

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

Betreuung:

M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Danksagungen

Bevor der Inhalt dieser Schriftlichen 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 Selbstverständlichkeit 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 hinaus, 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 Leidenschaft heraus tun, auch wenn es für sie keine Vorteile hat. Tobias¹ 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 habe², er hat sich bei der Korrektur dieser Schriftlichen 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 Selbstverständlichkeit, 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 dageben wohl mit der Zeit äußerst kritisch geworden. Das Prof. Dr. Scholl mir zu

¹Wie auch Prof. Dr. Scholl. Hier geht es aber erstmal um Tobias.

²Wofür 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 Selbstverständlichkeit, 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 auseinandergesetzt 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 persönliches Interessengebiet fällt⁶. Das Aufschreiben dieser Schriftlichen Ausarbeitung hat mir dagegen eher weniger Spaß gemacht⁷. 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 Programmiersprache kann in der Verwendung ziemlich willkürlich 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.

³Der PicoC-Compiler hätte schließlich mehr Funktionalitäten haben können.

⁴Vielleicht 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>.

⁶Womit nicht alle Studenten so viel Glück haben.

⁷Dieses ständige überlegen, wo man möglicherweise eine Erklärücke hat, ob man nicht was wichtiges ausgelassen hat usw.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Codeverzeichnis	II
Tabellenverzeichnis	III
Definitionsverzeichnis	V
Grammatikverzeichnis	VI
1 Theoretische Grundlagen	1
1.1 Compiler und Interpreter	1
1.1.1 T-Diagramme	4
1.2 Formale Sprachen	6
1.2.1 Ableitungen	9
1.2.2 Präzedenz und Assoziativität	12
1.3 Lexikalische Analyse	13
1.4 Syntaktische Analyse	16
1.5 Code Generierung	26
1.5.1 Monadische Normalform	26
1.5.2 A-Normalform	28
1.5.3 Ausgabe des Maschinencodes	32
1.6 Fehlermeldungen	32
2 Ergebnisse und Ausblick	34
2.1 Funktionsumfang	34
2.1.1 Kommandozeilenoptionen	34
2.1.2 Shell-Mode	37
2.1.3 Show-Mode	39
2.2 Qualitätssicherung	41
2.3 Erweiterungsideen	45
Literatur	A

Abbildungsverzeichnis

1.1	Horizontale Übersetzungszwischenschritte zusammenfassen.	6
1.2	Veranschaulichung von Linksassoziativität und Rechtsassoziativität.	13
1.3	Veranschaulichung von Präzedenz.	13
1.4	Veranschaulichung der Lexikalischen Analyse.	16
1.5	Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.	23
1.6	Veranschaulichung der Darstellung eines Baumes beim PicoC-Compilers.	24
1.7	Veranschaulichung der Syntaktischen Analyse.	25
1.8	Codebeispiel dafür Code in die Monadische Normalform zu bringen.	28
1.9	Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke.	30
1.10	Codebeispiel für das Entfernen Komplexer Ausdrücke aus Operationen.	31
2.1	Show-Mode in der Verwendung.	41

Codeverzeichnis

2.1	Shellaufruf und die Befehle <code>compile</code> und <code>quit</code>	38
2.2	Shell-Mode und der Befehl <code>most_used</code>	39
2.3	Typischer Test.	43
2.4	Testdurchlauf.	45
2.5	Beispiel für Tail Call.	48

Tabellenverzeichnis

1.1	Beispiele für Lexeme und ihre entsprechenden Tokens.	15
2.1	Kommandozeilenoptionen, Teil 1.	36
2.2	Kommandozeilenoptionen, Teil 2.	37
2.3	Makefileoptionen.	40
2.4	Testkategorien.	42

Definitionsverzeichnis

1.1	Pipe-Filter Architekturpattern	1
1.2	Interpreter	2
1.3	Compiler	2
1.4	Maschinensprache	2
1.5	Immediate	3
1.6	Cross-Compiler	3
1.7	T-Diagramm Programm	4
1.8	T-Diagramm Übersetzer (bzw. eng. Translator)	4
1.9	T-Diagramm Interpreter	5
1.10	Symbol	6
1.11	Alphabet	6
1.12	Wort	6
1.13	Formale Sprache	7
1.14	Syntax	7
1.15	Semantik	7
1.16	Formale Grammatik	7
1.17	Chomsky Hierarchie	8
1.18	Reguläre Grammatik	9
1.19	Kontextfreie Grammatik	9
1.20	Wortproblem	9
1.21	1-Schritt-Ableitungsrelation	9
1.22	Ableitungsrelation	10
1.23	Links- und Rechtsableitungableitung	10
1.24	Formaler Ableitungsbaum	10
1.25	Mehrdeutige Grammatik	12
1.26	Assoziativität	12
1.27	Präzedenz	13
1.28	Lexeme	13
1.29	Token	14
1.30	Lexer (bzw. Scanner oder auch Tokenizer)	14
1.31	Literal	15
1.32	Konkrete Syntax	17
1.33	Konkrete Grammatik	18
1.34	Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)	18
1.35	Parser	18
1.36	Erkenner (bzw. engl. Recognizer)	19
1.37	Transformer	20
1.38	Visitor	21
1.39	Abstrakte Syntax	21
1.40	Abstrakte Grammatik	21
1.41	Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)	21
1.42	Pass	26
1.43	Ausdruck (bzw. engl. Expression)	26
1.44	Anweisung (bzw. engl. Statement)	27
1.45	Reiner Ausdruck / Reine Anweisung (bzw. engl. pure expression)	27
1.46	Unreiner Ausdruck / Unreine Anweisung	27
1.47	Monadische Normalform (bzw. engl. monadic normal form)	28

1.48	Location	29
1.49	Atomarer Ausdruck	29
1.50	Komplexer Ausdruck	30
1.51	A-Normalform (ANF)	30
1.52	Fehlermeldung	33

Grammatikverzeichnis

1.1	Grammatik für einen Ableitungsbaum in EBNF	11
1.2	Produktionen für Ableitungsbaum in EBNF	23
1.3	Produktionen für Abstrakten Syntaxbaum in ASF	23

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

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

Eine **Pipe** stellt ein **Bindeglied** zwischen zwei **Filtern** dar.^{a,b}



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

^bWestphal, „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 2.2), weshalb es auch nochmal wichtig ist die Definition eines **Interpreters** eingeführt zu haben.

^aSammlung von Compilern für Linux bzw. GNU-Linux, steht für **GNU Compiler Collection**

Definition 1.2: Interpreter

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

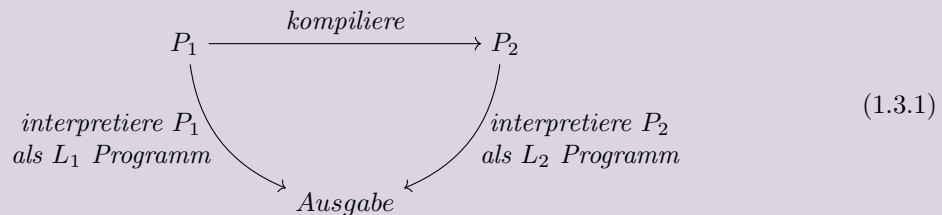
In einer konkreten Implementierung arbeitet ein Interpreter auf einem compilerinternen **Abstrakten Syntaxbaum** (wird später eingeführt unter Definition 1.41) und führt je nach Komposition der **Knoten** des Abstrakten Syntaxbaumes, auf die er während des Darüber-Iterierens stösst unterschiedliche **Aufgaben** aus.^a

^aG. Siek, *Essentials of Compilation*.

Definition 1.3: Compiler

Übersetzt ein **beliebiges** Programm 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



^aG. Siek, *Essentials of Compilation*.

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 ?? im ??) oder **Cross-Compiler** (Definition 1.6)¹.

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**, genauer gesagt in der **Execute-Phase** übernehmen kann oder allgemein in einer **geringen, konstanten** Anzahl von Fetch- und Execute Phasen im **komplexeren Fall**.

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

¹Des Weiteren sind **Maschinensprache** und **Assemblersprache** (Definition ?? im ??) voneinander zu unterscheiden.

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

^aViele 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.

^bBzw. **Wörter**.

^cSo wie die **Programme** des **PicoC-Compilers** dargestellt werden.

^dC. 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 ausgeschriebenem 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.

^aEine **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 **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

^aLjohhuh, *What is an immediate value?*

Definition 1.6: Cross-Compiler

Kompiliert auf einer **Maschine** M_1 ein Programm, dass in einer **Wunschsprache** L_w 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}

^aBeim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C_{PicoC}^{Python} , der in der Sprache L_{Python} geschrieben ist und die Sprache L_{PicoC} kompiliert.

^bEarley und Sturgis, „A formalism for translator interactions“.

Ein **Cross-Compiler** ist entweder notwendig, wenn eine Zielmaschine M_2 nicht ausreichend **Rechenleistung** besitzt, um ein Programm in der Wunschsprache L_w selbst **zeitnah** zu kompilieren oder wenn noch **keine** Compiler C_w oder C_o für die **Wunschsprache** L_w oder eine **andere** Programmiersprache L_o , in welcher der

Wunschcompiler C_w implementiert ist existieren, die unter der **Zielmaschine** M_2 laufen.²

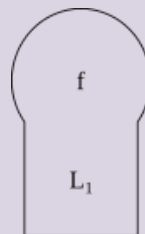
1.1.1 T-Diagramme

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 und Interpretern in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Programm (Definition 1.7), einen Übersetzer (Definition 1.8), einen Interpreter (Definition 1.9) und eine Maschine (Definition ??) zusammen.

Definition 1.7: T-Diagramm Programm

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



^aEarley und Sturgis, „A formalism for translator interactions“.

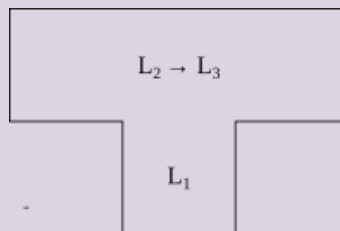
Anmerkung

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-Diagramm Übersetzer (bzw. eng. Translator)

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

Für einen **Übersetzer** gelten genauso, wie für einen **Compiler** die Relationen in Abbildung 1.3.1.^a



^aEarley und Sturgis, „A formalism for translator interactions“.

²Die an vielen Universitäten und Schulen eingesetzten programmierbaren Roboter von **Leg Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren **Microcontroller** eine zu L_C **ähnliche Sprache** in die **Maschinensprache** des Microcontrollers zu kompilieren, da es **schneller** geht ein Programm direkt auf der Maschine, auf der man programmiert 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, also die Relationen in Abbildung 1.3.1 erfüllt sind^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**.

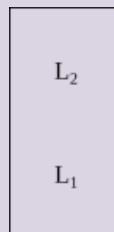
Kompilieren ist also auch **Übersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

^aAuch **Semantik** (Definition 1.15) genannt.

^bUnd ist auch zwischen **Passes** (Definition 1.42) möglich.

Definition 1.9: T-Diagramm Interpreter ✓

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



^aEarley 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 **horizontale Adjazenz** für **Übersetzung** steht.

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

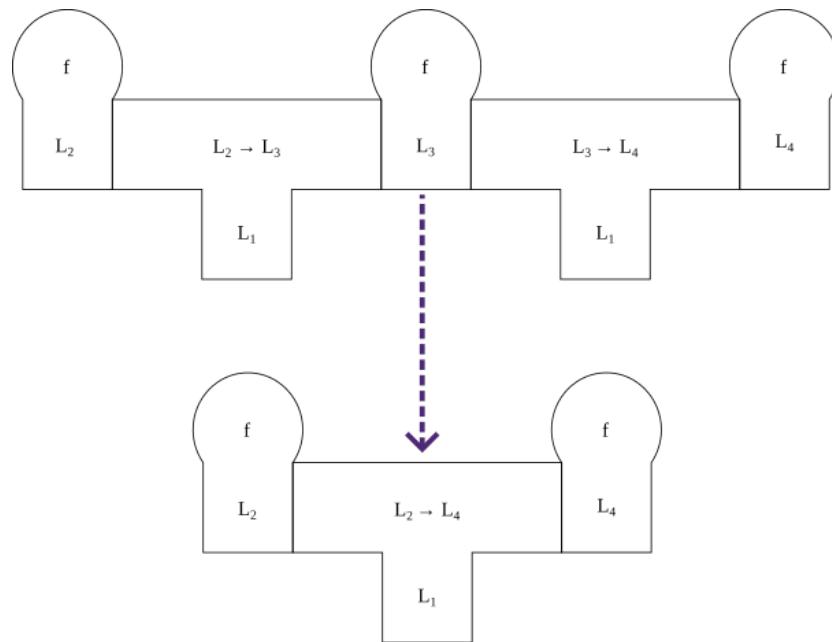


Abbildung 1.1: Horizontale Übersetzungszwischenschritte zusammenfassen.

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** vorher eingeführt zu haben, was die Begriffe **Symbol** (Definition 1.10), **Alphabet** (Definition 1.11) und **Wort** (Definition 1.12) beinhaltet.

Definition 1.10: Symbol

„Ein **Element** eines **Alphabets** Σ .“^a

^aNebel, „Theoretische Informatik“.

Definition 1.11: Alphabet

„Ist eine **endliche, nicht-leere** Menge von **Symbolen** (Definition 1.10).“^a

^aNebel, „Theoretische Informatik“.

Definition 1.12: Wort

„Ein **Wort** $w = a_1 \dots a_n \in \Sigma^*$ 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. Die **Kleenesche Hülle** ist die **größte Sprache** über Σ und **jede Sprache** über Σ ist eine **Teilmenge** davon. Es gilt des Weiteren: $\varepsilon \in \Sigma^*$.^a

^aNebel, „Theoretische Informatik“.

Definition 1.13: Formale Sprache

„**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.

^aNebel, „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 in verschiedenen **Syntaxen** (Definition 1.14) dargestellt sein.

Definition 1.14: Syntax

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

^aZ.B. die **Konkrete** (Definition 1.32) und **Abstrakte Syntax** (Definition 1.39), die später eingeführt werden.

^bThiemann, „Einführung in die Programmierung“.

Definition 1.15: Semantik

Bezeichnet alles was mit der **Bedeutung** von Wörtern einer **Formalen Sprache** zu tun hat.^a

^aThiemann, „Einführung in die Programmierung“.

