Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Motivation

1.1 RETI

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

Definition 1.1: Caller-save Register

a

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

Definition 1.2: Callee-save Register

a

[a] 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

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

Definition 1.4: Definition

a

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

Definition 1.5: Allokation

a

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

Definition 1.6: Initialisierung

[a]

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

Definition 1.7: Scope

[a]

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

Definition 1.8: Call by value

[a]

[a]Bast, „Programmieren in C“.

Definition 1.9: Call by reference

[a]

[a]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**[1], nämlich das **Lark Parsing Toolkit**[2] 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.

[1] *Welcome to Lark's documentation! — Lark documentation*.
[2] *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**[1] und von **Tests** die **Beziehungen** in 2.48.1 zu belegen (siehe Subkapitel **??**).

Definition 2.1: Interpreter

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

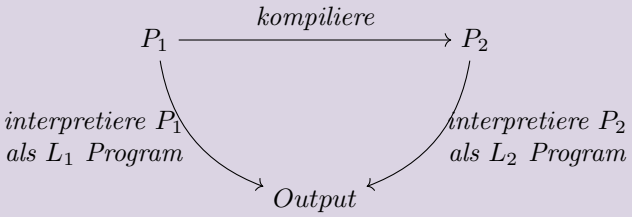
Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 2.47) und führt je nach Komposition der* ***Nodes*** *des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*[a]

[a] 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.*[a]



(2.2.1)

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

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

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

Üblicherweise kompiliert ein **Compiler** ein **Program**, dass in einer **Programmiersprache** geschrieben ist zu **Maschienenncode**, der in **Maschienensprache** (Definition 2.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 2.9) oder **Cross-Compiler** (Definition 2.10). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition 2.4) voneinander zu unterscheiden.

Definition 2.3: Maschienensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** eines möglicherweise zuvor hierzu kompilierten bzw. assemblierten Programmes darstellen. Jeder Maschienenbefehl entspricht einer bestimmten **Aufgabe**, die die CPU im **vereinfachten Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauergesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen konstanten** Anzahl von Fetch- und Execute Phasen im **Komplexeren Fall**. Die Maschienenbefehle sind meist so designed, dass sie sich innerhalb bestimmter **Wortbreiten**, die 2er Potenzen sind codieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.*[a][b]

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschienenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschienenbefehle keine Operanden mit zu großen **Immediates** (Definition 2.8) haben.

[b] C. Scholl, „Betriebssysteme“.

Definition 2.4: Assemblersprache (bzw. engl. Assembly Language)

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

[a] Instructions der Assemblersprache, die mehreren Maschienenbefehlen entsprechen werden auch als **Pseudo-Instructions** bezeichnet und entsprechen dem, was man im allgemeinen als Macro bezeichnet.

[b] Z.B. erlaubt die Assemblersprache des **GCC** für die X_{86_64}**-Architektur** für manche Operanden die Syntax `n(%r)`, die einen **Speicherzugriff** mit **Offset** n zur Adresse, die im **Register** `%r` steht durchführt, wobei z.B. die Klammern `()` usw. nur „syntaktischer Zucker“sind und natürlich nicht mitcodiert werden.

[c] Z.B. sind im X_{86_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.

[d] 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 **Maschienencode** 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 **Maschienencode** 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 **Maschienencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschienensprache** geschrieben ist.*[a]

[a] 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 Maschienenbefehlen in **binärer Repräsentation** auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschienencode zusammengesetzt hat.*[a]

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

Definition 2.7: Linker

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

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

Der **Maschienencode**, denn ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschienenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschiene, 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 Maschienencode, der die Maschienenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, 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[2].

Definition 2.8: Immediate

***Konstanter Wert**, der als **Teil** eines **Maschienenbefehls** gespeichert ist und dessen **Wertebereich** dementsprechend auch durch die die Anzahl an Bits, die ihm innerhalb dieses **Maschienenbefehls** zur Verfügung gestellt sind, **beschränkter** ist als bei sonstigen Werten innerhalb des Hauptspeichers, denen eine ganze Speicherzelle des Hauptspeichers zur Verfügung steht.*[a]

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

[2] Eine **RETI-CPU** zu bauen, die menschenlesbaren Maschienencode 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 Maschienenbefehle übergibt, anstatt z.B. eine unnötig **platzverbrauchenden** UTF-8 Codierung zu verwenden, die nur in sehr vielen Schritt 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*[a][b]

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

Definition 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 **Maschienensprachen** B_1 und B_2 haben.*[a][b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C^{Python}_{PicoC}.
[b] 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 **Maschienensprache** B_2 einer Zielmaschine M_2 läuft.[3]

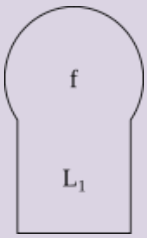
2.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus 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 Program (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschiene (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.*[a]



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

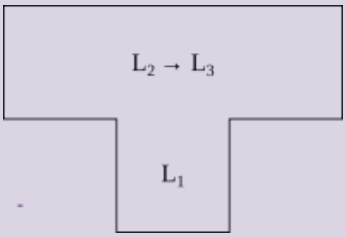
Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 2.11 also reichen einfach eine 1 hinzuschreiben.

