Entwicklung und Programmierung eines rJava-Bytecode-Compilers in JAVA

Fach: Informatik
Fachlehrer: Herr Fassbender

Johannes Bockhorst, 2023

Inhaltsverzeichnis

1 Einleitung 3
2 Die Programmiersprache „rJava“ 3
3 Planung des Compilers 4
3.1 Ziele 4
3.2 Vorgehen 5
3.3 Struktur 5
4 Lexer 6
5 Parser 9
5.1 Der Syntaxbaum 9
5.2 Parsen der Expressions 10
5.3 Parsen der Statements 13
6 Überprüfungsschritt 15
7 Überführung in JVM-Bytecode 18
7.1 JVM und der JVM-Bytecode 18
7.2 Implementation des Code-Generators 20
8 Fazit 23
8.1 Fachbezogene Reflexion und Ausblick 23
8.2 Persönliches Fazit 24
9 Literaturverzeichnis 26
10 Anhang 27
10.1 Beispielprogramm „Fibonacci“ 27
10.2 Beispielprogramm „Sierpinski Dreieck“ 28
10.3 Beispielprogramm „Mandelbrot-Menge“ 29

1 Einleitung

Programmiersprachen bilden heute die Grundlage für jegliche Software, die auf Servern, auf dem Smartphone oder auf dem Heimrechner genutzt wird. Wo früher noch hunderte Lochkarten genutzt wurden, um ein Rechenprogramm zu spezifizieren, finden heute Sprachen mit ausgeprägter Syntax und Semantik Anwendung. Diese Entwicklung hat die Softwareentwicklung um ein Vielfaches effizienter gemacht, da Programme nun für den Menschen weitaus verständlicher sind und beispielsweise das Finden von Fehlern vereinfacht wurde.

Damit jedoch ein Computer ein Programm, welches in einer Programmiersprache festgehalten wurde, interpretieren und ausführen kann, muss jenes Programm zunächst in ein für den Computer verständliches Format übersetzt werden. Dafür ist ein Programm, das im allgemeinen als „Compiler“ bezeichnet wird, notwendig, welches den Quellcode analysiert und verarbeitet.

Mit der Erstellung und Programmierung eines solchen Compilers in Java soll sich in dieser Lernleistung befasst werden.

Die dem Quelltext zu Grunde liegende Programmiersprache soll dabei die fiktive Programmiersprache „rJava“ sein, welche sich an den grundlegenden Strukturen von „Java“ orientiert. rJava-Programme sollen dann mithilfe des Compilers – wie bei einem Java-Compiler – zu einer „.class“-Datei kompiliert werden. Solche Dateien lassen sich dann mithilfe der Java Virtual Machine (JVM) ausführen.

2 Die Programmiersprache „rJava“

Vor der Implementation des Compilers ist es notwendig, die Eingabesprache rJava genauer festzulegen und Ziele zu setzen, welche Strukturen enthalten sowie welche Programme in rJava implementierbar sein sollen.

Zu den Features von Java, die in rJava übernommen werden, gehören

- grundlegende Rechenoperatoren (zum Beispiel „+“, „-“, „*“, „/“),
- die wichtigsten Syntax-Statements (zum Beispiel „if“-Abfragen und „while“-Schleifen),
- Methoden- bzw. Funktionsaufrufe und
- Variablen mit den primitiven Datentypen „int“, „boolean“, und „char“.

Bei der Frage, welche Programme in einer Programmiersprache implementierbar seien, wird sich häufig auf die Turing-Vollständigkeit bezogen. Oft wird davon ausgegangen, sollte eine Programmiersprache Turing-vollständig sein, könne jede Berechnung in jener

Programmiersprache implementiert werden. Dies ist jedoch eine ungenaue Definition. Eine exakte Definition beschreibt eine Turing-vollständige Programmiersprache als eine Programmiersprache, mit der eine universelle Turing-Maschine simuliert werden kann[1].
Auf einen formalen Beweis, ob dieses Kriterium auf rJava zutrifft, soll hier verzichtet werden. Dennoch lässt sich auf triviale Weise beobachten, dass rJava eine sehr begrenzte Programmiersprache ist. Ausschlaggebend dafür ist der Verzicht auf dynamische Datenstrukturen. So müsste beispielsweise bei einer Implementation von Conway's Game of Life jede Zelle des Spielbretts durch eine eigene Variable repräsentiert werden. Dies ist in der Praxis nicht nur sehr umständlich, sondern macht eine Implementation eines Spielbretts mit willkürlichen Dimensionen unmöglich.
Trotz dieser Einschränkung, sowie der offensichtlichen Abwesenheit von Klassen- und Methodendefinitionen, lässt sich eine Vielzahl von Programmen in rJava implementieren. Wie dem Anhang zu entnehmen ist, gehören dazu

- ein Programm zur Berechnung der Fibonaccizahlen,
- ein Programm zur Ausgabe eines Sierpinski-Dreiecks und
- sogar eine Visualisierung der Mandelbrot-Menge.

3 Planung des Compilers

In diesem Kapitel soll die Planung der Compilersoftware dokumentiert werden.

3.1 Ziele

Zunächst ist es wichtig, die genaue Funktion der Compilersoftware festzulegen:

1. Quellcode, der in rJava verfasst wurde, soll aus einer Text-Datei eingelesen und verarbeitet werden.
2. Das Programm wird auf syntaktische und semantische Fehler überprüft. Gegebenenfalls soll auch eine informative Fehlermeldung ausgegeben werden.
3. Am Ende wird JVM-Assembly (der Code, der in der Theorie von der JVM ausgeführt werden kann) generiert.

Dass der generierte JVM-Assemblycode nur in der Theorie von der JVM ausgeführt werden kann, lässt sich damit begründen, dass unser Compiler nicht direkt eine binäre „.class"-Datei ausgeben wird. Stattdessen wird eine Text-Datei ausgegeben, die den JVM-Bytecode beschreibt. Diese Datei kann dann mithilfe eines externen Tools „Jasmin" in eine „.class"-Datei überführt werden.

---

[1] Vgl. Turing Completeness CS390, Spring 2023 (Quelle 1)

Ich habe mich hier dafür entschieden, diesen Umweg über Jasmin zu wählen, da eine eigene Implementation eines „.class“-Datei-Generators den ohnehin schon großen Rahmen dieser Lernleistung sprengen würde.

3.2 Vorgehen

Bei der Implementation eines Compilers ist es in der Regel üblich, den Compiler in folgende Schritte zu gliedern:

1. Der „Lexer“ oder „Tokenizer“, der den Quelltext in einzelne syntaktische Einheiten, sogenannte „Tokens“, unterteilt.
2. Der „Parser“, der Tokens zu syntaktischen Strukturen wie Expressions oder Statements zusammenfügt und am Ende einen Syntax-Baum ausgibt.
3. Die semantische Analyse, in der sich auf die Fehlerfindung konzentriert wird. Beispielsweise wird das Programm auf die fehlerhafte Nutzung von Rechenoperatoren und Referenzen von nicht definierten Variablen überprüft.
4. Übersetzung in einen oder mehrere Zwischencode(s). An dieser Stelle wird normalerweise der Syntaxbaum in beliebig vielen Schritten in diverse Zwischenrepräsentationen überführt. Dies kann zum Beispiel der Optimierung des Ausgabeprogramms dienen. Im Fall unseres rJava-Compilers soll jedoch auf Programmoptimierung weitestgehend verzichtet werden.
5. Im letzten Schritt wird dann die Zwischenrepräsentation des Compilers in das Ausgabeformat übersetzt und ausgegeben. Dieser Teil des Compilers wird meist „Codegenerator“ genannt.