Definition 1.16: Formale Grammatik

„Beschreibt, wie **Wörter** einer **Formalen 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 **Formale Grammatik** wird durch das Tupel $G = \langle N, \Sigma, P, S \rangle$ dargestellt, wobei“:

- $N \hat{=}$ **Nicht-Terminalsymbole**.
- $\Sigma \hat{=}$ **Terminalsymbole**, wobei $N \cap \Sigma = \emptyset$.^{a, b}
- $P \hat{=}$ Menge von **Produktionsregeln** $w \rightarrow v$, wobei $w, v \in (N \cup \Sigma)^*$ und $w \notin \Sigma^*$.^{c, d}
- $S \hat{=}$ **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.“^e

„Des Weiteren gibt es die von einer **Grammatik erzeugte Sprache**: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$ “^{f,g}

^aWeil mit ihnen **terminiert** wird.

^bKann auch als **Alphabet** bezeichnet werden.

^c w muss **mindestens** ein **Nicht-Terminalsymbol** enthalten.

^dBzw. $w, v \in C^*$ und $w \notin \Sigma^*$.

^eDie **Grammatik des PicoC-Compilers** lässt sich in eine Grammatik für die **Lexikalische Analyse** G_{Lex} und eine für die **Syntaktische Analyse** G_{Parse} unterteilen. Die **gesamte Grammatik** des **PicoC-Compilers** steht allerdings **vereinigt** in einer Datei.

^fDie **Nicht-Terminalsymbole** in w fallen dabei weg, weil $w \in \Sigma^*$

^gNebel, „Theoretische Informatik“.

Formale Sprachen lassen sich in Klassen der **Chromsky Hierarchie** (Definition 1.17) einteilen.

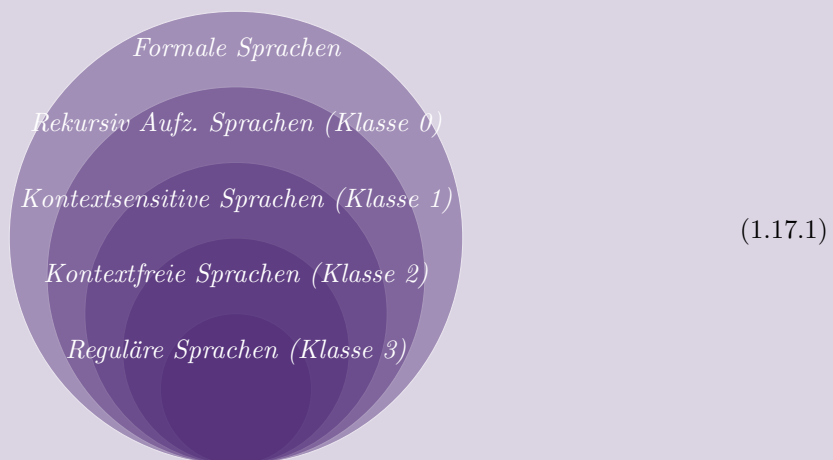
Definition 1.17: Chromsky Hierarchie



Eine Hierarchie, in der **Formale Sprachen** nach der **Komplexität** ihrer **Formalen Grammatiken** in verschiedene **Klassen** unterteilt werden. Jede dieser Klassen hat verschiedene **Eigenschaften**, wie **Entscheidungsprobleme**, die in einer Klasse **entscheidbar** und in einer anderen **unentscheidbar** sind usw.

Eine Sprache L_i ist in der **Chromsky Hierarchie** in der **Klasse** $i \in \{0, \dots, 3\}$, falls sie von einer Grammatik dieser **Klasse** i erzeugt^a werden kann.

Zwischen den Sprachmengen **benachbarter Klassen** in Abbildung 1.17.1 besteht eine **echte Teilmengebeziehung**: $L_3 \subset L_2 \subset L_1 \subset L_0$.^{b,c}



^a**Erzeugen** meint hier, dass **genau** die **Wörter** der Sprache sich mit der **Grammatik** ableiten lassen, **keines mehr** oder **weniger**.

^bZ.B. ist jede **Reguläre Sprache** auch eine **Kontextfreie Sprache**, aber nicht jede **Kontextfreie Sprache** ist auch eine **Reguläre Sprache**.

^cNebel, „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) bestimmt werden.

Definition 1.18: Reguläre Grammatik

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

$$A \rightarrow cB, \quad A \rightarrow c, \quad A \rightarrow \varepsilon \quad (1.18.1)$$

haben, wobei A, B **Nicht-Terminalsymbole** sind und c ein **Terminalsymbol** ist^{a,b}.“^c

^aDiese Definition einer **Regulären Grammatik** ist **rechtsregulär**, es ist auch möglich diese Definition **linksregulär** zu formulieren, aber diese Details sind für die Bachelorarbeit nicht relevant.

^bDadurch, dass die **linke** Seite immer nur ein **Nicht-Terminalsymbol** sein darf ist jede **Reguläre Grammatik** auch eine **Kontextfrei Grammatik**.

^cNebel, „Theoretische Informatik“.

Definition 1.19: Kontextfreie Grammatik

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

$$A \rightarrow v \quad (1.19.1)$$

haben, wobei A ein **Nicht-Terminalsymbol** ist und v ein beliebige Folge von **Grammatiksymbolen**^a ist.“^b

^aAlso eine beliebige Folge von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

^bNebel, „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 **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 Entscheidungsproblem, bei dem man zu einem **Wort** $w \in \Sigma^*$ und einer **Sprache** L als **Eingabe** 1 oder 0 **ausgibt**^a, je nachdem, ob dieses **Wort** w Teil der **Sprache** L ist ($w \in L$) oder **nicht** ($w \notin L$).

Das Wortproblem kann durch die folgende **Indikatorfunktion**^b zusammengefasst werden:^c

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

^aBzw. „ja“ oder „nein“ usw., es muss **nicht** umgedingt 1 oder 0 sein.

^bAuch **Charakteristische Funktion** genannt.

^cNebel, „Theoretische Informatik“.

1.2.1 Ableitungen

Jedes mit einem Compiler kompilierbare **Programm** kann mithilfe der **Grammatik** der **Sprache** des Compilers **abgeleitet** werden. Hierbei wird zwischen der **1-Schritt-Ableitungsrelation** (Definition 1.21) und der normalen **Ableitungsrelation** (Definition 1.22) unterschieden.

Definition 1.21: 1-Schritt-Ableitungsrelation

„Eine **binäre Relation** \Rightarrow , welche alle Anordnungen zweier Wörter $w_1, w_2 \in (N \cup \Sigma)^*$ in Relation zueinander setzt, die sich nur durch das **einmalige** Anwenden einer Produktionsregel auf w_1 voneinander

unterscheiden.

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

^aNebel, „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 v aus $(N \cup \Sigma)^*$, welches durch **beliebig häufiges** Anwenden von Produktionsregeln auf ein Wort u entsteht.

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

^aNebel, „Theoretische Informatik“.

Beim Ableiten kann auf verschiedene Weisen vorgegangen werden. Dasselbe **Programm** 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 und bei den **Linksableitungen** im Unterkapitel ?? relevant.

Definition 1.23: Links- und Rechtsableitung

„In jedem **Ableitungsschritt** wird bei **Typ-3- und Typ-2-Grammatiken** auf das am **weitesten links** (**Linksableitung**) bzw. **rechts** (**Rechtsableitung**) stehende **Nicht-Terminalsymbol** eine Produktionsregel angewandt. ^a

Mit diesem Vorgehen kann man jedes ableitbare Wort generieren, denn dieses Vorgehen entspricht **Tiefensuche** von **links-nach-rechts** bzw. **rechts-nach-links**. ^b

^aBei **Typ-1- und Typ-0-Grammatiken** ist es statt einem **Nicht-Terminalsymbol** die **linke Seite** einer Produktion.

^bNebel, „Theoretische Informatik“.

Ob eine Grammatik **mehrdeutig** (Definition 1.25) ist, kann durch Betrachtung **Formaler Ableitungsbäume** (Definition 1.24) festgestellt werden. **Formale Ableitungsbäume** werden im Unterkapitel 1.4 nochmal relevant, da in der **Syntaktischen Analyse** Formale Ableitungsbäume (Definition 1.34) als eine **compilerinterne Datenstruktur** umgesetzt werden.

Definition 1.24: Formaler Ableitungsbaum

Ist ein Baum, in dem die Syntax eines **Wortes**^a nach den **Produktionen** einer 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 **compilerinterne Datenstruktur** umsetzt.

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

^aZ.B. **Programmcode**.

^bNebel, „Theoretische Informatik“.

In Abbildung 1.24.2 ist ein Beispiel für einen **Formalen Ableitungsbaum** zu sehen, der sich aus der **Ableitung 1.24.1** nach der im **Dialekt der Erweiterten Backus-Naur-Form des Lark Parsing Toolkit** (Definition ??) angegebenen **Grammatik $G = \langle N, \Sigma, P, add \rangle$** 1.1 ergibt.

NUM	$::=$	$"4" \mid "2"$	L_Lex
ADD_OP	$::=$	$"+"$	
MUL_OP	$::=$	$"*"$	
mul	$::=$	$mul \ MUL_OP \ NUM \mid NUM$	L_Parse
add	$::=$	$add \ ADD_OP \ mul \mid mul$	

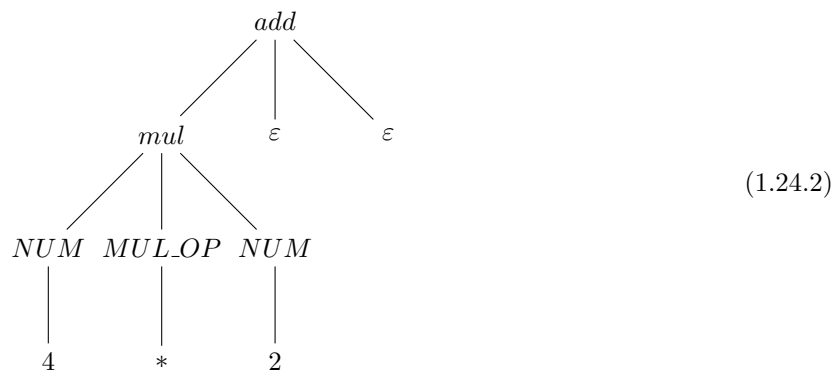
Grammatik 1.1: Grammatik für einen Ableitungsbaum in EBNF

Anmerkung 🔍

Werden die **Produktionen** einer Grammatik 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.

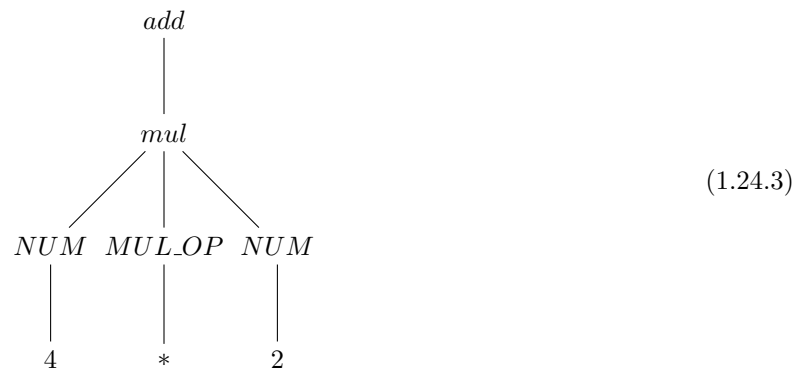
$$add \Rightarrow mul \Rightarrow mul \ MUL_OP \ NUM \Rightarrow NUM \ MUL_OP \ NUM \Rightarrow^* "4" \ "*" \ "2" \quad (1.24.1)$$

Bei Ableitungsbäumen gibt es **keine** einheitliche **Regelung**, wie damit umgegangen wird, wenn die **Alternativen** einer Produktion unterschiedliche viele **Nicht-Terminalsymbole** enthalten. Es gibt einmal die Möglichkeit, wie im Ableitungsbaum 1.24.2 von der **Maximalzahl** auszugehen und beim Nicht-Erreichen der Maximalzahl entsprechend der **Differenz zur Maximalzahl** viele **Blätter** mit dem **leeren Wort ε** hinzuzufügen.



Eine andere Möglichkeit ist, wie im Ableitungsbaum 1.24.3, nur die in den gewählten Produktionen vorhandenen **Nicht-Terminalsymbole** als Kinder hinzuzufügen³.

³Diese Option wurde beim **PicoC-Compiler** gewählt.



1.2.2 Präzedenz und Assoziativität

Für einen Compiler ist es notwendig, dass die Grammatik des Compilers keine **Mehrdeutige Grammatik** (Definition 1.25) ist, denn sonst können unter anderem die **Assoziativität** (Definition 1.26) und die **Präzedenz** (Definition 1.27) der verschiedenen **Operatoren** nicht gewährleistet werden, wie später in Unterkapitel ?? an einem Beispiel demonstriert wird. Ein Schema, um die Grammatiken zu definieren, die **nicht mehrdeutig** sind, wird in Unterkapitel ?? genauer erklärt.

Definition 1.25: Mehrdeutige Grammatik

„Eine Grammatik ist **mehrdeutig**, wenn es ein Wort $w \in L(G)$ gibt, das mehrere **Ableitungsbäume** zulässt“^{a,b,c}

^aFür die Bedeutung von $L(G)$, siehe Definition 1.16.

^b**Alternativ** geht auch die Definition: „Wenn es für w **mehrere** unterschiedliche **Linksableitungen** gibt“.

^cNebel, „Theoretische Informatik“.

Definition 1.26: Assoziativität

„Bestimmt, welcher Operator aus einer Reihe von Operatoren mit **gleicher Präzedenz** (Definition 1.27) **zuerst** ausgewertet wird.“

Es wird 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,b}

^aNystrom, *Parsing Expressions - Crafting Interpreters*.

^b2.1.7 Vorrangregeln und Assoziativität.

Bei **Assoziativität** ist z.B. der **Multiplikationsoperator** $*$ ein Beispiel für einen **linksassoziativen** Operator und ein **Zuweisungsoperator** = ein Beispiel für einen **rechtsassoziativen** Operator. In Abbildung 1.2 ist ein Beispiel hierfür dargestellt, indem die resultierenden **Auswertungsreihenfolgen** mithilfe von Klammern () veranschaulicht sind.

