

---

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 Motivation</b>	<b>1</b>
1.1 RETI . . . . .	1
1.2 PicoC . . . . .	1
1.3 Aufgabenstellung . . . . .	1
1.4 Eigenheiten der Sprache C . . . . .	1
1.5 Richtlinien . . . . .	2
<b>2 Einführung</b>	<b>3</b>
2.1 Compiler und Interpreter . . . . .	3
2.1.1 T-Diagramme . . . . .	6
2.2 Formale Sprachen . . . . .	8
2.2.1 Ableitungen . . . . .	11
2.2.2 Mehrdeutige Grammatiken . . . . .	14
2.2.3 Präzidenz und Assoziativität . . . . .	14
2.3 Lexikalische Analyse . . . . .	15
2.4 Syntaktische Analyse . . . . .	18
2.5 Code Generierung . . . . .	24
2.5.1 Monadische Normalform . . . . .	25
2.5.2 A-Normalform . . . . .	26
2.5.3 Ausgabe des Maschinenencodes . . . . .	28
2.6 Fehlermeldungen . . . . .	29
2.6.1 Kategorien von Fehlermeldungen . . . . .	29
<b>Literatur</b>	<b>A</b>

---

---

# Abbildungsverzeichnis

2.1	Horizontale Übersetzungszwischenschritte zusammenfassen . . . . .	8
2.2	Vertikale Interpretierungszwischenschritte zusammenfassen . . . . .	8
2.3	Veranschaulichung von Linksassoziativität und Rechtsassoziativität . . . . .	15
2.4	Veranschaulichung von Präzidenz . . . . .	15
2.5	Veranschaulichung der Lexikalischen Analyse . . . . .	18
2.6	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum. . . . .	22
2.7	Veranschaulichung der Syntaktischen Analyse . . . . .	23
2.8	Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten . . . . .	26
2.9	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen . . . . .	28

---

---

# Codeverzeichnis

---

---

# Tabellenverzeichnis

---

---

# Definitionsverzeichnis

1.1	Caller-save Register	1
1.2	Callee-save Register	1
1.3	Deklaration	1
1.4	Definition	1
1.5	Allokation	1
1.6	Initialisierung	2
1.7	Scope	2
1.8	Call by value	2
1.9	Call by reference	2
2.1	Interpreter	3
2.2	Compiler	3
2.3	Maschinensprache	4
2.4	Assemblersprache (bzw. engl. Assembly Language)	4
2.5	Assembler	5
2.6	Objectcode	5
2.7	Linker	5
2.8	Immediate	5
2.9	Transpiler (bzw. Source-to-source Compiler)	6
2.10	Cross-Compiler	6
2.11	T-Diagram Programm	6
2.12	T-Diagram Übersetzer (bzw. eng. Translator)	7
2.13	T-Diagram Interpreter	7
2.14	T-Diagram Maschine	7
2.15	Symbol	8
2.16	Alphabet	9
2.17	Wort	9
2.18	Formale Sprache	9
2.19	Syntax	9
2.20	Semantik	9
2.21	Formale Grammatik	9
2.22	Chromsky Hierarchie	10
2.23	Reguläre Grammatik	11
2.24	Kontextfreie Grammatik	11
2.25	Wortproblem	11
2.26	1-Schritt-Ableitungsrelation	12
2.27	Ableitungsrelation	12
2.28	Links- und Rechtsableitungableitung	12
2.29	Linksrekursive Grammatiken	12
2.30	Formaler Ableitungsbaum	13
2.31	Mehrdeutige Grammatik	14
2.32	LL(k)-Grammatik	14
2.33	Assoziativität	14
2.34	Präzidenz	15
2.35	Pipe-Filter Architekturpattern	15
2.36	Pattern	16
2.37	Lexeme	16
2.38	Lexer (bzw. Scanner oder auch Tokenizer)	16

---

---

2.39	Bezeichner (bzw. Identifier)	17
2.40	Literal	18
2.41	Konkrete Syntax	19
2.42	Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)	19
2.43	Parser	19
2.44	Recognizer (bzw. Erkennen)	20
2.45	Rekursiver Abstieg	21
2.46	Transformer	21
2.47	Visitor	21
2.48	Abstrakte Syntax	21
2.49	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)	22
2.50	Pass	24
2.51	Reiner Ausdruck (bzw. engl. pure expression)	25
2.52	Unreiner Ausdruck	25
2.53	Monadische Normalform (bzw. engl. monadic normal form)	25
2.54	Location	26
2.55	Atomarer Ausdruck	27
2.56	Komplexer Ausdruck	27
2.57	A-Normalform (ANF)	27
2.58	Fehlermeldung	29

---



---

---

# Grammatikverzeichnis

2.1	Produktionen des Ableitungsbaums . . . . .	13
-----	--	----

---

---

# 1 Motivation

## 1.1 RETI

... basiert auf ... der Vorlesung C. Scholl, „Betriebssysteme“.

