Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionen

1 Motivation

1.1 PicoC und RETI

1.2 Aufgabenstellung

1.3 Eigenheiten der Sprache C

1.4 Richtlinien

2 Einführung

2.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 2.2) und eines **Interpreters** (Definition 2.1), da das Schreiben eines Compilers von der **PicoC-Sprache** in die **RETI-Sprache** das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[1] und von **Tests** die **Beziehungen** in 2.1 zu belegen (siehe Subkapitel 4.3).

Definition 2.1: Interpreter

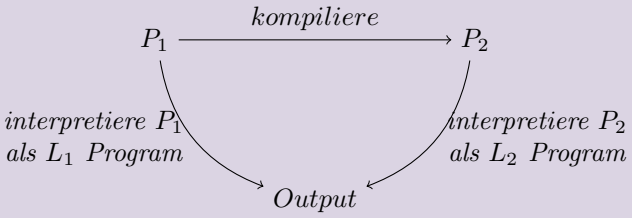
Interpretiert *die* ***Instructions*** *bzw.* ***Statements*** *eines Programmes P direkt.*

Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 2.41) und führt je nach Komposition der* ***Nodes*** *des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*

Definition 2.2: Compiler

Kompiliert *ein Program P_1, welches in einer Sprache L_1 geschrieben ist, in ein Program P_2, welches in einer Sprache L_2 geschrieben ist.*

Wobei ***Kompilieren*** *meint, dass das Program P_1 in das Program P_2 übersetzt und bei beiden Programmen, wenn sie von* ***Interpreter*** *ihrer jeweiligen Sprachen L_1 und L_2* ***interpretiert*** *werden, der gleiche* ***Output*** *rauskommt. Also beide Programme P_1 und P_2 die gleiche* ***Semantik*** *haben und sich nur* ***syntaktisch*** *durch die Sprachen L_1 und L_2, in denen sie geschrieben stehen unterscheiden.*



(2.1)

Im Folgenden wird ein voll ausgeschriebener **Compiler** als $C^{o\_j}_{i\_w\_k\_min}$ geschrieben, wobei C_w die **Sprache** bezeichnet, die der Compiler als **Input** nimmt und zu einer nicht näher spezifizierten Maschienensprache L_{B_i} einer Maschiene M_i kompiliert. Fall die Notwendigkeit besteht die **Maschiene** M_i anzugeben, zu dessen **Maschienensprache** L_{B_i} der Compiler kompiliert, wird das als C_i geschrieben. Falls die Notwendigkeit besteht die **Sprache** L_o anzugeben, in der der Compiler selbst geschrieben

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

ist, wird das als C^o geschrieben. Falls die Notwendigkeit besteht die Version der Sprache, in die der Compiler kompiliert L_{w_k} oder in der er selbst geschrieben ist L_{o_j} anzugeben, wird das als $C^{o\_j}_{w\_k}$ geschrieben. Falls es sich um einen **minimalen Compiler** handelt (Definition 4.2) kann man das als C_{min} schreiben.

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

Definition 2.3: Maschienensprache

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

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

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

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

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

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

[c] Z.B. sind in X_{86_64} die Instructions in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels `jmp <label>` gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet kann keine direkte Entsprechung in einem Prozessor und Hauptspeicher haben.

Ein **Assembler** (Definition 2.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschienencode** bzw. **Objectcode** (Definition 2.6) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur das als Output liefern, was er eigentlich haben will, nämlich den **Maschienencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 2.7) zu Maschiendencode zusammengesetzt wird ausführbar ist.

Definition 2.5: Assembler

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschienencode** der in **Maschienensprache** geschrieben ist.*

Definition 2.6: Objectcode

*Bei komplexeren Compilern, die es erlauben den Programmcode in **mehrere Dateien** aufzuteilen wird **Objectcode** erzeugt, der neben der Folge von Maschienenbefehlen in binärer Repräsentation auch noch Informationen für den **Linker** enthält, die im späteren **Maschiendencode** nicht mehr enthalten sind, sobald der **Linker** die Objektdateien zum Maschienencode zusammengesetzt hat.*

Definition 2.7: Linker

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

Der **Maschienencode**, denn ein üblicher Compiler einer Programmiersprache generiert, enthält seine Folge von Maschienenbefehlen üblicherweise in **binärer Repräsentation**, da diese in erster Linie für die Maschiene, die binär arbeitet verständlich sein sollen und nicht für den Programmierer.

Der **PicoC-Compiler**, der den Zweck erfüllt für Studenten ein **Anschauungs- und Lernwerkzeug** zu sein, generiert allerdings Maschienencode, der die Maschienenbefehle bzw. RETI-Befehle in **menschenlesbarer Form** mit ausgeschriebenen RETI-Operationen, RETI-Registern und Immediates (Definition 2.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht dessen mögliche interne Umsetzung[2].

Definition 2.8: Immediate

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

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

Definition 2.9: Transpiler (bzw. Source-to-source Compiler)

*Kompiliert zwischen Sprachen, die ungefähr auf dem **gleichen** Level an **Abstraktion** arbeiten*[a]

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

Definition 2.10: Cross-Compiler

*Kompiliert auf einer **Maschine** M_1 ein Program, dass in einer **Sprache** L_w geschrieben ist für eine **andere Maschine** M_2, wobei beide Maschinen M_1 und M_2 unterschiedliche **Maschienensprachen** B_1 und B_2 haben.*[a]

[a] Beim **PicoC-Compiler** handelt es sich um einen **Cross-Compiler** C_{PicoC}^{Python}.

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

Ein **Cross-Compiler** ist entweder notwendig, wenn noch kein Compiler C_w für die **Wunschsprache** L_w existiert, der unter der **Maschienensprache** B_2 einer Zielmaschine M_2 läuft oder die Zielmaschine M_2 nicht ausreichend **Rechenleistung** hat, um ein Programm in der Wunschsprache L_w selbst zeitnah zu kompilieren.[3]

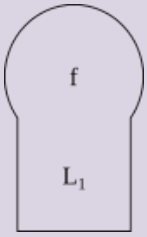
2.1.1 T-Diagramme

Um die Architektur von **Compilern** und **Interpretern** übersichtlich darzustellen eignen sich **T-Diagramme**[4] besonders gut, da diese optimal darauf **zugeschnitten** sind die Eigenheiten von Compilern in ihrer Art der Darstellung unterzubringen.

Die **Notation** setzt sich dabei aus den **Blöcken** für ein Program (Definition 2.11), einen Übersetzer (Definition 2.12), einen Interpreter (Definition 2.13) und eine Maschiene (Definition 2.14) zusammen.

Definition 2.11: T-Diagram Programm

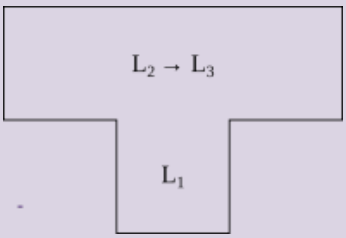
*Repräsentiert ein **Programm**, dass in der **Sprache** L_1 geschrieben ist und die **Funktion** f berechnet.*



Es ist nicht notwendig beim einem entsprechenden **Platzhalter** für einen **Namen**, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht.

Definition 2.12: T-Diagram Übersetzer

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



Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** ein einem Schritt. **Kompilieren** ist also auch

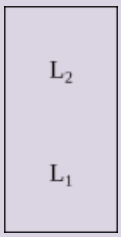
[3] Die an vielen Universitäten und Schulen eingesetzen programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren Microcontroller eine **C-ähnliche Sprache** in die Maschienensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst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

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

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

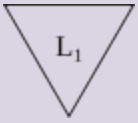
Definition 2.13: T-Diagram Interpreter

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



Definition 2.14: T-Diagram Maschiene

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



[a] Wenn die Maschiene **Programme** in einer höheren Sprache als **Maschienensprache** ausführt, ist das auch erlaubt das so aufzuschreiben, dann handelt es sich um eine **Abstrakte Maschiene**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM)

