Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 13. September 2022

Autor:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Theoretische Grundlagen

In diesem Kapitel wird auf die **Theoretischen Grundlagen** eingegangen, die zum Verständnis der **Implementierung** in Kapitel **??** notwendig sind. Zuerst wird in Unterkapitel 1.1 genauer darauf eingegangen was ein **Compiler** und **Interpreter** eigentlich sind und damit in Verbindung stehende **Begriffe** und **T-Diagramme** erklärt. Danach wird in Unterkapitel 1.2 eine kleine Einführung zu einem der Grundpfeiler des Compilerbau, den **Formalen Sprachen** gegeben. Danach werden die einzelnen **Filter** des üblicherweise bei der Implementierung von Compilern genutzten **Pipe-Filter-Architekturpatterns** (Definition 1.1) nacheinander erklärt. Die **Filter** beinhalten die **Lexikalische Analyse** 1.3, **Syntaktische Analyse** 1.4 und **Code Generierung** 1.5. Zum Schluss wird in Unterkapitel 1.6 darauf eingegangen in welchen Situationen **Fehlermeldungen** auszugeben sind.

Definition 1.1: 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. 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“.

1.1 Compiler und Interpreter

Die wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 1.3) und eines **Interpreters** (Definition 1.2), 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.

Anmerkung

Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[a] und von **Tests** die **Beziehungen** in 1.3.1 zu belegen (siehe Unterkapitel **??**), weshalb es auch nochmal wichtig ist die Definition eines **Interpreters** eingeführt zu haben.

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

Definition 1.2: Interpreter

*Programm, dass die **Anweisungen** eines Programmes mehr oder weniger **direkt** ausführt.*

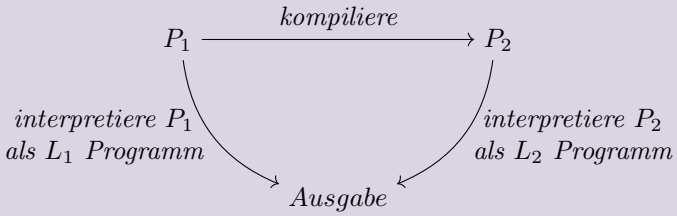
*In einer konkreten Implementierung arbeitet ein Interpreter auf einem compilerinternen **Abstrakten Syntaxbaum** (wird später eingeführt unter Definition 1.42) und führt je nach Komposition der **Knoten** des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche **Aktionen** aus.*[a]

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

Definition 1.3: Compiler

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

*Dabei muss gelten, dass die beiden Programme P_1 und P_2, wenn sie von **Interpretern** ihrer jeweiligen Sprachen L_1 und L_2 **interpretiert** werden die gleiche **Ausgabe** haben. Dies ist in Diagramm 1.3.1 dargestellt. Beide Programme P_1 und P_2 sollen die gleiche **Semantik** (Definition 1.15) haben und unterscheiden sich nur **syntaktisch** (Definition 1.14).*[a]



(1.3.1)

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

Üblicherweise kompiliert ein **Compiler** ein **Programm**, das in einer **Programmiersprache** geschrieben ist zu einem **Maschinenprogramm**, das in **Maschinensprache** (Definition 1.4) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition **??**) oder **Cross-Compiler** (Definition 1.6). Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition **??**) voneinander zu unterscheiden.

Definition 1.4: Maschinensprache

*Programmiersprache, deren mögliche Programme die **hardwarenaheste Repräsentation** von zuvor hierzu kompilierten bzw. assemblierten Programmen darstellen.*

*Jeder Maschinenbefehl entspricht einer bestimmten **Aufgabe**, welche eine CPU im **einfachen Fall** in einem **Zyklus** der **Fetch-** und **Execute-Phase**, genauergesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen, konstanten** Anzahl von Fetch- und Execute Phasen im **komplexeren Fall**.*

*Die Maschinenbefehle sind meist so entworfen, dass sie sich innerhalb bestimmter **Wortbreiten**, die **Zweierpotenzen** sind kodieren lassen. Im einfachsten Fall innerhalb einer **Speicherzelle** des **Hauptspeichers**.*[a]

*Die Programme[b] einer Maschinensprache können dabei in verschiedenen **Repräsentationen** dargestellt werden, wie z.B. in **binärer** Rerpräsentation, **hexadezimaler** Repräsentation, aber auch in **menschenlesbarer**[c] Repräsentation.*[d]

[a] Viele Prozessorarchitekturen erlauben es allerdings auch z.B. **zwei** Maschinenbefehle in **eine** Speicherzelle des Hauptspeichers zu komprimieren, wenn diese zwei Maschinenbefehle keine Operanden mit zu großen **Immediates** (Definition 1.5) haben.
[b] Bzw. **Wörter**.
[c] So wie die **Programme** des **PicoC-Compilers** dargestellt werden.
[d] Scholl, „Betriebssysteme“.

Die **Folge von Maschinenbefehlen**, die ein üblicher Compiler generiert, ist üblicherweise in **binärer Repräsentation**, da die Maschinenbefehle in erster Linie für die Maschine, die **binär** arbeitet verständlich sein sollen und nicht für den Programmierer.

Anmerkung

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings RETI-Code, der die RETI-Befehle in **menschenlesbarer Repräsentation** mit menschenlesbar ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 1.5) enthält. Für den **RETI-Interpreter** ist es ebenfalls **nicht** notwendig, dass der RETI-Code, den der PicoC-Compiler generiert, in **binärer Repräsentation** ist, denn es ist für den RETI-Interpreter ebenfalls leichter diesen einfach direkt in **menschenlesbarer Repräsentation** zu interpretieren. Der RETI-Interpreter soll nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** und **nicht** deren mögliche **interne Umsetzung**[a].

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

Definition 1.5: 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 stehen **beschränkt** ist. Der Wertebereich ist **beschränkter** als bei sonstigen Werten innerhalb des Hauptspeichers, denen eine **ganze Speicherzelle** des Hauptspeichers zur Verfügung steht.*[a]

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

Definition 1.6: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschinensprachen** L_{M_1} und L_{M_2} haben.*[a][b]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C^{Python}_{PicoC}, der in der Sprache L_{Python} geschrieben ist und die Sprache L_{PicoC} kompiliert.
[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 selbst **zeitnah** zu kompilieren oder wenn noch kein Compiler C_w für die **Wunschsprache** L und andere Programmiersprachen L_o, in denen man Programmieren wollen

würde existiert, der unter der **Maschinensprache** B_2 einer Zielmaschine M_2 läuft.[1]

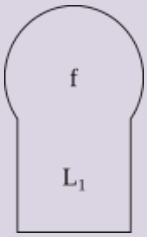
1.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**, deren Spezifikation aus der Wissenschaftlichen Publikation 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 1.7), einen Übersetzer (Definition 1.8), einen Interpreter (Definition 1.9) und eine Maschine (Definition **??**) zusammen.

Definition 1.7: 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“.

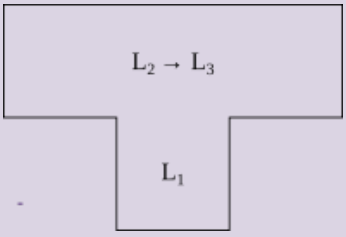
Anmerkung

Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 1.7 also reichen einfach eine 1 hinzuschreiben.

Definition 1.8: 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** die **Beziehungen** in 1.3.1.*[a]



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

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

Anmerkung

Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen **Unterschied**.

Übersetzung ist der **allgemeinere** Begriff und verlangt nur, dass **Eingabe** und **Ausgabe** des **Übersetzers** die **gleiche Bedeutung**[a] haben müssen[b].

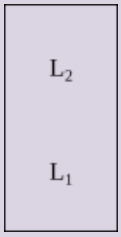
Kompilierung beinhaltet dagegen meist auch das **Lexen** und **Parsen** oder irgendeine Form von **Umwandlung** eines Programmes von der **Textrepräsentation** in eine **compilerinterne Datenstruktur** und erst dann ein oder mehrere **Übersetzungsschritte**.

Kompilieren ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

[a] Auch **Semantik** (Definition 1.15) genannt.
[b] Und ist auch zwischen **Passes** (Definition 1.43) möglich.

Definition 1.9: 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“.

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

Die **horinzontale Adjazenz** lässt sich, wie man in Abbildung 1.1 erkennen kann zusammenfassen.