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

Definition 2.12: T-Diagram Übersetzer (bzw. eng. Translator)

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

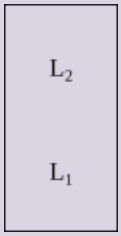
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 2.48.1.*[b]



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

Definition 2.13: T-Diagram Interpreter

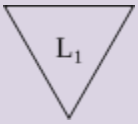
*Repräsentiert einen **Interpreter**, der in der **Sprache** L_1 geschrieben ist und **Programme** in der **Sprache** L_2 interpretiert.*[a]



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

Definition 2.14: T-Diagram Maschiene

*Repräsentiert eine **Maschiene**, welche ein **Programm** in **Maschienensprache** L_1 ausführt.*[a b]



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

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

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

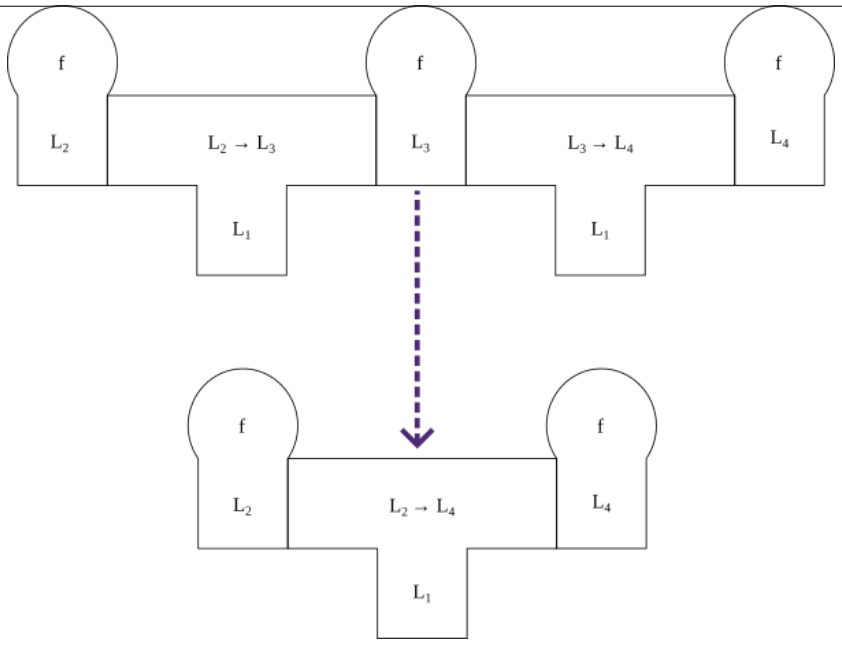


Abbildung 2.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen*

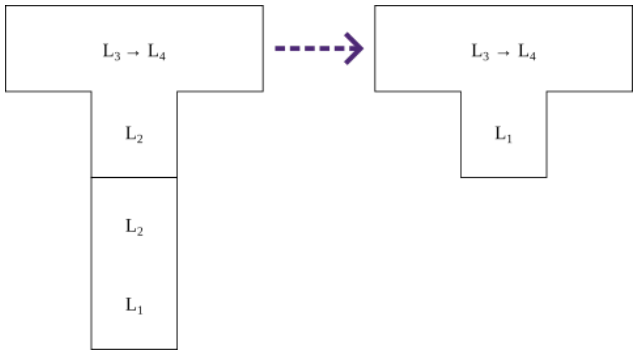


Abbildung 2.2: *Vertikale Interpretierungszwischenschritte zusammenfassen*

2.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 2.18) zu tuen, da bereits das Kompilieren an sich das Übersetzen eines Programmes aus der Sprache L_1 in eine Sprache L_2 ist. Aus diesem Grund ist es wichtig die **Grundlagen Formaler Sprachen** vorher eingeführt zu haben.

Definition 2.15: Symbol

*„Ein Symbol ist ein **Element** eines **Alphabets** Σ.“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.16: Alphatbet

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 2.15).“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 2.17: Wort

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

[a] Nebel, „Theoretische Informatik“.

Definition 2.18: Formale Sprache

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 2.17) über dem **Alphabet** Σ (Definition 2.16).“*[a]

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

[a] Nebel, „Theoretische Informatik“.

