Dokumentation

Compilerbau - INF3339c

WS 2023/24

Dozent:	Martin Plümicke	
Abgabedatum:	07.03.2024	
Von:	Jan Keller	6049995
	Timon Martins	6454072
	Eric Sabel	6443283
	Timo Schmidberger	6608383

Inhaltsverzeichnis

Funktionsumfang

- If-Else, While
- Datentypen int, char, boolean, void, null
- Ein- und mehrzeilige Kommentare
- Parsen aller Grundstrukturen (außer For-Schleifen)
- Parsen von Operatorpräzedenz
- Vererbung
    - @Override muss Methode überschreiben
    - Return-Typ darf nicht spezieller sein
    - Sichtbarkeit darf nicht verringert werden
- Überladung
- Instanzvariablen mit Initialisierung
- Dead-Code Erkennung
- Schöne Fehlermeldungen durch Position-Tracking aller Tokens
- Einfügen von Default-Konstruktor, super(); und return;
- Subset der Standardbibliothek, insbesondere System.out.println
- Debug-Modus

Allgemeiner Aufbau

Der allgemeine Aufbau des Compilers richtet sich nach dem in der Vorlesung vorgestellten Vorgehen. Es werden die Phasen Scanner, Parser, Typecheck, Constant Pool-Erzeugung und Code-Erzeugung durchlaufen.

Der Compiler wurde in Haskell mit Stack umgesetzt. Stack ist ein Tool zum Erstellen von Haskell-Projekten und zum Verwalten ihrer Abhängigkeiten. Weitere Informationen zur Installation etc. sind unter https://docs.haskellstack.org/en/stable/ zu finden. Kompilieren und direkt ausführen lässt sich das Projekt mit dem Befehl “stack run <path>”, wobei <path> den relativen Pfad zur Java-Quelldatei angibt. In dieser Datei sollte sich der gesamte Code des entsprechenden Programms befinden, d.h. alle Klassen sollten in einer Datei definiert werden. Die Erzeugten Class-Files befinden sich dann im Directory der .java Datei.

Startpunkt des Projekts ist die “Main.hs” im Ordner “app”. Hier wird auch über die Variable “debug” bestimmt, ob der Debug-Modus aktiv ist. Durch den Debug-Modus werden die Ausgaben der einzelnen Phasen in der Konsole ausgegeben. Dadurch kann Schritt für Schritt nachvollzogen werden, welche Ergebnisse die einzelnen Phasen erzeugt haben. Die erstellten Class Files werden im Ordner der Quelldatei gespeichert und können dort wie normale Java-Programme ausgeführt werden, z.B. durch “java Main”.

Abstrakte Syntax

Die abstrakte Syntax (AST) und getypte abstrakte Syntax (TAST) orientieren sind an der Volage aus der Volesung, enthalten aber einige Änderungen.
AST und TAST unterscheiden sich in den enthaltenen Informationen und sind deshalb um bessere Typsicherheit zu erreichen in den separaten Dateien “src/Types/AST.hs” und “src/Types/TAST.hs” definiert. Gemeinsamkeiten finden sich in “src/Types/core.hs”.
Die wichtigsten Änderungen zur Vorlesung sind:

- Im AST werden zu Expressions, Statements und fast allen Eigenschaften Positionen gespeichert, um darauf basierend anschauliche Fehlermeldungen anzeigen zu können
- Es wird der Record-Syntax verwendet, um eine bessere Lesbarkeit des Codes zu gewährleisten
- Methoden und Konstruktoren sind getrennt, da diese oft unterschiedliche Eigenschaften haben, z.B. sind “this();” und “super();” nur in Konstruktoren erlaubt
- Statische Felder/Methoden werden unterstützt
- Vererbung wird unterstützt, damit gehen Eigenschaften wie Access-Modifier, @Override-Annotationen und Extends einher.
- Im TAST werden besonders in Statements und Expressions wie “MethodCall” einige Informationen gespeichert, die für die korrekte Ansteuerung der JVM benötigt werden.