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

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

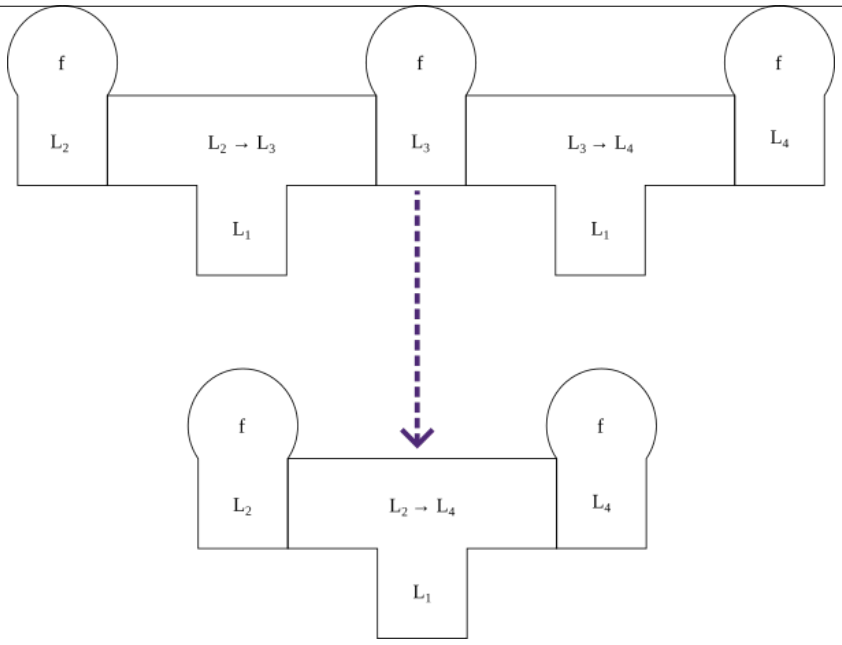


Abbildung 2.1: Horinzontale Übersetzungszwischenschritte zusammenfassen

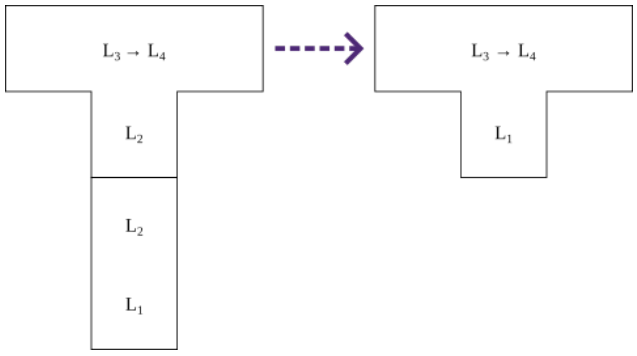


Abbildung 2.2: Vertikale Interpretierungszwischenschritte zusammenfassen

2.2 Grammatiken

2.3 Grundlagen

Definition 2.15: Sprache

Definition 2.16: Chromsky Hierarchie

Definition 2.17: Grammatik

Definition 2.18: Reguläre Sprachen

Definition 2.19: Kontextfreie Sprachen

Definition 2.20: Ableitung

Definition 2.21: Links- und Rechtsableitung

Definition 2.22: Linksrekursive Grammatiken

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

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

$$A \Rightarrow^* Aa,$$

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

2.3.1 Mehrdeutige Grammatiken

Definition 2.23: Ableitungsbaum

Definition 2.24: Mehrdeutige Grammatik

2.3.2 Präzidenz und Assoziativität

Definition 2.25: Assoziativität

Definition 2.26: Präzidenz

Definition 2.27: Wortproblem

Definition 2.28: LL(k)-Grammatik

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

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

2.4 Lexikalische Analyse

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb der **Pipe Architektur** bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.29) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 2.29: Pattern

***Beschreibung** aller möglichen **Lexeme** einer Menge $\mathbb{P}_T$, die einem bestimmten **Token** T zugeordnet werden. Die Menge $\mathbb{P}_T$ ist eine möglicherweise unendliche Menge von **Wörtern**, die sich mit den Produktionen einer **regulären Grammatik** G_{Lex} einer **regulären Sprache** L_{Lex} beschreiben lassen [a], die für die Beschreibung eines **Tokens** T zuständig sind.[b]*

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] *What is the difference between a token and a lexeme?*

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 2.30) genannt.

Definition 2.30: Lexeme

*Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} matched.[a]*

[a] *What is the difference between a token and a lexeme?*

Diese **Lexeme** werden vom **Lexer** im **Inputstring** identifziert und **Tokens** T zugeordnet (Definition 2.31). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.31: Lexer (bzw. Scanner)

*Ein **Lexer** ist eine **partielle** Funktion $lex : \Sigma^* \rightharpoonup (N \times W)^*$, welche ein **Wort** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls diese Folge von Symbolen sich unter der **regulären Grammatik** G_{Lex}, der **regulären Sprache** L_{Lex} ableiten lässt.[a]*

[a] *lecture-notes-2021*.

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

Um Verwirrung verzubäugen ist es wichtig folgende Unterscheidung hervorzuheben:

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

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

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

In der Code Generierung sind die **Bezeichner** (Definition 2.32) von **Variablen, Konstanten und Funktionen** die Symbole[a].

[a] Das ist der Grund, warum die **Tabelle**, in der Informationen zu **Bezeichnern** gespeichert werden, in Kapitel 3 **Symboltabelle** genannt wird.

Definition 2.32: Bezeichner (bzw. Identifier)

***Lexeme** bzw. **Tokenwert**, das bzw. der eine Konstante, Variable, Funktion usw. **eindeutig** benennt, außer wenn z.B. bei Funktionen die Programmiersprache das Überladen erlaubt usw.*[a]

[a] In diesem Fall wird die **Signatur** der Funktion als weiteres Unterschiedungsmerkmal hinzugenommen, damit es eindeutig ist

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[5] und Tabs `\t` aus dem Inputstring herauszufiltern. Das geschieht mittels des **Lexers**, der allen für die **Syntaktische Analyse** unwichtige Zeichen das leere Wort ϵ zuordnet. Das ist auch im Sinne der Definition, denn $\epsilon \in \Sigma^*$. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie `42`, `314` oder `12`. Die **Überbegriffe** bzw. **Tokennamen** für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. `NAME` und `NUM`[6], bzw. wenn man sich nicht Kurzformen sucht `IDENTIFIER` und `NUMBER`. Für Zeichenfolgen, wie `if` oder `}` sind die **Überbegriffe** genau die Bezeichnungen, die man diesen Zeichenfolgen geben würde, nämlich `IF` und `RBRACE`.

Ein **Lexeme** ist damit aber nicht das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann z.B. der Wert 99 durch zwei verschiedene Literale darstellt werden, einmal als ASCII-Zeichen `'c'` und des Weiteren auch in Dezimalschreibweise als `99`[7]. Der **Tokenwert** ist jedoch der letztendliche Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik** G_{Lex}, die zur Beschreibung der Token T einer regulären Sprache L_{Lex} verwendet wird, ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[8], unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

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

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

[7] Die Programmiersprache Python erlaubt es z.B. diesern Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

[8] Man nennt das auch einem **Lookahead** von 1

Definition 2.33: Literal

*Eine von möglicherweise vielen weiteren **Darstellungsformen** für ein und denselben **Wert**.*

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

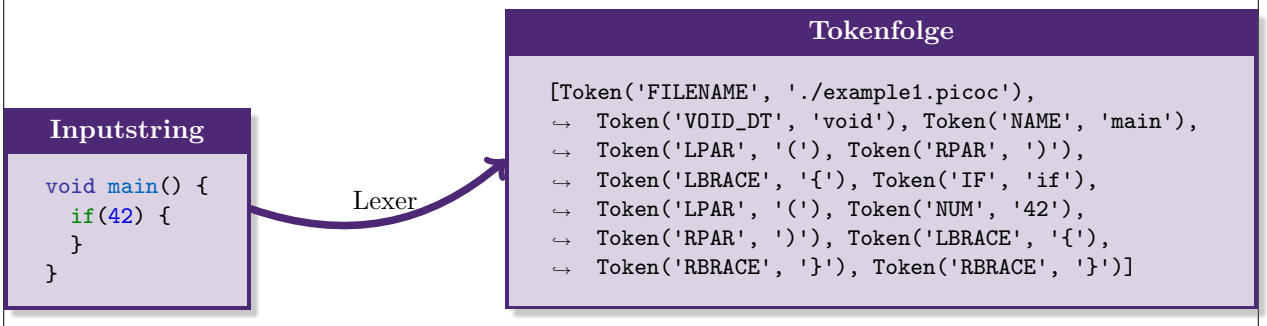


Abbildung 2.3: Veranschaulichung der Lexikalischen Analyse

2.5 Syntaktische Analyse

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

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 2.34) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.36), ein **Derivation Tree** (Definition 2.35) generiert, der als Zwischenstufe hin zum einem **Abstrakt Syntax Tree** (Definition 2.41) dient. Für einen ordentlichen Code ist es vor allem im Compilerbau förderlich kleinschrittig vorzugehen, deshalb erst die Generierung des **Derivation Tree** und dann der **Abstrakt Syntax Tree**.

Definition 2.34: Konkrette Syntax

***Syntax** einer **Sprache**, die durch die **Grammatiken** G_{Lex} und G_{Parse} zusammengenommen beschrieben wird.*

*Ein **Programm** in seiner **Textrepräsentation**, wie es in einer Textdatei nach den Produktionen der **Grammatiken** G_{Lex} und G_{Parse} abgeleitet steht, bevor man es kompiliert, ist in **Konkretter Syntax** aufgeschrieben.*[a]

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

Definition 2.35: Derivation Tree (bzw. Parse Tree)

__Compilerinterne Darstellung__ eines in __Konkretter Syntax__ geschriebenen Inputstrings als __Baumdatenstruktur__, in der __Nichtterminalsymbole__ die __Inneren Knoten__ des Baumes und __Terminalsymbole__ die __Blätter__ des Baumes bilden. Jede __Produktions__ der __Grammatik__ G_{Parse}, die ein Teil der __Konkrette Syntax__ ist, wird zu einem eigenen __Knoten__.

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

Definition 2.36: Parser

Ein Programm, dass eine __Eingabe__ in eine für die __Weiterverbeitung__ taugliche Form bringt.

2.36.1: *In Bezug auf Compilerbau ist ein __Parser__ ein Programm, dass einen Inputstring von __Konkretter Syntax__ in die compilerinterne Darstellung eines __Derivation Tree__ übersetzt, was auch als __Parsen__ bezeichnet wird*[a, b].

[a] Es gibt allerdings auch alternative Definitionen, denen nach ein Parser in Bezug auf Compilerbau ein Programm ist, dass einen Inputstring von **Konkretter Syntax** in **Abstrakte Syntax** übersetzt. Im Folgenden wird allerdings die obigte Definition 2.36.1 verwendet.

[b] *Compiler Design - Phases of Compiler*.

