Compilerbau Prüfungsleistungs Projekt

Dies ist die Dokumentation des Mini-Java-Compiler-Bauprojekt der Gruppe NichtHaskell für das 4. Semester Informatik der Dualen Hochschule Baden-Württemberg in Stuttgart (Horb).

1. Grundlegende Informationen

Syntax: Java 4.0

Basistypen: int, boolean, char

Zugriffsmodifikator: public

Operatoren: + - * / || && == != < > <= => ! =

Anweisungen: if-Schleife, if-else-Schleife, while-Schleife, return

Weitere Schlüsselwörter: class, new, this

Ausgabe: print()

ausdrücklich weggelassen:

- keine statischen Methoden
- nur „public static void main(String[] args)“ für die Hauptmethode
- keine Vererbung
- keine Importe und Pakete
- keine Schnittstellen und abstrakten Klassen
- keine Arrays und Exceptions
- „this“ nur in Verbindung mit Zugriff auf Felder oder Methoden
- keine Überladung

Anmerkungen:

- Einlesen nur einer Datei
- Ausdrücke erfordern ein Leerzeichen zwischen Operator und dem Vorzeichen der nächsten Zahl, da es nach Princ. Of. Long. Match den Operator sonst nicht scannt
- Einer Variable kann nicht explizit der Wert null zugewiesen werden „print()“ unterstützt nur lokale Int-Variablen und Felder der eigenen Klasseninstanz, keine anderen Typen oder Konstanten
- Variablen werden mit Standardwerten initialisiert
- output.jar ist nicht für die Ausführung gedacht