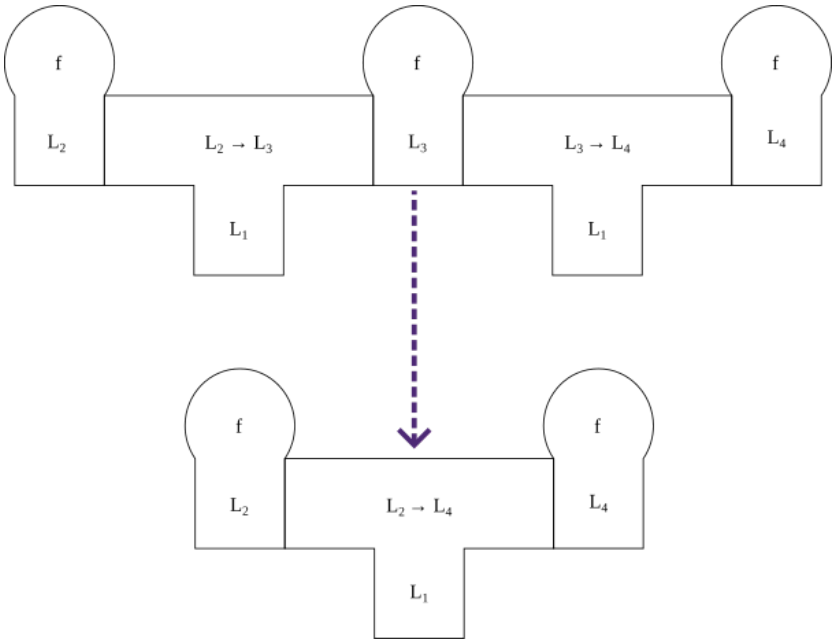


Abbildung 1.1: *Horinzontale Übersetzungszwischenschritte zusammenfassen.*

1.2 Formale Sprachen

Das **Kompilieren** eines Programmes hat viel mit dem Thema **Formaler Sprachen** (Definition 1.13) 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** (Definiton 1.10), **Alphabet** (Definition 1.11), **Wort** (Definition 1.12) beinhaltet vorher eingeführt zu haben.

Definition 1.10: Symbol

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

[a] Nebel, „Theoretische Informatik“.

Definition 1.11: Alphabet

*„Ein Alphabet ist eine **endliche**, **nicht-leere** Menge aus **Symbolen** (Definition 1.10).“*[a]

[a] Nebel, „Theoretische Informatik“.

Definition 1.12: Wort

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

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

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

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

[a] Nebel, „Theoretische Informatik".

Definition 1.13: Formale Sprache

*„Eine Formale Sprache ist eine Menge von **Wörtern** (Definition 1.12) über dem **Alphabet** Σ (Definition 1.11). "*[a]

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

[a] Nebel, „Theoretische Informatik".

Bei der Übersetzung eines Programmes von einer Sprache L_1 zur Sprache L_2 muss die **Semantik** (Definition 1.15) **gleich** bleiben. Beide Sprachen L_1 und L_2 haben eine **Grammatik** (Definition 1.16), welche diese beschreibt und können verschiedene **Syntaxen** (Definition 1.14) haben.

Definition 1.14: Syntax

*Bezeichnet alles was mit dem **Aufbau** von Wörtern einer **Formalen Sprache** zu tun hat. Eine **Formale Grammatik**, aber auch in **Natürlicher Sprache** ausgedrückte Regeln können die Syntax einer Sprache beschreiben. Es kann auch mehrere **verschiedene Syntaxen** für die **gleiche Sprache** geben*[a]*.*[b]

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

Definition 1.15: Semantik

*Die Semantik bezeichnet alles was mit der **Bedeutung** von Wörtern einer **Formalen Sprache** zu tun hat.*[a]

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

Definition 1.16: Formale Grammatik

*„Eine Formale Grammatik beschriebt wie **Wörter** einer **Sprache** abgeleitet werden können.*

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

- $S \mathrel{\hat{=}}$ ***Startsymbol***, *wobei* $S \in N$.

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

Es ist möglich ***zwei Grammatiken*** G_1 *und* G_2 *in einer* ***Vereinigungsgrammatik*** $G_1 \uplus G_2 = \langle N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S ::= S_1 \mid S_2\}, S\rangle$ *zu vereinigen.“*[efg]

[a] Weil mit ihnen **terminiert** wird.
[b] Kann auch als **Alphabet** bezeichnet werden.
[c] w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.
[d] Bzw. $w, v \in C^*$ und $w \notin \Sigma^*$.
[e] Die **Grammatik des PicoC-Compilers** lässt sich in Produktionen für die **Lexikalische Analyse** und **Syntaktische Analyse** unterteilen. Die **gesamte Grammatik** steht allerdings **vereinigt** in einer Datei.
[f] Die Produktion $S ::= S_1 \mid S_2$ kann hierbei durch **beliebige** andere Produktionen ersetzt werden, welche die beiden Grammatiken **miteinander verbinden**.
[g] Nebel, „Theoretische Informatik“.

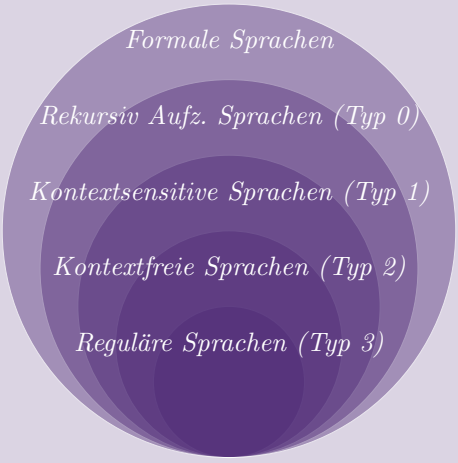
Die gerade definierten **Formale Sprachen** lassen sich des Weiteren in Klassen der **Chromsky Hierarchie** (Definition 1.17) einteilen.

Definition 1.17: Chromsky Hierarchie

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

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

Zwischen den Sprachmengen ***benachbarter Klassen*** *in Abbildung 1.17.1 besteht eine* ***echte Teilmengenbeziehung****:* $L_3 \subset L_2 \subset L_1 \subset L_0$*. Jede* ***Reguläre Sprache*** *ist auch eine* ***Kontextfreie Sprache****, aber nicht jede* ***Kontextfreie Sprache*** *ist auch eine* ***Reguläre Sprache****.*[a]



(1.17.1)

[a] Nebel, „Theoretische Informatik“.

Für diese Bachelorarbeit sind allerdings nur die **Spracheklassen** der **Chromsky-Hierarchie** relevant, die von **Regulären** (Definition 1.18) und **Kontextfreien Grammatiken** (Definition 1.19) beschrieben werden.

Definition 1.18: Reguläre Grammatik

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

$$A \to cB, \qquad A \to c, \qquad A \to \varepsilon \tag{1.18.1}$$

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

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

Definition 1.19: Kontextfreie Grammatik

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

$$A \to v \tag{1.19.1}$$

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

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

Ob sich ein Programm überhaupt kompilieren lässt entscheidet sich anhand des **Wortproblems** (Definition 1.20). In einem **Compiler** oder **Interpreter** ist das Wortproblem üblicherweise immer **entscheidbar**. Wenn das Programm ein **Wort** der **Sprache** ist, die der Compiler kompiliert, so klappt das Kompilieren, ist es **kein Wort** der **Sprache**, die der Compiler kompiliert, wird eine **Fehlermeldung** ausgegeben.

Definition 1.20: Wortproblem

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

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

$$\mathbb{1}_L : \Sigma^* \to \{0,1\} : w \mapsto \begin{cases} 1 & \text{falls } w \in L \\ 0 & \text{sonst} \end{cases} \tag{1.20.1}$$

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

1.2.1 Ableitungen

Um sicher zu wissen, ob ein Compiler ein **Programm**[2] kompilieren kann, ist es möglich das Programm mithilfe der **Grammatik** der **Sprache** des Compilers **abzuleiten**. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 1.21) und der normalen **Ableitungsrelation** (Definition 1.22) unterschieden.

[2] Bzw. **Wort**.

Definition 1.21: 1-Schritt-Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

Definition 1.22: Ableitungsrelation

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

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

[a] Nebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden, dasselbe **Programm**[3] kann z.B. über eine **Linksableitung** als auch eine **Rechtsableitung** (Definition 1.23) abgeleitet werden. Das ist später bei den verschiedenen **Ansätzen** für das **Parsen** eines Programmes in Unterkapitel 1.4 relevant.

Definition 1.23: Links- und Rechtsableitungableitung

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

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

[a] Nebel, „Theoretische Informatik“.

Manche der **Ansätze** für das **Parsen** eines Programmes haben ein Problem, wenn die Grammatik, die zur Entscheidung des **Wortproblems** für das Programm verwendet wird eine **Linksrekursive Grammatik** (Definition 1.24) ist[4].

Definition 1.24: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

[a] **noauthor˙parsing˙nodate**.

[3] Bzw. **Wort**.
[4] Für den im **PicoC-Compiler** verwendeten **Earley Parsers** stellt dies allerdings **kein** Problem dar.