3.3 Struktur

Da sich unser Compiler – wie in 3.2 beschrieben – in vier klar getrennte Einheiten gliedern lässt, bietet es sich an, für jede dieser Einheiten eine eigene Klasse anzulegen. Neben den Klassen `Lexer`, `Parser` und `Codegenerator` wird der Überprüfungsschritt in der `TypeChecker`-Klasse implementiert.

Der Syntaxbaum wird mithilfe der Klassen `Token`, `Expression` und `Statement` repräsentiert. Während `Tokens` die kleinste syntaktische Einheit bilden, sind `Expression` und `Statement` abstrakte Klassen, deren Unterklassen die übergeordneten syntaktischen Strukturen bilden. Eine `Expression` kann dabei sowohl aus `Token`s als auch aus anderen `Expression`s bestehen. `Statement`s sind übergeordnete Strukturen, die jeweils aus mehreren `Expression`s, `Token`s und Unter-`Statement`s zusammengesetzt sein können.

Die Fehlerbehandlung wird mithilfe der `Errors`- und `Location`-Klassen realisiert. Die `Errors`-Klasse hat dabei lediglich statische Methoden, die einen Fehler ausgeben und das Programm danach direkt beenden. Man mag zwar von Compilern gewohnt sein, dass nach einmaligem Ausführen bereits mehrere Fehler ausgegeben werden. Ich habe mich hier zur Vereinfachung gegen eine solche Art der Fehlermeldung entschieden. Die `Location`-Klasse wird dabei genutzt, um die Stelle im Quelltext anzugeben, an der es zu einem Fehler kam. Jedem `Token`, jeder `Expression` und jedem `Statement` wird im Kompilierungsprozess dann eine `Location` zugeordnet. Im Falle, dass ein Fehler beim Traversieren des Syntaxbaums gefunden wird, kann dieser so immer einer `Location` zugeordnet werden.
Auf die genaue Implementation der hier beschriebenen Klassen soll in den folgenden Kapiteln eingegangen werden.

4 Lexer

Der erste Schritt des Kompilierungsprozesses ist wie zuvor beschrieben in der `Lexer`-Klasse implementiert. Einem `Lexer`-Objekt wird im Konstruktor lediglich der Dateipfad des Quelltextes übergeben. Die jeweilige Textdatei wird dann als komplettes `String`-Objekt eingelesen. Neben dem Quelltext-`String source` hat die `Lexer`-Klasse die Datenfelder `currentLocation` und `position`, die Auskunft darüber geben, an welcher Stelle im Quelltext der `Lexer` momentan arbeitet. `currentLocation` ist dabei ein Objekt der `Location`-Klasse, welches die momentane Zeile und Spalte im Quelltext sowie den Dateipfad der Quelltextdatei speichert. `position` dient hingegen als Zeiger, der den Index im Quelltext-String als `int`-Wert speichert.
Zentral für den `Lexer` ist die Methode `nextToken`. Hier analysiert der Lexer, welche Art von Token an der momentanen Stelle im Quelltext zu finden ist, setzt den Wert von `position` und `currentLocation` an die nächste Position nach jenem `Token` und gibt ein jeweiliges `Token`-Objekt als `return`-Wert zurück.
Jedes `Token`-Objekt speichert dabei seine Position im Quelltext als `Location`-Objekt, die Art des Tokens als Aufzählungswert der Aufzählung `TokenType` und den zugrunde liegenden Text als `String`. Zu separaten Aufzählungswerten von `TokenType` gehören

- Sonderzeichen und besondere Zeichenfolgen (z.B. „=“, „+“, „>=“, „&&“),
- vorgegebene Schlüsselwörter (z.B. „if“ oder „int“),
- benutzerdefinierte Variablennamen,
- Literale (z.B. `int`-Literale wie „23“) und

- der End-of-File Token, der das Ende der Quelltextdatei kennzeichnet.

Wichtig zu beachten ist dabei, dass jegliche Leerzeichen vom `Lexer` weder als `Token` gespeichert noch in sonst einer Weise berücksichtigt werden. Dieses Feature soll das Verhalten von Java imitieren, ist aber auch in vielen anderen modernen Programmiersprachen zu finden.

Implementiert wird dies in der `nextToken`-Methode, in der zu Beginn so lange die aktuelle Position im Quelltext erhöht wird, bis kein Leerzeichen mehr an erster Stelle steht.

```
while (Character.isWhitespace(peekChar())) {
    advance();
}
```

Darauf folgt ein `switch`-Statement, welches je nachdem, welches Zeichen an der momentanen Stelle im Quelltext steht, anders verfährt:

- Bei Sonderzeichen ist das Vorgehen weitestgehend trivial. Beispielsweise muss, sollte ein „+“ aufgefunden werden, lediglich die Position im Quelltext um eine Stelle aufgerückt und ein jeweiliger Token zurückgegeben werden:

  ```
  case '+': return new Token(start, TokenType.Plus,    "+");
  ```

- Bei Zeichen wie „>“ muss zusätzlich das nächste Zeichen berücksichtigt werden: Sollte es sich um „=“ oder „>“ handeln, ist eine andere Art `Token` zurückzugeben, als wenn das „>“-Zeichen „alleine“ steht. So könnte es sich sowohl um ein „größer als“, als auch um ein „größer-gleich“ oder den binären Operator „>>“ handeln:

  ```
  case '>':
  if (peekChar() == '=') {
      advance();
      return new Token(start, TokenType.GreaterEqual, ">=");
  } else if (peekChar() == '>') {
      advance();
      return new Token(start, TokenType.GreaterGreater, ">>");
  } else {
      return new Token(start, TokenType.Greater, ">");
  }
  ```

- Einen Sonderfall bilden Quelltextkommentare. Treten zwei „/“-Zeichen hintereinander auf, so soll der Rest der Zeile im Quelltext ignoriert werden, um dem User die Möglichkeit zu geben, seinen Quelltext mit willkürlichen Zeichenfolgen zu kommentieren. Hier wird über eine `while`-Schleife die Position im Quelltext erhöht, bis das Ende der Zeile oder das Ende des Quelltextes erreicht

wird. Damit die `nextToken`-Methode hier auch einen Rückgabewert liefert, wird `nextToken` rekursiv aufgerufen und das Ergebnis jenes Aufrufes zurückgegeben:

```
case '/':
if (peekChar() == '/') {
    while (peekChar() != '\n' && peekChar() != '\0') advance();
    return nextToken();
}
```

- Um Zahlenwerte als eigene `Token`s einzulesen, wird im `default`-Zweig des `switch`-Statements mithilfe der `Character.isDigit`-Methode überprüft, ob das nächste Zeichen im Quelltext eine Ziffer ist. Anschließend wird, solange an der momentanen Position eine Ziffer steht, die Position erhöht und ein `IntLiteral-Token` zurückgegeben. Der Zahlenwert wird dabei mit der `substring`-Methode dem Quelltext entnommen und als Text des zurückgegebenen `Token`s gespeichert:

```
if (Character.isDigit(next)) {
    int startPos = position-1;
    while (Character.isDigit(peekChar())) {
        advance();
    }
    String number = source.substring(startPos, position);
    return new Token(start, TokenType.IntLiteral, number);
}
```

- Beim Einlesen der Variablen und Schlüsselwörter wird ähnlich wie bei Zahlenwerten vorgegangen. Hier wird in der Bedingung des `if`-Statements auf ein alphabetisches Zeichen überprüft. In der Bedingung der `while`-Schleife werden hingegen sowohl alphabetische als auch numerische Zeichen akzeptiert, um Variablennamen wie „x1“ oder „button23“ zu erlauben. Zur Ermittlung der Art des `Token`s wird hier eine Hashtabelle verwendet, in der jedem Schlüsselworttext ein `TokenType`-Wert zugeordnet wird. Wird der Text des so ermittelten `Token`s in jener Hashtabelle gefunden, handelt es sich um ein Schlüsselwort und der jeweilige `TokenType`-Wert kann der Hashtabelle entnommen werden. Andernfalls wird der `Token` als Variablenname klassifiziert:

```
if (Character.isAlphabetic(next)) {
    int startPos = position-1;
    while (Character.isLetterOrDigit(peekChar())) {
        advance();
    }

    String name = source.substring(startPos, position);
    TokenType type = keywordTable.getOrDefault(name, TokenType.Variable);
    return new Token(start, type, name);
}
```

Anwendung findet die `nextToken`-Methode in der `collectTokens`-Methode der `Lexer`-Klasse. Diese fügt `Token`s so lange einer Liste hinzu, bis ein End-of-File-Token erreicht wird. Diese Liste wird dann ausgegeben und im weiteren Verlauf des Kompilierungsprozesses verwendet.