2. Benutzte Werkzeuge:

- ANTLR4 v4.9.2

  Wird verwendet, um den Code in einen abstrakten Syntaxbaum zu analysieren

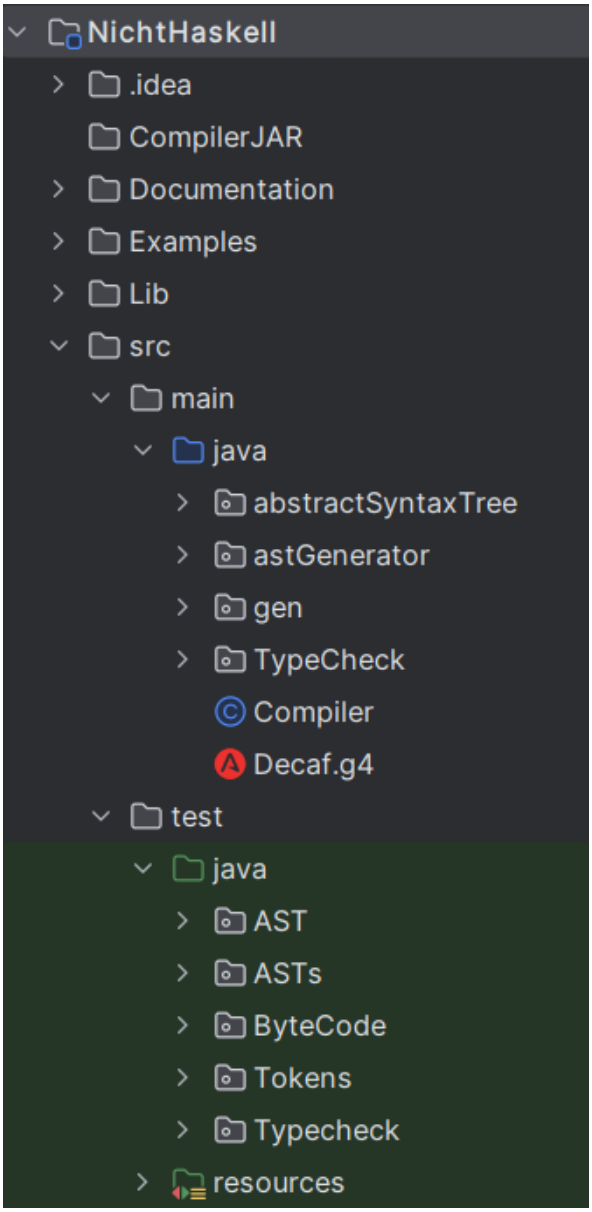
- ASM v9.3

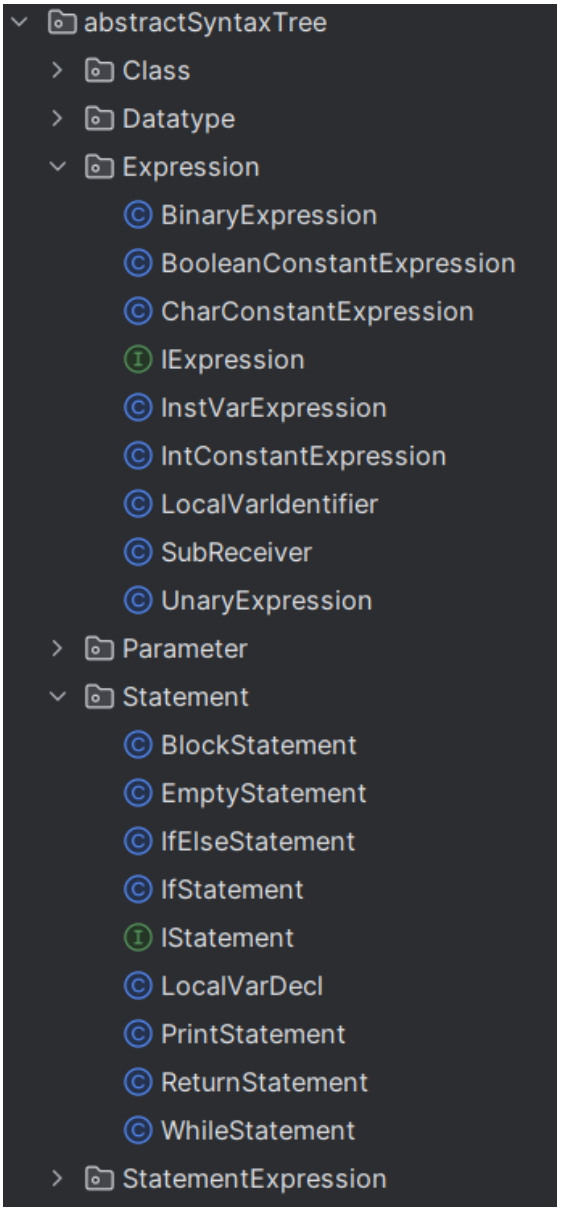
  Wird verwendet, um Bytecode aus dem typisierten Syntaxbaum zu generieren

- JUNIT v4.13.1

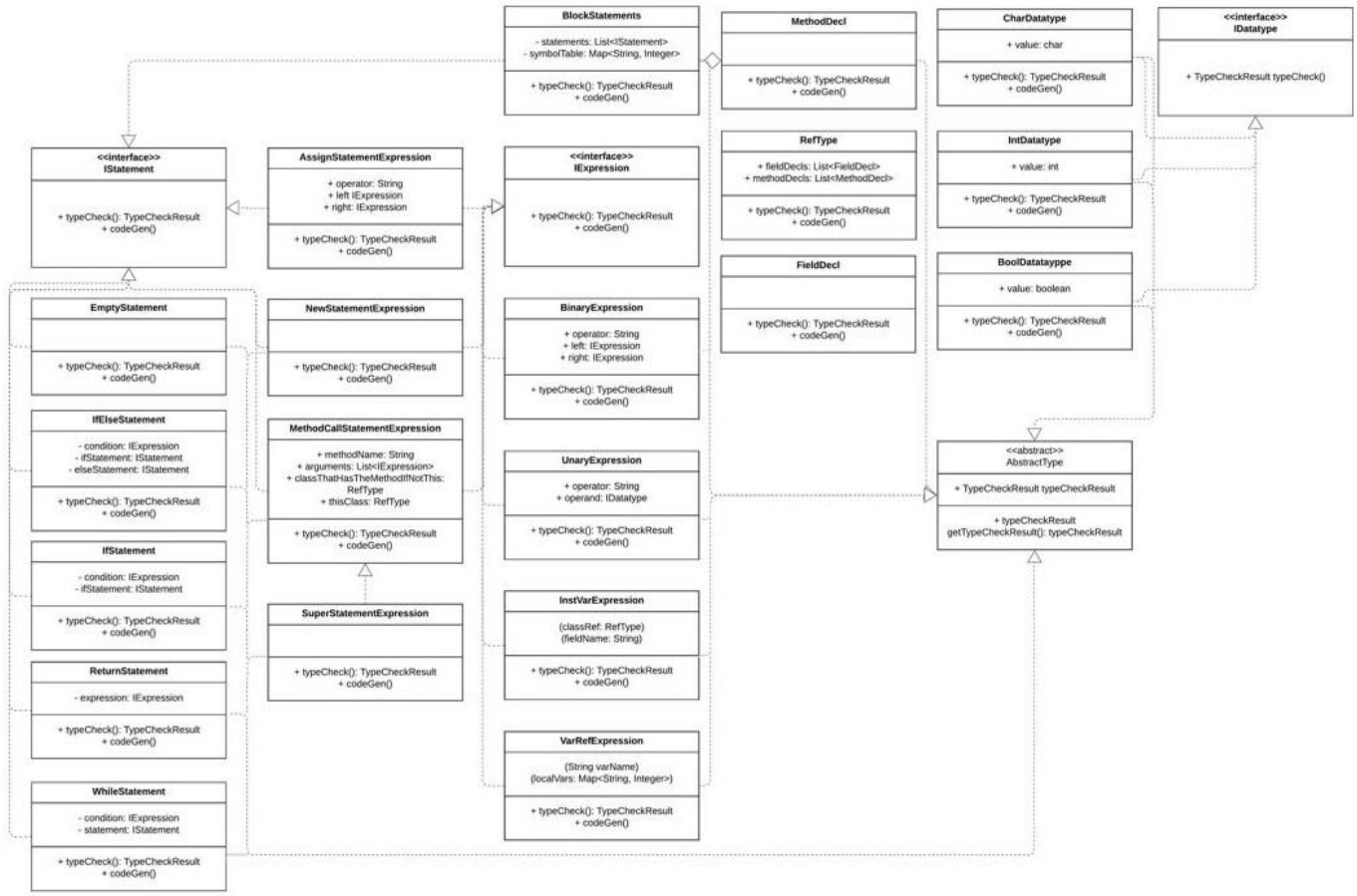
  Wird verwendet, um Tests für den Compiler zu schreiben und auszuführen.