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die **Semantik** (Definition 2.20) **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 tuen 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*[a]*.*[b]

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

Definition 2.20: Semantik

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

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

Definition 2.21: Formale Grammatik

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

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

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

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

Zusätzlich ist es praktisch ***Nicht-Terminalsymbole*** N *und* ***Terminalsymbole*** Σ *allgemein als Menge der* ***Grammatiksymbole*** $V = N \cup \Sigma$ *bezeichnen zu können.*

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

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* ***Entscheidungeprobleme****, die in dieser Klasse* ***entscheidbar*** *bzw.* ***unentscheidbar*** *sind usw.*

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

Zwischen den Sprachmengen ***benachbarter Klassen*** *in Abbildung 2.22.1 besteht eine* ***echte Teilmengenbeziehung****: $L_3 \subset L_2 \subset L_1 \subset L_0$. Jede* ***Reguläre Sprache*** *ist auch eine* ***Kontextfreie Sprache****, aber nicht jede* ***Kontextfreie Sprache*** *ist auch eine* ***Reguläre Sprache****.*[a]

(2.22.1)

[a] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 2.23) und der **Kontextfreien Grammatiken** (Definition 2.24) erkannt werden.

Definition 2.23: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{2.23.1}$$

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

[a] Diese Grammatikdefinition ist **rechtsregulär**, es ist auch möglich diese Grammatdefinition **linksregulär** zu fomulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.
[b] Anders ausgedrückt:
[c] Nebel, „Theoretische Informatik“.

Definition 2.24: Kontextfreie Grammatik

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

$$A \to v \tag{2.24.1}$$

haben, wobei A ein Terminalsymbol ist und v"[a]

[a] Nebel, „Theoretische Informatik".

Der Kompiliervorgang lässt sich in viele verschiedene Phasen unterteilen

Definition 2.25: Ableitung

[a]

[a] Nebel, „Theoretische Informatik".

Definition 2.26: Links- und Rechtsableitung

[a]

[a] Nebel, „Theoretische Informatik".

Definition 2.27: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

[a] *Parsing Expressions · Crafting Interpreters*.

2.2.1 Mehrdeutige Grammatiken

Definition 2.28: Ableitungsbaum

[a]

[a] Nebel, „Theoretische Informatik".

Definition 2.29: Mehrdeutige Grammatik

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

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

Definition 2.30: Wortproblem

[a]

[a]Nebel, „Theoretische Informatik“.

Definition 2.31: 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[a]. Dabei steht **LL** für left-to-right und leftmost-derivation, da das **Eingabewort** von **links nach rechts** geparsed und immer **Linksableitungen** genommen werden müssen[b], damit die obige Bedingung mit den **nächsten** k Symbolen gilt.[c]*

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

2.2.2 Präzidenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Grammatik** für eine **Konkrette Syntax** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 2.32) und **Präzidenz** (Definition 2.33) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel **??** genauer erklärt.

Definition 2.32: Assoziativität

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

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

[a]*Parsing Expressions · Crafting Interpreters*.

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

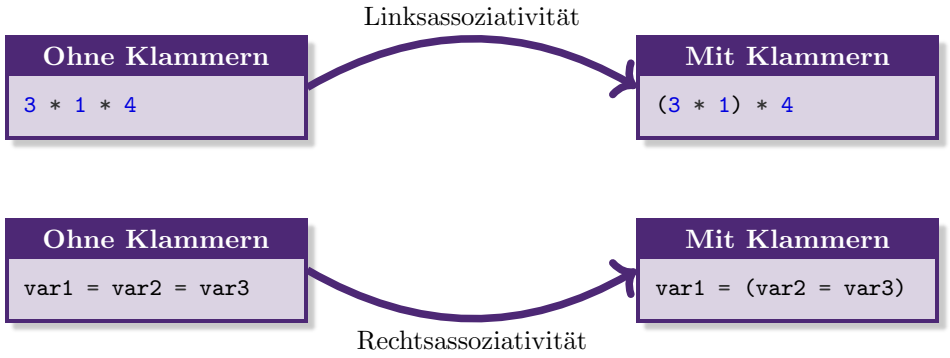


Abbildung 2.3: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität*

Definition 2.33: 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.“*[a]

[a] *Parsing Expressions · Crafting Interpreters.*

Bei **Präzidenz** ist die Mischung der Operatoren für **Subraktion** `'-'` 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 **Multitplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzidenz** im Spiel.

Abbildung 2.4: *Veranschaulichung von Präzidenz*

2.3 Lexikalische Analyse

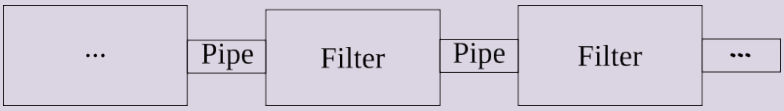
Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb des **Pipe-Filter Architekturpatterns** (Definition 2.34) bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.35) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 2.34: Pipe-Filter Architekturpattern

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

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

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



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

[b] Westphal, „Softwaretechnik“.

