Albert Ludwigs Universität Freiburg

Technische Fakultät

PicoC-Compiler

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

Bachelorarbeit

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Grammatikverzeichnis

1 Einführung

1.1 Compiler und Interpreter

Der wohl wichtigsten zu klärenden Begriffe, sind die eines **Compilers** (Definition 1.2) und eines **Interpreters** (Definition 1.1), da das Schreiben eines Compilers von der **PicoC-Sprache** L_{PicoC} in die **RETI-Sprache** L_{RETI} das Thema dieser Bachelorarbeit ist und die Definition eines **Interpreters** genutzt wird, um zu definieren was ein **Compiler** ist. Des Weiteren wurde zur **Qualitätsicherung** ein **RETI-Interpreter** implementiert, um mithilfe des **GCC**[1] und von **Tests** die **Beziehungen** in 1.2.1 zu belegen (siehe Subkapitel **??**).

Definition 1.1: Interpreter

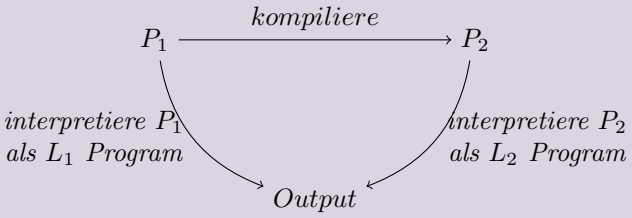
Interpretiert *die* ***Instructions*** *bzw.* ***Statements*** *eines Programmes P direkt.*

Auf die Implementierung bezogen arbeitet ein Interpreter auf den compilerinternen ***Sub-Bäumen*** *des* ***Abstract Syntax Tree*** *(Definition 1.41) und führt je nach Komposition der* ***Nodes*** *des Abstract Syntax Tree, auf die er während des Darüber-Iterierens stösst unterschiedliche Anweisungen aus.*

Definition 1.2: 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 so übersetzt wird, dass bei beiden Programmen, wenn sie von* ***Interpretern*** *ihrer jeweiligen Sprachen L_1 und L_2* ***interpretiert*** *werden, der gleiche* ***Output*** *rauskommt. Also beide Programme P_1 und P_2 die gleiche* ***Semantik*** *haben und sich nur* ***syntaktisch*** *durch die Sprachen L_1 und L_2, in denen sie geschrieben stehen unterscheiden.*



(1.2.1)

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 **??**) 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 1.3) geschrieben ist, aber es gibt z.B. auch **Transpiler** (Definition 1.9) oder **Cross-Compiler** (Definition 1.10). Des Weiteren sind **Maschienensprache** und **Assemblersprache** (Definition 1.4) voneinander zu unterscheiden.

Definition 1.3: Maschienensprache

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

[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 1.8) haben.

Definition 1.4: 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 im X_{86_64} Assembler die Instructions in **Blöcken** untergebracht, die ein **Label** haben und zu denen mittels `jmp <label>` gesprungen werden kann. Ein solches Konstrukt, was vor allem auch noch relativ beliebig wählbare Bezeichner verwendet hat keine direkte Entsprechung in einem handelsüblichen Prozessor und Hauptspeicher.

Ein **Assembler** (Definition 1.5) ist in üblichen Compilern in einer bestimmten Form meist schon integriert sein, da Compiler üblicherweise direkt **Maschienencode** bzw. **Objectcode** (Definition 1.6) erzeugen. Ein **Compiler** soll möglichst viel von seiner internen Funktionsweise und der damit verbundenen Theorie für den Benutzer abstrahieren und dem Benutzer daher standardmäßig einfach nur den Output liefern, den er in den allermeisten Fällen haben will, nämlich den **Maschienencode** bzw. **Objectcode**, der direkt ausführbar ist bzw. wenn er später mit dem **Linker** (Definition 1.7) zu Maschiendencode zusammengesetzt wird ausführbar ist.