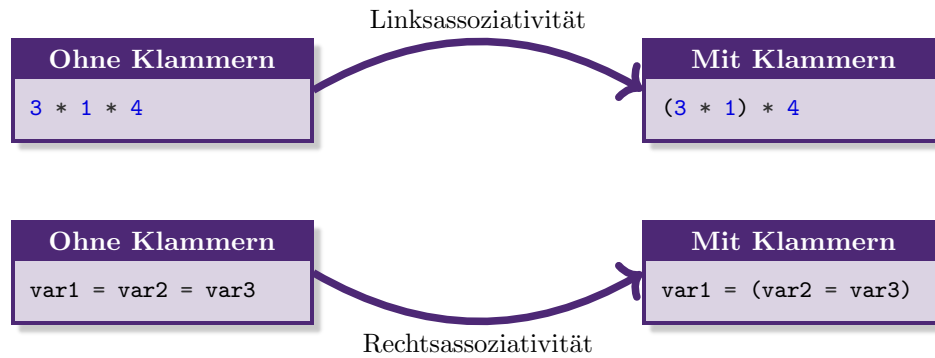


Abbildung 1.2: Veranschaulichung von Linksassoziativität und Rechtsassoziativität.

Definition 1.27: Präzedenz

„Bestimmt für einem Ausdruck, der eine Mischung *verschiedener* Operatoren enthält, welcher Operator *zuerst* ausgewertet wird. Operatoren mit einer *höheren Präzedenz* werden *vor* Operatoren mit einer *niedrigeren Präzedenz* ausgewertet.“^{ab}

^aEin anderes Wort für **Präzedenz** ist **Vorrang**.

^bNystrom, *Parsing Expressions · Crafting Interpreters*.

Die Mischung der Operatoren für **Subtraktion** '-' und für **Multiplikation** *, welche beide eine **unterschiedliche Präzedenz** haben, ist ein Beispiel für den Einfluss von Präzedenz. In Abbildung 1.3 ist ein Beispiel hierfür dargestellt, indem mithilfe der Klammern () die resultierende **Auswertungsreihenfolge** veranschaulicht ist. Im Beispiel in Abbildung 1.3 ist durch die beiden **Subtraktionsoperatoren** '-' nacheinander und den darauffolgenden **Multiplikationsoperator** *, 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 Architektur-patterns** (Definition 1.1) bei der Implementierung von Compilern. Die Aufgabe der Lexikalischen Analyse ist in einem ersten Schritt in einem Eingabewort⁴ **Lexeme** (Definition 1.28) zu finden, die mit einer **Grammatik** für die Lexikalische Analyse G_{Lex} abgeleitet werden können.

Definition 1.28: Lexeme

Ein **Lexeme** ist ein **Teilwort** aus dem **Eingabewort**, welches mit einer **Grammatik** für die Lexikalische Analyse G_{Lex} abgeleitet werden kann.^a

^aThiemann, „Compilerbau“.

⁴Z.B. dem Inhalt einer Datei, welche in **UTF-8** kodiert ist.

Diese **Lexeme** werden vom **Lexer** (Definition 1.30) im **Eingabewort** identifiziert und **Tokens** (Definition 1.29) zugeordnet. Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 1.29: Token

Ist ein **Tupel** (T, V) mit einem **Token** T und einem **Tokenwert** V . Ein **Token** T kann hierbei als ein **Überbegriff** für eine möglicherweise unendliche Menge verschiedener **Tokenwerte** V verstanden werden^a.

^aZ.B. gibt es im **PicoC-Compiler** viele verschiedene **Tokenwerte**, wie z.B. 42, 314 oder 12, welche alle unter dem **Token** **NUM**, für Zahl zusammengefasst sind.

Definition 1.30: Lexer (bzw. Scanner oder auch Tokenizer)

Ein **Lexer** ist eine **Totale Funktion**^a: $lex : \Sigma^* \rightarrow (T \times V)^*$, $w \mapsto (t_1, v_1) \dots (t_n, v_n)$, welche ein **Eingabewort**^b $w \in \Sigma^*$ auf eine **endliche Folge von Tokens** $(t_1, v_1) \dots (t_n, v_n) \in (T \times V)^*$ abbildet.

Die **Definitionsmenge** der **Totalen Funktion** lex erlaubt nur **Wörter**, die sich mit der **Grammatik** für die Lexikalische Analyse G_{Lex} ableiten lassen.^c

^aAlternativ könnte man die Funktion lex auch als **Partielle Funktion** definieren, aber das würde zum Ausdruck bringen, dass der **Lexer** bei einem **Eingabewort**, das sich **nicht** mit der Grammatik für die Lexikalische Analyse G_{Lex} ableiten lässt trotzdem durchläuft. Die Funktion lex als **Totale Funktion** zu definieren drückt eher aus, dass ein **Eingabewort**, das **nicht** in der **Definitionsmenge** liegt zu einer **Fehlermeldung** führt.

^bZ.B. **Quellcode** eines **Eingabeprogramms**.

^cThiemann, „Compilerbau“.

Ist das Abbilden eines **Eingabeworts** w auf eine **Folge von Tokens** $(t_1, v_1) \dots (t_n, v_n)$ nicht möglich, da das Eingabewort **Teilwörter** enthält, die sich **nicht** mit der **Grammatik** für Lexikalische Analyse G_{Lex} ableiten lassen, so wird in diesem Fall eine **Fehlermeldung** (Definition 1.52) ausgegeben.

Anmerkung

Um Verwirrung vorzubeugen ist es wichtig die **kontextabhängigen** unterschiedliche Bedeutungen des Wortes **Symbol** hervorzuheben:

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

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

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

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

^aDas ist der Grund, warum in Kapitel ?? die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, **Symbole** genannt wird.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die **Syntaktische Analyse** unwichtigen Symbole, wie Leerzeichen $_$, Newline $\backslash n$ ⁵ und Tabs $\backslash t$ aus dem Eingabewort w **herauszufiltern**. Das geschieht im **Lexer**, indem dieser für alle unwichtigen Symbole bzw. Folgen von Symbolen⁶ **kein Token** in der Folge von Tokens $(t_1, v_1) \dots (t_n, v_n)$ vorsieht.

⁵In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Programmiersprache, die man zur Implementierung des Lexers nutzt **wegabstrahiert**.

⁶Bzw. **Teilwörter** des Eingabeworts.

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

Wie es in Tabelle 1.1 zu sehen ist, gibt es für verschiedene **Bezeichner**, wie z.B. `my_fun`, `my_var` oder `my_const` und verschiedene **Zahlen**, wie z.B. 42, 314 oder 12 passende **Tokentypen** `NAME` und `NUM`^{7 8}, die einen **Überbegriff** darstellen. Für **Lexeme**, wie `if` oder `}` sind die **Tokentypen** dagegen genau die Bezeichnungen, die man diesen beiden Folgen von Symbolen geben würde, nämlich `IF` und `RBRACE`.

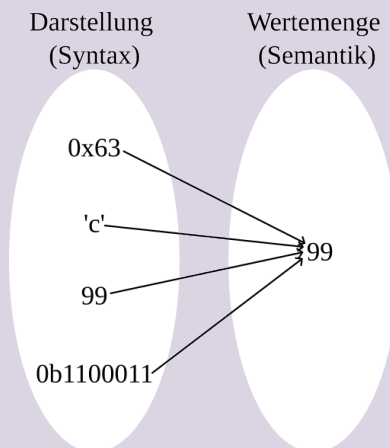
Lexeme	Token
42, 314	<code>Token('NUM', '42')</code> , <code>Token('NUM', '314')</code>
<code>my_fun</code> , <code>my_var</code> , <code>my_const</code>	<code>Token('NAME', 'my_fun')</code> , <code>Token('NAME', 'my_var')</code> , <code>Token('NAME', 'my_const')</code>
<code>if</code> , <code>}</code>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

Tabelle 1.1: Beispiele für Lexeme und ihre entsprechenden Tokens.

Ein **Lexeme** ist nicht immer das gleiche, wie der **Tokenwert** V , denn wie in Tabelle 1.1 zu sehen ist, kann z.B. im Fall von L_{PicoC} der **Tokenwert** 99 durch zwei verschiedene **Literale** (Definition 1.31) dargestellt werden. Einmal kann der **Tokenwert** 99 als **ASCII-Zeichen** `'c'` dargestellt werden, das dann als **Tokenwert** den entsprechenden **Index** in der ASCII-Tabelle erhält, nämlich 99 und des Weiteren auch in **Dezimalschreibweise** als 99⁹. Der **Tokenwert** ist der letztendlich verwendete **Wert** an sich, **unabhängig** von der Darstellungsform.

Definition 1.31: Literal

Eine von möglicherweise vielen weiteren **Darstellungsformen**^a für ein und denselben **Wert** eines **Datentyps**.^b



^aAlso verschiedene **Zeichenketten**.

^bThieman, „Einführung in die Programmierung“.

⁷Bzw. wenn man sich **nicht Kurzformen** sucht, wären `IDENTIFIER` und `NUMBER` die **passenden Bezeichnungen**.

⁸Die **Bezeichnungen** der **Tokentypen** wurden im **PicoC-Compiler** so gewählt, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner haben will, damit unter anderem auch **mehr Code** in eine Zeile passt.

⁹Die Programmiersprache L_{Python} erlaubt es z.B. den **Wert** 99 auch mit den **Literalen** `0b1100011` und `0x63` darzustellen.

Die **Grammatik** für die Lexikalische Analyse G_{Lex} ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut¹⁰ und sich **nichts merkt**, also unabhängig davon, **was** für Symbole und **wie oft** bestimmte Symbole **davor** aufgetaucht sind funktioniert.

Der Grund, weshalb ein **Lexer** relativ **unkompliziert** zu implementieren ist, ist weil dieser nur **Lexeme** erkennen muss, welche durch eine **Reguläre Grammatik** beschrieben sind. **Parser**, welche im nächsten Unterkapitel 1.4 erklärt werden, sind dagegen deutlich **komplexer** zu implementieren, da diese bei den meisten Programmiersprachen eine **Kontextfreie Grammatik** umsetzen müssen¹¹.

Anmerkung 🔍

Auch für den **PicoC-Compiler** lässt sich aus der im **Dialekt der Backus-Naur-Form des Lark Parsing Toolkit** (Definition ??) spezifizierten Grammatik für die Lexikalische Analyse ?? G_{PicoC_Lex} schlussfolgern, dass diese **Grammatik** eine **Reguläre Grammatik** ist, da alle ihre **Produktionen** die Definition 1.18 einer **Regulären Grammatik** erfüllen.

Produktionen mit **Alternative**, wie z.B. $DIG_WITH_0 ::= "0" \mid DIG_NO_0$ in Grammatik ?? sind **unproblematisch**, denn sie können immer auch als $\{DIG_WITH_0 ::= "0", DIG_WITH_0 ::= DIG_NO_0\}$ umgeschrieben werden und z.B. $DIG_WITH_0^*$, $(LETTER \mid DIG_WITH_0 \mid _)"^+$ und $"_"._"\sim$ in Grammatik ?? können alle zu **Alternativen** umgeschrieben werden, wie es in Definition ?? gezeigt wird und diese **Alternativen** können wie gerade gezeigt umgeformt werden, 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 **Chomsky Hierarchie** (Definition 1.17) **regulär**.

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 1.4 die Lexikalische Analyse an einem Beispiel veranschaulicht.

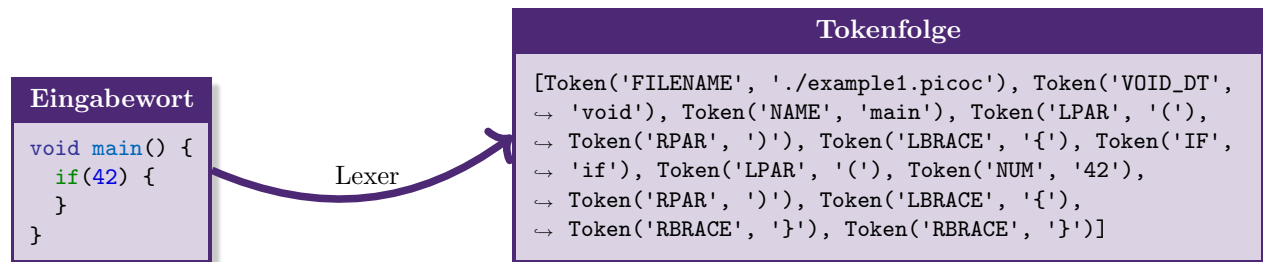


Abbildung 1.4: Veranschaulichung der Lexikalischen Analyse.

Anmerkung 🔍

Das Symbol ↵ zeigt in **Codebeispielen** 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** $\text{fun}(\text{arg})$ und **Codeblöcke** $\text{if}(1)\{\}$ syntaktische Mittel verwenden, die es notwendig machen sich zu merken, wieviele

¹⁰Man nennt das auch einem **Lookahead** von 1.

¹¹Hierzu kann z.B. der **Earley Erkennen** in Definition ?? im ?? als Beispiel genannt werden, der ein Bestandteil eines **Earley Parsers** (Definition ??) ist und bereits deutlich **komplexer** ist als ein typischer **Lexer**.

öffnende runde Klammern (bzw. öffnende geschweifte Klammern { es zu einem bestimmten Zeitpunkt gibt, die noch nicht durch eine entsprechende schließende runde Klammer) bzw. schließende geschweifte Klammer } geschlossen wurden. Diese syntaktischen Mittel lassen 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.

Anmerkung

Für den **PicoC-Compiler** lässt sich aus der im **Dialekt der Backus-Naur-Form des Lark Parsing Toolkit** (Definition ??) spezifizierten Grammatik für die Syntaktische Analyse ?? G_{PicoC_Parse} schlussfolgern, dass diese eine **Kontextfreie Grammatik**, aber **keine Reguläre Grammatik** ist, da alle ihre Produktionen die Definition 1.19 einer **Kontextfreien Grammatik** erfüllen, aber **nicht** die Definition 1.18 einer **Regulären Grammatik**.

Es lässt sich sehr leicht erkennen, dass die Grammatik ?? eine **Kontextfreie Grammatik** ist, da **alle Produktionen** auf der **linken Seite** des $::=$ -Symbols immer nur ein einzelnes **Nicht-Terminalsymbol** haben und sich auf der **rechten Seite** eine **beliebige Folge** von **Grammatiksymbolen**^a befindet.

Es lässt sich wiederum sehr einfach erkennen, dass die Grammatik ?? **keine Reguläre Grammatik** ist, denn z.B. bei der Produktion $if_stmt ::= "if" ("logic_or") exec_part$ ist das **Nicht-Terminalsymbol** *logic_or* von den **Terminalsymbolen** für eine **öffnende Klammer** { und eine **schließende Klammer** } eingeschlossen, was mit einer Regulären Grammatik **nicht** ausgedrückt werden kann.