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.36) genannt.

Definition 2.35: Pattern

__Beschreibung__ aller möglichen __Lexeme__, die eine Menge $\mathbb{P}_T$ bilden und einem bestimmten __Token__ T zugeordnet werden. Die Menge $\mathbb{P}_T$ ist eine möglicherweise unendliche Menge von __Wörtern__, die sich mit den Produktionen einer __regulären Grammatik__ G_{Lex} einer __regulären Sprache__ L_{Lex} beschreiben lassen [a], die für die Beschreibung eines __Tokens__ T zuständig sind.[b]

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

Definition 2.36: Lexeme

Ein __Lexeme__ ist ein __Wort__ aus dem Inputstring, welches das __Pattern__ für eines der __Token__ T einer __Sprache__ L_{Lex} matched.[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 2.37) im **Inputstring** identifziert und **Tokens** T zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** (Definition 2.37) sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.37: Lexer (bzw. Scanner oder auch Tokenizer)

Ein __Lexer__ ist eine __partielle__ Funktion $lex : \Sigma^* \rightharpoonup (N \times W)^*$, welche ein __Wort__ bzw. __Lexeme__ aus Σ^* 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.[a]

[a] 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 verzubäugen 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 2.38) von **Variablen, Konstanten und Funktionen** die Symbole[a].

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

Definition 2.38: Bezeichner (bzw. Identifier)

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

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

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[4] und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ 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`[5], 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`.

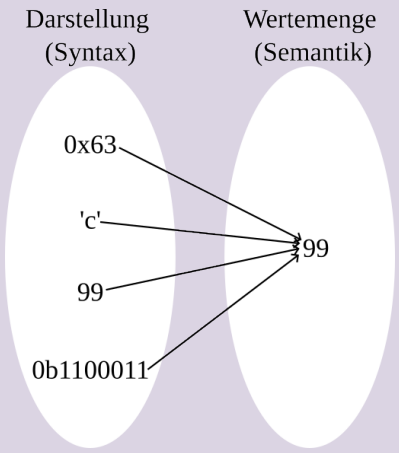
Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 2.39) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[6]. 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[7], 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[8]. Somit existiert eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} **regulär**.

[4] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.
[5] 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.
[6] Die Programmiersprache **Python** erlaubt es z.B. dieser Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.
[7] Man nennt das auch einem **Lookahead** von 1
[8] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

Definition 2.39: Literal

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



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

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

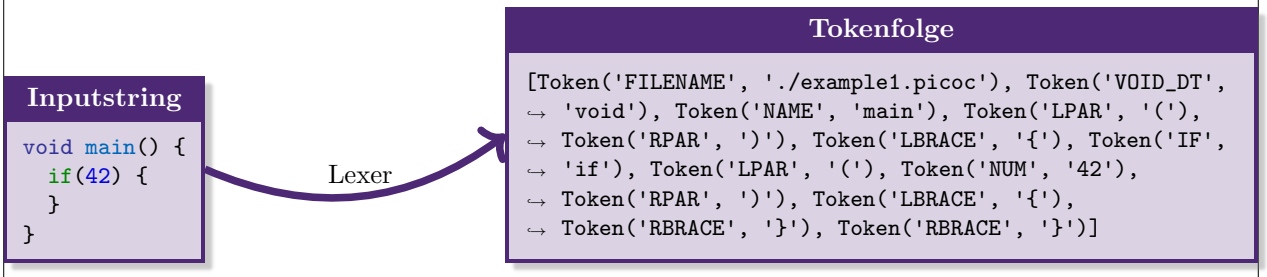


Abbildung 2.5: *Veranschaulichung der Lexikalischen Analyse*

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 der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.40) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.42), ein **Derivation Tree** (Definition 2.41) generiert, der als Zwischenstufe hin zum einem **Abstract Syntax Tree** (Definition 2.47) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann erst des **Abstract Syntax Tree**.

Definition 2.40: Konkrette Syntax

__Syntax__ einer __Sprache__, die durch die __Grammatiken__ G_{Lex} und G_{Parse} zusammengenommen beschrieben wird.

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 __Konkretter Syntax__ aufgeschrieben.[a]

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

Definition 2.41: Derivation Tree (bzw. Parse Tree)

__Compilerinterne Darstellung__ eines in __Konkretter Syntax__ geschriebenen Inputstrings als __Baumdatenstruktur__, in der __Nichtterminalsymbole__ die __Inneren Knoten__ der Baumdatenstruktur und __Terminalsymbole__ die __Blätter__ der Baumdatenstruktur bilden. Jedes zum Ableiten des Inputstrings verwendetete __Nicht-Terminalsymbol__ einer __Produktion__ der __Grammatik__ G_{Parse}, die ein Teil der __Konkrette Syntax__ ist, bildet einen eigenen __Inneren Knoten__.