Definition 1.5: Assembler

*Übersetzt im allgemeinen **Assemblercode**, der in **Assemblersprache** geschrieben ist zu **Maschienencode** bzw. **Objectcode** in **binärerer Repräsentation**, der in **Maschienensprache** geschrieben ist.*

Definition 1.6: Objectcode

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

Definition 1.7: 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 1.8) enthält. Für den **RETI-Interpreter** ist es ebenfalls nicht notwendig, dass der Maschienencode, denn der PicoC-Compiler generiert in binärer Darstellung ist, denn es ist für den RETI-Interpreter ebenfalls leichter diese einfach direkt in menschenlesbarer Form zu interpretieren, da der RETI-Interpreter nur die sichtbare Funktionsweise einer RETI-CPU **simulieren** soll und nicht deren mögliche interne Umsetzung[2].

Definition 1.8: 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 1.9: 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**.

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

Definition 1.10: 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}.

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

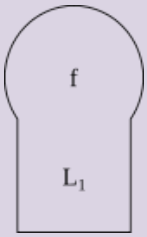
1.1.1 T-Diagramme

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 1.11), einen Übersetzer (Definition 1.12), einen Interpreter (Definition 1.13) und eine Maschiene (Definition 1.14) zusammen.

Definition 1.11: T-Diagram Programm

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



Es ist bei **T-Diagrammen** nicht notwendig beim entsprechenden **Platzhalter**, in den man die genutzte **Sprache** schreibt, den **Namen der Sprache** an ein L dranzuhängen, weil hier immer eine **Sprache** steht. Es würde in Definition 1.11 also reichen einfach eine 1 hinzuschreiben.

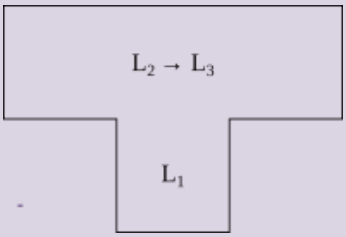
[3] Die an vielen Universitäten und Schulen eingesetzen programmierbaren Roboter von **Lego Mindstorms** nutzen z.B. einen **Cross-Compiler**, um für den programmierbaren Microcontroller eine **C-ähnliche Sprache** in die Maschienensprache des Microcontrollers zu kompilieren, da der Microcontroller selbst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

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

Definition 1.12: T-Diagram Übersetzer (bzw. eng. Translator)

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

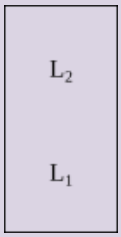
*Für den **Übersetzer** gelten genauso, wie für einen **Compiler**[a] die **Beziehungen** in 1.2.1.*



[a]Zwischen den Begriffen **Übersetzung** und **Kompilierung** gibt es einen kleinen Unterschied, **Übersetzung** ist **kleinschrittiger** als **Kompilierung** und ist auch zwischen **Passes** möglich, **Kompilierung** beinhaltet dagegen bereits alle **Passes** in einem Schritt. **Kompilieren** ist also auch **Übsersetzen**, aber **Übersetzen** ist nicht immer auch **Kompilieren**.

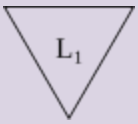
Definition 1.13: T-Diagram Interpreter

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



Definition 1.14: 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 es auch erlaubt diese Notation zu verwenden, dann handelt es sich um eine **Abstrakte Maschiene**, wie z.B. die **Python Virtual Machine** (PVM) oder **Java Virtual Machine** (JVM).

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

Sowohl **horinzontale** als auch **vertikale Adjazents** lassen sich, wie man in den Abbildungen 1.1 und 1.2 erkennen kann zusammenfassen.