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung enstehen, ob ein **Lexer** nach der obigen Definition nicht auch ein **Parser** ist.

In Bezug auf Compilerbau ist ein **Lexer** ein Teil eines Parsers. Der Parser vereinigt sowohl die **Lexikalische Analyse**, als auch einen Teil der **Syntaktischen Analyse** in sich. Aber für sich isoliert, ohne Bezug zu Compilerbau betrachtet, ist ein Lexer nach Definition 2.36 ebenfalls ein Parser. Aber im Compilerbau hat **Parser** eine spezifischere Definition und hier überwiegt beim **Lexer** seine Funktionalität, dass er den Inputstring lexikalisch weiterverarbeitet, um ihn als Lexer zu bezeichnen, der Teil eines Parsers ist.

Die vom **Lexer** im Inputstring identifizierten **Token** werden in der **Syntaktischen Analyse** vom **Parser** (Definition 2.36) als **Wegweiser** verwendet, da je nachdem, in welcher Reihenfolge die **Token** auftauchen, dies einer anderen Ableitung in der **Grammatik** G_{Parse} entspricht. Dabei wird in der Grammatik nach dem **Tokennamen** unterschieden und nicht nach dem Tokenwert, da es nur von Interesse ist, ob an einer bestimmten Stelle z.B. eine `Zahl` steht und nicht, welchen konkretten Wert diese `Zahl` hat. Der **Tokenwert** ist erst später in der **Code Generierung** in 2.6 relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 2.37), denn ein Parser löst das **Wortproblem** (Definition 2.27) für die **Sprache**, die durch die **Konkrette Syntax** beschrieben wird und konstruiert parallel dazu oder im Nachgang aus den Informationen, die während der Ausführung des Recognition Algorithmus gesichert wurden den **Derivation Tree**.

Definition 2.37: Recognizer (bzw. Erkenner)

Entspricht dem Maschinenmodell eines __Automaten__. Im Bezug auf Compilerbau entspricht der __Recognizer__ einem __Kellerautomaten__, in dem __Wörter__ bestimmter __Kontextfreier Sprachen__ erkannt werden. Der __Recognizer__ erkennt, ob ein Iputstring bzw. __Wort__ sich mit den Produktionen der __Konkrette Syntax__ ableiten lässt, also ob er bzw. es Teil der Sprache ist, die von der __Konkretten Syntax__ beschrieben wird oder nicht[a].

[a] Das vom **Recognizer** gelöste Problem ist auch als **Wortproblem** bekannt.

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

- **Top-Down Parsing:** Der **Derivation Tree** wird von **oben-nach-unten** generiert, also von der **Wurzel** zu den **Blättern**. Dementsprechend fängt die Generierung des **Derivation Tree** mit dem **Startsymbol** der **Grammatik** an und wendet in jedem Schritt eine **Linksableitung** auf die **Nicht-Terminalsymbole** an, bis man **Terminalsymbole** hat und der gewünschte **Inputstring** abgeleitet wurde oder es sich herausstellt, dass dieser nicht abgeleitet werden kann.[a]

  Der Grund, warum die **Linksableitung** verwendet wird und nicht z.B. die **Rechtsableitung** ist, weil der das **Eingabewert** bzw. der **Inputstring** von **links nach rechts** eingelesen wird, was gut damit zusammenpasst, dass die **Linksableitung** die **Blätter** von **links-nach-rechts** generiert.

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

  Eine sehr einfach zu implementierende Technik für **Top-Down Parser** ist hierbei der **Rekursive Abstieg**. Dabei wird jedem **Nicht-Terminalsymbol** eine **Prozedur** zugeordnet, welche die Produktionsregeln dieses **Nicht-Terminalsymbols** umsetzt. Prozeduren rufen sich dabei wechselseitig gegenseitig entsprechend der **Produktionsregeln** auf, falls eine entsprechende Produktionsregel eine **Rekursion** enthält.

  Rekursiver Abstieg kann mit **Backtracking** verbunden werden, um auch Grammatiken parsen zu können, die nicht **LL(k)** (Definition 2.28) sind. Dabei werden meist nach dem **Depth-First-Search Prinzip** alle **Produktionen** für ein **Nicht-Terminalsymbol** solange durchgegangen bis der gewüschte Inpustring abgeleitet ist oder alle **Alternativen** für einen Schritt abgesucht sind, bis man wieder beim ersten Schritt angekommen ist und da auch alle **Alternativen** abgesucht sind.

  Mit dieser Methode ist das Parsen **Linksrekursiver Grammatiken** (Definition 2.22) allerdings nicht möglich, ohne die Grammatik vorher umgeformt zu haben und jegliche **Linksrekursion** aus der **Grammatik** entfernt zu haben, da diese zu **Unendlicher Rekursion** führt[b]

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

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

- **Chart Parser:** Es wird **Dynamische Programmierung** verwendet und partielle Zwischenergebnisse werden in einer **Tabelle** (bzw. einem **Chart**) gespeichert und können wiederverwendet werden. Das macht das Parsen **Kontextfreier Grammatiken** effizienter, sodass es nur noch **polynomielle** Zeit braucht, da **Backtracking** nicht mehr notwendig ist.[e]

[a] *What is Top-Down Parsing?*

[b] Diese Art von Parser wurde im **PicoC-Compiler** implementiert, als dieser noch auf dem Stand des **Bachelorprojektes** war, bevor er durch den nicht selbst implementierten **Earley Parser** von **Lark** (siehe *Lark - a parsing toolkit for Python*) ersetzt wurde.

[c] Diese Art von Parser ist im **RETI-Interpreter** implementiert, da die **RETI-Sprache** eine besonders simple **LL(1) Grammatik** besitzt. Dieser **Parser** wird auch als **Predictive Parser** oder **LL(k) Recursive Descent Parser** bezeichnet, wobei **Recursive Descent** das englische Wort für **Rekursiven Abstieg** ist.

[d] *What is Bottom-up Parsing?*

[e] Der **Earley Parser**, den **Lark** und damit der **PicoC-Compiler** verwendet fällt unter diese Kategorie

Der **Abstrakt Syntax Tree** wird mithilfe von **Transformern** (Definition 2.38) und **Visitors** (Definition 2.39) generiert und ist das Endprodukt der **Syntaktischen Analyse**. Wenn man die gesamte **Syntaktische Analyse** betrachtet, so übersetzt diese einen Inpustring von der **Konkretten Syntax** in die **Abstrakte Syntax** (Definition 2.40).

Die **Baumdatenstruktur** des **Derivation Tree** und **Abstrakt Syntax Tree** ermöglicht es die Operationen, die der Compiler bei der Weiterverarbeitung des Inputstrings ausführen muss möglichst **effizient** auszuführen.

Definition 2.38: Transformer

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

Definition 2.39: Visitor

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

*Kann theoretisch auch zur Konstruktion eines **Abstrakt Syntax Tree** verwendet werden, wenn z.B. eine externe Klasse verwendet wird, welches für die Konstruktion des **Abstrakt Syntax Tree** verantwortlich ist, aber dafür ist ein **Transformer** besser geeignet.*

Definition 2.40: Abstrakte Syntax

***Syntax**, die beschreibt, was für Arten von **Komposition** bei den **Knoten** eines **Abstrakt Syntax Trees** möglich sind.*[a]

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

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

Definition 2.41: Abstrakt Syntax Tree

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

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

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

Je weiter **unten**[9] und **links** ein Knoten im **Abstrakt Syntax Tree** liegt, desto eher wird dieser Knoten komplett abgearbeitet sein, da in der **Code Generierung** die Knoten nach dem **Depth First Search**

[9] In der Informatik wachsen **Bäume** von **oben-nach-unten**. Die **Wurzel** ist also **oben**.

Prinzip abgearbeitet werden.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.4 die Syntaktische mit dem Beispiel aus Subkapitel 2.4 fortgeführt.