5 Parser

5.1 Der Syntaxbaum

Die Aufgabe des Parsers besteht also darin, die lineare Token-Liste in einen nicht-linearen Syntaxbaum zu überführen. Diese Baumstruktur ist im Falle unseres Compilers über die abstrakten Klassen `Expression` und `Statement` implementiert. Eine Baumstruktur entsteht dabei völlig natürlich aufgrund der verschachtelten Struktur einer Programmiersprache. So lässt sich beispielsweise auf beiden Seiten eines „+“-Operators sowohl eine einzelne Zahl, eine Variable oder auch ein Funktionsaufruf, in dessen Argument eine weitere „+“-Operation steht, einsetzten:

- `1 + 2`
- `a + b`
- `f(1+2)+g(3+4)`

Für die linke und rechte Seite des „+“-Operators muss also ein beliebiger Ausdruck einsetzbar sein, sodass die jeweilige Klasse in der Implementation auf die allgemeine Expression-Klasse verweist. So entsteht eine rekursive Klassendefinition, wie sie auch bei einem Baum verwendet wird.

Zu den verschiedenen `Expression`-Arten und somit zu den Unterklassen der `Expression`-Klasse gehören

- `BinaryExpression` (die diversen Operationen mit zwei Operanden wie „+“, „-“, „*“ oder „/“),
- `UnaryExpression` (Operatoren mit nur einem Operanden, z.B. „`!true`“ oder „-3“)
- `LiteralExpression` (Literale wie „23“ oder „true“),

- `VariableExpression` (Ausdrücke für Variablen)
- `CallExpression` (Funktionsaufrufe wie „print(23)“)
- `AssignmentExpression` (Ausdrücke zur Belegung von Variablen),
- `CastExpression` (Typecasts, z.B. „`(int)'A'`“)

Dabei ist erneut zu beachten, dass einige `Expression`-Arten sich aus mehreren Unter-`Expression`s zusammensetzen müssen (z.B. `BinaryExpression` oder `CallExpression`) während andere aus einem einzelnen Token zusammengesetzt sind (z.B. `VariableExpression`). Die letzteren `Expression`-Arten bilden die Blätter des Syntaxbaumes.

Ähnliches ist bei der `Statement`-Klasse zu beobachten. So setzt sich zum Beispiel ein `IfStatement` aus einer `Expression` als Bedingung, einem weiteren `Statement` als „then“-Zweig und optional einem weiteren `Statement` als „else“-Zweig.

Zu den `Statement`-Unterklassen gehören

- `ExpressionStatement` (Ein `Statement`, das aus einer einzelnen `Expression`s besteht),
- `DefinitionStatement` (Ein `Statement` zur Definition einer Variable, z.B. „int a = 23;“),
- `WhileStatement` (Eine `while`-Schleife),
- `BreakStatement` und `ContinueStatement` (Kontrollstruktur-`Statement`s, die in `while`-Schleifen genutzt werden),
- `IfStatement`,
- `CompoundStatement` (Eine Kette von `Statement`s, die von zwei geschweiften Klammern eingeschlossen ist).

5.2 Parsen der Expressions

Sowohl beim Parsen der `Expression`s als auch beim Parsen der Statements sind die Hilfsmethoden `peekToken`, `consumeToken` und `expectToken` der `Parser`-Klasse von zentraler Bedeutung. Während `peekToken` den `Token` an der nächsten Stelle in der `Token`-Liste zurückgibt, wird bei `consumeToken` zusätzlich der `position`-Zeiger um eine Stelle erhöht. Dazu ist zu bemerken, dass, ähnlich wie die `Lexer`-Klasse, die `Parser`-Klasse ein Datenfeld `position` hat, welches die aktuelle Position in der `Token`-Liste als `int`-Wert speichert. Die Methode `expectToken` erleichtert die Fehlerbehandlung im Parser um ein Vielfaches. Sie funktioniert ähnlich wie die `consumeToken`-Methode, übernimmt jedoch zusätzlich als Parameter einen

`TokenType`-Wert. Dieser wird mit dem `TokenType`-Wert des nächsten Tokens verglichen. Stimmen die `TokenType`-Werte nicht überein, so wird ein Fehler ausgegeben. Da an vielen Stellen im `Parser` von vornhinein bekannt ist, welcher Token zu erwarten ist, kann der notwendige Code so maßgeblich komprimiert werden.

Die zentrale Methode für das Parsen der `Expressions` ist die `parseBinaryExpression`-Methode. Diese implementiert einen sogenannten Precedence-Climbing-Algorithmus, welcher eine gängige Methode zum Parsen von arithmetischen und logischen Ausdrücken ist. In Pseudocode lässt sich der Algorithmus so darstellen:

```
parseBinaryExpression(elternPräzedenz)
    linkeSeite = parsePrimäreExpression()
    operator = nächsterToken()
    präzedenz = gibPräzedenz(operator)
    SOLANGE istValiderOperator(operator) und präzedenz >= elternPräzedenz:
        überspringeToken()
        rechteSeite = parseBinaryExpression(elternPräzedenz)
        linkeSeite = BinaryExpression(linkeSeite, operator, rechteSeite)
        operator = nächsterToken()
        präzedenz = gibPräzedenz(operator)
    gib linkeSeite zurück
```

Vereinfacht ausgedrückt ist `parseBinaryExpression` eine rekursive Methode, die Anhand der Präzedenz, der Priorität eines Operators, einen binären Syntaxbaum aufbaut. Dieser Syntaxbaum soll die Bedingung erfüllen, dass die Operationen, die zuerst ausgeführt werden sollen, „weiter unten“ und „weiter links“ im Baum als die folgenden Operationen stehen. So kann die richtige Reihenfolge der Operationen über eine Post-Order-Traversierung in Erfahrung gebracht werden.

Die `Expression`-Typen neben `BinaryExpressions` gehen hauptsächlich aus der `parsePrimaryExpression`-Methode, welche von der `parseBinaryExpression`-Methode verwendet wird, hervor. Hier wird über ein `switch`-Statement anhand des `TokenType`-Wertes des nächsten Tokens entschieden, welche Art von `Expression` vorliegt:

- Bei `int`- und `char`-Literalen sowie einem „`true`“- oder „`false`“-Schlüsselwort kann davon ausgegangen werden, dass eine `LiteralExpression` zurückzugeben ist:

  ```
  return new LiteralExpression(consumeToken());
  ```

- Ist der nächste `Token` ein `Token` vom Typen `TokenType.Variable`, also eine beliebige Folge alphanumerischer Zeichen, so kann es sich sowohl um eine Variable als auch um einen Funktionsaufruf handeln. Wie verfahren wird, hängt davon ab, ob der nächste Token ein „(“-Token ist. In diesem Fall wird in einer

`while`-Schleife die Argumentliste geparst und eine `CallExpression` zurückgegeben. Andernfalls wird eine `VariableExpression` zurückgegeben:

```
Token variable = consumeToken();
if (peekToken().type == TokenType.OpenParenthesis) {
    ArrayList<Expression> parameters = new ArrayList<>();
    do {
        consumeToken();
        Expression p = parseExpression();
        parameters.add(p);
    } while (peekToken().type == TokenType.Comma);
    expectToken(TokenType.CloseParenthesis, "at the end of call expression");
    return new CallExpression(variable, parameters);
} else {
    return new VariableExpression(variable);
}
```