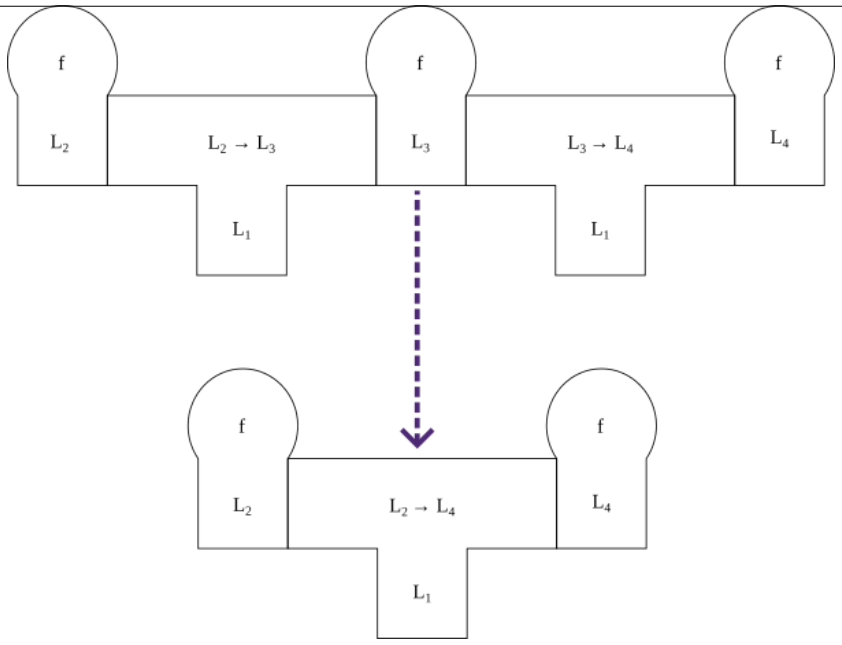


Abbildung 1.1: Horinzontale Übersetzungszwischenschritte zusammenfassen

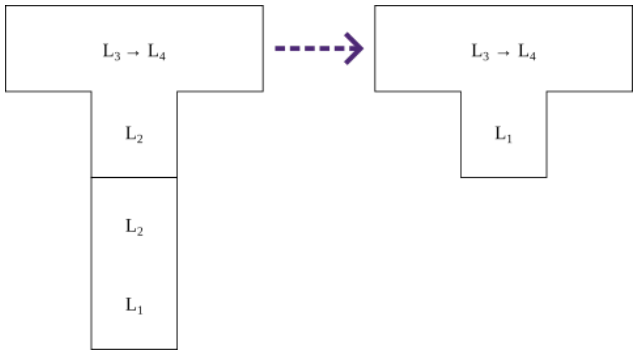


Abbildung 1.2: Vertikale Interpretierungszwischenschritte zusammenfassen

1.2 Grammatiken

1.3 Grundlagen

Definition 1.15: Sprache

Definition 1.16: Chromsky Hierarchie

Definition 1.17: Grammatik

Definition 1.18: Reguläre Sprachen

Definition 1.19: Kontextfreie Sprachen

Definition 1.20: Ableitung

Definition 1.21: Links- und Rechtsableitung

Definition 1.22: 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.*

1.3.1 Mehrdeutige Grammatiken

Definition 1.23: Ableitungsbaum

Definition 1.24: Mehrdeutige Grammatik

1.3.2 Präzidenz und Assoziativität

Definition 1.25: Assoziativität

Definition 1.26: Präzidenz

Definition 1.27: Wortproblem

Definition 1.28: 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.

1.4 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 1.29) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Diese Folgen endlicher Symoble werden auch **Lexeme** (Definition 1.30) genannt.

Definition 1.29: Pattern

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

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

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

Definition 1.31: Lexer (bzw. Scanner)

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

[a] *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 1.32) 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 2 **Symboltabelle** genannt wird.

Definition 1.32: Bezeichner (bzw. Identifier)

***Tokenwert**, der eine Konstante, Variable, Funktion usw. **eindeutig** benennt.*[a]

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

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 $\epsilon \in (N \times W)^*$ ist immer der Fall beim **Kleene Stern Operator** *. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

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

Ein **Lexeme** ist damit aber nicht immer das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann der Wert 99 durch zwei verschiedene **Literale** (Definition 1.33) dargestellt werden, einmal als ASCII-Zeichen `'c'`, dass den entsprechenden Wert in der ASCII-Tabelle hat und des Weiteren auch in Dezimalschreibweise als `99`[7]. Der **Tokenwert** ist jedoch der letztendlich verwendete Wert an sich, unabhängig von der Darstellungsform.

Die **Grammatik** G_{Lex}, die zur Beschreibung der Token T der Sprache L_{Lex} verwendet wird, ist üblicherweise

[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.

