Ausarbeitung

Entwicklung eines Compilers für die Sprache FARE zur Zielsprache Java

Development of a compiler for the language FARE to the target language Java

An der Fachhochschule Dortmund
im Fachbereich Informatik
Studiengang Medizinische Informatik Master
im Modul Formale Sprachen und Compilerbau
erstellte Ausarbeitung eines FARE-Compilers

von
Johannes Lang
Matr.-Nr. 7217450

Henning Müller
Matr.-Nr. 7105852

Wladislaw Jerokin
Matr.-Nr. 7205290

Betreuung durch:
Prof. Dr. Robert Rettinger

Dortmund, 4. Oktober 2023

Inhaltsverzeichnis

1 Einleitung

Grundlegend definiert sich ein Compiler als Programm, welches einen gegebenen Quellcode zu Maschinencode, Bytecode oder einer anderen Programmiersprache übersetzen kann (vgl. Robert Sheldon, 2023). Die Entwicklung eines solchen Compilers ist eine komplexe Aufgabe, die aus mehreren Teilgebieten besteht. In dieser Ausarbeitung werden folgende Teilgebiete behandelt:

Num	Name
1	Lexikalische Analyse
2	Syntaxanalyse
3	Semantische Analyse
4	Fehlerbehandlung
5	Codeerzeugung

Tabelle 1: Behandelte fünf Teilbereiche des Compilerbaus

Die in dieser Ausarbeitung behandelte Aufgabe besteht darin, die Bereiche in Tabelle 1 für eine Scriptsprache für den Umgang mit Dateien und Pfaden zu entwickeln. Folgend werden die herausgearbeiteten Token, Grammatik und Semantik in jeweils eigenen Kapiteln beschrieben.

1.1 Herangehensweise

Der Compiler wird in Java geschrieben und benutzt die Bibliotheken JavaCC und die JavaCC-interne JJTree. Somit wird die Lexikalische Analyse und die Syntaxanalyse durch die definierte Grammatik durchgeführt. Um dies kurz auszuführen wird mit JavaCC geprüft, ob ein gegebener Quellcode zur definierten Sprache und Grammatik zugehört (Lexikalische Analyse). Durch erweiterte Annotation der Grammatik wird folgend der AST (Abstract Syntax Tree) durch JJTree generiert (Syntaxanalyse). Falls keine lexikalischen Fehler vorliegen, wird der AST zur semantischen Analyse und Fehlerbehandlung übergeben, wo auf semantische Korrektheit geprüft wird. Parallel wird der AST zu Java Source Code übersetzt und abschließend nach Ablauf der semantischen Analyse compiliert.