Um herauszufinden, ob eine Grammatik **mehrdeutig** (Definition 1.26) ist, werden **Ableitungen** als **Formale Ableitungsbäume** (Definition 1.25) dargestellt. **Formale Ableitungsbäume** werden im Unterkapitel 1.4 nochmal relevant, da in der **Syntaktischen Analyse** Ableitungsbäume (Definition 1.35) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 1.25: Formaler Ableitungsbaum

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

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

*Den Knoten dieses Baumes sind **Grammatiksymbole** $C = N \cup \Sigma \cup \varepsilon$ (Definition 1.16) zugeordnet. Die **Inneren Knoten** des Baumes sind **Nicht-Terminalsymbole** N und die **Blätter** sind entweder **Terminalsymbole** Σ oder das **leere Wort** ε.*[b]

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

In Abbildung 1.25.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung** 1.25.1 nach den im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition **??**) angegebenen **Produktionen** 1.1 einer **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_OP	::=	”+”	
MUL_OP	::=	”∗”	
mul	::=	*mul MUL_OP NUM* | *NUM*	*L_Parse*
add	::=	*add ADD_OP mul* | *mul*	

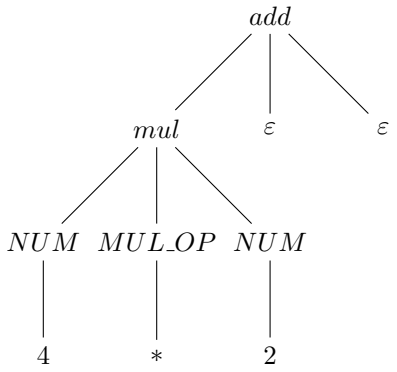
Grammatik 1.1: *Produktionen für einen Ableitungsbaum in EBNF*

Anmerkung

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

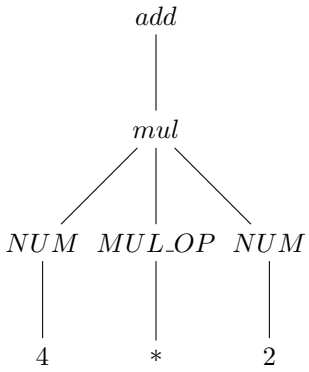
$$add \Rightarrow mul \Rightarrow mul\ \ MUL\_OP\ \ NUM \Rightarrow NUM\ \ MUL\_OP\ \ NUM \Rightarrow^* \text{”4”}\ \ \text{”}*\text{”}\ \ \text{”2”} \tag{1.25.1}$$

Bei Ableitungsbäumen gibt es **keine** einheutliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 1.25.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort** ε hinzuzufügen.



(1.25.2)

Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.25.3 nur die vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen[5].



(1.25.3)

Für einen Compiler ist es notwendig, dass die **Konkrete Grammatik** keine **Mehrdeutige Grammatik** (Definition 1.26) ist, denn sonst können unter anderem die **Präzedenzregeln** der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel **??** an einem Beispiel demonstriert wird.

Definition 1.26: Mehrdeutige Grammatik

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

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

1.2.2 Präzedenz und Assoziativität

Will man die **Operatoren** aus einer **Programmiersprache** in einer **Konkreten Grammatik** ausdrücken, die **nicht mehrdeutig** ist, so lässt sich das nach einem klaren Schema machen, wenn die **Assoziativität** (Definiton 1.27) und **Präzedenz** (Definition 1.28) dieser **Operatoren** festgelegt ist. Dieses Schema wird in Unterkapitel **??** genauer erklärt.

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

Definition 1.27: Assoziativität

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

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

[a] **noauthor˙parsing˙nodate**.

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 1.2 mithilfe von Klammern `()` veranschaulicht.

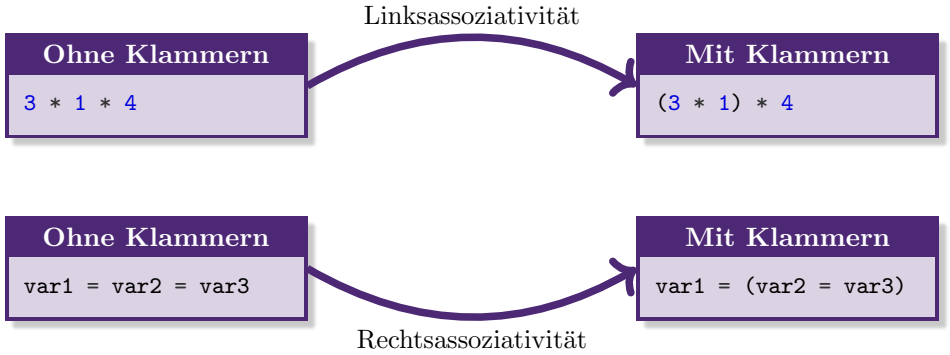


Abbildung 1.2: *Veranschaulichung von Linksassoziativität und Rechtsassoziativität.*

Definition 1.28: Präzedenz

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

[a] **noauthor˙parsing˙nodate**.

Bei **Präzedenz** ist die Mischung der Operatoren für **Subraktion** `'-'` und für **Multiplikation** `*` ein Beispiel für den Einfluss von Präzedenz. Dies ist in Abbildung 1.3 mithilfe der Klammern `()` veranschaulicht. Im Beispiel in Abbildung 1.3 ist bei den beiden **Subtraktionsoperatoren** `'-'` nacheinander und dem darauffolgenden **Multitplikationsoperator** `*` sowohl **Assoziativität** als auch **Präzedenz** im Spiel.

Abbildung 1.3: *Veranschaulichung von Präzedenz.*

1.3 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise den ersten Filter innerhalb des **Pipe-Filter Architekturpatterns** (Definition 1.1) bei der Implementierung von Compilern. Die Aufgabe der Lexikalischen Analyse

ist vereinfacht gesagt in einem Eingabewort[6] **endliche Folgen von Symbolen**[7] zu finden, die durch eine **reguläre Grammatik** erkannt werden. Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 1.29) genannt.

Definition 1.29: Lexeme

*Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches unter einer **Grammatik** G_{Lex} abgeleitet werden kann.*[a]

[a] Thiemann, „Compilerbau“.

Diese **Lexeme** werden vom **Lexer** (Definition 1.31) im **Eingabewort** identifziert und **Tokens** (Definition 1.30) zugeordnet. Das jeweils nächste **Lexeme** fängt dabei genau nach dem letzten Symbol des **Lexemes** an, das zuletzt vom **Lexer** erkannt wurde. Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 1.30: Token

*Ist ein **Tupel** (T, W) mit einem **Tokentyp** T und einem **Tokenwert** W. Ein **Tokentyp** T kann hierbei als ein **Oberbegriff** für eine möglicherweise unendliche Menge verschiedener **Tokenwerte** W verstanden werden*[a].

[a] Z.B. gibt es viele verschiedene **Tokenwerte**, wie z.B. `42`, `314` oder `12`, welche alle unter dem **Tokentyp** `NUM`, für Zahl zusammengefasst sind.

Definition 1.31: Lexer (bzw. Scanner oder auch Tokenizer)

*Ein **Lexer** ist eine **partielle** Funktion $lex : \Sigma^* \rightharpoonup (T \times W)^*$, welche ein **Lexeme** aus Σ^* auf ein **Token** (T, W) abbildet.*[a]

[a] Thiemann, „Compilerbau“.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die sich unter der **Grammatik** G_{Lex} nicht ableiten lassen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** (Definition 1.53) ausgegeben.

Anmerkung

Um Verwirrung verzubeugen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

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

In der Code Generierung sind die **Bezeichner** (Definition **??**) 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.

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

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[8] und Tabs `\t` aus dem Eingabewort 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 $\epsilon \in (T \times W)^*$ ist immer der Fall bei der **Kleeneschen Hülle** Σ^*, wobei $\Sigma = T \times W$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokentyp** T und **Tokenwert** W, ist, weil z.B. die **Bezeichner** von Variablen, Konstanten und Funktionen und auch **Zahlen** beliebige Zeichenfolgen sein können. Später in der **Syntaktischen Analyse** in Unterkapitel 1.4 wird sich nur dafür interessiert, ob an einer bestimmten Stelle ein bestimmter **Tokentyp** T, z.B. eine Zahl `NUM` steht und der **Tokenwert** W ist erst wieder in der **Code Generierung** in Unterkapitel 1.5 relevant.

Wie in Tabelle 1.1 zu sehen, gibt es für Bezeichner, wie `my_fun`, `my_var` oder `my_const` und verschiedenen Zahlen, wie `42`, `314` oder `12` passende **Tokentypen** `NAME`[9] und `NUM`[10] [11] [12]. Für **Lexeme**, wie `if` oder `}` sind die **Tokentypen** dagegen genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

Lexeme	Token
`42`, `314`	`Token('NUM', '42')`, `Token('NUM', '314')`
`my_fun`, `my_var`, `my_const`	`Token('NAME', 'my_fun')`, `Token('NAME', 'my_var')`, `Token('NAME', 'my_const')`
`if`, `}`	`Token('IF', 'if')`, `Token('RBRACE', '}')`
`99`, `'c'`	`Token('NUM', '99')`, `Token('CHAR', '99')`

Tabelle 1.1: *Beispiele für Lexeme und ihre entsprechenden Tokens.*

Ein **Lexeme** ist nicht immer das gleiche wie der **Tokenwert**, denn wie in Tabelle 1.1 zu sehen ist, kann z.B. im Fall von L_{PicoC} der Wert 99 durch zwei verschiedene **Literale** (Definition 1.32) dargestellt werden, einmal als ASCII-Zeichen `'c'`, das dann als Tokenwert den entsprechenden Wert aus der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[13]. Der **Tokenwert** ist der letztendlich verwendete **Wert** an sich, unabhängig von der Darstellungsform.