- Bei einem „(“-Token kann sowohl ein eingeklammerter Ausdruck als auch eine `CastExpression` folgen. Sollte der Token nach der Klammer ein „int“-, „char“- oder „boolean“-Schlüsselwort sein, wird von einer `CastExpression` ausgegangen. In diesem Fall wird lediglich ein „)“-Token erwartet und die `parsePrimaryExpression`-Methode rekursiv aufgerufen. Sonst wird die `parseExpression`-Methode aufgerufen, da innerhalb einer Klammer jegliche `Expression`s stehen können:

```
if (isTypeKeyword(peekToken().type)) {
    Token typeName = consumeToken();
    expectToken(TokenType.CloseParenthesis, "after type name at the start of cast expression");
    Expression casted = parsePrimaryExpression();
    return new CastExpression(typeName, casted);
} else {
    Expression parenthesized = parseExpression();
    expectToken(TokenType.CloseParenthesis, "at the end of parenthesized expression");
    return parenthesized;
}
```

- Sollte der nächste `Token` der Operator einer einstelligen Verknüpfung, also ein Operator mit nur einem Operanden, sein, wird, ähnlich wie beim Type-Cast, `parsePrimaryExpression` rekursiv aufgerufen, um den Operanden zu parsen. Anschließend wird eine `UnaryExpression` zurückgegeben:

```
if (isUnaryOperator(next.type)) {
    Token operator = consumeToken();
    Expression operand = parsePrimaryExpression();
    return new UnaryExpression(operator, operand);
}
```

- Wird keiner dieser Bedingungen erfüllt, wird ein Fehler an der Stelle des momentanen `Token`s ausgegeben, da keine andere Art von Token an dieser Stelle möglich ist.

`AssignmentExpressions` werden auch in der `parseBinaryExpression`-Methode geparst. Dabei wird das „=“-Zeichen als eine Art Operator mit der Präzedenz 0 gewertet:

```
if (isAssignmentOperator(operator.type) && parentPrecedence == 0) {
    consumeToken();
    Expression rValue = parseExpression();
    return new AssignmentExpression(left, operator, rValue);
}
```

Die bereits erwähnte übergeordnete Methode `parseExpression` besteht darin, die `parseBinaryExpression`-Methode mit der Präzedenz 0 als Parameter aufzurufen. So wird eine allgemeine Expression geparst, da die `parseBinaryExpression`-Methode lediglich das Ergebnis der `parsePrimaryExpression`-Methode zurückgibt, sollte kein binärer Operator darauf folgen.

5.3 Parsen der Statements

Für das Parsen der `Statements` ist im Kontrast zum Parsen der `Expressions` kein fundierter Algorithmus notwendig. So lässt sich jedes `Statement` anhand des ersten Tokens identifizieren:

- Ein `WhileStatement` fängt immer mit einem „while“-Token an. Darauf folgt eine in Klammern formulierte `Expression`, die als Bedingung dient. Anschließend wird erneut ein Statement rekursiv geparst:

```
consumeToken();
expectToken(TokenType.OpenParenthesis);
Expression condition = parseExpression();
expectToken(TokenType.CloseParenthesis);
Statement body = parseStatement();
return new WhileStatement(condition, body);
```

- Das Parsen eines `IfStatement` wird ähnlich implementiert. Nach dem Statement, das als „then“-Zweig dient, muss hier aber auch noch geprüft werden, ob ein „else“-Schlüsselwort folgt. In diesem Fall muss noch ein weiteres `Statement` rekursiv geparst werden, welches den „else“-Zweig bildet:

```
consumeToken();
expectToken(TokenType.OpenParenthesis);
Expression condition = parseExpression();
expectToken(TokenType.CloseParenthesis);
Statement body = parseStatement();
next = peekToken();
Statement elseBody = null;
if (next.type == TokenType.ElseKeyword) {
    consumeToken();
    elseBody = parseStatement();
}
return new IfStatement(condition, body, elseBody);
```

- `BreakStatements` und `ContinueStatements` bestehen nur in einem Schlüsselwort und einem Semikolon, sodass das Parsen entsprechend trivial ist:

```
case BreakKeyword: {
    Location location = consumeToken().location;
    expectToken(TokenType.SemiColon);
    return new BreakStatement(location);
}
case ContinueKeyword: {
    Location location = consumeToken().location;
    expectToken(TokenType.SemiColon);
    return new ContinueStatement(location);
}
```

- Ein `DefinitionStatement` fängt immer mit einem Typ-Schlüsselwort an. Würde rJava Klassendefinitionen und somit willkürliche Typennamen unterstützen, wäre hier ein komplexerer Ansatz notwendig, bei dem im `Parser`-Schritt bereits eine Symboltabelle mit den benutzerdefinierten Typennamen gefüllt wird.

  Nach dem Typschlüsselwort folgt dann der Name der zu definierenden Variable, ein „=“-Zeichen und ein rekursiv geparstes Statement, welches den Wert der Variable definiert:

```
if (isTypeKeyword(next.type)) {
    Token typeToken = consumeToken();
    Token name = expectToken(TokenType.Variable, "in definition statement");
    expectToken(TokenType.Equal);
    Expression definingExpression = parseExpression();
    expectToken(TokenType.SemiColon);
    return new DefinitionStatement(typeToken, name, definingExpression);
}
```

- `CompoundStatement`s fangen immer mit einem „{“-Zeichen an. Darauf folgen beliebig viele `Statements`, die in einer `while`-Schleife geparst werden. Am Ende des Statements wird ein „}“-Zeichen erwartet:

```
Token open = consumeToken();
ArrayList<Statement> statements = new ArrayList<>();
while (peekToken().type != TokenType.CloseCurly) {
    int startPos = position;
    statements.add(parseStatement());
    if (startPos == position) {
        break;
    }
}
expectToken(TokenType.CloseCurly, "at the end of compound statement");
return new CompoundStatement(open.location, statements);
```

- Falls keine der bisher genannten Bedingungen zutrifft, wird ein `ExpressionStatement` geparst, indem die `parseExpression`-Methode aufgerufen wird. Nach der Expression wird ein Semikolon erwartet:

```
Expression expression = parseExpression();
expectToken(TokenType.SemiColon, "at the end of statement");
return new ExpressionStatement(expression);
```

Da rJava weder Klassen- noch Methodendefinitionen unterstützt, besteht eine Quelltextdatei lediglich in einem Statement. Ein Programm mit mehreren Befehlen kann dann mithilfe eines einzelnen `CompoundStatements` beschrieben werden.

6 Überprüfungsschritt

Nach dem Parser folgt in unserem rJava-Compiler ein Überprüfungsschritt, der hauptsächlich in einer Analyse des Programms auf semantische Fehler besteht. Dabei wird meist überprüft, ob an jeder Stelle des Codes die passenden Datentypen eingesetzt werden, weshalb die zugehörige Klasse auch „TypeChecker“ heißt.

Ein Beispiel für eine Fehlerhafte Programmstelle könnte so aussehen:

```
boolean a = true;
a = a * 5;
```

Hier wird im zweiten Statement eine `boolean`-Variable mit einem `int`-Literal multipliziert. Syntaktisch ist dies zwar kein Fehler, jedoch ist semantisch die Multiplikation eines Wahrheitswertes mit einem Zahlenwert nicht sinnvoll.