2 Token

```
1  // de/fh/javacc/Grammar1.jjt, Zeile 20 - 78
2  TOKEN : {
3    // TYPES
4    <TYPE_SPECIFIERS: "Map" | "int" | "char" | "String" | "boolean" | "Files"
         | "Path" | "Set" > |
5    <TYPE_SPECIFIERS_VOID: "void"> |
6
7    // SEPARATORS
8    <BRACKET_SQUARE_LEFT: "[">          | // PRIO 15
9    <BRACKET_SQUARE_RIGHT: "]">         | // PRIO 15
10   <BRACKET_ROUND_LEFT: "(">           | // PRIO 15
11   <BRACKET_ROUND_RIGHT: ")">          | // PRIO 15
12   <BRACKET_CURLY_LEFT: "{">           |
13   <BRACKET_CURLY_RIGHT: "}">          |
14   <DOT: ".">                          | // PRIO 15
15   <PATH_SLASH_SEPARATOR: "\\">        | // AND OP_MUL
16   <SEMICOLON: ";">                    |
17   <COLON: ":">                        |
18   <COMMA: ",">                        |
19
20   // OPERATION
21   <OP_ASSIGNMENT: "="| "+=" |"-=" |"*="| "/=" | "%=">     | // PRIO 1
22   <OP_LOGICAL_OR: "||">               | // PRIO 3
23   <OP_LOGICAL_AND: "&&">              | // PRIO 4
24   <OP_BITWISE_EXCLUSIVE_OR: "^">      | // PRIO 6
25   <OP_RELATIONAL_EQUALS: "==" | "!="> | // PRIO 8
26   <OP_LESS_THAN: "<">                 | // PRIO 9
27   <OP_GREATER_THAN: ">">              | // PRIO 9
28   <OP_RELATIONAL_COMPARE: "<=" | ">=">    | // PRIO 9
29   <OP_SUM: "+" | "-">                 | // PRIO 11
30   <OP_DIV: "/">                       | // PRIO 12
31   <OP_MUL: "*" | "%">           | // PRIO 12
32   <OP_INCREMENT: "++" | "--">         | // PRIO 13 (pre) AND 14 (post)
33   <OP_PRE: "!">                       | // PRIO 13
34   // PRIO 15 Round Brackets, Array Subscript, Member Selection
35
36   // STATEMENTS
37   <ST_IF: "if">           |
38   <ST_ELSE: "else">       |
39   <ST_WHILE: "while">     |
40   <ST_FOR: "for">         |
41   <ST_RETURN: "return">   |
42
43   // LITERAL
44   <LITERAL_BOOLEAN: "true" | "false">             |
45   <LITERAL_IDENTIFIER: ["A"-"Z", "a"-"z", "_"](["A"-"Z", "a"-"z", "_", "0"
         - "9"])*> |
46   <LITERAL_STRING: "\""(~["\"","\n"])*"\"">       |
47   <LITERAL_CHAR: "'" (~["'","\n"]) "'">           |
```

```
48    <LITERAL_INTEGER: (["0"-"9"])+>                    |
49    <LITERAL_PATH: (<DOT> | ".." | (["A"-"Z"] <COLON>))? ((<
          PATH_SLASH_SEPARATOR> | <OP_DIV>) | (((<PATH_SLASH_SEPARATOR> | <
          OP_DIV>) (["A"-"Z", ".", "a"-"z", "_", "0" - "9"])+ ((["A"-"Z", ".",
          "a"-"z", " ", "_", "0" - "9"])* (["A"-"Z", ".", "a"-"z", "_", "0" - "
          9"])+))+) (<PATH_SLASH_SEPARATOR> | <OP_DIV>)?) >
50  }
51
52  SKIP : {
53    " "                       |
54    "\t"                      |
55    "\n"                      |
56    "\r"                      |
57    <COMMENT_SINGLE_LINE: "//" (~["\n"])*>      |
58    <COMMENT_MULTI_LINE: "/*" (~["*"] | "*" ~["/"] )* "*/">
59  }
```

Listing 1: Definierte JavaCC Token

Alle definierten Token sind in Listing 1 aufgelistet. Wie an den Kommentaren in Zeilen 3, 7, 20, 36, und 43 erkannt werden kann, liegen 5 verschiedene Kategorien der Token vor. Die unterstützten Datentypen werden zuerst genannt (vgl. Listing 1, Z. 4f). Folgend werden vielseitige Zeichen definiert, die zur Trennung von Befehlen oder auch für Operationen wie Memberselektion gebraucht werden (vgl. Listing 1, Z. 8 - 18). Die dritte Kategorie beschreibt unäre sowie binäre Operatoren (vgl. Listing 1, Z. 21 - 33). Die Kategorie der Statements beschreibt Schlüsselwörter für Verzweigungen, Schleifen und die Rückgabe (vgl. Listing 1, Z. 37 - 41).