### Definition 1.1: Caller-save Register

$a$

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

### Definition 1.2: Callee-save Register

$a$

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

## 1.2 PicoC

## 1.3 Aufgabenstellung

## 1.4 Eigenheiten der Sprache C

### Definition 1.3: Deklaration

$a$

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

### Definition 1.4: Definition

$a$

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

### Definition 1.5: Allokation

$a$

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

**Definition 1.6: Initialisierung***a*<sup>a</sup>Thiemann, „Einführung in die Programmierung“.**Definition 1.7: Scope***a*<sup>a</sup>Thiemann, „Einführung in die Programmierung“.**Definition 1.8: Call by value***a*<sup>a</sup>Bast, „Programmieren in C“.**Definition 1.9: Call by reference***a*<sup>a</sup>Bast, „Programmieren in C“.

## 1.5 Richtlinien

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

Beim **PicoC-Compiler** wurde eher darauf Wert gelegt **sauberen, strukturierten Code** zu schreiben, den die Studenten sogar selber verstehen könnten und eine **unkomplizierte Bibliothek mit guter Dokumentation**<sup>1</sup>, nämlich das **Lark Parsing Toolkit**<sup>2</sup> für das **Parsen** zu verwenden. Vor allem, da zu erwarten ist, dass der **PicoC-Compiler** vielleicht in einigen anderen Projekten eingebunden werden könnte, ist es von **Vorteil** bei der Notwendigkeit kleiner **Erweiterungen**, diese Erweiterungen **unkompliziert** durchführen zu können.

<sup>1</sup>[Welcome to Lark's documentation! — Lark documentation.](#)

<sup>2</sup>[Lark - a parsing toolkit for Python.](#)

# 2 Einführung

## 2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache**  $L_{PicoC}$  in die **RETI-Sprache**  $L_{RETI}$  das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**<sup>1</sup> und von **Tests** die **Beziehungen** in 2.50.1 zu belegen (siehe Subkapitel ??).

### Definition 2.1: Interpreter

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

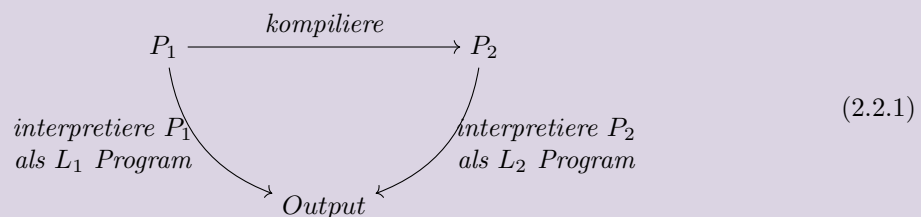
*Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen **Sub-Bäumen** des **Abstrakter Syntaxbaum** (Definition 2.49) und führt je nach Komposition der **Nodes** des Abstrakter Syntaxbaum, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.<sup>a</sup>*

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

### Definition 2.2: Compiler

***Kompiliert** ein 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 das Program  $P_1$  so in das Program  $P_2$  übersetzt wird, dass bei beiden Programmen, wenn sie von **Interpretern** ihrer jeweiligen Sprachen  $L_1$  und  $L_2$  **interpretiert** werden, der gleiche **Output** rauskommt. Also beide Programme  $P_1$  und  $P_2$  die gleiche **Semantik** haben und sich nur **syntaktisch** 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 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

### Definition 2.3: Maschinensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, 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 2.8) haben.

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

### Definition 2.4: 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. Scholl, „Einführung in Embedded Systems“.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschinenenncode** bzw. **Objectcode** (Definition 2.6) 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 2.7) zu Maschiendencode zusammengesetzt wird ausführbar

ist.

### Definition 2.5: 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. Scholl, „Einführung in Embedded Systems“.

### Definition 2.6: 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 **Maschiencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschinencode zusammengesetzt hat.<sup>a</sup>

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

### Definition 2.7: 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. 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 2.8) 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 2.8: 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 2.9: 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 2.10: 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>

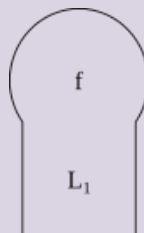
**2.1.1 T-Diagramme**

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 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschine (Definition 2.14) zusammen.