Der __Derivation Tree__ wird optimalerweise immer so konstruiert bzw. die __Konkrette Syntax__ immer so definiert, dass sich möglichst einfach ein __Abstract Syntax Tree__ daraus konstruieren lässt.[a]

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

Definition 2.42: Parser

Ein __Parser__ ist ein Programm, dass aus einem Inputstring, der in __Konkretter Syntax__ geschrieben ist, eine compilerinterne Darstellung, den __Derivation Tree__ generiert, was auch als __Parsen__ bezeichnet wird[a]*.*[b]

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 2.42 verwendet.
[b] *JSON parser - Tutorial — Lark documentation*.

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

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

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

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

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

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 konkretten 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 2.43), denn ein Parser löst das **Wortproblem** (Definition 2.30) für die **Sprache**, die durch die **Konkrette Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Derivation Tree**.

Definition 2.43: Recognizer (bzw. Erkenner)

*Entspricht dem Maschienenmodell 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 **Konkrette Syntax** ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkretten Syntax** beschrieben wird oder nicht*[a][b]

[a] Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.
[b] Thiemann, „Compilerbau“.

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

- **Top-Down Parsing:** Der **Derivation Tree** 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.[a]

  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**. Dabei 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-Terminal** enthält.

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.27) 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 2.31) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind, was dann bedeutet, dass der **Inputstring** sich **nicht** mit der verwendeten Grammatik

ableiten lässt.[b]

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

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** bzw. **Inputstring** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** der **Konkretten Syntax** rückwärts anzuwenden, bis man beim **Startsymbol** landet.[d]
- **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.[e]

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

Der **Abstract Syntax Tree** wird mithilfe von **Transformern** (Definition 2.44) und **Visitors** (Definition 2.45) 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 einen Inputstring von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.46).

Definition 2.44: Transformer

*Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** besucht und beim Antreffen eines bestimmten Knoten des **Derivation Tree** einen entsprechenden Knoten des **Abstract Syntax Tree** erzeugt und diesen anstelle des Knotens des **Derivation Tree** setzt und so Stück für Stück den **Abstract Syntax Tree** konstruiert.*[a]

[a] *Transformers & Visitors — Lark documentation*.

Definition 2.45: Visitor

*Ein Programm, dass von **unten-nach-oben**, nach dem **Breadth First Search** Prinzip alle Knoten des **Derivation Tree** 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.*[a][b]

[a] Kann theoretisch auch zur Konstruktion eines **Abstract Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstract Syntax Tree** verantwortlich ist. Aber dafür ist ein **Transformer** besser geeignet.
[b] *Transformers & Visitors — Lark documentation*.

Definition 2.46: Abstrakte Syntax

***Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstract Syntax Trees** möglich sind.*

*Jene Produktionen, die in der **Konkretten Syntax** für die Umsetzung von **Präzidenz** notwendig waren, sind in der **Abstrakten Syntax** abgeflacht.*[a]

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

Definition 2.47: Abstract Syntax Tree (AST)

***Compilerinterne Darstellung** eines Programs, in welcher sich anhand der Knoten auf dem **Pfad** von der **Wurzel** zu einem **Blatt** nicht mehr direkt nachvollziehen lässt, durch welche **Produktionen** dieses Blatt abgeleitet wurde.*

*Der **Abstract Syntax Tree** hat einmal den Zweck, dass die **Kompositionen**, die die Knoten bilden können **semantisch** näher an den **Instructions eines Assemblers** dran sind und, dass man mit einem **Abstract Syntax Tree** bei der Betrachtung eines **Knoten**, der für einen Teil des Programms steht, möglichst schnell die Fragen beantworten kann, welche **Funktionalität** der Sprache dieser umsetzt, welche **Bestandteile** er hat und welche **Funktionalität** der Sprache diese Bestandteile umsetzen usw.*[a]

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

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstract Syntax Tree** ermöglicht es die Operationen, die ein Compiler bzw. Interpreter bei der Weiterverarbeitung des Inputstrings 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 2.6 die Syntaktische mit dem Beispiel aus Subkapitel 2.3 fortgeführt.

