Bachelorthesis

im Studiengang

Allgemeine Informatik

Entwicklung eines Interpreters für die Programmiersprache Monkey mit Scala Parser Combinators

Referent	:	Prof. Dr. Lothar Piepmeyer
Koreferent	:	Prof. Dr. Bernhard Hollunder
Vorgelegt am	:	30.06.2023
Vorgelegt von	:	Leon-César Steinbach
		Matrikelnummer: 268032
		steinbal@hs-furtwangen.de

Abstract

Diese Arbeit konzentriert sich auf die Entwicklung und Analyse eines Interpreters für die Programmiersprache Monkey unter Verwendung von Scala Parser Combinators. Der Schwerpunkt liegt auf dem Vergleich eines konventionellen Parsers und eines auf Scala Parser Combinators basierten Parsers in Bezug auf Leistung und Anwendbarkeit. Ein Ziel besteht darin, die grundlegenden Konzepte des Parsens und des Interpreter- und Compilerbaus ausführlich zu erläutern und dabei die Implementierung eines Interpreters zu veranschaulichen. Die Arbeit bietet einen Einblick in die Aspekte, die den Compiler effizienter gestalten. Zudem wird eine alternative Implementierung des Interpreters analysiert und verglichen. Diese Arbeit fungiert als Leitfaden für alle, die sich mit der Architektur von Parsern und Interpretern beschäftigen und Entscheidungen bezüglich Design und Implementierung treffen möchten.

This thesis focuses on the development and analysis of an interpreter for the Monkey programming language using Scala Parser Combinators. The emphasis is on comparing a conventional parser with a parser based on Scala Parser Combinators in terms of performance and applicability. An objective is to elaborate on the fundamental concepts of parsing and interpreter and compiler construction, illustrating the implementation of an interpreter in the process. The work provides insight into aspects that make the compiler more efficient. Additionally, an alternative interpreter implementation is analyzed and compared. This work serves as a guide for anyone interested in the architecture of parsers and interpreters and who wants to make decisions regarding design and implementation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ALU Arithmetic Logic Unit

AOT Ahead-Of-Time

AST Abstract Syntax Tree

BNF Backus-Naur Form

EBNF Extended Backus-Naur Form

FILO First-in-last-out

JIT Just-In-Time

JVM Java Virtual Machine

VM Virtual Machine

1 Einleitung

Diese Bachelorarbeit fokussiert sich auf die Entwicklung eines Interpreters für die Programmiersprache Monkey mit Scala Parser Combinators. Dabei werden ein konventioneller Parser und ein Parser, der auf Scala Parser Combinators basiert, miteinander verglichen. Darüber hinaus umfasst die Arbeit die Implementierung eines Interpreters sowie einen Ausblick auf die grundlegenden Konzepte, die bei der Implementierung eines Compilers und einer Virtual Machine (VM) relevant sind.

1.1 Hintergrund

Monkey ist eine einfache Programmiersprache, die von Thorsten Ball in den Büchern „Writing an Interpreter in Go“ [3] und „Writing a Compiler in Go“ [4] vorgestellt wurde. Ihre Struktur und Funktion bietet einen geeigneten Rahmen, um fundamentale Aspekte des Interpreter- und Compilerbaus zu erlernen und anzuwenden.

In dieser Arbeit wird eine solche Implementierung eines Interpreters und Compilers für Monkey entwickelt. Dabei kommt für die Entwicklung des Parsers neben einer konventionellen Implementierung die Technologie der Scala Parser Combinators zum Einsatz.

Im Kontext dieser Arbeit wird außerdem eine bestehende Implementierung eines Interpreters für Monkey untersucht und durch die in dieser Arbeit vorgestellte Implementierung optimiert.

1.2 Zielstellung

Die Hauptziele dieser Bachelorarbeit umfassen die Analyse und den Vergleich der Leistungsfähigkeit zwischen dem konventionellen Parser und dem Parser, der auf Scala Parser Combinators basiert, unter Berücksichtigung ihrer jeweiligen Vor- und Nachteile. Des Weiteren wird der Interpreter mit dem Compiler und der VM gegenübergestellt. Zusätzlich soll die Arbeit eine detaillierte Erklärung der Konzepte im Kontext des Parsens und des Baus von Interpretern und Compilern bieten. Letztlich dient die Arbeit als Handbuch zur Implementierung eines Parsers und Interpreters (jedoch nicht

eines Compilers) und soll als Entscheidungshilfe für Implementationsstrategien und Designentscheidungen in der Architektur von Parsern und Interpretern dienen.

1.3 Aufbau und Methodik der Arbeit

Die vorliegende Arbeit beginnt mit einer Erklärung der theoretischen Grundlagen. Anschließend erfolgen umfassende Beschreibungen der Implementierung der beiden Parser-Versionen - einer konventionellen Implementierung und einer Implementierung basierend auf Scala Parser Combinators. Der nächste Abschnitt befasst sich mit der Implementierung des Interpreters.

Im Anschluss daran wird ein Ausblick auf die grundlegenden Konzepte eines Compilers und einer VM gegeben. Hierbei werden insbesondere die Aspekte hervorgehoben, die zur gesteigerten Effizienz des Compilers beitragen. Zum Abschluss der Arbeit werden die entwickelten Parser und die Systeme des Interpreters und Compilers anhand verschiedener Merkmale miteinander verglichen und analysiert.

Die Arbeit setzt ein grundlegendes Verständnis der Programmiersprache Scala sowie der allgemeinen Konzepte der Informatik und von Programmiersprachen voraus. Daher richtet sich diese Arbeit an Leserinnen und Leser mit entsprechenden Vorkenntnissen. Die vertiefende Betrachtung und Anwendung der genannten Konzepte soll als Entscheidungshilfe bei der Umsetzung eigener Projekte im Bereich der Interpreter- und Compiler-Entwicklung dienen.

Für tiefergehende Einblicke und detailliertere Informationen zu den in dieser Arbeit vorgestellten Themen wird empfohlen, die Publikationen von Thorsten Ball heranzuziehen. Darüber hinaus steht die vollständige Implementierung in Scala, die im Rahmen dieser Arbeit erstellt wurde, öffentlich zur Verfügung. Sie kann unter folgendem Link im GitHub-Repository eingesehen und heruntergeladen werden: https://github.com/LeonSteinbach/MonkeyLang-Scala. Diese zusätzlichen Ressourcen können zur Vertiefung des Verständnisses und als Referenz für eigene Projekte im Bereich Interpreter- und Compilerbau dienen.

2 Theoretische Grundlagen

2.1 Interpreter- und Compilerbau

Das Entwickeln eines Interpreters oder Compilers ist ein mehrstufiger Prozess, der mehrere Schritte der Datenverarbeitung beinhaltet. Im Kern handelt es sich dabei um die Übersetzung des menschenlesbaren Quellcodes in eine Form, die von einem Computer effizient ausgeführt werden kann. Die typische Reihenfolge der Datenverarbeitung ist die folgende:

Zunächst wird der Quellcode durch einen Lexer (siehe Unterabschnitt 3.3.3) analysiert. Der Lexer liest den Eingabetext Zeichen für Zeichen und gruppiert diese in sogenannte Tokens, die die kleinsten syntaktischen Einheiten des Quellcodes darstellen (siehe Unterabschnitt 3.3.2). Tokens können verschiedene Kategorien repräsentieren, wie zum Beispiel Identifikatoren, Schlüsselwörter, Zahlen, Zeichenketten oder Symbole.

Nachdem der Quellcode in Tokens übersetzt wurde, kommt der Parser zum Einsatz (siehe Abschnitt 3.3). Dieser liest die Tokens und erstellt basierend auf den Beziehungen zwischen ihnen den Abstract Syntax Tree (AST) (siehe Unterabschnitt 2.1.1). Der AST ist eine hierarchische Struktur, die die syntaktischen Beziehungen zwischen den verschiedenen Teilen des Programms repräsentiert.

Im nächsten Schritt wird der AST entweder von einem Interpreter (siehe Unterabschnitt 2.1.2) oder einem Compiler (siehe Unterabschnitt 2.1.3) verarbeitet. Ein Interpreter liest den AST und führt die im Code definierten Aktionen unmittelbar aus, indem er die entsprechenden internen Objekte erzeugt und manipuliert.

Im Gegensatz dazu erzeugt ein Compiler aus dem AST einen Bytecode oder maschinennahen Code, der dann von der Zielplattform ausgeführt werden kann. Dieser Prozess beinhaltet oft eine Reihe von Optimierungen, um den erzeugten Code so effizient wie möglich zu machen.

Schließlich, wenn ein Bytecode-Compiler verwendet wurde, wird der erzeugte Bytecode in eine VM geladen, die die Ausführung des Programms übernimmt. Die VM übersetzt den Bytecode in interne Objekte und führt die im Code definierten Aktionen aus.

Zusammenfassend lässt sich sagen, dass der Prozess des Compilerbaus oder Interpre-

terbaus eine Reihe von Schritten beinhaltet, die darauf abzielen, den menschenlesbaren Quellcode in eine Form zu übersetzen, die von einem Computer ausgeführt werden kann. Jeder Schritt in diesem Prozess trägt dazu bei, die Struktur und Bedeutung des Quellcodes zu analysieren und in eine effizient ausführbare Form zu übersetzen.

2.1.1 Abstract Syntax Tree

Der AST, oder abstrakter Syntaxbaum, ist eine grundlegende Datenstruktur im Interpreter- und Compilerbau und spielt eine wesentliche Rolle bei der Verarbeitung von Programmiersprachen. Er stellt eine hierarchische Repräsentation der Programmstruktur dar und spiegelt sowohl Syntax als auch Semantik des Quellcodes wider.

Ein AST wird durch das Parsen des Quellcodes erstellt. Dabei wird der Quellcode zunächst durch den Lexer in seine Bestandteile zerlegt (Tokenisierung) und anschließend vom Parser in eine Form umgewandelt, die die hierarchische Struktur des Programms widerspiegelt. Jeder Knoten in dieser Baumstruktur repräsentiert dabei einen Teil des Quellcodes. Die Knoten können verschiedene Typen haben, abhängig von der Art der Struktur, die sie repräsentieren. Beispielsweise könnte ein Knoten eine Variablendeklaration, eine Funktion, einen Funktionsaufruf, eine Bedingung (wie ein if-Statement) oder eine Schleife (wie ein for- oder while-Statement) repräsentieren.

Der Begriff „Abstrakt“ im Abstract Syntax Tree bezieht sich auf seine Funktion als vereinfachte Darstellung des Quellcodes. Im Gegensatz zum Quellcode, der alle syntaktischen Details der Programmiersprache – wie Semikolons, Klammern oder bestimmte Schlüsselwörter – umfasst, konzentriert sich der AST auf die wesentlichen syntaktischen Strukturen und Semantiken des Programms. Dies bedeutet, dass weniger relevante Details im AST weggelassen werden, wodurch er eher eine abstrakte Repräsentation als eine direkte Kopie des Quellcodes ist.

Die spezifische Art der Abstraktion in einem AST besteht darin, dass er die hierarchischen und logischen Beziehungen zwischen den verschiedenen Teilen des Programms hervorhebt und gleichzeitig weniger relevante syntaktische Elemente ausblendet. Ein Beispiel dafür ist die Darstellung von Anweisungsblöcken: Während diese in der Quelltextform häufig durch geschweifte Klammern umrahmt sind, wird im AST ein solcher Block als eine unter einem Knoten assoziierte Liste von Anweisungen dargestellt.

Ein wesentlicher Vorteil des AST ist, dass er die Struktur des Programms in einer Weise darstellt, die für die weitere Verarbeitung leicht zugänglich ist. Er ermöglicht es dem Interpreter oder Compiler, durch die aufbereitete Struktur des Programms zu navigieren, Operationen auf bestimmten Knoten auszuführen und Informationen über das Programm zu sammeln.

Im Kontext der Entwicklung eines Interpreters für die Programmiersprache Monkey wird der AST benutzt, um den Quellcode hierarchisch darzustellen und zu interpretieren. Dabei wird der Quellcode in eine interne Form überführt, die vom Interpreter anschließend traversiert werden kann. Die Baumstruktur des AST ist besonders geeignet für die Anwendung von sogenannten „Recursive-Descent-Parsern“ [5]. In der Informatik bezeichnet man einen solchen Parser, der von oben nach unten arbeitet, als „Top-Down-Parser“ [5]. Diese Art von Parser beginnt mit dem Wurzelknoten der Grammatik und navigiert dann nach unten, um die Baumstruktur aufzubauen. Ein Recursive-Descent-Parser ist eine spezielle Art von Top-Down-Parser. Er besteht aus einer Reihe von Prozeduren, die gegenseitig rekursiv sind. Jede dieser Prozeduren implementiert eines der sogenannten „Nicht-Terminale“ der Grammatik, was dazu führt, dass die Struktur des resultierenden Programms eng an die der zu erkennenden Grammatik angelehnt ist. Diese effiziente Ausnutzung der Struktur bildet die Grundlage für die Implementierung eines Recursive-Descent-Parsers in dieser Arbeit.

2.1.2 Interpreter

Ein Interpreter ist ein Programm, das den vom Parser erzeugten AST einer Programmiersprache liest und die darin beschriebenen Anweisungen direkt ausführt. Im Gegensatz zu Compilern, die den gesamten Quellcode in Maschinencode oder Bytecode übersetzen, bevor das Programm ausgeführt wird, führt der Interpreter den Code Schritt für Schritt aus, indem er jede Anweisung liest, analysiert und dann ausführt.

Ein sogenannter „Tree-Walking“ Interpreter traversiert in der Ausführungsphase durch den AST, beginnend bei dem Wurzelknoten und folgt dabei den Verzweigungen zu den Kindknoten. Dieser Prozess ermöglicht es, die arithmetische Logik des Programms in der richtigen Reihenfolge zu interpretieren und auszuführen. Das kann das Auswerten von Ausdrücken und Anweisungen oder das Aufrufen von Funktionen umfassen.

Obwohl Interpreter im Vergleich zu Compilern oft langsamer in der Ausführung sind, bieten sie bestimmte Vorteile. Dazu gehören eine schnellere Entwicklungszeit, da keine Kompilierung erforderlich ist, und eine dynamischere Ausführungsumgebung, die Funktionen wie dynamisches Laden von Code und interaktive Debugging- und Testmöglichkeiten unterstützt (siehe Abschnitt 5.3).

2.1.3 Compiler

Ein Compiler ist ein spezielles Programm, das Quellcode in einer bestimmten Hochsprache in eine andere Form umwandelt, in der Regel in eine niedrigere Sprache wie Maschinencode oder Bytecode. Im Gegensatz zu einem Interpreter analysiert und

übersetzt der Compiler den gesamten Quellcode in einem zusätzlichen Schritt, bevor das Programm ausgeführt wird.

Es gibt verschiedene Arten von Compilern, darunter die oft verwendeten Ahead-Of-Time (AOT)-Compiler und Just-In-Time (JIT)-Compiler:

- AOT-Compiler: Im Kontext der Kompilierung bezieht sich der Begriff „Ahead-Of-Time“ auf den Zeitpunkt der Kompilierung. Bei der AOT-Kompilierung erfolgt die Umwandlung des durch den Programmierer erstellten Codes in Maschinen- oder Bytecode vor der Ausführung des Programms. Dieser Ansatz erlaubt es, zur Laufzeit Leistungsvorteile durch vorherige Optimierungen zu erzielen. AOT-Compiler eignen sich zur Kompilierung höherer Programmiersprachen wie C++ in Maschinencode, aber auch für Zwischendarstellungen wie Java-Bytecode [6].
- JIT-Compiler: JIT-Compiler unterscheiden sich von AOT-Compilern hauptsächlich durch den Zeitpunkt der Kompilierung. Bei JIT-Compilern erfolgt die Umwandlung des Quellcodes in Maschinencode während der Laufzeit des Programms, oftmals inkrementell oder „Just-In-Time“. Das erlaubt dem Compiler, auf die aktuelle Ausführungsumgebung des Programms zu reagieren. Es können außerdem Optimierungen vorgenommen werden wie beispielsweise die Eliminierung von Endrekursionen (Tail Recursion Elimination), die Inline-Ausführung von Methoden (Inlining) und die Reduzierung und Inversion von Schleifen (Loop Reduction and Inversion) [6]. Viele solcher Optimierungen können auch von AOT-Compilern vorgenommen werden. Allerdings sind diese auf Informationen angewiesen, die zur Kompilierzeit zur Verfügung stehen.

Ein besonderer Compiler-Typ ist der sogenannte Bytecode-Compiler. Dieser übersetzt den Quellcode nicht in Maschinencode, sondern in Bytecode. Dieser Prozess kann entweder Ahead-Of-Time oder Just-In-Time durchgeführt werden. Ein Bytecode ist eine Zwischenrepräsentation, die in der Regel von einer virtuellen Maschine ausgeführt wird. Ein Beispiel für einen Bytecode-Compiler ist der javac-Compiler für Java, der den Quellcode in Bytecode übersetzt, der dann von der Java Virtual Machine (JVM) ausgeführt wird. Der Vorteil von Bytecode besteht darin, dass er im Gegensatz zu nativem Maschinencode plattformunabhängig ist, was bedeutet, dass er auf jeder Plattform ausgeführt werden kann, die eine entsprechende VM besitzt.

Die Bytecode-Compiler können weiterhin in zwei Hauptkategorien eingeteilt werden, basierend auf der Art und Weise, wie sie den Bytecode für die Ausführung optimieren: Stack-basierte und Register-basierte Compiler.

Stack-basierte Bytecode-Compiler, wie die meisten Java-Compiler, generieren einen Bytecode, der auf einem Stack ausgeführt wird. In diesem Modell werden die Ope-

randen von Ausdrücken auf einen Stack gelegt und Operationen (wie Addition oder Multiplikation) arbeiten anschließend mit den obersten Elementen dieses Stacks. Nachdem die Operation abgeschlossen ist, werden die Operanden vom Stack entfernt und das berechnete Ergebnis auf den Stack gelegt.

Im Gegensatz dazu erzeugen Register-basierte Bytecode-Compiler einen Bytecode, der ähnlich wie Maschinencode funktioniert, indem er virtuelle Register für das Speichern von Zwischenergebnissen verwendet. Ein Beispiel hierfür ist der offizielle Compiler für die Programmiersprache Lua. In diesem Modell werden die Operanden in den Registern abgelegt und sämtliche Operationen interagieren unmittelbar mit diesen Registern. Das Ergebnis einer Operation kann dann in einem anderen Register gespeichert werden.

Jedes Modell hat seine eigenen Vor- und Nachteile. Stack-basierte Modelle sind oft einfacher zu implementieren und der generierte Bytecode ist in der Regel kompakter. Register-basierte Modelle können jedoch effizienter sein, da sie in einigen Fällen weniger Speicherzugriffe erfordern [7].

2.2 EBNF (Extended Backus-Naur Form)

Die Extended Backus-Naur Form (EBNF), ist eine Metasprache, die zur präzisen Beschreibung kontextfreier Grammatiken verwendet wird. In diesem Zusammenhang kennzeichnet „kontextfrei“ eine Grammatik, deren Regeln unabhängig von ihrem Kontext angewendet werden können. Das bedeutet, dass die linke Seite einer Produktionsregel stets nur aus einer einzigen „Nicht-Terminalen“ besteht [8].

Die EBNF ist eine Erweiterung der Backus-Naur Form (BNF), die in den 1960er Jahren von John Backus und Peter Naur entwickelt wurde, um die Syntax der Algol-60-Programmiersprache zu definieren [9]. EBNF fügt der BNF mehrere zusätzliche Konstrukte hinzu, um die Beschreibung von Sprachsyntaxen flexibler und ausdrucksstärker zu gestalten.

EBNF-Grammatiken bestehen aus Regeln, die definieren, wie ein bestimmtes Symbol (ein „nicht-terminales“ Symbol) in eine Kombination anderer Symbole umgewandelt wird. Diese anderen Symbole können entweder „terminale“ oder andere „nicht-terminale“ Symbole sein. Terminale Symbole sind die grundlegenden Bausteine der Sprache, ähnlich wie Wörter oder Buchstaben in der menschlichen Sprache. Nicht-terminale Symbole hingegen repräsentieren komplexere Strukturen, die durch die Regeln der Grammatik definiert werden. So können mit den Regeln der EBNF-Grammatik größere Strukturen aus den grundlegenden Bausteinen der Sprache aufgebaut werden [9].

2.3 Die Programmiersprache Monkey

Die Programmiersprache Monkey wurde in den Werken „Writing An Interpreter In Go“ [3] und „Writing A Compiler In Go“ [4] von Thorsten Ball vorgestellt. Es gibt keine offizielle Implementierung von Monkey, stattdessen wird sie in diesen Büchern behandelt und es liegt an den Leserinnen und Lesern, sie selbst zu implementieren.

In Monkey gibt es Integer, Booleans, Strings, Arrays, Hashmaps, arithmetische Ausdrücke und durch den Benutzer definierte, sowie integrierte Funktionen. Außerdem unterstützt Monkey Bedingungsanweisungen, implizite und explizite Rückgaben und rekursive Funktionen. Ein weiteres Merkmal von Monkey sind Closures, die freie Variablen nutzen können (siehe Unterabschnitt 3.4.2).

Im Rahmen dieser Arbeit wurde die Syntax von Monkey durch eine EBNF-Grammatik definiert. Diese beschreibt die Hierarchie und Beziehungen zwischen verschiedenen Sprachkonstrukten. Diese Grammatik stellt sicher, dass die Ausdrücke in der richtigen Reihenfolge ausgewertet werden, entsprechend den Regeln der Arithmetik.

