Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Compilerbau Prüfungsleistung

Compilerbau Projekt in Haskell

Fabian Rostomily, Lara Schierenberg, Anabel Stammer

07.03.2024

Reviewer

Martin Andreas Plümicke
Department of Todo
DHBW: Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

Chapter 1

Einleitung

In dieser Prüfungsleistung für das Modul "Spezielle Kapitel der Praktischen Informatik: Compilerbau" beschäftigen wir uns mit dem Entwurf und der Implementierung eines Compilers.
Unser Ziel ist es, einen Compiler für Mini-Java-Code zu entwickeln, der diesen in ausführbaren Bytecode für die Java Virtual Machine (JVM) übersetzt.
Die Implementierung erfolgt in der funktionalen Sprache Haskell, die sich gut für die Entwicklung von Compilern eignet. Die Funktionalität von `Haskell` [1] erleichtert die Implementierung der einzelnen Aufgaben durch eine Vielzahl eingebauter Funktionen und die Unterstützung bei der Arbeit mit Datenstrukturen wie Listen oder Bäumen.

Wir werden die verschiedenen Phasen eines Compilers durchlaufen, angefangen beim Scannen und Parsen des Quellcodes bis hin zur Generierung eines Java Classfiles, das zu ausführbarem Code umgewandelt wird.
Zunächst wird der Quellcode gescannt und in lexikalische Einheiten zerlegt, die als Token bezeichnet werden. Nach dem Scannen des Codes wird dieser anschließend einer Syntaxüberprüfung unterzogen, um seine syntaktische Struktur zu analysieren und einen abstrakten Syntaxbaum (AST) zu generieren. Die syntaktische Analyse stellt sicher, dass der Code grammatikalisch korrekt ist, während die semantische Analyse sicherstellt, dass der Code auch logisch korrekt ist. Dabei werden Deklarationen, Typen und Bereiche identifiziert und überprüft, um sicherzustellen, dass der Code den Regeln der Programmiersprache entspricht.

Die nächste Phase ist die Codegenerierung, bei der der Compiler den AST in ausführbaren Code umwandelt. In unserem Fall wird der AST in ein Java Classfile umgewandelt, das Bytecode enthält, der von der JVM ausgeführt werden kann.
Dabei müssen alle relevanten Informationen, einschließlich Klassen- und Methodennamen, im Konstantenpool abgespeichert werden. Dieser Konstantenpool

wird während der Codegenerierung für den Zugriff verwendet. Der AST wird daraufhin nochmals durch iteriert für den Aufbau der Bytecode-Befehle.

Die Aufgaben innerhalb unseres Teams sind wie folgt aufgeteilt:

Einrichtung eines Git-Repositories und Aufbau der abstrakten Syntax.	Gemeinsam
Scannen/Parsen/Grammatik: Implementieren eines Scanners, erstellen einer Mini-Java-Grammatik und Parsen des Quellcodes.	Lara
Semantische Analyse: Typisierung der abstrakten Syntax verantwortlich.	Lara
Codeerzeugung: Aufbau eines abstrakten ClassFiles.	Fabian
Codeerzeugung: Erstellung des Konstantenpools.	Anabel
Codeerzeugung: Umwandlung des abstrakten ClassFiles in Bytecode durch Funktionen bereitgestellt aus der Vorlesung	

Alle Teammitglieder waren daran beteiligt, eine Testsuite von Java-Dateien zu erstellen. Da hierfür keine Person explizit zugeteiligt wurde, bedingt durch die Teamgröße. Daher ist die Anzahl der automatischen Tests auf eine angemessene Anzahl beschränkt.

Unser Repository ist unter `https://github.com/LaraSchi/Compilerbau.git` zu finden. Der Compiler befindet sich im Ordner 'mini-java', der wiederum aus den Ordnern 'app' (Anwendung/main), 'code' (Beispiel-Mini-Java-Code) und 'src' (Quellcode) besteht. Der 'test' Ordner enthält alle Dateien um die Tests laufen zu lassen. Die Installation und Anwendung ist in der README beschrieben.
Mit diesem Projekt konnten wir wertvolle Erfahrungen im Bereich Compilerbau sammeln.