Die letzte Kategorie der Literalen beschreibt die atomaren Werte, die vom Benutzer eingegeben werden. Unter diese fallen die Werte eines Booleans, Variablenidentifiers, Strings, Chars, Zahlen (als Integer) sowie des Paths. Bis auf den Path können die Ausprägungen mit Java Werten verglichen werden. Der Path ist eine Sonderform der Sprache FARE und beschreibt die Ausprägung eines Pfads. Dieser muss mit *., .., /, \, <LAUFWERK>:/, <LAUFWERK>:* beginnen. Darauf kann der Name des Verzeichnisses oder einer Datei folgen. Nach einem Separator (*/, *) kann der Name beliebig viel wiederholt werden. Ein Name kann aus Buchstaben, Zahlen, Punkten und Leerzeichen bestehen, wobei am Anfang und am Ende eines Namens Leerzeichen verboten sind. Sonderzeichen sind nicht gestattet. (vgl. Listing 1, Z. 44 - 49)

Neben den Token die eingelesen werden, gibt es Regeln, anhand deren Zeichen ignoriert werden. Zu diesen Zählen Leerzeichen, Umbrüche, Tabulatoren sowie ein- und mehrzeilige Kommentare. Einzeilige Kommentare werden mit *//* eingeleitet und gehen bis zum Zeilenumbruch. Mehrzeilige Kommentare werden mit */** eingeleitet und enden mit **/*. (vgl. Listing 1, Z. 53 - 58)

3 Grammatik

Wie in Kapitel 1.1 genannt, wird JavaCC und JJTree genutzt. Um die erweiterte Grammatik zu demonstrieren, wird im Folgenden ein Beispiel angebracht.

```
1  // de/fh/javacc/Grammar1.jjt, Zeile 471 - 485
2  void atom() :
3  {
4    Token t;
5  }
6  {
7    (t = <LITERAL_INTEGER> {jjtThis.value = t.image;})     #LITERAL_INTEGER |
8    (t = <LITERAL_BOOLEAN> {jjtThis.value = t.image;})     #LITERAL_BOOLEAN |
9    (t = <LITERAL_CHAR> {jjtThis.value = t.image;})        #LITERAL_CHAR    |
10   (t = <LITERAL_STRING> {jjtThis.value = t.image;})      #LITERAL_STRING  |
11   (t = <LITERAL_PATH> {jjtThis.value = t.image;})        #LITERAL_PATH    |
12   identifier()                                                            |
13   array_container()                                                       |
14   native_array_container()                                                |
15   <BRACKET_ROUND_LEFT> op_prio_3() <BRACKET_ROUND_RIGHT>
16 }
```

Listing 2: Beispiel der erweiterten Grammatik für Nutzung von JavaCC und JJTree anhand der Regel atom()

Analog zum reinen JavaCC können Regeln für die Grammatik definiert werden (vgl. Listing 2, Z. 7 - 15). Die Erweiterung liegt in der Annotation für die Erstellung des AST, welche die Werte *void, LITERAL_INTEGER, LITERAL_BOOLEAN, LITERAL_CHAR, LITERAL_STRING, LITERAL_PATH* annehmen kann. Im Sachkontext wird im Normalfall keine Node im AST angelegt (vgl. Listing 2, Z. 2). Sollte jedoch eine Regel in den Zeilen 7 - 11 angewandt werden, so wird ein entsprechend benannter Knoten zum AST mit dem Inhalt des Tokens hinzugefügt (vgl. Listing 2, Z. 7 - 8). Nach diesem Prinzip wird für alle Regeln an sinnvollen Stellen ein Knoten zum AST hinzugefügt. Diese Kombination von JavaCC und JJTree ermöglicht die simultane Lexikalische Analyse und Syntaxanalyse, was in einem AST resultiert. Nach der Anführung eines Beispiels wird folgend grob die Grammatik beschrieben.

```
1  // de/fh/javacc/Grammar1.jjt, Zeile 62 - 70
2  SimpleNode program() throws ParseException #PROGRAM :
3  {
4    boolean first = true;
5    SimpleNode result = null;
6  }{
7    (class_contents() { if (first) result = jjtThis; else result.jjtAddChild(
         jjtThis, result.jjtGetNumChildren()); })+
8
9    { return result; }
10 }
```