Um solche Fehler zu erkennen, wird der in der `Parser`-Klasse zusammengesetzte Syntaxbaum postordertraversiert. Diese Form der Traversierung ist sinnvoll, da so die Blätter des Syntaxbaumes immer zuerst behandelt werden. Die Blätter als erstes zu betrachten hat den Vorteil, dass hier der Datentyp der Expression meist eindeutig ist. Ein `int`-Literal ist beispielsweise immer durch einen `TokenType.IntLiteral`-Token repräsentiert. Anschließend kann bei den Knoten des Syntaxbaumes, die näher an der Wurzel liegen, anhand der Kindesknoten entschieden werden, welcher Typ vorliegt und ob die richtigen Typen verwendet wurden.

Für die Traversierung des Syntaxbaumes kann sich der Polymorphismus durch die abstrakten Klassen `Expression` und `Statement` zu Nutze gemacht werden. So ist in beiden Klassen die abstrakte Methode `resolveType` definiert, die je von allen Unterklassen implementiert werden muss. Beispielsweise ist es so möglich, in der `resolveType`-Methode der `BinaryExpression`-Klasse, unabhängig davon, welche Art von `Expression` auf der rechten bzw. linken Seite der `Expression` steht, die `resolveType`-Methode zu nutzen, um zu überprüfen, ob die richtigen Typen für den jeweiligen Operator vorliegen. Dabei wird jeweils immer ein Objekt der `TypeChecker`-Klasse als Parameter übergeben, welches Daten über das Programm speichert und diverse Hilfsmethoden implementiert.

Die Implementation der `resolveType`-Methode sieht für die jeweiligen `Expression`s wie folgt aus:

- Wie bereits beschrieben ist bei `LiteralExpression`s der Datentyp immer bereits durch den `TokenType`-Wert bestimmt, und somit die Implementation von `resolveType` trivial.
- Bei `BinaryExpression`s wird zunächst die `resolveType`-Methode der linken und rechten `Expression` aufgerufen. Anschließend wird die Methode `getBinaryOperatorType` des `TypeChecker`s aufgerufen. Diese besteht darin, über ein zuvor definiertes Array zu iterieren, in dem jeder mögliche Operator für eine `BinaryExpression` mit den jeweiligen Typen für die linke und rechte Seite sowie dem Rückgabetypen definiert sind. Wird ein Operator gefunden, der mit den angegebenen Werten für das Operatorzeichen, den rechten sowie den linken Datentypen übereinstimmt, wird der jeweilige Rückgabetyp zurückgegeben. Wird kein solcher Operator gefunden, gibt die Methode `null` zurück. In diesem Fall wird ein Fehler ausgegeben. Ansonsten wird der Rückgabetyp als Datentyp der `BinaryExpression` festgelegt.
- Bei `UnaryExpressions` ist die Vorgehensweise ähnlich wie bei `BinaryExpressions`. Hier wurde eine entsprechende Methode `getUnaryOperatorType` in der `TypeChecker`-Klasse implementiert.
- Für die Implementation der `VariableExpression` wurde in der `TypeChecker`-Klasse ein Datenfeld `scope` angelegt. Hier werden alle Variablen und Funktionen und ihre Typen festgehalten. Ob und mit welchem Datentyp eine Variable definiert wurde, lässt sich mit der Methode `getSymbol` der `TypeChecker`-Klasse erfahren. Sollte es sich um eine Variable handeln, wird dann der jeweilige Datentyp als `return`-Wert der `resolveType`-Methode zurückgegeben. Darüber hinaus wird das Feld `id` der `VariableExpression` mit einem Wert aus `scope` belegt. Diese `id` ist für den `CodeGenerator`-Schritt relevant.
- Bei der Implementation der `CallExpression` wird ebenfalls auf die `getSymbol`-Methode zurückgegriffen. Hier ist jedoch zu beachten, dass in Java, im Gegensatz zu Variablen, mehrere Methoden den gleichen Namen aufweisen können. Dieses Feature wird im Allgemeinen als Funktions- bzw. Methoden-Overloading bezeichnet. Dabei kann eine Funktion mehrfach definiert werden, solange jedes Mal andere Parameter übergeben werden. Dieses Feature wird

insofern für rJava übernommen, als dass die vordefinierten Funktionen „print“ und „println“ sowohl `int`- als auch `char`-Werte als Argumente akzeptieren.

Um dieses Feature zu implementieren, wird für jede Funktion eine Liste `overloads` gespeichert, die Auskunft über die verschiedenen Parameterkombinationen gibt. Für jede `CallExpression` muss dann über alle Elemente der `overloads`-Liste iteriert werden, um zu erfahren, ob eine valide Kombination von Argumenten verwendet wurde.

- Für `AssignmentExpressions` wird zunächst überprüft, ob die Expression auf der linken Seite des „=“-Zeichens eine `VariableExpression` ist. Ist dies nicht der Fall, wird ein Fehler ausgegeben, da in rJava nur Variablen ein Wert zugewiesen werden kann. Dies scheint zunächst offensichtlich, jedoch wäre zum Beispiel, falls Arrays zu unserer Sprache gehören würden, auch eine Expression wie „array[i][j]“ an dieser Stelle valide. Anschließend wird überprüft, ob beide Seiten der `Expression` den gleichen Datentypen haben.
- Für `CastExpressions` muss lediglich überprüft werden, ob ein Typecast zwischen den jeweiligen Typen vorgesehen ist. Für rJava ist dies nur bei Typecasts zwischen `char` und `int` der Fall.

Die Implementation der `resolveType`-Methode für die Unterklassen von `Statement` erfolgt wie folgt:

- Für ein `ExpressionStatement` wird die `resolveType`-Methode der zugrundeliegenden `Expression` ausgeführt.
- Bei einem `DefinitionStatement` wird zunächst überprüft, ob der über das Schlüsselwort zu Beginn des `Statement`s angegebene Typ mit dem Typ der definierenden `Expression` übereinstimmt. Anschließend wird anhand der `getSymbol`-Methode sichergestellt, dass eine Variable mit dem definierten Namen nicht bereits existiert. Zuletzt wird die `id` der jeweiligen Variable festgelegt und die Variable wird dem `scope` hinzugefügt.
- Zur Überprüfung von `WhileStatements` wird zuerst kontrolliert, ob der Datentyp der `Expression`, die als Bedingung des `WhileStatement`s dient, `boolean` ist. Darüber hinaus wird sichergestellt, dass es sich beim Schleifenkörper nicht um ein `DefinitionStatement` handelt, da solche `Statement`s an dieser Stelle zu semantischen Fehlern führen würden. Bevor

der Schleifenkörper über die `resolveType`-Methode überprüft wird, wird eine Referenz auf das `WhileStatement` auf den Stapel `whileStatementStack` des `TypeChecker`-Objekts gelegt.

- Der `whileStatementStack` ist in der `resolveType`-Methode der `Break`- und `ContinueStatement`s relevant. Hier kann nun das jeweilige Kontrollstrukturstatement als Referenz an die Liste `controlFlowStatements` der `while`-Schleife, die das oberste Element des `whileStatementStacks` bildet, angehangen werden. So wird jeder `while`-Schleife eine Liste von Kontrollstruktur-Statements zugeordnet, die bei der Generierung des JVM-Assemblycodes genutzt werden kann.
- Die Überprüfung von `IfStatement`s verläuft ähnlich der eines `WhileStatement`s: Die Bedingung soll den Datentypen `boolean` haben und das Statement `body` soll kein `DefinitionStatement` sein. Sollte ein `else`-Teil des `IfStatement`s vorliegen, muss dieser entsprechend auch überprüft werden.
- Für `CompoundStatement`s wird bei jedem Unterstatement die `resolveType`-Methode aufgerufen. Zuvor wird im `TypeChecker` jedoch noch eine neuer Scope „angelegt“. Dazu ist zu beachten, dass das Datenfeld `scope` des `TypeChecker`s sich aus einer Liste von HashTabellen zusammensetzt. Wird ein neuer Scope angelegt, wird eine neue Tabelle an jene Liste angehängt. Wird nun nach einem Symbol gesucht, wird über die Liste iteriert und in jeder Tabelle nach der Variablen gesucht. Nachdem die `resolveType`-Methode jedes Unterstatements aufgerufen wurde, wird das letzte Element der `scope`-Liste wieder entfernt. So kommt es zu der Scopesemantik, wie sie in Java vorzufinden ist, bei der jede Variable nur innerhalb des `CompoundStatements`, in dem es definiert wurde, genutzt werden kann.