Anmerkung

Die **Konkrete Grammatik** G_{Lex}, die zur Beschreibung der Tokens T der Sprache L_{Lex} verwendet wird ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[a], sich **nichts merkt**, also unabhängig davon, **was** für Symbole und **wie oft** bestimmte Symbole **davor** aufgetaucht sind funktioniert. Auch für den PicoC-Compiler lässt sich aus der im **Dialekt der Backus-Naur-Form des Lark Parsing Toolkit** (Definition **??**) spezifizierten Grammatik **??** schlussfolgern, dass die **Sprache** des **PicoC-Compilers** für die **Lexikalische Analyse** L_{PicoC_Lex} **regulär** ist, da alle ihre **Produktionen** die Definition 1.18 erfüllen.

Produktionen mit **Alternative**, wie z.B. $DIG\_WITH\_0 ::= "0" \mid DIG\_NO\_0$ sind **unproblematisch**,

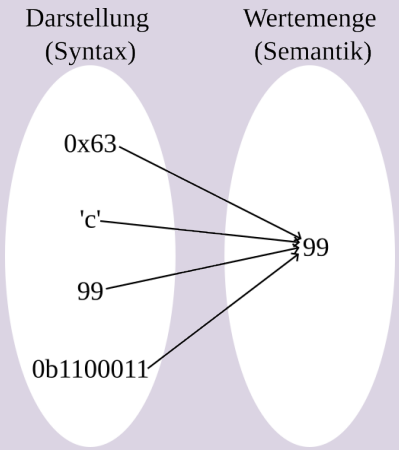
[8] In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Porgrammiersprache, die man zur Inplementierung des Lexers nutzt **wegabstrahiert**.
[9] Für z.B. `my_fun`, `my_var` und `my_const`.
[10] Für z.B. `42`, `314` und `12`.
[11] Diese **Tokentypen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Knoten haben will, damit unter anderem **mehr Code** in eine Zeile passt.
[12] Bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`.
[13] Die Programmiersprache L_{Python} erlaubt es z.B. den Wert `99` auch mit den Literalen `0b1100011` und `0x63` darzustellen.

denn sie können immer auch als $\{DIG\_WITH\_0 ::= \text{"0"},\ DIG\_WITH\_0 ::= DIG\_NO\_0\}$ ausgedrückt werden und z.B. `DIG_WITH_0*`, `(LETTER | DIG_WITH_0 | "_")+` und "␣".."∼" in Grammatik **??** können alle zu **Alternativen** umgeschrieben werden, womit diese **Alternativen** wie gerade gezeigt umgeformt werden können, um ebenfalls **regulär** zu sein. Somit existiert mit der Grammatik **??** eine **reguläre Grammatik**, welche die **Sprache** L_{PicoC_Lex} beschreibt und damit ist die **Sprache** L_{PicoC_Lex} nach der **Chromsky Hierarchie** (Definition 1.17) **regulär**.

[a] Man nennt das auch einem **Lookahead** von 1.

Definition 1.32: 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 1.4 die Lexikalische Analyse an einem Beispiel veranschaulicht.

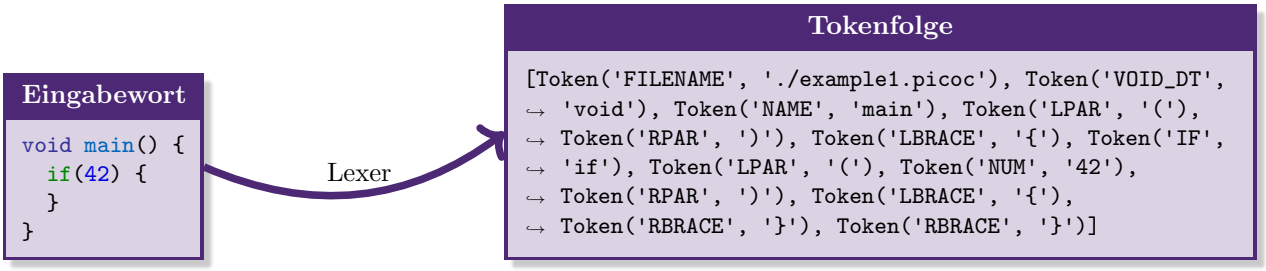


Abbildung 1.4: *Veranschaulichung der Lexikalischen Analyse.*

Anmerkung

Das Symbol ↪ zeigt im Code der Tokens in Abbildung 1.4 und in den folgenden Codes einen **Zeilenumbruch** an, wenn eine **Zeile zu lang** ist.

1.4 Syntaktische Analyse

In der **Syntaktischen Analyse** ist für einige Sprachen eine **Kontextfreie Grammatik** G_{Parse} notwendig, um diese Sprachen zu beschreiben, da viele Programmiersprachen z.B. für **Funktionsaufrufe** `fun(arg)` und **Codeblöcke** `if(1){}` syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele öffnende runde Klammern `'('` bzw. öffnende geschweifte Klammern `'{'` es momentan gibt, die noch nicht durch eine entsprechende schließende runde Klammer `')'` bzw. schließende geschweifte Klammer `'}'` geschlossen wurden. Dies lässt sich nicht mehr mit einer **Regulären Grammatik** (Definition 1.18) beschreiben, sondern es braucht eine **Kontextfreie Grammatik** (Definition 1.19) hierfür, die es erlaubt zwischen **zwei Terminalsymbolen** ein **Nicht-Terminalsymbol** abzuleiten.

Für den PicoC-Compiler lässt sich aus der Grammatik **??** schlussfolgern, dass die **Sprache** des **PicoC-Compilers** für die **Syntaktische Analyse** L_{PicoC_Parse} **kontextfrei**, aber nicht mehr **regulär** ist, da **alle** ihre **Produktionen** die Definition für **Kontextfreie Grammatiken** 1.19 erfüllen, aber **nicht** die Definition für **Reguläre Grammatiken** 1.18.

Dass die Grammatik **kontextfrei** ist lässt sich auch sehr leicht erkennen, weil **alle Produktionen** auf der **linken Seite** des `::=`-Symbols immer nur ein **Nicht-Terminalsymbol** haben und auf der **rechten Seite** eine **beliebige Folge** von **Grammatiksymbolen**[14]. Dass diese Grammatik aber nicht **regulär** sein kann, lässt sich sehr einfach an z.B. der Produktion $if\_stmt ::= "if""("logic\_or")"\ \ exec\_part$ erkennen, bei der das **Nicht-Terminalsymbol** *logic_or* von den **Terminalsymbolen** für **öffnende Klammer** { und **schließende Klammer** } eingeschlossen sein muss, was mit einer **Regulären Grammatik** nicht ausgedrückt werden kann.

Somit existiert mit der Grammatik **??** eine **Kontextfreie Grammatik** und **nicht Reguläre Grammatik**, welche die **Sprache** L_{PicoC_Parse} beschreibt und damit ist die **Sprache** L_{PicoC_Parse} nach der **Chromsky Hierarchie** (Definition 1.17) **kontextfrei**, aber **nicht regulär**.

Die **Syntax**, in welcher ein **Programm** aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 1.33) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Programm mithilfe eines **Parsers** (Definition 1.36) ein **Ableitungsbaum** (Definition 1.35) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 1.42) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 1.33: Konkrete Syntax ☑

*Steht für alles, was mit dem **Aufbau** von nach einer **Konkreten Grammatik** (Definition 1.34) **abgeleiteten Wörtern**[a] zu tun hat.*

*Die **Konkrete Syntax** ist die Teilmenge der **gesamten Syntax** einer Sprache, welche für die **Lexikalische** und **Syntaktische Analyse** relevant ist. In der **gesamten Syntax** einer Sprache[b] kann es z.B. Wörter geben, welche die gesamte Syntax **nicht einhalten**, die allerdings **korrekt** nach der **Konkreten Grammatik** abgeleitet sind[c].*

*Ein **Programm** in seiner **Textrepräsentation**, wie es in einer **Textdatei** nach der Konkreten Grammatik $G_{Lex} \uplus G_{Parse}$[d] abgeleitet steht, bevor man es kompiliert, ist in **Konkreter Syntax** aufgeschrieben.*[e]

[a] Bzw. **Programmen**.
[b] Vor allem bei **Programmiersprachen**.
[c] Wenn ein Programm z.B. **nicht deklarierte Variablen** hat und aufgrund dessen **nicht kompiliert** werden kann, hält dieses die gesamten Syntax **nicht** ein, kann allerdings so nach der **Konkreten Grammatik** abgeleitet werden.
[d] **Vereinigungsgrammatik** wie in Definition 1.16 erklärt.

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

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

Um einen kurzen Begriff für die **Grammatik** zu haben, welche die **Konkrete Syntax** einer Sprache beschreibt, wird diese im Folgenden als **Konkrete Grammatik** (Definition 1.34) bezeichnet.

Definition 1.34: Konkrete Grammatik

*Grammatik, die eine **Konkrete Syntax** einer Sprache beschreibt und die Grammatiken G_{Lex} und G_{Parse} miteinander vereinigt: $G_{Lex} \uplus G_{Parse}$[a].*

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

[a] **Vereinigungsgrammatik** wie in Definition 1.16 erklärt.
[b] Wobei das **Lark Parsing Toolkit**, welches später bei der **Implementierung** verwendet wird eine spezielle **Metasyntax** zur Spezifikation von Grammatiken nutzt, bei der für bestimmten häufig genutzte **Terminalsymbolen** ein **Tokenwert** in die Grammatik geschrieben wird.