Listing 3: Wurzel des AST sowie erste Regel der Grammatik

Da die Ausführung der ersten Regel einen AST als Rückgabewert liefert, stellt die Regel in Listing 3 eine Sonderform dar. Wie erkannt werden kann, wird in jedem Fall ein neuer Knoten mit dem Namen *PROGRAM* erzeugt. Unter diesem Knoten können nun beliebig viele *class_ contents()* und vorliegen.
Nach diesem Muster werden die restlichen Regeln ebenfalls umgesetzt. Insgesamt können 66 verschiedene AST Knotentypen vorliegen, die im Folgenden alphabetisch aufgelistet werden.

AST Typ
ASTARRAY_CONTAINER
ASTARRAY_CONTAINER_NATIVE
ASTARRAY_ELEMENT
ASTBLOCK
ASTLITERAL_BOOLEAN
ASTLITERAL_CHAR
ASTLITERAL_IDENTIFIER
ASTLITERAL_INTEGER
ASTLITERAL_PATH
ASTLITERAL_STRING
ASTMAP_ELEMENT
ASTMAP_ELEMENT_KEY
ASTMAP_ELEMENT_VALUE
ASTMETHOD_DECLARATION
ASTMETHOD_PARAM
ASTMETHOD_PARAMETERS
ASTOP_PRIO_1
ASTOP_PRIO_3
ASTOP_PRIO_4
ASTOP_PRIO_6
ASTOP_PRIO_8
ASTOP_PRIO_9
ASTOP_PRIO_11
ASTOP_PRIO_12
ASTOP_PRIO_13
ASTOP_PRIO_14
ASTOP_PRIO_15
ASTOPERATOR_3
ASTOPERATOR_4
ASTOPERATOR_6
ASTOPERATOR_8
ASTOPERATOR_9
ASTOPERATOR_11

ASTOPERATOR_12
ASTOPERATOR_13
ASTOPERATOR_14
ASTOPERATOR_15_ARRAY_INDEX_CALL
ASTOPERATOR_15_MEMBER_SELECTOR
ASTOPERATOR_15_METHOD_CALL
ASTOPERATOR_15_METHOD_CALL_PARAM
ASTOPERATOR_15_METHOD_CALL_PARAMS
ASTPROGRAM
ASTSEMICOLON
ASTST_ELSE_EXPR
ASTST_ELSE_IF
ASTST_FOR
ASTST_FOR_EACH_BODY
ASTST_FOR_EACH_BODY_ARR
ASTST_FOR_EACH_BODY_DECL
ASTST_FOR_EXPR
ASTST_FOR_NORMAL_BODY
ASTST_FOR_NORMAL_BODY_COND
ASTST_FOR_NORMAL_BODY_DECL
ASTST_FOR_NORMAL_BODY_INC
ASTST_IF
ASTST_IF_COND
ASTST_IF_EXPR
ASTST_RETURN
ASTST_WHILE
ASTST_WHILE_COND
ASTST_WHILE_EXPR
ASTTYPE
ASTTYPE_ARRAY
ASTTYPE_TYPESAFE
ASTVAR_DECLARATION
ASTVAR_INIT

Tabelle 2: Alle möglichen Typen von AST Knoten

Um die Funktionsweise der Grammatik zu verdeutlichen, wird ein AST in Listing 5 aus dem Quelltext in Listing 4 erzeugt und verschriftlicht.

```
// resources/beispiel.txt
void main() {
  Path p = D:\Arbeit\testing\test2;
  Set<int> setA = [0, 1, 2];

  int x = 5;
  int y;
  int z = y = x++ + x / 2;
}
```