7 Überführung in JVM-Bytecode

7.1 JVM und der JVM-Bytecode

Die Java Virtual Machine (JVM) ist eine Stack basierte virtuelle Maschine, die zur Ausführung des JVM-Bytecodes dient. Sie wurde für die Programmiersprache Java entwickelt, um Java-Programme in Form von .class-Dateien plattformunabhängig ausführen zu können. Jedoch ist die Anwendung der JVM nicht nur auf die

Programmiersprache Java begrenzt. Mittlerweile gibt es zahlreiche andere Programmiersprachen wie Clojure oder Kotlin, die JVM-Bytecode generieren und auf die Nutzung der JVM ausgelegt sind. Die einzige Bedingung, um ein Programm mithilfe der JVM ausführen zu können, ist, das Programm in jenen JVM-Bytecode in Form einer .class-Datei zu überführen[2].

Ein zentrales Element der JVM ist der Operanden-Stack, welcher essenziell für Stack basierte virtuelle Maschinen ist. Für jede Operation wie Addition, Subtraktion oder auch das Aufrufen von Methoden werden zuvor alle Operanden oder Parameter auf dem Operanden-Stack abgelegt. Das Ergebnis der Operation wird dann auf dem Stapel abgelegt, sodass es von weiteren Operationen genutzt werden kann. Der Ausdruck „1+2*3“ würde demnach so im JVM-Bytecode repräsentiert werden:

```
1 iconst_1
2 iconst_2
3 iconst_3
4 imul
5 iadd
```

Die Instruktion „iconst_x“ legt dabei jeweils den Wert x auf den Stapel und „imul“ bzw. „iadd“ dienen jeweils zur Multiplikation bzw. Addition der beiden obersten Elemente des Stacks.

Hier ist zu berücksichtigen, dass auf den Stack Werte jeglicher Datentypen abgelegt werden können, wobei Objekte lediglich als Referenzen, die auf das jeweilige Objekt zeigen, repräsentiert werden. Ein Element des Stacks wird dabei als „Wort“ bezeichnet, dessen Speichergröße plattformspezifisch ist (meist 32 oder 64 Bit) [3].

Kontrollstrukturen werden im JVM-Bytecode über Sprung- und bedingte Sprunginstruktion implementiert. Während eine Sprunginstruktion immer an eine bestimmte Stelle im Programm springt, geschieht dies bei bedingten Sprunginstruktionen nur unter einer angegebenen Bedingung[4].

Neben dem Operanden-Stack werden Daten auch in Form von lokalen Variablen gespeichert. Diese sind weitestgehend äquivalent zu den lokalen Variablen im Java- bzw.

---

[2] Vgl. Spezifikation der JVM, 1.2 (Quelle 2)
[3] Vgl. Spezifikation der JVM, 2.6.2 (Quelle 2)
[4] Vgl. Spezifikation der JVM, 2.11.7 (Quelle 2)

rJava-Code und können jederzeit durch Instruktionen auf dem Stack abgelegt oder mit Werten vom Stack belegt werden[5].
Die Anzahl der lokalen Variablen sowie die maximale Größe des Operanden-Stacks, die in einer Methode gebraucht werden, müssen beim Kompilierungsprozess ermittelt und in der .class-Datei angegeben werden.

7.2 Implementation des Code-Generators

Ähnlich wie beim Überprüfungsschritt wird die Code-Generierung mithilfe von abstrakten Methoden der `Expression`- und `Statement`-Klassen sowie einer Hilfsklasse `CodeGenerator` implementiert. Der Syntax-Baum wird dabei erneut postordertraversiert, wobei die Instruktionen in Textform an einen String konkateniert werden. Hier ist erneut zu beachten, dass unser rJava-Compiler nicht den JVM-Bytecode selbst, sondern eine Textrepräsentation des Bytecodes, welche von einem externen Tool „Jasmin“ zu einer class-Datei umgewandelt werden kann, generiert.
Wie in den beiden vorherigen Kapiteln, soll hier erneut für jede Expression die Implementation der `emit`-Methode, welche die zugehörigen Bytecode-Instruktionen erzeugt, näher erläutert werden:

- Für `BinaryExpressions` wird (s.a. 7.1) zunächst die linke und rechte Seite der Expression generiert. Durch den Aufbau des Syntaxbaumes (s.a. 5.1) wird dabei sichergestellt, dass alle Instruktionen in der richtigen Reihenfolge an den Ausgabe-String konkateniert werden. Anschließend wird bei arithmetischen und binären Operationen die zugehörige Instruktion angehängt, beispielsweise „imul“ für „*“-Operationen oder „iand“ für „&“-Operationen.

  Dieses Vorgehen lässt sich jedoch nicht auf logische und Vergleichsoperationen übertragen. Ausschlaggebend dafür ist, dass im JVM-Bytecode Vergleichsoperationen nur über bedingte Sprunginstruktionen möglich sind. In unserer Implementation wird diese Eigenschaft des Bytecodes insofern umgangen, dass bei Vergleichsoperationen immer - je nach Ergebnis - entweder zu einer „iconst_0“ oder „iconst_1“ Operation gesprungen wird. So landet nach jeder Vergleichsoperation immer ein Wahrheitswert (0 oder 1) auf dem Operanden-Stack.

[5] Vgl. Spezifikation der JVM, 2.11.2 (Quelle 2)

Eine weitere Ausnahme bilden das logische „&&“ (AND-Operator) und das logische „||“ (OR-Operator). Diese werden in Java über eine Kurzschlussauswertung implementiert. Das bedeutet, dass je nach Ergebnis der linken Seite des Ausdrucks die rechte Seite des Ausdrucks nicht berücksichtigt werden muss und somit nicht ausgeführt wird. Beispielsweise ist ein „&&“-Ausdruck unabhängig von der rechten Seite immer falsch, solange die linke Seite falsch ist. Hier wird mithilfe von bedingten Sprunginstruktionen die rechte Seite je nach Ergebnis der linken Seite übersprungen.

- `UnaryExpressions` werden ähnlich wie `BinaryExpressions` generiert, mit dem Unterschied, dass nur eine Expression als Operand generiert werden muss.
- Für `VariableExpressions` muss lediglich eine „iload x“-Instruktion angehängt werden. Der Operand x besteht dabei in der `id` der Variable, welche zuvor im Überprüfungsschritt ermittelt wurde.
- Für `CallExpression`s ist der Generierungsprozess insofern einfach, als dass keine benutzerdefinierten Funktionen berücksichtigt werden müssen. Des Weiteren wird die Standardbibliothek von Java auch nicht unterstützt. Die einzigen Funktionen, die in rJava genutzt werden können, sind die hartkodierten Funktionen „print“ und „println“. Diese sind äquivalent zu den statischen Methoden „System.out.println“ und „System.out.print“ in Java. Zur Implementation dieser Funktionen werden jeweils die Instruktionen „getstatic“ und „invokevirtual“ verwendet. „getstatic“ dient dazu, eine Referenz auf das statische Objekt System.out auf den Stack zu legen und „invokevirtual“ führt die jeweilige Methode aus. Bevor die Methode ausgeführt wird, muss das als `Expression` angegebene Argument der Funktion noch generiert werden.
- Für `AssignmentExpression`s wird zunächst die rechte Seite der Variablenbelegung generiert. Anschließend wird die Instruktion „istore x“ angewendet, welche das oberste Element des Stacks als Wert der lokalen Variable mit `id` x speichert.
- CastExpressions müssen auf Bytecode-Ebene nur für Type-Casts von `int`-Werten zu `char`-Werten implementiert werden, da die Repräsentation der Datentypen in anderen Fällen äquivalent ist. Die zugehörige Instruktion lautet „i2c“.