**Definition 2.11: 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“.

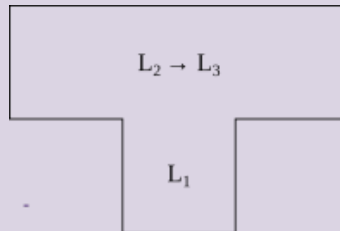
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein  $L$  dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

<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 2.12: 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 2.50.1.<sup>b</sup>

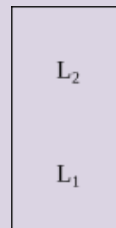


<sup>a</sup>Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren**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

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

**Definition 2.13: T-Diagramm Interpreter**

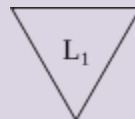
Repräsentiert einen **Interpreter**, der in der **Sprache**  $L_1$  geschrieben ist und **Programme** in der **Sprache**  $L_2$  interpretiert.<sup>a</sup>



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

**Definition 2.14: 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 2.1 und 2.2 erkennen kann zusammenfassen.



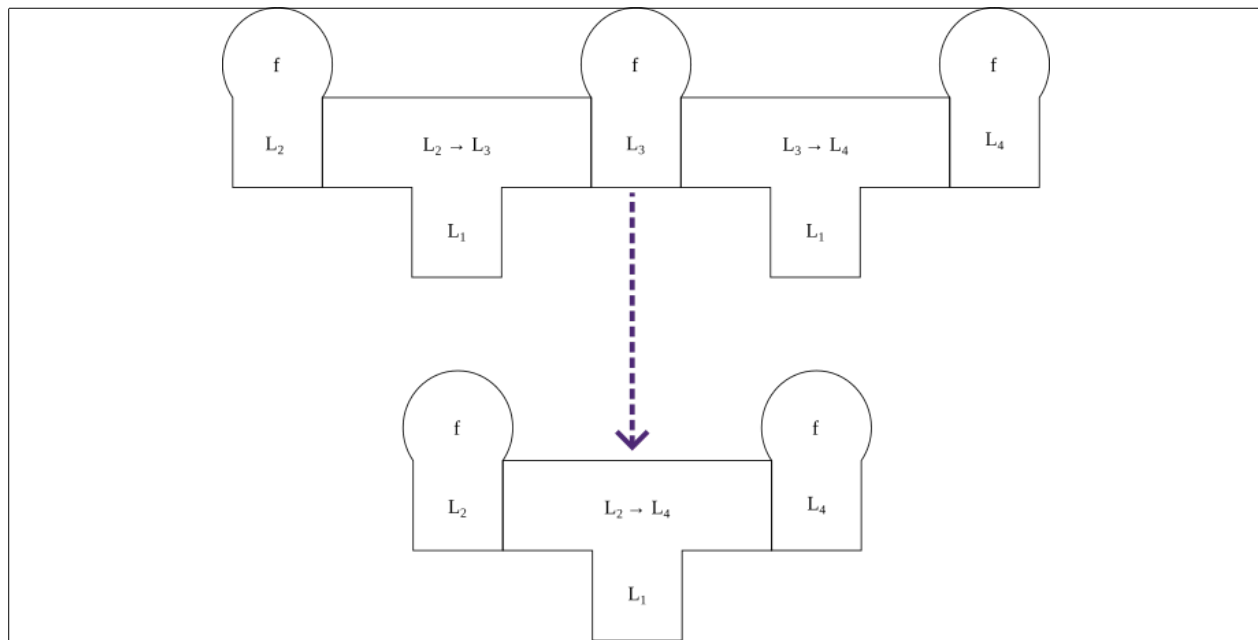


Abbildung 2.1: Horizontale Übersetzungszwischenschritte zusammenfassen

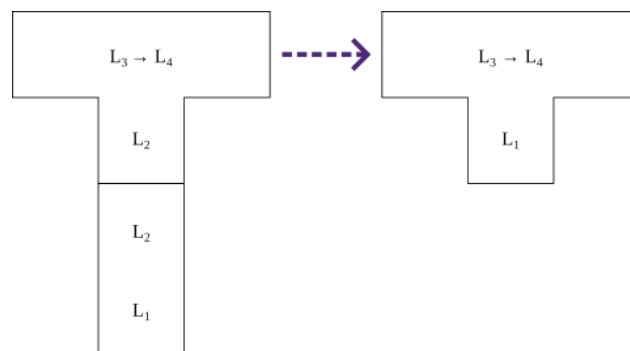


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

## 2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.18) 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 2.15), **Alphabet** (Definition 2.16), **Wort** (Definition 2.17) usw. beinhaltet vorher eingeführt zu haben.