Scanner und Parser

Erstellt von Eric Sabel

Scanner

Zu Beginn des Kompiliervorgangs steht das Scannen, bei dem die Quelltextdatei als String eingelesen wird und daraus eine Liste von Tokens erzeugt wird. Alle möglichen Tokens werden in der Datei “src/Scanner/Token.hs” definiert, wobei auch drei temporäre Token enthalten sind, die nach erfolgreichem Scannen nicht mehr auftreten. Die Kern-Funktionalitäten sind in der Datei “src/Scanner/Lexer.hs” definiert.

Als Erstes wird in der Funktion `lexer`, welche den Java Quelltext als String erhält, dieser String in eine Liste von Tokens umgewandelt. Die Funktion orientiert sich an der, in der Vorlesung vorgestellten, Vorlage, wobei auch noch einige Änderungen vorgenommen wurden. Beispielsweise werden Leerzeichen oder Zeilenumbrüche nicht gänzlich ignoriert, sondern auch zunächst als Tokens in die Liste aufgenommen, um zu einem späteren Zeitpunkt noch Positionsinformation über die tatsächlich relevanten Tokens herausfinden zu können. Zudem werden sowohl InLine- als auch MultiLine-Kommentare erkannt und durch entsprechende Leer- oder Zeilenumbruchzeichen temporär ersetzt. Zusätzlich werden nicht geschlossene oder falsch geschlossene characters als fehlerhaft erkannt und somit zunächst als *WRONGTOKEN* klassifiziert. Dadurch bleiben die Positionen aller Tokens zunächst erhalten, und es kann dem Nutzer potenziell mitgeteilt werden, wo ein entsprechender Fehler auftritt.

Nach dem initialen Einlesen wird die ursprüngliche Position der Tokens anschließend anhand der beibehaltenen *SPACE* und *NEWLINE* Token ermittelt, wobei auch die textliche Länge einzelner Tokens wie z.B. *‘null’* oder *‘public’* über eine Zuordnungstabelle berücksichtigt wird. Die temporären Tokens werden dabei entfernt und sind nicht in der Liste von `PositionedToken` enthalten. Der Record-Typ `PositionedToken` enthält dabei das eigentliche Token und eine Position, bestehend aus Start- und End-Koordinaten des jeweiligen Tokens.

Als letztes werden die Tokens noch validiert durch `validateTokens`. Diese Funktion geht noch ein letztes mal über die Token-Liste und überprüft, dass keine zu großen Integer-Literale auftreten, keine falsch eingegenbenen Character-Literale auftreten (zuvor durch *WRONGTOKEN* erkannt) oder auch, ob eventuell ein String-Literal eingegeben wurde, was unser Compiler nicht unterstützt. Wird ein solcher Fehler gefunden, wird der gesamte Kompiliervorgang mit einer entsprechenden Fehlermeldung abgebrochen.
Nach der Validierung ist die Scan-Phase abgeschlossen und die gesamte Funktionalität wird durch die Funktion scanner zusammengefasst.

Parser

Die unstrukturierte Token-Liste muss nun durch den Parser in die gewünschte AST-Struktur überführt werden. Für die Implementation wurde der Kombinatorparser, nach der Vorlage aus der Vorlesung, gewählt.
Die Kombinatoren wurden in der Datei “src/Parser/Combinators.hs” definiert. Zusätzlich, zu den bereits bekannten Kombinatoren wurde noch der Kombinator `posLexem` hinzugefügt, der äquivalent zum Parser `lexem` funktioniert und einzelne Lexeme erkennt, wobei dieser jedoch auch für positionierte Token, also `PositionedToken` funktioniert. Außerdem wird für den Parse-Vorgang bei den Tokens *IDENTIFIER*, *INTLITERAL, BOOLLITERAL, CHARLITERAL*, welche alle jeweils ein Konstruktor- Argument besitzen um das tatsächliche Literal bzw, den tatsächlichen Identifier-namen zu beinhalten, jeweils das Argument ignoriert, da der Parser hierfür nur erkennen soll ob ein entsprechender Token-Typ vorliegt, wobei beispielsweise der tatsächliche Integer-Wert an dieser Stelle für das Parsen keine Rolle spielt. Außerdem wurde der Parser `many` hinzugefügt, welcher für einen gegebenen Parser einen Parser erzeugt, der entweder nichts, oder beliebig oft sequenziell das erkennt, was der gegebene Parser erkennt und dann das Ergebnis in einer Liste zusammenfasst. Die `many` Funktion realisiert sozusagen die Kleenesche Hülle eines Parsers.
Der Parser für ein Java Programm mit den entsprechenden Vereinfachungen befindet sich in der Datei “src/Parser/Parser.hs”. Hier wird für jeden Teilparser in einem Kommentar davor symbolisch beschrieben, welche Grammatik dieser jeweils