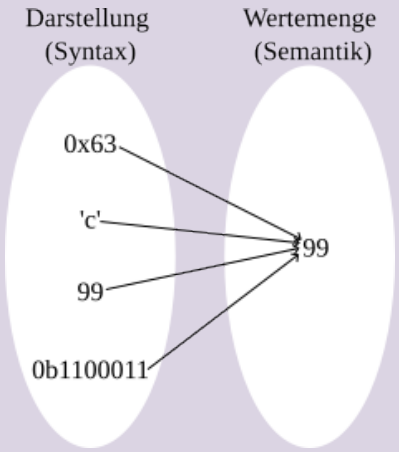
[6] Diese **Tokennamen** wurden im **PicoC-Compiler** verwendet, da man beim Programmieren möglichst **kurze** und **leicht verständliche** Bezeichner für seine Nodes haben will, damit unter anderem **mehr Code** in eine Zeile passt.

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

regulär, da ein typischer **Lexer** immer nur **ein Symbol** vorausschaut[8], sich nichts merken muss und unabhängig davon, was für Symbole davor aufgetaucht sind läuft. Die Grammatik 2.2.1 liefert den Beweis, dass die Sprache L_{PicoC_Lex} des **PicoC-Compilers** auf jeden Fall **regulär** ist, da sie die Definition 1.18 erfüllt. Einzig die Produktion `CHAR ::= "'"ASCII_CHAR"'"` sieht problematisch aus, kann allerdings auch mit regulären Produktionen ausgedrückt werden durch `{CHAR ::= "'"CHAR2, CHAR2 ::= ASCII_CHAR"'"}`.[9]

Definition 1.33: Literal

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



Um eine Gesamtübersicht über die **Lexikalische Analyse** zu geben, ist in Abbildung 1.3 die Lexikalische Analyse an einem Beispiel veranschaulicht.

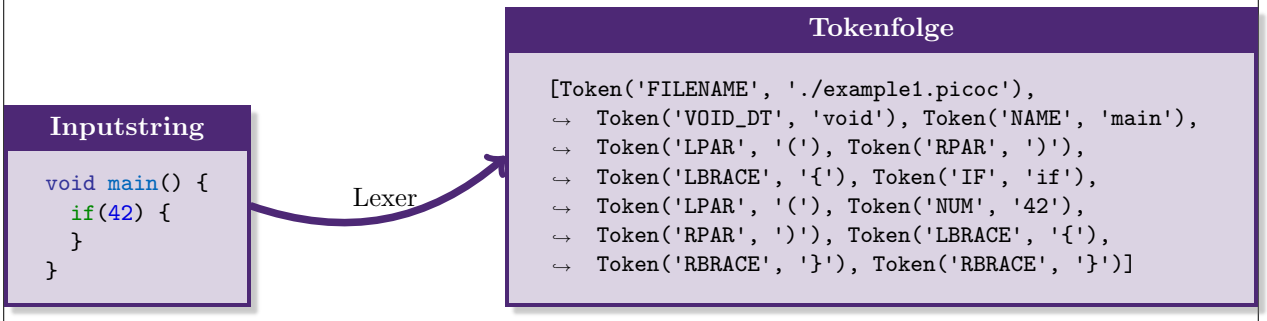


Abbildung 1.3: Veranschaulichung der Lexikalischen Analyse

1.5 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.

[8] Man nennt das auch einem **Lookahead** von 1
[9] Eine derartige Regel würde nur Probleme bereiten, wenn sich aus `ASCII_CHAR` **beliebig breite** Wörter ableiten liesen.

Die **Syntax**, in welcher der **Inputstring** aufgeschrieben ist, wird auch als **Konkrette Syntax** (Definition 1.34) bezeichnet. In einem Zwischenschritt, dem **Parsen** wird aus diesem Inputstring mithilfe eines **Parsers** (Definition 1.36), ein **Derivation Tree** (Definition 1.35) generiert, der als Zwischenstufe hin zum einem **Abstrakt Syntax Tree** (Definition 1.41) 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 1.34: 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 1.35: 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 1.36: Parser

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

1.36.1: *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 1.36.1 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 1.36 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 1.36) 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 1.6 relevant.

Ein **Parser** ist genauergesagt ein erweiterter **Recognizer** (Definition 1.37), denn ein Parser löst das **Wortproblem** (Definition 1.27) 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 1.37: 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 **Konkrete 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 1.28) 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 1.22) 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 1.38) und **Visitors** (Definition 1.39) 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 1.40).