Listing 4: Beispiel eines akzeptierten Source Code

```
// Ausgabe
PROGRAM = null
 METHOD_DECLARATION = null
  TYPE = void
  LITERAL_IDENTIFIER = main
  METHOD_PARAMETERS = null
  BLOCK = null
   VAR_DECLARATION = null
    TYPE = Path
     VAR_INIT = null
      LITERAL_IDENTIFIER = p
      OP_PRIO_1 = =
       LITERAL_PATH = D:\Arbeit\testing\test2
   SEMICOLON = null
   VAR_DECLARATION = null
    TYPE = Set
     TYPE_TYPESAFE = null
      TYPE = int
    VAR_INIT = null
     LITERAL_IDENTIFIER = setA
      OP_PRIO_1 = =
       ARRAY_CONTAINER = null
        ARRAY_ELEMENT = null
         LITERAL_INTEGER = 0
        ARRAY_ELEMENT = null
         LITERAL_INTEGER = 1
        ARRAY_ELEMENT = null
         LITERAL_INTEGER = 2
    SEMICOLON = null
    VAR_DECLARATION = null
     TYPE = int
      VAR_INIT = null
       LITERAL_IDENTIFIER = x
       OP_PRIO_1 = =
        LITERAL_INTEGER = 5
    SEMICOLON = null
    VAR_DECLARATION = null
```

```
38      TYPE = int
39      LITERAL_IDENTIFIER = y
40    SEMICOLON = null
41    VAR_DECLARATION = null
42      TYPE = int
43       VAR_INIT = null
44        LITERAL_IDENTIFIER = z
45        OP_PRIO_1 = =
46         VAR_INIT = null
47          LITERAL_IDENTIFIER = y
48          OP_PRIO_1 = =
49           OP_PRIO_11 = null
50            OP_PRIO_14 = null
51             LITERAL_IDENTIFIER = x
52             OPERATOR_14 = ++
53            OPERATOR_11 = +
54            OP_PRIO_12 = null
55             LITERAL_IDENTIFIER = x
56             OPERATOR_12 = /
57             LITERAL_INTEGER = 2
58    SEMICOLON = null
```

Listing 5: Erzeugter AST eines akzeptierten Source Codes in Listing 4

4 Semantik und Fehlermeldung

Nach der Erzeugung des AST wird dieser der semantischen Prüfung übergeben. Hierbei wird gleichzeitig auch die Fehlerbehandlung (und die Codeerzeugung in Kapitel 5) abgedeckt. Um den Umfang der semantischen Analyse zu beschreiben, lassen sich hierfür die definierten Exceptions anbringen.

Num	Name
1	ExpectedTypeMissmatchSemanticException
2	IllegalClassContentSemanticException
3	IllegalInitializerSemanticException
4	IllegalOperationSemanticException
5	MainMethodWithArgsSemanticException
6	MethodDeclaredSemanticException
7	MethodNotDeclaredSemanticException
8	MethodParameterMismatchSemanticException
9	NoMainMethodSemanticException
10	NoReturnSemanticException
11	NotArrayExceptionSemanticException
12	ReturnTypeMismatchSemanticException
13	StatementExpectedSemanticException
14	UnreachableCodeSemanticException
15	VariableDeclarationNotAllowedSemanticException
16	VariableDeclaredSemanticException
17	VariableNotDeclaredSemanticException
18	VariableNotInitializedSemanticException
19	WrongArrayDefinitionSemanticException

Tabelle 3: Alle für die semantische Analyse definierten Exceptions

Folgend werden in Listings 6 bis 24 Beispiele für fehlerhaften Code gezeigt, welche in Exceptions in Tabelle 3 resultieren.

```
1 // ExpectedTypeMissmatchSemanticException
2 void main() {
3   for(;34;) {} // EXCEPTION -> Condition != Boolean
4   Set<int> mySet = [1, 2, 3];
5   for(String val : mySet) {} // String != int
6   Set<boolean> myBoolSet = [true];
7   myBoolSet = myBoolSet + mySet; // Set<int> != Set<boolean>
8   int x = true; /* int != boolean */ }
```

Listing 6: Fehlerhafter Code resultierend in einer ExpectedTypeMissmatchSemanticException

```
for (;;) {} // EXCEPTION -> Außerhalb einer Methode
void main() {}
```