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

^aAlso eine **beliebige Folge** von **Nicht-Terminalsymbolen** und **Terminalsymbolen**.

Die **Syntax**, in welcher ein **Programm** vor dem kompilieren in einer Textdatei aufgeschrieben ist, wird auch als **Konkrete Syntax** (Definition 1.32) bezeichnet. In einem Zwischenschritt, dem **Parsen**, wird aus diesem Programm mithilfe eines **Parsers** (Definition 1.35) ein **Ableitungsbaum** (Definition 1.34) generiert, der als Zwischenstufe hin zum einem **Abstrakten Syntaxbaum** (Definition 1.41) dient. Beim Compilerbau ist es förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Ableitungsbaumes** und dann erst des **Abstrakten Syntaxbaumes**.

Definition 1.32: Konkrete Syntax

Bezeichnet alles, was mit dem **Aufbau** von Wörtern^a zu tun hat, die nach einer **Konkreten Grammatik** $G_{Lex} \uplus G_{Parse}$ (Definition 1.33) abgeleitet wurden.

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 einer **Konkreten Grammatik** abgeleitet sind^c.

Ein **Programm**, wie es in einer **Textdatei** nach der Konkreten Grammatik^d abgeleitet steht, bevor man es kompiliert, ist in **Konkreter Syntax** aufgeschrieben.^e

^aBzw. **Programmen**.

^bVor allem bei **Programmiersprachen**.

^cWenn ein Programm z.B. **nicht deklarierte Variablen** hat und aufgrund dessen **nicht kompiliert** werden kann, hält dieses die gesamte Syntax **nicht** ein, kann allerdings so nach einer **Konkreten Grammatik** abgeleitet werden. Solche Details werden üblicherweise **nicht** in eine **Konkrete Grammatik** mitaufgenommen.

^d**Vereinigungsgrammatik**, wie in Definition 1.16 erklärt.

^eG. Siek, *Essentials of Compilation*.

Um einen kurzen Begriff für die **Grammatik** zu haben, welche die **Konkrete Syntax** einer Sprache beschreibt, wird diese im Folgenden als **Konkrete Grammatik** (Definition 1.33) bezeichnet.

Definition 1.33: Konkrete Grammatik

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

^aVereinigungsgrammatik, wie in Definition 1.16 erklärt.

Definition 1.34: Ableitungsbaum (bzw. Konkreter Syntaxbaum, engl. Derivation Tree)

Compilerinterne Datenstruktur für den **Formalen Ableitungsbaum** (Definition 1.24) eines in **Konkreter Syntax** geschriebenen Programmes.

Die **Blätter**, die beim Formalen Ableitungsbaum das **leere Wort** ε oder **Terminalsymbole** Σ einer Konkreten Grammatik $G = \langle N, \Sigma, P, S \rangle$ sein können, sind in dieser Datenstruktur **Tokens** (T, W) und die **Inneren Knoten** entsprechen weiterhin **Nicht-Terminalsymbolen** N einer Konkreten Grammatik $G = \langle N, \Sigma, P, S \rangle$. In dieser Datenstruktur wird allerdings nur der Teil eines Formalen Ableitungsbaumes dargestellt, der den **Ableitungen** einer Grammatik G_{Parse} entspricht. Die **Tokens** sind in der Syntaktischen Analyse ein **atomarer Grundbaustein**^a, daher sind die **Ableitungen** der Grammatik G_{Lex} uninteressant.^b

^aNicht mehr weiter teilbar.

^bJSON parser - Tutorial — Lark documentation.

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

Definition 1.35: Parser

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

^aZ.B. ein **Programm** in einer **Textdatei**.

^bEs gibt allerdings auch **alternative Definitionen**, denen nach ein Parser ein Programm ist, dass ein Eingabewort von **Konkreter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die hiesige Definition 1.35 verwendet.

^cJSON parser - Tutorial — Lark documentation.

Anmerkung

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines **Parsers** und ist ausschließlich für die **Lexikalische Analyse** verantwortlich. In einem alltäglicheren Szenario entspricht **lexen** z.B. bei einem Wanderausflug dem Nachschlagen in einem Insekten**lexikon** und dem Aufschreiben, welcher Insektenart man in welcher **Reihenfolge** begegnet ist, mit jeweils einem Bild des konkreten Exemplars, dem man 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 gerade angefangene Beispiel zurückzukommen, dem **Weiterverarbeiten** der **Beziehungen** zwischen den Insektenbegegnungen unter Berücksichtigung des **örtlichen Kontexts**^b in eine **taugliche Form**^c.

In der Weiterverarbeitung könnte ein **Interpreter** die in eine taugliche Form gebrachten Daten interpretieren und daraus bestimmte **Schlüsse ziehen** und ein **Compiler** könnte die Daten vielleicht in eine für Menschen **leichter verständliche Sprache** kompilieren^d.

^aDas würde z.B. der Rolle eines **Semikolon** ; in der Sprache L_{PicoC} entsprechen.

^bDas entspricht z.B. **Semikolons** ;, die vorhin bereits als Beispiel genannt wurden.

^cZ.B. gibt es bestimmte **Wechselbeziehungen** zwischen Insekten, Insekten beeinflussen sich gegenseitig und ihre Umwelt.

^dNormalerweise kompiliert man in eine für die **CPU** verständliche Sprache.

Die vom **Lexer** aus dem Eingabewort generierten **Tokens** werden in der **Syntaktischen Analyse** vom **Parser** als **Wegweiser** verwendet, da je nachdem, in welchem Kontext bestimmte **Tokens** (T, V) mit einem bestimmten **Token** T auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht, die zum Parsen eines Eingabeworts notwendig ist. Dabei wird in der Konkreten Grammatik G_{Parse} nach **Token** T unterschieden und **nicht** nach **Token** V , 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 **Token** V ist erst später in der **Code Generierung** wieder relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Erkenner** (Definition 1.36), denn ein Parser löst das **Wortproblem** (Definition 1.20) für die **Sprache**, in welcher das Programm, welches kompiliert werden soll geschrieben ist und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Erkennungsalgorithmus¹² gesammelt wurden den **Ableitungsbaum**.

Definition 1.36: Erkenner (bzw. engl. Recognizer)



Entspricht in seiner Funktion einem **Kellerautomaten**^a, in dem **Wörter** bestimmter **Kontextfreier Sprachen** erkannt werden.

Es ist ein **Algorithmus**, der erkennt, ob ein **Eingabewort** sich mit der **Konkreten Grammatik** einer Sprache ableiten lässt. Der Algorithmus überprüft also, ob das **Eingabewort** Teil der **Sprache** ist, die von der **Konkreten Grammatik** beschrieben wird oder **nicht**. Ein **Erkenner** löst folglich das **Wortproblem** (Definition 1.20).^b

^a**Automat** mit dem **Kontextfreie Grammatiken** erkannt werden.

^bThiemann, „Compilerbau“.

Anmerkung



Für das **Parsen** gibt es grundsätzlich **drei** geläufige Ansätze, die unterschieden werden:

- **Top-Down Parsing:** Der Algorithmus arbeitet von **oben-nach-unten**, also anschaulich von der **Wurzel** zu den **Blättern** eines im Nachhinein oder parallel dazu generierten Ableitungsbaumes. Dementsprechend fängt der Algorithmus mit dem **Startsymbol** einer **Konkreten Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man die **Folge von Terminalsymbolen** hat, die sich zum gewünschten **Eingabewort** abgeleitet hat oder sich herausstellt, dass dieses **nicht** abgeleitet werden kann.^a

Der Grund, warum **Linksableitungen** verwendet werden und nicht z.B. **Rechtsableitungen**, ist, weil versucht wird das **Eingabewort** von **links nach rechts** mithilfe von Terminalsymbolen aus Ableitungen nachzubilden. Das passt gut damit zusammen, dass durch die **Linksableitung**

¹²Bzw. engl. recognition algorithm.

die Terminalsymbole von **links-nach-rechts** erzeugt werden.

Welche der **Produktionen** für ein **Nicht-Terminalsymbol** angewandt wird, wenn es mehrere **Alternativen** gibt, wird entweder durch **Backtracking** oder durch **Vorausschauen** entschieden. Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg** (Definition ??). Für diese Methode muss allerdings, wenn die Konkrete Grammatik eine **Linksrekursive Grammatik** (Definition ??) ist, diese umgeformt werden, um jegliche **Linksrekursion** aus dieser zu **entfernen**, da diese sonst 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 das gewünschte Eingabewort abgeleitet ist oder alle **Alternativen** für das momentane **Nicht-Terminalsymbol** abgesucht sind. Das wird solange durchgeführt, bis man wieder beim **Startsymbol** angekommen ist und da auch alle **Alternativen** abgesucht sind. Dies bedeutet dann, dass das **Eingabewort** sich **nicht** mit der verwendeten Konkreten Grammatik ableiten lässt.^b

Wenn man eine **LL(k)**-Grammatik hat, ist **Backtracking** nicht notwendig und es reicht einfach nur immer k **Tokens** im Eingabewort **vorausschauen**^c. **Mehrdeutige Grammatiken** sind dadurch ausgeschlossen, weil **LL(k)** keine **Mehrdeutigkeit** zulässt.^d

- **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.^e
- **Chart Parsing:** Es wird **Dynamische Programmierung** verwendet, indem **partielle Zwischenergebnisse** in einer **Tabelle**^f gespeichert werden und **wiederverwendet** werden können. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist. **Chart Parser** können dabei **top-down** oder **bottom-up** Ansätze umsetzen^{g h}.

^aWhat is Top-Down Parsing?

^bDiese Methode des Parsens 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.

^cDas wird auch als **Lookahead** von k bezeichnet.

^dDiese 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.

^eWhat is Bottom-up Parsing?

^fBzw. einem **Chart**.

^gDa 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.

^hDer **Earley Parser** von **Lark**, welchen der **PicoC-Compiler** verwendet, fällt unter diese Kategorie.

Ein **Abstrakter Syntaxbaum** (Definition 1.41) wird durch einen **Transformer** (Definition 1.37) und **Visitors** (Definition 1.38) mithilfe eines **Ableitungsbaumes** 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 1.39).

Definition 1.37: Transformer



Ein Programm, das von **unten-nach-oben**^a, nach dem Prinzip der **Breitensuche** alle Knoten des **Ableitungsbaumes** besucht. Beim Antreffen eines bestimmten Knoten des **Ableitungsbaumes** erzeugt

es je nach Kontext einen entsprechenden Knoten des **Abstrakten Syntaxbaumes** und setzt diesen anstelle des Knotens des **Ableitungsbaumes** und konstruiert so Stück für Stück den **Abstrakten Syntaxbaum**.^b

^a**Zur Erinnerung:** In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.

^b*Transformers & Visitors — Lark documentation.*

Definition 1.38: Visitor

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

^a**Zur Erinnerung:** In der **Informatik** wachsen Bäume von **oben-nach-unten**, von der **Wurzel** zur den **Blättern**.

^bKann 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 1.39: Abstrakte Syntax

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

Ein **Abstrakter Syntaxbaum**, der zur Kompilierung eines Eingabewortes^a generiert wurde, befindet sich in **Abstrakter Syntax**.^b

^aZ.B. ein **Programm** in einer **Textdatei**.

^bG. Siek, *Essentials of Compilation*.

Um einen kurzen Begriff für die **Grammatik** zu haben, welche die **Abstrakte Syntax** einer Sprache beschreibt, wird diese im Folgenden als **Abstrakte Grammatik** (Definition 1.40) bezeichnet.

Definition 1.40: Abstrakte Grammatik

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

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

Definition 1.41: Abstrakter Syntaxbaum (bzw. engl. Abstract Syntax Tree, kurz AST)

Ist eine **compilerinterne Datenstruktur**, welche eine **Abstraktion** eines **Ableitungsbaumes** darstellt. In den Aufbau des **Abstrakten Syntaxbaumes** ist das Erfordernis eines **leichten Zugriffs** und einer **leichten Weiterverarbeitbarkeit** eingeflossen ist. Bei der Betrachtung eines **Knoten**, der zusammen mit seinen Kinderknoten für einen Teil eines Eingabewortes^a 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 wichtige Informationen für den **Kompiliervorgang** und **Fehlermeldungen** enthalten.^b

^aZ.B. ein **Programm** in einer **Textdatei**.

^bG. Siek, *Essentials of Compilation*.

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**^a erzeugt wird. Es ist damit **immer** die Sprache gemeint, die **kompiliert** werden soll^b und zu deren Zweck der **Abstrakte Syntaxbaum** überhaupt konstruiert bzw. die **Abstrakte Grammatik** überhaupt definiert wird. Für die tatsächliche Sprache, die durch die **Abstrakte Grammatik** beschrieben wird, interessiert man sich nie wirklich explizit. Die Sprache L besitzt eine **Konkrete** und eine **Abstrakte Syntax** mit entsprechenden **Grammatiken** und **Bäumen**, die man zu beiden **definieren** bzw. **ableiten**^c kann. Diese **Konvention** wurde aus dem Buch G. Siek, *Essentials of Compilation* übernommen.

^aNach welcher der **Abstrakte Syntaxbaum** konstruiert ist.

^bBzw. es ist die **Sprache**, welche durch die **Konkrete Grammatik** beschrieben wird.

^cBzw. **konstruieren** beim **Abstrakten Syntaxbaum**.

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** zu ersetzen, damit der **Zugriff** auf Knoten des Abstrakten Syntaxbaumes immer **einheitlich** erfolgen kann und auch, da manche **Token**typen noch **nicht optimal benannt** sind.

In z.B. der Sprache L_{PicoC} werden manche Symbole mehrfach verwendet, wie z.B. das Symbol ‘-’, welches für die **binäre Subtraktionsoperation** als auch die **unäre Minusoperation** verwendet wurde. Der verwendete **Token**typ dieses Symbols lautet beim **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 **unterschiedlich benannte 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 unterscheiden, 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 Verarbeitungsschritten¹³ möglichst **einfach weiterverarbeitet** werden kann.