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 1.38: 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 1.39: 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 1.40: 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 1.41: 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**[10] 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 1.4 die Syntaktische mit dem Beispiel aus Subkapitel 1.4 fortgeführt.

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

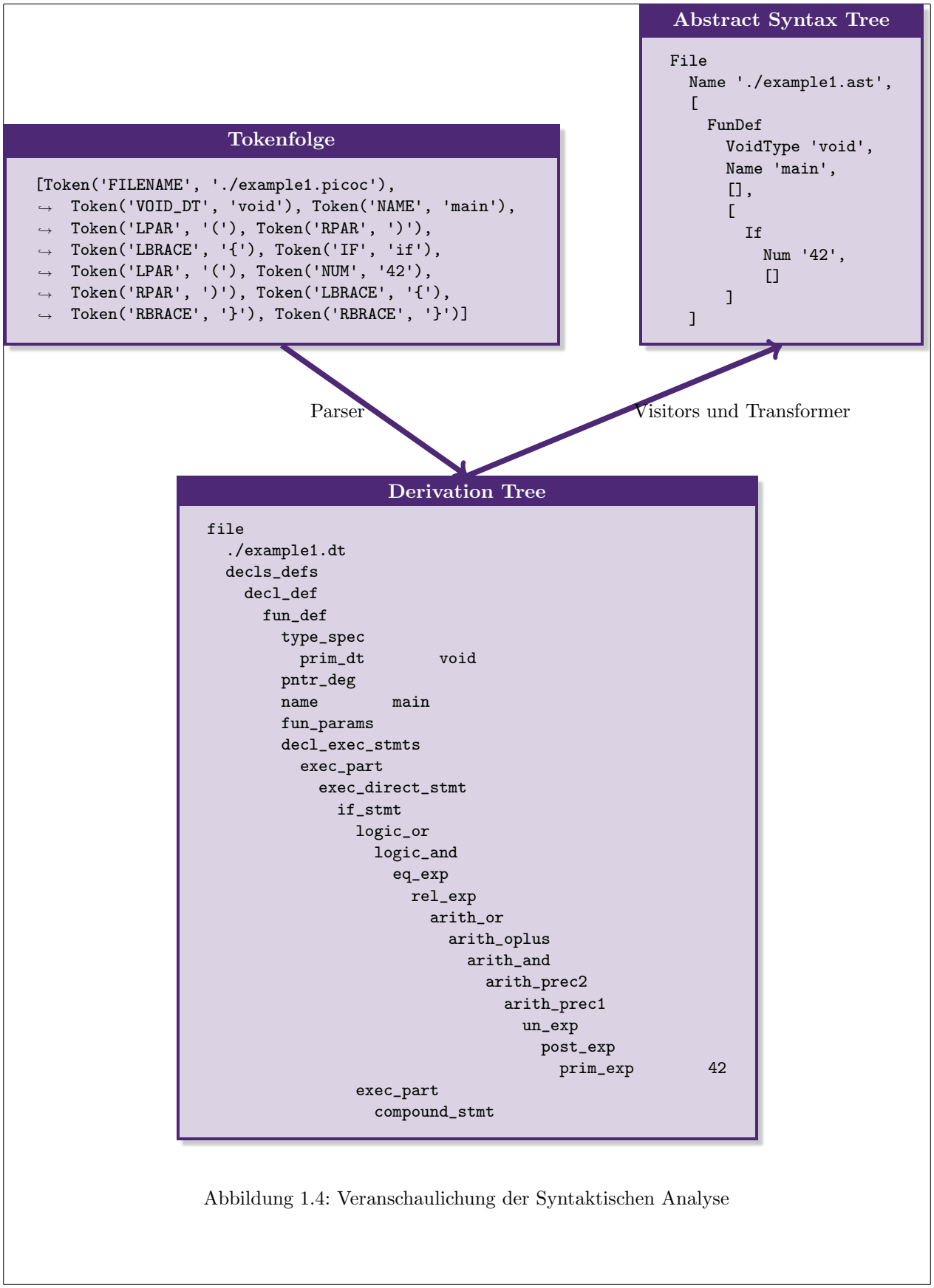


Abbildung 1.4: Veranschaulichung der Syntaktischen Analyse

1.6 Code Generierung

Definition 1.42: Pass

Definition 1.43: Monadische Normalform

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

1.7 Fehlermeldungen

Definition 1.44: Fehlermeldung

1.7.1 Kategorien von Fehlermeldungen

2 Implementierung

2.1 Architektur

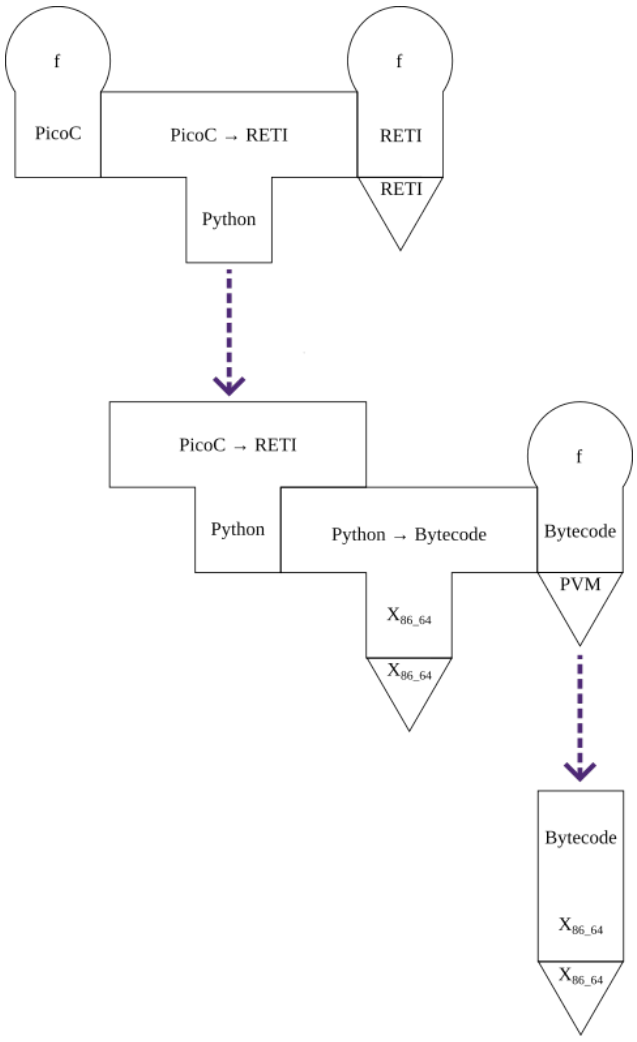


Abbildung 2.1: Cross-Compiler Kompiliervorgang ausgeschrieben

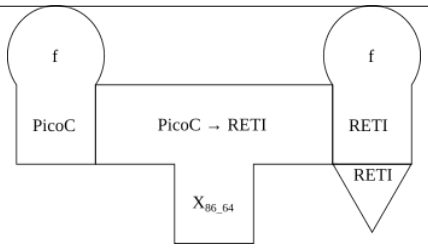


Abbildung 2.2: Cross-Compiler Kompiliervorgang Kurzform

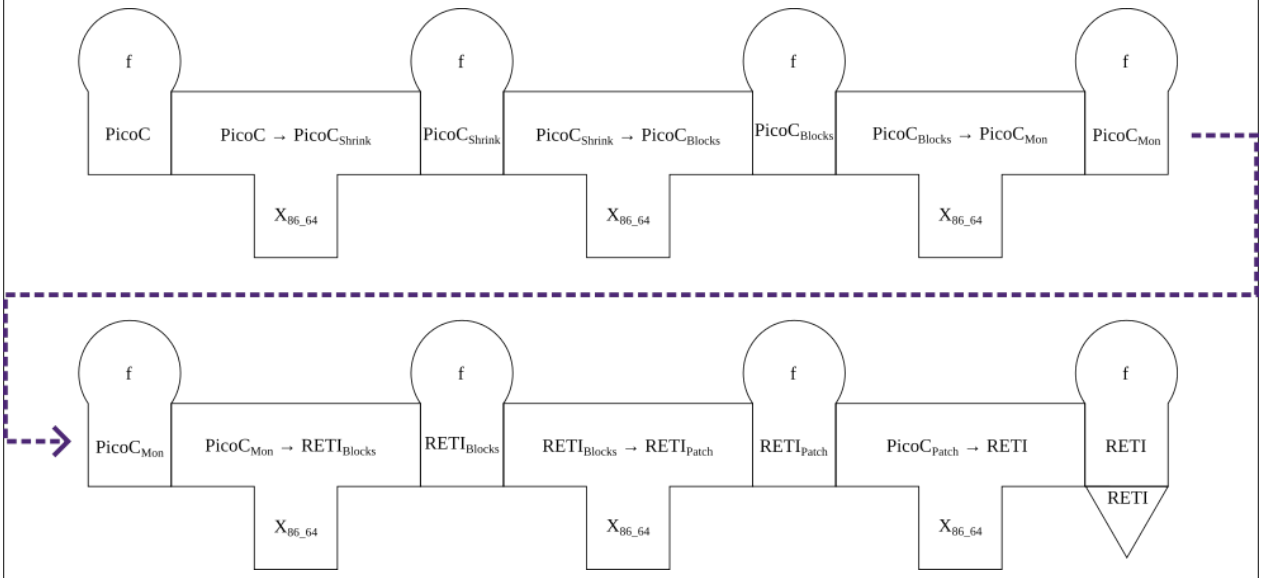