Diese Besonderheiten machen Monkey interessant für das Erlernen der Grundlagen des Interpreter- und Compilerbaus. Trotz ihrer Einfachheit beinhaltet die Sprache eine Reihe von modernen Sprachmerkmalen. Obwohl sie nicht für umfangreiche Anwendungen konzipiert ist, dient Monkey dem Zweck, die grundlegenden Prinzipien und Techniken bei der Entwicklung von Interpretern und Compilern zu vermitteln.

```
// Integers and arithmetic expressions...
let version = 1 + (50 / 2) - (8 * 3);

// ... and strings
let name = "The Monkey programming language";

// ... booleans
let isMonkeyFastNow = true;

// ... arrays and hash maps
let people = [{"name": "Anna", "age": 24}, {"name": "Bob", "age": 99}];
```

Listing 2.1: Datentypen von Monkey [1]

```
let fibonacci = fn(x) {
  if (x == 0) {
    0
  } else {
    if (x == 1) {
      return 1;
    } else {
      fibonacci(x - 1) + fibonacci(x - 2);
    }
  }
};
```

Listing 2.2: Bedingungen, Implizite- und explizite Returns und Funktionen [1]

```
// `newAdder` returns a closure that makes use of the free variables `a` and `b`:
let newAdder = fn(a, b) {
    fn(c) { a + b + c };
};
// This constructs a new `adder` function:
let adder = newAdder(1, 2);

adder(8); // => 11
```

Listing 2.3: Closures in Monkey [1]

2.4 Kombinatoren

Kombinatoren sind ein Konzept in der mathematischen Logik und haben eine signifikante Relevanz in der Theorie der funktionalen Programmierung. Sie wurden ursprünglich 1920 von dem Mathematiker Moses Schönfinkel eingeführt. Sein Forschungsinteresse galt dem Eliminieren von Variablen und dem Aufbau von Funktionen durch den Einsatz von Kombinatoren [10].

Um die Funktionsweise der kombinatorischen Logik zu verdeutlichen, wird der B-Kombinator als Beispiel herangezogen. Dieser ist definiert als $(\mathbf{B}fgx) = f(g(x))$ für alle Funktionen f und g und alle Werte x [10]. Dieser Kombinator repräsentiert das Konzept der Funktionskomposition: Er nimmt zwei Funktionen f und g und einen Wert x an und gibt das Ergebnis der Anwendung von f auf das Ergebnis der Anwendung von g auf x zurück.

Die kombinatorische Logik ist eine Untermenge der mathematischen Logik, die sich auf die Untersuchung von Kombinatoren und deren Eigenschaften konzentriert. Sie ist eng mit der funktionalen Programmierung verwandt, da sich beide Konzepte auf die Verwendung von Funktionen als „First-Class-Citizens“ und auf die Vermeidung von Zuständen und Mutationen konzentrieren. In diesem Sinne können Kombinatoren als grundlegende Bausteine in funktionalen Programmiersprachen angesehen werden, die dazu verwendet werden können, komplexere Funktionen und Strukturen zu konstruieren, ohne auf Variablen oder Zustände angewiesen zu sein.

In der Informatik spielen Kombinatoren eine wichtige Rolle in der Theorie der funktionalen Programmierung und der kombinatorischen Logik. Ein Kombinator ist in diesem Kontext eine Funktion höherer Ordnung, die ausschließlich auf Funktionsanwendungen und zuvor definierte Kombinatoren zurückgreift, um ein Resultat aus ihren Argumenten zu generieren. Dieses Konzept veranschaulicht die Vorteile des funktionalen Programmieransatzes und dessen Fähigkeit, übersichtlichen und präzisen Code zu produzieren.

Ein weiterer wichtiger Anwendungsbereich von Kombinatoren in der Informatik ist die

Parsertheorie. Parser, die nach den Prinzipien der kombinatorischen Logik konstruiert sind, nutzen die natürlichen Präzedenzregeln, die aus der Anwendung und Komposition von Funktionen resultieren. Das ermöglicht eine elegante und effiziente Umsetzung von Syntaxanalysen und bietet eine robuste Methode zur Behandlung von Ausdruckspräzedenzen in Programmiersprachen. Durch Verwendung von Kombinatoren bei der Konstruktion von Parsern erübrigt sich die Notwendigkeit, explizite Präzedenzregeln zu definieren. Stattdessen entsteht die Präzedenz implizit aus der Art und Weise, wie die kombinatorischen Funktionen angewendet und zusammengesetzt werden.

3 Implementierung

In diesem Kapitel wird die konkrete Implementierung der Programmiersprache Monkey vorgestellt, die auf der im nächsten Abschnitt definierten EBNF-Grammatik basiert. Die Darstellung und Ausarbeitung der Implementierungsdetails, sowie der praktischen Anwendung der Sprache, stehen dabei im Vordergrund. Der Fokus liegt auf der Übersetzung der syntaktischen Struktur von Monkey und ihrer Transformation in ausführbaren Code. Die ausführliche Darstellung der angewandten Implementierungsstrategien und Techniken ermöglicht ein tiefergehendes Verständnis für den Aufbau und die Arbeitsweise von Programmiersprachen. Dieses Kapitel stellt somit einen wesentlichen Teil der vorliegenden Arbeit dar.

3.1 Definition der EBNF-Grammatik für Monkey

Obwohl in den von Thorsten Ball veröffentlichten Büchern keine offizielle EBNF-Grammatik oder Definition für die Programmiersprache Monkey bereitgestellt wird, hat diese Arbeit den Anspruch, eine solche Grammatik zu definieren. Die hier präsentierte EBNF-Grammatik für Monkey definiert eine Reihe von Regeln, die die Syntax der Programmiersprache bestimmen. Die EBNF-Grammatik stellt sicher, dass die Reihenfolge der Auswertung die Regeln der Arithmetik und Logik respektiert und umfasst neben den Kernelementen der Sprache auch erweiterte Datenstrukturen, wie Strings, Arrays und Hashmaps.

3.1.1 Formulierung der Grammatikregeln

Die Grammatikregeln der EBNF werden anschließend ausführlich formuliert und dienen als Basis zur Analyse und Erzeugung von Monkey Programmen. Sie beginnen mit dem höchsten Abstraktionsgrad, dem Programm `<program>`, welches aus einer Liste von Anweisungen `<statement-list>` besteht. Die Anweisungen können entweder eine Variablenzuweisung `<let-statement>`, eine Rückgabeanweisung `<return-statement>` oder eine Ausdrucksanweisung `<expression-statement>` sein.

Die Regeln für Ausdrücke `<expression>` sind in aufsteigender Priorität de-

finiert, um die korrekte Präzedenz der Operatoren sicherzustellen. Die Regel `<primary-expression>` dient als grundlegende Baueinheit für Ausdrücke und kann mehrere verschiedene Ausdrücke enthalten, darunter bedingte Ausdrücke `<if-expression>`, Funktionen `<function>`, gruppierte Ausdrücke `<grouped-expression>`, Identifikatoren `<identifier>` und Werte `<value>`.

Die Werte können weiter unterteilt werden in ganze Zahlen `<int>`, Boolsche Ausdrücke `<bool>`, Strings `<string>`, Arrays `<array>` und Hashmaps `<hash>`.

Im Folgenden wird die vollständige EBNF-Grammatik für Monkey aufgelistet, die in dieser Arbeit definiert wurde. Für eine übersichtlichere Darstellung wird „statement“ mit „stmt“ und „expression“ mit „expr“ abgekürzt.

```
<program>             ::= <stmt-list>
<stmt-list>           ::= { <stmt> }
<stmt>                ::= <let-stmt>
                        | <return-stmt>
                        | <expr-stmt>
<let-stmt>            ::= "let" <identifier> "=" <expr> ";"
<return-stmt>         ::= "return" <expr> ";"
<expr-stmt>           ::= <expr> ";"
<block-stmt>          ::= "{" <stmt-list> "}"

<expr>                ::= <equality-expr>
<equality-expr>       ::= <comparative-expr> {("==" | "!=") <comparative-expr>}
<comparative-expr>    ::= <additive-expr> {("<" | ">") <additive-expr>}
<additive-expr>       ::= <multiplicative-expr> {("+" | "-") <multiplicative-expr>}
<multiplicative-expr> ::= <prefix-expr> {("*" | "/") <prefix-expr>}
<prefix-expression>   ::= ("-" | "!") <prefix-expr>
                        | <postfix-expr>
<postfix-expr>        ::= <primary-expr> {<call-postfix> | <index-postfix>}
<call-postfix>        ::= "(" [<expr-list>] ")"
<index-postfix>       ::= "[" <expr> "]"
<primary-expr>        ::= <grouped-expr>
                        | <if-expr>
                        | <function>
                        | <identifier>
                        | <value>

<prefix-expr>         ::= ("-" | "!") <expr>
<grouped-expr>      ::= "(" <expr> ")"
<if-expr>             ::= "if" "(" <expr> ")" <block-stmt> ["else" <block-stmt>]
<function>            ::= "fn" "(" [<parameter-list>] ")" <block-stmt>
<identifier>          ::= <alpha> { <alpha> | <digit> | "_" }
<value>               ::= <int>
                        | <bool>
                        | <string>
                        | <array>
                        | <hash>

<int>                 ::= <digit> { <digit> }
<digit>               ::= "0..9"
<alpha>               ::= "a..zA..Z"
<bool>                ::= "true" | "false"
<string>              ::= """ { <~any valid non-quotation-marks character> } """
```

```
<array>               ::= "[" [<expr-list>] "]"
<hash>                ::= "{" [<key-value-pairs>] "}"
<key-value-pairs>     ::= <expr> ":" <expr> { "," <expr> ":" <expr> }

<expr-list>           ::= <expr> { "," <expr> }
<parameter-list>      ::= <identifier> { "," <identifier> }
```

Listing 3.1: Definition der EBNF-Grammatik von Monkey

3.1.2 Validierung der Grammatik durch Testfälle

Um die Validität der definierten Grammatik sowie ihre korrekte Funktionsweise zu gewährleisten, wurden mehrere Testfälle erstellt. Die Testfälle umfassen verschiedene Syntaxkonstruktionen und Ausdrucksformen, die in Monkey möglich sind. Jeder Testfall besteht aus einem Beispielcode, der analysiert wird, sowie dem erwarteten AST, der aus der Analyse resultieren sollte.

Die Testfälle sind so konzipiert, dass sie ein breites Spektrum an Funktionalitäten abdecken, darunter mathematische Berechnungen, Logikoperationen, die Verwendung von eingebauten Funktionen, bedingte Anweisungen und Funktionen. Bei korrekter Implementierung sollte der Parser in der Lage sein, alle Testfälle korrekt zu verarbeiten und den erwarteten AST zu erzeugen.

Folgend wird ein spezifischer Testfall für die Bedingungsanweisungen vorgestellt. Dieses Beispiel soll veranschaulichen, wie der Testaufbau erfolgt und wie die korrekte Umsetzung der Grammatikregeln überprüft wird.

```
1  test("parser.ifExpression") {
2    val program = parser.parse("if (a > 0) { a; }; if (a < b) { } else { b; };")
3    assert(program ===
4      Program(List(
5        ExpressionStatement(IfExpression(
6          InfixExpression(">", Identifier("a"), IntegerLiteral(0)),
7          BlockStatement(List(ExpressionStatement(Identifier("a")))),
8          BlockStatement(List()))),
9        ExpressionStatement(IfExpression(
10         InfixExpression("<", Identifier("a"), Identifier("b")),
11         BlockStatement(List()),
12         BlockStatement(List(ExpressionStatement(Identifier("b")))))))))
13 }
```

Listing 3.2: Test des Parsers für Bedingungsanweisungen

Eine umfassende Sammlung aller Testfälle für sämtliche Softwarekomponenten des Projekts ist im erwähnten GitHub-Repository verfügbar und kann dort eingesehen werden.

3.2 Übersicht über die Implementierung

Dieser Abschnitt bietet einen Überblick über den Prozess der Implementierung aller benötigten Komponenten für Monkey. Es wird der gesamte Datenfluss im Interpreter- und Compilerbau, von Lexer über Parser zu Interpreter und Compiler und schließlich zur virtuellen Maschine, detailliert dargestellt. Dabei wird besonderes Augenmerk auf die Analyse der Vor- und Nachteile von Parser-Kombinatoren im Vergleich zu einer konventionellen Implementierung des Parsers gelegt. Dieser Vergleich bietet Erkenntnisse über die Wirksamkeit und Effizienz der verschiedenen Methoden zur Implementierung eines Parsers.

3.2.1 Ziel der Implementierung

Das primäre Ziel der folgenden Implementierung der Systeme besteht in der Untersuchung und Dokumentation des Prozesses zur Implementierung eines Parsers und Interpreters für Monkey. Hierbei liegt der Schwerpunkt nicht alleine auf der technischen Implementierung, sondern ebenso auf der strategischen Herangehensweise. Ein wichtiger Aspekt dieses Vorgehens ist die Identifizierung potenzieller Probleme und Herausforderungen, die während der Implementierung auftreten können, und die Entwicklung geeigneter Lösungsstrategien.

Darüber hinaus soll dieser Teil als praktischer Leitfaden sowie theoretisches Nachschlagewerk fungieren. Er bietet eine detaillierte Beschreibung des gesamten Prozesses, von der Anfangsplanung bis hin zur endgültigen Ausführung der Implementierung, wobei besonderer Wert auf die Darstellung und Erläuterung der Schlüsselkonzepte und -techniken gelegt wird.

Ein zusätzliches Ziel, das im Laufe der Arbeit hinzukam, war die Erweiterung des Projekts über die ursprüngliche Aufgabenstellung hinaus. Angesichts der Tatsache, dass die Implementierung des Parsers und des Interpreters schneller abgeschlossen werden konnte als erwartet, wurde entschieden, zusätzlich einen Compiler und eine virtuelle Maschine zu entwickeln. Dies bot die Möglichkeit, weitere Aspekte des Compilerbaus zu untersuchen und Ergebnisse zu dokumentieren sowie einen Vergleich zwischen dem Interpreter und dem Compiler zu ziehen.

3.2.2 Wahl der Programmiersprache Scala

Die Wahl von Scala als Programmiersprache für dieses Projekt ist das Ergebnis einer Abwägung mehrerer Faktoren. Zu Beginn sei festgehalten, dass die Richtlinien der akademischen Betreuung die Nutzung von Scala vorsahen. Diese Vorgabe war jedoch

in keinster Weise einschränkend, da Scala mehrere Eigenschaften aufweist, die sie für dieses spezifische Projekt als besonders geeignet auszeichnen. In erster Linie stellt Scala durch die Unterstützung für funktionale Programmierstile eine besonders geeignete Umgebung für die Implementierung von kombinatorischen Parsern dar. Diese Funktionalität ermöglicht einen effizienten und übersichtlichen Code, der die inhärente Kombinatorik dieser Art von Parsern widerspiegelt.

Des Weiteren verfügt Scala über ein Paket namens „Scala Parser Combinators“. Dieses stellt eine umfangreiche Schnittstelle zur Verfügung, die speziell auf die Implementierung von Parsern zugeschnitten ist. Durch die Nutzung dieser Funktionen konnten im Rahmen dieser Arbeit effektive und flexible Parser erstellt werden.

Außerdem erlaubt Scala eine effiziente Konstruktion einer konventionell implementierten Version eines Parsers ohne die Verwendung von Scala Parser Combinators. Dieser Ansatz orientiert sich eng an den Techniken, die in den Büchern von Thorsten Ball vorgestellt werden. Dadurch konnte eine effektive Vergleichsbasis zu der Parser Combinator-basierten Implementierung geschaffen werden.

3.3 Entwicklung des Parsers

3.3.1 Überblick über die Architektur

In diesem Abschnitt wird die Entwicklung zweier Arten von Parsern vorgestellt: Eine konventionelle Implementierung, die sich an den Werken von Thorsten Ball orientiert. Bei der zweiten Art wird der Parser mithilfe des Pakets „Scala Parser Combinators“ entwickelt.

Die Architektur der beiden Parserarten ist in beiden Fällen ähnlich. Im Gegensatz zur konventionellen Implementierung muss allerdings bei der Verwendung von Scala Parser Combinators kein separater Lexer programmiert werden. Dieser wird in den kombinatorischen Parsern implizit durch Verwendung von „Regulären Ausdrücken“ [11] integriert.

Da der Parsing-Prozess im Grunde eine Transformation von Daten ist, kann dieser Prozess als Datenflussdiagramm dargestellt werden:

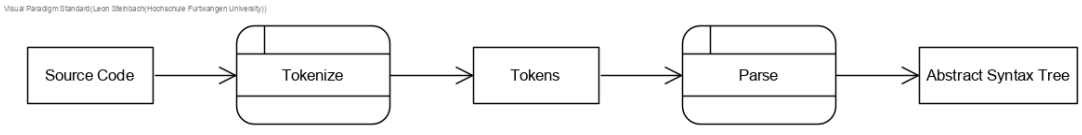

Abbildung 1: Datenflussdiagramm eines konventionellen Parsers

An erster Stelle steht der Quelltext, also der vom Benutzer eingegebene und zu übersetzende Programmcode. Der erste Transformationsprozess ist, diesen in Textform vorliegenden Programmcode in eine interne Datenstruktur zu überführen. Diese Datenstruktur soll dieselben Informationen beinhalten wie die ursprüngliche Textform, jedoch hat sie den Vorteil, dass sie für nachfolgende Verarbeitungsschritte besser geeignet ist.

Diese interne Datenstruktur wird als Tokenstrom oder einfach nur Tokens bezeichnet und stellt das Ergebnis des Lexing- oder Tokenisierungsprozesses dar. Der Lexer nimmt die rohe Eingabe, analysiert sie und zerlegt sie in diese Tokens. Jedes Token repräsentiert eine atomare Komponente des Codes, wie zum Beispiel eine Variable, einen Operator oder eine Konstante.

Die Tokenisierung erleichtert den nächsten Verarbeitungsschritt, das Parsing. Der Parser liest die Tokens und erzeugt daraus den AST. Dieser stellt die Hierarchie der verschiedenen Elemente des Quellcodes dar und ist daher von grundlegender Bedeutung für das Verständnis und die Interpretation des Codes.

Folgend wird dargestellt, wie dieser Datenfluss am Beispiel einer Variablenzuweisung und eines arithmetischen Ausdrucks aussieht:

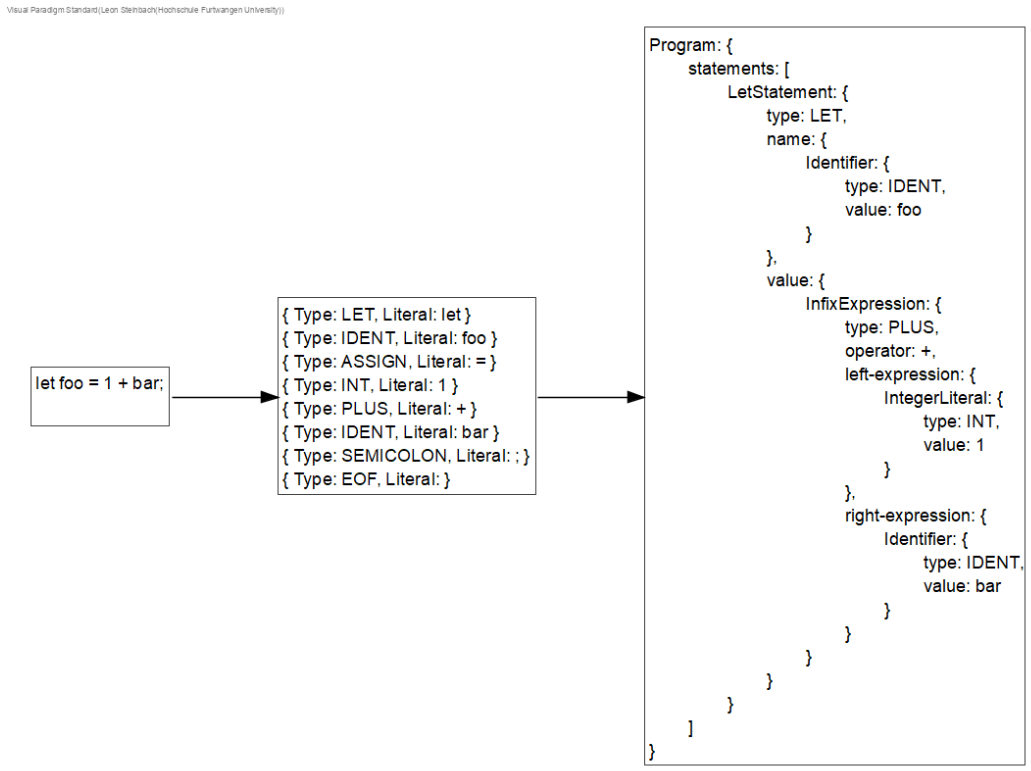

Abbildung 2: Beispiel für Datenfluss eines konventionellen Parsers

Hier wird außerdem die in dieser Arbeit gewählte Datenstruktur der Tokens und des AST sichtbar:

- **Tokens** haben einen Typ und ein „Literal“, welches die tatsächliche Zeichenfolge repräsentiert. Jede logisch zusammenhängende Zeichenfolge aus dem Quelltext wird in ein entsprechendes Token transformiert. Beispielsweise wird die Zeichenfolge `foo` als das Token `{ Type: IDENT, Literal: foo }` dargestellt.
- Der **AST** ist eine hierarchische Darstellung der Tokens als Baumstruktur. Beispielsweise wird die Zeichenfolge `1 + bar` als `InfixExpression` dargestellt, welche ein bestimmter Typ eines Ausdrucks ist. Diese besitzt einen beliebigen linken und rechten Ausdruck, sowie einen Operator, der die beiden Ausdrücke verbindet.