Auf der **linken** Seite in Abbildung 1.5 wird das Beispiel 1.24.3 aus Unterkapitel 1.2.1 fortgeführt. Dieses Beispiel stellt die **Ableitung** des **Arithmetischen Ausdruck** $4 * 2$ nach der Konkreten Grammatik 1.2¹⁴ in einem **Ableitungsbaum** dar. Die Konkrete Grammatik 1.2 berücksichtigt die **höhere Präzedenz** der **Multiplikation** $*$. Allerdings handelt es sich bei diesem Ableitungsbaum **nicht** um einen **Formalen Ableitungsbaum**, sondern um eine **compilerinterne Datenstruktur** für einen solchen¹⁵. Die **Blätter** sind dementsprechend **Tokens**, die mithilfe der Grammatik L_{Lex} generiert wurden. Der **Ableitungsbaum** beschränkt sich somit auf den Teil der **Ableitung**, der sich aus der Grammatik L_{Parse} ergibt.

Auf der **rechten** Seite in Abbildung 1.5 wird der Ableitungsbaum zu einem Abstrakten Syntaxbaum **abstrahiert**, der nach der Abstrakten Grammatik 1.3 konstruiert ist. In der **Abstrakten Grammatik** 1.3

¹³Den verschiedenen **Passes**.

¹⁴Die **Konkrete Grammatik** ist hierbei im **Dialekt der Erweiterter Backus-Naur-Form** des Lark Parsing Toolkit (Definition ??) angegeben.

¹⁵Die **Baumdarstellung** wird nur zur **Veranschaulichung** genutzt.

sind jegliche Produktionen **wegabstrahiert**, die in der **Konkreten Grammatik 1.2** den Zweck erfüllen die **Präzidenz** umzusetzen und **mehrdeutig** zu verhindern. Aus diesem Grund gibt es nur noch einen **allgemeinen Knoten** für **binäre Operationen** $\text{BinOp}(\langle \text{exp} \rangle, \langle \text{bin_op} \rangle, \langle \text{exp} \rangle)$.

NUM	$::=$	"4" "2"	L_Lex
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	$::=$	$\text{BinOp}(\langle \text{exp} \rangle, \langle \text{bin_op} \rangle, \langle \text{exp} \rangle)$ $\text{Num}(\langle \text{str} \rangle)$

Grammatik 1.3: Produktionen für Abstrakten Syntaxbaum in ASF

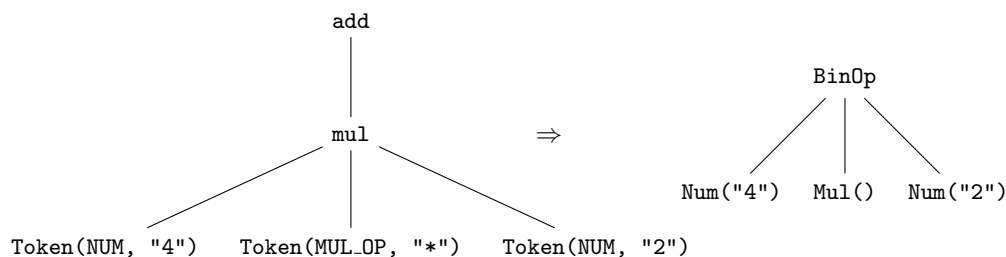


Abbildung 1.5: Veranschaulichung des Unterschieds zwischen Ableitungsbaum und Abstraktem Syntaxbaum.

Die **Baumdatenstruktur** des **Ableitungsbaumes** erfüllt den Zweck, dass sich aus dieser durch Iteration über diese besonders einfach ein **Abstrakter Syntaxbaum** konstruieren lässt, da auf diese Weise der Ableitungsbaum und Abstrakte Syntaxbaum, beide durch eine **Baumdatenstruktur** umgesetzt sind¹⁶. Beim **Abstrakten Syntaxbaum** ermöglicht die **Baumdatenstruktur** es die Operationen, die bei einem Compiler bzw. einem Interpreter bei der Weiterverarbeitung auszuführen sind, möglichst **effizient** auszuführen und auf **unkomplizierte** Weise direkt zu erkennen, welche auszuführen sind.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, sind in Abbildung 1.7 die einzelnen **Zwischenschritte** von den **Tokens** der Lexikalischen Analyse zum **Abstrakten Syntaxbaum** anhand des fortgeführten Beispiels aus Abbildung 1.4 und Unterkapitel 1.3 veranschaulicht. In Abbildung 1.7 werden die Darstellungen des **Ableitungsbaumes** und des **Abstrakten Syntaxbaumes** verwendet, wie sie vom **PicoC-Compiler** ausgegeben werden.

In der Darstellung von Bäumen beim **PicoC-Compiler**, stellen die verschiedenen **Einrückungen** die verschiedenen **Ebenen** dieser Bäume dar. Kanten gibt es **keine**, diese müssen sich zwischen den Knoten **dazugedacht** werden. Diese dazuzudenkenden Kanten bestehen immer **zwischen** einem **Knoten** und den **darauffolgenden Knoten**, die um eine Ebene **eingerrückt** sind, bis vor dem nächsten Knoten mit der selben **Einrückung**. Diese Bäume **wachsen** von **links-nach-rechts**, von der **Wurzel** zu den **Blättern**. In Abbildung 1.6 ist zur Veranschaulichung an einem **Beispielbaum** dargestellt, wie dieser in der Darstellung des **PicoC-Compilers** aussieht.

¹⁶Und zwischen **gleichen Datenstrukturen** ist es einfacher ineinander umzuformen.

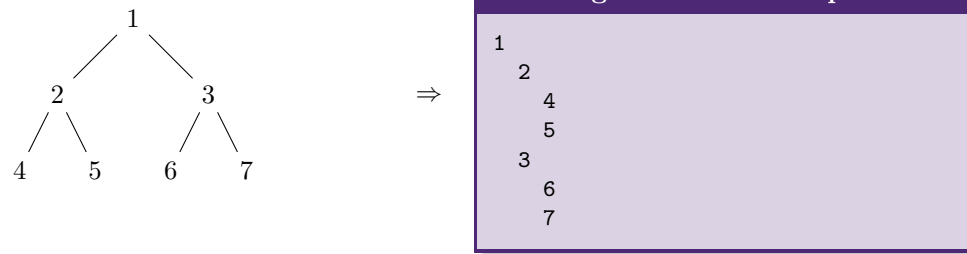


Abbildung 1.6: Veranschaulichung der Darstellung eines Baumes beim PicoC-Compilers.

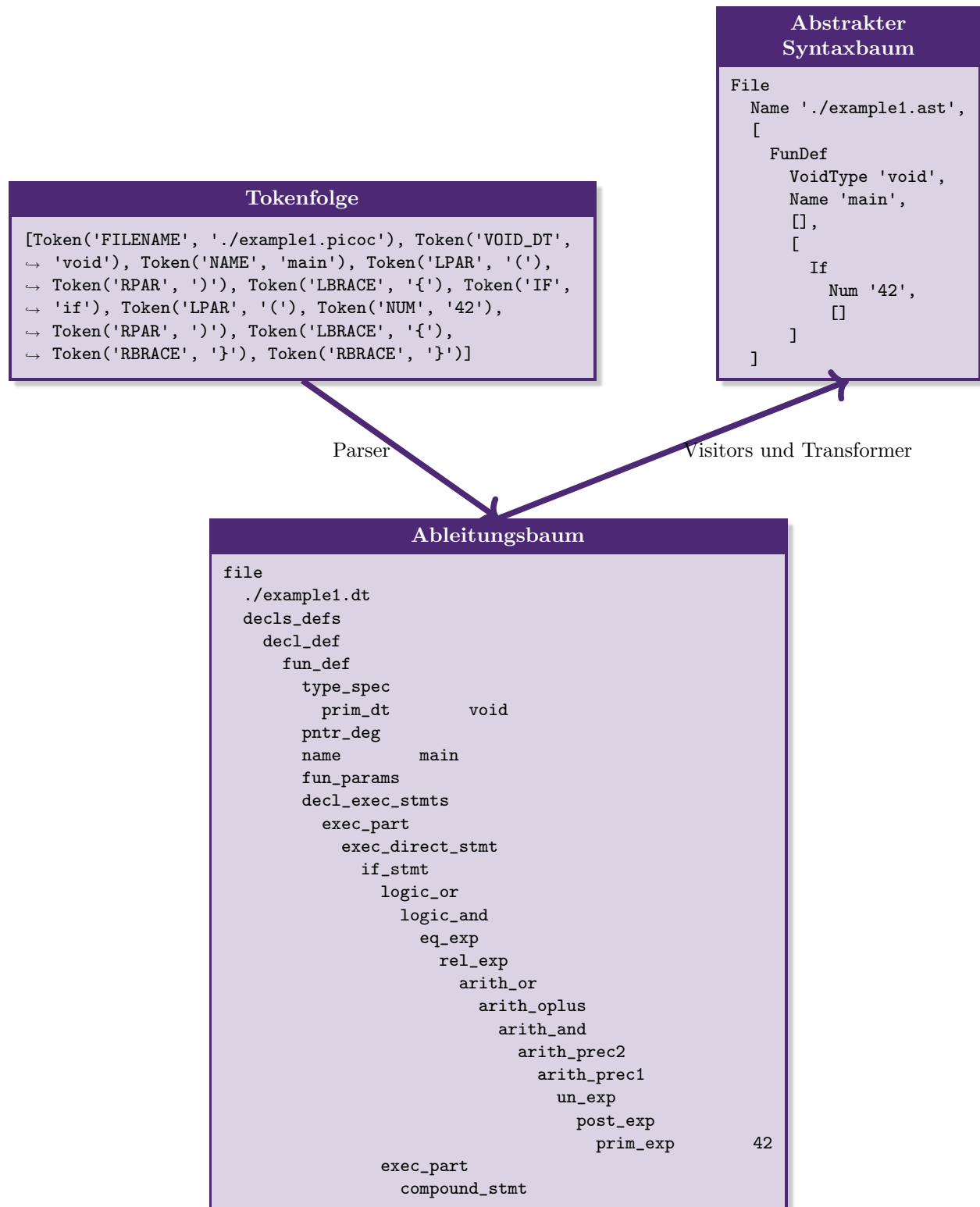


Abbildung 1.7: Veranschaulichung der Syntaktischen Analyse.

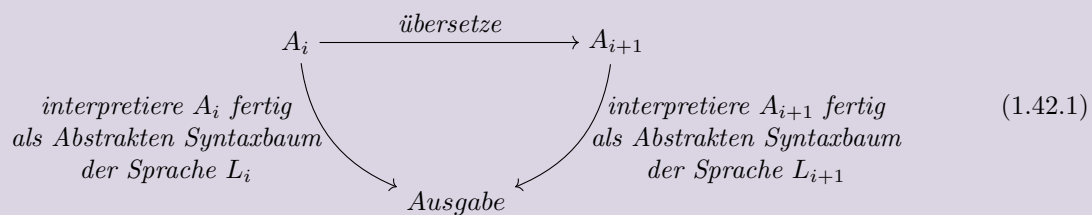
1.5 Code Generierung

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 1.42) nennt. So wie es auch schon mit dem **Ableitungsbaum** in der Syntaktischen Analyse gemacht wurde, den man als Zwischenstufe zum **Abstrakten Syntaxbaum** konstruiert hatte. Aus dem Ableitungsbaum konnte dann unkompliziert und einfach mit **Transformern** und **Visitors** ein **Abstrakter Syntaxbaum** generiert werden.

Definition 1.42: Pass

*Einzelner Übersetzungsschritt in einem Kompilervorgang 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 1.42.1, dass:*



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

^aEin **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.

^bDer 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.

^dG. Siek, *Essentials of Compilation*.

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 1.43) und **Anweisung** (Definition 1.44) wichtig.

Definition 1.43: 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 **Konstanten**, **Variablen**, **Funktionsaufrufen**, **Operatoren** usw.^{a,b}

^aEin **Ausdruck** ist z.B. `21 * 2`;

^bG. Siek, *Essentials of Compilation*.

Definition 1.44: Anweisung (bzw. engl. Statement)

Code, der eine **Vorschrift** 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}

^aEine **Anweisung** ist z.B. `int var = 21 * 2`;

^bG. Siek, *Essentials of Compilation*.

Hat man es mit einer Programmiersprache zu tun, deren Programme **Unreine Anweisungen** (Definition 1.46) besitzen, so ist es sinnvoll einen **Pass** einzuführen, der **Unreine Ausdrücke** von den Anweisungen **trennt**, damit diese zu **Reinen Anweisungen** (Definition 1.45) 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 1.45: 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 **Kompilervorgang** ändert^a.

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

^aZ.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**.

^bG. Siek, *Essentials of Compilation*.

Definition 1.46: 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

^aG. Siek, *Essentials of Compilation*.

Auf diese Weise sind alle **Anweisungen** in **Monadischer Normalform** (Definition 1.47).

Definition 1.47: 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.

^aG. Siek, *Essentials of Compilation*.

Ein **Beispiel** für das Vorgehen, Code in die **Monadische Normalform** zu bringen, ist in Abbildung 1.8 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser Schriftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.8 sind daher in **Konkreter Syntax**¹⁷ aufgeschrieben.

Links in der Abbildung 1.8 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 1.8 dargestellt ist¹⁸.



Abbildung 1.8: Codebeispiel dafür Code in die Monadische Normalform zu bringen.

Die Aufgabe eines solchen **Passes** ist es, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** anzunähern, indem Subbäume vorangestellt werden, die keine Entsprechung in **Maschinenbefehlen** haben. Somit wird eine **Separation** 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 1.43) und **Anweisung** (Definition 1.44) wichtig.

Eine Programmiersprache L_1 soll in eine Maschinsprache 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 **Maschinsprache** handelt, ist es fast unerlässlich einen **Pass** einzuführen, der **Komplexe Ausdrücke** (Definition 1.50) in

¹⁷Für deren Kompilierung die **Abstrakte Syntax** überhaupt definiert wurde.

¹⁸Obwohl 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 Schriftliche 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** (Definition 1.48) zugewiesen werden und dann anstelle des **Komplexen Ausdrucks** auf die jeweilige **temporäre Location** zugegriffen wird.

Definition 1.48: 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

^aG. Siek, *Essentials of Compilation*.

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** (Definition 1.49) 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 1.49: Atomarer Ausdruck

*Ein **Atomarer Ausdruck** ist ein **Reiner Ausdruck** (Definition 1.45), der **keinem kompletten Maschinenbefehl** entspricht, sondern nur ein **Argument**, wie z.B. einen **Immediate** in einer Folge von Maschinenbefehlen festlegt.*^a

^aG. Siek, *Essentials of Compilation*.