Chapter 2

Scannen Parsen Grammatik

In diesem Kapitel wird die Implementierung des Lexers und Parsers beschrieben, sowie die Definition der Syntax unserer Grammatik. Hierfür verwenden wir Alex und Happy, da sie eine effiziente und elegante Generierung von Lexer und Parser ermöglichen.

Der Lexer, der in der Datei `Lexer.x` definiert ist, ist für die Aufteilung des Eingabequellcodes in Tokens verantwortlich.
Die Token werden dann vom Parser verwendet, um die syntaktische Struktur des Quellcodes zu analysieren. Der Lexer definiert Regeln für verschiedene Token, wie Schlüsselwörter (z.B. `class`, `public`), Datentypen (z.B. `int`, `boolean`), Operatoren (z.B. `+`, `-`), und Literale (z.B. `int`, `boolean`, `char`). Jedes Token wird durch einen entsprechenden Konstruktor im Token-Datentyp repräsentiert.

Der Parser (Happy), der in der Datei `Parser.y` definiert ist, nimmt die Tokens vom Lexer und erzeugt einen abstrakten Syntaxbaum (AST) gemäß der Grammatik der Sprache, welche in der `Syntax.hs` defniiert ist.
Dieser Syntaxbaum wird dann verwendet, um semantische Analysen und Codegenerierung durchzuführen. Der Parser definiert Regeln für verschiedene syntaktische Konstrukte wie Klassen, Methoden, Anweisungen und Ausdrücke. Jede Regel beschreibt, wie ein syntaktisches Konstrukt aus den Tokens aufgebaut wird, und erzeugt entsprechende AST-Knoten.

Das Modul `Syntax.hs` stellt die Datentypen und Strukturen bereit, die den abstrakten Syntaxbaum (AST) der analysierten Programme repräsentieren.
Dieser AST wird während des Parsens des Quellcodes durch den Parser erstellt und bildet die Grundlage für die folgenden Schritte, wie die semantische Analyse und die Codegenerierung.

Chapter 3

Semantische Analyse

In diesem Abschnitt wird die semantische Analyse unseres Compilerprojekts vorgestellt. Die semantische Analyse ist ein entscheidender Schritt im Compilerprozess, bei dem die Struktur und Typisierung des Quellcodes überprüft wird.

3.1 Implementierung des Semantikmoduls

Das Semantikmodul besteht aus Funktionen zur Überprüfung der Typen im Abstract Syntax Tree (AST) und der Repräsentation des analysierten Quellcodes. Die Hauptfunktion `checkSemantics` führt die semantische Analyse für das gesamte Programm durch und gibt, falls Fehler auftreten eine Liste von passenden Meldungen zurück.

```
checkSemantics ::  Program -> (Program, [String])
```

3.2 Typüberprüfung der Programmkomponenten

Die Überprüfung der Typen erfolgt rekursiv durch das AST.
Dabei werden die Klassentypen, Felder und Methoden deklariert und deren Typen gemäß der in der Vorlesung vorgestellten Regeln überprüft.
Hierfür wird eine State-Monade verwendet, um den Typzustand während der semantischen Analyse zu verwalten. Dieser enthält Informationen über den aktuellen Klassentyp, die lokalen und Feldtypen, sowie Fehlermeldungen.

```
    data TypeState = TypeState
        { classType         :: Type,
          localTypeset      :: [(String, Type)],
          fieldTypeset      :: [(String, Type)],
          errors            :: [String] }
            deriving Show

type TypeStateM = State TypeState
```