3.3.2 Repräsentation der Tokens

Der Lexer wird in der Implementierung dieser Arbeit nur für die konventionelle Art des Parsers benötigt, um den Quelltext in Tokens zu transformieren. Für die Erstellung von Tokens wird zunächst eine Menge an gültigen Tokentypen definiert:

```
1 enum TokenType {
2     case ILLEGAL, EOF, IDENT, INT, ASSIGN, PLUS, MINUS, BANG, ASTERIX, SLASH, LT, GT,
       COMMA, COLON, SEMICOLON, LPAREN, RPAREN, LBRACE, RBRACE, LBRACKET, RBRACKET,
      STRING, TRUE, FALSE, IF, ELSE, RETURN, EQ, NEQ, FUNCTION, LET
3 }
```

Listing 3.3: Definition von gültigen Tokentypen

Anzumerken sind die besonderen Typen `ILLEGAL` und `EOF`:

- `ILLEGAL`: Alle Tokentypen, die nicht in der als gültig definierten Menge vorhanden sind, werden als Token vom Typ `ILLEGAL` dargestellt.
- `EOF`: Das Ende des Textes wird mit dem Token vom Typ `EOF` dargestellt („End of file“)

Mit dieser Menge an Tokentypen kann der Lexer Tokens generieren, welche folgende Struktur aufweisen:

```
case class Token(tokenType: TokenType, literal: String)
```

Jedes Token hält neben dem Tokentyp zusätzlich die Information über die eigentliche Zeichenfolge. Das ist beispielsweise notwendig für die Tokens vom Typ `IDENT` und `INT`, dessen Literale nicht immer aus den gleichen Zeichenfolgen bestehen, anders als bei einem `PLUS` welches immer das selbe Zeichen repräsentiert.

Die Schlüsselwörter von Monkey (`FUNCTION`, `LET`, `TRUE`, `FALSE`, `IF`, `ELSE` und `RETURN`) werden in einer Hashmap ihren zugehörigen Zeichenfolgen zugeordnet. So

kann über die Methode `lookupIdent` vom Lexer festgestellt werden, ob es sich bei einem Textabschnitt um ein Schlüsselwort von Monkey oder um einen frei definierten Identifikator handelt:

```
object Token {
    val keywords: Map[String, TokenType] = Map(
        "fn" -> TokenType.FUNCTION,
        "let" -> TokenType.LET,
        "true" -> TokenType.TRUE,
        "false" -> TokenType.FALSE,
        "if" -> TokenType.IF,
        "else" -> TokenType.ELSE,
        "return" -> TokenType.RETURN
    )

    def lookupIdent(ident: String): TokenType = keywords.getOrElse(ident, TokenType.
    IDENT)
}
```

Listing 3.4: Definition der Schlüsselwörter von Monkey

3.3.3 Implementierung des Lexers

Der Lexer hat die Aufgabe, die Tokens aus dem Quelltext heraus zu generieren. Falls in dieser lexikalischen Analyse syntaktische Fehler erkannt werden, gibt der Lexer eine Fehlermeldung zurück und unterbricht das Programm. Fehler, die vom Lexer erkannt werden, beinhalten:

- Ungültige Zeichen: Wenn der Lexer auf ein Zeichen trifft, das in der Programmiersprache nicht definiert ist, wird dies als Fehler erkannt. Zum Beispiel könnte ein Sonderzeichen, das nicht als Teil eines gültigen Tokens definiert ist, einen Fehler auslösen.
- Fehlerhafte Tokens: Wenn eine Sequenz von Zeichen nicht zu einem gültigen Token gemäß den Regeln der Programmiersprache zusammengesetzt werden kann, erkennt der Lexer dies als Fehler. Beispielsweise würde ein ungeschlossener String (ein String ohne abschließendes Anführungszeichen) einen Fehler auslösen.

Bei der lexikalischen Analyse durchläuft der Lexer den Quelltext zeichenweise von links nach rechts und konvertiert diesen in Tokens. Dabei behält er stets die Information über das aktuell verarbeitete Zeichen sowie das folgende Zeichen im Quelltext. Um das nächste Zeichen einzulesen wird die Methode `readCharacter` aufgerufen:

```
def readCharacter(): Unit = {
    currentCharacter =
        if (readPosition >= input.length)
            Some(-1.toByte)
        else
```

```
6               Some(input.charAt(readPosition).toByte)
7       position = readPosition
8       readPosition += 1
9   }
```

Listing 3.5: Methode readCharacter zum Lesen des nächsten Zeichens

Um an bestimmten Stellen das nächste Zeichen lediglich temporär einzulesen (beispielsweise zur Entscheidungsfindung basierend auf einer Bedingung), wurde zusätzlich die Methode `peekCharacter` definiert. Sie verhält sich ähnlich wie `readCharacter`, allerdings ohne die Position zu verändern.

Der Lexer wird außerdem so implementiert, dass nicht einmalig alle Tokens generiert werden. Vielmehr funktioniert er wie ein Iterator, der jeweils bei Aufruf der Methode `nextToken` das nächste Token generiert und zurück gibt. Diese Methode wird später vom Parser wiederholt aufgerufen.

In dieser Methode wird die Position zunächst solange erhöht, bis das aktuelle Zeichen kein „Whitespace"-Zeichen ist. In Monkey werden Anweisungen (Statement) mit einem Semikolon getrennt, wodurch sämtlicher Whitespace keine weitere Relevanz hat, wie z.B. in Python. Anschließend wird das aktuelle Zeichen auf die in Monkey definierten Zeichen der Tokens überprüft. Dabei gibt es viele einfachere Fälle, in denen ein einzelnes Zeichen direkt in ein Token übersetzt werden kann.

```
1 val token = currentCharacter match {
2     ...
3     case Some('+') => Token(TokenType.PLUS, "+")
4     ...
```

Listing 3.6: Erstellung eines einfachen Tokens

Es gibt allerdings auch Fälle, in denen eine Zeichenfolge aus mehr als einem Zeichen ein Token ergeben:

```
1     ...
2     case Some('=') => getDoubleCharacterToken(TokenType.EQ, TokenType.ASSIGN, '=')
3     case Some('!') => getDoubleCharacterToken(TokenType.NEQ, TokenType.BANG, '=')
4     ...
```

Listing 3.7: Erstellung von Tokens mit mehreren Zeichen

Für solche Fälle müssen zusätzlich spezielle Methoden entwickelt werden, die überprüfen, ob zwei (oder mehrere) bestimmte Zeichen aufeinander folgen.

Zusätzlich gibt es Fälle, in denen Zeichenfolgen (zunächst) unbekannter Länge eingelesen werden sollen, wie z.B. bei `STRING` und `IDENT`.

```
    ...
    case Some('"') => Token(TokenType.STRING, readString())
    ...
    case _ => getIdentifier
}
```

Listing 3.8: Erstellung von Tokens mit beliebiger (unbekannter) Länge

Identifikatoren werden zum Schluss ausgewertet, falls kein vorheriges Zeichen im Pattern-Matching zugetroffen hat. Dabei werden solange Zeichen eingelesen, bis der Lexer auf ein Zeichen trifft, welches kein Buchstabe und keine Zahl ist. Es wird anschließend überprüft, ob es sich bei dem eingelesenen Wort um einen gültigen Identifikator handelt, also z.B. ein Wort, welches nicht mit einer Zahl beginnt. Danach wird mit Hilfe der Methode `lookupIdent` überprüft, ob es sich um ein Schlüsselwort von Monkey handelt, oder um einen frei definierten Identifikator. Trifft keiner der gültigen Fälle zu, wird ein Token vom Typ `ILLEGAL` zurückgegeben.

Der letzte spezielle Fall ist das Token `EOF`, welches das Ende des Quelltextes repräsentiert. In Scala wird das Ende eines Textes oder einer eingelesenen Datei mit `-1` dargestellt, woraus sich folgender Fall ergibt:

```
    ...
    case Some(-1) => Token(TokenType.EOF, "")
    ...
```

Listing 3.9: Erstellung des Tokens EOF

Am Schluss der Methode `nextToken` wird das nächste Zeichen gelesen und das zuvor erstellte Token zurückgegeben.

Dieser Tokenisierungs-Algorithmus transformiert den Quelltext auf eine äußerst effiziente Weise. Jedes Zeichen wird genau einmal gelesen, wodurch kein Backtracking notwendig ist. Daher hat der Algorithmus eine lineare Laufzeit, welche direkt von der Länge des Quelltextes abhängt.

3.3.4 Repräsentation der AST-Objekte

Damit der Parser später aus den Tokens einen AST erzeugen kann, müssen die Objekte, die den AST aufbauen, zunächst definiert werden. Alle diese Objekte können in zwei Knotentypen unterteilt werden, welche beide vom Basistyp `Node` erben:

- **Ausdrücke** (Expressions) sind Knoten, die einen Wert erzeugen. Zum Beispiel erzeugen die Ausdrücke `5` oder `add(1, 2)` jeweils einen Wert. Die in Monkey definierten Werte sind ganze Zahlen (Integer), Wahrheitswerte (Boolean), Zeichenfolgen (String), Listen (Array) und Hashmaps.

- **Anweisungen** (Statements) sind Knoten, die keinen Wert zurückgeben. Zum Beispiel liefert die Anweisung `let x = 1;` keinen Wert zurück, sondern weist einer Variablen einen Wert zu. Anweisungen erzeugen eher ein bestimmtes Verhalten, anstatt einen Wert zu generieren. Jede Anweisung in Monkey muss mit einem Semikolon enden.

In verschiedenen Programmiersprachen werden Anweisungen und Ausdrücke verschieden interpretiert. Beispielsweise gibt es in Monkey If-Expressions, welche anhand einer Bedingung (Ausdruck) entscheiden, welcher Codeblock (Anweisung) ausgeführt werden soll. Diese If-Expressions erzeugen neben dem oben beschriebenen Verhalten auch einen Rückgabewert, wodurch folgendes Programm gültig ist:

```
let foo = if (true) { "fizz"; } else { "buzz"; };
```

Listing 3.10: If-Expression in Monkey

Hier nimmt die Variable `foo` den Wert `fizz` an, was ein besonderes funktionales Verhalten in Monkey ist, welches sonst nur in manchen Programmiersprachen möglich ist, vor allem in funktionalen Sprachen wie Scala.

Für die Unterteilung der Knoten in Ausdrücke und Anweisungen werden Schnittstellen (Trait) definiert, welche von den jeweiligen Knoten implementiert werden:

```
sealed trait Node
sealed trait Statement extends Node
sealed trait Expression extends Node
```

Listing 3.11: Schnittstellen für Ausdrücke und Anweisungen

Im nächsten Schritt erfolgt die Definition der individuellen Knoten sowie deren zugehörige Attribute:

```
case class Program(statements: List[Statement]) extends Node
case class Identifier(name: String) extends Expression
case class IntegerLiteral(value: Int) extends Expression
case class BooleanLiteral(value: Boolean) extends Expression
case class StringLiteral(value: String) extends Expression
case class ArrayLiteral(elements: List[Expression]) extends Expression
case class HashLiteral(pairs: Map[Expression, Expression]) extends Expression
case class LetStatement(name: Identifier, value: Expression) extends Statement
case class ReturnStatement(value: Expression) extends Statement
case class BlockStatement(statements: List[Statement]) extends Statement
case class IfExpression(condition: Expression, consequence: BlockStatement,
    alternative: BlockStatement) extends Expression
case class FunctionLiteral(parameters: List[Identifier], body: BlockStatement, var
    name: String = "") extends Expression
case class CallExpression(function: Expression, arguments: List[Expression]) extends
    Expression
case class IndexExpression(left: Expression, index: Expression) extends Expression
```

```
15 case class ExpressionStatement(expression: Expression) extends Statement
16 case class PrefixExpression(operator: String, value: Expression) extends Expression
17 case class InfixExpression(operator: String, left: Expression, right: Expression)
       extends Expression
```

Listing 3.12: Definition der Knoten des AST

Der Knoten `Program` bildet in jedem Monkey-Programm den Wurzelknoten. Jedes Programm besteht also aus einer Menge von Anweisungen, die im Attribut `statements` dieses Wurzelknotens gespeichert werden. Ein alleinstehender Ausdruck ist daher kein gültiges Programm, solange dieser nicht als `ExpressionStatement` mit einem Semikolon am Ende abgeschlossen wird.

Wie bei den zuvor definierten Werte-Ausdrücken erkennbar ist, repräsentiert Monkey sämtliche Werte zur Vereinfachung durch die Werte der Host-Sprache Scala.

3.3.5 Implementierung des konventionellen Parsers

Der in den folgenden Abschnitten vorgestellte Algorithmus zum Parsen der Tokens implementiert einen sogenannten Pratt-Parser, zuerst beschrieben durch Vaughan R. Pratt im Paper „Top down operator precedence“ [12]. Dieser Algorithmus nutzt die Strategie eines „Recursive-Descent Parsers“, welche eine rekursive Traversierung des AST vom Wurzelknoten ausgehend nach unten vorsieht.

Die größte Herausforderung beim Parsen besteht darin, die korrekte Präzedenz, das heißt die verschiedenen „Bindungsstärken“ der einzelnen Tokens, sicherzustellen. Somit liegt die fundamentale Aufgabe eines Parsers nicht nur im Aufbau eines AST aus den Tokens, sondern insbesondere in der korrekten hierarchischen Verschachtelung dessen. Beispielsweise soll der Ausdruck `1 + 2 * 3` zu der Hierarchie `(1 + (2 * 3))` umgewandelt werden, wobei anstelle der Zahlen beliebig komplexe Ausdrücke stehen können. In diesem Beispiel hat der Operator `*` per Definition eine stärkere Bindung (höhere Präzedenz) als `+` und soll deshalb vorher ausgewertet werden.

Im Folgenden wird der Pratt-Parsing Algorithmus detailliert beschrieben. Zunächst wird jedem Token eine Präzedenz nach den Regeln der EBNF-Grammatik zugeordnet:

```
1  enum Precedence {
2    case LOWEST, EQUALS, LESSGREATER, SUM, PRODUCT, PREFIX, CALL, INDEX
3  }
4  ...
5  val precedences: Map[TokenType, Precedence] = Map(
6      TokenType.EQ -> Precedence.EQUALS,
7      TokenType.NEQ -> Precedence.EQUALS,
8      TokenType.LT -> Precedence.LESSGREATER,
9      TokenType.GT -> Precedence.LESSGREATER,
10     TokenType.PLUS -> Precedence.SUM,
```

```
    TokenType.MINUS -> Precedence.SUM,
    TokenType.ASTERIX -> Precedence.PRODUCT,
    TokenType.SLASH -> Precedence.PRODUCT,
    TokenType.LPAREN -> Precedence.CALL,
    TokenType.LBRACKET -> Precedence.INDEX
)
```

Listing 3.13: Zuordnung der Präzedenzen der Tokens

In einem Pratt-Parser werden spezifische Methoden für jede Art von Ausdruck und Anweisung implementiert. Diese sind jeweils für das Parsen des entsprechenden Ausdruckes oder der Anweisung verantwortlich. Alle diese Methoden erzeugen den zugehörigen Knoten für den AST und geben diesen zurück. Dabei können sich die Methoden teilweise gegenseitig oder auch selbst rekursiv aufrufen.

Zunächst wird eine Methode implementiert, die den Wurzelknoten `Program` auswertet:

```
def parseProgram: Program = {
    val statements = ListBuffer[Statement]()
    advanceTokens()

    while (currentToken.get.tokenType != TokenType.EOF) {
        parseStatement match {
            case Some(statement: Statement) => statements += statement
            case _ =>
        }
        advanceTokens()
    }
    Program(statements.toList)
}
```

Listing 3.14: Parser für Program-Knoten

Ein Programm setzt sich aus mehreren Anweisungen zusammen. Daher wird die Methode zum Parsen von Anweisungen solange aufgerufen, bis das Ende des Quelltextes erreicht ist. Danach wird ein `Program`-Objekt mit den ausgewerteten Anweisungen erzeugt und zurückgegeben. Die Methode `advanceTokens` interargiert mit dem Lexer und ruft dessen `nextToken`-Methode auf, speichert das resultierende Token als „nächstes Token“ und ersetzt anschließend das „aktuelle Token“ durch dieses.

Die Methode zum Parsen von Anweisungen gibt das Ergebnis der jeweiligen Methode zurück, die für das Parsen der bestimmten Art von Anweisung zuständig ist:

```
def parseStatement: Option[Statement] = {
    currentToken.get.tokenType match {
        case TokenType.LET => parseLetStatement
        case TokenType.RETURN => parseReturnStatement
        case _ => parseExpressionStatement
    }
}
```

Listing 3.15: Parser für Anweisungen

Beispielsweise wird eine `Return`-Anweisung dann wie folgt ausgewertet:

```
def parseReturnStatement: Option[ReturnStatement] = {
    advanceTokens()
    val value: Expression = parseExpression(Precedence.LOWEST) match {
        case Some(expression: Expression) => Some(expression).get
        case None => return None
    }
    if (peekToken.get.tokenType == TokenType.SEMICOLON)
        advanceTokens()
    Some(ReturnStatement(value))
}
```

Listing 3.16: Parser für Return-Anweisungen

Die Methode `parseReturnStatement` stellt ein typisches Beispiel für die Implementierung der meisten Parser-Methoden dar. In diesem Prozess wird zuerst das nächste Token abgerufen. Anschließend erfolgt die sequenzielle Ausführung der entsprechenden Parser-Methoden, welche den in der EBNF-Grammatik definierten Ausdruck oder die Anweisung konstruieren. Zuletzt wird das Objekt des AST zurückgegeben.

In diesem Beispiel besteht die Return-Anweisung aus einem Ausdruck, gefolgt von einem Semikolon. Die Auswertung des Ausdrucks erfolgt durch Aufruf der Methode `parseExpression`, die den Kernmechanismus des Pratt-Parsers darstellt. Die Implementierung und Erläuterung dieser Methode erfolgen in den folgenden Abschnitten.

```
def parseExpression(precedence: Precedence): Option[Expression] = {
    val parsePrefix = prefixParseFunctions.get(currentToken.get.tokenType) match {
        case Some(function) => function
        case _ =>
            noPrefixParseFunctionError(currentToken.get.tokenType)
            return None
    }
    var leftExpression: Expression = parsePrefix() match {
        case Some(expression: Expression) => Some(expression).get
        case _ => return None
    }

    while (peekToken.isDefined && peekToken.get.tokenType != TokenType.SEMICOLON &&
    precedence.ordinal < peekPrecedence.ordinal) {
        val parseInfix = infixParseFunctions.get(peekToken.get.tokenType) match {
            case Some(function) => function
            case _ => return Some(leftExpression)
        }
        advanceTokens()
        leftExpression = parseInfix(Some(leftExpression).get) match {
            case Some(expression: Expression) => Some(expression).get
            case _ => return None
        }
    }
    Some(leftExpression)
}
```

Listing 3.17: Allgemeiner Parser für Ausdrücke

Der Kern dieses Algorithmus basiert auf der Idee, spezifische Parser-Methoden mit verschiedenen Tokentypen zu verknüpfen. Sobald ein Token identifiziert wird, wird die entsprechende Parser-Methode für seinen Typ ausgeführt, um ein entsprechendes AST-Objekt zurückzugeben. Jeder Tokentyp kann dabei bis zu zwei zugehörige Methoden haben, die jeweils von seiner Position im Ausdruck abhängen - speziell ob das Token in einer Präfix- oder Infix-Position steht.

```
val prefixParseFunctions: Map[TokenType, () => Option[Node]] = Map(
    TokenType.IDENT -> (() => parseIdentifier: Option[Node]),
    TokenType.INT -> (() => parseIntegerLiteral: Option[Node]),
    TokenType.TRUE -> (() => parseBooleanLiteral: Option[Node]),
    TokenType.FALSE -> (() => parseBooleanLiteral: Option[Node]),
    TokenType.STRING -> (() => parseStringLiteral: Option[Node]),
    TokenType.LBRACKET -> (() => parseArrayLiteral: Option[Node]),
    TokenType.LBRACE -> (() => parseHashLiteral: Option[Node]),
    TokenType.BANG -> (() => parsePrefixExpression: Option[Node]),
    TokenType.MINUS -> (() => parsePrefixExpression: Option[Node]),
    TokenType.LPAREN -> (() => parseGroupedExpression: Option[Node]),
    TokenType.IF -> (() => parseIfExpression: Option[Node]),
    TokenType.FUNCTION -> (() => parseFunctionLiteral: Option[Node]),
)

val infixParseFunctions: Map[TokenType, Expression => Option[Node]] = Map(
    TokenType.LPAREN -> (leftExpr => parseCallExpr(leftExpr): Option[Node]),
    TokenType.LBRACKET -> (leftExpr => parseIndexExpr(leftExpr): Option[Node]),
    TokenType.PLUS -> (leftExpr => parseInfixExpr(leftExpr): Option[Node]),
    TokenType.MINUS -> (leftExpr => parseInfixExpr(leftExpr): Option[Node]),
    TokenType.ASTERIX -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
    TokenType.SLASH -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
    TokenType.EQ -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
    TokenType.NEQ -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
    TokenType.LT -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
    TokenType.GT -> (leftExpr => parseinfixExpr(leftExpr): Option[Node]),
)
```

Listing 3.18: Zuordnung der Tokentypen mit Präfix- und Infix-Parsermethoden

Als Beispiel wird der Tokentyp `MINUS` herangezogen: Falls das Token in einer Präfix-Position steht (z.B. beim Ausdruck `-5`) wird das Token mit der Präfix-Methode `parsePrefixExpression` ausgewertet. Steht es in einer Infix-Position (z.B. beim Ausdruck `foo - 3`) wird die Infix-Methode zum Parsen verwendet. Dabei wird, wie bei allen anderen Infix-Methoden, der bereits zuvor ausgewertete linke Ausdruck als Argument übergeben (`leftExpr`).

