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 4.2).

Definition 2.1: Interpreter

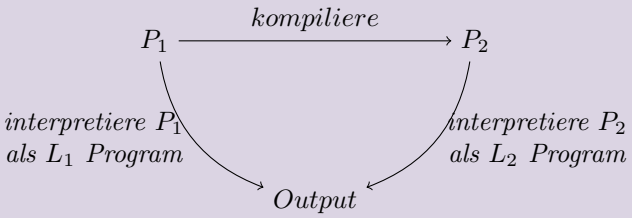
Interpretiert *die* ***Instructions*** *bzw.* ***Statements*** *eines Programmes P direkt.*

Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 2.32) 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)

Üblicherweise kompiliert ein **Compiler** von einer **komplexeren Sprache** zu **Maschinensprache**, aber es gibt z.B. auch **Transpiler** (Definition 2.3) oder **Cross-Compiler** (Definition 2.4).

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

Definition 2.3: Transpiler (bzw. Source-to-source Compiler)

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

[a]Die im aktuellen Zeitgeist in Mode gekommene Sprache **TypeScript** will als Obermenge von JavaScript und wird daher zu **JavaScript** transpiliert.

Definition 2.4: Cross-Compiler

Kompiliert ein Program für eine Sprache, die auf der eigenen Maschine garnicht läuft.

Ein **Cross-Compiler** ist entweder notwendig, wenn noch kein oder niemals ein Compiler für die Wunschsprache existiert, der unter der **Maschienensprache** L_2 einer Zielmaschine M_2 läuft oder die Zielmaschine M_2 nicht ausreichend **Rechenleistung** hat, um ein Programm in der Wunschsprache zeitnah selbst zu kompilieren.[2]

Hat die Zielmaschine M_2 allerdings ausreichend Leistung, um Programme selbst zu kompilieren und es fehlt lediglich ein Compiler, der das übernimmt, so ist eine mögliche Lösung in einem ersten einen **minimalen Compiler** zu schreiben, der die **Wunschsprache** in die **Maschienensprache** der Zielmaschine M_2 kompiliert. Diesen **minimalen Compiler** könnte man nun komplett in der Maschinensprache

und sich dafür die **Rechenleistung** einer anderen Maschine borgt und eine beliebige **Sprache**, die auf dieser Maschien läuft, in der dieser Minimale Compiler geschrieben wird.

Hat man diesen **minimalen Compiler** kann man Programme in der Wunschsprache schreiben und mithilfe des **minimalen Compilers** kompilieren, sodass diese Programme auf der Zielmaschine laufen.

Nun kann man den **minimalen Compiler**, denn man gerade eben in einer **Programmesprache** implementiert hat, in der **Wunschsprache** selbst implementieren und dann mit dem **minamlen Compiler** für ebendiese Wunschsprache selbst kompilieren. Was man als Output bekommt ist ein **minimmaler Compiler**, der aber auf der **Zielmaschine** läuft und die **Wunschsprache** in die **Maschienensprache** der Zielmaschine kompiliert.

Aufbauend auf diesem **minimalen Compiler**, der auf der **Zielmaschine** läuft, kann man nun auf der Zielmaschine selbst **iterativ** den minimalen Compiler schrittweise zu einem umfangreicheren Compiler, der mehr Funktionalitäten unterstützt weiterentwickeln und braucht die ursprüngliche Maschine, auf dem man die allererste Version des minimalen Compilers implementiert hat nicht mehr. Dieses Vorgehen wird auch als **Bootstrapping** (Definition 2.5) bezeichnet.[3]

[2]Lego Mindstorms

[3]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.

Definition 2.5: Bootstrapping

*Es gibt zwei verschiedene Formen von **Bootstrapping**:*

2.5.1: *Wenn man ein **Programm**, dass auf einer **Maschine** M_1 läuft, auch auf einer **Zielmaschine** M_2 zum laufen bringt.*

2.5.2: *Wenn man einen Compiler mithilfe von **früheren Versionen** seiner selbst schreibt. Man schreibt den erweiterten Compiler in der Sprache, welche von der früheren Version des Compilers kompiliert wird und schafft es so iterativ immer umfangreichere Compiler zu bauen.*[a]

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

Die Form von Bootstrapping in 2.5.2 bildet im Falle dessen, dass es sich beim dem Programm um einen Compiler für eine Programmiersprache handelt die Lösung für ein Problem, dass auf das **Henne-Ei-Problem**[4] reduziert werden kann.