3. Projekt-Struktur:





4. Klassendiagramm für die (T)AST-Struktur:



Das in dieser Dokumentation enthaltene UML-Diagramm stellt eine vereinfachte Version der gesamten Struktur dar, um die grundlegenden Konzepte und Beziehungen übersichtlich darzustellen. Für ein detaillierteres und vollständigeres UML-Diagramm, das alle Klassen, Methoden und ihre Interaktionen umfasst, besuchen Sie bitte das entsprechende Gitea-Repository. Dort finden Sie die vollständige UML-Darstellung, die eine tiefere Einsicht in das gesamte System ermöglicht.

5. Komponenten:

5.1 Parsing + Scanner + Abstrakter Syntaxbaum:

Implementiert von David Müller und Stefan Zimmerer

5.1.1 ANTLR-Grammatik für das JavaSubset:

Für das Parsen und Scannen haben wir ANTLR verwendet, ein leistungsfähiges Tool zur Erstellung von Parsern und Lexer, mit dessen Hilfe wir haben eine JavaSubset-Grammatik erstellt haben. Die Grammatikdatei definiert die Regeln für Tokens (wie Schlüsselwörter, Operatoren und Bezeichner) und die Struktur der Programmiersprache.

5.1.2 Scanner (Lexer):

Der Scanner oder Lexer ist für das Zerlegen des Quellcodes in einzelne Tokens zuständig. Diese Tokens sind die grundlegenden Bausteine der Sprache, wie Schlüsselwörter, Operatoren, Literale und Bezeichner. Die Lexer-Regeln in der ANTLR-Grammatik spezifizieren, wie die Zeichenketten des Quellcodes in diese Tokens umgewandelt werden.

- **Schlüsselwörter**: Reservierte Wörter der Sprache, wie `class`, `public`, `void`, `if`, `else`, etc. werden direkt als spezifische Tokens erkannt. Auch die main
- **Operatoren**: Arithmetische und logische Operatoren, wie +, -, *, /, &&, ||, etc., wurden ebenfalls als eigene Tokens definiert.
- **Literale**: Konstanten wie Ganzzahlen (`IntValue`), Zeichen (`CharValue`), Booleans (`true`, `false`) und `null` werden als Literale erkannt.
- **Bezeichner**: Namen für Variablen, Methoden und Klassen werden als `Identifier` erkannt und folgen bestimmten Regeln (z.B. beginnen mit einem Buchstaben und können Zahlen, $ oder _ enthalten).

5.3 Parser

Der Parser verwendet die Tokens, die vom Scanner erzeugt wurden, um den Quellcode gemäß der definierten Grammatik zu analysieren. Er überprüft, ob die Reihenfolge der Token den Sprachregeln entspricht und erstellt dabei eine Struktur, die als Parsebaum oder Syntaxbaum bekannt ist. Die Parser-Regeln definieren, wie die Tokens zu grammatikalisch korrekten Konstrukten kombiniert werden. Konstrukte der Programmiersprachen und der Aufbau von Deklarationen, Zuweisungen und Ausdrücke werden dabei modelliert.