Eine weitere Besonderheit ist das Definieren der Tokentypen `LPAREN` und `LBRACKET` (also die Zeichen `(` und `[`) innerhalb der `infixParserFunctions`. Dies sind die Tokens, die eine `callExpression` bzw. eine `indexExpression` einleiten, welche wiederum grundsätzlich keine typischen Infix-Ausdrücke sind. Sie werden dennoch wie solche behandelt, da die Syntax ähnlich ist: Auf der linken und rechten Seite steht ein Ausdruck und dazwischen ein fest definiertes Zeichen. Der einzige Unterschied

ist, dass anders als bei „normalen“ Infix-Ausdrücken ein zusätzliches, fest definiertes Zeichen nach dem rechtsseitigen Ausdruck folgt (`)` bzw. `]`). Um diesen Sonderfall zu behandeln, werden den beiden Tokenytypen separate Methoden zum Parsen zugewiesen.

Innerhalb der zuvor gezeigten Methode `parseExpression` wird als Erstes der linke Ausdruck mithilfe der zugehörigen Präfix-Methode ausgewertet. Das Ergebnis dieser Auswertung wird der Variablen `leftExpression` zugewiesen.

Anschließend setzt eine `while`-Schleife die Auswertung der rechtsseitigen Ausdrücke fort, sofern deren Präzedenz höher als die des bereits ausgewerteten linken Ausdrucks ist. Falls es ein rechtsseitiges Token gibt, das eine stärkere Bindungskraft besitzt, wird das aktuelle Token als Teil eines Infix-Ausdrucks betrachtet. Im entgegengesetzten Fall wird es als Teil eines Präfix-Ausdrucks angesehen. Im Fall, dass es in einem Infix-Ausdruck steht, wird die jeweils passende Infix-Methode zum Parsen des rechtsseitigen Ausdrucks verwendet. Danach würde ein AST-Objekt vom Typ `InfixExpression` generiert werden, welches die Information über linken und rechten Ausdruck sowie den verbindenden Operator enthält.

Pratt-Parsing behebt das Problem der Linksrekursion, das in vielen Grammatiken und Parsing-Methoden auftritt. Linksrekursion entsteht, wenn eine syntaktische Regel sich selbst am Anfang ihrer eigenen Definition aufruft [13]. In Recursive-Descent Parsern kann dies zu einer unendlichen Rekursion führen und somit das Programm blockieren. Die zuvor definierte EBNF-Grammatik wurde bereits so aufgebaut, dass keine Linksrekursion auftritt. Zur Vermeidung der Linksrekursion im Programm muss allerdings auch die Implementierung entsprechende Anpassungen vornehmen.

Der vorgestellte Pratt-Parser umgeht dieses Problem durch sein iteratives Design. In der Praxis wird zunächst der linke Ausdruck (Präfix) ausgewertet, bevor zu den rechtsseitigen Ausdrücken (Infixen) übergegangen wird. Jeder nachfolgende Infix-Ausdruck wird nur dann ausgewertet, wenn seine Bindungsstärke höher als die des vorherigen Ausdrucks ist. Durch die Designentscheidung, das Parsen iterativ in der Reihenfolge der Bindungsstärke durchzuführen, wird effektiv eine Linksrekursion vermieden.

Dies ist ein entscheidender Vorteil des implementierten Parsers gegenüber ausschließlich rekursiven Recursive-Descent Parsern, da es eine effiziente Lösung für das Problem der Linksrekursion bietet. Der Pratt-Parser bleibt dabei flexibel genug, um eine Vielzahl von Grammatiken zu parsen, einschließlich solcher, die anderswo zu Linksrekursion führen würden.

Wenn während des Parsing-Prozesses ein syntaktischer oder semantischer Fehler erkannt wird, speichert der Parser eine entsprechende Fehlermeldung in einer Liste. Anstatt den Prozess sofort zu stoppen, fährt der Parser jedoch mit der Analyse fort.

Dies bietet den bedeutenden Vorteil, dass alle aufgetretenen Fehler in einem Durchlauf erkannt und dem Benutzer gebündelt angezeigt werden können, anstatt die Fehler einzeln und nacheinander aufzudecken. Auf diese Weise erhält der Benutzer einen umfassenden Überblick über alle Korrekturen, die vorgenommen werden müssen, was die Fehlerbehebung erheblich effizienter gestaltet.

3.3.6 Scala Parser Combinators

In den folgenden Kapiteln wird der zuvor beschriebene Pratt-Parser mithilfe des Pakets „Scala Parser Combinators“ implementiert.

Die in diesem Paket verwendeten Parser-Kombinatoren stellen im Kern eine Implementierung eines Recursive-Descent Parsers dar. Dabei wird eine Kombination von Parsern genutzt, die jeweils kleinere Segmente des Quelltextes analysieren. Die Ergebnisse dieser kleinteiligen Analysen werden dann rekursiv kombiniert, um den gesamten Text zu parsen. In Analogie zu Recursive-Descent Parsern, wo jede rekursive Prozedur eines der „Nicht-Terminale“ der Grammatik implementiert, entspricht jeder kleinere Parser innerhalb der Parser-Kombinatoren einer Regel in der Grammatik der Sprache.

Die Syntax und Funktionsweise von Scala Parser Combinators wird folgend anhand des offiziellen Beispielprogramms dargestellt:

```
1  import scala.util.parsing.combinator._
2
3  case class WordFreq(word: String, count: Int) {
4      override def toString = s"Word <$word> occurs with frequency $count"
5  }
6
7  class SimpleParser extends RegexParsers {
8      def word: Parser[String] = """[a-z]+""".r ^^ { _.toString }
9      def number: Parser[Int] = """(0|[1-9]\d*)""".r ^^ { _.toInt }
10     def freq: Parser[WordFreq] = word ~ number ^^ { case wd ~ fr => WordFreq(wd,fr) }
11 }
12
13 object TestSimpleParser extends SimpleParser {
14     def main(args: Array[String]) = {
15         parse(freq, "johnny 121") match {
16             case Success(matched,_) => println(matched)
17             case Failure(msg,_) => println(s"FAILURE: $msg")
18             case Error(msg,_) => println(s"ERROR: $msg")
19         }
20     }
21 }
```

Listing 3.19: Beispiel für einen Parser mit Scala Parser Combinators [2]

Das vorgestellte Beispielprogramm definiert einen einfachen Parser, der Eingabestrings in der Form „Wort Zahl“ in Objekte der Klasse `WordFreq` umwandelt.

Scala Parser Combinators basiert auf einer Kombination von Parsern, was bedeutet, dass kleinere Parser definiert werden, die spezifische Teile der Eingabe verarbeiten und dann zu komplexeren Parsern kombiniert werden.

In diesem Beispiel gibt es drei definierte Parser: `word`, `number` und `freq`.

- Der `word`-Parser nutzt einen regulären Ausdruck, der nur auf Kleinbuchstaben zutrifft. Nachdem er einen Teil des Textes erkannt hat, der auf den regulären Ausdruck passt, wird dieser in einen String umgewandelt und zurückgegeben.
- Ähnlich funktioniert `number`, ein Parser, der auf eine bestimmte Zahl in der Eingabe zurückgreift und diese in einen Integer umwandelt.
- `freq` ist ein *kombinierter* Parser, der zuerst auf den Parser `word` und dann auf `number` zurückgreift. Das `~`-Symbol wird verwendet, um zwei Parser in einer sequentiellen Verknüpfung zu kombinieren.

Die `^^`-Syntax ist ein Transformationsoperator, der verwendet wird, um das Ergebnis eines Parsers zu transformieren. In den beiden ersten Parsern (`word` und `number`) wird es genutzt, um die mit dem regulären Ausdruck übereinstimmenden Resultate in den gewünschten Datentyp umzuwandeln, also zu einem String bzw. Int.

In dem Parser `freq` wird der Transformationsoperator verwendet, um die Ergebnisse der beiden Parser `word` und `number` in ein `WordFreq`-Objekt umzuwandeln. Hier wird Scalas Pattern-Matching verwendet, um auf die beiden Teile des „Matches“ zugreifen zu können (`wd` und `fr`).

Die geschweiften Klammern `{}` in Scala werden in diesem Kontext zur Definition von anonymen Funktionen verwendet. Die innerhalb der geschweiften Klammern definierte Funktion wird auf das Ergebnis des Parser-Matches angewendet.

Jede Klasse, die von `RegexParsers` (oder einer anderen Parser-Art aus Scala Parser Combinators) erbt, erhält automatisch eine `parse`-Methode. Diese Methode benötigt zwei Argumente: einen in der Parser-Klasse definierten Parser und den zu analysierenden Text. Die `parse`-Methode wird dann verwendet, um den übergebenen Text mit dem spezifizierten Parser auszuwerten. Abhängig davon, ob der Parsing-Prozess erfolgreich war oder nicht, liefert sie unterschiedliche Statusmeldungen sowie das ausgewertete Ergebnis zurück. Im Beispielprogramm würde das Aufrufen der `parse`-Methode die Ausgabe `Word <johnny> occurs with frequency 121` generieren und anzeigen.

Scala Parser Combinators bietet neben den beschriebenen syntaktischen Elementen auch weitere nützliche Kurzschreibweisen, vordefinierte Parser für häufige Strukturen (JavaTokenParsers) und logische Operatoren zur Kombination von Parsern.

Bis zur Version 2.11 war das Paket Scala Parser Combinators Teil der Standardbibliothek von Scala. Danach wurde es jedoch in ein separates Paket ausgegliedert und ist nicht mehr standardmäßig enthalten. Daher muss es ab dieser Version als zusätzliche Abhängigkeit eingebunden werden. Mit dem Build-Tool sbt [14] geht das wie folgt:

```
libraryDependencies +=
    "org.scala-lang.modules" %% "scala-parser-combinators" % <version>
```

Listing 3.20: Einbinden von Scala Parser Combinators mit sbt

Es gibt für Scala noch viele weitere Pakete für die Verwendung von Parser-Kombinatoren, welche einige Probleme von Scala Parser Combinators beheben. Diese verbessern teilweise die Leistung und Flexibilität oder erweitern den Umfang [15].

3.3.7 Implementierung des Parsers mit Scala Parser Combinators

Bei der Transformation der in der EBNF-Grammatik definierten Regeln in funktionierende Parser mit Scala Parser Combinators werden die Regeln der Grammatik nahezu in ihrer gegebenen Form direkt als Parser ausdrückt. Dies erlaubt es, eine unmittelbare und intuitiv nachvollziehbare Übersetzung der Grammatik in Parsersyntax zu erreichen.

Als Beispiel wird die Regel für Return-Anweisungen betrachtet:

```
<return-statement> ::= "return" <expression> ";"
```

Die Parser-Syntax wird nahezu identisch zur ursprünglichen Grammatikregel formuliert:

```
def returnStatement: Parser[ReturnStatement] = "return" ~> expression <~ ";" ^^ {
    value => ReturnStatement(value)
}
```

Listing 3.21: Parser für Return-Anweisungen

In der oben stehenden Definition wird die Rückgabeanweisung als Zeichenkette `"return"` formuliert, gefolgt von einer Ausdrucksregel `expression` und abgeschlossen durch ein Semikolon `";"`. Der Operator `~>` sorgt dabei für die Verbindung zwischen dem Return-Literal und der Ausdrucksregel, während das abschließende Semikolon durch den Operator `<~` hinzugefügt wird. Dieser Operator, auch bekannt als „sequentielle Kombination mit Filter“, ist ein leistungsstarkes Werkzeug in den Scala Parser Combinators. Seine Funktion ähnelt dem Operator `~`, der als „sequentielle Kombination“ interpretiert wird. Allerdings bietet der Operator `~>` einen bedeutenden Vorteil: Bei der Anwendung wird lediglich der Teil des Parsers, auf den der Operator zeigt, an den Transformationsoperator `^^` weitergegeben.

Der Teil des Parsers, der nicht auf den `~>`-Operator zeigt (in dem gegebenen Beispiel `"return"` und `";"`), wird im Parservorgang berücksichtigt und ist für das erfolgreiche Pattern-Matching essentiell, jedoch wird er nicht zur Konstruktion des AST herangezogen.

Der Transformationsoperator `^^` wird anschließend genutzt, um das Ergebnis der Ausdrucksanalyse in ein `ReturnStatement`-Objekt umzuwandeln. Hierbei stellt `value` das Ergebnis des vorhergehenden Parsers dar und wird als Argument für den Konstruktor von `ReturnStatement` verwendet.

Ähnlich wie bei der Implementierung des konventionellen Parsers wird auch in dieser Version ein Parser als Einstiegspunkt definiert. Dieser wird zum Parsen des gesamten Programms verwendet und ist deshalb so definiert, damit er auf hintereinander folgende Anweisungen passt:

```
...
def statement: Parser[Statement] = letStatement | returnStatement |
    expressionStatement
def statementList: Parser[List[Statement]] = rep(statement)
def program: Parser[Program] = statementList ^^ { statements => Program(statements) }
```

Listing 3.22: Einstiegspunkt: Parser für Programme

Die Implementierung dieses Parsers zeichnet sich durch eine effiziente Wiederverwendbarkeit und kombinatorische Anwendung der einzelnen Parser aus. Beispielsweise ist der Parser für eine Liste von Anweisungen, `statementList`, als eigenständiger Parser definiert. Diese Modularität ermöglicht es, den Parser in verschiedenen Kontexten zu nutzen und die Konsistenz und Effizienz des gesamten Parsers zu steigern.

Ein spezifischer Anwendungsfall dieser Wiederverwendbarkeit ist die Integration des `statementList`-Parsers in den Parser für Blockanweisungen, `blockStatement`. Durch die Wiederverwendung des bestehenden `statementList`-Parsers wird redundanter Code vermieden, die Struktur des Programms bleibt übersichtlich und die Wartbarkeit verbessert sich.

```
def blockStatement: Parser[BlockStatement] = "{" ~> statementList <~ "}" ^^ {
    statements => BlockStatement(statements)
}
```

Listing 3.23: Parser für Block-Anweisungen

Auf diese Weise lassen sich auch komplexere Parser in nur wenigen Zeilen Code definieren. Beispielsweise wurde der Parser für Variablenzuweisungen in weniger als der Hälfte der Zeilen im Vergleich zur konventionellen Version definiert:

```
def letStatement: Parser[LetStatement] = "let" ~> identifier ~ ("=" ~> expression <~
    ";") ^^ {
    case name ~ expression => expression match {
        case function: FunctionLiteral =>
            function.name = name.name
            LetStatement(name, function)
        case expression: Expression => LetStatement(name, expression)
    }
}
```

Listing 3.24: Parser für Variablenzuweisungen

Die kompakte Syntax von Scalas Pattern-Matching erleichtert das Abfragen des Ergebnistyps des Parsers und fördert somit eine präzise Darstellung des Parsers. So kann in diesem Parser effizient überprüft werden, um welchen Typ es sich bei dem rechtsseitigen Ausdruck handelt und anhand dessen ein spezielles Verhalten hervorgerufen werden.

`integer` ist einer der Parser, welche die Werte von Monkey parsen können. Um diese Parser zu definieren, können in vielen Fällen vordefinierte Parser aus `JavaTokenParsers` sowie logische Operatoren verwendet werden. Auch in diesen Parsern werden, wie in der konventionellen Version, die Wertetypen der Host-Sprache Scala zur Darstellung der Werte von Monkey verwendet:

```
def integer: Parser[IntegerLiteral] = wholeNumber ^^ { value => IntegerLiteral(value.
    toInt) }

def boolean: Parser[BooleanLiteral] = ("true" | "false") ^^ { value => BooleanLiteral
    (value.toBoolean) }

def string: Parser[StringLiteral] = stringLiteral ^^ { value => StringLiteral(value.
    substring(1, value.length - 1) ) }

def array: Parser[ArrayLiteral] = "[" ~> repsep(expression, ",") <~ "]" ^^ { elements
     => ArrayLiteral(elements) }

def hash: Parser[HashLiteral] = "{" ~> repsep((expression <~ ":") ~ expression, ",")
    <~ "}" ^^ { keyValuePairs =>
    val pairs = keyValuePairs.map {
        case key ~ value => (key, value)
    }
    HashLiteral(pairs.toMap)
}
```

In den Parsern `integer` und `string` werden vordefinierte Parser für den jeweils passenden Anwendungsfall verwendet: `wholeNumber` für das Parsen von ganzen Zahlen und `stringLiteral` für das Parsen von Texten, welche von doppelten Anführungszeichen umschlossen sind. Außerdem werden weitere „Convenience-Methoden“ verwendet: Die Methode `repsep(<parser>, <separator>)` stellt ein Pattern-Matching dar,

das eine Sequenz von Elementen erkennt, die durch den spezifizierten Parser interpretiert werden und durch einen Separator getrennt sind.

Diese Menge an Werte-Parsern wird zusätzlich in einen generellen Parser für die weitere kombinatorische Verwendung zusammengefasst:

```
def value: Parser[Expression] = integer | boolean | string | array | hash
```

Listing 3.25: Parser für sämtliche Werte in Monkey

Die Parser Combinators von Scala ermöglichen die Implementierung von Parsersystemen, die in ihrer Lesbarkeit und Übersichtlichkeit der ursprünglichen EBNF-Grammatik nahekommen. Dies fördert nicht nur eine verbesserte Verständlichkeit und Wartbarkeit des Codes, sondern unterstützt auch bei der Fehlerbehebung und der Durchführung von Modifikationen oder Erweiterungen des Parsersystems.

3.3.8 Anwenden der Präzedenzen für Ausdrücke

Diese Ähnlichkeit zwischen dem Aufbau der EBNF-Grammatik und den Parser-Kombinatoren zeigt sich besonders bei der Implementierung der verschiedenen Präzedenzen für Ausdrücke. Diese werden in der EBNF-Grammatik wie folgt dargestellt:

```
<expr>                ::= <equality-expr>
<equality-expr>       ::= <comparative-expr> {("==" | "!=") <comparative-expr>}
<comparative-expr>    ::= <additive-expr> {("<" | ">") <additive-expr>}
<additive-expr>       ::= <multiplicative-expr> {("+" | "-") <multiplicative-expr>}
<multiplicative-expr> ::= <prefix-expr> {("*" | "/") <prefix-expr>}
<prefix-expression>   ::= ("-" | "!") <prefix-expr>
                        | <postfix-expr>
<postfix-expr>        ::= <primary-expr> {<call-postfix> | <index-postfix>}
<call-postfix>        ::= "(" [<expr-list>] ")"
<index-postfix>       ::= "[" <expr> "]"
<primary-expr>        ::= <grouped-expr>
                        | <if-expr>
                        | <function>
                        | <identifier>
                        | <value>
```

Listing 3.26: Definition der Präzedenzen für Infix-Ausdrücke

Die kombinatorische Struktur dieser Regeln kann wie folgt implementiert werden:

```
def expression: Parser[Expression] = equalityExpression

def equalityExpression: Parser[Expression] = infixExpression("==" | "!=",
    comparativeExpression)

def comparativeExpression: Parser[Expression] = infixExpression("<" | ">",
    additiveExpression)

```

```
7 def additiveExpression: Parser[Expression] = infixExpression("+" | "-",
      multiplicativeExpression)
8
9 def multiplicativeExpression: Parser[Expression] = infixExpression("*" | "/",
      prefixExpression)
```

Listing 3.27: Kombinatorische Parser für Infix-Ausdrücke und deren Präzedenz

Die Rekursionstiefe eines Parsers in der kombinierten Parser-Hierarchie bestimmt dessen Präzedenz. Das bedeutet, je tiefer ein Parser in dieser Hierarchie geschachtelt ist, desto höher ist seine Präzedenz. Dies resultiert aus der Funktionsweise des Parsers, der von der obersten Ebene startet und dann progressiv durch alle Kombinationen von Parsern navigiert, bis er einen Parser erreicht, dessen Pattern-Matching mit dem Eingabetext übereinstimmt.

Ein sogenannter „parametrisierter“ Parser `infixExpression` wird verwendet, um diese Verschachtelung zu erreichen:

```
1 def infixExpression(operatorParser: Parser[String], operandParser: Parser[Expression
      ]): Parser[Expression] =
2     operandParser ~ rep(operatorParser ~ operandParser) ^^ {
3         case expression ~ list => (list foldLeft expression) {
4             case (left, operator ~ right) => InfixExpression(operator, left, right)
5         }
6     }
```

Listing 3.28: Parametrisierter Parser für Infix-Ausdrücke

Der `infixExpression`-Parser ist ein parameterisierter Parser, da er zwei Parameter annimmt: `operatorParser` und `operandParser`.

`operatorParser` ist ein Parser, der den Operator in einem Infix-Ausdruck erkennt. Zum Beispiel könnte es ein Parser sein, der arithmetische Operatoren wie `+`, `-`, `*` und `/` erkennt.