### Definition 2.15: Symbol

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

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

**Definition 2.16: Alphabet**

„Ein Alphabet ist eine **endliche, nicht-leere** Menge aus **Symbolen** (Definition 2.15).“<sup>a</sup>

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

**Definition 2.17: 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 2.18: Formale Sprache**

„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.17) über dem **Alphabet**  $\Sigma$  (Definition 2.16).“<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 2.20) **gleich** bleiben. Beide Sprachen  $L_1$  und  $L_2$  haben eine **Grammatik** (Definition 2.21), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 2.19) haben.

**Definition 2.19: 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 2.20: 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 2.21: 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**.

- $\Sigma \hat{=} \text{Terminalsymbole}$ , 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>de</sup>
- $S \hat{=} \text{Startsymbol}$ , 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.  $w, v \in V^* \wedge w \notin \Sigma^*$ .

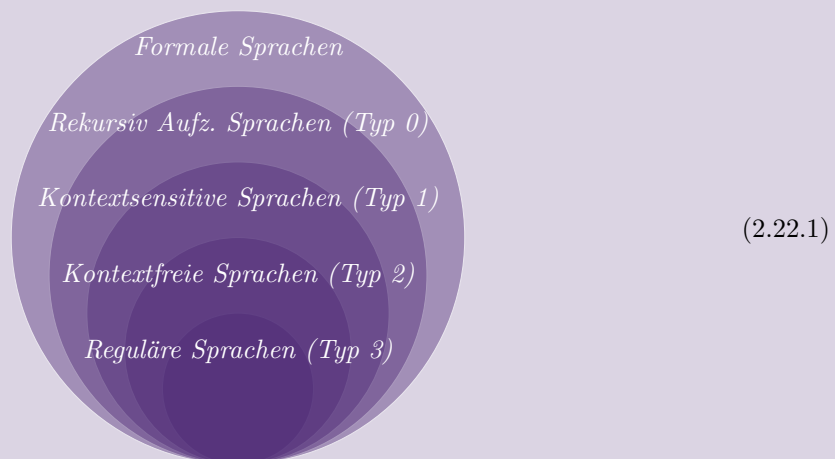
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 2.22) einteilen.

### Definition 2.22: 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 2.22.1 besteht eine **echte Teilmengebeziehung**:  $L_3 \subset L_2 \subset L_1 \subset L_0$ . 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 2.23) und **Kontextfreien Grammatiken** (Definition 2.24) beschrieben werden.

**Definition 2.23: Reguläre Grammatik**

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

$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \varepsilon \quad (2.23.1)$$

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 2.24: Kontextfreie Grammatik**

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

$$A \rightarrow v \quad (2.24.1)$$

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 2.25). 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 2.25: 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:

$$\mathbb{1}_L : \Sigma^* \rightarrow \{0, 1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \quad (2.25.1)$$

<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.

**2.2.1 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 2.26) und der normalen **Ableitungsrelation** (Definition 2.27) unterscheiden.

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

**Definition 2.26: 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 2.27: 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 2.28) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 2.4 relevant.

**Definition 2.28: 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 2.29) ist<sup>6</sup>.

**Definition 2.29: 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 2.2.2) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 2.30) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 2.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 2.42) als eine **compilerinterne Datenstruktur** umgesetzt werden.

### Definition 2.30: 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 2.21) 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 2.2.1 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung 2.30.1** nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition ??) angegebenen **Produktionen 2.1** einer ansonsten nicht näher spezifizierten **Grammatik**  $G = \langle N, \Sigma, P, add \rangle$  ergibt.

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

**Grammar 2.1:** Produktionen des Ableitungsbaums

$$add \Rightarrow mul \Rightarrow mul \text{ "*" } NUM \Rightarrow NUM \text{ "*" } NUM \Rightarrow 4 \text{ "*" } NUM \Rightarrow 4 \text{ "*" } 2 \quad (2.30.1)$$



### 2.2.2 Mehrdeutige Grammatiken

Für einen Compiler ist es notwendig, dass die **Grammatik**, welche die **Konkrete Syntax** beschreibt keine **Mehrdeutige Grammatik** (Definition 2.31) ist, dann sonst können unter anderem die **Präzidenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel ?? an einem Beispiel demonstriert wird.

#### Definition 2.31: 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 **Parzen** eines Programmes, ist es notwendig eine **LL(k)-Grammatik** (Definition 2.32) vorliegen zu haben. Bei diesen Ansätzen, die meist die Methode des **Rekursiven Abstiegs** (Definition 2.45) verwenden lässt sich eine bessere minimale **Laufzeit** garantieren, da aufgrund der **LL(k)-Eigenschaft** ausgeschlossen werden kann, dass **Backtracking** notwendig ist<sup>7</sup>.