erkennt. Dabei ist immer mit großgeschriebenen Wörtern ein Nichtterminal gemeint, mit kleingeschriebenen in '...' Apostrophen geschriebenen Wörtern ein Terminal, mit dem vertikalen Strich | die Unterscheidung, mit dem Doppelpunkt : die Aneinanderreihung und mit einem 'e' das leere Wort gemeint. Die einzelnen Parser Bausteine funktionieren bis auf die im Folgenden aufgeführten Ausnahmen intuitiv und sollten durch die Annotationen ausreichend erklärt sein.

Klasseneinträge

In Java Dateien können Klassenmethoden, Klassen-Konstruktoren und Klassenfelder in beliebiger Reihenfolge stehen. Daher werden auf Klassenebene diese drei Fälle durch den Typ 'ClassEntry' zusammengefasst, wobei die einzelnen Komponenten dann hieraus extrahiert und schließlich strukturiert in den AST eingefügt werden können.

Elimination von Linksrekursion in Expressions und Operatorpräzedenz

Um die Linksrekursion in Ausdrücken zu eliminieren, sodass die Parser Phase nicht zu einem Stackoverflow führt, wurden Ausdrücke, bei denen ganz links wieder ein Ausdruck steht von denen, bei denen dies kein Problem ist, getrennt.
Z.B. $Expr = Expr + Expr$ oder $Expr = Expr.Field$ würde zu einem solchen Problem führen.
Ausdrücke ohne dieses Problem (terminierende Ausdrücke) werden nun mit $TExpr$ geparsed und bei den anderen wird einmal der Linke Teil durch $TExpr$ geparsed und dann der rechte Teil mit $Expr'$, der eigentliche Ausdruck wird also aufgeteilt in linken und rechten Teil. Diese Aufteilung wird auch bei $StmtOrExpr$ gemacht, da z.B. eine Zuweisung einer lokalen Variablen, links keinen Ausdruck besitzt, die Zuweisung einer Feldvariablen auf einem Objekt jedoch schon.
Da im AST jedoch nur vollständige Expressions vorkommen sollen wird der rechte Teil, also z.B. das `+ 5` vom Ausdruck `4 + 5` in einer Temporären Struktur `RightSideExpr` repräsentiert. Nachdem linker und rechter Teil erfolgreich nacheinander geparst wurden, wird dann der linke Teil (`4`) und der rechte Teil (`+5`) durch die Funktion `resolveRightSideExpr` zu einem AST-Konformen Ausdruck zusammengesetzt. Ein günstiger Nebeneffekt dabei ist, das sich bei diesem

Zusammensetzen von linkem und rechtem Teil, die Operatorpräzedenz leicht umsetzen lässt. So wird beispielsweise beim Ausdruck `4 * 5 + 6` am Ende der linke Teil (`4`) und rechte Teil (`* 5 + 6`) zusammengesetzt. Die Funktion `resolveRightSideExpr` erkennt nun, dass im rechten Teil wieder ein binärer Ausdruck ist, bei dem eventuell eine Umsortierung notwendig ist. Also werden die Präzedenzen von `+` und `*` verglichen, und die Ausdrücke dann entsprechend umsortiert, bevor sie in den AST geschrieben werden.