`operandParser` ist ein Parser, der die Operanden in einem Infix-Ausdruck erkennt. Diese könnten beispielsweise numerische Werte oder andere Ausdrücke sein.

Daraus wird ein neuer Parser erstellt, der einen Operanden erkennt, gefolgt von einer beliebigen Anzahl von Operator-Operand-Paaren.

In der Transformationsmethode wird dann eine Liste von Operator-Operand-Paaren (aufgebaut durch den `rep`-Parser) zusammen mit dem ersten Operanden in eine einzelne `InfixExpression`-Instanz gefaltet. Dieser Vorgang wird durch die Methode `foldLeft` ausgeführt.

In der `foldLeft`-Operation wird jedes Operator-Operand-Paar der Liste in die Funktion eingefügt. Das Ergebnis dieser Funktion wird dann mit dem nächsten Paar in der

Liste verarbeitet und so weiter, bis alle Paare verarbeitet sind. Die Funktion selbst erstellt eine neue `InfixExpression`-Instanz mit dem aktuellen linken Ausdruck `left` (der anfangs der erste Operand ist), dem Operator `operator` und dem rechten Operanden `right`.

So wird letztendlich ein Objekt des AST erzeugt, das den gesamten Infix-Ausdruck repräsentiert. Jeder Knoten des Baums repräsentiert dabei eine einzelne Operation (bestehend aus Operator und beiden Operanden).

Die Implementation der Parser mit den höchsten Präzedenzen `prefixExpression` und `postfixExpression` sieht wie folgt aus:

```
def prefixExpression: Parser[Expression] =
    rep("-" | "!") ~ postfixExpression ^^ {
        case operators ~ expression =>
            (operators foldRight expression) {
                (operator, expression) => PrefixExpression(operator, expression)
            }
    }

def postfixExpression: Parser[Expression] =
    primaryExpression ~ rep(postfixOperator) ^^ {
        case left ~ list => (list foldLeft left) {
            case (left, (operator, right)) => operator match {
                case "(" => CallExpression(left, right.asInstanceOf[List[Expression
    ]])
                case "[" => IndexExpression(left, right.asInstanceOf[Expression])
            }
        }
    }

def postfixOperator: Parser[(String, Any)] =
    "(" ~> repsep(expression, ",") <~ ")" ^^ { args => ("(", args) } |
    "[" ~> expression <~ "]" ^^ { expr => ("[", expr) }
```

Listing 3.29: Parser für Prefix- und Postfix-Ausdrücke

Der Parser `prefixExpression` beginnt mit dem Einlesen einer oder mehrerer Wiederholungen von Präfixoperatoren (`-` oder `!`), gefolgt von einem Postfix-Ausdruck. Nach dem Parsen werden die Präfixoperatoren von rechts nach links auf den Postfix-Ausdruck angewandt, um ein neues Objekt `PrefixExpression` zu erzeugen. Dies geschieht mithilfe der `foldRight` Methode, welche die Akkumulatorfunktion `((operator, expression) => PrefixExpression(operator, expression))` auf jedes Element der Liste von Präfixoperatoren und den initialen Postfix-Ausdruck anwendet.

Der Parser `postfixExpression` liest zunächst eine `primaryExpression`, was jeder beliebiger unäre Ausdruck sein kann. Dann wird eine Liste von `postfixOperator` gelesen. Jeder `postfixOperator` besteht aus einem Operator (ein `"("` für Funkti-

onsaufrufe oder ein `"["` für Array-Indizierung) und den entsprechenden Ausdruck oder eine Ausdrucksliste. Die `foldLeft` Methode wird verwendet, um ein neues Objekt `PostfixExpression` zu erzeugen, indem sie den Operator und den entsprechenden Ausdruck von links nach rechts auf die `primaryExpression` anwendet.

Der Parser `postfixOperator` ist eine Hilfsfunktion, die verwendet wird, um die verschiedenen Postfix-Operatoren zu parsen. Dieser liest einen Operator und den zugehörigen Ausdruck oder Ausdrucksliste und gibt ein Paar, bestehend aus dem Operator und dem gelesenen Ausdruck oder Ausdrucksliste, zurück. So wird die Syntax für Funktionsaufrufe und Array-Indizierung implementiert.

3.3.9 Auflösen der Linksrekursion

In der Implementierung des Parsers mit Scala Parser Combinators wurden, ähnlich wie im konventionellen Parser, Designentscheidungen getroffen, um das Problem der Linksrekursion und potenzieller Endlosschleifen zu lösen.

Die vorgestellte Implementierung adressiert das Problem der Linksrekursion auf effektive Weise. Das Schlüsselkonzept hierbei ist der Einsatz von *iterativen* Wiederholungsoperatoren anstelle der direkten Rekursion in den Parsermethoden.

Bei den Parsern wie `infixExpression`, `prefixExpression` und `postfixExpression` wird die potentielle Linksrekursion durch den Gebrauch des `rep`-Operators vermieden, der für iterative Wiederholung steht. Beispielsweise liest `operandParser ~ rep(operatorParser ~ operandParser)` den Operanden gefolgt von einem wiederholten Operator-Operanden-Paar, wodurch die rekursive Struktur dargestellt wird, ohne linksrekursiv zu sein.

Die Verwendung des `rep`-Operators ermöglicht es den Parsern, die gleiche Sprachstruktur zu interpretieren, die eine linksrekursive EBNF-Regel beschreiben würde, jedoch ohne tatsächliche Linksrekursion zu erzeugen. Das bedeutet, dass die Rekursion durch die Iteration ersetzt wird.

Daher kann, obwohl die Parserstruktur die gleiche Sprache beschreibt, wie sie von einer linksrekursiven Grammatik erzeugt werden könnte, die Sprachstruktur sicher analysiert werden, ohne dass das Risiko eines „Stack-Overflows“ durch Linksrekursion besteht. Dies ist ein allgemein anerkannter Ansatz zur Auflösung von Linksrekursion, der häufig verwendet wird, um Recursive-Descent Parser zu ermöglichen, die sonst Schwierigkeiten mit linksrekursiven Grammatiken hätten.

3.3.10 Implementierung der Fehlerbehandlung

In der konventionellen Parser-Version sind spezifische Fehlerbehandlungen implementiert. Diese generieren aussagekräftige Fehlermeldungen beim Parsen fehlerhafter oder unvollständiger Syntaxen, die anschließend dargestellt werden können. Um äquivalente Fehlerbehandlungen in der Version mit Scala Parser Combinators zu realisieren, sind Modifikationen an den betroffenen Parsern erforderlich.

Die Herausforderung besteht dabei in der spezifischen Arbeitsweise der Scala Parser Combinators: Ein Parser wird nur dann ausgeführt, wenn seine Pattern-Matching-Bedingung vollständig erfüllt ist, was letztendlich zu einer Transformation des Quellcodes führt. Im Beispiel des Parsers für ganze Zahlen entspricht das dem vordefinierten Muster von JavaTokenParsers `wholeNumber`, das als regulärer Ausdruck `-?\d+` definiert ist [16]. Wenn der Quelltext nun eine inkorrekte Syntax für ganze Zahlen enthält (z.B. `123abc`), würde dieser Parser nicht ausgeführt werden. Dadurch entsteht keine Möglichkeit zur Erstellung einer Fehlermeldung.

Eine mögliche Lösung besteht darin, das initial definierte Muster für das Pattern-Matching allgemeiner zu gestalten und danach eine detailliertere Textanalyse durchzuführen. Dadurch wird sichergestellt, dass der Parser überhaupt ausgelöst wird und der Text auf mögliche Fehler hin untersucht werden kann. In folgendem Beispiel wird gezeigt, wie das im Fall eines Parsers für ganze Zahlen umgesetzt werden kann:

```
1 def integer: Parser[IntegerLiteral] = {
2     """\d+\w*""".r >> {
3         case value if value.matches("(0|[1-9]+\\d*)") => success(IntegerLiteral(value
      .toInt))
4         case invalidInput => err(s"Could not parse '$invalidInput' as integer.")
5     }
6 }
```

Listing 3.30: Implementieren von Fehlerbehandlungen

Der Parser im gegebenen Beispiel nutzt eine zweistufige Strategie zur Überprüfung von Text auf korrekte Syntax, entsprechend dem erwähnten Ansatz.

In der ersten Stufe verwendet der Parser einen recht allgemeinen regulären Ausdruck `\d+\w*`, um eine Vorabselektion zu treffen. Dieser reguläre Ausdruck passt auf jeglichen Text, der mit einer oder mehreren Ziffern beginnt und anschließend beliebige Wort-Zeichen beinhaltet.

Nachdem diese erste Überprüfung durchgeführt wurde, führt der Parser eine zweite, detailliertere Analyse durch. Dies geschieht innerhalb der `case` Anweisungen. Es gibt zwei Möglichkeiten:

Wenn der Wert dem präziseren regulären Ausdruck `(0|[1-9]+\\d*)` entspricht, wird er als `IntegerLiteral` ausgewertet. Dieser reguläre Ausdruck entspricht einer Null oder einer Zahl, die nicht mit Null beginnt und eine oder mehrere Zahlen enthält. Wenn der Wert nicht dem präziseren regulären Ausdruck entspricht, wird ein Fehler generiert, der angibt, dass der Wert nicht als ganze Zahl ausgewertet werden konnte.

Durch die Kombination dieser beiden Stufen kann einerseits sichergestellt werden, dass der Parser auch für Eingaben aktiviert wird, die nicht genau dem gewünschten Format entsprechen. Andererseits ermöglicht es, detaillierte Fehlermeldungen zu generieren, die den Benutzer präzise darüber informieren, warum bestimmte Eingaben nicht ausgewertet werden konnten.

Um die von den Parsern generierten Fehlermeldungen nach dem Parsen anzuzeigen, muss die Methode `parse` entsprechend erweitert werden. Wenn die Methode `parseAll` einen Rückgabewert des Typs `NoSuccess` liefert, wird der entsprechende Fehler in einem speziell formatierten Text dargestellt. Hierbei kann zudem die genaue Position des Fehlers in Bezug auf Zeile und Spalte angegeben werden.

```
override def parse(input: String): Program = {
    val parseResult = parseAll(program, input)
    parseResult match {
        case Success(parsed, _) => parsed
        case error: NoSuccess =>
            val pos = error.next.pos
            val msg = s"Parsing error at line ${pos.line}, column ${pos.column}: ${
    error.msg}"
            println(msg + "\n")
            throw new ParserException("Parsing failed with errors.")
    }
}
...
class ParserException(message: String) extends RuntimeException(message)
```

Listing 3.31: Anzeigen von Fehlermeldungen

```
>>> let foo = 123abc;
Parsing error at line 1, column 17: Could not parse '123abc' as integer.
>>> true(0);
Parsing error at line 1, column 5: Identifier 'true' is not acceptable here.
```

Listing 3.32: Beispiele für ausgewählte Fehlerarten

Für fehlerhafte Eingaben, die nicht explizit durch eigene Fehlerbehandlungen abgedeckt sind, generiert Scala Parser Combinators automatische Fehlermeldungen. Diese sind jedoch nicht in jedem Fall ausreichend aussagekräftig, um eine benutzerfreundliche Erfahrung zu gewährleisten.

```
>>> let foo = 1.0;
Parsing error at line 1, column 12: ';' expected but '.' found
>>> let bar = [1, 2, ];
Parsing error at line 1, column 18: '{' expected but ']' found
>>> return 123
Parsing error at line 1, column 11: ';' expected but end of source found
```

Listing 3.33: Beispiele für automatisch generierte Fehler

3.4 Entwicklung des Interpreters

Anschließend wird zur Evaluation des AST ein „Tree-Walking“ Interpreter erstellt. Dieser beginnt beim Wurzelknoten `program` und durchläuft dann die Verzweigungen zu den Kindknoten. Sobald ein Kindknoten erreicht wird, wandelt der Interpreter das zugehörige AST-Objekt in eine Datenstruktur um, die die entsprechenden Werte von Monkey repräsentiert. Diese Strukturen spiegeln die grundlegenden Elemente wider, die aus allen möglichen Ausdrücken und Anweisungen resultieren können.

3.4.1 Repräsentation der internen Objekte

Die Datenobjekte von Monkey werden durch eine Reihe von Klassen repräsentiert, die die `Object`-Schnittstelle implementieren. Jedes Objekt zeichnet sich dadurch aus, dass es eine Methode `type()` bereitstellt, welche den jeweiligen Objekttyp zurückgibt und eine Methode `toString()`, die eine Text-Repräsentation des Objekts generiert.

```
trait Object {
    def `type`(): ObjectType
    override def toString: String
}

enum ObjectType {
  case INTEGER, BOOLEAN, STRING, ARRAY, HASH, NULL, RETURN, ERROR, FUNCTION,
    COMPILED_FUNCTION, BUILTIN, CLOSURE
}
```

Listing 3.34: Object-Schnittstelle für die Repräsentation der Objekte

Die Enum-Klasse ObjectType definiert eine Aufzählung der verschiedenen Objekttypen, die in Monkey existieren. Dazu gehören neben den bereits erwähnten Werten von Monkey ebenfalls Funktionen (`FUNCTION`, `COMPILED_FUNCTION`, `BUILTIN`, `CLOSURE`) sowie spezielle Typen wie `NULL`, `RETURN` und `ERROR`.

Die konkreten Klassen, die die `Object`-Schnittstelle implementieren, repräsentieren diese unterschiedlichen Typen. So gibt es beispielsweise die Klasse `IntegerObject` für ganze Zahlen, `BooleanObject` für Boolesche Werte, `StringObject` für Zeichenketten, `ArrayObject` für Arrays und so weiter. Jede dieser Klassen speichert die

zugrunde liegenden Werte und stellt eine entsprechende Text-Repräsentation bereit. In Anlehnung an die Vorgehensweise bei der Implementierung des Parsers werden auch im Interpreter die Werte der Host-Programmiersprache Scala zur Darstellung der Monkey-Werte genutzt.

```
case class ArrayObject(elements: List[Object]) extends Object {
  override def toString: String = s"[${elements.mkString(", ")}]"
  def `type`(): ObjectType = ObjectType.ARRAY
}
```

Listing 3.35: Definition der internen Datenklasse für Arrays

Einige der Objekte sind „hashbar“, wie es durch die Schnittstelle `Hashable` gekennzeichnet wird. Dies ermöglicht es, sie als Schlüssel in Hashmaps zu verwenden, die durch die Klasse `HashObject` repräsentiert werden.

```
case class IntegerObject(value: Int) extends Object with Hashable {
    override def toString: String = value.toString
    def `type`(): ObjectType = ObjectType.INTEGER

    override def hashKey: HashKey = HashKey(value.hashCode())
}
```

Listing 3.36: Hashbare Datenklasse für Integer

Für Funktionen gibt es mehrere spezifische Objekttypen. Die Klasse `FunctionObject` repräsentiert normale Funktionen, die durch den Benutzer definiert werden. `BuiltinObject` steht für eingebaute Funktionen der Sprache (z.B. `len`, `first`, `last`, `rest`, `push` und `puts`), `CompiledFunctionObject` für kompilierte Funktionen und `ClosureObject` für Closure-Funktionen, die den Zugriff auf den Kontext ihrer Erstellung beim Kompilieren ermöglichen. Letztere beiden werden nur für die Entwicklung des Compilers benötigt.

3.4.2 Der Evaluationsalgorithmus des Interpreters

Bei der Entwicklung des Interpreters ist der erste Schritt die Definition jener Objekte, die häufig wiederverwendet werden, deren tatsächlicher Wert sich jedoch nie ändert. Im Falle von Monkey sind dies die Werte `true`, `false` und eine Repräsentation für die Abwesenheit von Werten, `null`. Obwohl `null` nicht als Wert in einem Monkey-Programm genutzt werden kann, kann es dennoch als Rückgabewert einer Anweisung oder eines Ausdrucks dienen. Der wesentliche Vorteil der einmaligen Definition dieser drei Konstanten liegt in der Leistungssteigerung des Interpreters. Da diese Werte häufig das Ergebnis von Ausdrücken sind, müssen sie nicht ständig neu erstellt werden, was die Effizienz erhöht.

```
object Evaluator {
    val TRUE: BooleanObject = BooleanObject(true)
    val FALSE: BooleanObject = BooleanObject(false)
    val NULL: NullObject.type = NullObject
    ...
```

Listing 3.37: Einmalige Definition der wiederverwendbaren Werte

Anschließend wird ähnlich vorgegangen wie bei der Implementation des konventionellen Parsers: Für jeden Objekttyp wird eine spezifische Methode entwickelt, welche für das Auswerten dieses Objektes zuständig ist. Es wird eine generelle Methode `evaluate` definiert, welche als Einstiegspunkt des Interpreters verwendet wird und rekursiv die spezifischen Evaluationsmethoden aufruft. Diese geben das ausgewertete Ergebnis an ihre aufrufenden Evaluationsmethoden zurück und reduzieren so jede Anweisung bzw. jeden Ausdruck zu einem einzigen Wert.

```
def evaluate(node: Option[Node], env: Environment): Object = node match {
    case Some(node: Program) => evaluateProgram(node, env)
    case Some(node: ExpressionStatement) => evaluateExpressionStatement(node, env)
    case Some(node: IntegerLiteral) => evaluateInteger(node)
    case Some(node: BooleanLiteral) => evaluateBoolean(node)
    case Some(node: StringLiteral) => evaluateString(node)
    case Some(node: ArrayLiteral) => evaluateArray(node, env)
    case Some(node: HashLiteral) => evaluateHash(node, env)
    case Some(node: PrefixExpression) => evaluatePrefixExpression(node, env)
    case Some(node: InfixExpression) => evaluateInfixExpression(node, env)
    case Some(node: BlockStatement) => evaluateBlockStatement(node, env)
    case Some(node: IfExpression) => evaluateIfExpression(node, env)
    case Some(node: ReturnStatement) => evaluateReturnStatement(node, env)
    case Some(node: LetStatement) => evaluateLetStatement(node, env)
    case Some(node: Identifier) => evaluateIdentifier(node, env)
    case Some(node: FunctionLiteral) => evaluateFunctionLiteral(node, env)
    case Some(node: CallExpression) => evaluateCallExpression(node, env)
    case Some(node: IndexExpression) => evaluateIndexExpression(node, env)
    case _ => NULL
}
```

Listing 3.38: Methode `evaluate` des Interpreters

Einige dieser Evaluationsmethoden sind direkte Umwandlungen der AST-Objekte in die Datenobjekte von Monkey:

```
def evaluateInteger(integerLiteral: IntegerLiteral): Object = {
    IntegerObject(integerLiteral.value)
}
```

Listing 3.39: Direkte Umwandlung von IntegerLiteral in Datenobjekt IntegerObject

Für komplexere Knoten, die aus mehreren Unterknoten zusammengesetzt sind, wie beispielsweise bei einem Präfix-Ausdruck, erfolgt die Evaluation ihrer einzelnen Ausdrücke oder Anweisungen sequenziell durch das Aufrufen der `evaluate`-Methode.

Anschließend wird das Datenobjekt aus den individuellen Ergebnissen zusammengestellt und zurückgegeben.

Beim Umgang mit einem Präfix-Ausdruck beispielsweise wird zunächst der rechtsseitige Ausdruck evaluiert. Danach wird eine entsprechende Aktion auf diesen berechneten Wert angewendet, abhängig vom Operator des Präfix-Ausdrucks.

```
def evaluatePrefixExpression(prefixExpression: PrefixExpression, environment:
    Environment): Object = {
    val right: Object = evaluate(Some(prefixExpression.value), environment)

    prefixExpression.operator match {
        case "!" if isTruthy(right) => FALSE
        case "!" => TRUE
        case "-" => right match {
            case right: IntegerObject => IntegerObject(-right.value)
            case _ => NULL
        }
    }
}
...
def isTruthy(obj: Object): Boolean = if (obj == FALSE || obj == NULL) false else true
```

Listing 3.40: Methode zur Evaluation von Präfix-Ausdrücken

In diesem Beispiel wird der spezielle Wertetyp `NULL` genutzt. Wenn der Operator eines Präfix-Ausdrucks `-` ist und der rechtsseitige Ausdruck keine ganze Zahl darstellt, wie im Fall von `-true;`, erfolgt eine Evaluierung zu `NULL`.

Infix-Ausdrücke werden auf ähnliche Weise evaluiert, erfordern aber zusätzliche Überprüfungen auf Fehler: Die Ausdruckstypen müssen mit dem Operator und miteinander kompatibel sein. Bei Regelverstößen wird ein `ErrorObject` mit ausführlicher Fehlermeldung anstelle einer Evaluation generiert.

```
...
(infixLeftValue, infixExpression.operator, infixRightValue) match {
    case (left: Object, opr: String, right: Object)
        if left.`type`() != right.`type`() =>
            ErrorObject(s"type mismatch: ${left.`type`()} $opr ${right.`type`()}")
    case (left: IntegerObject, operator: String, right: IntegerObject) =>
        evaluateIntegerInfixExpression(operator, left, right)
    case (left: StringObject, operator: String, right: StringObject) =>
        evaluateStringInfixExpression(operator, left, right)
    case (left: BooleanObject, "==", right: BooleanObject) =>
        if (left.value == right.value) TRUE else FALSE
    case (left: BooleanObject, "!=", right: BooleanObject) =>
        if (left.value != right.value) TRUE else FALSE
    case (left, operator, right) =>
        ErrorObject(s"unknown operator: ${left.`type`()} $operator ${right.`type`()
    }")
}
```

Listing 3.41: Fehlerbehandlung bei Infix-Ausdrücken

In diesem Beispiel wird die Fehlerbehandlung durch den Interpreter bei Infix-Ausdrücken veranschaulicht. Der Interpreter erkennt innerhalb der einzelnen Evaluationsmethoden spezifische Arten von *semantischen* Fehlern. Insbesondere bei der Evaluation von Infix-Ausdrücken wird auf Inkonsistenz der Datentypen und die Verwendung unzulässiger Operatoren geprüft.