Abbildung 2.3: Architektur mit allen Passes ausgeschrieben

2.2 Lexikalische Analyse

2.2.1 Verwendung von Lark

Grammar 2.2.1: Konkrette Syntax des Lexers

2.2.2 Basic Parser

2.3 Syntaktische Analyse

2.3.1 Verwendung von Lark

In 2.3.1

prim_exp	::=	*name* | *NUM* | *CHAR* | "("*logic_or*")"	*L_Arith +*
post_exp	::=	*array_subscr* | *struct_attr* | *fun_call*	*L_Array +*
	|	*input_exp* | *print_exp* | *prim_exp*	*L_Pntr +*
un_exp	::=	*un_opun_exp* | *post_exp*	*L_Struct + L_Fun*
input_exp	::=	"*input*""("")"	*L_Arith*
print_exp	::=	"*print*""("*logic_or*")"	
*arith_prec*1	::=	*arith_prec*1 *prec*1*_op un_exp* | *un_exp*	
*arith_prec*2	::=	*arith_prec*2 *prec*2*_op arith_prec*1 | *arith_prec*1	
arith_and	::=	*arith_and* "&" *arith_prec*2 | *arith_prec*2	
arith_oplus	::=	*arith_oplus* "∧" *arith_and* | *arith_and*	
arith_or	::=	*arith_or* "|" *arith_oplus* | *arith_oplus*	
rel_exp	::=	*rel_exp rel_op arith_or* | *arith_or*	*L_Logic*
eq_exp	::=	*eq_exp eq_oprel_exp* | *rel_exp*	
logic_and	::=	*logic_and* "&&" *eq_exp* | *eq_exp*	
logic_or	::=	*logic_or* "||" *logic_and* | *logic_and*	