Positions-Übersetzung

Da insbesondere für den anschließenden Typecheck die Positionsinformation für sinnvolle Fehlermeldungen wichtig ist, werden während des parsens die Positionen aus der Token-Liste mit in den AST übernommen. Wie zuvor bereits erwähnt, erkennt der Parser `posLexem` Tokens, die mit ihrer Position gewrapped sind. Die elementaren Parser lassen sich dann wie gewohnt mit dem (+.+) - Kombinator für sequenzielle Erkennung zusammenbauen, beim “reduktions”-Schritt durch den (<<<) Kombinator muss dann noch darauf geachtet werden, dass die einzelnen Parser Ergebnisse mit ihren Positionen als `PositionedToken` vorliegen, wovon sich jedoch ebenfalls mit lambda Ausdrücken einfach die Positionsinformation extrahieren und dann schließlich in die jeweilige AST-Teilstruktur übernehmen lässt.

Finalisieren

Nachdem der Parse-Vorgang abgeschlossen wird und das Ergebnis als Liste vom Typ `[(Program, [PositionedToken])]` vorliegen sollte, wird ebendieses Liste noch nach korrekten Lösungen gefiltert, welche sich durch leere Listen an zweiter Tupel Position auszeichnen. Bei korrekten Lösungen konnte der Input also vollständig geparsed werden. Ist dies nicht der Fall, also gab es keine einzige Lösung, bei der der Input vollständig geparsed werden konnte, so wird vom Parser ein parse-Error angegeben. Für genauere Fehlermeldungen hätte bestimmt werden müssen, an welcher Stelle der Parser gestoppt wurde und diese Fehlermeldung hätte dann als Einzige weitergetragen werden müssen. Somit wäre der Performancevorteil der lazy-Evaluation verlorengegangen, da die Kombinatoren hier in jedem Schritt die Zwischenergebnisse vollständig hätten auswerten müssen.

Type checking

Erstellt von Timon Martins.

Äußere Eigenschaften/Funktionalität

Auf alle Eigenschaften einzugehen ist aufgrund des großen Feature-Sets nicht möglich, daher hier einige interessante Funktionalitäten.

Überladung/Vererbung

Methoden-Überladung und Klassenvererbung werden unterstützt.

Interessant ist hier beispielsweise das Vorgehen beim Auflösen einer Methode. Dieses orientiert sich stark am Vorgehen aus der Java-Language-Specification (JLS):

1. Durchsuche die Vererbungshierarchie der Klasse. Niedrigere Ergebnisse überschreiben höhere.
    a. Filtere Methoden nach gesuchtem Namen
    b. Filtere Methoden, ob sie auf die Argumente anwendbar sind
    c. Filtere Methoden, ob sie vom Aufrufer aus sichtbar sind
    d. Wähle die speziellste Methode aus
2. Prüfe ob die Methode statisch ist und auf der Klasse/Instanz aufgerufen werden kann

In allen Teilbereichen werden viele Fehler des Nutzers abgefangen. Auch bezüglich Überladung/Vererbung sind es einige, darunter:

- Mehrfache Methoden-Definitionen mit denselben Parametertypen
- Uneindeutigkeit der Methode beim Methodenaufruf
- Wenn eine überschreibende Method einen strikteren Access-Modifier hat, als die überschriebene Methode
- Zyklische Vererbungsstrukturen

Dead-Code-Erkennung, “return”-Statements

Das Vorgehen in diesem Abschnitt unterscheidet sich von der Vorlesung.
Die Funktion “checkStmt” gibt u.a. zurück, ob in diesem Statement auf jedem Pfad ein “return” stattfindet. Befindet sich in einem Block nach einem solchen Statement noch Code, ist dieser unerreichbar und wird als Fehler behandelt. Dieselbe Eigenschaft wird genutzt, um bei nicht-”void” Funktionen in “checkMethod” zu prüfen, ob die Funktion auch tatsächlich immer einen Wert zurückgibt.

Ob die zurückgegebenen Werte auch den richtigen Typ haben, wird dagegen in “checkStmt” bei jedem “return” einzeln überprüft.

Schöne Fehlermeldungen

Unter “src/Error/PrintError.hs” ist eine Funktion “printError” zu finden, die zu einer Fehlermeldung den passenden Ausschnitt der Quelldatei ausgibt und den Fehler farblich hinterlegt.
Alle Fehlermeldungen des Typecheckers verwenden diese Funktion mittels “throwPretty”. Dafür müssen alle Funktionen, die einen Fehler werfen können, die entsprechenden Position (siehe Innere Eigenschaften)

Standart-Bibliothek

Es ist möglich, Klassen der Standard-Bibliothek (Stdlib) zu verwenden. Die Typen dafür sind in “src/SemanticCheck/StdLib.hs”. Aktuell ist nur ein Subset implementiert, aber auf diese Weise lassen sich große Teile der Stdlib verfügbar machen.
Insbesondere kann man dadurch die “System.out.println(...);” aufrufen.