3.4.3 Verwendung des Environments

Die vorgestellten Prinzipien ermöglichen die Implementierung der meisten weiteren Evaluationsmethoden auf ähnliche Weise. Um jedoch die Deklaration und Zuweisung von Variablen zu ermöglichen, sind zusätzliche Maßnahmen erforderlich. In Monkey wird das Schlüsselwort `let` verwendet, um Variablen mit einem Namen (Identifier) einen Wert zuzuweisen. Dieser Wert soll gespeichert werden, um zu einem späteren Zeitpunkt wieder abgerufen werden zu können. Einer Variable sollte es möglich sein, alle Arten von Ausdrücken als Wert anzunehmen, einschließlich Funktionen. Diese sollten danach durch eine `callExpression` ausgeführt werden können optional unter Angabe von Argumenten.

Zusätzlich sollen Variablen innerhalb eines spezifischen Namensraums definiert werden, aus dem sie anschließend wieder referenziert werden können. Außerhalb dieser Namensräume sollte eine Referenzierung nicht möglich sein. Dies ermöglicht beispielsweise die Definition von Variablen mit demselben Namen in unterschiedlichen Namensräumen:

```
>>> let x = 1;
>>> let y = 2;
>>> let foo = fn(x) { let y = 3; x + y; };
>>> foo(4);
7
>>> y;
2
```

Listing 3.42: Globale und lokale Namensräume von Variablen

In diesem Beispiel wird beim Aufruf der Funktion `foo` die lokale Variable `x` der Funktion mit dem Wert 4 verwendet, anstelle der globalen Variable `x` mit dem Wert 1. Ebenso wird `y` innerhalb der Funktion als lokale Variable mit dem Wert 3 neu definiert. Bei einem Zugriff auf `y` außerhalb der Funktion wird hingegen wieder die globale Definition von `y` mit dem Wert 2 verwendet.

Um dieses Verhalten zu realisieren, bedarf es einer zusätzlichen Datenstruktur, die für das Speichern von Variablen und deren Werten zuständig ist. Hierfür wird eine Klasse `Environment` erstellt, die die Objekte in einer Hashmap zuordnet und optional eine Referenz zu einer äußeren Umgebung speichert, welche auf den Identifikator durchsucht wird, falls dieser nicht im eigenen Namensraum definiert ist.

```
class Environment(var outer: Option[Environment] = None) {
  private val store: mutable.Map[String, Object] = mutable.Map()

  def get(name: String): (Option[Object], Boolean) = {
    store.get(name) match {
      case Some(value) => (Some(value), true)
      case None => outer match {
        case Some(env) => env.get(name)
        case None => (None, false)
      }
    }
  }

  def set(name: String, value: Object): Object = {
    store(name) = value
    value
  }
}
```

Listing 3.43: Definition der Klasse `Environment`

Die Klasse `Environment` kann dann beispielsweise in der Methode zur Evaluation von `LetStatement` verwendet werden, um Variablen zu speichern:

```
def evaluateLetStatement(letStatement: LetStatement, env: Environment): Object = {
    val letValue: Object = evaluate(Some(letStatement.value), env)
    if (isError(letValue))
        letValue
    else {
        environment.set(letStatement.identifier.name, letValue)
        NULL
    }
}
```

Listing 3.44: Evaluation von `LetStatement`

Das Gegenstück dazu ist die Methode zur Evaluation von `Identifier`. Ein Identifikator kann entweder zu einem „normalen“ Wert, einer durch den Benutzer definierten Funktion oder einer „Builtin“-Funktion evaluiert werden. In Monkey sind die folgenden Builtin-Funktionen vordefiniert:

- `len`: Gibt die Länge eines Arrays oder eines Strings zurück.
- `first`: Gibt das erste Element in einem Array zurück.
- `last`: Gibt das letzte Element in einem Array zurück.
- `rest`: Gibt ein neues Array zurück, welches alle Elemente des ursprünglichen Arrays außer dem ersten Element enthält.
- `push`: Gibt ein neues Array zurück, welches alle Elemente des ursprünglichen Arrays und zusätzlich das angegebene Element am Ende enthält.

- `puts`: Gibt den angegebenen String aus und gibt `NULL` zurück.

Die Builtin-Funktionen werden in einer Klasse `Builtins` definiert, indem einem Namen ein `BuiltinObject` und dessen definiertes Verhalten zugewiesen wird:

```
object Builtins {
  val builtins: Map[String, BuiltinObject] = Map(
    "len" -> BuiltinObject((args: Array[objects.Object]) => {
      if (args.length != 1) {
        ErrorObject(s"wrong number of arguments. got ${args.length} but expected 1.")
      } else {
        args(0) match {
          case arrayObject: ArrayObject =>
            IntegerObject(arrayObject.elements.size)
          case stringObject: StringObject =>
            IntegerObject(stringObject.value.length)
          case _ =>
            ErrorObject(s"argument to `len` not supported, got ${args(0).`type`()}")
        }
      }
    }),
    ...
```

Listing 3.45: Definition der Builtin-Funktion `len`

So kann in der Methode zur Evaluation von `Identifier` entschieden werden, ob es sich bei dem gespeicherten Objekt um einen Wert, eine Funktion oder eine Builtin-Funktion handelt:

```
def evaluateIdentifier(identifier: Identifier, env: Environment): Object = {
    val (result: Option[Object], ok: Boolean) = env.get(identifier.name)
    if (ok) {
        result.getOrElse(NULL)
    } else {
        val maybeBuiltin = Builtins.builtins.get(identifier.name)
        maybeBuiltin match {
            case Some(builtinObject: BuiltinObject) => Some(builtinObject).get
            case None => ErrorObject(s"identifier not found: ${identifier.name}")
        }
    }
}
```

Listing 3.46: Methorde zur Evaluation von Identifikatoren

3.4.4 Ausführen von Funktionen

Die letzte bedeutende Herausforderung bei der Implementierung des Interpreters besteht im Ausführen von Funktionen und vordefinierten Builtin-Funktionen. Gemäß der EBNF-Grammatik sind die speziellen Eingaben des folgenden Beispiels gültig und sollen ebenfalls vom Interpreter ausgewertet werden können. Funktionen können

als Argumente übergeben werden, „higher-order“-Funkionen werden unterstützt und sogenannte „Closures“ können definiert werden:

```
>>> let addThree = fn(x) { return x + 3; };
>>> let callTwoTimes = fn(x, func) { func(func(x)); };
>>> callTwoTimes(3, addThree);
9
>>> callTwoTimes(3, fn(x) { x + 1; });
5
>>> let newAdder = fn(x) { fn(n) { x + n; } };
>>> let addTwo = newAdder(2);
>>> addTwo(2);
4
```

Listing 3.47: Beispiele für Funktionen und Funktionsaufrufe [3]

Closures sind ein fundamentales Konzept in vielen modernen Programmiersprachen, insbesondere in jenen, die Funktionen als „First-Class-Citizens“ unterstützen. Es handelt sich dabei um eine besondere Art von Funktion, die den Zugriff auf Variablen aus einem umgebenden Kontext – also außerhalb ihres eigenen Geltungsbereichs („Scope“) – ermöglichen. Dieser umgebende Kontext wird oft als „Closure“ oder „geschlossener Bereich“ bezeichnet, woraus sich der Name „Closure-Funktion“ ableitet. Das bedeutet, dass die Funktion Zugriff auf mehr hat, als nur ihre eigenen lokalen Variablen. Dieses Konzept ist entscheidend für die Implementierung von Funktionen als „First-Class-Citizens“, wo Funktionen wie jede andere Variable behandelt werden können [17].

Im gegebenen Beispiel gibt es einige Closures:

- `addThree` ist eine Funktion, die auf die freie Variable `x` aus dem umgebenden Kontext zugreift.
- `callTwoTimes` ist eine Funktion, die auf die freien Variablen `x` und `func` aus dem umgebenden Kontext zugreift.
- Die anonyme Funktion `fn(x) { x + 1; }`, die an `callTwoTimes` übergeben wird, ist ebenfalls eine Closure, die auf die freie Variable `x` aus ihrem umgebenden Kontext zugreift.
- `newAdder` ist eine Funktion, die eine andere Funktion zurückgibt, die auf die Variable `x` aus dem umgebenden Kontext zugreift.
- `addTwo` ist auch eine Closure, die durch den Aufruf von `newAdder(2)` erstellt wurde. Sie behält eine Referenz auf die Variable `x` aus dem umgebenden Kontext von `newAdder`.

Closures erlauben das Arbeiten mit zustandsbehafteten Funktionen und deren Behandlung als „First-Class-Citizens“. Sie ermöglichen beispielsweise Konstrukte, in denen

Funktionen generiert und anderen Funktionen als Argument übergeben werden, wie im oben angeführten Beispiel dargestellt.

Der Evaluationsprozess der oben gezeigten Funktionsarten beginnt mit dem Aufruf der Evaluationsmethode für Funktionsaufrufe, `evaluateCallExpression`:

```
def evaluateCallExpression(callExpression: CallExpression, env: Environment): Object
    = {
    val callFunction: Object = evaluate(Some(callExpression.function), env)
    if (isError(callFunction))
        return callFunction

    val callArguments: List[Object] =
        evaluateExpressions(Some(callExpression.arguments), env)
    if (callArguments.length == 1 && isError(callArguments.head))
        callArguments.head
    else
        applyFunction(callFunction, callArguments)
}
```

Listing 3.48: Methode zur Evaluation von Funktionsaufrufen

Durch den Aufruf der Methode `applyFunction` wird die Funktion unter der Angabe der Argumente ausgeführt:

```
def applyFunction(function: Object, arguments: List[Object]): Object = {
    function match {
        case functionObject: FunctionObject =>
            val extendedEnv = extendEnvironment(functionObject, arguments)
            val evaluated = functionObject.body match {
                case Some(body) => evaluate(Some(body), extendedEnv)
                case _ => return NULL
            }
            unwrapReturnValue(evaluated)
        case builtinObject: BuiltinObject =>
            builtinObject.builtinFunction(arguments.toArray)
        case _ =>
            ErrorObject(s"not a function: ${function.`type`()}")
    }
}
```

Listing 3.49: Methode zum Ausführen von Funktionen

Die Methode `extendEnvironment` erlaubt den Zugriff auf äußere Namensräume und deren Variablen innerhalb einer Funktion. Hierfür wird ein neuer Namensraum erstellt, der die Variablen des umgebenden Namensraums enthält. Anschließend wird dieser neue Namensraum anstelle des ursprünglichen Namensraums der Funktion für die weitere Auswertung verwendet.

```
def extendEnvironment(function: FunctionObject, args: List[Object]): Environment = {
    val environment = new Environment(Some(function.environment))
    function.parameters.foreach { parameters =>
```

```
4         for ((parameter, argument) <- parameters.zip(args)) {
5             environment.set(parameter.name, argument)
6         }
7     }
8     environment
9 }
```

Listing 3.50: Methode zum Erweitern der Namensräume beim Funktionsaufruf

Wenn das evaluierte Funktionsobjekt ein `ReturnObject` ist, stellt die Methode `unwrapReturnValue` sicher, dass nur der Wert der Return-Aweisung und nicht die gesamte Return-Anweisung zurückgegeben wird:

```
1 def unwrapReturnValue(obj: Object): Object = obj match {
2     case ReturnObject(value) => value.getOrElse(NULL)
3     case _ => obj
4 }
```

Listing 3.51: Methode zum Entpacken der Werte von Return-Anweisungen

Diese Methode überprüft, ob das übergebene Objekt vom Typ `ReturnObject` ist. Ist dies der Fall, gibt sie den in diesem `ReturnObject` gespeicherten Wert zurück. Wenn der Wert nicht vorhanden ist, wird NULL zurückgegeben. Wenn das übergebene Objekt kein `ReturnObject` ist, wird es unverändert zurückgegeben.

3.4.5 Zusammenfassung des Evaluations-Algorithmus

In den vorherigen Kapiteln wurde die Evaluation des AST durch den Interpreter im Detail erläutert. Die vorgestellte Umsetzung des Interpreters ermöglicht es, einfache und komplexe Ausdrücke in Monkey zu bearbeiten und entsprechende Ergebnisse zu liefern. Ein wesentlicher Aspekt der Evaluation besteht in der korrekten Umsetzung von Variablenzuweisungen und -zugriffen durch das `Environment`.

Ein weiteres Schlüsselelement des Interpreters sind die Builtin-Funktionen, die in die Sprache eingebettet und deren Verhalten definiert wurde. Durch ihre Implementierung konnte die Funktionalität der Sprache erheblich erweitert werden.

Der hier beschriebene Evaluationsprozess ist eine robuste Grundlage für die Ausführung von Monkey-Programmen. Die Implementierung von Funktionen, einschließlich des Umgangs mit Closures und Builtin-Funktionen, zeigt die Fähigkeit des entwickelten Interpreter-Algorithmus, komplexere Programmierparadigmen zu unterstützen.

3.5 Entwicklung des Compilers

Die inhärente Komplexität eines Compilers und einer VM lässt nur eine oberflächliche Betrachtung der Hauptunterschiede zum Interpreter und der grundlegenden Konzepte in diesem Kapitel zu. Im Unterschied zum vorherigen Kapitel, das den Algorithmus des Interpreters behandelt, soll dieses Kapitel keinen umfassenden Leitfaden oder Handbuch für die Compilerentwicklung bieten. Es dient vielmehr dazu, einen Einblick in die grundlegenden Konzepte zu geben und ein Basisverständnis zu ermöglichen. Falls die Leserinnen und Leser Interesse daran haben, einen Compiler für Monkey zu implementieren, wird empfohlen, das Buch „Writing a Compiler in Go“ [4] von Thorsten Ball heranzuziehen, oder die vollständige Implementierung in Scala aus dieser Arbeit im zuvor erwähnten GitHub-Repository zu konsultieren.

Im Rahmen dieser Arbeit wurde ein Compiler entwickelt, der einen sogenannten Bytecode generiert. Dieser erzeugte Bytecode kann nachfolgend von einer VM ausgeführt werden. Der entwickelte Compiler gehört zur Kategorie der AOT-Compiler, welche durch eine klare Trennung zwischen der Erstellung des Bytecodes und seiner Ausführung durch die VM gekennzeichnet sind. Die Vorteile und Herausforderungen dieser Art von Compilern wurden bereits im vorangegangenen Kapitel der Grundlagen diskutiert (siehe Abschnitt Unterabschnitt 2.1.3).

3.5.1 Erzeugen des Bytecodes

Ein Bytecode wird vom Compiler generiert und stellt eine Zwischenform des Programmcodes dar, die als eine „Low-Level“-Version des ursprünglichen Monkey-Programms fungiert. Seine Hauptaufgabe besteht darin, durch Optimierungen und den Einsatz von Stack-Operationen eine beschleunigte Ausführung des Programms im Vergleich zur Evaluation durch einen Interpreter zu ermöglichen [18].

Im Folgenden wird ein exemplarisches Monkey-Programm und der korrespondierende, generierte Bytecode präsentiert, um ein grundlegendes Verständnis für das Thema zu vermitteln. Allerdings wird in dieser Arbeit nicht detailliert auf die Funktionsweise des Algorithmus zur Erzeugung des Bytecodes eingegangen. Die dafür notwendige Diskussion würde den Umfang dieser Arbeit übersteigen.

Folgend wird ein Beispiel für Bytecode des Programms `1 + 2;` gezeigt:

```
0000 OpConstant 0
0003 OpConstant 1
0006 OpAdd
0007 OpPop
```

Listing 3.52: Formatierte Ausgabe des Bytecodes für das Beispielprogramm `1 + 2;`

Der Bytecode setzt sich aus individuellen Operationen, den sogenannten „Opcodes“, zusammen. Diese repräsentieren die atomaren Befehle, die von der VM auf dem Stack ausgeführt werden. Der zuvor gezeigte Bytecode ist zur Lesbarkeit als Text formatiert, besteht aber tatsächlich aus einem einzigen Array des Typs „UnsignedByte“. Jeder Opcode entspricht einer bestimmten Stack-Operation und kann, abhängig von seinem Typ, bis zu zwei Operanden (ebenfalls vom Typ UnsignedByte) enthalten.

Beispielsweise signalisiert der Opcode `OpConstant` aus dem gegebenen Beispiel, dass die VM eine Konstante aus dem Konstanten-Speicher laden soll. Die zusätzlichen Operanden spezifizieren in diesem Kontext den Index der Konstante innerhalb des Konstanten-Speichers. Da die Indizes in diesem Beispiel klein sind und durch ein einziges Byte repräsentiert werden können, wird nur ein Byte zur Darstellung des Index verwendet. Das zweite Byte bleibt ungenutzt und ist standardmäßig auf 0 initialisiert.

Im vorgestellten Beispiel resultiert die Anweisung `OpConstant 0` folglich in das Auflegen der Zahl 1 auf den Stack, während die Anweisung `OpConstant 1` die Zahl 2 auf den Stack legt. Das Konstanten-Array würde wie folgt aussehen: `Array(1, 2)`

Das erklärt auch den linken Teil in der oben dargestellten Bytecode-Repräsentation: Die Zahlen auf der linken Seite geben die Positionen der einzelnen Operationen innerhalb des gesamten Bytecodes an. So benötigen beispielsweise die ersten beiden `OpConstant`-Operationen insgesamt drei Bytes, während `OpAdd` und `OpPop` lediglich ein Byte belegen, da sie keine zusätzlichen Operanden enthalten.

```
1 Array(0u, 0u, 0u, 0u, 0u, 1u, 1u, 5u)
```

Listing 3.53: Array-Darstellung des Bytecodes

In der Bytecode-Darstellung des Arrays werden die einzelnen Opcodes als Bytefolgen sichtbar. Folgend wird eine Gegenüberstellung der formatierten Version und der korrespondierenden Byte-Sequenzen innerhalb der Array-Darstellung vorgenommen:

```
1 0u, 0u, 0u -> 0000 OpConstant 0
2 0u, 0u, 1u -> 0003 OpConstant 1
3 1u         -> 0006 OpAdd
4 5u         -> 0007 OpPop
```

Listing 3.54: Darstellung des Bytecodes als formatierter Text und Array

Die Befehle `OpAdd` und `OpPop` repräsentieren atomare Operationen, die keine zusätzlichen Operanden benötigen. Deshalb werden sie durch ein einziges Byte dargestellt, das ihrem jeweiligen Opcode entspricht (`1u -> OpAdd` und `5u -> OpPop`). Die Operation `OpAdd` instruiert die VM, die zwei obersten Stack-Objekte zu addieren. Dabei werden zunächst die beiden obersten Objekte vom Stack entfernt und temporär

gespeichert, danach addiert und das Resultat zurück auf den Stack gelegt. In ähnlicher Weise agieren die Opcodes für Subtraktion, Multiplikation und Division.

Der Befehl `OpPop` signalisiert der VM, dass das oberste Objekt vom Stack entfernt werden soll. Dies ist nach jeder Anweisung notwendig, um sicherzustellen, dass das vorherige Ergebnis nicht bei der Evaluierung der nächsten Anweisung oder des nächsten Ausdrucks auf dem Stack verbleibt und fälschlicherweise in dessen Berechnung einbezogen wird.

Die Erstellung des Bytecodes für die Darstellung von If-Ausdrücken erweist sich als komplexer und nutzt sogenannte Jump-Befehle, vergleichbar mit denen aus Assembly-Code. Diese Jump-Befehle lenken die VM an spezifische Positionen innerhalb des Bytecodes. Eine Herausforderung hierbei liegt in der vorausschauenden Bestimmung der Entfernung, die die VM springen soll. Dafür muss der Bytecode für die Anweisungen zwischen den Jump-Befehlen zunächst vollständig generiert werden, um die Gesamtzahl an Bytes ermitteln zu können, die mittels des Jump-Befehls zu überspringen sind. Nach Erstellung des Bytecodes wird der Operand des Jump-Befehls nachträglich aktualisiert, um die Anzahl der zu überspringenden Bytes anzugeben.

Damit ergibt sich für das Programm `if (true) { 1; }; 2;` folgender Bytecode:

```
0000 OpTrue
0001 OpJumpNotTruthy 10
0004 OpConstant 0
0007 OpJump 11
0010 OpNull
0011 OpPop
0012 OpConstant 1
0015 OpPop
```

Listing 3.55: Bytecode für If-Ausdruck

Die Anweisung `OpJumpNotTruthy 10` impliziert hierbei, dass die VM an die Position des zehnten Bytes im Bytecode springen und die Verarbeitung von dort aus weiterführen soll. Sollte die Bedingung des If-Ausdrucks nicht erfüllt sein, würde folglich der alternative Pfad des If-Ausdrucks ausgeführt werden. Da im Programm keine explizite Alternative (Else-Block) definiert wurde, wird standardmäßig `NULL` zurückgegeben. Sollte die Bedingung jedoch zutreffen, wird der tatsächliche Konsequenzpfad des If-Ausdrucks durchlaufen und schließlich mittels `OpJump 11` unabhängig vom Ausgang auf die nachfolgende Anweisung des gesamten If-Ausdrucks gesprungen.