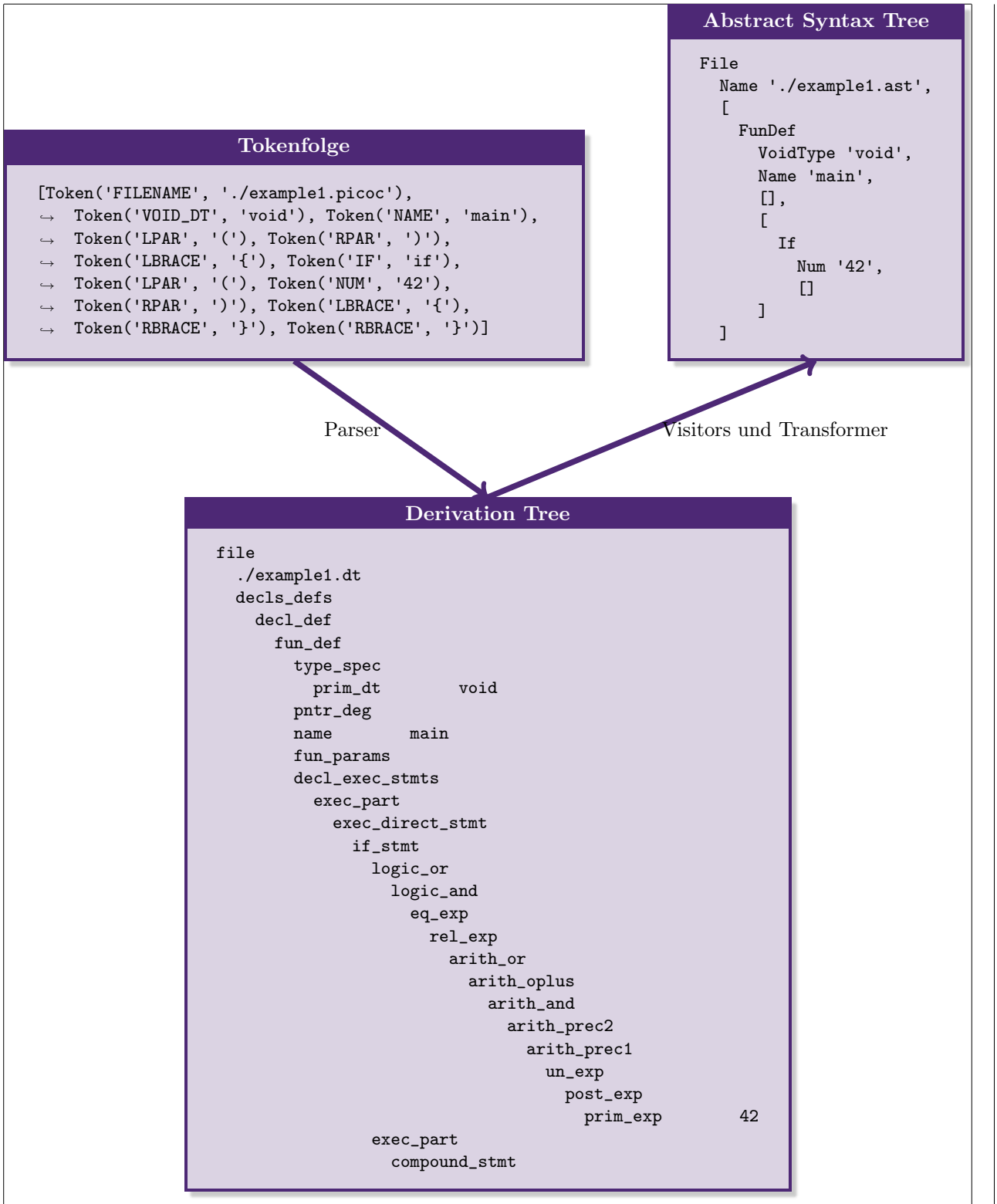


Abbildung 2.4: Veranschaulichung der Syntaktischen Analyse

2.6 Code Generierung

Definition 2.42: Pass

Definition 2.43: Monadische Normalform

Ein echter Compiler verwendet Graph Coloring ... Register ...

2.7 Fehlermeldungen

Definition 2.44: Fehlermeldung

2.7.1 Kategorien von Fehlermeldungen

3 Implementierung

3.1 Architektur

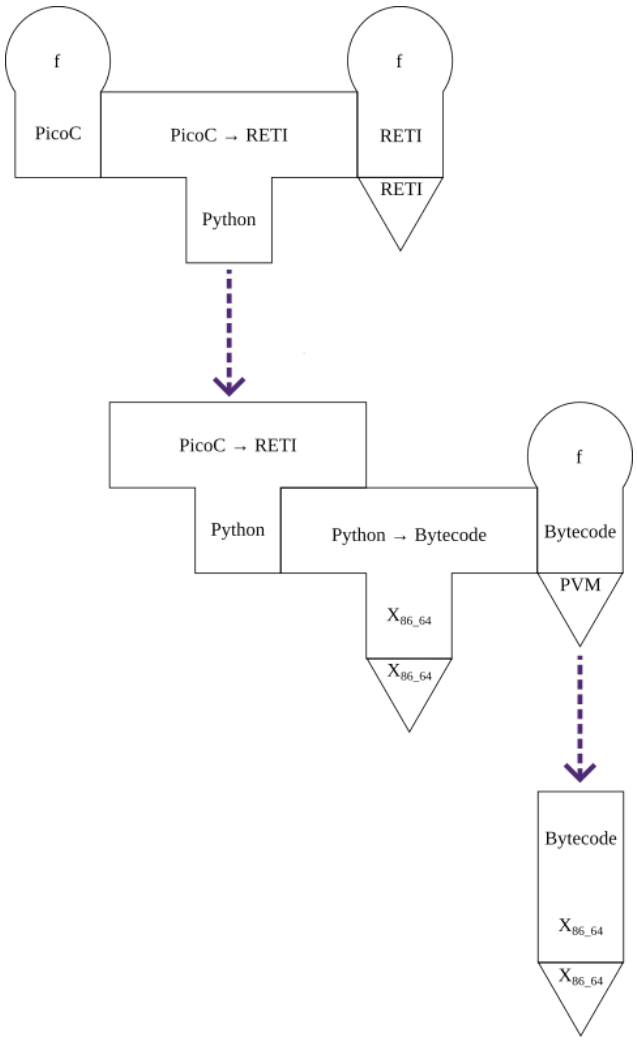


Abbildung 3.1: Cross-Compiler Kompiliervorgang ausgeschrieben

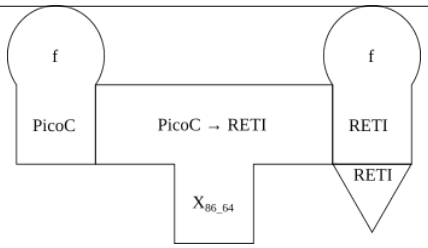


Abbildung 3.2: Cross-Compiler Kompiliervorgang Kurzform

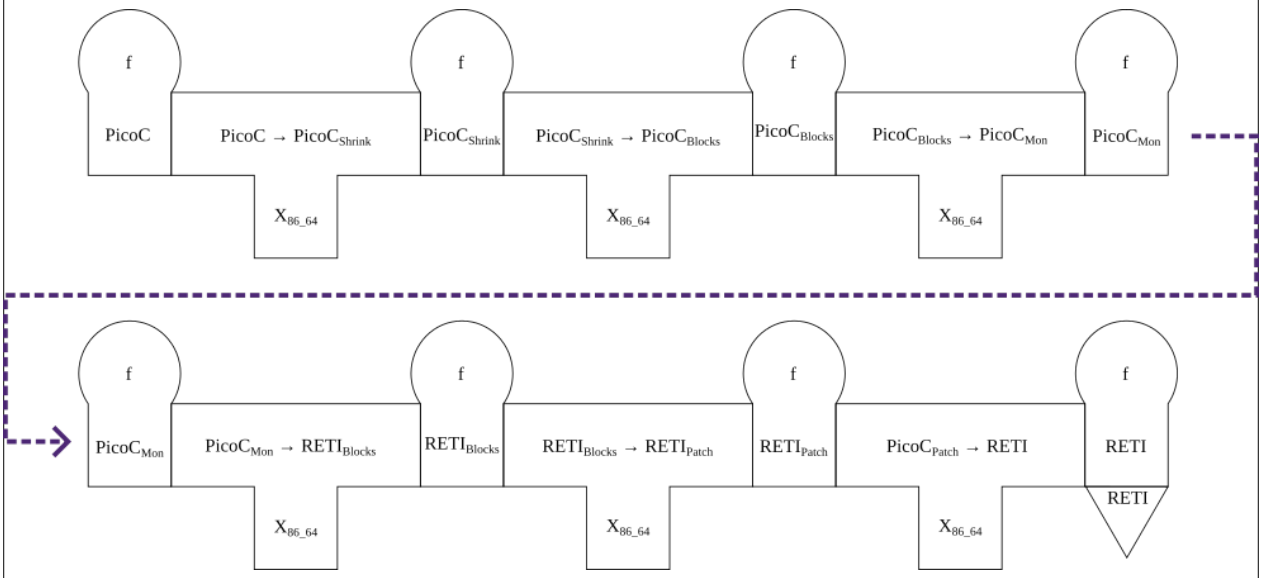


Abbildung 3.3: Architektur mit allen Passes ausgeschrieben

3.2 Lexikalische Analyse

3.2.1 Verwendung von Lark

3.2.2 Basic Parser

3.3 Syntaktische Analyse

3.3.1 Verwendung von Lark

3.3.2 Umsetzung von Präzidenz

Die **PicoC Sprache** hat dieselben **Präzidenzregeln** implementiert, wie die **Sprache C**[1]. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 3.4 aufgelistet.

[1] *C Operator Precedence - cppreference.com*.

