Dokumentation Compilerbau Projekt

Compsognathus Compiler

1. Spezifikation des Projektes

Deklarationen:

- Σ: Eingabe–Alphabet
  - JC: Menge aller syntaktisch korrekten Java–Klassen mit folgenden Einschränkungen:
  - keine generischen Klassen
  - keine abstrakten Klassen
  - keine Vererbung
  - keine Interfaces
  - keine Threads
  - keine Exceptions – keine Arrays
  - als Basistypen sind nur int, boolean und char zugelassen string?
  - keine Packages
  - keine Imports
  - keine Lambda–Expressions
- BC: Menge aller Bytecode–Files

Eingabe:

- $p \in \Sigma*$

Vorbedingung:

- $\emptyset$

Ausgabe:

- $bc \in BC* \cup \{error\}$

Nachbedingungen:

- falls $p \in (JC)*$, so ist $bc \in (BC)*$ und p wird nach bc übersetzt wie es durch die Sprache Java definiert ist.
- falls $p 6\in (JC) *$, so ist bc = error.

2. Reihenfolge der Implementierung

1. Leere Klasse
2. Leere Methode
3. Übergabeparameter
4. Rückgabewerte
5. Lokale Variablen
6. Globale Variablen
7. Schleifen und Bedingungen
8. Methodenaufrufe
9. Objekt erstellen

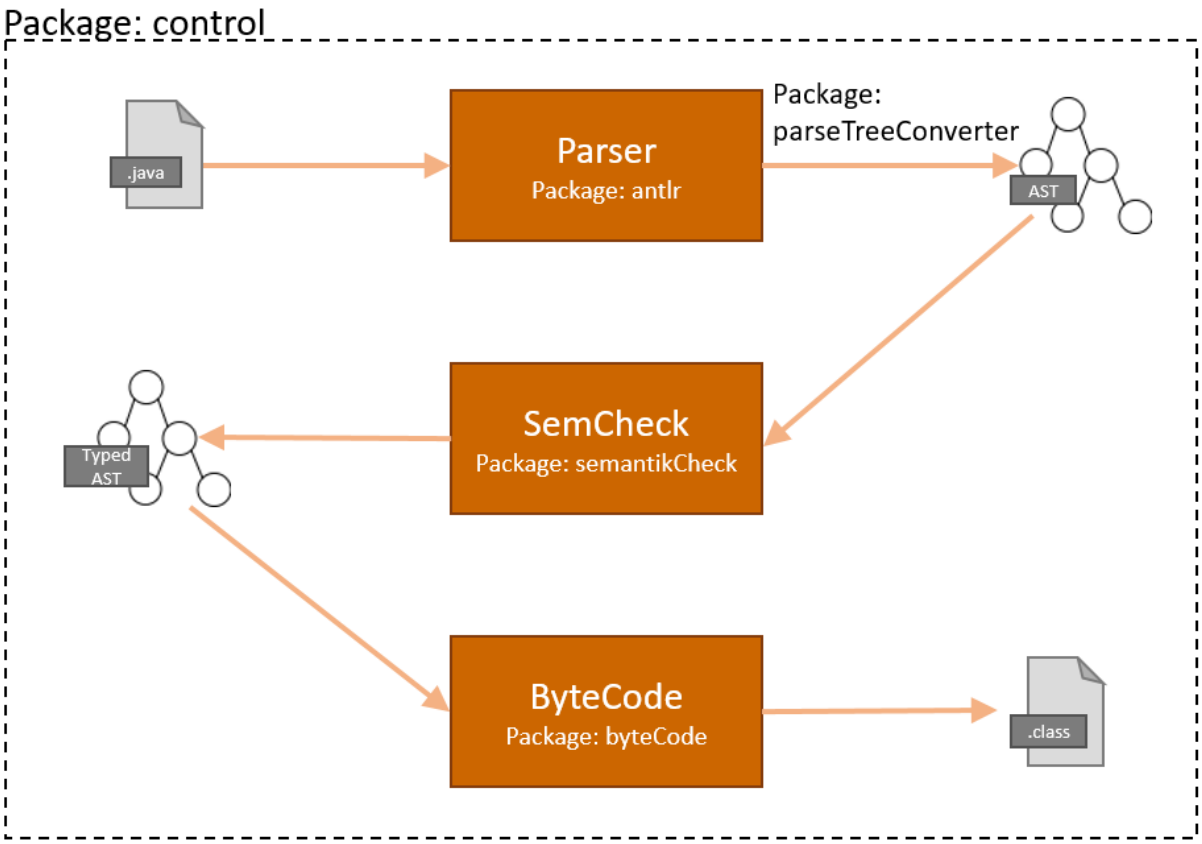
3. Einteilung der Gruppe

- Tobias Hahn: ANTLR | Scannen, Parsen, Grammatik
- Timo Zink: Semantische Analyse | Typisierung
- Fabian Eilber: Codeerzeugung | Bytecodegenerierung
- Lars Holweger: Tester | Testsuite, JUnit-Tests