Definition 1.35: Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)

***Compilerinterne Datenstruktur** für den **Formalen Ableitungsbaum** (Definition 1.25) eines in **Konkreter Syntax** geschriebenen Programmes.*

*Die **Blätter**, die beim Formalen Ableitungsbaum **Terminalsymbole** einer **Konkretten Grammatik** $G_{Lex} \uplus G_{Parse}$[a] sind, sind in dieser Datenstruktur **Tokens**. In dieser Datenstruktur werden allerdings nur die **Ableitungen** eines Formales Ableitungsbauemes dargestellt, die sich aus den **Produktionen** einer Grammatik G_{Parse} ergeben. Die **Tokens** sind in der Syntaktischen Analyse ein **atomarer Grundbaustein**[b], daher sind die **Ableitungen** der Grammatik G_{Lex} uninteressant.*[c]

[a] **Vereinigungsgrammatik** wie in Definition 1.16 erklärt.
[b] **Nicht** mehr weiter **teilbar**.
[c] *JSON parser - Tutorial — Lark documentation*.

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

Definition 1.36: Parser

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

[a] Z.B. wiederum ein **Programm**.
[b] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass ein Eingabewort von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die Definition 1.36 verwendet.
[c] *JSON parser - Tutorial — Lark documentation*.

Anmerkung

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

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

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

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

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

Die vom **Lexer** im Eingabewort identifizierten **Tokens** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Tokens** auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht. Dabei wird in der Grammatik L_{Parse} nach dem **Tokentypen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und nicht, welchen konkreten Wert diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 1.5 wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Erkenner** (Definition 1.37), denn ein Parser löst das **Wortproblem** (Definition 1.20) für die **Sprache**, in der das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Erkennungsalgorithmus[15] gesichert wurden den **Ableitungsbaum**.

Definition 1.37: Erkenner (bzw. engl. Recognizer)

*Entspricht einem **Kellerautomaten**[a], in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden. Der **Erkenner** ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit den Produktionen der **Konkreten Grammatik** einer Sprache ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der **Konkreten Grammatik** beschrieben wird oder nicht. Das vom **Erkenner** gelöste Problem ist auch als **Wortproblem** (Definition 1.20) bekannt.*[b]

[a] **Automat** mit dem **Kontextfreie Grammatiken** erkannt werden.
[b] Thiemann, „Compilerbau“.

Anmerkung

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

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

  Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung**, ist,

[15] Bzw. engl. recognition algorithm.

weil das **Eingabewort** von **links nach rechts** eingelesen wird, was gut damit zusammenpasst, dass die **Linksableitung** die **Blätter** von **links-nach-rechts** generiert.

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

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

Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 1.24) allerdings nicht möglich, ohne die Konkrete Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der Konkreten Grammatik entfernt zu haben, da diese zu **Unendlicher Rekursion** führt.

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

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

- **Bottom-Up Parsing:** Es wird mit dem **Eingabewort** gestartet und versucht **Rechtsableitungen** entsprechend der **Produktionen** einer **Konkreten Grammatik** rückwärts anzuwenden, bis man beim **Startsymbol** landet.[d]
- **Chart Parsing:** Es wird **Dynamische Programmierung** verwendet und **partielle 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]. **Chart Parser** können dabei **top-down** oder **bottom-up** Ansätze umsetzen. Da die **Implementierung** von **Chart Parsern** fundamental anders ist als bei **Top-Down** und **Bottom-Up Parsern**, wird diese **Kategorie** von Parsern nochmal **speziell unterschieden** und nicht gesagt, es sei ein **Top-Down Parser** oder **Bottom-Up Parser**, der **Dynamische Programmierung** verwendet.

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

Der **Abstrakte Syntaxbaum** wird mithilfe von **Transformern** (Definition 1.38) und **Visitors** (Definition 1.39) generiert und ist das Endprodukt der **Syntaktischen Analyse**, welches an die **Code Generierung** weitergegeben wird. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese ein Programm von der **Konkreten Syntax** in die **Abstrakte Syntax** (Definition 1.40).

Definition 1.38: Transformer

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

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

Definition 1.39: Visitor

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

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

Definition 1.40: Abstrakte Syntax

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

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

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

Um einen kurzen Begriff für die **Grammatik**, welche die **Abstrakte Syntax** einer **Sprache** beschreibt zu haben, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 1.41) bezeichnet.

Definition 1.41: Abstrakte Grammatik

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

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

Definition 1.42: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

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

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

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

Anmerkung

In dieser Bachelorarbeit wird häufig von der **„Abstrakten Syntax“**, der **„Abstrakten Grammatik“** bzw. dem **„Abstrakten Syntaxbaum“** einer **„Sprache“** L gesprochen. Gemeint ist hier mit der Sprache L **nicht** die Sprache, welche durch die **Abstrakte Grammatik**, nach welcher der **Abstrakte Syntaxbaum** abgeleitet ist beschrieben wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll[a] und zu deren Zweck der **Abstrakte Syntaxbaum** überhaupt konstruiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Grammatik** beschrieben wird, interessiert man sich nie wirklich explizit. Diese **Konvention** wurde aus dem Buch G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)* übernommen.

[a] Bzw. es ist die **Sprache**, welche durch die **Konkrete Grammatik** beschrieben wird.

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

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

Die **Abstrakte Grammatik** nach der ein **Abstrakter Syntaxbaum** konstruiert ist wird optimalerweise immer so definiert, dass der **Abstrakte Syntaxbaum** in den darauffolgenden Verarbeitungsschritten[16] möglichst **einfach weiterverarbeitet** werden kann.

Auf der **linken** Seite in Abbildung **??** wird das Beispiel 1.25.2 aus Unterkapitel 1.2.1 fortgeführt. Dieses Beispiel stellt den **Arithmetischen Ausdruck** `4 * 2` in Bezug auf die Konkrete Grammatik 1.2[17], welche die **höhere Präzedenz** der **Multipikation** $*$ berücksichtigt in einem **Ableitungsbaum** dar. Allerdings handelt es sich bei diesem Ableitungsbaum **nicht** um einen **Formalen Ableitungsbaum**, sondern um eine **compilerinterne Datenstruktur** für einen solchen. Dementsprechend sind die **Blätter** nun **Tokens**, die mithilfe der Grammatik L_{Lex} generiert wurden, womit die Darstellung von **Ableitungen** sich auf die Grammatik L_{Parse} beschränkt.

Auf der **rechten** Seite in Abbildung **??** wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**, der nach der Abstrakten Grammatik 1.3 konstruiert ist. Die **Abstrakte Grammatik** ist hierbei in **Abstrakter Syntaxform** (Definition **??**) angegeben. In der Abstrakten Grammatik 1.3 sind jegliche Produktionen **wegabstrahiert**, die in der **Konkreten Grammatik** 1.2 so umgesetzt sind, damit diese **Präzidenz** beachtet und nicht **mehrdeutig** ist. Aus diesem Grund gibt es nur noch einen **allgemeinen**

[16] Den verschiedenen **Passes**.
[17] Die **Konkrette Grammatik** ist hierbei im **Dialekt der Erweiterter Backus-Naur-Form des Lark Parsing Toolkit** (Definition **??**) angegeben.

Knoten für **binäre Operationen** $BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle)$.

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_OP	::=	"+"	
MUL_OP	::=	"*"	
mul	::=	*mul* MUL_OP NUM | NUM	*L_Parse*
add	::=	*add* ADD_OP *mul* | *mul*	

Grammatik 1.2: *Produktionen für Ableitungsbaum in EBNF*

bin_op	::=	$Add()$ | $Mul()$
exp	::=	$BinOp(\langle exp \rangle, \langle bin\_op \rangle, \langle exp \rangle)$ | $Num(\langle str \rangle)$

Grammatik 1.3: *Produktionen für Abstrakten Syntaxbaum in ASF*

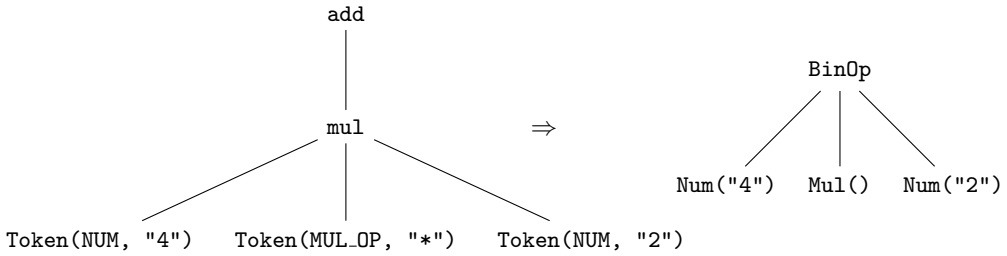


Abbildung 1.5: *Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.*

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

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, sind in Abbildung 1.6 die einzelnen **Zwischenschritte** von den Tokens der **Lexikalischen Analyse** zum **Abstrakten Syntaxbaum** anhand des fortgeführten Beispiels aus Unterkapitel 1.3 veranschaulicht. In Abbildung 1.6 werden die Darstellungen des **Ableitungsbaumes** und des **Abstrakten Syntaxbaumes** verwendet, wie sie vom **PicoC-Compiler** ausgegeben werden. In der Darstellung des **PicoC-Compilers** stellen die verschiedenen **Einrückungen** die verschiedenen **Ebenen** dieser Bäume dar. Die Bäume **wachsen** von der **Wurzel** von **links-nach-rechts** zu den **Blättern**.