Figure 3.1: Definition des TypeState-Datentyps und der TypeStateM-Monade

3.3 Besonderheiten der Implementierung

- Typüberprüfung von Ausdrücken mit State Monade: Es wird überprüft, ob Ausdrücke die erwarteten Typen haben. Durch das Verwenden der State Monade kann rekursiv der gesamte AST durchlaufen und modular die einzelnen Regeln geprüft werden.
- Behandlung von Fehlern: Fehlermeldungen werden im Zustand gesammelt und am Ende der Analyse ausgegeben. Hierbei werden Typfehler an der jeweiligen Stelle im AST gekennzeichnet, um sie später zu kennzeichnen.
- Durch die Implementierung eines PrettyPrinter kann nach dem Durchführen des Semantikcheck, der getypte Code, sowie eventuell aufgetretene Fehlermeldungen, übersichtlich und farblich dargestellt werden.

3.4 Anwendung

Um den Semantikcheck getrennt vom restlichen Projekt zu überprüfen, kann man wie folgt vorgehen:

- Die in der `Main.hs` auskommentierten Funktionen `parseAndCheck` und `checkAllExamples` einkommentieren
- Durch `stack ghci` das ghci starten.
- `checkAllExamples` ausführen. (anstatt `main`)

Nun werden alle Beispiele aus dem Ordner `code/semantikCheck` geparsed, gecheckt und geprettyprinted.

Chapter 4

Codeerzeugung

Im folgenden Abschnitt widmen wir uns der Codeerzeugung, einem entscheidenden Schritt im Kompilierungsprozess. Hierbei betrachten wir insbesondere den Konstantenpool sowie das abstrakte Classfile.

4.1 Konstantenpool

Der Konstantenpool (CP) in Java-Klassendateien ist eine Datenstruktur, die Konstanten, Zeichenfolgen, Zahlen und Referenzen speichert. Er spielt eine entscheidende Rolle bei der Verwaltung von Konstanten im erzeugten Bytecode. Der Code für den Aufbau des Konstantenpools im Compilerkontext befindet sich in `src/ConstPoolGen.hs`.
Für diese Aufgabe wurde der State Monad-Ansatz verwendet, um den CP zu verwalten und global darauf zuzugreifen. Der CP ist eine Liste von `CP_Info` gemäß der Struktur in `ClassFormat.hs`. Setter und Getter wurden implementiert, wobei die Methode `addElement` sicherstellt, dass jedes Element nur einmal im Konstantenpool vorhanden ist.
Ursprünglich wurde versucht, die Originalreihenfolge beizubehalten, indem beispielsweise `Class_Info` vor dem darauf verweisenden `Utf8_Info` platziert wurde. Dies erfolgte durch vorübergehende Platzhalter, die später entsprechend angepasst wurden. Aufgrund von Problemen, die daraus resultierten, wurde jedoch auf eine alternative Reihenfolge umgestellt, beispielsweise durch das Platzieren von `Utf8_Info` vor `Class_Info` mit dem korrekten Verweis. Während des Durchlaufs des ASTs mit Pattern-Matching wird der CP für Klassen, Felder und Methoden erstellt, wobei Referenzen berücksichtigt werden.
Ein wichtiger Aspekt des Codes ist die Handhabung von Referenzen. Field-Referenzen können an zwei Stellen auftreten: erstens direkt in den Field-Deklarationen, wobei sie durch den Ausdruck `expr` in `FieldDecl fieldType fieldName expr` indiziert werden, andernfalls wird

`expr` durch `Nothing` repräsentiert. Zweitens können Field-Referenzen als `FieldVarExpr` auftreten.
Unter `MethodCallExpr` werden Methoden-Referenzen generiert. Die Funktion `resolveAndGenerateMethodRefs`, erhält Namen und Eingabewerte des Methoden-Aufrufs, und holt sich anhand beidem die entsprechende Methoden-Deklaration. Das hat den Zweck, dass nur implementierte Methoden als Methoden-Referenz zum CP zugefügt werden, außerdem ist Methoden-Overload somit möglich. Daher mehrere Methoden mit selben Namen aber unterschiedliche Parameter können implementiert werden.
Jede Java-Klasse erbt automatisch von `java/lang/Object`, wenn keine andere Elternklasse explizit angegeben wird. Somit wird der Standardkonstruktor von `java/lang/Object` im CP jeder Klasse als Methoden-Referenz aufgeführt. Bei einer Objektinstanziierung, im AST aufgeführt als `NewExpr`, wird mit einem Abgleich mit dem Klassennamen festgestellt, ob es sich um eine Instanz der jeweiligen Klasse handelt. In diesem Fall wird eine Methoden-Referenz auf den Klassen-Konstruktor abgespeichert. Dies funktioniert mit und ohne einen selbst geschriebenen Konstruktor.