Die Stdlib wird im “GlobalCtx” getrennt vom Nutzerprogramm gehalten, so dass die Klassen nicht zum Bytecode-Generierer gelangen. Beim Auflösen einer Klassen wird zuerst geschaut, ob eine Stdlib-Klasse gemeint ist.
(“System” resolved zu “Class{ cname=’java/lang/System’, … }”)

Innere Eigenschaften/Umsetzung

Aufruf-Struktur

Der obere Teil des Aufruf-Baums sieht in etwa wie folgt aus:

- typechek/typecheckM
    - precheckProgram
        - precheckClass
            - precheckConstructor
        - precheckTypes
    - checkProgram
        - checkClass
            - checkField
            - checkMethod
                - checkStmt/checkExpr
            - checkConstructor
                - checkStmt/checkExpr

Code

Fast alle Funktionen des Typcheckers sind im ExceptState-Monad, der eine Kombination aus State-Monad mit GlobalCtx als State und ExceptT-Monad ist. Dadurch müssen der AST, die Stdlib und die Quelldatei nicht immer als Argument übergeben werden. Würden alle den ExceptT-Monad verwendet, wäre außerdem das Testen von Fehlerhaften Code in Unit-Tests möglich, da dieser Exception-Handling realisiert.

In “src/SemanticCheck/Util.hs” finden sich vor allem Typklassen zu Abstraktion über den AST. So kann zum Beispiel “static m” statt “untag $ AST.mstatic m” (m :: AST.Method) geschrieben werden und u.a. duplizierter Code vermieden werden.

In “src/SemanticCheck/Typecheck.hs” befinden sich auch einige Lenses mit Suffix “L”. Diese erleichtern das Modifizieren des AST, z.B. beim Einfügen des Standard-Konstruktors.

Es existieren einige Namenskonventionen, die zum Verständnis hier aufgeführt werden:

- Erstes Zeichen des Type ist Präfix (“moverride” in “Method”)
- Präfix t
    - tagged: Wert mit Position (z.B. Typ “WithPosition Identifier”)
    - type: Ein Type, z.B. “this” wäre der Type von “this”
- Präfix m
    - maybe: “mfound” ist Wert des nur Vielleicht gefunden wurde.

Constant Pool

Erstellt von Jan Keller

Der Constant Pool bildet die Basis, um im Bytecode auf Funktionen, Felder, Konstanten und Klassen zuzugreifen.
Der Constant Pool verwendet das in der Vorlesung vorgestellte JVM Package, mit der Änderung, dass die meisten Datentypen die Show- und Equality(EQ) Instanz deriven um Vergleiche zu vereinfachen, um die Einträge sinnvoll darzustellen. Die Konstruktion und die Suche des Konstanten Pools sind getrennt und unabhängig voneinander. Das bedeutet, der Konstanten Pool wird erst vollständig aufgebaut und erst dann wird er an die Bytecode-Generierung als eine Liste von Funktionen übergeben. Diese Funktionen können verschiedenen Typen von Einträgen im Konstanten Pool suchen. Alle Funktionen die zur Konstruktion oder Suche im Konstanten Pool verwendet werden, verwenden den Haskell `State` Monad um die Übergabe von verschiedensten Parametern, wie dem Konstanten Pool und Field Informationen, zu vermeiden und somit den Code signifikant zu vereinfachen.