Bei einem **üblichen Compiler**, bei dem z.B. **Register** für **temporäre Zwischenergebnisse** genutzt werden und der **Maschinenbefehlssatz** es erlaubt **zwei Register** miteinander zu verrechnen¹⁹ 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** verrechnet werden können^{20 21}.

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 verrechnen²². Aus diesem Grund braucht es mindestens einen Maschinenbefehl²³, um z.B. eine Zahl überhaupt für einen **Maschinenbefehl** zugreifbar zu machen, was der Definition 1.49 widerspricht. Daher sind z.B. Zahlen beim **PicoC-Compiler** keine **Atomaren Ausdrücke**.

¹⁹Z.B. **Addieren** oder **Subtraktion** von zwei **Registerinhalten**.

²⁰Mit dem **RETI-Befehlssatz** wäre das durchaus möglich, durch z.B. `MULT ACC IN2`.

²¹Werden allerdings keine **Register** für **Zwischenergebnisse** genutzt werden, braucht man **mehrere Maschinenbefehle**, um die Zwischenergebnisse auf den **Stack** zu speichern und ein **Stackpointer Register** anzupassen.

²²Der **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 C. Scholl, „Betriebssysteme“ gehalten werden.

²³Genauer gesagt 4.

Definition 1.50: Komplexer Ausdruck

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

^aG. Siek, *Essentials of Compilation*.

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 1.50 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 1.9 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.

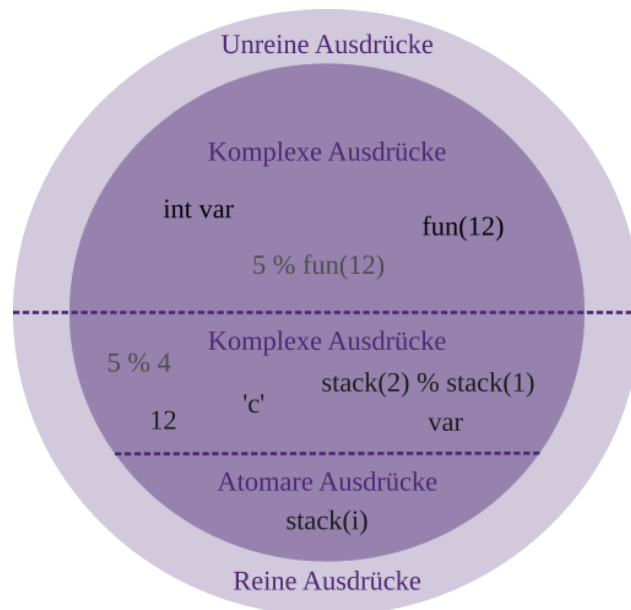


Abbildung 1.9: Übersicht über Komplexe, Atomare, Unreine und Reine Ausdrücke.

Es wird in dem gerade beschriebenen **Pass** dafür gesorgt, dass alle **Anweisungen** und **Ausdrücke** in **A-Normalform**²⁴ (Definition 1.51) sind. Wenn eine **Konkrete Grammatik** in **A-Normalform** ist, ist diese per Definition 1.51 auch automatisch in **Monadischer Normalform**, genauso, wie ein **Atomarer Ausdruck** nach Definition 1.49 auch ein **Reiner Ausdruck** ist.

Definition 1.51: 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 1.47) ist und wenn alle **Komplexen Ausdrücke** nur **Atomare Ausdrücke** enthalten,

²⁴Das '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.^{abc}

^aA-Normalization: Why and How (with code).

^bBolingbroke und Peyton Jones, „Types are calling conventions“.

^cG. Siek, *Essentials of Compilation*.

Ein **Beispiel** für dieses Vorgehen, Code in die **A-Normalform** zu bringen, ist in Abbildung 1.10 zu sehen. Der Einfachheit halber wurde auf die Darstellung in **Abstrakter Syntax** verzichtet, welche allerdings zum großen Teil in dieser Schriftlichen Ausarbeitung der Bachelorarbeit verwendet wird. Die Codebeispiele in Abbildung 1.8 sind daher in **Konkreten Syntax**²⁵ aufgeschrieben.

Um **konsistent** mit der Implementierung zu sein und später keine Verwirrung zu erzeugen, wird beim Beispiel in Abbildung 1.10 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**.

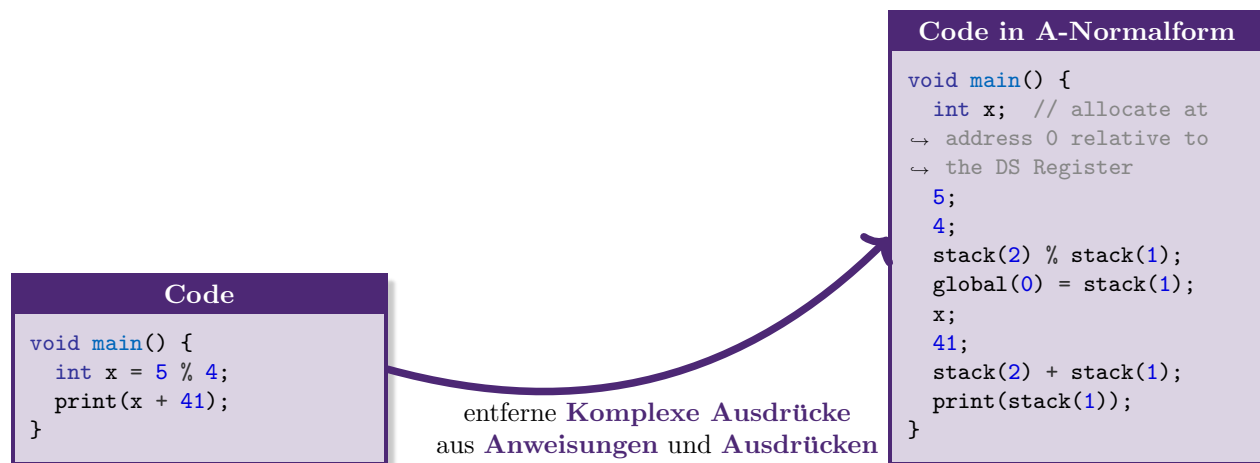


Abbildung 1.10: Codebeispiel für das Entfernen komplexer Ausdrücke aus Operationen.

Ein **Pass**, wie er gerade beschrieben wurde hat vor allem in erster Linie die Aufgabe, den **Abstrakten Syntaxbaum** der **Syntax** von **Maschinenbefehlen** besonders dadurch anzunähern, 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.

²⁵Für 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** gewählt²⁶. 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 **zurückgeben**.

1.6 Fehlermeldungen

Wenn bei einem Compiler ein **unerwünschtes Verhalten** der folgenden **Kategorien**²⁷ 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**²⁸ entscheidet das **Wortproblem** (Definition 1.20) für ein **Eingabeprogramm**²⁹ 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.:

²⁶Da der **PicoC-Compiler** vor allem zu **Lernzwecken** konzipiert ist, wird bei diesem der **Maschinencode** allerdings in einer **menschenlesbaren Repräsentation** ausgegeben

²⁷*Errors in C/C++ - GeeksforGeeks.*

²⁸Bzw. der **Erkennung** innerhalb des Parsers.

²⁹Bzw. 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 bereitgestellt.

wird eine **Fehlermeldung** (Definition 1.52) ausgegeben.

Definition 1.52: 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**.

^aDieses Programm kann z.B. ein **Compiler** sein oder ein **Programm**, dass dieser **Compiler selbst kompiliert** hat.

^bIn engl. **Graphical User Interface**, kurz **GUI**.

^cIn engl. **Text-based User Interface**, kurz **TUI**.

^dIn engl. **log file**.

2 Ergebnisse und Ausblick

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

2.1 Funktionsumfang

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

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

Die kurzen **Anleitungen** in dieser **Schriftlichen Ausarbeitung** der Bachelorarbeit sollen nur zu einem **schnellen, grundlegenden Verständnis** der Verwendung des **PicoC-Compilers** und seiner **Kommandozeilenoptionen** und **Befehle** beihelfen, sowie zum Verständnis der **weiteren implementierten Tools**. Alle weiteren **Kommandozeilenoptionen** und **Befehle** sind für die Verwendung des PicoC-Compilers **unwichtig** und erweisen sich nur in **speziellen Situationen** als nützlich, weshalb für diese auf die **ausführlichere Dokumentation** unter [Link¹](https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt) verwiesen wird.

2.1.1 Kommandozeilenoptionen

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

¹https://github.com/matthejue/PicoC-Compiler/blob/new_architecture/doc/help-page.txt

Basisname der Datei `program` an eine neue **Dateiendung** `.ret` angehängt wird².

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

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

Kommandozeilen-option	Beschreibung	Standard-wert
<code>-i, --intermediate_stages</code>	Gibt Zwischenschritte der Kompilierung in Form der verschiedenen Tokens , Ableitungsbäume , Abstrakten Syntaxbäume der verschiedenen Passes in Dateien mit entsprechenden Dateiendungen aber gleichem Basinamen aus. Im Shell-Mode erfolgt keine Ausgabe in Dateien, sondern nur im Terminal .	false , most_used: true
<code>-p, --print</code>	Gibt alle Dateiausgaben auch im Terminal aus. Diese Option ist im Shell-Mode dauerhaft aktiviert.	false (true im Shell-Mode und für den most_used-Befehl)
<code>-v, --verbose</code>	Fügt den verschiedenen Zwischenschritten der Kompilierung , unter anderem auch dem finalen RETI-Code Kommentare hinzu, welche eine Anweisung oder einen Befehl aus einem vorherigen Pass beinhalten, der durch die darunterliegenden Anweisungen oder Befehle ersetzt wurde. Wenn die <code>--run</code> -Option aktiviert ist, wird der Zustand der virtuellen RETI-CPU vor und nach jedem Befehl angezeigt.	false
<code>-vv, --double_verbose</code>	Hat dieselben Effekte , wie die <code>--verbose</code> -Option, aber bewirkt zusätzlich weitere Effekte . PicoC-Knoten erhalten bei der Ausgabe in den Abstrakten Syntaxbäumen zusätzliche runde Klammern , sodass direkter abgelesen werden kann, wo ein Knoten anfängt und wo einer aufhört. In Fehlermeldungen werden mehr Tokens angezeigt, die an der Stelle der Fehlermeldung erwartet worden wären. Bei Aktivierung der <code>--intermediate_stages</code> -Option werden in den dadurch ausgegebenen Abstrakten Syntaxbäumen ebenfalls versteckte Attribute, die Informationen zu Datentypen und für Fehlermeldungen beinhalten angezeigt.	false
<code>-h, --help</code>	Zeigt die Dokumentation , welche ebenfalls unter Link gefunden werden kann im Terminal an. Mit der <code>--color</code> -Option kann die Dokumentation mit farblicher Hervorhebung im Terminal angezeigt werden.	false
<code>-l</code>	Es lässt sich einstellen, wieviele Zeilen rund um die Stelle an welcher ein Fehler aufgetreten ist angezeigt werden sollen.	2
<code>-c</code>	Aktiviert farbige Ausgabe .	false , most_used: true
<code>-t, --thesis</code>	Filtert für die Codebeispiele in dieser Schriftlichen Ausarbeitung der Bachelorarbeit bestimmte Kommentare in den Abstrakten Syntaxbäumen heraus, damit alles übersichtlich bleibt.	false

Tabelle 2.1: Kommandozeilenoptionen, Teil 1.

Kommandozeilen-option	Beschreibung	Standard-wert
-R, --run	Führt die RETI-Befehle , die das Ergebnis der Kompilierung sind mit einer virtuellen RETI-CPU aus. Wenn die --intermediate_stages -Option aktiviert ist, wird eine Datei <basename>.reti_states erstellt, welche den Zustand der RETI-CPU nach dem letzten ausgeführten RETI-Befehl enthält. Wenn die --verbose - oder --double_verbose -Option aktiviert ist, wird der Zustand der RETI-CPU vor und nach jedem Befehl auch noch zusätzlich in die Datei <basename>.reti_states ausgegeben.	false , most_used: true
-B, --process_begin	Setzt die relative Adresse , wo der Prozess bzw. das Codesegment für das ausgeführte Programm beginnt.	3
-D, --datasegment_size	Setzt die Größe des Datensegments . Diese Option muss mit Vorsicht gesetzt werden, denn wenn der Wert zu niedrig gesetzt wird, dann können die Globalen Statischen Daten und der Stack miteinander kollidieren.	32

Tabelle 2.2: Kommandozeilenoptionen, Teil 2.

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

2.1.2 Shell-Mode

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

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

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


```

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

Compilation successfull

PicoC> quit

```

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

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






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


```

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

```

Code 2.2: Shell-Mode und der Befehl *most_used*.

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

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

2.1.3 Show-Mode

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

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

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

Der **Show-Mode** kann auf die **einfachste Weise** mittels der `/Makefile` des **PicoC-Compilers** mit dem Befehl `make show FILEPATH=<path-to-file> <more-options>` gestartet werden. Alle **einstellbaren Optionen**, die z.B. unter `<more-options>` noch für die **Makefile** gesetzt werden können sind in Tabelle 2.3 aufgelistet.

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

Tabelle 2.3: Makefileoptionen.

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

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

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

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

³Home - Neovim.