Listing 7: Fehlerhafter Code resultierend in einer IllegalClassContentSemanticException

```
NoExample f = /* init */;
void main() {
  f.getName() = "dsf"; // EXCEPTION -> Prio 15
}
```

Listing 8: Fehlerhafter Code resultierend in einer IllegalInitializerSemanticException

```
void main() {
  int x = true + 1; // EXCEPTION -> boolean + int
  String x1 = "hello" + 1; // KEINE EXCEPTION -> String + _ = String
}
```

Listing 9: Fehlerhafter Code resultierend in einer IllegalOperationSemanticException

```
void main(int x) {} // EXCEPTION -> Main Methode darf keine Args haben
```

Listing 10: Fehlerhafter Code resultierend in einer MainMethodWithArgsSemanticException

```
void main() {}
void main() {} // EXCEPTION -> Methode bereits deklariert
```

Listing 11: Fehlerhafter Code resultierend in einer MethodDeclaredSemanticException

```
void main() {
  m404(); // EXCEPTION -> Methode noch nicht deklarierts
}
```

Listing 12: Fehlerhafter Code resultierend in einer MethodNotDeclaredSemanticException

```
void main() {
  main(5, true); // EXCEPTION -> Parameter stimmen nicht überein
}
```

Listing 13: Fehlerhafter Code resultierend in einer MethodParameterMismatchSemanticException

```
int x = 5; <EOF> // EXCEPTION -> Keine Main Methode definiert
```

Listing 14: Fehlerhafter Code resultierend in einer NoMainMethodSemanticException

```
int calc() { int x = 5; } // EXCEPTION -> Kein Return Statement
```

Listing 15: Fehlerhafter Code resultierend in einer NoReturnSemanticException

```
void main() {
  int x = 5;
  int y = x[1]; // EXCEPTION -> Arrayzugriff obwohl keine Array
}
```

Listing 16: Fehlerhafter Code resultierend in einer NotArrayExceptionSemanticException

```
void main() {}
int calc() {
  return true; // EXCEPTION -> int erwartet, aber boolean bekommen
}
```

Listing 17: Fehlerhafter Code resultierend in einer ReturnTypeMismatchSemanticException

```
void main() {
  for(5;;) {} // EXCEPTION -> 5 ist kein Statement
  for(;;6) {} // EXCEPTION -> 6 ist kein Statement
}
```

Listing 18: Fehlerhafter Code resultierend in einer StatementExpectedSemanticException

```
void main() {
  return;
  int x = 5; // EXCEPTION -> Nicht erreichbar
}
```

Listing 19: Fehlerhafter Code resultierend in einer UnreachableCodeSemanticException

```
void main() {
  for(;;) int x = 5; // EXCEPTION -> Nicht erlaubt
  while(true) int x = 5; // EXCEPTION -> Nicht erlaubt
  if(true) int x = 5; // EXCEPTION -> Nicht erlaubt
}
```

Listing 20: Fehlerhafter Code resultierend in einer VariableDeclarationNotAllowedSemanticException

```
void main() {
  int x = 5;
  boolean x = false; // EXCEPTION -> Bereits deklariert
}
```

Listing 21: Fehlerhafter Code resultierend in einer VariableDeclaredSemanticException

```
void print(int i) {}
void main() {
  print(x); // EXCEPTION -> x nicht deklariert
}
```

Listing 22: Fehlerhafter Code resultierend in einer VariableDeclaredSemanticException

```
  void print(int i) {}
  void main() {
    int x;
    print(x); // EXCEPTION -> x nicht initialisiert
  }
```

Listing 23: Fehlerhafter Code resultierend in einer VariableNotInitializedSemanticException

```
1 void main() {
2   Set<int> = {1,2,3}; // EXCEPTION -> [] für Set
3   int[] = [1,2,3]; // EXCEPTION -> {} für Arrays
4   Map<int[], Set<int>> ms = [{1,2,3}: [3,4,5]] // KEINE EXCEPTION
5 }
```