#### Definition 2.32: 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>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“.

### 2.2.3 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** (Definition 2.33) und **Präzidenz** (Definition 2.34) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel ?? genauer erklärt.

#### Definition 2.33: 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 2.3 mithilfe von Klammern  $()$  veranschaulicht.

<sup>7</sup>Mehr **Erklärung** hierzu findet sich im Unterkapitel 2.4.

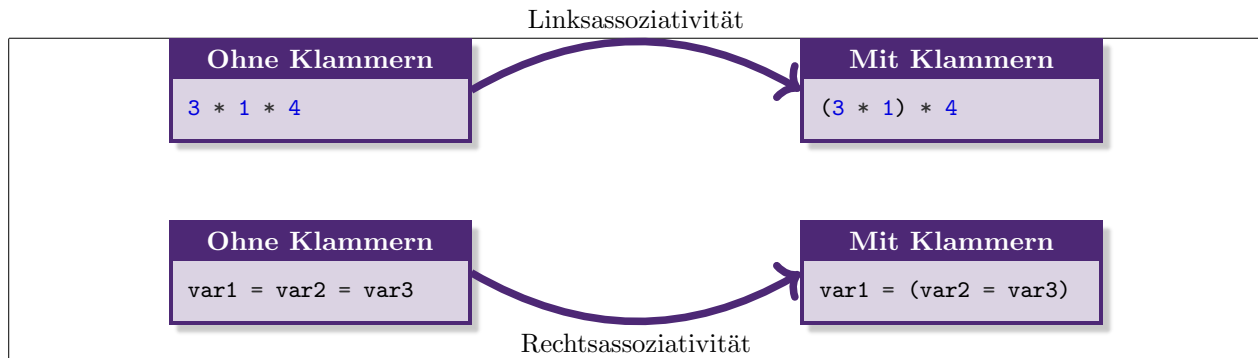


Abbildung 2.3: Veranschaulichung von Linksassoziativität und Rechtsassoziativität

**Definition 2.34: 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 2.4 mithilfe der Klammern `()` veranschaulicht. Im Beispiel in Abbildung 2.4 ist bei den beiden **Subtraktionsoperatoren** `-` nacheinander und dem darauffolgenden **Multiplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzidenz** im Spiel.



Abbildung 2.4: Veranschaulichung von Präzidenz

## 2.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.35) 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 2.36) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

**Definition 2.35: 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>ab</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 2.37) genannt.

### Definition 2.36: 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 2.37: 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 2.38) 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 2.38) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

### Definition 2.38: 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>Thiemann, „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 2.39) 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 2.39: 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>Thiemann, „Einführung in die Programmierung“.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`<sup>8</sup> und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort  $\epsilon$  zuordnet. Das ist auch im Sinne der Definition, denn  $\epsilon \in (N \times W)^*$  ist immer der Fall 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 beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`<sup>9</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 2.40) 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>10</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>11</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 2.23 erfüllt. Einzig die Produktion `CHAR ::= "'ASCII_CHAR'"` sieht problematisch aus, kann allerdings auch als `{CHAR ::= "'CHAR2', CHAR2 ::= ASCII_CHAR"}` **regulär** ausgedrückt werden<sup>12</sup>. Somit existiert eine **reguläre Grammatik**, welche die **Sprache**  $L_{PicoC\_Lex}$  beschreibt und damit ist die **Sprache**  $L_{PicoC\_Lex}$  **regulär**.

<sup>8</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.

<sup>9</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.

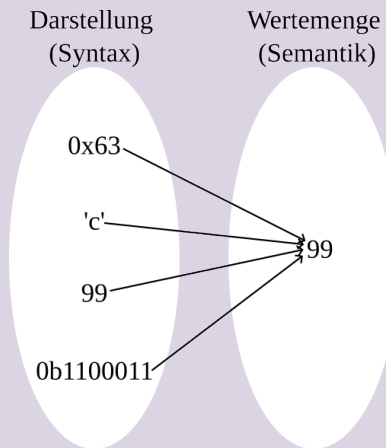
<sup>10</sup>Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

<sup>11</sup>Man nennt das auch einem **Lookahead** von 1

<sup>12</sup>Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten lassen.

**Definition 2.40: 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 2.5 die Lexikalische Analyse an einem Beispiel veranschaulicht.