Github Repository: https://github.com/FABI-BFR/Compsognathus

4. Aufbau des Compilers

Unser Compiler ist in verschiedenen Paketen unterteilt. Dabei entspricht jedes Paket einer Komponente. Der Compiler hat die Komponenten „antlr“, „parseTreeConverter“, „semantikCheck“, „bytecode“, „control“ und „test“. In der Komponente „control“ befindet sich die Klasse Control, die die Main-Methode enthält, die jeweils für die einzelnen Komponenten die Methoden aufruft, um ein fertiges Programm zu erstellen.
Im Paket „test“ befindet sich ein Unterordner, in dem eine Vielzahl syntaktisch korrekter .java-Files liegen. Diese werden mittels einer Testmethode einzeln geprüft.



5. Beschreibung der einzelnen Komponenten

Im folgenden Kapitel werden die einzelnen Komponenten, aus denen der Compiler aufgebaut ist, genauer mittels Text oder UML Klassendiagrammen erläutert.

Parser: antlr

Hauptverantwortlich: Tobias Hahn

Mit dem ANTLR Tool kann aus einer g4-Grammatik ein Parser generiert werden. Für das Projekt wurde die gegebene Grammatik auf das von der Gruppe erstellte UML-Klassendiagramm und die Einschränkungen, die in der Aufgabenstellung gegeben waren, angepasst. Zudem wurden Linksrekursionen aufgelöst sowie weitere Änderungen getroffen, um die Grammatik den Anforderungen von ANTLR zu entsprechen. In der Datenstruktur finden sich alle von ANTLR generierten Dateien in dem Pfad „src/main/java/antlr/“. Dabei sind alle Dateien, bis auf „AntlrParser.java“ und „Compiler_grammar.g4“. Die restlichen Dateien die den Präfix „Compiler_grammar“ enthalten, sind alle von ANTLR aus der g4-Grammatik generiert.

Über die Klasse „AntlrParser“ wird der von ANTLR generierte Parser angesprochen.

Bei Aufruf des Parsers mittels der Methode „parse“, wird dann aus dem übergebenen File ein der Grammatik entsprechender ParseTree erzeugt. Der ParseTree ist so aufgebaut, dass an oberster Stelle die CompilationUnit steht. Die CompilationUnit steht hierbei für das komplette Programm. Durch Herabsteigen des ParseTrees, verzweigt sich der Baum in alle Klassen, Methoden, Deklarationen, usw. die das Programm enthält. Dabei bildet der ParseTree die Syntax des Baums ab.

Konverter: parseTreeConverter

Hauptverantwortlich: Tobias Hahn und Lars Holweger

Der parseTreeConverter bildet aus dem ParseTree der nach dem parsen entsteht einen abstrakten Syntaxbaum (AST). Der Converter geht dabei den ParseTree von der CompileUnit bis zu den einzelnen Statements und Expressions durch und baut so den AST aus dem ParseTree auf. Durch den Converter wird der Knotenpunkt CompilationUnit zu einem Program. Diese Program besteht aus mehreren Klassen, welche aus Methoden bestehen. So ist der AST bis zur Statement und Expression Ebene aufgebaut. Aufgerufen wird der Converter durch die Methode „convert(ParseTree)“.