Präzidenz	Operator	Beschreibung	Assoziativität

1	`a()`
	`a[]`
	`a.b`
		Funktionsaufruf
		Indexzugriff
		Attributzugriff
			Links, dann rechts →

2	`-a`
	`!a ~a`
	`*a &a`
		Unäres Minus
		Logisches NOT und Bitweise NOT
		Dereferenz und Referenz, auch Adresse-von
			Rechts, dann links ←

3	`a*b a/b a%b`	Multiplikation, Division und Modulo	Links, dann rechts →
4	`a+b a-b`	Addition und Subtraktion	
5	`a<b a<=b` `a>b a>=b`	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	`a==b a!=b`	Gleichheit und Ungleichheit	
7	`a&b`	Bitweise UND	
8	`a^b`	Bitweise XOR (exclusive or)	
9	`a|b`	Bitweise ODER (inclusive or)	
10	`a&&b`	Logiches UND	
11	$a \| \| b$	Logisches ODER	
12	`a=b`	Zuweisung	Rechts, dann links ←
13	`a,b`	Komma	Links, dann rechts →

Abbildung 3.4: Präzidenzregeln von PicoC

3.3.3 Derivation Tree Generierung

3.3.4 Early Parser

3.3.5 Derivation Tree Vereinfachung

3.3.6 Abstrakt Syntax Tree Generierung

3.3.6.1 ASTNode

3.3.6.2 PicoC Nodes

3.3.6.3 RETI Nodes

3.4 Code Generierung

3.4.1 Passes

3.4.1.1 PicoC-Shrink Pass

3.4.1.2 PicoC-Blocks Pass

3.4.1.3 PicoC-Mon Pass

Definition 3.1: Symboltabelle

3.4.1.4 RETI-Blocks Pass

3.4.1.5 RETI-Patch Pass

3.4.1.6 RETI Pass

3.4.2 Umsetzung von Pointern

3.4.2.1 Pointer Dereferenzierung durch Zugriff auf Arrayindex ersetzen

```
1 void main() {
2   int var = 42;
3   int *pntr = &var;
4   *pntr;
5 }
```

Abbildung 3.5: PicoC Code für Pointer Dereferenzierung

```
1  File
2    Name './example_pntr_deref.ast',
3    [
4      FunDef
5        VoidType 'void',
6        Name 'main',
7        [],
8        [
9          Assign
10           Alloc
11             Writeable,
12             IntType 'int',
13             Name 'var',
14           Num '42',
15         Assign
16           Alloc
17             Writeable,
18             PntrDecl
19               Num '1',
20               IntType 'int',
21             Name 'pntr',
22           Ref
23             Name 'var',
24         Exp
25           Deref
26             Name 'pntr',
27             Num '0'
28       ]
29   ]
```

Abbildung 3.6: Abstract Syntax Tree für Pointer Dereferenzierung

```
File
  Name './example_pntr_deref.picoc_shrink',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign
          Alloc
            Writeable,
            IntType 'int',
            Name 'var',
          Num '42',
        Assign
          Alloc
            Writeable,
            PntrDecl
              Num '1',
              IntType 'int',
            Name 'pntr',
          Ref
            Name 'var',
        Exp
          Subscr
            Name 'pntr',
            Num '0'
      ]
  ]
```

Abbildung 3.7: PicoC Shrink Pass für Pointer Dereferenzierung

3.4.2.2 Referenzierung

3.4.3 Umsetzung von Arrays

3.4.3.1 Initialisierung von Arrays

```
void main() {
  int ar[2][1] = {{4}, {2}};
}
```

Abbildung 3.8: PicoC Code für Array Initialisierung

```
File
  Name './example_array_init.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '2',
                Num '1'
              ],
              IntType 'int',
            Name 'ar',
          Array
            [
              Array
                [
                  Num '4'
                ],
              Array
                [
                  Num '2'
                ]
            ]
      ]
  ]
```

Abbildung 3.9: Abstract Syntax Tree für Array Initialisierung

```
File
  Name './example_array_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '4',
        Exp
          Num '2',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '2',
        Return
          Empty
      ]
  ]
```

Abbildung 3.10: PicoC Mon Pass für Array Initialisierung

```
File
  Name './example_array_init.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 4,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 1,
        LOADIN SP ACC 2,
        STOREIN DS ACC 0,
        ADDI SP 2,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.11: RETI Blocks Pass für Array Initialisierung

3.4.3.2 Zugriff auf Arrayindex

Der Zugriff auf einen bestimmten Index eines Arrays ist wie folgt umgesetzt:

```
void main() {
  int ar[2] = {1, 2};
  ar[2];
}
```

Abbildung 3.12: PicoC Code für Zugriff auf Arrayindex

```
File
  Name './example_array_access.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '2'
              ],
              IntType 'int',
            Name 'ar',
          Array
            [
              Num '1',
              Num '2'
            ],
        Exp
          Subscr
            Name 'ar',
            Num '2'
      ]
  ]
```

Abbildung 3.13: Abstract Syntax Tree für Zugriff auf Arrayindex

```
File
  Name './example_array_access.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '1',
        Exp
          Num '2',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '2',
        Ref
          GlobalRead
            Num '0',
        Exp
          Num '2',
        Ref
          Subscr
            Tmp
              Num '2',
            Tmp
              Num '1',
        Exp
          Subscr
            Tmp
              Num '1',
            Num '0',
        Return
          Empty
      ]
  ]
```

Abbildung 3.14: PicoC Mon Pass für Zugriff auf Arrayindex

```
File
  Name './example_array_access.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 1,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 1,
        LOADIN SP ACC 2,
        STOREIN DS ACC 0,
        ADDI SP 2,
        SUBI SP 1,
        LOADI IN1 0,
        ADD IN1 DS,
        STOREIN SP IN1 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP IN1 2,
        LOADIN SP IN2 1,
        MULTI IN2 1,
        ADD IN1 IN2,
        ADDI SP 1,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        LOADIN IN1 ACC 0,
        STOREIN SP ACC 1,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.15: RETI Blocks Pass für Zugriff auf Arrayindex

3.4.3.3 Zuweisung an Arrayindex

```
void main() {
  int ar[2];
  ar[2] = 42;
}
```

Abbildung 3.16: PicoC Code für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.ast',
  [
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Exp
          Alloc
            Writeable,
            ArrayDecl
              [
                Num '2'
              ],
              IntType 'int',
            Name 'ar',
        Assign
          Subscr
            Name 'ar',
            Num '2',
          Num '42'
      ]
  ]
```

Abbildung 3.17: Abstract Syntax Tree für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '42',
        Ref
          GlobalRead
            Num '0',
        Exp
          Num '2',
        Ref
          Subscr
            Tmp
              Num '2',
            Tmp
              Num '1',
        Assign
          Subscr
            Tmp
              Num '1',
            Num '0',
          Tmp
            Num '2',
        Return
          Empty
      ]
  ]
```

Abbildung 3.18: PicoC Mon Pass für Zuweisung an Arrayindex

```
File
  Name './example_array_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 42,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI IN1 0,
        ADD IN1 DS,
        STOREIN SP IN1 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP IN1 2,
        LOADIN SP IN2 1,
        MULTI IN2 1,
        ADD IN1 IN2,
        ADDI SP 1,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        LOADIN SP ACC 2,
        ADDI SP 2,
        STOREIN IN1 ACC 0,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.19: RETI Blocks Pass für Zuweisung an Arrayindex

3.4.4 Umsetzung von Structs

3.4.4.1 Deklaration von Structs

```
struct st1 {int *ar[3];};

struct st2 {struct st1 st;};

void main() {
}
```

Abbildung 3.20: PicoC Code für Deklaration von Structs

```
SymbolTable
  [
  Symbol(
    {
      type qualifier:         Empty()
      datatype:               ArrayDecl([Num('3')], PntrDecl(Num('1'), IntType('int')))
      name:                   Name('ar@st1')
      value or address:       Empty()
      position:               Pos(Num('1'), Num('17'))
```

```
10      size:                   Num('3')
11    },
12  Symbol(
13    {
14      type qualifier:         Empty()
15      datatype:               StructDecl(Name('st1'), [Alloc(Writeable(),
        ↪  ArrayDecl([Num('3')], PntrDecl(Num('1'), IntType('int'))), Name('ar'))])
16      name:                   Name('st1')
17      value or address:       [Name('ar@st1')]
18      position:               Pos(Num('1'), Num('7'))
19      size:                   Num('3')
20    },
21  Symbol(
22    {
23      type qualifier:         Empty()
24      datatype:               StructSpec(Name('st1'))
25      name:                   Name('st@st2')
26      value or address:       Empty()
27      position:               Pos(Num('3'), Num('23'))
28      size:                   Num('3')
29    },
30  Symbol(
31    {
32      type qualifier:         Empty()
33      datatype:               StructDecl(Name('st2'), [Alloc(Writeable(),
        ↪  StructSpec(Name('st1')), Name('st'))])
34      name:                   Name('st2')
35      value or address:       [Name('st@st2')]
36      position:               Pos(Num('3'), Num('7'))
37      size:                   Num('3')
38    },
39  Symbol(
40    {
41      type qualifier:         Empty()
42      datatype:               FunDecl(VoidType('void'), Name('main'), [])
43      name:                   Name('main')
44      value or address:       Empty()
45      position:               Pos(Num('5'), Num('5'))
46      size:                   Empty()
47    }
48  ]
```

Abbildung 3.21: Symboltabelle für Deklaration von Structs

3.4.4.2 Initialisierung von Structs

```
1 struct st1 {int *pntr[1];};
2
3 struct st2 {struct st1 st;};
4
5 void main() {
6   int var = 42;
7   struct st1 st = {.st={.pntr={{&var}}}};
8 }
```

Abbildung 3.22: PicoC Code für Initialisierung von Structs

```
File
  Name './example_struct_init.ast',
  [
    StructDecl
      Name 'st1',
      [
        Alloc
          Writeable,
          ArrayDecl
            [
              Num '1'
            ],
            PntrDecl
              Num '1',
              IntType 'int',
          Name 'pntr'
      ],
    StructDecl
      Name 'st2',
      [
        Alloc
          Writeable,
          StructSpec
            Name 'st1',
          Name 'st'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
      [
        Assign
          Alloc
            Writeable,
            IntType 'int',
            Name 'var',
          Num '42',
        Assign
          Alloc
            Writeable,
            StructSpec
              Name 'st1',
            Name 'st',
          Struct
            [
              Assign
                Name 'st',
                Struct
                  [
                    Assign
                      Name 'pntr',
                      Array
                        [
                          Array
                            [
                              Ref
                                Name 'var'
```