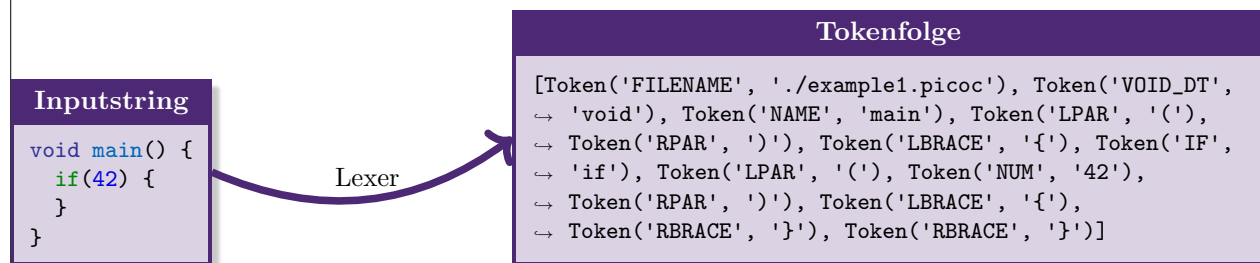


Abbildung 2.5: Veranschaulichung der Lexikalischen Analyse

## 2.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik**  $G_{Parse}$  notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern '(' bzw. öffnende geschweifte Klammern '{' es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer ')' bzw. schließende geschweifte Klammer '}' geschlossen wurden.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 2.41) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 2.43), ein **Ableitungsbaum** (Definition 2.42) generiert, der als Zwischenstufe hin zum einem **Abstrakter Syntaxbaum** (Definition 2.49) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaums** und dann erst des **Abstrakten Syntaxbaumes**.

**Definition 2.41: 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 2.42: Ableitungsbaum (bzw. Konkretter Syntaxbaum, engl. Derivation Tree)**

**Compilerinterne Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 2.30) 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 2.43: 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 2.43 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  $L_{PicoC}$  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 2.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.44), denn ein Parser löst das **Wortproblem** (Definition 2.25) 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 2.44: 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 2.45).

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.29) 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 2.32) 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 **vorausschauen**. **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 2.45: 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 2.46) und **Visitors** (Definition 2.47) 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 2.48).

#### Definition 2.46: 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 2.47: 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 2.48: 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 2.49: 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 2.6



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

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

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.7 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