Semantische Analyse: semantikCheck

Hauptverantwortlich: Timo Zink

Der vom parseTreeConverter erstellte abstrakte Syntaxbaum wird im semantikCheck zu einem getypten Syntaxbaum umgewandelt. Dafür nimmt er die Typen der Statements und Expressions, also die Blätter des Baums, und trägt diese nach oben. Währenddessen überprüft der semantikCheck, ob alle Typen übereinstimmen. Stimmen die Typen überein, werden sie in den höher liegenden Knoten des Baums eingetragen. Jede Klasse des abstrakten Syntaxbaums implementiert das Interface mit der Methode semCheck. Die Methode überprüft die Semantik der jeweiligen Klasse bzw. Objektes. Die Prüfung durch den semantikCheck wird durch das Aufrufen von checker.check(generatedProgram) gestartet. generatedProgram wird zuvor von parseTreeConverter erstellt und zurückgegeben. Dies löst in der Hilfsklasse Checker die check Methode aus, die zunächst die Fehlerliste leert und die semCheck-Methode der Program Klasse aufruft. Der semantikCheck durchläuft den Syntaxbaum von oben nach unten, beginnend bei Program. Hier wird überprüft, ob das eingegebene Program

alle seine Klassen kennt und ob die Klassen verschiedene Namen haben. Wenn dies übereinstimmt, wird die semCheck-Methode der Klasse Class aufgerufen. Die semCheck-Methoden der Elternknoten rufen jeweils die semCheck-Methode ihrer Kinder auf. Die semCheck-Methoden prüfen, ob es sich dabei um die richtigen Typen handelt. Aus den Expressions und Statements werden dadurch konkrete typisierte Expressions und Statements. Falls dabei Fehler in der Semantik gefunden werden, werden diese in einer Fehlerliste gesammelt, die nach dem gesamten Durchlaufen des Programms ausgegeben wird. Wenn es Fehler gibt, wird die Fehlerliste ausgegeben und der Compiler konnte das Programm nicht kompilieren. Die Fehlerliste wird als Ganzes ausgegeben und der Benutzer sieht, in welcher Klasse es zu welchem Fehler kam. Der getypte Syntaxbaum wird wieder in generatedProgram gespeichert und sollte es keine Fehler geben, wird er dem Bytecode übergeben.

Bytecode-Erzeugung: bytecode

Hauptverantwortlich: Fabian Eilber

Die Bytecodegenerierung dieses Projekts wurde mit der ASM–Bibliothek realisiert. Im Package bytecode liegt die Klasse ByteCodeGenerator. Durch den Methodenaufruf „generator.generate(generatedProgram)“ in der Main-Methode im Package Control wird die Bytecodegenerierung gestartet. Dabei beinhaltet das generatedProgram den getypten Syntaxbaum des semantikChecks. Nachdem der Bytecodegenerator den getypten Syntaxbaum erhalten hat, geht er das „generatedProgram“ einmal durch. Bei diesem Durchgehen wird für jede Klasse, die in dem generatedProgram gespeichert ist, ein Java-File erstellt. In diesen Java-Files wird dann jeweils der Bytecode für den Konstruktor, Methoden und Felder erzeugt. Durch den Funktionsaufruf von visitBlockStmt() werden anschließend alle Statements des Programms analysiert. Dabei werden die Statements einzeln darauf überprüft, um welche Art von Statement es sich handelt. Dem Ergebnis der Analyse entsprechend werden dann ASM-Befehle ausgeführt, um den Bytecode zu generieren.

6. Tests

Hauptverantwortlich: Lars Holweger

Die Tests für dieses Projekt mittels dem JUnit Framework realisiert. Im Package tests liegen verschiedene Unterordner. Einer dieser Unterordner beinhalten syntaktisch

korrekte Beispiele zu möglichst vielen Variationen der Grammatik. In einem anderen Unterordner finden sich jeweils in gleichnamigen Dateien die daraus resultierenden Abstrakten Syntax Bäume in einem json ähnlichen Format. Jeder dieser Bäume wurde automatisch generiert und von Hand auf Korrektheit bezüglich seiner Ursprungsdatei geprüft. Gegen diese korrekten Bäume kann der Parser und Converter getestet werden. Für den Semantikcheck wird in einem anderen Ordner in der gleichermaßen erstellt und getestete Syntax Baum gespeichert, wobei dieser um die Typen der Objekte erweitert wurde. Für den Bytecode gibt es keine Tests, außer dem händischen Dekompilieren mittels IntelliJ und der Überprüfung ob die beiden Dateien übereinstimmen.

7. Nutzung des Compilers

Der Compiler nimmt als Eingabe Argumente, welche auf .java enden. Um dem Compiler solche Argumente zu übergeben ist es notwendig, den Absoluten Pfad der Datei ausfindig zu machen. Dieser muss dann in den Run/Debug Configurations des Compilers unter Parameter eingetragen werden. Ist dies geschehen, wird die .java-File durch das Ausführen des Compilers kompiliert und in dem Ordner abgelegt, aus dem die .java-File stammt.