Eine äquivalente Methode `emit` wird auch für alle `Statement`s implementiert:

- Ein `ExpressionStatement` wird im Bytecode-Assembly genauso repräsentiert, wie die zugehörige `Expression`. Am Ende der Instruktionen für die Expression muss jedoch zusätzlich noch eine „pop“-Instruktion angehängt werden, falls die `Expression` einen Wert auf dem Operanden-Stack hinterlässt. So wird sichergestellt, dass die Größe des Stacks minimal bleibt.
- Die Generierung von `DefinitionStatements` erfolgt so wie die Generierung von `AssignmentStatements` über die „istore“-Instruktion.
- Für `IfStatement`s werden bedingte Sprunginstruktionen verwendet. Aufgrund der Implementation von logischen Operationen, die gewährleistet, dass nach einer logischen Operation immer ein Wahrheitswert auf dem Stack liegt, kann hier in jedem Fall die „ifeq“-Instruktion verwendet werden. Diese springt nur dann an eine angegebene Stelle im Programmcode, wenn der Wert 0 auf dem Stack liegt. Demnach kann nach Generierung der Bedingung des `IfStatement`s mithilfe jener „ifeq“-Instruktion an das Ende des Statement-Körpers oder zum `else`-Zweig gesprungen werden. Zu welcher Instruktion eine goto-Instruktion im Programmcode springt, wird im Bytecode-Assembly über Labels spezifiziert. Die Methode `generateLabel` des `CodeGenerators` generiert dabei bei jedem Aufruf ein eindeutiges Label, welches beim Anhängen einer goto-Instruktion beigefügt wird. Damit festgelegt wird, zu welcher Stelle im Bytecode ein Label gehört, muss an jene Stelle im Assembly das Label mit einem Doppelpunkt angehängt werden. Ein `IfStatement` mit `else`-Zweig würde im Bytecode-Assembly demnach so realisiert werden:

```
1 <Bedingung>
2 ifeq ELSE
3 <if-Körper>
4 goto ENDE
5 ELSE:
6 <else-Körper>
7 ENDE:
```

- `WhileStatement`s sind im Grunde ähnlich aufgebaut wie `IfStatement`s. Nach einer Bedingung wird über eine „ifeq“-Instruktion entweder zum Schleifenkörper oder zur ersten Instruktion nach der Schleife gesprungen. Nach jedem Durchlauf des Schleifenkörpers wird dann zurück zur Bedingung gesprungen. Zusätzlich müssen jedoch noch `Break`- und `ContinueStatement`s

berücksichtigt werden. Jedem Statement in der `controlFlowStatement`-Liste des `WhileStatement`s wird dazu ein Label übergeben, zu welchem gesprungen werden soll, falls das jeweilige `Break`- oder `ContinueStatement` erreicht wird. Ein allgemeines `WhileStatement` sähe im Bytecode-Assembly demnach so aus:

```
1 <Bedingung>
2 CONTINUE:
3 ifeq BREAK
4 <Schleifenkörper>
5 goto CONTINUE:
6 BREAK:
```

- Da `CompoundStatement`s lediglich eine Abfolge mehrerer Statements beschreiben, muss auf Bytecode-Assembly-Ebene nur noch jedes Unterstatement in chronologischer Reihenfolge generiert werden.

Da jedes rJava-Programm aus einem einzelnen Statement besteht, kann nun jedes Programm in Form einer Instruktionsabfolge mithilfe der `emit`-Methode als `String` generiert werden. Da jedoch die JVM die objektorientierte Struktur von Java übernimmt, muss ein Programm immer als `main`-Methode einer Klasse definiert werden. Daher leitet sich auch die Etymologie der „.class“-Datei, welche ein JVM-Programm beschreibt, her, weil jede .class-Datei nur eine bestimmte Klasse beschreibt.

Um nun eine vollständige .class-Datei generieren zu können, muss also die Befehlsabfolge, die in `emit` erzeugt wurde, in die statische main-Methode einer Klasse eingebettet werden. Dies wird über ein vordefiniertes Rahmenprogramm in der `CodeGenerator`-Klasse implementiert, in welches der Assemblycode eingefügt wird.

Die so erzeugte Klassendefinition wird dann in eine „.j“-Datei geschrieben, die mit dem Jasmin-Assembler in eine ausführbare .class-Datei überführt werden kann.

Die einzelnen Schritte des Compilers `Lexer`, `Parser`, `TypeChecker` und `CodeGenerator` können nun in der `main`-Methode des Compilerprogramms hintereinander ausgeführt werden. Die Eingabedatei und der Name der Ausgabeklasse werden mit Befehlszeilenargumenten spezifiziert.

8 Fazit

8.1 Fachbezogene Reflexion und Ausblick

Durch die anfangs formulierten Einschränkungen ist der Anwendungsbereich des rJava-Compilers stark begrenzt. Da die Programmiersprache rJava weder Methoden- noch

Klassendefinitionen bereitstellt, wird die Programmierung von komplexen Programmen schnell umständlich. Darüber hinaus lassen sich durch den eingeschränkten Speicherzugriff einige Programme nicht realisieren. Des Weiteren ist zu betrachten, dass keine Funktionen in rJava definiert sind, die Interaktion mit dem User erlauben würden, zum Beispiel durch Texteingaben in der Konsole oder Graphical User Interfaces (GUI).
Ein weiterer Aspekt, der in dieser Lernleistung nicht beleuchtet wurde, ist die Optimierung des Compilers. Heutzutage besteht ein Großteil der Arbeit, der in die Compilerentwicklung gesteckt wird, darin, die erzeugten Programme möglichst effektiv zu gestalten, um die Laufzeit zu minimieren.
Die gewählte Implementation des Compilers, welche mit abstrakten Klassen arbeitet, lässt sich jedoch trivial durch weitere syntaktische Strukturen, Datentypen und Übersetzungsschritte erweitern, sodass der Compiler in Zukunft durch genannte Features erweitert werden könnte. Eine dynamisch allokierte Array-Struktur, wie sie in Java genutzt wird, könnte zum Beispiel hinzugefügt werden, um rJava Turing-vollständig zu machen.
Trotz der genannten Beschränkungen lassen sich, wie sich in den Beispielprogrammen zeigt, immer noch einige „interessante“ Programme in rJava implementieren. Es mag sich zwar meist um mathematische Spielereien handeln, jedoch ließe sich durch einfache Erweiterungen des Compilers schon ein breites Spektrum nützlicher Anwendungen erstellen.

8.2 Persönliches Fazit

Persönlich war diese besondere Lernleistung eine sehr lehrreiche Erfahrung für mich. Sich mit Compilerbau zu befassen, zeigt einem, wie Programmiersprachen in ihren Grundsätzen aufgebaut sind; sowohl syntaktisch als auch semantisch. Die Implementation kann dabei zu einem gewissen Grad erfüllend sein, da einige Teile des Compilers, wie der `TypeChecker`, für jemanden der ein gewisses Maß an Programmiererfahrung hat, sehr intuitiv funktionieren. Ein weiterer interessanter Aspekt des Compilerbaus war für mich, wie Baumstrukturen und ihre Traversierung, die oft abstrakt erscheinen, eine praktische Anwendung finden.
Etwas überwältigend war hingegen das Ausmaß eines Compilers. Selbst unter den getätigten Einschränkungen ist der Quelltext ca. 1500 Zeilen lang und die Dokumentation umfasst über 23 Seiten. Besonders bei der Betrachtung der Zeitplanung ist das zum Problem geworden, da das Schreiben der Dokumentation unerwartet viel Zeit in

Anspruch nahm. Durch den Aufbau des Compilers, der zu einem Großteil in Traversierungen besteht, war der Schreibprozess zeitweise repetitiv und langwierig.
Alles in allem würde ich die Arbeit an dieser Lernleistung dennoch als positive Erfahrung einordnen, da ich sowohl fachbezogen als auch methodisch im Zusammenhang mit dem Schriftteil viel lernen konnte.