Die vorangestellten Beispiele sollen ein grundlegendes Verständnis über die Arbeitsweise des Bytecodes vermitteln. Mit diesem Wissen lassen sich bereits einfache Programme in einen Bytecode überführen. Die Erstellung eines Bytecodes für Funktionen hingegen stellt eine erhebliche Herausforderung dar. Bei deren Implementierung müssen diverse

Überlegungen zur Unterstützung von Konzepten wie freie Variablen, Namensräume und Closures angestellt werden. Die Umsetzung dieser Konzepte gestaltet sich wesentlich komplexer als bei der Entwicklung des Interpreters, da in diesem Fall alle notwendigen Informationen bereits zur Kompilierzeit verfügbar sein müssen. Diese Konzepte werden folgend nur kurz erläutert:

- **Symbols**: Symbole repräsentieren die Namen von Variablen und Funktionen in Ihrem Programm. Ein Symbol ist durch einen Namen und einen „Scope“ (Namensraum) definiert. Der `SymbolTable` ist eine Datenstruktur, die die Beziehung zwischen diesen Symbolen und den tatsächlichen Werten, auf die sie verweisen, verwaltet. Die Symboltabelle kann Symbole auf globaler oder lokaler Ebene definieren und auflösen. Sie speichert auch freie Symbole, also Variablen, die außerhalb der aktuellen Umgebung, aber innerhalb des umschließenden Namensraum einer Funktion definiert sind. In [19] wird eine detaillierte Erklärung dieser Konzepte bereitgestellt.

- **Frames**: Ein Frame ist ein Kontext, in dem ein bestimmter Teil des Programms ausgeführt wird, typischerweise eine Funktion oder ein Codeblock. Jedes Frame hat ein `ClosureObject`, auf die es sich bezieht, und speichert den „Basispointer“ und den „Anweisungspointer“, die den aktuellen Zustand der Ausführung innerhalb dieses Frames darstellen. Der Basispointer bezieht sich auf den Beginn des Frames im Stack, während der Anweisungszeiger auf die aktuelle Position der auszuführenden Anweisung verweist. In [20] wird eine Implementierung von Frames für den Compiler vorgestellt.

- **Closures**: Ein `ClosureObject` ist eine Funktion, die ihren umgebenden Kontext, also den Kontext, in dem sie erstellt wurde, „einschließt“. Diese umgebenden Variablen werden als „freie Variablen“ bezeichnet. Ein `ClosureObject` beinhaltet ein `CompiledFunctionObject`, welches die Anweisungen zur Ausführung der Funktion enthält, sowie eine Liste von freien Objekten. Während der Ausführung wird ein neues `ClosureObject` auf den Stack gelegt, das eine Referenz auf das `CompiledFunctionObject` und die freien Variablen enthält. Bei einem Funktionsaufruf wird ein neues Frame erzeugt und auf den Stack gelegt, um den Kontext für die Ausführung des Closure zu erstellen. In [21] wird ein Compiler unter der Verwendung von Closures implementiert.

Diese Konzepte bilden gemeinsam die Grundlage für den Umgang mit Variablen und Funktionen während der Kompilierung des Bytecodes. Sie ermöglichen die Verwaltung von Variablen- und Funktionsnamen (Symbole), die Ausführung von Funktionen und Codeblöcken in ihrem eigenen Kontext (Frames) und die Erstellung von Funktionen, die Zugriff auf Variablen außerhalb ihres direkten Namensraum haben (Closures).

3.5.2 Funktionsweise der virtuellen Maschine

Die Hauptaufgabe einer VM besteht darin, den Bytecode auszuführen, der zuvor vom Compiler generiert wurde. Der Bytecode repräsentiert das ursprüngliche Programm in einer Form, die von der VM interpretiert werden kann. Jede Bytecode-Instruktion entspricht einer Operation, die die VM ausführen kann, wie z.B. das Addieren zweier Zahlen, das Abrufen eines Objekts aus dem Speicher oder das Springen zu einer anderen Stelle im Code.

Die VM benutzt dafür drei Arten von Speicher: Den Stapelspeicher (Stack), den globalen Speicher und den Konstantenspeicher. Der Stack ist eine First-in-last-out (FILO) Datenstruktur und wird zum Speichern von Variablen und Rückgabewerten von Funktionen während der Laufzeit des Programms verwendet. Der globale Speicher speichert globale Variablen und Funktionen, die im gesamten Programm zugänglich sind. Der Konstantenspeicher hält Konstanten, die während der Laufzeit des Programms nicht geändert werden können.

Die VM verwendet einen Stackpointer, der auf das erste Objekt des Stacks zeigt. Wenn eine Operation ein Objekt auf den Stack legt (push), wird der Stackpointer erhöht. Wenn eine Operation ein Objekt vom Stack nimmt (pop), wird der Stackpointer vermindert. Außerdem verwendet die VM Frames, um den Kontext von Funktionen und deren Aufrufen zu verwalten.

Die VM führt Bytecode in einer Schleife aus. In jeder Iteration liest sie die nächste Instruktion, inkrementiert den Instruktionszeiger und führt die entsprechende Operation aus. Die Art der Operation wird durch den Opcode der Instruktion bestimmt. Die VM hat eine Operation für jeden Opcode, der vom Compiler generiert werden kann.

Beispielsweise liest die `OpConstant`-Operation den entsprechenden Konstantenwert aus dem Konstantenspeicher. Anschließend legt sie diesen Wert auf den Stack. Die `OpAdd`-Operation hingegen holt zwei Werte vom Stack, addiert sie und legt das Ergebnis wieder auf den Stack.

Die Art und Weise, wie die VM den Bytecode evaluiert, ähnelt stark dem Verhalten physischer Maschinen bei der Ausführung von Maschinencode. Indem sie sowohl den physischen Speicher als auch die Operationen, die bei echten Maschinen von der Arithmetic Logic Unit (ALU) durchgeführt werden, simuliert, verkörpert die VM das Prinzip einer „virtuellen Maschine“ besonders treffend.

Die vollständige Implementierung der VM kann ebenfalls im zuvor erwähnten GitHub-Repository eingesehen werden. Eine detaillierte Erklärung der Funktionsweise einer Stack-basierten VM wird durch [21] bereitgestellt.

4 Analyse der Ergebnisse

In diesem Kapitel erfolgt eine Analyse der Ergebnisse, die sich aus der Implementierung und Ausführung der beiden Parser-Versionen, sowie des Interpreters, des Compilers und der VM ergeben haben.

Es ist zu beachten, dass alle Leistungsvergleiche auf derselben Hardware und Softwareumgebung durchgeführt wurden. Konkret wurden die Tests auf einem System mit einem AMD Ryzen 5 5600x Prozessor und 32 GB RAM durchgeführt. Als Entwicklungsumgebung wurde die IntelliJ IDEA genutzt.

Diese homogene Testumgebung stellt sicher, dass die resultierenden Leistungsunterschiede ausschließlich auf Unterschiede in den Implementierungsstrategien und Designentscheidungen zurückzuführen sind und nicht auf Unterschiede in der zugrundeliegenden Hardware oder Software.

4.1 Überprüfung der Funktionalität

Die Implementierungen der beiden Parser-Versionen wurden erfolgreich abgeschlossen und zeigten eine vollständige Erfüllung der spezifizierten EBNF-Grammatik. Sie konnten die eingegebenen Programme korrekt analysieren und daraus den entsprechenden AST erzeugen. Diese Funktionalität wurde durch umfangreiche Tests sichergestellt, bei denen verschiedene Aspekte der EBNF-Grammatik abgedeckt wurden. Die Ergebnisse bestätigen die Effektivität beider Parser in ihrer jeweiligen Anwendung.

Die Implementierungen des Interpreters und des Compilers erfüllten ebenfalls die gestellten Anforderungen. Beide Systeme waren in der Lage, die ihnen zugeführten Programme korrekt zu interpretieren bzw. zu kompilieren und auszuführen. Dies wurde durch umfangreiche Tests bestätigt, die im GitHub-Repository einsehbar sind.

4.2 Leistungsvergleich der beiden Parser-Versionen

In den folgenden Grafiken wird die Leistung der beiden Parser-Versionen verglichen, indem sie auf identischen Texteingaben getestet wurden. Für den Vergleich, der in Abbildung 3 dargestellt ist, wurde ein Text gewählt, der möglichst alle in Monkey

definierten Elemente, Ausdrücke und Anweisungen mindestens einmal beinhaltet. Dieser Text wurde bis zu 5000-mal dupliziert, und bei jedem Durchlauf wurde die zum Parsen benötigte Zeit in Millisekunden erfasst.

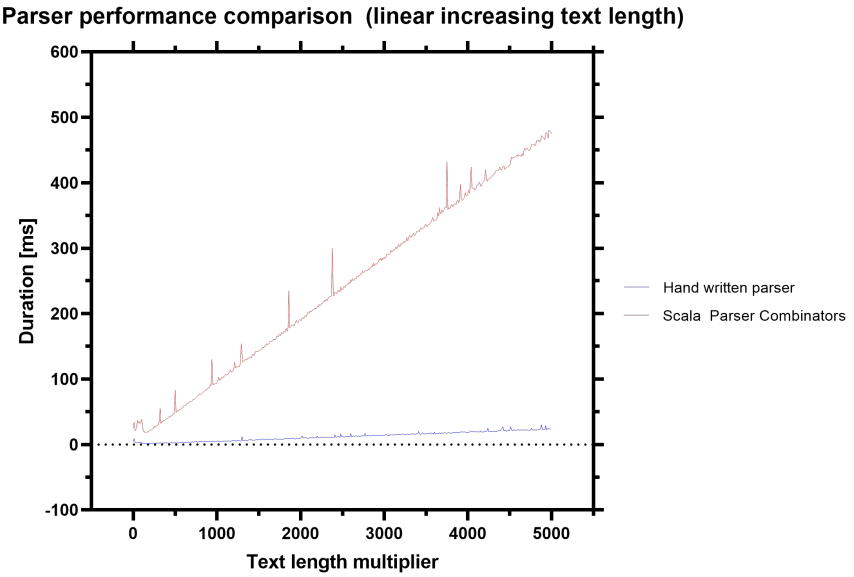

Abbildung 3: Vergleich der Parser-Versionen bei linear erhöhten Textlängen

Die Analyse der erfassten Daten zeigt, dass die Zeit, die beide Parser-Versionen zum Parsen des Textes benötigen, linear mit der zunehmenden Textlänge steigt. Dieses Verhalten bestätigt die erwartete lineare Laufzeit der Parser und ist konsistent mit den theoretischen Erwartungen.

Es ist jedoch anzumerken, dass der Parser, der die Scala Parser Combinators verwendet, deutlich schlechter abschneidet. Bei gleicher Texteingabe ist der konventionelle Parser im Durchschnitt etwa 18-mal schneller. Dieser signifikante Unterschied kann durch verschiedene Faktoren erklärt werden:

- **Rekursionstiefe und Speicheraufwand**: Sowohl die konventionelle Version als auch der Parser mit Scala Parser Combinators verwenden einen Recursive-Descent Algorithmus. Es gibt jedoch Unterschiede in ihren Implementierungen, die möglicherweise zu Leistungsunterschieden führen. Die Scala Parser Combinators können insbesondere bei komplexen und tief verschachtelten Eingaben, eine hohe Speichernutzung und tiefe Rekursion erfordern. Dies kann insbesondere bei Sprachen mit tief verschachtelten Strukturen zu einer erhöhten Laufzeit führen.

  Obwohl auch der konventionelle Parser einen Recursive-Descent Algorithmus nutzt, verwendet er für die gesamte Implementierung der Präzedenzregeln einen iterativen Ansatz. Dies könnte zu einer geringeren Rekursionstiefe und damit zu einer besseren Leistung führen, insbesondere bei komplexen Eingaben.

- **Generizität und Overhead**: Scala Parser Combinators sind darauf ausgelegt, sehr generisch und flexibel zu sein. Sie können eine Vielzahl von Sprachen und Syntaxen auswerten. Diese Flexibilität führt jedoch zu zusätzlichem Overhead, da sie auf allen Ebenen auf die Möglichkeit komplexer und variabler Syntaxen vorbereitet sein müssen. In diesem konkreten Fall konnte die Syntax von Monkey deutlich schneller von einem spezialisierten Parser verarbeitet werden.
- **Verwendung von Backtracking**: Scala Parser Combinators verwenden standardmäßig Backtracking, um mit Ambiguitäten umzugehen. Das bedeutet, sie probieren verschiedene Pfade aus wenn mehrere passen könnten und kehren zurück wenn ein Pfad fehlschlägt. Dieses Verhalten ist bei kleineren Eingaben unproblematisch, kann aber bei größeren Eingaben oder komplexeren Grammatiken zu erheblichen Leistungseinbußen führen.

In Abbildung 4 wurde ein Vergleich durchgeführt, der dem zuvor dargestellten ähnelt. Der entscheidende Unterschied besteht jedoch darin, dass der analysierte Text dieses Mal aus zunehmend verschachtelten Funktionsaufrufen besteht. Konkret werden die Funktionsaufrufe bis zu 10000-mal ineinander verschachtelt: `foo(foo(foo(...)))`.

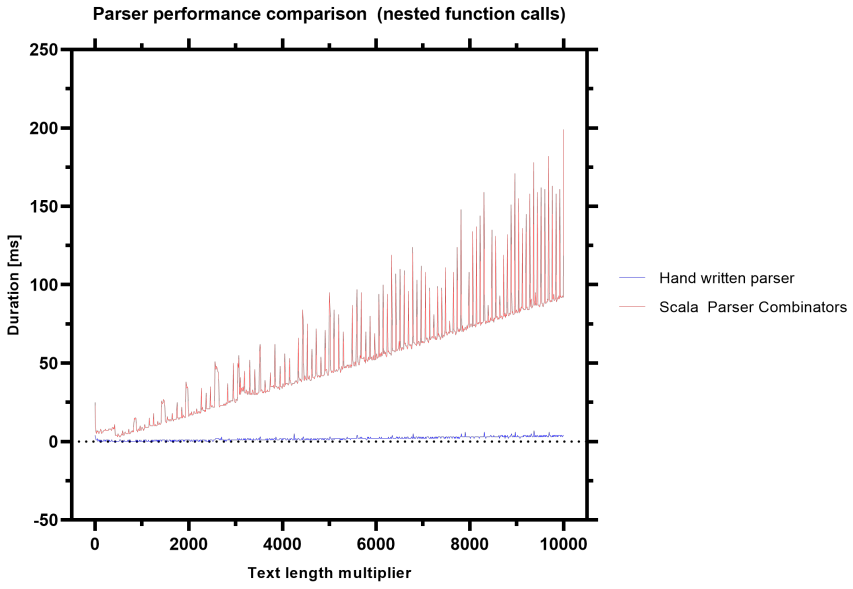

Abbildung 4: Vergleich der Parser-Versionen bei linear erhöhten Textlängen mit verschachtelten Funktionsaufrufen

Die Resultate dieses Vergleichs weisen eine ähnliche Tendenz wie der vorherige Test auf, jedoch zeigt sich eine geringfügige Verschlechterung der durchschnittlichen Ausführungszeit des Parsers mit Scala Parser Combinators. Im Durchschnitt war dieser etwa 22-mal langsamer bei der Auswertung des identischen Textes. Besonders auffällig sind in diesem Fall die regelmäßigen, signifikanten Ausschläge in den Ausführungszeiten

des Parsers mit Scala Parser Combinators. Diese Spitzenwerte treten in Abständen von etwa 80 verschachtelten Funktionsaufrufen auf, es gibt jedoch gelegentlich Ausnahmen, in denen sie nicht auftreten. Die Erklärung für diese Spitzen könnte in der Art und Weise liegen, wie Scala Parser Combinators mit tiefen Rekursionen umgeht. Möglicherweise führt eine bestimmte Schwelle der Verschachtelungstiefe zu einer Verzögerung bei der Speicherfreigabe oder zu einem Wechsel in die Verarbeitungsstrategie.

Die beobachteten Spitzen in den Ausführungszeiten könnten ebenso auf die spezifischen Eigenschaften der im Testfall verwendeten Parsermethoden des Parsers mit Scala Parser Combinators zurückzuführen sein. Vor allem die Methoden, die für Funktionsaufrufe verwendet werden, sind hier relevant. Es kann daher sein, dass die unerwarteten Spitzen in den Leistungsdaten auf Besonderheiten in der Implementierung genau dieser Methoden zurückzuführen sind. Der beobachtete Effekt kann also nicht unbedingt ein allgemeines Leistungsmerkmal des Parsers mit Scala Parser Combinators repräsentieren, sondern spezifisch mit der Verarbeitung von verschachtelten Funktionsaufrufen in Zusammenhang stehen. Insbesondere die Parsermethode für Funktionsaufrufe hat in der Präzedenzstruktur eine sehr hohe Stellung, nur übertroffen von Indexausdrücken. Dies führt dazu, dass diese Methode eine besonders hohe Rekursionstiefe aufweist. Zudem kann es aufgrund dieser hohen Präzedenz zu verstärktem Backtracking kommen. Beide Aspekte können zu erhöhter Laufzeit führen, insbesondere wenn Funktionsaufrufe stark verschachtelt sind, wie im Testfall.

4.3 Leistungsvergleich von Interpreter und Compiler

Die folgenden Abbildungen zeigen einen Leistungsvergleich zwischen dem Interpreter und Compiler mit VM, indem die Ausführungszeiten identischer Programme gemessen und gegenübergestellt werden. Um die gemessenen Ergebnisse absolut zu bewerten, dient ein gleichartiges Programm in Python als Referenzmaßstab.

Abbildung 5 stellt die erforderliche Zeit für die rekursive Berechnung der Fibonacci-Folge mit $n = 35$ dar. Das hierfür verwendete Programm sieht wie folgt aus:

```
let fib = fn(n) {
    if (n < 2) {
        return n;
    };
    fib(n-1) + fib(n-2);
};
fib(35);
```

Listing 4.1: Rekursive Berechnung der Fibonacci-Folge in Monkey mit n

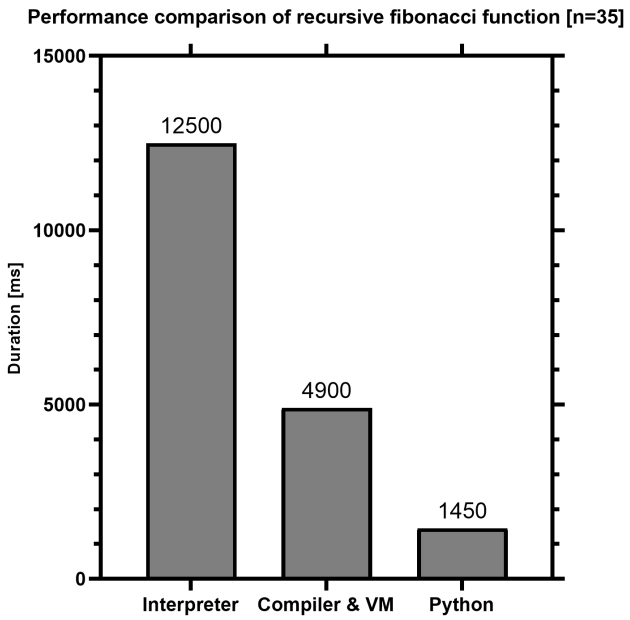

Abbildung 5: Leistungsvergleich von Interpreter, Compiler & VM und Python

Diese Auswertung bestätigt die erwartete Leistungssteigerung des Compilers und der VM im Vergleich zum Interpreter. Konkret sind Compiler und VM bei dieser Berechnung etwa 2,6-mal schneller als der Interpreter. Dieser Wert entspricht annähernd dem erwarteten Wert aus Thorsten Balls Werk „Writing a Compiler in Go“ [4]. Dort wurde eine dreifache Leistungssteigerung erzielt, jedoch unter unterschiedlichen Hardware-Bedingungen und mit Go als Host-Sprache statt Scala.

Es ist zu beobachten, dass Python die gleiche Berechnung in lediglich 1,45 Sekunden durchführt, was sowohl die Leistung des Interpreters als auch des Compilers und der VM deutlich übertrifft. Diese Ergebnisse illustrieren ein beträchtliches Potential für Leistungssteigerungen innerhalb der Implementierungen.

Der unmittelbare Vergleich mit Python birgt gewisse Limitationen, insbesondere aufgrund der Differenzen in den Abstraktionsebenen der verwendeten Programmiersprachen. Python ist in der hardwarenahen Sprache C implementiert, wohingegen Monkey in dem vorgestellten Interpreter bzw. Compiler umgesetzt wird. Diese werden von Scala ausgeführt, welches auf der JVM basiert, die ihrerseits in C++ geschrieben ist. Diese zusätzlichen Abstraktionsebenen können die direkte Leistungsmessung verzerren und den Vergleich mit Python komplexer gestalten.