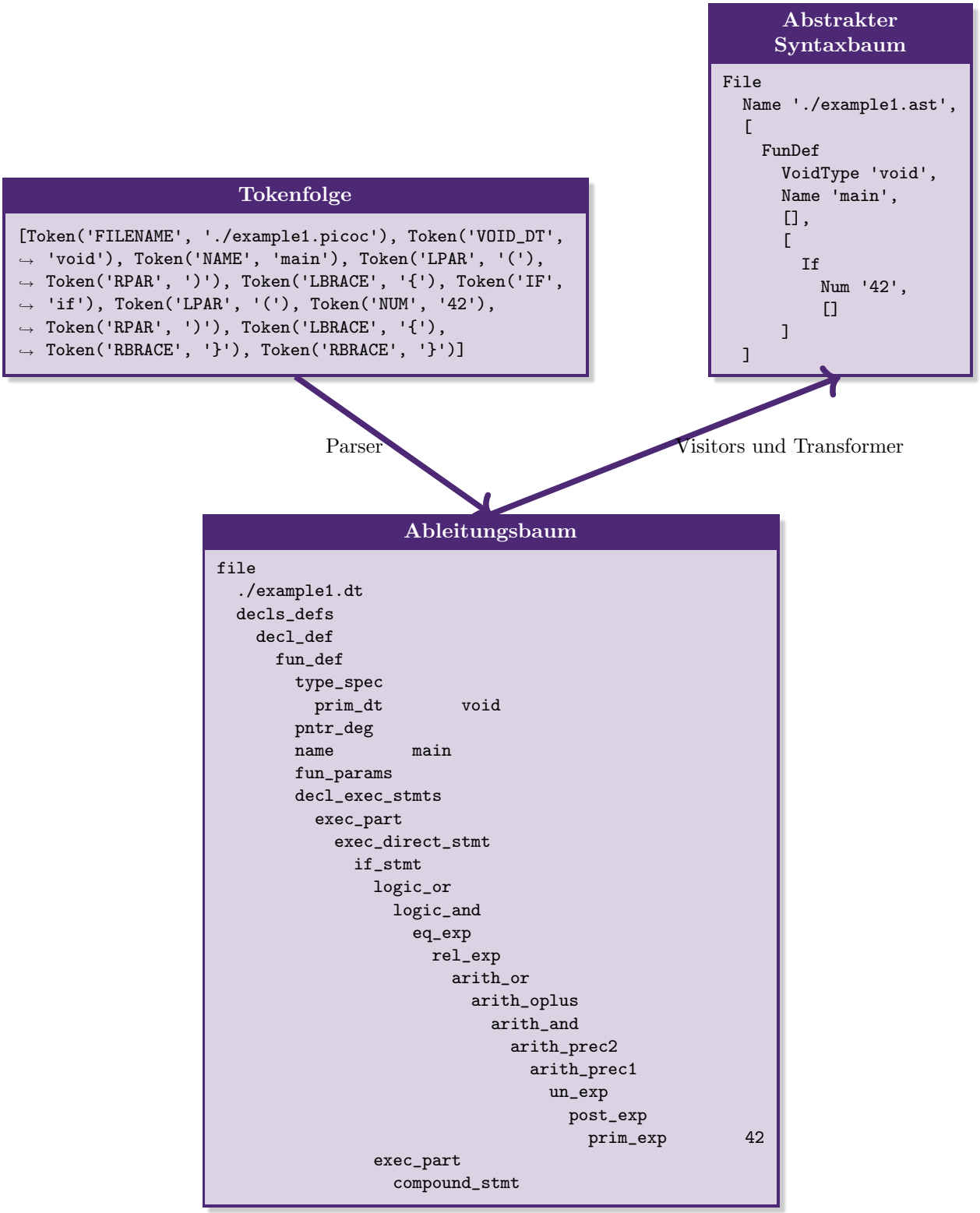


Abbildung 1.6: *Veranschaulichung der Syntaktischen Analyse.*

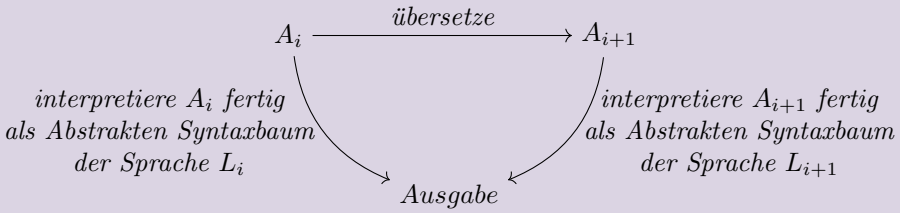
1.5 Code Generierung

In der **Code Generierung** steht man nun dem Problem gegenüber einen **Abstrakten Syntaxbaum** einer Sprache L_1 in den **Abstrakten Syntaxbaum** einer Sprache L_2 umformen zu müssen. Dieses Problem lässt sich vereinfachen, indem man das Problem in mehrere Schritte unterteilt, die man **Passes** (Definition 1.43) nennt. So wie es auch schon mit dem **Ableitungsbaum** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakten Syntaxbaum** kontstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

Definition 1.43: Pass

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

Für ***jeden Pass*** *und für einen* ***beliebigen*** *Abstrakten Syntaxbaum* A_i *gilt ähnlich, wie bei einem* ***vollständigen Compiler*** *in 1.43.1, dass:*



(1.43.1)

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

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

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

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

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

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

1.5.1 Monadische Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 1.44) und **Anweisung** (Definition 1.45) wichtig.

Definition 1.44: Ausdruck (bzw. engl. Expression)

Code, der eine ***semantische Bedeutung*** *hat und in einem bestimmten* ***Kontext ausgewertet*** *werden kann, um einen* ***Wert*** *zu liefern oder etwas zu* ***deklarieren.***

*Aufgebaut sind **Ausdrücke** meist aus **Kostanten**, **Variablen**, **Funktionsaufrufen**, **Operatoren** usw.*[a][b]

[a] Ein **Ausdruck** ist z.B `21 * 2;`.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.45: Anweisung (bzw. engl. Statement)

*Code, der eine **Vorschrifft** darstellt, die ausgeführt werden soll und **als ganzes keinen Wert** liefert und **nichts deklariert**. Eine Anweisung kann sich jedoch aus ein oder mehreren **Ausdrücken** zusammensetzen, die dies tun.*

*Anweisungen sind zentrale Elemente **Imperativer Programmiersprachen**, die sich zu einem großen Teil aus **Folgen von Anweisungen** zusammensetzen.*

*In **Maschinensprachen** werden **Anweisungen** häufig als **Befehle** bezeichnet.*[a][b]

[a] Eine **Anweisung** ist z.B `int var = 21 * 2;`.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Hat man es mit einer Programmiersprache zu tun, deren Programme **Unreine Anweisungen** (Definition 1.47) besitzen, so ist es sinnvoll einen **Pass** einzuführen, der **Unreine Ausdrücke** von den Anweisungen **trennt**, damit diese zu **Reinen Anweisungen** (Definition 1.46) werden. Das wird erreicht, indem man aus jedem Unreinen Ausdruck einen **vorangestellten Ausdruck** macht, den man **vor** die jeweilige Anweisung setzt, mit welcher der Unreine Ausdruck **gemischt** war. Der **Unreine Ausdruck** muss als **erstes** ausgeführt werden, für den Fall, dass der **Effekt**, den ein **Unreiner Ausdruck** hat die **Reine Anweisung**, mit der er gemischt war in irgendeinerweise beeinflussen könnte.

Definition 1.46: Reiner Ausdruck / Reine Anweisung (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 **Maschinencode** darstellen lässt. Sondern z.B. **intern** etwas am weiteren **Kompiliervorgang** ändert*[a].

*Eine **Reine Anweisung** ist eine Anweisung, bei der **alle Ausdrücke** aus denen sich die Anweisung unter anderem zusammensetzt **rein** sind.*[b]

[a] Z.B. ist die **Allokation von Variablen** `int var` **kein Reiner Ausdruck**. Eine Allokation bestimmt den Wert einiger **Immediates** im finalen **Maschinencode**, aber entspricht **keiner Folge** von **Maschinenbefehlen**.
[b] G. Siek, *Course Webpage for Compilers (P423, P523, E313, and E513)*.

Definition 1.47: Unreiner Ausdruck / Unreine Anweisung

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

*Eine **Unreine Anweisung** ist eine Anweisung, bei der **mindestens** einer der **Ausdrücke** aus denen sich die Anweisung unter anderem zusammensetzt **unrein** ist.*[a]

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

Auf diese Weise sind alle **Anweisungen** in **Monadischer Normalform** (Definiton 1.48).