Der Konstanten Pool wird durch den folgenden Typ dargestellt:

$$type\ ConstantPool\ =\ [(Int,\ CF.CP\_Info)]$$

wobei der Integer den Index des gegebenen Konstanten Pool Eintrag darstellt.

$$type\ MICP\ =\ (CF.Field\_Infos,\ [(Int,\ CF.CP\_Info)])$$

$$type\ ConstantPoolState\ =\ State\ MICP$$

Der `ConstantPoolState` beinhaltet zudem auch die Field Informationen für alle Fields einer Klasse.
Außerdem werden alle Konstanten, egal welcher Größe, auf den Konstanten Pool geschrieben, sodass keine Unterscheidung zwischen Konstanten verschiedener Längen notwendig ist.
Die wichtigste Funktion für das Einfügen der Einträge ist die `insertEntry` Funktion, die Funktion fügt einen Eintrag in den Konstanten Pool ein. Die Funktion wird immer in dem `ConstantPoolState` Aufgerufen und gibt einen Integer zurück. Dieser Index ist der Index des eingefügten Eintrags. Um zu gewährleisten, dass die Einträge eindeutig sind, wird bei jeder Insertion der gesamte Constant Pool durchsucht, ob

ein Eintrag mit dem gleichen *CF.CP_Info* bereits existiert, falls das der Fall ist, wird der Index des vorhandenen Eintrags zurückgegeben. Desweiteren existieren noch verschiedenste Funktionen, die mit *insert* beginnen, diese Funktionen nutzen die *insertEntry* und *insert*-Funktionen, um Einträge verschiedener Typen in den Constant Pool einzufügen. Durch die Eigenschaft, dass die Insert Entry Funktion immer den richtigen Index im Konstanten Pool zurückgibt, ist es einfach möglich, die Funktionen jedes Mal zu verwenden, wenn der Index eines bestimmten Elementes benötigt wird.
Zudem existiert eine Basisfunktion, welche für die Erstellung eines Descriptor Strings für einen Typen, Felder, Methoden und für einen Return- und Parameter-Typen. Der nächste Schritt ist das Durchlaufen des TAST's, wofür die *traverseStmt*, *traverseExpr*, traverseStmtOrExpr und die traverseLit Funktionen, welche den TAST durchlaufen und gegebenenfalls die Notwendigen Einträge hinzufügen.

Der nächste Schritt ist die Suche nach den Indizes für ein bestimmtes Codesegment (z.b. Konstruktor, Methode, Literal …). Auch bei der Suche wird überall der State Monad verwendet, um den Code zu vereinfachen, was jedoch auch einige Probleme mit sich bringt, da der Code für die Bytecode-Generierung keine Monads und State-Monads verwendet. Die Verwendung des State Monads in der Bytecode-Generierung würde den Code für die Konstruktion des Konstanten Pools stark vereinfachen, da dadurch der Konstanten Pool bzw. die verwendeten Einträge generiert werden könnten, wenn sie im Bytecode benötigt werden.
Da sich die Suche im Konstanten Pool als schwierig herausgestellt hat, da Einträge bottom-up rekursiv durch ihre Teil-Einträge gesucht werden müssen, existieren einige Funktionen, wie die getResult Funktion um Fehlermeldungen zu erhalten, wenn kein passender Eintrag gefunden wird oder die *debugStr* Funktion, welche eine Debug String erstellt, welcher die vorherigen Suchschritte auflistet und so die Herkunft bestimmter Indizes auflistet.
Um in der Bytecode-Generierung einfach und ohne Funktionen den Konstanten Pool durch alle Funktion zu reichen, existiert ein Datentyp welcher die Suchfunktionen einfach zur verfügung stellt darunter sind die folgenden Funktionen:

```
findLiteral :: TAST.Literal -> Int
findMethodCall :: TAST.StmtOrExpr -> Int
```

```
findField :: String -> String -> Core.Type -> Int
findName :: String -> Int
findDesc :: TAST.Method -> Int
findAnyDesc :: Core.Type -> [Core.Type] -> Int
findClass :: String -> Int
findUtf8 :: String -> [Int]
findConstructor :: String -> [Core.Type] -> Int
```

Um die Funktionen ohne den Konstanten Pool und den *State*-Monad nutzbar zu machen, muss für alle Funktionen, die den State Monad verwenden, hilfsfunktionen erstellt werden, welche mithilfe von higher Order Functions, die Funktion im *State*-Monad mithilfe der *runstate*-Funktion und dem finalen Konstanten Pool aufrufen.
Durch diese Schritte ist es bei der Bytecode-Generierung sehr einfach, nach allen Einträgen zu suchen.