4.2 Abstraktes Classfile

Das abstrakte Classfile stellt die Beschreibung der Struktur der binären Class-Dateien in Java dar. Beim kompilieren einer Java Source-Datei wird ein abstraktes Classfile aufgebaut, welches anschließend in Binärcode umgewandelt wird. Das Classfile-Format ist hierbei Bestandteil der Java Virtual Machine (JVM). Hier sind alle wichtigen Informationen einer definierten Klasse über ihre Methoden, Felder und weiteres vorhanden.
Für den Aufbau eines abstrakten Classfiles wird zunächst das Classfile-Format als Datenstruktur benötigt, welches wir aus der Vorlesung übernommen und angepasst haben und in `src/ClassFormat.hs` zu finden ist. In `src/ClassFileGen.hs` ist die Logik implementiert, welche die Informationen aus dem AST und dem Konstantenpool nimmt und das abstraktes Classfile aufbaut. Einer der zentralen Punkte ist hierbei die Erstellung sogenannter Code-Attribute, welche den Code jeder definierten Methode auf Assembler-Ebene beschreibt. In `ByteCodeInstr.h` ist eine Datenstruktur mit den gängigsten JVM Byte-Code-Instructions definiert, um eine Abstraktion der binären Instruktion in eine leserliche Form, bessere Code-Generierung und leichteres Debuggen zu ermöglichen.
Letzlich befindet sich in `src/CodeGenerator.hs` die Implementierung zur Generierung der JVM-Byte-Codes aus dem Java-Source-Code. Hierzu wird der AST rekursiv in seine einzelnen Bestandteile zerlegt und für jede getypte Expression die jeweilige Bytecode-Instruktion zu einer List hinzugefügt. Dabei

werden für Fieldreferenzen und Methodenreferenzen auf den zugehörigen Eintrag im Konstantenpool verwiesen. Für die Erstellung des Bytecodes wird eine State Monade verwendet in der globale Informationen, wie die aktuelle Bytecode-Länge oder der Return-Type der betrachteten Methode vorgehalten werden. Die aktuelle Bytecode-Länge dient hierbei als Zähler, der bei Hinzufügen einer Instruktion um die Anzahl an Bytes hochgezählt wird. Dies ermöglicht die Implementation der Branch-Instruktionen `while` und `if-else`. Durch den Zähler kann bei Branches angegeben werden, an welche Stelle im Bytecode gesprungen werden muss.
Die implementierte Code-Generierung deckt, bis auf wenige Ausnahmen, die gesamte definierte Syntax unserer Minijava-Sprache ab. Die boolschen Operatoren `and` (&&), `or` (||) und `not` sind nicht implementiert. Außerdem sind Vergleiche von Objekt-Instanzen nicht möglich. Da wir aufgrund der Gruppengröße die Minijava-Sprache auf lediglich eine Klasse reduziert haben, ist die Anzahl der Anwendungsfälle für Instanz-Vergleiche per Definition eingeschränkt.
Nach Erzeugung des Bytecodes auf Assembler Ebene wird schließlich die maximale Stack-Größe anhand der verwendeten Instruktionen und deren Einfluss auf den Stack berechnet bzw. abgeschätzt. Es handelt sich hierbei um eine Abschätzung nach oben, da im Falle von Branch-Statements (`if-else`) beide Pfade zusammen betrachtet werden. Dies erleichtert die Berechnung und ist eine Abschätzung zur sichereren Seite hin, da es besser ist zu viel Speicher auf dem Stack zu reservieren, als zu wenig.