Definition 1.48: Monadische Normalform (bzw. engl. monadic normal form)

*Code ist in **Monadischer Normalform**, wenn dieser nach einer Grammatik in **Monadischer Normalform** abgeleitet wurde.*

*Eine **Konkrete Grammatik** ist in **Monadischer Normalform**, wenn **alle** ableitbaren Anweisungen **rein** sind. Oder sehr **allgemein** ausgedrückt, wenn **Reines** und **Unreines** klar voneinander getrennt ist.*[a]

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

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

Ein **Beispiel** für das Vorgehen, Code in die **Monadische Normalform** zu bringen, ist in Abbildung 1.7 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.7 sind daher in **Konkreter Syntax**[18] aufgeschrieben.

Links in der Abbildung 1.7 ist der Ausdruck mit dem **Nebeneffekt**, eine Variable zu **definieren**: `int var`, mit dem Ausdruck für eine **Zuweisung** `exp = 5 % 4` gemischt: `int var = 5 % 4`. Der **Unreine** Definitionsausdruck `int var` muss daher **vorangestellt** werden, wie es **rechts** in Abbildung 1.7 dargestellt ist[19].

Abbildung 1.7: *Codebeispiel dafür Code in die Monadische Normalform zu bringen.*

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** anzunähren, indem Subbäume vorangestellt werden, die keine Entsprechung in **Maschinenbefehlen** haben. Somit wird eine **Seperation** von Subbäumen, die keine Entsprechung in **Maschinenbefehlen** haben und denen, die eine haben bewerkstelligt wird. Eine **Reine Anweisung** ist **Maschinenbefehlen** ähnlicher als eine **Unreine Anweisung**. Somit sparrt man sich in der Implementierung **Fallunterscheidungen**, indem **Reine Ausdrücke** und **Reine Anweisungen** direkt in **Maschinenbefehle** übersetzt werden können und **nicht** unterschieden werden muss, ob darin **Unreine Ausdrücke** vorkommen.

1.5.2 A-Normalform

Zum Verständnis dieses Kapitels sind die Begriffe **Ausdruck** (Definition 1.44) und **Anweisung** (Definition 1.45) wichtig.

Eine Programmiersprache L_1 soll in eine Maschinensprache L_2 kompiliert werden. Im Falle dessen, dass es sich bei einer **Sprache** L_1 um eine **höhere Programmiersprache** und bei L_2 um eine **Maschinensprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 1.51) in

[18] Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.
[19] Obwohl hinter `int var` ein `;` steht, ist es immer noch ein **Ausdruck**. Allerdings gibt es **keine** einheitliche Festlegung, was eine **Anweisung** ist und was **nicht**, es wurde für diese Schrifftliche Ausarbeitung der Bachelorarbeit nur so definiert.

Anweisungen und **Ausdrücken** verhindert. Das wird erreicht, indem man aus den Komplexen Ausdrücken **vorangestellte** Ausdrücke macht, in denen die **Komplexen Ausdrücke temporären Locations** (Definiton 1.49) zugewiesen werden und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Definition 1.49: Location

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

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

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

Sollte der **Komplexe Ausdruck**, welcher einer **temporären Location** zugewiesen wird, **Teilausdrücke** enthalten, die **komplex** sind, muss das gleiche Vorgehen erneut für die **Teilausdrücke** angewandt werden, bis alle **Komplexen Ausdrücke** nur noch **Atomare Ausdrücke** (Definiton 1.50) enthalten, falls sie sich überhaupt in **weitere Teilausdrücke** aufteilen lassen.

Sollte es sich bei dem **Komplexen Ausdruck** um einen **Unreinen Ausdruck** handeln, welcher nur einen **Nebeneffekt** ausführt und sich nicht in **Maschinenbefehle** übersetzen lässt, so wird aus diesem ein **vorangestellter Ausdruck** gemacht, welcher einfach nur den **Nebeneffekt** dieses **Unreinen Ausdrucks** ausführt und keiner **temporären Location** zugewiesen wird.

Definition 1.50: Atomarer Ausdruck

*Ein **Atomarer Ausdruck** ist ein **Reiner Ausdruck** (Definition 1.46), der **keinem kompletten Maschinenbefehl** entspricht, sondern nur ein **Argument**, wie z.B. einen **Immediate** in einer Folge von Maschinenbefehlen festlegt.*[a]

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

Bei einem **üblichen Compiler**, bei dem z.B. **Register** für **temporäre Zwischenergebnisse** genutzt werden und der **Maschinenbefehlssatz** es erlaubt **zwei Register** miteinander zu verechnen[20] sind **Atomare Ausdrücke** z.B. eine **Variable** (z.B. `var`), eine **Zahl** (z.B. `12`) oder ein **ASCII-Zeichen** (z.B. `'c'`), da diese häufig direkt über **Register** zugreifbar sind, die direkt mit einem **Maschinenbefehl** verechnet werden können[21] [22].

Im Fall des **PicoC-Compilers** ist ein **Zugriff auf eine Location** (z.B. `stack(i)`) der einzige **Atomare Ausdruck**, da der **PicoC-Compiler** so umgesetzt ist, dass er alle **Zwischenergebnisse** auf dem **Stack** speichert und dort dann auf diese zugreift, um sie in **Register** zu laden und miteinander zu verechnen[23]. Aus diesem Grund braucht es mindestens einen Maschinenbefehl[24], um z.B. eine Zahl überhaupt für einen **Maschinenbefehl** zugreifbar zu machen, was der Definition 1.50 widerspricht. Daher sind z.B. Zahlen beim **PicoC-Compiler** keine **Atomaren Ausdrücke**.

[20] Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.
[21] Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.
[22] Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse auf den **Stack** zu speichern und ein **Stackpointer Register** anzupassen.
[23] Der **PicoC-Compiler** nutzt, anders als es geläufig ist keine **Register** und **Graph Coloring** (Definition **??**) inklusive **Liveness Analysis** (Definition **??**) usw., um Werte von Variablen, temporäre Zwischenergebnisse usw. abzuspeichern, sondern immer nur den **Hauptspeicher**, wobei **temporäre Zwischenergebnisse** auf den **Stack** gespeichert werden. Beim **PicoC-Compiler** sollte sich an die **Paradigmen** aus der Vorlesung Scholl, „Betriebssysteme“ gehalten werden.
[24] Genauergesagt 4.

Definition 1.51: Komplexer Ausdruck

*Ein **Komplexer Ausdruck** ist ein **Ausdruck**, der **nicht atomar** ist.*[a]

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

Im Fall des **PicoC-Compilers** sind **Komplexe Ausdrücke** z.B. `5 % 4`, `-1`, `fun(12)` oder `int var`, da diese zur Berechnung auf jeden Fall mehrere **Maschinenbefehle** benötigen, was Definition 1.51 widerspricht oder **unrein** sind. Die Teilausdrücke `4`, `5`, `1` müssen erst auf den **Stack** geschrieben werden, um dann in **Register** geladen zu werden, damit dann der gesamte **Komplexe Ausdruck** berechnet werden kann. Die Ausdrücke `fun(12)` und `int var` sind **unrein** und daher **Komplexe Ausdrücke**.

In Abbildung 1.8 ist zur besseren Vorstellung die Einteilung von **Komplexen**, **Atomaren**, **Unreinen** und **Reinen** Ausdrücken veranschaulicht. Des Weiteren sind in der Abbildung alle Ausdrücke ausgegraut, welche die **Monadische Normalform** nicht erfüllen. Hierbei wird vom **PicoC-Compiler** ausgegangen, bei dem nur ein **Zugriff auf eine Location** (z.B. `stack(i)`) einen **Atomaren Ausdruck** darstellt.

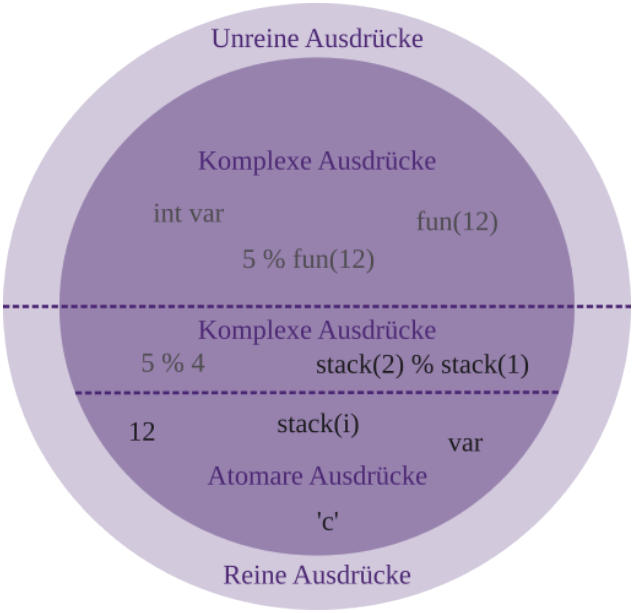


Abbildung 1.8: *Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke.*

Es wird in dem gerade beschriebenen **Pass** dafür gesorgt, dass alle **Anweisungen** und **Ausdrücke** in **A-Normalform**[25] (Definition 1.52) sind. Wenn eine **Konkrete Grammatik** in **A-Normalform** ist, ist diese per Defintition 1.52 auch automatisch in **Monadischer Normalform**, genauso, wie ein **Atomarer Ausdruck** nach Definition 1.50 auch ein **Reiner Ausdruck** ist.