Index: 84	00019 ADD ACC CS;	00057 LOADIN SP ACC 2;	00095 MOVE BAF ACC;	00133 0
Instruction: STOREIN SP ACC 2;	00020 STOREIN BAF ACC -1;	00058 LOADIN SP IN2 1;	00096 ADDI SP 3;	00134 0
ACC: 42	00021 JUMP 44;	00059 ADD ACC IN2;	00097 MOVE SP BAF;	00135 0
ACC_SIMPLE: 42	00022 MOVE BAF IN1;	00060 STOREIN SP ACC 2;	00098 SUBI SP 4;	00136 0
IN1: 0	00023 LOADIN IN1 BAF 0;	00061 ADDI SP 1; <- PC	00099 STOREIN BAF ACC 0;	00137 0
IN1_SIMPLE: 0	00024 MOVE IN1 SP;	00062 LOADIN SP ACC 1;	00100 LOADI ACC 101;	00138 0 <- SP
IN2: 2	00025 SUBI SP 1;	00063 ADDI SP 1;	00101 ADD ACC CS;	00139 2
IN2_SIMPLE: 2	00026 STOREIN SP ACC 1;	00064 LOADIN BAF PC -1;	00102 STOREIN BAF ACC -1;	00140 42
PC: 2147483709	00027 LOADIN SP ACC 1;	00065 SUBI SP 1;	00103 JUMP -58;	00141 2
PC_SIMPLE: 61	00028 STOREIN DS ACC 1;	00066 LOADI ACC 2;	00104 MOVE BAF IN1;	00142 40
SP: 2147483786	00029 ADDI SP 1;	00067 STOREIN SP ACC 1;	00105 LOADIN IN1 BAF 0;	00143 2147483752
SP_SIMPLE: 138	00030 SUBI SP 1;	00068 LOADIN SP ACC 1;	00106 MOVE IN1 SP;	00144 2147483797 <- BAF
BAF: 2147483792	00031 LOADIN DS ACC 1;	00069 STOREIN BAF ACC -3;	00107 SUBI SP 1;	00145 40
BAF_SIMPLE: 144	00032 STOREIN SP ACC 1;	00070 ADDI SP 1;	00108 STOREIN SP ACC 1;	00146 2
CS: 2147483651	00033 SUBI SP 1;	00071 SUBI SP 1;	00109 LOADIN SP ACC 1;	00147 38
CS_SIMPLE: 3	00034 LOADI ACC 2;	00072 LOADIN BAF ACC -2;	00110 ADDI SP 1;	00148 2147483670
DS: 2147483766	00035 STOREIN SP ACC 1;	00073 STOREIN SP ACC 1;	00111 CALL PRINT ACC;	00149 2147483650
DS_SIMPLE: 118	00036 LOADIN SP ACC 2;	00074 SUBI SP 1;	00112 SUBI SP 1;	UART:
SRAM:	00037 LOADIN SP IN2 1;	00075 LOADIN BAF ACC -3;	00113 LOADIN BAF ACC -4;	00000 0
00000 JUMP 0;	00038 ADD ACC IN2;	00076 STOREIN SP ACC 1;	00114 STOREIN SP ACC 1;	00001 0
00001 2147483648	00039 STOREIN SP ACC 2;	00077 LOADIN SP ACC 2;	00115 LOADIN SP ACC 1;	00002 0
00002 0	00040 ADDI SP 1;	00078 LOADIN SP IN2 1;	00116 ADDI SP 1;	00003 0
00003 CALL INPUT ACC; <- CS	00041 LOADIN SP ACC 1;	00079 ADD ACC IN2;	00117 LOADIN BAF PC -1;	EPROM:
00004 SUBI SP 1;	00042 ADDI SP 1;	00080 STOREIN SP ACC 2;	00118 38 <- DS	00000 LOADI DS -2097152; <- IN1
00005 STOREIN SP ACC 1;	00043 CALL PRINT ACC;	00081 ADDI SP 1;	00119 0	00001 MULTI DS 1024;
00006 LOADIN DS ACC 1;	00044 LOADIN BAF PC -1;	00082 LOADIN SP ACC 1;	00120 0	00002 MOVE DS SP; <- IN2
00007 STOREIN DS ACC 0;	00045 SUBI SP 1;	00083 STOREIN BAF ACC -4;	00121 0	00003 MOVE DS BAF;
00008 ADDI SP 1;	00046 LOADI ACC 2;	00084 ADDI SP 1;	00122 0	00004 MOVE DS CS;
00009 SUBI SP 2;	00047 STOREIN SP ACC 1;	00085 SUBI SP 1;	00123 0	00005 ADDI SP 149;
00010 SUBI SP 1;	00048 LOADIN SP ACC 1;	00086 LOADIN BAF ACC -4;	00124 0	00006 ADDI BAF 2;
00011 LOADIN DS ACC 0;	00049 STOREIN BAF ACC -3;	00087 STOREIN SP ACC 1;	00125 0	00007 ADDI CS 3;
00012 STOREIN SP ACC 1;	00050 ADDI SP 1;	00088 LOADIN SP ACC 1;	00126 0	00008 ADDI DS 118;
00013 MOVE BAF ACC;	00051 SUBI SP 1;	00089 ADDI SP 1;	00127 0	00009 MOVE CS PC;
00014 ADDI SP 3;	00052 LOADIN BAF ACC -2;	00090 CALL PRINT ACC;	00128 0	
00015 MOVE SP BAF;	00053 STOREIN SP ACC 1;	00091 SUBI SP 2;	00129 0	Index: 85
00016 SUBI SP 5;	00054 SUBI SP 1;	00092 SUBI SP 1;	00130 0	Instruction: ADDI SP 1;
00017 STOREIN BAF ACC 0;	00055 LOADIN BAF ACC -3;	00093 LOADIN BAF ACC -4;	00131 0	ACC: 42
00018 LOADI ACC 19;	00056 STOREIN SP ACC 1;	00094 STOREIN SP ACC 1;	00132 0	ACC_SIMPLE: 42

Abbildung 2.1: Show-Mode in der Verwendung.

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

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

2.2 Qualitätssicherung

Um verifizieren zu können, dass der **PicoC-Compiler** sich genauso verhält, wie er soll, müssen die **Beziehungen** aus Diagramm 1.3.1 in Unterkapitel 1.1 genauso für den **PicoC-Compiler** gelten. Für den **PicoC-Compiler** lässt sich ein ebensolches Diagramm 2.0.1 definieren. Ein **beliebiges** Testprogramm P_{PicoC} in der Sprache L_{PicoC} muss die **gleiche Semantik** haben, wie das entsprechend **kompilierte** Programm P_{RETI} in der Sprache L_{RETI} , trotz der **unterschiedlichen Syntax**.

Die **Tests** für den **PicoC-Compiler** sind hierbei im Verzeichnis **/tests** bzw. unter **Link⁴** zu finden. **Eingeteilt** sind die Tests in die folgenden **Kategorien** in Tabelle 2.4.

⁴https://github.com/matthejue/PicoC-Compiler/tree/new_architecture/tests.

Testkategorie	Beschreibung
basic	Einfache Tests, welche die grundlegenden Funktionalitäten des Compilers testen.
advanced	Tests, die Spezialfälle und Kombinationen verschiedener Funktionalitäten des Compilers testen.
hard	Tests, die längere, komplexe Programme testen, für welche die Funktionalitäten des Compilers in perfekter Harmonie miteinander funktionieren müssen.
example	Tests, die bekannte Algorithmen darstellen und daher als gutes, repräsentatives Beispiel für die Funktionsfähigkeit des PicoC-Compilers dienen.
error	Tests, die Fehlermeldungen testen. Für diese Tests wird keine Verifikation ausgeführt.
exclude	Tests, für welche aufgrund vielfältiger Gründe keine Verifikation ausgeführt werden soll.
thesis	Tests, die eigentlich vorher Codebeispiele für diese Schriftliche Ausarbeitung der Bachelorarbeit waren.
tobias	Tests, die der Betreuer dieser Bachelorarbeit, Tobias geschrieben hat.

Tabelle 2.4: Testkategorien.

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

Die Kante vom Testprogramm P_{PicoC} zur Ausgabe aus Diagramm 2.0.1 drückt aus, dass jeder Test im `/tests`-Verzeichnis eine `// expected:<space_seperated_output>`-Zeile hat, in welcher der **Schreiber des Tests** die Rolle des entsprechenden **Interpreters**⁵ aus Diagramm 1.3.1 übernimmt und die **erwartete Ausgabe** seiner eigenen Interpretation des **PicoC-Codes** anstelle von `<space_seperated_output>` hineinschreibt.

Ein Beispiel für einen **Test** ist in Code 2.3 zu sehen. Sobald die Tests mithilfe des Bashscripts `/run_tests.sh` ausgeführt werden oder dieses mithilfe der `/Makefile` mit dem Befehl `> make test` ausgeführt wird, wird als erstes für **jeden** Test das Bashscript `/extract_input_and_expected.sh` ausgeführt, welches die Zeilen `// in:<space_seperated_input>`, `// expected:<space_seperated_output>` und `// datasegment:<datasegment_size>` extrahiert⁶ und die entsprechenden Werte in **neu** erstellte Dateien `<program>.in`, `<program>.out_expected` und `<program>.datasegment_size` schreibt. Das **letzte Skript** kann ebenfalls mit dem Befehl `> make extract` ausgeführt werden.

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

⁵Der die **Semantik** des Tests umsetzt.

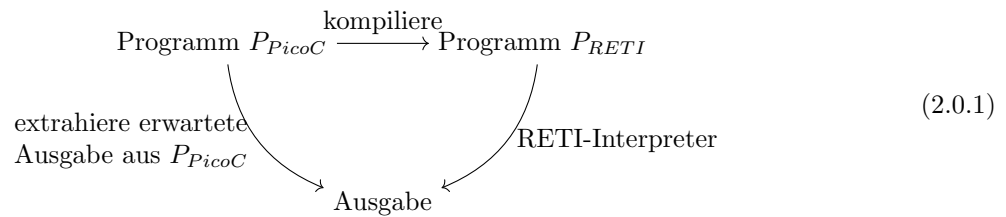
⁶Falls vorhanden.

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

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

Code 2.3: Typischer Test.

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

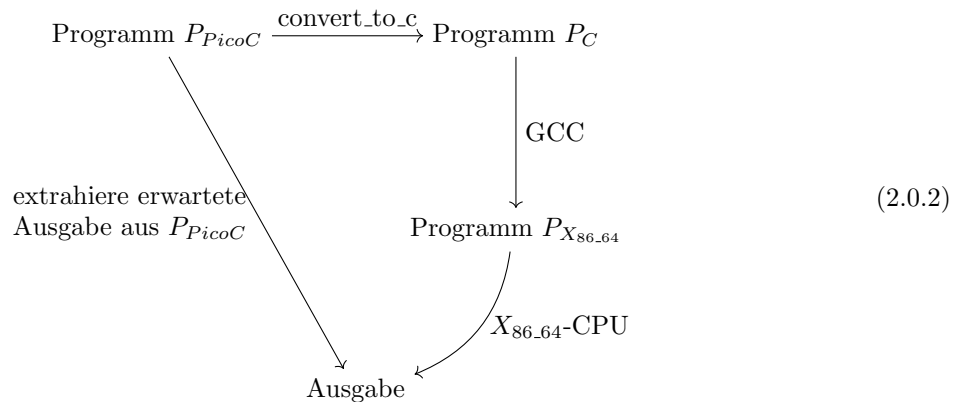


Allerdings gibt es bei dem Testverfahren, welches in Diagramm 2.0.1 dargestellt ist ein **Problem**, denn der **Schreiber** der Tests ist in diesem Fall die **gleiche Person**, die auch den **Compiler implementiert**. Wenn der **Schreiber** der Tests ein **falsches Verständnis** davon hat, wie das Ergebnis eines Ausdrucks berechnet wird, so wird dieser sowohl im **Test** als auch in seiner **Implementierung** etwas als Ergebnis erwarten bzw. etwas implementieren, was nicht der eigentlichen **Semantik** von L_{PicoC} entspricht⁷.

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

⁷Welche ja identisch zu der von L_C sein sollte.

⁸Die eine **Untermenge** von L_C ist.



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

Der **Trick** liegt hierbei in der Verwendung des **GCC** für die Kante von P_C zu P_{X86_64} . Beim **GCC** handelt es sich um einen Compiler der Sprache L_C , der somit auch mit Ausnahme der `print()` und `input()`-Funktionen auch die Sprache L_{PicoC} kompilieren kann. Der **GCC** setzt aufgrund seiner bekanntermaßen **vielfachen Verwendung** auf der Welt und seinem **sehr langem Bestehen** seit 1987⁹ ¹⁰ die **Semantik** der Sprache L_C , vor allem für die kleine Untermenge, welche L_{PicoC} darstellt mit sehr hoher Wahrscheinlichkeit **korrekt** um.

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

Dazu durchläuft jeder Test, wie in Diagramm 2.0.2 dargestellt ist eine **Verifikation**, in der **verifiziert** wird, ob bei der Kompilierung des Testprogramms P_C mit dem **GCC** und Ausführung des hieraus generierten X_{86_64} -Maschinencodes die Ausgabe **identisch** zur erwarteten Ausgabe // `expected:<space_seperated_output>` des Testschreibers ist. Erst dann ist ein Test **verifiziert**, d.h. man kann, wenn der Test **vernünftig definiert** ist mit **hoher Wahrscheinlichkeit** sagen¹¹, dass wenn dieser Test für den **PicoC-Compiler** durchläuft, der **Teil der Semantik** der Sprache L_{PicoC} , den dieser Test testet vom PicoC-Compiler **korrekt umgesetzt** ist.

Für diese **Verifikation** ist das Bashscript `/verify_tests.sh` verantwortlich, welches mithilfe der `/Makefile` mit dem Befehl `> make verify` ausgeführt wird. Beim Befehl `> make test` wird dieses Bashscript **vor** dem eigentlichen Testen¹² durchgeführt. In Code 2.4 ist ein Testdurchlauf mit `> make test` zu sehen. Wobei **Verified: 50/50** anzeigt, wieviele der Tests **verifizierbar** sind¹³, also beim **GCC** ohne Fehlermeldung durchlaufen, **Not verified:** die **nicht verifizierbaren** Tests angibt, **Running through: 88 / 88** anzeigt wieviele Tests mit dem **PicoC-Compiler** durchlaufen, **Not running through:** die **nicht** durchlaufenden Tests angibt, **Passed: 88 / 88** zeigt bei wievielen Tests die Ausgabe mit der erwarteten Ausgabe **identisch** ist, **Not passed:** die Tests anzeigt, bei denen das **nicht** der Fall ist.

⁹History - GCC Wiki.

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

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

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

¹³Also **alle** Tests aus den **Kategorien basic, advanced, hard** und **example**.


```

> make test
=====
= ./tests/basic_array_init.picoc =
=====
...
=====
=          Verification          =
=====
./tests/basic_array_init.c
...
=====
=          Results              =
=====
Verified: 80 / 80
Not verified:
Running through: 118 / 118
Not running through:
Passed: 118 / 118
Not passed:

```

Code 2.4: Testdurchlauf.

Der Befehl `make test <more-options>` lässt sich ebenfalls mit den **Makefileoptionen** `<more-options>` TESTNAME, VERBOSE und DEBUG aus Tabelle 2.3 kombinieren.