Wenn es auf der Zielmaschine M_2 noch gar keinen Compiler für die Programmiersprache gibt, in der der Compiler für ebendiese Programmiersprache geschrieben steht, so liegt eine **zirkulare Abhängigkeit vor**, die man nur auflösen kann, indem eine **externe Entität** zur Hilfe nimmt.

Eine Möglichkeit ist, indem man einen **Cross-Compiler** auf der Maschine M_1 nutzt, der den **Compiler** der Programmiersprache für die andere Maschine M_2 kompiliert bzw. **bootstraped**.

2.1.1 T-Diagramme

Definition 2.6: T-Diagram

2.2 Grammatiken

2.3 Grundlagen

Definition 2.7: Sprache

Definition 2.8: Chromsky Hierarchie

Definition 2.9: Grammatik

[4] 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.

Definition 2.10: Reguläre Sprachen

Definition 2.11: Ableitung

Definition 2.12: Links- und Rechtsableitung

Definition 2.13: 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 2.14: Ableitungsbaum

Definition 2.15: Mehrdeutige Grammatik

2.3.2 Präzidenz und Assoziativität

Definition 2.16: Assoziativität

Definition 2.17: Präzidenz

Definition 2.18: Wortproblem

Definition 2.19: 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.

Definition 2.20: Kontextfreie Sprachen

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 2.21) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 2.21: 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 2.22) genannt.

Definition 2.22: 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 2.23). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.23: 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.

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. `Zahl` und `Bezeichner`.

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`[6]. 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[7], unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind.

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 von Variablen, Konstanten und Funktionnen** die Symbole[a].

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

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

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

Definition 2.24: Literal

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

Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 2.1 die Lexikalische Analyse an einem Beispiel veranschaulicht.

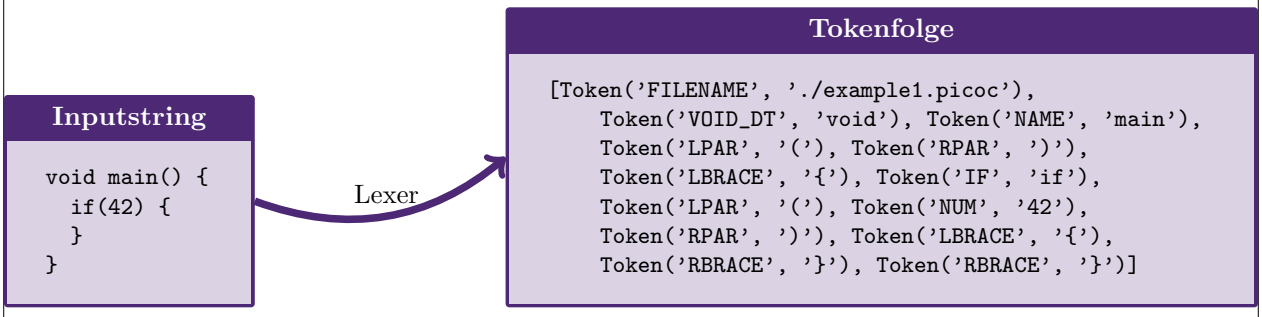


Abbildung 2.1: Veranschaulichung der Lexikalischen Analyse

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 2.25) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 2.27), ein **Derivation Tree** (Definition 2.26) generiert, der als Zwischenstufe hin zum einem **Abstrakt Syntax Tree** (Definition 2.32) 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 2.25: 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 2.26: 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 2.27: Parser

Ein Programm, dass eine ***Eingabe*** *in eine für die* ***Weiterverbeitung*** *taugliche Form bringt.*

2.27.1: *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 2.27.1 verendet.

[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 2.27 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 2.27) 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 2.28), denn ein Parser löst das **Wortproblem** (Definition 2.18) 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 2.28: 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 2.19) 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 2.13) 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 2.29) und **Visitors** (Definition 2.30) 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 2.31).

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 2.29: 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 2.30: 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 2.31: 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 2.32: 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**[8] 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** Prinzip abgearbeitet werden.

Um eine Gesamtübersicht über die **Syntaktische Analyse** zu geben, ist in Abbildung 2.2 die Syntaktische mit dem Beispiel aus Subkapitel 2.4 fortgeführt.