Definition 1.52: A-Normalform (ANF)

*Code ist in **A-Normalform**, wenn dieser nach einer **Konkreten Grammatik** in **A-Normalform** abgeleitet ist.*

*Eine **Konkrete Grammatik** ist in **A-Normalform**, wenn sie in **Monadischer Normalform** (Definition 1.48) ist und wenn alle **Komplexen Ausdrücke** nur **Atomare Ausdrücke** enthalten,*

[25] Das **'A'** kommt vermutlich von **„atomar“** bzw. engl. „atomic“, weil alle **Komplexen Ausdrücke** nur noch **Atomare Ausdrücke** enthalten dürfen.

*falls sie sich überhaupt in **weitere Teilausdrücke** einteilen lassen.*

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

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

Ein **Beispiel** für dieses Vorgehen, Code in die **A-Normalform** zu bringen, ist in Abbildung 1.9 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser Schrifftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.7 sind daher in **Konkreten Syntax**[26] aufgeschrieben.

Um **konsistent** mit der Implementierung zu sein und später keine Verwirrung zu erzeugen, wird beim Beispiel in Abbildung 1.9 vom **PicoC-Compiler** ausgegangen, bei dem **Variablen** (z.B. `var`), **Zahlen** (z.B. `12`) oder **ASCII-Zeichen** (z.B. `'c'`) **Komplexe Ausdrücke** darstellen.

Die Ausdrücke `4;`, `x;`, usw. für sich sind in diesem Fall **Komplexe Ausdrücke**, deren Wert einer **Location**, in diesem Fall einer **Speicherzelle des Stack** zugewiesen werden. Auf das Ergebnis dieser **Komplexen Ausdrücke** wird mittels `stack(2)` und `stack(1)` zugegriffen, um diese in **Register** zu schreiben und dann z.B. im **Komplexen Ausdruck** `stack(2) % stack(1)` miteinander zu verrechnen und das Ergebnis wiederum einer **Location**, in diesem Fall ebenfalls einer **Speicherzelle des Stack** zuzuweisen. Dieses Ergebnis wird dann vom **Stack** `stack(1)` in die **Globalen Statischen Daten** `global(0)` gespeichert, wo die Variable `x` allokiert ist. Die Zahlen `0`, `1`, `2` sind dabei hierbei **relative Adressen** auf dem **Stack** und in den **Globalen Statischen Daten**.

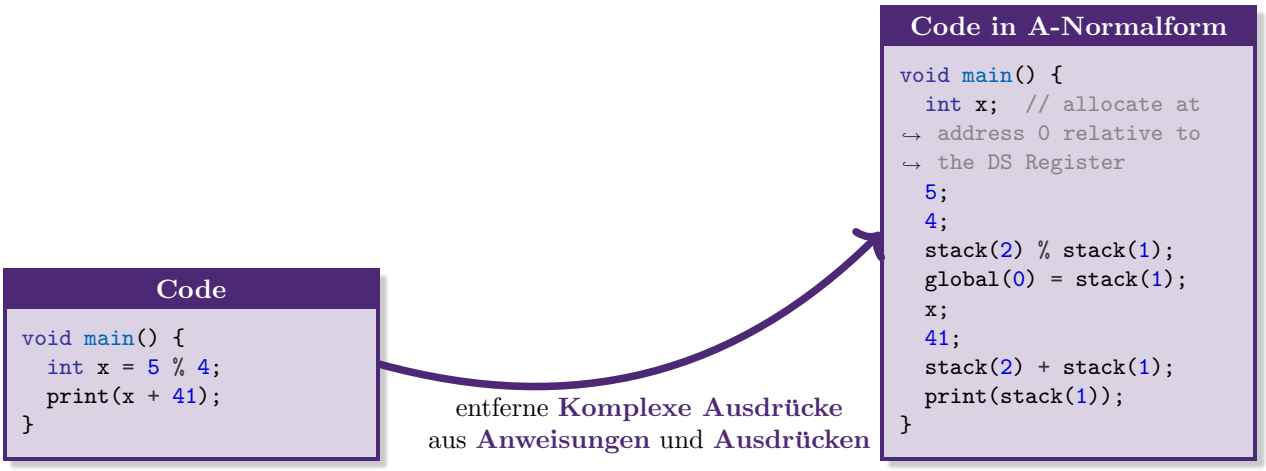


Abbildung 1.9: *Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.*

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

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

Alle weiteren denkbaren **Passes** sind zu **spezifisch** auf bestimmte **Anweisungen** 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 **??** erklärt.

1.5.3 Ausgabe des Maschinencodes

Nachdem alle **Passes** durchgearbeitet wurden, ist es notwendig aus dem finalen **Abstrakten Syntaxbaum** den eigentlichen **Maschinencode** in **Konkreter Syntax** zu generieren. In üblichen Compilern wird hier für den **Maschinencode** eine **binäre Repräsentation** gewählt[27]. Der Weg von der **Abstrakten Syntax** zur **Konkreten Syntax** ist allerdings wesentlich einfacher, als der Weg von der **Konkreten Syntax** zur **Abstrakten Syntax**, für die eine gesamte **Syntaktische Analyse**, die eine **Lexikalische Analyse** beinhaltet durchlaufen werden muss.

Jeder **Knoten** des **Abstrakten Syntaxbaumes** erhält dazu eine Methode, welche hier `to_string` genannt wird, die eine **Textrepräsentation** seiner selbst und all seiner Knoten, mit an den richtigen Stellen passend gesetzten **Semikolons** `;`, **öffnenden-** und **schließenden runden Klammern** `()`, **öffnenden-** und **schließenden geschweiften Klammern** {} usw. ausgibt. Dabei wird der gesamte **Abstrakte Syntaxbaum** durchlaufen und die Methode `to_string` zur Ausgabe der **Textrepräsentation** der verschiedenen Knoten aufgerufen, die wiederum die Methode `to_string` ihrer Kinder aufrufen und die zurückgegebene Textrepräsentation passend **zusammenfügen** und selbst **zurückgebeben**.

1.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**[28] eintritt:

1. In der **Lexikalischen** oder **Syntaktischen Analyse** tritt ein Fall ein, der **nicht** in der **Syntax** der Sprache des Compilers abgedeckt ist, z.B.:

   - Der **Lexer** kann ein Lexeme **nicht** mit der Konkreten Grammatik für die Lexikalische Analyse G_{Lex} **ableiten**. Der **Lexer** ist genaugenommen ein **Teil des Parsers** und ist damit bereits durch den nachfolgenden Punkt „Parser“ abgedeckt. Um die unterschiedlichen Ebenen, **Lexikalische** und **Syntaktische Analyse** gesondert zu betrachten wurde der Lexer an dieser Stelle ebenfalls kurz eingebracht.

   - Der **Parser**[29] entscheidet das **Wortproblem** (Definition 1.20) für ein **Eingabeprogramm**[30] mit `0`, also das **Eingabeprogramm** lässt sich **nicht** durch die Konkrete Grammatik $G_{Lex} \uplus G_{Parse}$ des Compilers **ableiten**.

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

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

   - Bei einem **Funktionsaufruf** werden **mehr** oder **weniger** Argumente, Argumente des **falschen Datentyps** oder Argumente in der **falschen Reihenfolge** übergeben, als sie im **Funktionsprototyp** angegeben sind.

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

[27] Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben
[28] *Errors in C/C++ - GeeksforGeeks*.
[29] Bzw. der **Erkenner** innerhalb des Parsers.
[30] Bzw. ein **Wort**.

- Eine **nicht erlaubte Operation**, wie **Division durch** 0 (z.B. `42 / 0`) soll ausgeführt werden.
- **Segmentation Fault:** Wenn auf **Speicher zugegriffen** wird, der vom **Betriebssystem geschützt** ist.

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

- Es gibt **keine** oder **mehr als eine** `main`-Funktion.
- Eine Funktion, die in einer **Objektdatei** (Definition **??**) benötigt wird, wird von **keiner** oder **mehr als einer** anderen Objektdatei bereitsgestellt.

wird eine **Fehlermeldung** (Definition 1.53) ausgegeben.

Definition 1.53: Fehlermeldung

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

- *über* `stdout` *oder* `stderr` *im einem* ***Terminal Emulator*** *oder* ***richtigen Terminal****.*
- *über eine* ***Dialogbox*** *in einer* ***Graphischen Benutzeroberfläche***[b] *oder* ***Zeichenorientierten Benutzerschnittstelle***[c]*.*
- *über ein* ***Register*** *oder eine* ***spezielle Adresse*** *des* ***Hauptspeichers*** *mithilfe eines* ***Wertes****.*
- *über eine* ***Logdatei***[d] *auf einem* ***Speichermedium****.*

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

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).
- *Transformers & Visitors — Lark documentation*. URL: https://lark-parser.readthedocs.io/en/latest/visitors.html (besucht am 09. 07. 2022).
- *What is Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 2022).
- *What is Top-Down Parsing?* URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22. 06. 2022).

Bücher

- G. Siek, Jeremy. *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).

Artikel

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

Vorlesungen

- Nebel, 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).

- 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).