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

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

2.3 Erweiterungsideen

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

- **Register Allokation:** Variablen werden nicht nur **Adressen** im **Hauptspeicher** zugewiesen, sondern an erster Stelle **Registern** und erst wenn alle Register **voll** sind werden Variablen an Adressen auf dem **Hauptspeicher** gespeichert. Da hat den Grund, dass der **Zugriff auf Register** deutlich **schneller** ist, als der **Zugriff auf den Hauptspeicher**. Um die Variablen möglichst optimal **Locations** (Definition 1.48) zuzuweisen wird mithilfe einer **Liveness Analyse** (Definition ??) ein **Interferenzgraph**

¹⁴Möglicherweise ja im Rahmen eines **Masterprojektes** 😊.

(Definition ??) aufgebaut. Auf den **Interferenzgraph** wird ein **Graph Coloring** Algorithmus (Definition ??) angewandt, der den **Locations** Zahlen zuordnet. Die **ersten** Zahlen entsprechen **Registern**, aber ab einem bestimmten Zahlenwert, wenn alle Register zugeordnet sind, entsprechen die Zahlen **Adressen auf dem Hauptspeicher**. Des Weiteren muss die **Liveness Analyse** nach Ansätzen der **Kontrollflussanalyse** (Definition ??) **iterativ** unter Verwendung eines **Kontrollflussgraphen** (Definition ??) auf die verschiedenen **Blöcke** angewendet werden, bis sich an den Live Variablen **nichts** mehr **ändert**.¹⁵

- **Tail Call:** Wenn ein Funktionsaufruf die **letzte** Anweisung in einem Funktionsblock ist, wird der Stackframe dieser aufrufenden Funktion **nicht** mehr **gebraucht**, da **nicht** mehr in diese Funktion zurückgekehrt werden muss¹⁶. Daher kann der **Stackframe** der aufrufenden Funktion **entfernt** werden, **bevor** der **Funktionsaufruf** getätigt wird. Der **Vorteil** ist, dass eine rekursive Funktion, die nur Tail Calls ausführt nur eine **konstante Menge** an **Speicherplatz** auf dem Stack verbraucht. In Code 2.5 sind **zwei Tail Calls** markiert.
- **Partielle Evaluation:** Bei Ausdrücken wie `4 + input() - 2`, `input() * 1` oder `0 + input() * 2` können **Teilausdrücke** bereits **während** des **Kompilierens partiell** zu `2 + input()`, `input()` und `input() * 2` berechnet werden. Dies kann durch einen neuen **PicoC-Eval Pass** umgesetzt werden, der **vor** oder **nach** dem **PicoC-Shrink Pass** den Abstrakten Syntaxbaum in eine neue Abstrakte Syntax der Sprache *LPicoC-Eval* umformt. In der Abstrakten Syntax der Sprache *LPicoC-Eval* sind **binäre Operationen** zwischen zwei `Num(str)`-PicoC-Knoten **nicht möglich**. Diese **partielle Vorberechnung** kann auch auf **Konstanten** und **Variablen** ausgeweitet werden. Der **Vorteil** ist, dass hierdurch weniger **RETI-Code** produziert wird und weniger **RETI-Code** bedeutet wiederum eine **schnellere Programmausführung**.
- **Lazy Evaluation:** Bei Ausdrücken wie `var1 && 42 / 0` oder `var2 || 42 / 0`, wobei `var1 = 0` und `var2 = 1` müssen diese Ausdrücke nur **soweit** berechnet werden, wie es **benötigt** wird. Sobald bei einer Aneinanderreihung von `&&`-Operationen einmal eine 0 auftaucht, muss der Rest des Ausdrucks **nicht** mehr berechnet werden, da mit dem Auftauchen der 0 bereits klar ist, dass dieser Ausdruck sich zu 0 auswertet. Genauso für eine Aneinanderreihung von `||`-Operationen und dem Auftauchen einer 1. Daher kommt es aufgrund der Division durch 0 nicht zu einer **DivisionByZero-Fehlermeldung**, da die Ausdrücke garnicht so weit ausgewertet werden. Im Unterschied zur **Partiellen Evaluation** läuft **Lazy Evaluation**¹⁷ zur **Laufzeit** ab.
- **Objektorientierung:** Wie in der Programmiersprache *LC++* müssen **Klassen** und `new`-, `new[]`-, `delete`-, `delete[]`- und `::`-Operatoren eingeführt werden. Die Speicherung eines **Objekts** ist ähnlich wie bei **Verbunden**.
- **Mehrere Dateien:** **Funktionen** werden zusammen mit **Attributen** in **mehrere Dateien** aufgeteilt, welche **seperat** programmiert und kompiliert werden können. Für die **Deklaration** von **Funktionen** und **Attributen** werden **.h-Headerdateien** verwendet, für die Definition sind **.c-Quellcodedateien** da. Hierbei ist der **Basisname** einer **.h-Headerdatei identisch** zur entsprechenden **.c-Quellcodedatei** mit den entsprechenden Definitionen. Dateien werden über `#include "file"` eingebunden, was einem **direkten einfügen** des entsprechenden Codes der eingebundenen Datei entspricht. Über einen **Linker** (Definition ??) können die **kompilierten .o-Objektdaten** (Definition ??) zusammengefügt werden, wobei der **Linker** darauf achtet **keinen doppelten Code** zuzulassen.
- **malloc und free:** Es wird eine **Bibltiothek** mit den Funktionen `malloc` und `free`, wie in der Bibliothek `stdlib`¹⁸ implementiert, deren **.h-Headerdatei** mittels `#include "malloc_and_free.h"` eingebunden wer-

¹⁵Die in diesem **Unterpunkt** erwähnten **Begriffe** werden nur **grob** erläutert, da sie für den **PicoC-Compiler** keine Rolle spielen. Aber sie wurden erwähnt, damit in dieser **Bachelorarbeit** auch das übliche Vorgehen Erwähnung findet und vom Vorgehen beim **PicoC-Compiler** abgegrenzt werden kann.

¹⁶Was der Grund ist, warum ein **Stackframe** überhaupt angelegt wird, damit später beim **Rücksprung** aus der **aufgerufenen Funktion** die Ausführung mit allen Variablen, wie **vor der Ausführung** fortgesetzt werden kann.

¹⁷Es gibt hierfür leider keinen **deutschen Begriff**, der geläufig ist.

¹⁸Auch engl. **General Purpose Standard Library** genannt.

den muss. Es braucht eine neue **Kommandozeilenoption** `-l` um dem **Linker** verwendete Bibliotheken mitzuteilen. Aufgrund der Einführung von `malloc` und `free` wird im **Datensegment** der Abschnitt nach den **Globalen Statischen Daten** als **Heap** bezeichnet, der mit dem **Stack** kollidieren kann. Im **Heap** wird von der `malloc`-Funktion **Speicherplatz allokiert** und ein **Zeiger** auf diesen **zurückgegeben**. Dieser **Speicherplatz** kann von der `free`-Funktion wieder **freigegeben** werden. Um zu wissen, wo und wieviel Speicherplatz im **Heap** zur **Allokation** frei ist, muss dies in einer **Datenstruktur** abgespeichert werden.

- **Garbage Collector:** Anstelle der `free`-Funktion kann auch einfach die `malloc`-Funktion direkt so implementiert werden, dass sobald der Speicherplatz auf dem **Heap** knapp wird, Speicherplatz, der sonst unmöglich in der Zukunft mehr genutzt werden würde freigegeben wird. Auf eine sehr einfache Weise lässt sich dies mit dem **Two-Space Copying Collector** (Definition ??) implementieren.
- **stdio.h:** Die Funktionen `print` und `input` werden nicht über den **Trick** einen eigenen **RETI-Befehl** `CALL (PRINT | INPUT) ACC` für den **RETI-Interpreter** zu definieren, der einfach **direkt** das **Ausgeben** und **Eingaben entgegennehmen** übernimmt gelöst, sondern über eine eigene **stdio-Bibliothek** mit `print`- und `input`-Funktionen, welche die **UART** verwenden, um z.B. an einem simpel gehaltenen simulierten **Monitor** Daten zu übertragen, die dieser anzeigt.
- **Feld mit Länge:** Man könnte in einer **Bibliothek** einen eigenen **Felddatentyp**, wie in der Programmiersprache L_{C++} mit dem Datentyp `std::vector` über eine **Klasse** implementieren, der seine **Anzahl Elemente** an den **Anfang** des Felds speichert, sodass über eine **Methode** `size` die **Anzahl Elemente** direkt über die **Variable des Felds** selbst ausgelesen werden kann (z.B. `vec.var.size`) und **nicht** in einer **seperaten Variable** gespeichert werden muss.
- **Maschinencode in binärer Repräsentation:** Maschinencode wird nicht, wie momentan beim **PicoC-Compiler** in **menschenlesbarer Repräsentation** ausgegeben, sondern in **binärer Repräsentation** nach dem **Intruktionsformat**, welches in der Vorlesung C. Scholl, „Betriebssysteme“ festgelegt wurde.
- **PicoPython:** Da das **Lark Parsing Toolkit** verwendet wurde, welches das **Parsen** über eine selbst angegebene **Konkrete Grammatik** übernimmt, könnte mit **relativ geringem Aufwand** ein Konkrete Grammatik definiert werden, die eine zur Programmiersprache L_{Python} **ähnliche Konkrete Syntax** beschreibt. Die **Konkrete Syntax** einer Programmiersprache lässt sich durch Austauschen der Konkreten Grammatik **sehr einfach** ändern, nur die **Semanatik** zu ändern kann **deutlich aufwändiger** sein. Viele der **PicoC-Knoten** könnten für die Programmiersprache $L_{PicoPython}$ **wiederverwendet** werden und viele **Passes** müssten nur erweitert werden.
- **Call by Reference:** Über das wiederverwenden des `&`-Symbols für **Parameter** bei **Funktionsdeklaration** und **Funktionsdefinition**, wie es in der Vorlesung P. Scholl, „Einführung in Embedded Systems“ erklärt wurde.
- **PicoC-Debugger:** Es wird eine neue **Kommandozeilenoption**, z.B. `-g` eingeführt durch welche spezielle **Informationen** in den RETI-Code geschrieben werden, die einem **Debugger** unter anderem mitteilen, wo die **RETI-Befehle** für eine Anweisungen **beginnen** und wo sie **aufhören** usw., damit der **Debugger** weiß, bis wohin er die **RETI-Befehle** ausführen soll, damit er eine Anweisung abgearbeitet hat.
- **Bootstrapping:** Mittels **Bootstrapping** lässt sich der **PicoC-Compiler** unabhängig von der Sprache L_{Python} und der **Maschine**, die das **cross-compile** (Definition 1.6) übernimmt machen. Im Unterkapitel ?? wird genauer hierauf eingegangen. Hierdurch wird der **PicoC-Compiler** zum einem **Compiler** für die **RETI-CPU** gemacht, der auf der RETI-CPU selbst läuft.

```
1 // in:42
2 // expected:0
3
4 int ret0() {
5     return 0;
6 }
7
8 int ret1() {
9     return 1;
10 }
11
12 int tail_call_fun(int bool_val) {
13     if (bool_val) {
14         return ret0();
15     }
16     return ret1();
17 }
18
19 void main() {
20     print(tail_call_fun(input()));
21 }
```

Code 2.5: Beispiel für Tail Call.

Anmerkung 🔍

Partielle Evaluation und Lazy Evaluation wurden im PicoC-Compiler **nicht** implementiert, da dieser als **Lerntool** gedacht ist und diese Funktionalitäten den **RETI-Code** für Studenten **schwerer verständlich** machen könnten, da die **Codeschnipsel** und damit verbundene **Paradigmen** aus der Vorlesung **nicht** mehr so einfach **nachvollzogen** werden können und das **schwerere Ausmachen** können von **Orientierungspunkten** und **Fehlen erwarteter Codeschnipsel** leichter zur **Verwirrung** bei den Studenten führen könnte.

Literatur

Online

- 2.1.7 Vorrangregeln und Assoziativität. URL: <https://www.tu-chemnitz.de/urz/archiv/kursunterlagen/C/kap2/vorrang.htm> (besucht am 05.09.2022).
- A-Normalization: Why and How (with code). URL: <https://matt.might.net/articles/a-normalization/> (besucht am 23.07.2022).
- Errors in C/C++ - GeeksforGeeks. URL: <https://www.geeksforgeeks.org/errors-in-cc/> (besucht am 10.05.2022).
- History - GCC Wiki. URL: <https://gcc.gnu.org/wiki/History> (besucht am 06.08.2022).
- Home - Neovim. URL: <http://neovim.io/> (besucht am 04.08.2022).
- JSON parser - Tutorial — Lark documentation. URL: https://lark-parser.readthedocs.io/en/latest/json_tutorial.html (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. *Essentials of Compilation*. 28. Jan. 2022. URL: <https://iucompilercourse.github.io/IU-Fall-2021/> (besucht am 28.01.2022).
- Nystrom, Robert. *Parsing Expressions · Crafting Interpreters*. Genever Benning, 2021. URL: <https://www.craftinginterpreters.com/parsing-expressions.html> (besucht am 09.07.2022).

Artikel

- Earley, J. und Howard E. Sturgis. „A formalism for translator interactions“. In: *CACM* (1970). DOI: [10.1145/355598.362740](https://doi.org/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).
- Scholl, Philipp. „Einführung in Embedded Systems“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://earth.informatik.uni-freiburg.de/uploads/es-2122/> (besucht am 09.07.2022).
- Thiemann, Peter. „Compilerbau“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2021ws/> (besucht am 09.07.2022).
- — „Einführung in die Programmierung“. Vorlesung. Vorlesung. Universität Freiburg, 2018. URL: <http://proglang.informatik.uni-freiburg.de/teaching/info1/2018/> (besucht am 09.07.2022).
- Westphal, Dr. Bernd. „Softwaretechnik“. Vorlesung. Vorlesung. Universität Freiburg, 2021. URL: <https://swt.informatik.uni-freiburg.de/teaching/SS2021/swtv1> (besucht am 19.07.2022).

Sonstige Quellen

- Bolingbroke, Maximilian C. und Simon L. Peyton Jones. „Types are calling conventions“. In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*. the 2nd ACM SIGPLAN symposium. Edinburgh, Scotland: ACM Press, 2009, S. 1. ISBN: 978-1-60558-508-6. DOI: [10.1145/1596638.1596640](https://doi.org/10.1145/1596638.1596640). URL: <http://portal.acm.org/citation.cfm?doid=1596638.1596640> (besucht am 23.07.2022).
- *Lark - a parsing toolkit for Python*. 26. Apr. 2022. URL: <https://github.com/lark-parser/lark> (besucht am 28.04.2022).