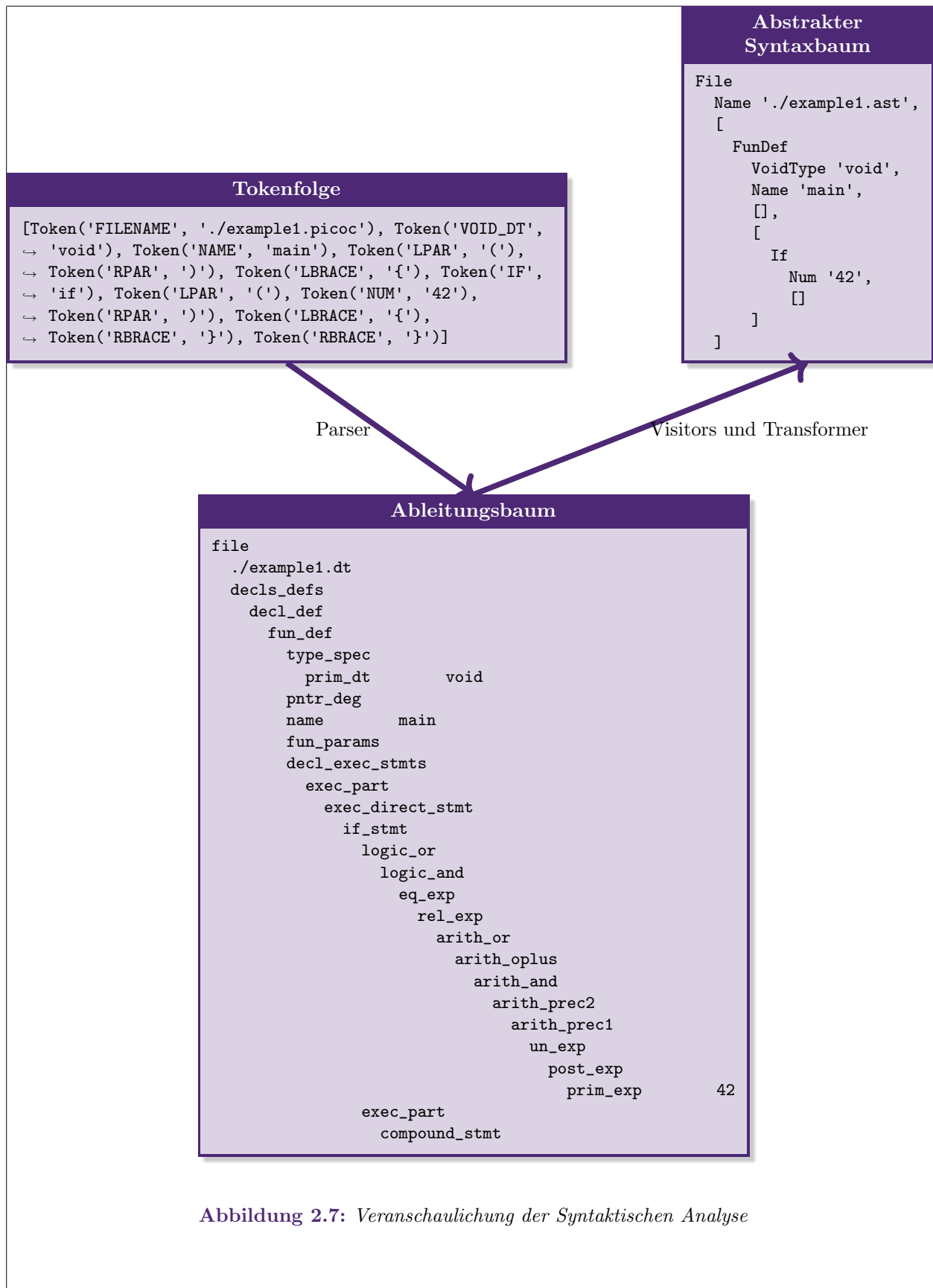


Abbildung 2.7: Veranschaulichung der Syntaktischen Analyse



## 2.5 Code Generierung

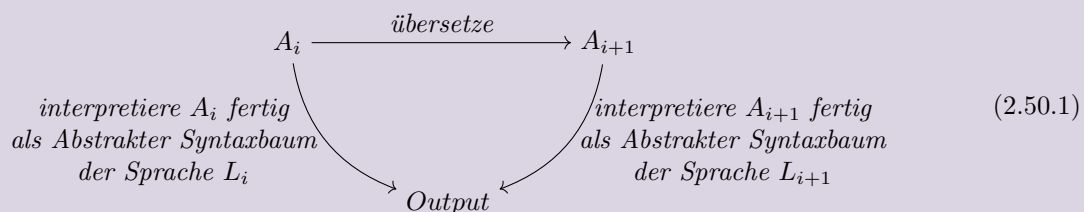
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakter Syntaxbaum** einer Sprache  $L_1$  in den **Abstrakter Syntaxbaum** einer Sprache  $L_2$  umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.50) 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 2.50: Pass

*Einzelner Übersetzungsschritt in einem Kompilervorgang von einem **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** gilt ähnlich, wie bei einem **vollständigen Compiler** in 2.50.1, dass:



wobei man hier so tut, als gäbe es zwei **Interpreter** für die zwei Sprachen  $L_i$  und  $L_{i+1}$ , welche den jeweiligen den **Abstrakter Syntaxbaum**  $A_i$  bzw.  $A_{i+1}$  fertig interpretieren.<sup>cd</sup>

<sup>a</sup>Ein **Pass** kann mit einem **Transpiler 2.9** (Definition 2.9) 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 **Abstrakter 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.

## 2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tun, welche **Unreine Ausdrücke** (Definition 2.52) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.51) 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 2.51: 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 2.52: 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 2.53).

### Definition 2.53: 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>*

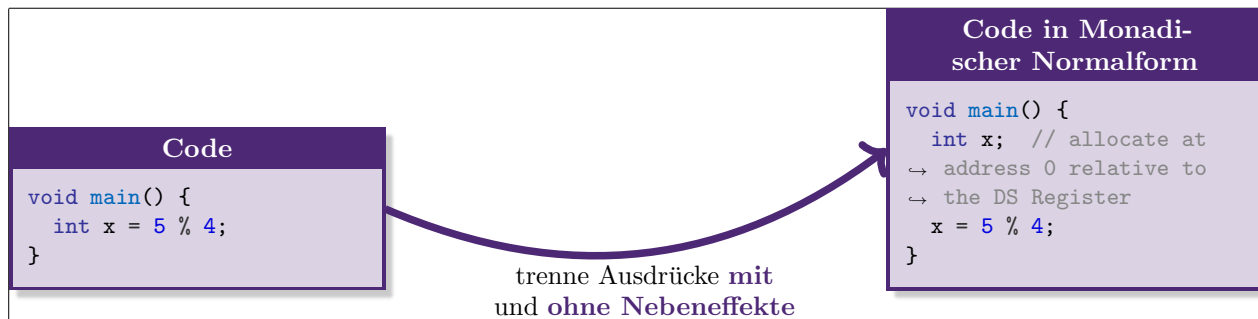
*Eine **Abstrakte Syntax** ist in **Monadischer Normalform**, wenn die **Konkrete Syntax** für welche sie definiert wurde in **Monadischer Normalform** ist.*

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkreten Syntax**<sup>13</sup> aufgeschrieben wurden.

