compiler in binärer Repräsentation 1010101:** asdf
- **Boolean:** asdff
- **Struct Spilling oder wie das hieß:** asd
- **malloc:** implementieren

```

1 // in:42
2 // expected:0
3
4 int ret0() {
5     return 0;
6 }
7
8 int ret1() {
9     return 1;

```

```
10 }
11
12 int tail_call_fun(int bool_val) {
13     if (bool_val) {
14         return ret0();
15     }
16     return ret1();
17 }
18
19 void main() {
20     print(tail_call_fun(input()));
21 }
```

**Code 2.5:** *Beispiel für Tail Call*

asdf

---

---

# Literatur

## Online

- *A-Normalization: Why and How (with code)*. URL: <https://matt.might.net/articles/a-normalization/> (besucht am 23.07.2022).
- *Errors in C/C++ - GeeksforGeeks*. URL: <https://www.geeksforgeeks.org/errors-in-cc/> (besucht am 10.05.2022).
- *History - GCC Wiki*. URL: <https://gcc.gnu.org/wiki/History> (besucht am 06.08.2022).
- *Home - Neovim*. URL: <http://neovim.io/> (besucht am 04.08.2022).
- *JSON parser - Tutorial — Lark documentation*. URL: [https://lark-parser.readthedocs.io/en/latest/json\\_tutorial.html](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).
- *What is Bottom-up Parsing?* URL: <https://www.tutorialspoint.com/what-is-bottom-up-parsing> (besucht am 22.06.2022).
- *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).

## Artikel

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

---

---

## Vorlesungen

- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index\\_de.html](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](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157) (besucht am 09.07.2022).
- 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](https://doi.org/10.1145/1596638.1596640). URL: <http://portal.acm.org/citation.cfm?doid=1596638.1596640> (besucht am 23.07.2022).
  - *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).
-