```
                            ]
                        ]
                    ]
                ]
            ]
        ]
    ]
```

Abbildung 3.23: Abstract Syntax Tree für Initialisierung von Structs

```
File
  Name './example_struct_init.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '42',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '1',
        Ref
          GlobalRead
            Num '0',
        Assign
          GlobalWrite
            Num '1',
          Tmp
            Num '1',
        Return
          Empty
      ]
  ]
```

Abbildung 3.24: PicoC Mon Pass für Initialisierung von Structs

```
File
  Name './example_struct_init.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 42,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 0,
        ADDI SP 1,
        SUBI SP 1,
        LOADI IN1 0,
        ADD IN1 DS,
        STOREIN SP IN1 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 1,
        ADDI SP 1,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.25: RETI Blocks Pass für Initialisierung von Structs

3.4.4.3 Zugriff auf Structattribut

```
struct pos {int x; int y;};

void main() {
  struct pos st = {.x=4, .y=2};
  st.y;
}
```

Abbildung 3.26: PicoC Code für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc
          Writeable,
          IntType 'int',
          Name 'x',
        Alloc
          Writeable,
          IntType 'int',
          Name 'y'
      ],
    FunDef
      VoidType 'void',
      Name 'main',
      [],
```

```
20      [
21        Assign
22          Alloc
23            Writeable,
24            StructSpec
25              Name 'pos',
26            Name 'st',
27          Struct
28            [
29              Assign
30                Name 'x',
31                Num '4',
32              Assign
33                Name 'y',
34                Num '2'
35            ],
36        Exp
37          Attr
38            Name 'st',
39            Name 'y'
40      ]
41  ]
```

Abbildung 3.27: Abstract Syntax Tree für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '4',
        Exp
          Num '2',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '2',
        Ref
          GlobalRead
            Num '0',
        Ref
          Attr
            Tmp
              Num '1',
            Name 'y',
        Exp
          Subscr
            Tmp
              Num '1',
            Num '0',
        Return
          Empty
      ]
  ]
```

Abbildung 3.28: PicoC Mon Pass für Zugriff auf Structattribut

```
File
  Name './example_struct_attr_access.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 4,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 1,
        LOADIN SP ACC 2,
        STOREIN DS ACC 0,
        ADDI SP 2,
        SUBI SP 1,
        LOADI IN1 0,
        ADD IN1 DS,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        ADDI IN1 1,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        LOADIN IN1 ACC 0,
        STOREIN SP ACC 1,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.29: RETI Blocks Pass für Zugriff auf Structattribut

3.4.4.4 Zuweisung an Structattribut

```
struct pos {int x; int y;};

void main() {
  struct pos st = {.x=4, .y=2};
  st.y = 42;
}
```

Abbildung 3.30: PicoC Code für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.ast',
  [
    StructDecl
      Name 'pos',
      [
        Alloc
          Writeable,
          IntType 'int',
          Name 'x',
        Alloc
```

```
12          Writeable,
13          IntType 'int',
14          Name 'y'
15        ],
16      FunDef
17        VoidType 'void',
18        Name 'main',
19        [],
20        [
21          Assign
22            Alloc
23              Writeable,
24              StructSpec
25                Name 'pos',
26              Name 'st',
27            Struct
28              [
29                Assign
30                  Name 'x',
31                  Num '4',
32                Assign
33                  Name 'y',
34                  Num '2'
35              ],
36          Assign
37            Attr
38              Name 'st',
39              Name 'y',
40            Num '42'
41        ]
42    ]
```

Abbildung 3.31: Abstract Syntax Tree für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.picoc_mon',
  [
    Block
      Name 'main.0',
      [
        Exp
          Num '4',
        Exp
          Num '2',
        Assign
          GlobalWrite
            Num '0',
          Tmp
            Num '2',
        Exp
          Num '42',
        Ref
          GlobalRead
            Num '0',
        Ref
          Attr
            Tmp
              Num '1',
            Name 'y',
        Assign
          Subscr
            Tmp
              Num '1',
            Num '0',
          Tmp
            Num '2',
        Return
          Empty
      ]
  ]
```

Abbildung 3.32: PicoC Mon Pass für Zuweisung an Structattribut

```
File
  Name './example_struct_attr_assignment.reti_blocks',
  [
    Block
      Name 'main.0',
      [
        SUBI SP 1,
        LOADI ACC 4,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI ACC 2,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        STOREIN DS ACC 1,
        LOADIN SP ACC 2,
        STOREIN DS ACC 0,
        ADDI SP 2,
        SUBI SP 1,
        LOADI ACC 42,
        STOREIN SP ACC 1,
        SUBI SP 1,
        LOADI IN1 0,
        ADD IN1 DS,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        ADDI IN1 1,
        STOREIN SP IN1 1,
        LOADIN SP IN1 1,
        LOADIN SP ACC 2,
        ADDI SP 2,
        STOREIN IN1 ACC 0,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.33: RETI Blocks Pass für Zuweisung an Structattribut

3.4.5 Umsetzung des Zusammenspiels der Derived Datatypes

3.4.5.1 Definition von Variablen mit den Derived Datatypes

```
struct ar_with_len {int ar[2]; int len;};

void main() {
  struct ar_with_len st_ar[3];
  int (*pntr1)[3];
  int *(*pntr2)[3];
}
```

Abbildung 3.34: PicoC Code für Definition von Variablen

```
SymbolTable
  [
  Symbol(
```

```
4     {
5       type qualifier:         Empty()
6       datatype:               ArrayDecl([Num('2')], IntType('int'))
7       name:                   Name('ar@ar_with_len')
8       value or address:       Empty()
9       position:               Pos(Num('1'), Num('24'))
10      size:                   Num('2')
11    },
12  Symbol(
13    {
14      type qualifier:         Empty()
15      datatype:               IntType('int')
16      name:                   Name('len@ar_with_len')
17      value or address:       Empty()
18      position:               Pos(Num('1'), Num('35'))
19      size:                   Num('1')
20    },
21  Symbol(
22    {
23      type qualifier:         Empty()
24      datatype:               StructDecl(Name('ar_with_len'), [Alloc(Writeable(),
        ↪  ArrayDecl([Num('2')], IntType('int')), Name('ar')), Alloc(Writeable(),
        ↪  IntType('int'), Name('len'))])
25      name:                   Name('ar_with_len')
26      value or address:       [Name('ar@ar_with_len'), Name('len@ar_with_len')]
27      position:               Pos(Num('1'), Num('7'))
28      size:                   Num('3')
29    },
30  Symbol(
31    {
32      type qualifier:         Empty()
33      datatype:               FunDecl(VoidType('void'), Name('main'), [])
34      name:                   Name('main')
35      value or address:       Empty()
36      position:               Pos(Num('3'), Num('5'))
37      size:                   Empty()
38    },
39  Symbol(
40    {
41      type qualifier:         Writeable()
42      datatype:               ArrayDecl([Num('3')], StructSpec(Name('ar_with_len')))
43      name:                   Name('st_ar@main')
44      value or address:       Num('0')
45      position:               Pos(Num('4'), Num('21'))
46      size:                   Num('9')
47    },
48  Symbol(
49    {
50      type qualifier:         Writeable()
51      datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], IntType('int')))
52      name:                   Name('pntr1@main')
53      value or address:       Num('9')
54      position:               Pos(Num('5'), Num('8'))
55      size:                   Num('1')
56    },
57  Symbol(
58    {
```

```
59      type qualifier:         Writeable()
60      datatype:               PntrDecl(Num('1'), ArrayDecl([Num('3')], PntrDecl(Num('1'),
        ↪  IntType('int'))))
61      name:                   Name('pntr2@main')
62      value or address:       Num('10')
63      position:               Pos(Num('6'), Num('9'))
64      size:                   Num('1')
65    }
66  ]
```

Abbildung 3.35: Symboltabelle für Definition von Variablen

3.4.6 Umsetzung von Funktionen

3.4.6.1 Funktionen auflösen zu RETI Code

3.4.6.1.1 Sprung zur Main Funktion

3.4.6.2 Funktionsdeklaration

3.4.6.3 Funktionsdefinition

3.4.6.3.1 Allocation von Variablen

3.4.6.4 Funktionsaufruf

3.4.6.4.1 Ohne Rückgabewert

```
1 void stack_fun();
2
3 void main() {
4   stack_fun();
5   return;
6 }
7
8 void stack_fun() {
9 }
```

Abbildung 3.36: PicoC Code für Funktionsaufruf ohne Rückgabewert

```
File
  Name './example_function_call_no_return_value.picoc_mon',
  [
    Block
      Name 'main.1',
      [
        StackMalloc
          Num '2',
        NewStackframe
          Name 'stack_fun',
          GoTo
            Name 'addr@next_instr',
        Exp
          GoTo
            Name 'stack_fun.0',
        RemoveStackframe,
        Return
          Empty
      ],
    Block
      Name 'stack_fun.0',
      [
        Return
          Empty
      ]
  ]
```

Abbildung 3.37: PicoC Mon Pass für Funktionsaufruf ohne Rückgabewert

```
File
  Name './example_function_call_no_return_value.reti_blocks',
  [
    Block
      Name 'main.1',
      [
        SUBI SP 2,
        MOVE BAF ACC,
        ADDI SP 2,
        MOVE SP BAF,
        SUBI SP 2,
        STOREIN BAF ACC 0,
        LOADI ACC GoTo
                    Name 'addr@next_instr',
        ADD ACC CS,
        STOREIN BAF ACC -1,
        Exp
          GoTo
            Name 'stack_fun.0',
        MOVE BAF IN1,
        LOADIN IN1 BAF 0,
        MOVE IN1 SP,
        LOADIN BAF PC -1
      ],
    Block
      Name 'stack_fun.0',
      [
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.38: RETI Blocks Pass für Funktionsaufruf ohne Rückgabewert