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

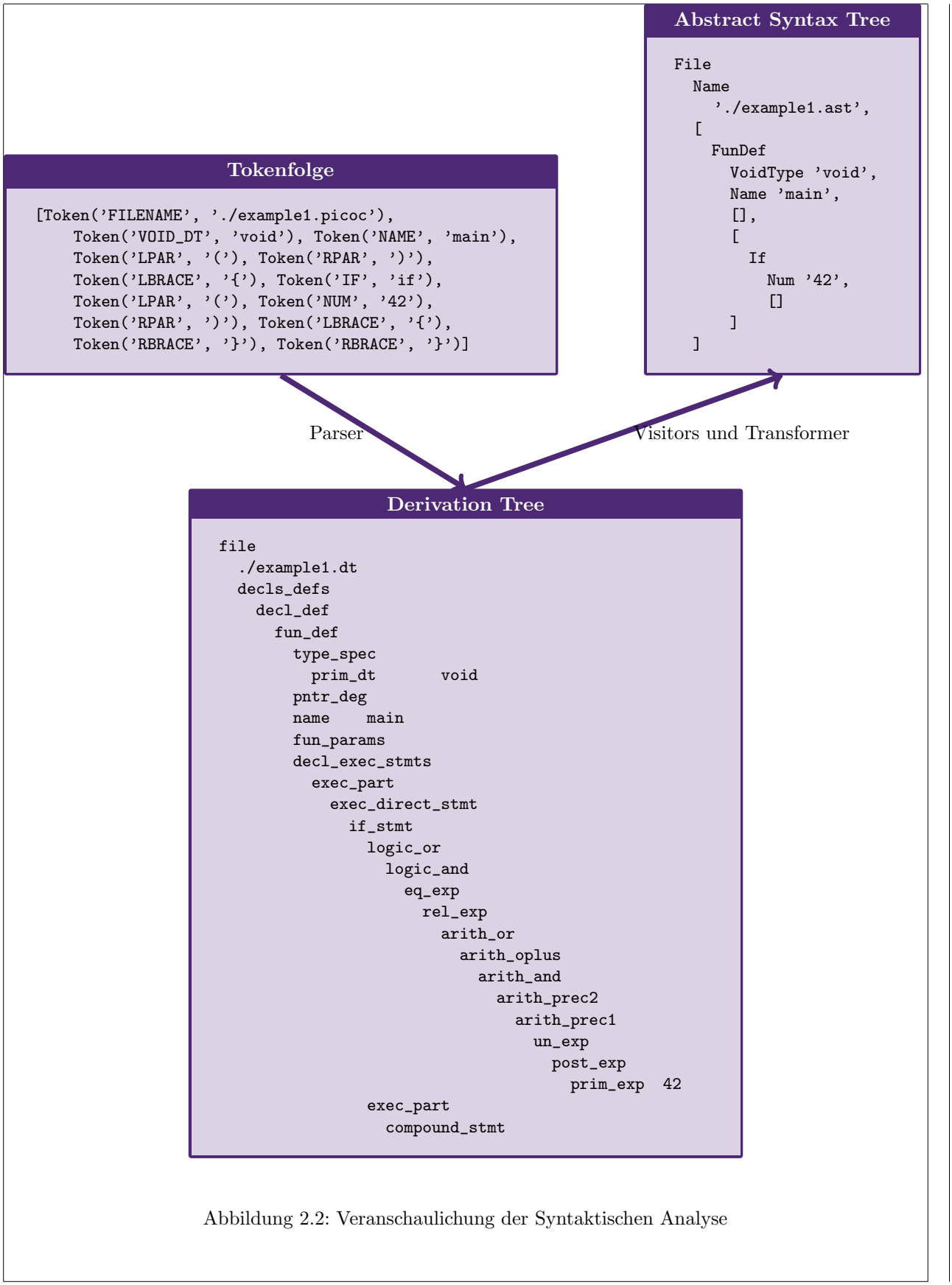


Abbildung 2.2: Veranschaulichung der Syntaktischen Analyse

2.6 Code Generierung

Definition 2.33: Pass

2.7 Fehlermeldungen

Definition 2.34: Fehlermeldung

3 Implementierung

3.1 Architektur

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 3.3.2 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 →

Tabelle 3.1: 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

ASTNode

PicoC Nodes

RETI Nodes

3.4 Code Generierung

3.4.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

Definition 3.1: Symboltabelle

RETI-Blocks Pass

RETI-Patch Pass

RETI Pass

3.4.2 Umsetzung von Pointern und Arrays

3.4.3 Umsetzung von Structs

3.4.4 Umsetzung von Funktionen

3.4.5 Umsetzung kleinerer Details

3.5 Fehlermeldungen

3.5.1 Error Handler

4 Ergebnisse und Ausblick

4.1 Funktionsumfang

4.2 Qualitätssicherung

4.3 Kommentierter Kompiliervorgang

4.4 Erweiterungsideen

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).
- *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).