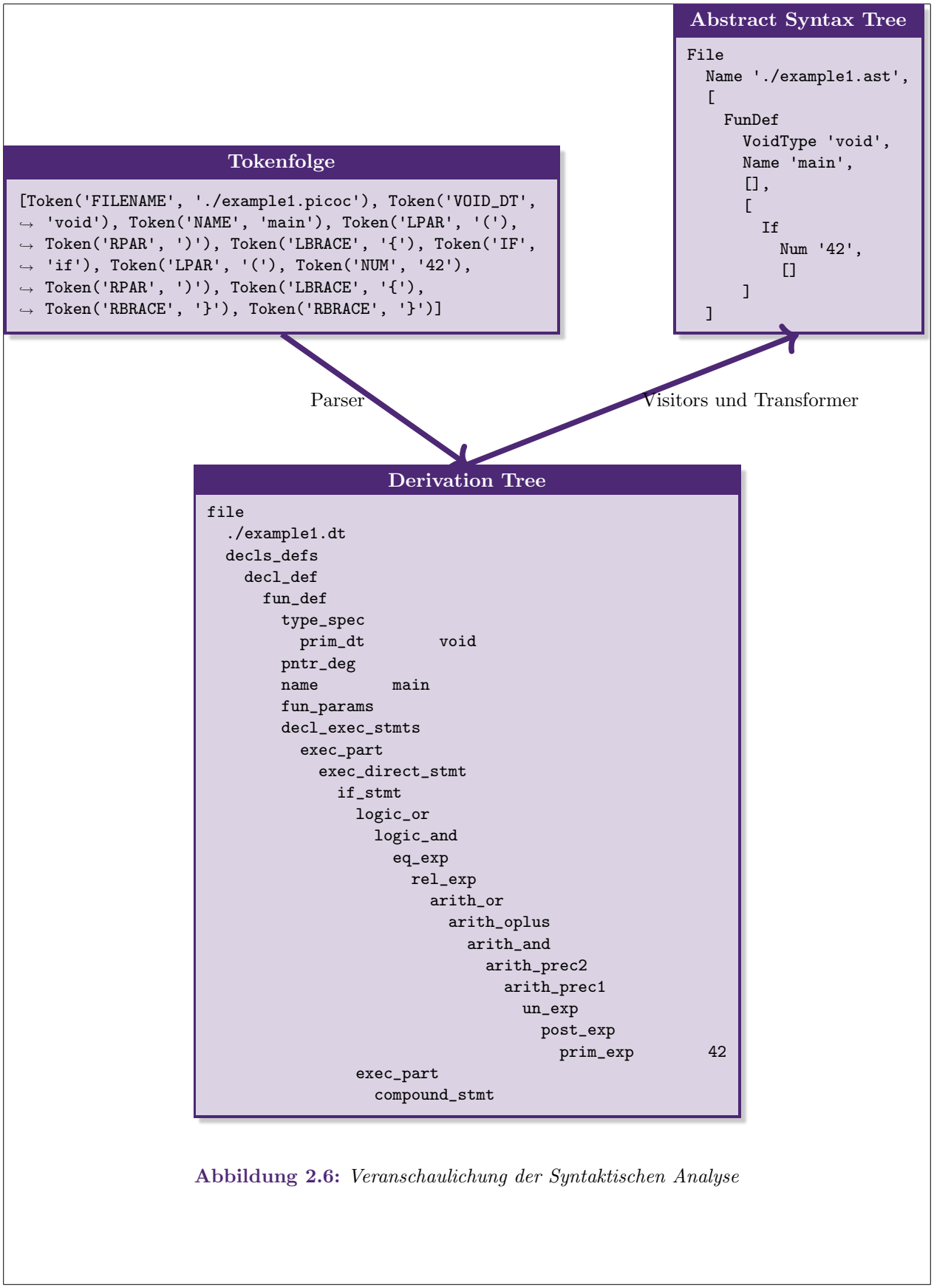


Abbildung 2.6: *Veranschaulichung der Syntaktischen Analyse*

2.5 Code Generierung

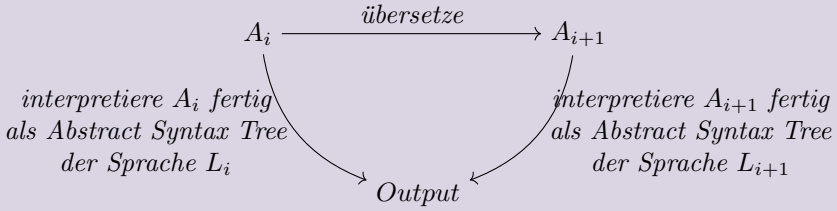
In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstract Syntax Tree** einer Sprache L_1 in den **Abstract Syntax Tree** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 2.48) nennt. So wie es auch schon mit dem **Dervivation Tree** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstract Syntax Tree** kontstruiert hatte. Aus dem Derivation konnte, dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstract Syntax Tree** generiert werden.