type_spec	::=	*prim_dt* | *struct_spec*	*L_Assign_Alloc*
alloc	::=	*type_spec pntr_decl*	
assign_stmt	::=	*un_exp* "=" *logic_or*";"	
initializer	::=	*logic_or* | *array_init* | *struct_init*	
init_stmt	::=	*alloc* "=" *initializer*";"	
const_init_stmt	::=	"*const*" *type_spec name* "=" *NUM*";"	
pntr_deg	::=	"∗"∗	*L_Pntr*
pntr_decl	::=	*pntr_deg array_decl* | *array_decl*	
array_dims	::=	("["*NUM*"]")∗	*L_Array*
array_decl	::=	*name array_dims* | "("*pntr_decl*")"*array_dims*	
array_init	::=	"{"*initializer*("," *initializer*) ∗ "}"	
array_subscr	::=	*post_exp*"["*logic_or*"]"	
struct_spec	::=	"*struct*" *name*	*L_Struct*
struct_params	::=	(*alloc*";")+	
struct_decl	::=	"*struct*" *name* "{"*struct_params*"}"	
struct_init	::=	"{""."*name*"="*initializer*("," "."*name*"="*initializer*) ∗ "}"	
struct_attr	::=	*post_exp*"."*name*	
if_stmt	::=	"*if*""("*logic_or*")" *exec_part*	*L_If_Else*
if_else_stmt	::=	"*if*""("*logic_or*")" *exec_part* "*else*" *exec_part*	
while_stmt	::=	"*while*""("*logic_or*")" *exec_part*	*L_Loop*
do_while_stmt	::=	"*do*" *exec_part* "*while*""("*logic_or*")""; "	