```
SUBI SP 2;
MOVE BAF ACC;
ADDI SP 2;
MOVE SP BAF;
SUBI SP 2;
STOREIN BAF ACC 0;
LOADI ACC 10;
ADD ACC CS;
STOREIN BAF ACC -1;
JUMP 5;
MOVE BAF IN1;
LOADIN IN1 BAF 0;
MOVE IN1 SP;
LOADIN BAF PC -1;
LOADIN BAF PC -1;
```

Abbildung 3.39: RETI Pass für Funktionsaufruf ohne Rückgabewert

3.4.6.4.2 Mit Rückgabewert

```
1 void stack_fun() {
2   return 42;
3 }
4
5 void main() {
6   int var = stack_fun();
7 }
```

Abbildung 3.40: PicoC Code für Funktionsaufruf mit Rückgabewert

```
1 File
2   Name './example_function_call_with_return_value.picoc_mon',
3   [
4     Block
5       Name 'stack_fun.1',
6       [
7         Exp
8           Num '42',
9         Return
10          Tmp
11            Num '1'
12      ],
13    Block
14      Name 'main.0',
15      [
16        StackMalloc
17          Num '2',
18        NewStackframe
19          Name 'stack_fun',
20          GoTo
21            Name 'addr@next_instr',
22        Exp
23          GoTo
24            Name 'stack_fun.1',
25        RemoveStackframe,
26        Assign
27          GlobalWrite
28            Num '0',
29          Tmp
30            Num '1',
31        Return
32          Empty
33      ]
34  ]
```

Abbildung 3.41: PicoC Mon Pass für Funktionsaufruf mit Rückgabewert

```
File
  Name './example_function_call_with_return_value.reti_blocks',
  [
    Block
      Name 'stack_fun.1',
      [
        SUBI SP 1,
        LOADI ACC 42,
        STOREIN SP ACC 1,
        LOADIN SP ACC 1,
        ADDI SP 1,
        LOADIN BAF PC -1
      ],
    Block
      Name 'main.0',
      [
        SUBI SP 2,
        MOVE BAF ACC,
        ADDI SP 2,
        MOVE SP BAF,
        SUBI SP 2,
        STOREIN BAF ACC 0,
        LOADI ACC GoTo
                    Name 'addr@next_instr',
        ADD ACC CS,
        STOREIN BAF ACC -1,
        Exp
          GoTo
            Name 'stack_fun.1',
        MOVE BAF IN1,
        LOADIN IN1 BAF 0,
        MOVE IN1 SP,
        LOADIN SP ACC 1,
        STOREIN DS ACC 0,
        ADDI SP 1,
        LOADIN BAF PC -1
      ]
  ]
```

Abbildung 3.42: RETI Blocks Pass für Funktionsaufruf mit Rückgabewert