Listing 24: Fehlerhafter Code resultierend in einer WrongArrayDefinitionSemanticException

5 Übersetzung

Die Übersetzung ist der letzte Schritt in Tabelle 1 und ist dafür verantwortlich, den erhaltenen AST zu Java Source Code zu übersetzen. Dieser wird im Anschluss compiliert und zu einer ausführbaren jar-Datei verpackt. Der Java Source Code wird hierbei während der semantischen Analyse erzeugt, um auf die kontextabhängigen Closures zuzugreifen. Als Beispiel einer Übersetzung lässt sich Listing 25 heranführen. Hier wird Mustercode in Listing 4 zu folgendem Java Source Code übersetzt:

```
1 // GENERATED CODE
2 public void main() {
3   Path p = new Path("D:\\Arbeit\\testing\\test2");
4   Set<Integer> setA = new Set<Integer>().add(0).add(1).add(2);
5   int x = 5;
6   int y;
7   int z = y = x++ / 2 + x / 2;
8 }
```

Listing 25: Ausschnitt des generierten Java Codes vom Source Code in Listing 4

Der in Listing 25 generierte Java Source Code wird in einer Wrapperklasse eingefügt, wobei Hilfsklassen wie *Set<T>, Map<TA, TB>, System, Path, Files* mit geforderten Methoden der Aufgabenstellung vorliegen. Die gezeigte Methode *main* in Listing 25 wird durch die Einsprungsmethode *main* der Oberklasse aufgerufen.

6 Anleitung zur Ausführung

In diesem Kapitel wird beschrieben, wie das kompilierte Programm verwendet werden kann sowie die selbstständige Erstellung aus dem Source Code.

Verwendung des Compilers
Das Programm *FARECompiler.jar* akzeptiert zwei Parameter und kann wie folgt aufgerufen werden:
java -jar FARECompiler.jar <InputFile> [OutputFolder]
InputFile gibt den Pfad zur Textdatei an, in welcher sich der Quellcode befindet. *OutputFolder* gibt jeweils den Pfad zum Ordner an, in welchem das kompilierte FARE Programm abgespeichert werden soll. Wird der *OutputFolder* nicht gesetzt, wird der generierte AST + Java Code nur auf der Konsole angezeigt. Um das angehängte Beispiel auszuführen, nutzt man z. B. folgenden Befehl:
java -jar ../FARECompiler.jar aufgabe.fare ./
Sollten zwei Parameter vorliegen, wird bei erfolgreichem Kompilieren eine ausführbare jar-Datei am *OutputPath* erstellt. Die kompilierte jar kann dann mit folgendem Befehl gestartet werden:
java -jar TranslatorTemplate.jar

Kompilierung des Source Codes
Um den FARE Compiler selbst zu kompilieren, muss der Source Code als Maven Projekt initialisiert werden. Für die bloße Generierung der JavaCC und JJTree Klassen, wird folgender Maven Befehl benötigt:
mvn generate-sources
Hierbei werden die durch die Grammatik1.jjt definierte JavaCC Klassen unter dem Ordner `target/generated-sources` angelegt. Um das Maven Projekt jedoch in eine ausführbare jar-Datei umzuwandeln, wird folgender Befehl benötigt:
mvn clean generate-sources package
Hierbei wird das Projekt kompiliert und unter `target` als ausführbare jar-Datei angelegt.

Tabellenverzeichnis

Listings

Literatur

Robert Sheldon (2023). *Was ist Compiler (Kompiler)? - Definition von WhatIs.com*. Hrsg. von ComputerWeekly. URL: https://web.archive.org/web/20230401195755/https://www.computerweekly.com/de/definition/Compiler-Kompiler (besucht am 18. 09. 2023).

Versicherung der Eigenständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Johannes Lang
Matr. 7217450
Henning Müller
Matr. 7105852
Wladislaw Jerokin
Matr. 7205290
Dortmund, den 4. Oktober 2023