9 Literaturverzeichnis

1. Turing Completeness CS390, Spring 2023, https://www.cs.odu.edu/~zeil/cs390/latest/Public/turing-complete/index.html, Zugriff am 27.03.23.
2. The Java Virtual Machine Specification, Java SE 7 Edition, https://docs.oracle.com/javase/specs/jvms/se7/html/index.html, Zugriff am 27.03.23.

10 Anhang

10.1 Beispielprogramm „Fibonacci“

Quelltext:

```
{
    int a = 0;
    int b = 1;
    int i = 0;
    int iterations = 10;
    while (i < iterations) {
        println(a);
        int t = a + b;
        a = b;
        b = t;
        i = i + 1;
    }
}
```

Ausgabe:

```
C:\Users\jojob\source\repos\rJava>java Fibonacci
0
1
1
2
3
5
8
13
21
34
```

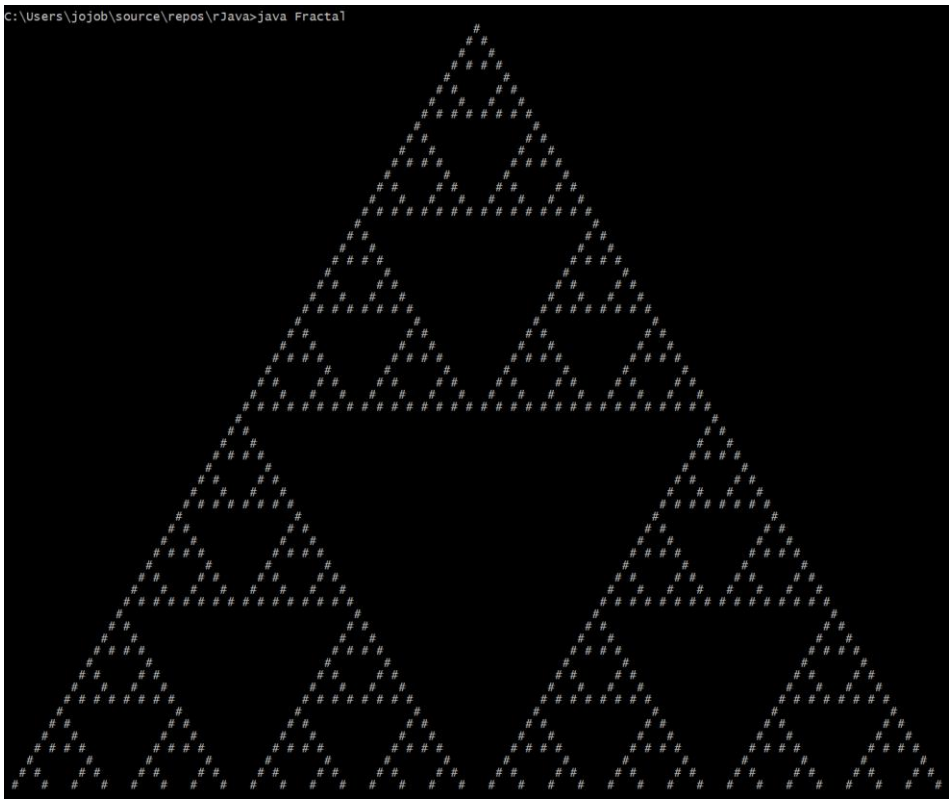
10.2 Beispielprogramm „Sierpinski Dreieck“

Quelltext:

```
{
    int i = 0;
    int w = 63;
    while (i < w) {
        int j = 0;
        int spaces = 0;
        int y = w - i;
        while (spaces < y) {
            print(' ');
            spaces = spaces + 1;
        }
        while (j < w) {
            int x = j;
            if ((x & y) == 0) print('#');
            else print(' ');
            print(' ');
            j = j + 1;
        }
        println(' ');
        i = i + 1;
    }
}

```

Ausgabe:



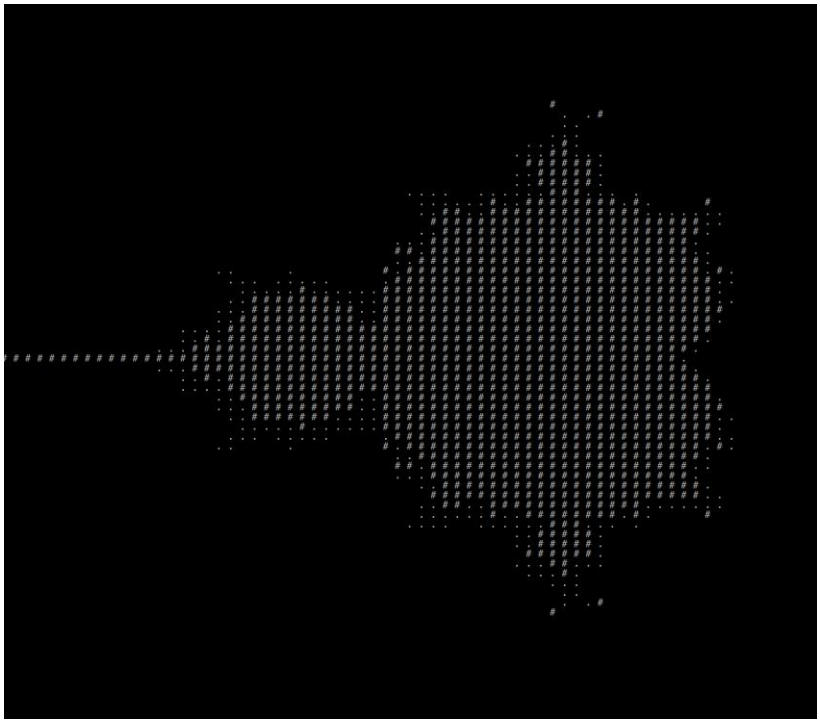
10.3 Beispielprogramm „Mandelbrot-Menge“

Quelltext:

```
{
    int dim = 100;
    int y = 0;
    while (y < dim) {
        int x = 0;
        while (x < dim) {
            int offset = 100;
            int cr = (x * 4 * offset) / dim - 2 * offset;
            cr = cr * -1;
            int ci = (y * 4 * offset) / dim - 2 * offset;
            int zr = 0;
            int zi = 0;
            int iteration = 0;
            int maxIteration = 1000;
            while (iteration < maxIteration) {
                int zr2 = (zr * zr) / offset;
                int zi2 = (zi * zi) / offset;
                if (zi2 + zr2 > 4 * offset) break;
                zi = (2 * zr * zi) / offset + ci;
                zr = zr2 - zi2 + cr;
                iteration = iteration + 1;
            }

            if (iteration == maxIteration) {
                print('#');
            } else if (iteration > 10) {
                print('.');
            } else {
                print(' ');
            }
            //print((char)((int)' ' + iteration));
            print(' ');
            x = x + 1;
        }
        print((char)10);
        y = y + 1;
    }
}
```

Ausgabe:



Versicherung der selbständigen Erarbeitung

Ich versichere, dass ich die vorliegende Arbeit einschließlich evtl. beigefügter Zeichnungen, Kartenskizzen, Darstellungen u. ä. m. selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

_____, den _____

(Ort) (Datum)

_____

(Unterschrift)