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

<sup>13</sup>Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.



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

Die Aufgabe eines solchen **Passes** ist es, den **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.

## 2.5.2 A-Normalform

Im Falle dessen, dass es sich bei der **Sprache**  $L_1$  um eine **höhere Programmiersprache** und bei  $L_2$  um **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.56) 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 2.54) 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 2.55) 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 2.54: 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 2.57). Wenn eine **Konkrete Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.57), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.55).

**Definition 2.55: 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 2.56: 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 2.57: 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 2.9 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.

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>15</sup>

Aus diesem Grund verwendet das Beispiel in Abbildung 2.9 eine andere Definition für **Komplexe** und **Atomare Ausdrücke**, da dieses Beispiel, um später keine Verwirrung zu erzeugen der Art nachempfunden ist, wie im **PicoC-ANF Pass** der **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.

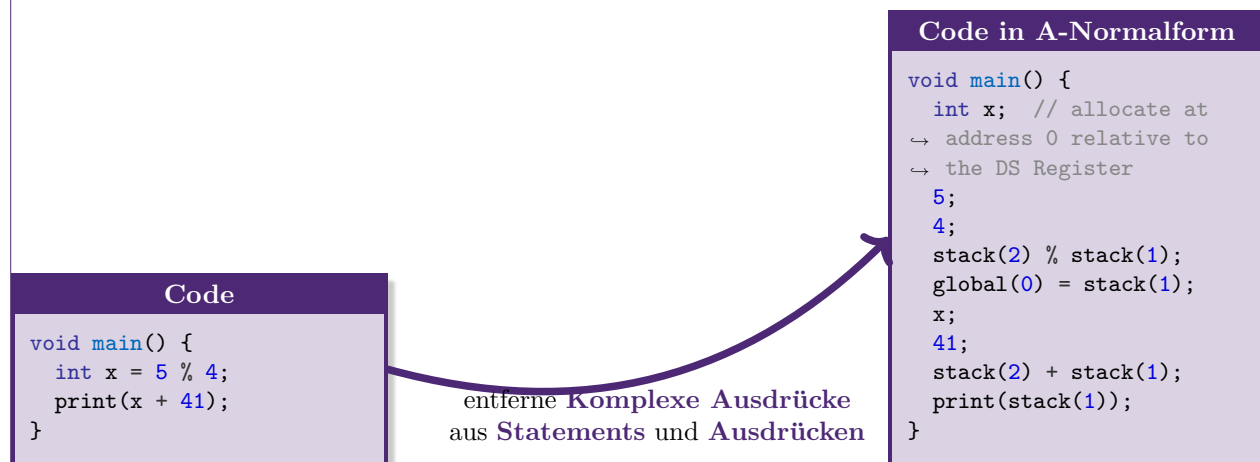
Im Fall, dass **Register** für z.B. **temporäre Zwischenergebnisse** genutzt werden und der **Maschinen-**

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

<sup>15</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>16</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>17</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 2.9:** 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 **einfachen Maschinenbefehlen** syntaktisch ähnlicher sind. Des Weiteren **vereinfacht** dieser Pass die **Implementierung** der nachfolgenden Passes enorm, da Statements z.B. nur noch die Form **global(rel\_addr) = stack(1)** haben, die viel **einfacher verarbeitet** werden kann.

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

### 2.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakter Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt<sup>18</sup>. Der Weg von **Abstrakter Syntax** zu **Konkreter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax**

<sup>16</sup>Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.

<sup>17</sup>Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. **MULT ACC IN2**.

<sup>18</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**.

## 2.6 Fehlermeldungen

### Definition 2.58: 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.

### 2.6.1 Kategorien von Fehlermeldungen

---

---

# 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).
- *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).
- *Welcome to Lark's documentation! — Lark documentation*. URL: <https://lark-parser.readthedocs.io/en/latest/> (besucht am 31.07.2022).
- *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

- Bast, Hannah. „Programmieren in C“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: <https://ad-wiki.informatik.uni-freiburg.de/teaching/ProgrammierenCplusplusSS2020> (besucht am 09.07.2022).
- Nebel, Prof. Dr. Bernhard. „Theoretische Informatik“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index\\_de.html](http://gki.informatik.uni-freiburg.de/teaching/ss20/info3/index_de.html) (besucht am 09.07.2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: [https://abs.informatik.uni-freiburg.de/src/teach\\_main.php?id=157](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157) (besucht am 09.07.2022).
- Scholl, 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, 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/swtv1> (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).