```
1  JUMP 7;
2  SUBI SP 1;
3  LOADI ACC 42;
4  STOREIN SP ACC 1;
5  LOADIN SP ACC 1;
6  ADDI SP 1;
7  LOADIN BAF PC -1;
8  SUBI SP 2;
9  MOVE BAF ACC;
10 ADDI SP 2;
11 MOVE SP BAF;
12 SUBI SP 2;
13 STOREIN BAF ACC 0;
14 LOADI ACC 17;
15 ADD ACC CS;
16 STOREIN BAF ACC -1;
17 JUMP -15;
18 MOVE BAF IN1;
19 LOADIN IN1 BAF 0;
20 MOVE IN1 SP;
21 LOADIN SP ACC 1;
22 STOREIN DS ACC 0;
23 ADDI SP 1;
24 LOADIN BAF PC -1;
```

Abbildung 3.43: RETI Pass für Funktionsaufruf mit Rückgabewert

3.4.6.4.3 Umsetzung von Call by Sharing für Arrays

3.4.6.4.4 Umsetzung von Call by Value für Structs

3.4.7 Umsetzung kleinerer Details

3.5 Fehlermeldungen

3.5.1 Error Handler

3.5.2 Arten von Fehlermeldungen

3.5.2.1 Syntaxfehler

3.5.2.2 Laufzeitfehler

4 Ergebnisse und Ausblick

4.1 Compiler

4.2 Showmode

4.3 Qualitätssicherung

4.4 Kommentierter Kompiliervorgang

4.5 Erweiterungsideen

Wenn eines Tages eine **RETI-CPU** auf einem **FPGA** implementiert werden sollte, sodass ein **provisorisches Betriebssystem** darauf laufen könnte, dann wäre der nächste Schritt einen **Self-Compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ (Defintion 4.1) zu schreiben. Dadurch kann die **Unabhängigkeit** von der Programmiersprache L_Python, in der der momentane Compiler C_{PicoC} für L_{PicoC} implementiert ist und die Unabhängigkeit von einer **anderen Maschiene**, die bisher immer für das Cross-Compiling notwendig war erreicht werden.

Definition 4.1: Self-compiling Compiler

*Compiler C^w_w, der in der Sprache L_w **geschrieben** ist, die er **selbst** kompiliert. Also ein Compiler, der sich **selbst** kompilieren kann.*

Will man nun für eine Maschiene M_{RETI}, auf der bisher keine anderen Programmiersprachen mittels **Bootstrapping** (Definition 4.4) zum laufen gebracht wurden, den gerade beschriebenen **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ implementieren und hat bereits den gesamtem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ in der Sprache L_{PicoC} geschrieben, so stösst man auf ein Problem, dass auf das **Henne-Ei-Problem**[1] reduziert werden kann. Man bräuchte, um den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auf der **Maschiene** M_{RETI} zu kompilieren bereits einen kompilierten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der mit der Maschienensprache B_{RETI} läuft. Es liegt eine **zirkulare Abhängigkeit** vor, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Da man den gesamten **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ nicht selbst komplett in der Maschienensprache B_{RETI} schreiben will, wäre eine Möglichkeit, dass man den **Cross-Compiler** C^{Python}_{PicoC}, den man bereits in der Programmiersprache L_{Python} implementiert hat, der in diesem Fall einen **Bootstrapping Compiler** (Definition 4.3) darstellt, auf einer anderen Maschiene M_{other} dafür nutzt, damit dieser den **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ für die Maschiene M_{RETI} kompiliert bzw. **bootstraped** und man den kompilierten

[1] Beschreibt die Situation, wenn ein System sich selbst als **Abhängigkeit** hat, damit es überhaupt einen **Anfang** für dieses System geben kann. Dafür steht das Problem mit der **Henne** und dem **Ei** sinnbildlich, da hier die Frage ist, wie das ganze seinen Anfang genommen hat, da beides **zirkular** voneinander abhängt.

RETI-Maschiendencode dann einfach von der Maschiene M_{other} auf die Maschiene M_{RETI} kopiert.[2]

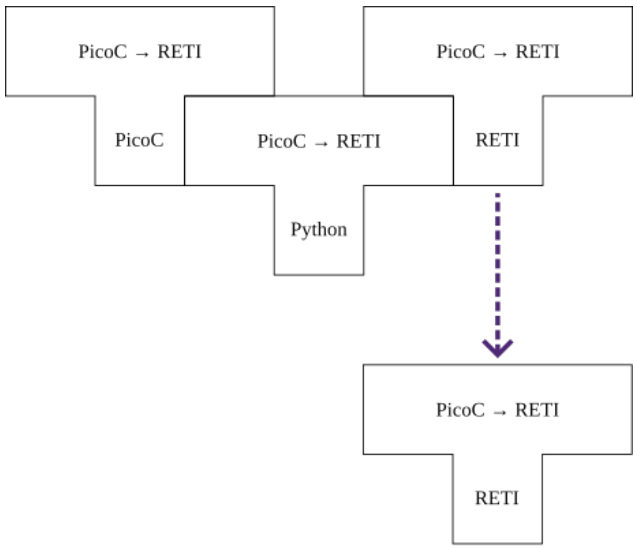


Abbildung 4.1: Cross-Compiler als Bootstrap Compiler

Einen ersten **minimalen Compiler** $C_{2\_w\_min}$ für eine Maschiene M_2 und Wunschsprache L_w kann man entweder mittels eines **externen Bootstrap Compilers** C^o_w kompilieren[a] oder man schreibt ihn direkt in der **Maschienensprache** B_2 bzw. wenn ein **Assembler** vorhanden ist, in der **Assemblesprache** A_2.

Die letzte Option wäre allerdings nur beim allerersten Compiler C_{first} für eine allererste **abstraktere Programmiersprache** L_{first} mit Schleifen, Verzweigungen usw. notwendig gewesen. Ansonsten hätte man immer eine Kette, die beim allersten Compiler C_{first} anfängt fortführen können, in der ein Compiler einen anderen Compiler kompiliert bzw. einen ersten minimalen Compiler kompiliert und dieser minimale Compiler dann eine umfangreichere Version von sich kompiliert usw.

[a] In diesem Fall, dem **Cross-Compiler** C^{Python}_{PicoC}.

Definition 4.2: Minimaler Compiler

*Compiler C_{w_min}, der nur die **notwendigsten Funktionalitäten** einer Wunschsprache L_w, wie **Schleifen**, **Verzweigungen** kompiliert, die für die Implementierung eines **Self-compiling Compilers** C^w_w oder einer **ersten Version** $C^{w_i}_{w_i}$ des Self-compiling Compilers C^w_w wichtig sind.*[a]

[a] Den **PicoC-Compiler** könnte man auch als einen **minimalen Compiler** ansehen.

[2] Im Fall, dass auf der Maschiene M_{RETI} die Programmiersprache L_{Python} bereits mittels **Bootstrapping** zum Laufen gebracht wurde, könnte der **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$ auch mithife des **Cross-Compilers** C^{Python}_{PicoC} als **externe Entität** und der Programmiersprache L_{Python} auf der Maschiene M_{RETI} selbst kompiliert werden.

Definition 4.3: Boostrap Compiler

*Compiler C^o_w, der es ermöglicht einen **Self-compiling Compiler** C^w_w zu **boostrapen**, indem der Self-compiling Compiler C^w_w mit dem **Bootstrap Compiler** C^o_w **kompiliert** wird[a]. Der Bootstrapping Compiler stellt die **externe Entität** dar, die es ermöglicht die **zirkulare Abhängigkeit**, dass initial ein **Self-compiling Compiler** C^w_w bereits kompiliert vorliegen müsste, um sich selbst kompilieren zu können, zu brechen.*

[a] Dabei kann es sich um einen **lokal** auf der Maschiene selbst laufenden Compiler oder auch um einen **Cross-Compiler** handeln.

Aufbauend auf dem **Self-compiling Compiler** $C^{PicoC}_{RETI\_PicoC}$, der einen **minimalen Compiler** (Definition 4.2) für eine Teilmenge der **Programmiersprache** C bzw. L_C darstellt, könnte man auch noch weitere Teile der Programmiersprache C bzw. L_C für die Maschiene M_{RETI} mittels **Bootstrapping** implementieren.[3]

Das bewerkstelligt man, indem man **iterativ** auf der Zielmaschine M_{RETI} selbst, aufbauend auf diesem **minimalen Compiler** $C^{PicoC}_{RETI\_PicoC}$, wie in Subdefinition 4.4.1 den minimalen Compiler schrittweise zu einem immer vollständigeren **C-Compiler** C_C weiterentwickelt.

Definition 4.4: Bootstrapping

*Wenn man einen **Self-compiling Compiler** C^w_w einer Wunschsprache L_w auf einer **Zielmaschine** M zum laufen bringt[abcd]. Dabei ist die Art von **Bootstrapping** in 4.4.1 nochmal gesondert hervorzuheben:*

4.4.1: *Wenn man die **aktuelle Version** eines **Self-compiling Compilers** $C^{w_i}_{w_i}$ der Wunschsprache L_{w_i} mithilfe von **früheren Versionen** seiner selbst kompiliert. Man schreibt also z.B. die aktuelle Version des Self-compiling Compilers in der Sprache $L_{w_{i-1}}$, welche von der früheren Version des Compilers, dem Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ kompiliert wird und schafft es so **iterativ** immer umfangreichere Compiler zu bauen.[efg]*

[a] Z.B. mithilfe eines **Bootstrap Compilers**.

[b] Der Begriff hat seinen Ursprung in der englischen **Redewendung** „pulling yourself up by your own bootstraps“, was im deutschen ungefähr der aus den **Lügengeschichten des Freiherrn von Münchhausen** bekannten Redewendung „sich am eigenen Schopf aus dem Sumpf ziehen“entspricht.

[c] Hat man einmal einen solchen **Self-compiling Compiler** C^w_w auf der Maschiene M zum laufen gebracht, so kann man den Compiler auf der Maschiene M weiterentwicklern, ohne von externen Entitäten, wie einer bestimmten Sprache L_o, in der der Compiler oder eine frühere Version des Compilers ursprünglich geschrieben war abhängig zu sein.

[d] Einen Compiler in der Sprache zu schreiben, die er selbst kompiliert und diesen Compiler dann sich selbst kompilieren zu lassen, kann eine gute **Probe aufs Exempel** darstellen, dass der Compiler auch wirklich funktioniert.

[e] Es ist hierbei theoretisch nicht notwendig den **letzten** Self-compiling Compiler $C^{w_{i-1}}_{w_{i-1}}$ für das Kompilieren des **neuen** Self-compiling Compilers $C^{w_i}_{w_i}$ zu verwenden, wenn z.B. der **Self-compiling Compiler** $C^{w_{i-3}}_{w_{i-3}}$ auch bereits alle Funktionalitäten, die beim Schreiben des **Self-compiling Compilers** C^w_w verwendet werden kompilieren kann.

[f] Der Begriff ist sinnverwandt mit dem **Booten** eines Computers, wo die wichtigste Software, der **Kernel** zuerst in den Speicher geladen wird und darauf aufbauend von diesem dann das Betriebssysteme, welches bei Bedarf dann **Systemsoftware**, Software, die das Ausführen von Anwendungssoftware ermöglicht oder unterstützt, wie z.B. Treiber. und **Anwendungssoftware**, Software, deren Anwendung darin besteht, dass sie dem Benutzer unmittelbar eine Dienstleistung zur Verfügung stellt, lädt.

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

[3] Natürlich könnte man aber auch einfach den **Cross-Compiler** C^{Python}_{PicoC} um weitere Funktionalitäten von L_C erweitern, hat dann aber weiterhin eine **Abhängigkeit** von der Programmiersprache L_{Python}.

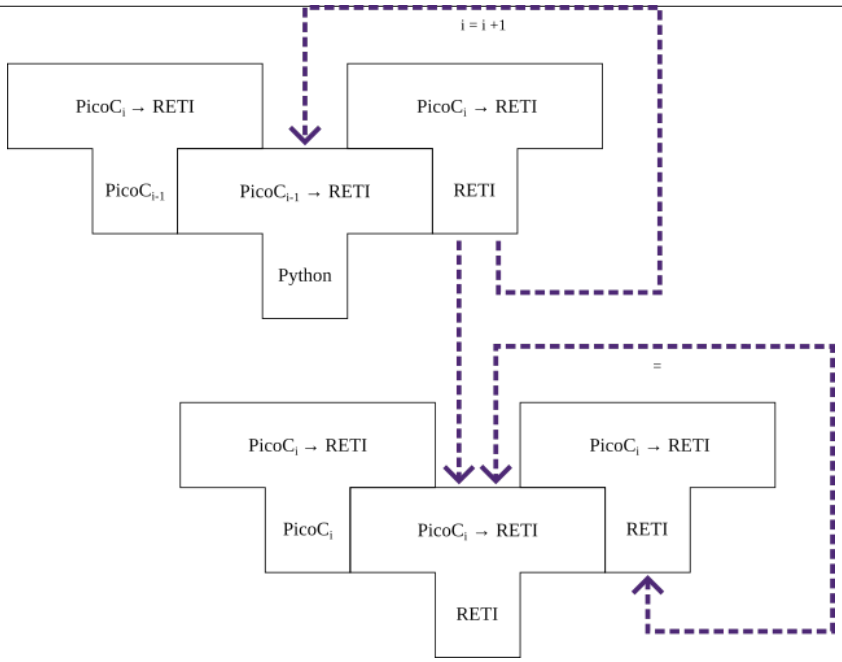


Abbildung 4.2: Iteratives Bootstrapping

Auch wenn ein **Self-compiling Compiler** $C_{w_i}^{w_i}$ in der Subdefinition 4.4.1 selbst in einer früheren Version $L_{w_{i-1}}$ der Programmiersprache L_{w_i} geschrieben wird, wird dieser nicht mit $C_{w_i}^{w_{i-1}}$ bezeichnet, sondern mit $C_{w_i}^{w_i}$, da es bei **Self-compiling Compilern** darum geht, dass diese zwar in der Subdefinition 4.4.1 eine frühere Version $C_{w_{i-1}}^{w_{i-1}}$ nutzen, um sich selbst kompilieren zu lassen, aber sie auch in der Lage sind sich selber zu kompilieren.

A Appendix

A.1 Konkrette und Abstrakte Syntax

A.2 Bedienungsanleitungen

A.2.1 PicoC-Compiler

A.2.2 Showmode

A.2.3 Entwicklertools

Literatur

Online

- *C Operator Precedence - cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27. 04. 2022).
- *Compiler Design - Phases of Compiler*. URL: https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm (besucht am 19. 06. 2022).
- *Course Webpage for Compilers (P423, P523, E313, and E513)*. 28. Jan. 2022. URL: https://iucompilercourse.github.io/IU-Fall-2021/ (besucht am 28. 01. 2022).
- *lecture-notes-2021*. 20. Jan. 2022. URL: https://github.com/Compiler-Construction-Uni-Freiburg/lecture-notes-2021/blob/56300e6649e32f0594bbbd046a2e19351c57dd0c/material/lexical-analysis.pdf (besucht am 28. 04. 2022).
- Ljohhuh. *What is an immediate value?* 4. Apr. 2018. URL: https://reverseengineering.stackexchange.com/q/17671 (besucht am 13. 04. 2022).
- *What is Bottom-up Parsing?* URL: https://www.tutorialspoint.com/what-is-bottom-up-parsing (besucht am 22. 06. 2022).
- *What is the difference between a token and a lexeme?* NewbeDEV. URL: http://newbedev.com/what-is-the-difference-between-a-token-and-a-lexeme (besucht am 17. 06. 2022).
- *What is Top-Down Parsing?* URL: https://www.tutorialspoint.com/what-is-top-down-parsing (besucht am 22. 06. 2022).