Grammar 2.3.1: Konkrette Syntax des Parsers, Teil 1

decl_exp_stmt	::=	*alloc*";"	*L_Stmt*
decl_direct_stmt	::=	*assign_stmt* | *init_stmt* | *const_init_stmt*	
decl_part	::=	*decl_exp_stmt* | *decl_direct_stmt* | *RETI_COMMENT*	
compound_stmt	::=	"{" *exec_part* ∗ "}"	
exec_exp_stmt	::=	*logic_or*";"	
exec_direct_stmt	::=	*if_stmt* | *if_else_stmt* | *while_stmt* | *do_while_stmt*	
	|	*assign_stmt* | *fun_return_stmt*	
exec_part	::=	*compound_stmt* | *exec_exp_stmt* | *exec_direct_stmt*	
	|	*RETI_COMMENT*	
decl_exec_stmts	::=	*decl_part* ∗ *exec_part*∗	
fun_args	::=	[*logic_or*("," *logic_or*)∗]	*L_Fun*
fun_call	::=	*name*"(" *fun_args*")"	
fun_return_stmt	::=	"*return*" [*logic_or*]";"	
fun_params	::=	[*alloc*("," *alloc*)∗]	
fun_decl	::=	*type_spec pntr_deg name*"(" *fun_params*")"	
fun_def	::=	*type_spec pntr_deg name*"(" *fun_params*")" "{" *decl_exec_stmts*"}"	
decl_def	::=	(*struct_decl* | *fun_decl*)";" | *fun_def*	*L_File*
decls_defs	::=	*decl_def*∗	
file	::=	*FILENAME decls_defs*	

Grammar 2.3.2: Konkrette Syntax des Parsers, Teil 2

2.3.2 Umsetzung von Präzidenz

Die **PicoC Sprache** hat dieselben **Präzidenzregeln** implementiert, wie die **Sprache C**[1]. Die **Präzidenzregeln** von **PicoC** sind in Tabelle 2.1 aufgelistet.

Präzidenz	Operator	Beschreibung	Assoziativität
1	`a()`	Funktionsaufruf	Links, dann rechts →
	`a[]`	Indexzugriff	
	`a.b`	Attributzugriff	
2	`-a`	Unäres Minus	Rechts, dann links ←
	`!a ~a`	Logisches NOT und Bitweise NOT	
	`*a &a`	Dereferenz und Referenz, auch Adresse-von	
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 2.1: Präzidenzregeln von PicoC

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

2.3.3 Derivation Tree Generierung

2.3.4 Early Parser

2.3.5 Derivation Tree Vereinfachung

2.3.6 Abstrakt Syntax Tree Generierung

2.3.6.1 ASTNode

2.3.6.2 PicoC Nodes

2.3.6.3 RETI Nodes

$$
\begin{array}{lcll}
T & ::= & \mathcal{V} & \textit{Variable} \\
 & | & (\mathcal{T}\ \mathcal{T}) & \textit{Application} \\
 & | & \lambda\mathcal{V} \cdot \mathcal{T} & \textit{Abstraction} \\
V & ::= & x, y, \ldots & \textit{Variables}
\end{array}
$$

Grammar 2.2.2: λ calculus syntax

$$
\begin{array}{lcllll}
A & ::= & \mathcal{T} & | & \mathcal{V} & \textit{Multiple option on a single line} \\
 & | & \mathcal{A} & & & \textit{Highlighted form} \\
 & | & \mathcal{B} & | & \mathcal{C} & \textit{Downplayed form} \\
 & | & \mathcal{A} & | & \mathcal{B} & \textit{Emphasize part of the line}
\end{array}
$$

Grammar 2.2.3: Advanced capabilities of `grammar.sty`

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).