Man spricht hier von dem **„Abstrakt Syntax Tree 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 **Abstract Syntax Tree** 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 2.48: Pass

Einzelner Übersetzungsschritt *in einem* ***Kompiliervorgang*** *von einem* ***Abstract Syntax Tree*** A_i *einer Sprache* L_i *zu einem* ***Abstract Syntax Tree*** 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.*[ab]

Für jeden ***Pass*** *gilt ähnlich, wie bei einem* ***vollständigen Compiler*** *in 2.48.1, dass:*



(2.48.1)

wobei man hier so tut, als gäbe es zwei ***Interpreter*** *für die zwei Sprachen* L_i *und* L_{i+1}*, welche den jeweiligen den* ***Abstract Syntax Tree*** A_i *bzw.* A_{i+1} *fertig interpretieren.*[cd]

[a] 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 auschließlich auf **Abstract Syntax Trees**, ohne Parsing usw.

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

[c] **Interpretieren** geht immer von einem Programm in **Konkretter Syntax** aus, wobei der **Abstract Syntax Tree** ein **Zwischenschritt** bei der **Interpretierung** ist.

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

Die von den **Passes** umgeformten **Abstract Syntax Trees** sollten dabei mit jedem **Pass** der **Syntax** von **Maschienenbefehlen** immer ähnlicher werden, bis es schließlich nur noch Maschienenbefehle sind.

2.5.1 Monadische Normalform

Hat man es mit einer Sprache zu tuen, welche **Unreine Ausdrücke** (Definition 2.50) besitzt, so ist es sinnvoll einen **Pass** einzuführen, der **Reine** (Definition 2.49) 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 2.49: Reiner Ausdruck (bzw. engl. pure expression)

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

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

Definition 2.50: Unreiner Ausdruck

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **Monadischer Normalform** (Definiton 2.51).

Definition 2.51: Monadische Normalform (bzw. engl. monadic normal form)

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

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

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

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.7 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[9] aufgeschrieben wurden.

In der Abbildung 2.7 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.

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

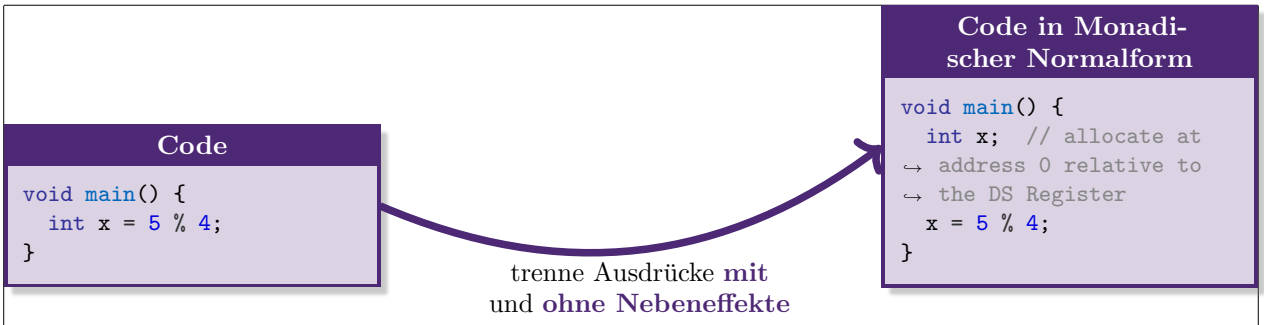


Abbildung 2.7: *Codebeispiel für das Trennen von Ausdrücken mit und ohne Nebeneffekten*

Die Aufgabe eines solchen **Passes** ist es, den **Abstract Syntax Tree** der **Syntax** von **Maschienenbefehlen** anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in **RETI-Knoten** haben. Somit wird eine **Seperation** von Subbäumen, die keine Entsprechung in **RETI-Knoten** haben und denen, die eine haben bewerkstelligt wird. Ein **Reiner Ausdruck** ist **Maschienenbefehlen** 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 **Maschienensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 2.54) 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 2.52) und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Sollte in dem **Statemtent**, indem der **Komplexe Ausdruck** einer **temporären Location** zugewiesen wird, der Komplexe Ausdruck **Teilausdrücke** enthalten, die **komplex** sind, muss die gleiche Prozedur erneut für die **Teilausdrücke** angewandt werden, bis **Komplexe Ausdrücke** nur noch in Statements zur Zuweisung an Locations auftauchen, aber die Komplexen Ausdrücke nur **Atomare Ausdrücke** (Definiton 2.53) 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 2.52: Location

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

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

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

Auf diese Weise sind alle **Statements** und **Ausdrücke** in **A-Normalform** (Definition 2.55). Wenn eine **Konkrette Syntax** in **A-Normalform** ist, ist diese auch automatisch in **Monadischer Normalform** (Definition 2.55), genauso, wie ein **Atomarer Ausdruck** auch ein **Reiner Ausdruck** ist (nach Definition 2.53).