Bytecode-Generierung

Erstellt von Timo Schmidberger

Um die entsprechenden Class-Files zu erzeugen, wird das in der Vorlesung vorgestellte JVM-Modul genutzt. Dafür wird für jede Klasse die entsprechende Haskell-Datenstruktur erstellt. Für die Bytecode-Generierung wird dafür der TAST sowie der Constant Pool mit seinen Suchfunktionen verwendet.

Neben den Access Flags der Klasse, dem Constant Pool, der This- und Super-Klasse werden vor allem die “Method Info” für die Methoden und Konstruktoren erstellt. Die “Method Info” enthält hauptsächlich die Verweise auf den Constant Pool und das “Attribute Code”. Im “Attribute Code” werden wichtige Informationen zu jeder Methode definiert, dazu zählen die Anzahl der lokalen Variablen, die maximale Stackgröße sowie der eigentliche Bytecode.

Es ist wichtig, die lokalen Variablen der Methoden zu erfassen. Zum einen muss die Anzahl der lokalen Variablen für jede Methode angegeben werden, zum anderen werden diese Informationen für die Bytecode-Generierung benötigt. Dafür wird mit Hilfe der Funktion “localVarGen” ein Array mit den lokalen Variablen erstellt. Sollte die Methode nicht statisch sein, befindet sich "This" an der Stelle null im Array, gefolgt von den Parametern und den Variablen, die innerhalb der Methode definiert werden. Sollte die Methode statisch sein, befindet sich "This" nicht an der Stelle null im Array, da statische Methoden nicht an eine spezifische Instanz der Klasse gebunden sind. Stattdessen beginnt das Array direkt mit den Parametern der Methode, gefolgt von den Variablen, die innerhalb der Methode definiert werden. Ob eine Methode statisch ist, sowie die Übergabeparameter sind direkt bekannt. Um die Variablen, die innerhalb der Methode definiert werden, zu erkennen, wird der TAST rekursiv durchsucht und die gefundenen Variablen ebenfalls in dem Array gespeichert.

Nachdem die lokalen Variablen der Methode bekannt sind, wird der Byte-Code mit Hilfe der Funktion “codeGenStmt” erstellt, da jede Funktion im TAST mit einem “Stmt” beginnt. Dafür wird ebenfalls rekursiv mittels Pattern Matching der TAST

durchgegangen, so wird der Code Bottum-Up zusammengesetzt. Das Ergebnis dieser Code-Generierung ist ein Array aus Integern. Da die genaue Beschreibung jeder Funktion den Rahmen dieser Dokumentation übersteigen würde, wird im Folgenden nur auf spezielle Teile eingegangen.

- Um auf lokale Variablen zugreifen zu können, wird der Index aus dem oben beschriebenen Array per Namen ausgelesen.
- Bei allen Loads und Stores wird vorher immer “wide” verwendet, um mehr als 256 lokale Variablen zu unterstützen.
- Binäre Operationen, die zu einem Bool führen, werden immer so ausgeführt, dass am Ende der Operation eine 0 für False bzw. eine 1 für True auf dem Stack liegt. Dadurch können While/If-Abfragen immer “ifeq” (Sprung bei 0) verwenden, was zu einer Vereinfachung des Codes führt.
- Je nachdem welche Methode aus dem Code aufgerufen wird, wird entweder “invokespecial”, “invokestatic” oder “invokevirtual” verwendet.
- Bei der Verwendung von "new" wird die Klasse-Referenz mit dem Befehl “dup” zwei mal auf den Stack gelegt, damit der Konstruktor aufgerufen werden kann und die Klasse-Referenz sich nach diesem Aufruf immer noch auf dem Stack befindet.

Um die maximale Stackgröße zu berechnen, wird durch die Funktion “calcMaxStackSize” der erstellte Bytecode analysiert und je nach Befehl die aktuelle Stackgröße erhöht, verringert oder gleich gelassen. Zusätzlich zur aktuellen Stackgröße wird immer die bereits berechnete maximale Größe gespeichert und am Ende als maximale Stackgröße verwendet. Durch diese Verfahren wird die maximale Stackgröße nie zu klein angegeben.