Als Special-Feature wurde bei der Verwendung von Vergleichs-Operationen ($==$, $!=$, $<=$, $<$, $>=$, $>$) bei der Sprungentscheidung in Branch-Statements (`if-else`, `while`), nach dem Vorbild des Java-Compilers, Code-Optimierung für den Bytecode implementiert. Eine einfach Vergleichsoperation resultiert in ein einfaches `if-else` -Statement der Form

$$
\begin{aligned}
&i-1|... \\
&\quad i|ifcmp \text{ jump to: } i+7 \\
&i+3|iconst_0 \\
&i+4|goto \text{ jump to: } i+8 \\
&i+7|iconst_1 \\
&i+8| \text{ start of } if-else
\end{aligned}
$$

Das Ergebnis ($true$ ($iconst_1$) oder $false$ ($iconst_0$)) könnte in einer ein rekursiven Bytecode-Generierung dann in einem `if-else` -Statement als Sprungentscheidung mit bspw. der if_eq-Instruktion weiter verwendet werden. Der Java Compiler optimiert an dieser Stelle allerdings und fast das einfache `if-else` -Statement in die eigentlich `if-else` -Operation mit ein und gibt

Code der Form

$$
\begin{aligned}
&i-1|...\\
&\quad i|ifcmp \text{ jump to: } i+7\\
&i \text{ to } j| \text{ if-block}\\
&i+4|goto \text{ jump to: } i+8\\
&i+7| \text{ else-block}\\
&i+8|...
\end{aligned}
$$

aus. Diese Form der Optimierung wurde ebenfalls implementiert. Nachteil hier ist jedoch, dass die Branch-Berechnung von verschachtelten Branch-Statements nicht mehr trivial ist und in der aktuellen Version zu falschen Referenzen der inneren Branch-Statements, wohin gesprungen werden soll, führt. Dieses Problem konnte bis zum Abgabeschluss nicht behoben werden.

Chapter 5

Special Features

1. print Befehle sind möglich. Allerdings können erstellte ClassFiles nicht ausgeführt werden, da die Implementierung des Kompilierens einer main-Methode fehlt.

2. Für die Leserlichkeit werden im Konstantenpool Deskriptoren eingebaut. Hier ein Beispiel:

```
3|Utf8_Info {tag_cp = TagUtf8, tam_cp = 16, cad_cp = "java/lang/Object", desc = ""}
4|Class_Info {tag_cp = TagClass, index_cp = 3, desc = "java/lang/Object"}
5|Utf8_Info {tag_cp = TagUtf8, tam_cp = 6, cad_cp = "<init>", desc = ""}
6|Utf8_Info {tag_cp = TagUtf8, tam_cp = 3, cad_cp = "()V", desc = ""}
7|NameAndType_Info {tag_cp = TagNameAndType, index_name_cp = 5, index_descr_cp = 6, desc = "<init>:()V"}
8|MethodRef_Info {tag_cp = TagMethodRef, index_name_cp = 4, index_nameandtype_cp = 7, desc = "java/lang/Object.<init>:()V"}
```

Figure 5.1: Beispiel des Deskriptors für eine Method-Referenz.

3. In der AST wird zwischen Globale und Lokale Variablen unterschieden. Handhabung von Lokale Variablen den Namen der Globalen Variable ist effektive gelöst, auch wenn diese den selben Namen teilen (siehe Beispiel-Code `globalLocalSameName.minijava`).

4. Objekt-Deklaration mit individuellem Konstruktor.

5. Method-Overload ist möglich.

6. Bytecode-Optimierung bei Verwendung von Vergleichsoperationen in Branch-Operationen (siehe 4.2).

7. PrettyPrint Darstelung der Zwischenschritte

Literaturverzeichnis

[1] Simon Marlow et al. “Haskell 2010 language report”. In: *Available online http://www. haskell. org/(May 2011)* (2010).