Es ist wichtig zu betonen, dass bei dem zuvor verwendeten rekursiven Programm zur Berechnung der Fibonacci-Folge der Interpreter bei geringen Werten für n eine höhere Effizienz als der Compiler und die VM aufweist. Für weniger komplexe Programme stellt der Overhead des Kompilierens eine Belastung dar, die eine direkte Evaluation

durch den Interpreter zur effizienteren Option macht, im Vergleich zum getrennten Kompilieren und Ausführen. Die Ausführung eines bereits kompilierten Programms durch die VM ist jedoch in jedem Fall schneller als die Ausführung durch den Interpreter. Dies liegt daran, dass die Kompilierungszeit für das rekursive Fibonacci-Programm konstant bei etwa 30 Millisekunden bleibt und unabhängig von den Werten von n nicht ansteigt. Beim rekursiven Berechnen der Fibonacci-Folge überschreitet der Compiler und die VM erst ab $n = 22$ die Effizienz des Interpreters. Danach nimmt die Differenz in der benötigten Ausführungszeit im Vergleich zum Interpreter jedoch deutlich zu. Die Ergebnisse dieses Vergleichs sind in Abbildung 6 dargestellt:

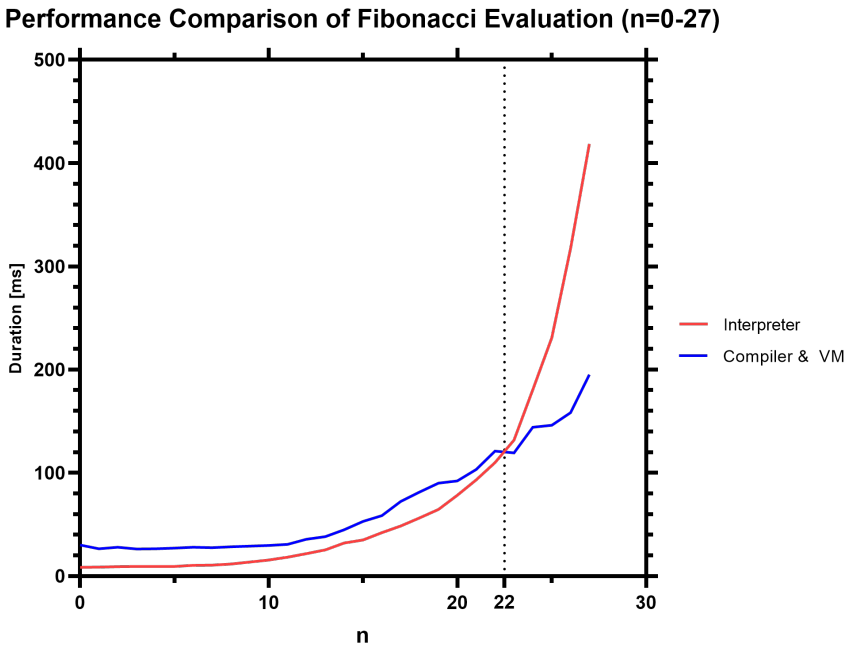

Abbildung 6: Vergleich der Ausführungszeiten für die Berechnung der Fibonacci-Sequenz mit n im Bereich [0, 27]

4.4 Effizienzsteigerung durch JVM-Warmup

Ein weiterer Aspekt bei der Ausführung von Programmen auf der JVM ist das sogenannte „JVM-Warmup“. Dabei optimiert die JVM durch die JIT-Kompilierung während der Laufzeit den Bytecode, basierend auf der tatsächlichen Code-Nutzung. Mit fortschreitender Ausführungszeit kann daher eine signifikante Verbesserung der Laufzeitleistung erreicht werden, da die JVM zunehmend optimierten Code generiert [22].

In Abbildung 7 ist dargestellt, wie sich das JVM-Warmup auf die Leistung der in dieser Arbeit implementierten Systeme auswirkt. Dazu wurde das rekursive Fibonacci-Programm mit $n = 20$ wiederholt ausgeführt und die benötigte Ausführungszeit für jede einzelne Durchführung gemessen.

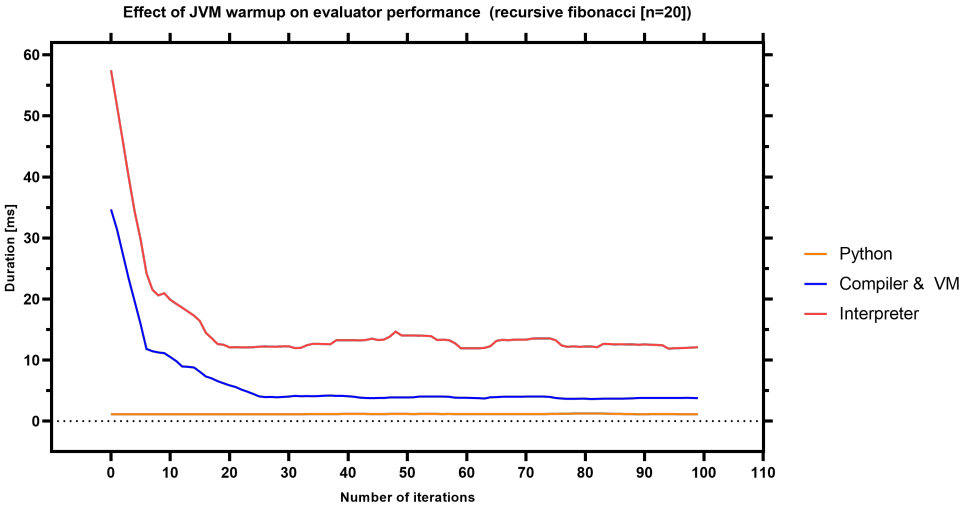

Abbildung 7: Leistungssteigerung durch JVM-Warmup im Vergleich zu Python

Die gemessenen Daten zeigen deutlich, dass sich die Ausführungszeit von Interpreter, Compiler und VM im Laufe der Zeit verbessert. Dies ist ein klares Indiz für die Optimierungen, die durch das JVM-Warmup ermöglicht werden. Es ist jedoch wichtig zu beachten, dass trotz dieser Leistungssteigerungen Python immer noch eine effizientere Ausführung bietet.

4.5 Vergleich mit anderen Implementierungen

Eine hilfreiche Methode zur Beurteilung der Qualität und Leistungsfähigkeit einer Implementierung ist der Vergleich mit anderen ähnlichen Implementierungen. Folgend wird die Leistung des entwickelten Interpreters mit der einer alternativen Implementierung vergleichen, die von Herrn Prof. Dr. Piepmeyer erstellt wurde.

Diese Implementierung verfolgt einen anderen Ansatz bei der Behandlung von Präzedenzen und der Implementierung des Parsers und Interpreters. Insbesondere wurden die Präzedenzen dabei rekursiv implementiert und der Parser und Interpreter in dieselben Datenstrukturen integriert. Diese Unterschiede in der Implementierung können Auswirkungen auf die Leistungsfähigkeit der resultierenden Interpreter haben.

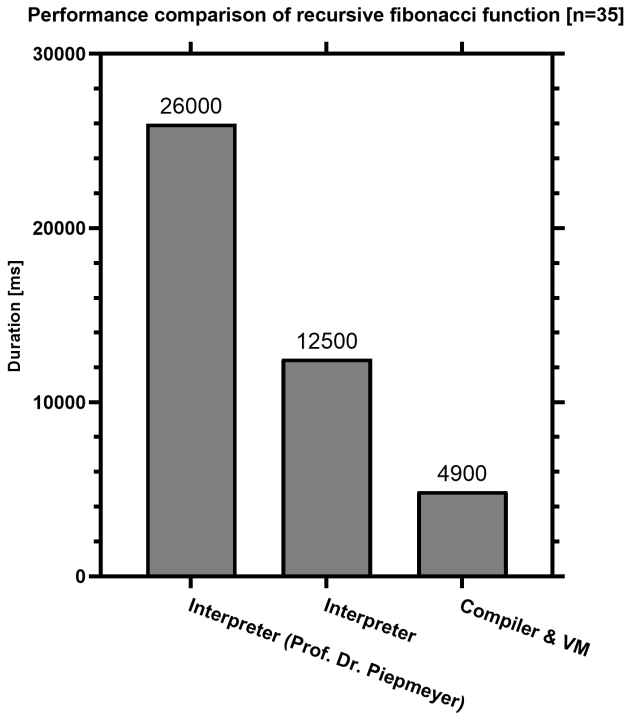

Abbildung 8: Leistungsvergleich mit dem Interpreter von Prof. Dr. Piepmeyer

In Abbildung 8 ist ein direkter Leistungsvergleich zwischen der Implementierung des Interpreters aus dieser Arbeit und der von Prof. Dr. Piepmeyer dargestellt. Dabei wurde das rekursive Fibonacci-Programm mit $n = 35$ als Testlauf verwendet und auf der selben Hardware ausgeführt. Die Ausführungszeit von Prof. Dr. Piepmeyers Implementierung ist etwa doppelt so lang wie die, des hier entwickelten Interpreters. Dies deutet darauf hin, dass die Wahl der zuvor vorgestellten Implementierungsstrategien, insbesondere die getrennte Behandlung von Parser und Interpreter und der iterative Ansatz zur Behandlung von Präzedenzen, einen positiven Einfluss auf die Leistungsfähigkeit des Interpreters hat.

Es ist jedoch wichtig zu beachten, dass es viele Faktoren gibt, die die Leistung einer Implementierung beeinflussen können. Unterschiedliche Implementierungen können unterschiedliche Stärken und Schwächen aufweisen. Trotz der schlechteren Ausführungszeit in diesem speziellen Testfall kann die Implementierung von Herrn Piepmeyer bestimmte Vorteile bieten. Sie ist beispielsweise insgesamt kürzer und kompakter gehalten. Diese Vergleiche dienen dazu, Einblicke in die Auswirkungen bestimmter Design- und Implementierungsentscheidungen auf die Leistung zu gewinnen und Bereiche für potenzielle Verbesserungen zu identifizieren.

5 Schlussfolgerung

5.1 Zusammenfassung der Ergebnisse

Die in dieser Arbeit vorgestellten Parser zeigten eine vollständige Erfüllung der spezifizierten EBNF-Grammatik und konnten die eingegebenen Programme korrekt analysieren und daraus den entsprechenden AST erzeugen. Diese Funktionalität wurde durch umfangreiche Tests sichergestellt. Des Weiteren konnten der Interpreter, der Compiler und die VM die ihnen zugeführten Programme korrekt interpretieren bzw. kompilieren und ausführen.

Bei der Leistungsbewertung wurde deutlich, dass die Zeit, die beide Parser-Versionen zum Parsen des Textes benötigen, wie erwartet linear mit der zunehmenden Textlänge steigt. Dennoch schnitt der Parser, der die Scala Parser Combinators verwendet, im Durchschnitt etwa 18-mal schlechter ab als der konventionelle Parser. Dieser Leistungsunterschied wurde auf die höhere Rekursionstiefe, den zusätzlichen Overhead durch die Generizität und die Verwendung von Backtracking zurückgeführt.

Beim Leistungsvergleich zwischen dem Interpreter und dem Compiler und der VM zeigten Letztere grundsätzlich eine höhere Leistung. Python übertraf jedoch sowohl den Interpreter als auch den Compiler und die VM. Dies lässt Spielraum für weitere Leistungsoptimierungen innerhalb der implementierten Systeme.

Es wurde auch festgestellt, dass der Interpreter bei weniger komplexen Programmen eine höhere Effizienz als der Compiler und die VM aufweist, was auf den Overhead des Kompilierens zurückzuführen ist. Bei größeren Programmen übertraf jedoch die Kombination aus Compiler und VM den Interpreter in der Leistung deutlich.

Schließlich zeigte der Effekt des JVM-Warmups eine deutliche Verbesserung der Laufzeitleistung, da die JVM zunehmend optimierten Code generierte. Bei wiederholter Ausführung des gleichen Programms konnten daher signifikante Leistungssteigerungen erreicht werden.

5.2 Anwendungsfälle von Scala Parser Combinators

Trotz der Leistungsunterschiede, die im Vergleich der beiden Parser festgestellt wurden, bedeutet dies nicht, dass der Parser mit Scala Parser Combinators keine Daseinsberechtigung hat. Dieser Ansatz hat spezifische Anwendungsfälle und Vorteile, die ihn unter bestimmten Umständen zu einer bevorzugten Option machen können:

Die Scala Parser Combinators zeichnen sich durch ihre ausdrucksstarke, deklarative Syntax und ihre hervorragende Unterstützung für das Parsen von formalen Sprachen aus. Sie ermöglichen es, Grammatiken auf eine Art und Weise zu definieren, die eng an die EBNF-Notation angelehnt sind und die der EBNF-Notation in ihrer Lesbarkeit und Verständlichkeit ähneln. Dadurch sind sie insbesondere für Entwicklerinnen und Entwickler attraktiv, die eine schnelle, einfache und intuitive Art und Weise suchen, um formale Sprachen oder Grammatiken zu parsen. Darüber hinaus bieten die Scala Parser Combinators eingebaute Unterstützung für Fehlermeldungen und die Erkennung von Fehlern während des Parsens.

5.3 Anwendungsfälle des Interpreters

Der Interpreter hat trotz seiner geringeren Leistung im Vergleich zu Compiler und VM einige Vorteile, die in bestimmten Anwendungsfällen seine Verwendung rechtfertigen.

Einer dieser Vorteile ist, dass der Interpreter keinen Kompilierungsschritt erfordert und somit Programme direkt ausführt. Dies kann in Entwicklungsumgebungen, in denen häufig Änderungen vorgenommen und getestet werden, vorteilhaft sein. Ein solcher Ansatz ermöglicht es, den Code Schritt für Schritt auszuführen und zu testen, was das Debugging erleichtert. Bei Kompilierungsschritten können Fehler auftreten, die schwer zu verfolgen sind, welche durch die Interpretation vermieden werden können.

Darüber hinaus kann der Interpreter für kleinere Programme, bei denen der Overhead des Kompilierungsprozesses signifikant ist, effizienter sein. Die Kompilierung kann besonders bei großen Programmen viel Zeit in Anspruch nehmen, wodurch der Interpreter daher schneller sein kann, falls die eigentliche Berechnung des Programms einfach ist.

Ein weiterer Vorteil des Interpreters ist, dass er in der Regel weniger Kompatibilitätsprobleme aufweist. Compiler müssen normalerweise für spezifische Architekturen oder Betriebssysteme erstellt werden, während Interpreter im Allgemeinen plattformübergreifend sind. Dieses Problem wurde in dieser Arbeit durch die Entwicklung eines Bytecode-Compilers und einer VM umgangen, was jedoch einen wesentlich höheren Entwicklungsaufwand im Vergleich zum Interpreter darstellte.

5.4 Fazit

Im Rahmen dieser Arbeit konnten alle gestellten Ziele (siehe Abschnitt 1.2) erfolgreich umgesetzt werden. Ein wesentlicher Bestandteil der Arbeit war der Vergleich und die Analyse der Leistungsfähigkeit zwischen einem konventionellen Parser und einem Parser, der auf Scala Parser Combinators basiert. Unter Berücksichtigung der jeweiligen Vor- und Nachteile beider Ansätze konnte ein Verständnis für die Funktionsweise und die Anwendungsfälle dieser Techniken entwickelt werden.

Des Weiteren wurde die Funktion und Effizienz eines Interpreters mit der eines Compilers und einer VM gegenübergestellt. Dieser Vergleich trug dazu bei, die Leistungsvorteile und -nachteile der verschiedenen Ansätze von Interpretern, Compilern und virtuellen Maschinen aufzudecken.

Ein zentraler Aspekt der Arbeit war die Implementierung der beiden vorgestellten Parser-Versionen und eines Interpreters für Monkey. Dadurch konnten die theoretischen Konzepte in praktische Anwendungen überführt werden.

Die Arbeit dient als Handbuch zur Implementierung eines Parsers und Interpreters und bietet somit eine praxisnahe Unterstützung für alle, die sich in diesem Bereich weiterentwickeln möchten. Durch die aufgeführten Implementationsstrategien und Designentscheidungen in der Architektur von Parsern und Interpretern liefert die Arbeit eine Entscheidungshilfe für zukünftige Projekte in diesem Bereich.

5.5 Ausblick auf weiterführende Arbeiten

Es gibt mehrere Bereiche, die in zukünftigen Projekten verbessert werden können, um die Effizienz und Funktionen der vorgestellten Parser, sowie des Compilers und der VM zu optimieren.

Die Leistung und der Speicherbedarf des Parsers, der die Scala Parser Combinators verwendet, könnten durch den Einsatz von „Packrat-Parsing“ optimiert werden. Diese Technik verbessert die Effizienz von Parsern, indem sie Zwischenergebnisse in einem Cache speichert und so redundantes Parsen vermeidet. Packrat-Parsing ist eine Erweiterung des Recursive-Descent Algorithmus, die Backtracking effizienter gestaltet und somit die Verarbeitungsgeschwindigkeit und den Speicherbedarf verbessern kann. Weitere Optimierungen können durch die Verwendung von „Makros“ und „Lazy-Evaluation“ in Kombination mit Scala Parser Combinators erreicht werden [23].

Für konventionelle Parser wird von [24] vorgeschlagen, den Parser und Lexer in einer Datenstruktur zu kombinieren. Dadurch kann die Leistung nochmals erheblich verbessert werden, ohne dass Modularität oder Flexibilität verloren geht.

Darüber hinaus wurde in dieser Arbeit die Implementierung des Compilers und der VM lediglich auf einem grundlegenden Niveau vorgestellt. Eine detailliertere Beschreibung der dazugehörigen Algorithmen in zukünftigen Arbeiten könnte ein tieferes Verständnis der Funktionsweise und der potenziellen Optimierungsmöglichkeiten ermöglichen.

Eine weitere Möglichkeit zur Steigerung der Leistungsfähigkeit liegt in der Entwicklung eines Register-basierten Compilers, anstatt eines Stack-basierten Compilers. Register-basierte Compiler verwenden eine begrenzte Anzahl von Registern zur Speicherung und Manipulation von Daten, im Gegensatz zu Stack-basierten Compilern, die Operationen ausführen, indem sie Daten auf einen Stapel legen und daraus entfernen. Durch die Reduktion der notwendigen Operationen kann die Ausführungsgeschwindigkeit verbessert werden [7].

Literaturverzeichnis

[1] Monkey. Accessed: 2023-05-12. [Online]. Available: https://monkeylang.org/

[2] scala-parser-combinators. Accessed: 2023-06-05. [Online]. Available: https://index.scala-lang.org/scala/scala-parser-combinators

[3] T. Ball, *Writing an Interpreter in Go*. Thorsten Ball, 2019, version 1.6.

[4] ——, *Writing a Compiler in Go*. Thorsten Ball, 2019, version 1.2.

[5] C. S. Saxon, "Object-oriented recursive descent parsing in c#," *SIGCSE Bull.*, vol. 35, no. 4, p. 82–85, dec 2003. [Online]. Available: https://doi.org/10.1145/960492.960534

[6] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, "Aot vs. jit: Impact of profile data on code quality," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–10. [Online]. Available: https://doi.org/10.1145/3078633.3081037

[7] C. Zimmer, S. R. Hines, P. Kulkarni, G. Tyson, and D. Whalley, "Facilitating compiler optimizations through the dynamic mapping of alternate register structures," in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 165–169. [Online]. Available: https://doi.org/10.1145/1289881.1289912

[8] H.-J. Böckenhauer and J. Hromkovic, *Formale Sprachen*. Springer-Verlag, 2012.

[9] D. D. McCracken and E. D. Reilly, *Backus-Naur Form (BNF)*. GBR: John Wiley and Sons Ltd., 2003, p. 129–131.

[10] C. Hoekstra, "Combinatory logic and combinators in array languages," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ser. ARRAY 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–57. [Online]. Available: https://doi.org/10.1145/3520306.3534504

[11] J. E. Friedl, *Mastering regular expressions*. Ö'Reilly Media, Inc.", 2006.

[12] V. R. Pratt, "Top down operator precedence," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 41–51. [Online]. Available: https://doi.org/10.1145/512927.512931

[13] R. C. Moore, "Removing left recursion from context-free grammars," in *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, ser. NAACL 2000. USA: Association for Computational Linguistics, 2000, p. 249–255.

[14] sbt - the interactive build tool. Accessed: 2023-06-05. [Online]. Available: https://www.scala-sbt.org/

[15] Parsing packages in scala. Accessed: 2023-06-05. [Online]. Available: https://index.scala-lang.org/awesome/parsing?sort=stars

[16] scala-parser-combinators/javatokenparsers.scala. Accessed: 2023-06-10. [Online]. Available: https://github.com/scala/scala-parser-combinators/blob/v1.0.6/shared/src/main/scala/scala/util/parsing/combinator/JavaTokenParsers.scala

[17] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin, "Orbit: An optimizing compiler for scheme," *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 219–233, 1986.

[18] J.-j. Zhao, "Static analysis of java bytecode," *Wuhan University Journal of Natural Sciences*, vol. 6, no. 1, pp. 383–390, Mar 2001. [Online]. Available: https://doi.org/10.1007/BF03160273

[19] J. Urquiza-Fuentes, F. Manso, J. A. Velázquez-Iturbide, and M. Rubio-Sánchez, "Improving compilers education through symbol tables animations," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 203–207. [Online]. Available: https://doi.org/10.1145/1999747.1999805

[20] Lindig, "Declarative composition of stack frames," in *Compiler Construction*, E. Duesterwald, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 298–312.

[21] L. Cardelli, "Compiling a functional language," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ser. LFP '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 208–217. [Online]. Available: https://doi.org/10.1145/800055.802037

[22] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133876

[23] E. Béguet and M. Jonnalagedda, "Accelerating parser combinators with macros," in *Proceedings of the Fifth Annual Scala Workshop*, ser. SCALA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 7–17. [Online]. Available: https://doi.org/10.1145/2637647.2637653

[24] J. Yallop, N. Xie, and N. Krishnaswami, "Flap: A deterministic parser with fused lexing," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591269

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Leon Steinbach

Leon Steinbach
CN=Leon Steinbach, E=leon@steinba.de, C=DE

2023.06.25 21:04:47+02'00'

Buchenbach, 30.06.2023 Leon-César Steinbach