Definition 2.53: Atomarer Ausdruck

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

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

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

Definition 2.54: Komplexer Ausdruck

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

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

Definition 2.55: A-Normalform (ANF)

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

*Eine **Konkrette 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 **Konkrette Syntax** für welche sie definiert wurde in **A-Normalform** ist.*[a][b][c]

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

Ein **Beispiel** für dieses Vorgehen ist in Abbildung 2.8 zu sehen, wo der Einfachheit halber auf die Darstellung in **Abstrakter Syntax** verzichtet wurde und die Codebeispiele in der entsprechenden **Konkretten Syntax**[10] 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.[11]

Aus diesem Grund verwendet das Beispiel in Abbildung 2.8 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 **Abstract Syntax Tree** umgeformt wird. Weil beim PicoC-Compiler **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden, wird nur noch ein **Zugriffen auf den Stack**, wie z.B. `stack('1')` als **Atomarer Ausdrück** angesehen. Dementsprechend werden **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ nun ebenfalls zu den **Komplexen Ausdrücken** gezählt.

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

[10] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[11] 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 verechnen[12], 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[13]. Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse vom **Stack** zu holen, zu **verrechnen** und das Ergebnis wiederum auf den **Stack** zu **speichern** und das `SP`-Register **anzupassen**. Daher werden die **Ausdrücke** für **Zahl** 4, **Variable** *var* und **ASCII-Zeichen** $'c'$ als **Komplexe Ausdrücke** gewertet, da sie niemals in einem **Maschinenbefehl** miteinander verechnet werden können.

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

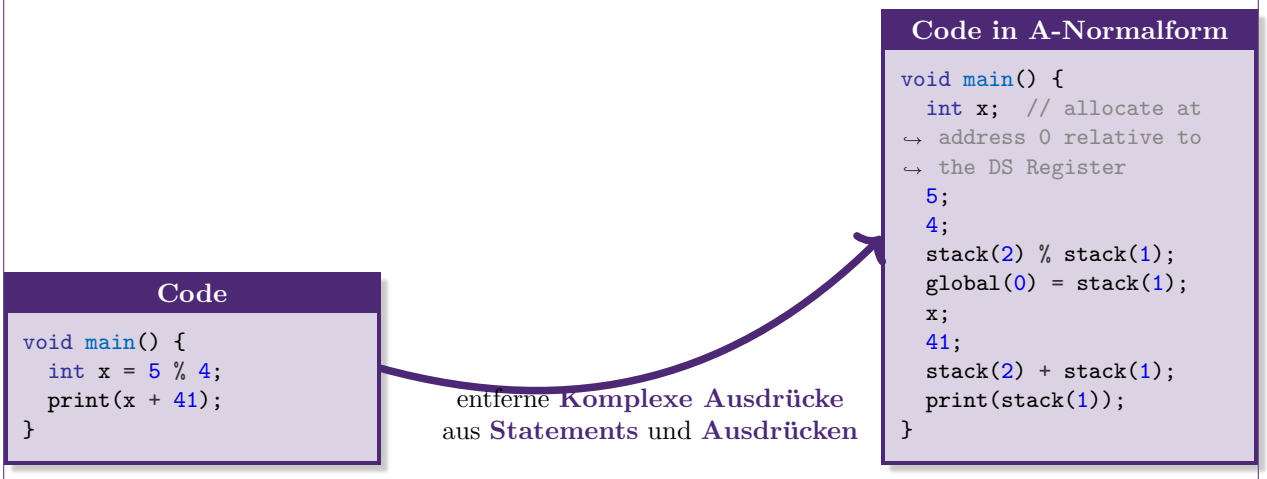


Abbildung 2.8: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen*

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

2.5.3 Ausgabe des Maschienencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstract Syntax Tree** den eigentlichen **Maschinencode** in **Konkretter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt[14]. Der Weg von **Abstrakter Syntax** zu **Konkretter Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkretten Syntax**

[12] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[13] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.
[14] Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschienencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben.

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

Jeder **Knoten** des **Abstract Syntax Trees** 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 **zurückgebeben**.

2.6 Fehlermeldungen

Definition 2.56: Fehlermeldung

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

1. *Ein Program beim* ***Kompilieren*** *von der* ***Konkretten Syntax*** *abweicht, also der* ***Inpustring*** *sich nicht mit der Konrektten Syntax* ***ableiten*** *lässt oder auf etwas* ***zugegriffen*** *werden soll, was noch* ***nicht*** *deklariert oder definiert wurde.*

2. *Beim Ausführen eine* ***verbotene*** *Operation ausgeführt wurde.*[a]

[a] *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 (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 (besucht am 09. 07. 2022).
- Scholl, Christoph. „Betriebssysteme“. Vorlesung. Vorlesung. Universität Freiburg, 2020. URL: https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=157 (besucht am 09. 07. 2022).
- 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/swtvl (besucht am 19. 07. 2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596640. URL: http://portal.acm.org/citation.cfm?doid=1596638.1596640 (besucht am 23. 07. 2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: https://github.com/lark-parser/lark (besucht am 28. 04. 2022).