- **Programmstruktur**: Das Programm besteht aus einer oder mehreren Klassendeklarationen (`classdecl`).
- **Klassendeklaration**: Jede Klasse kann öffentliche Zugriffsmodifizierer (`AccessModifierPublic`), einen Namen (`Identifier`), Konstruktoren (`constuctorDecl`), lokale Variablendeklarationen (`localVarDecl`) und Methoden (`methodDecl`) enthalten. Optional kann eine Hauptmethode (`MainMethodDecl`) mit einem Block (`block`) enthalten sein.
- **Methoden und Konstruktoren**: Methoden können einen Rückgabewert (`type` oder `void`), einen Namen und eine Parameterliste (`parameterList`) haben. Konstruktoren haben ähnliche Strukturen, jedoch ohne Rückgabewert.
- **Ausdrücke und Anweisungen**: Ausdrücke (`expression`) können arithmetische oder logische Operationen, Methodenaufrufe (`methodCall`) oder Zuweisungen (`assign`) umfassen. Anweisungen (`statement`) umfassen Rückgaben (`returnStmt`), Variablendeklarationen, Blöcke, Schleifen (`whileStmt`),

Bedingungsanweisungen (`ifElseStmt`) und Ausdrucksanweisungen (`stmtExpr`).

- **Operatoren und Werte**: Die Grammatik definiert verschiedene Arten von Operatoren, darunter Punktoperatoren (`DotOperator`), Zeilenoperatoren (`LineOperator`), Vergleichsoperatoren (`ComparisonOperator`) und logische Operatoren (`LogicalOpertor`). Werte (`value`) umfassen Ganzzahlen, Booleans, Zeichen und `null`.

5.4 Abstrakter Syntaxbaum (AST):

Der Abstrakte Syntaxbaum (AST) ist eine vereinfachte Darstellung des Parsebaums, die nur die wesentlichen syntaktischen Informationen enthält. Der AST abstrahiert unnötige syntaktische Details und stellt die logische Struktur des Quellcodes dar, was die weitere Verarbeitung erleichtert, da im weiteren Verlauf Details wie beispielsweise ein '=' bei einer Zuweisung uninteressant sind. Man braucht lediglich die Information welcher Wert zu welcher Variable zugeordnet wird.

Der AST wird mithilfe von Java-Klassen realisiert. Jeder innere Knoten des Parsebaums wird durch eine eigene Klasse repräsentiert. Diese Klassen befinden sich im Package "abstractSyntaxTree", welche in weitere Packages unterteilt sind. Ein Beispiel dafür ist das "Statement" Package, das alle Statement-Klassen enthält. Jede Klasse hat Attribute, die die Kindknoten des Parsebaums darstellen. Ein Beispiel hierfür ist die Klasse "RefType", die eine Klasse in Java repräsentiert. Sie enthält die Attribute "name", "fieldDecls" und "methodDecls". Zusätzlich hat es das boolean Attribut "hasMain", das nicht direkt aus der Grammatik kommt, aber für die weitere Verarbeitung relevant ist.

Neben dem Lexer und Parser generiert ANTLR auch die BaseVisitor Klasse, welche für jedes Nichtterminal der Grammatik eine Visitor-Methode erstellt. Jeder dieser Visitors hat einen Kontext des Nichtterminals als Eingabeparameter, der dessen Inhalt enthält. Um den AST zu erstellen, gibt es die Klasse „ASTGenerator", welche von der BaseVisitor Klasse erbt. In ASTGenerator werden die Visitors überschrieben.

Der Start der Grammatik und somit die Wurzel des Parsebaums ist das Nichtterminal "program". Der Visitor dessen hat ein ProgramContext als Eingabeparameter und gibt ein neues Objekt der Klasse "Program" zurück. Da ein Programm in der Grammatik mehrere Klassen enthalten kann, hat die Klasse "Program" eine Liste "classes" als Attribut, die RefType-Objekte enthält. Der ProgramContext enthält deshalb mehrere "ClassDeclContext"-Objekte, die die Inhalte der Klasse enthalten. Im Visitor von "program" wird der Visitor von "classDecl" aufgerufen, um die RefType-Objekte zu erstellen. Und in dem Visitor von "classDecl" werden dann die Visitors der Kindknoten aufgerufen um deren Objekte zu erstellen. Dieser Vorgang wiederholt sich für jeden Knoten bis zum letzten.

Der generierte AST dient als Grundlage für die weiteren Schritte der Codeanalyse und -generierung.

5.2 Semantische Prüfung und typisierter Syntaxbaum

Implementiert von Josefine Krauß:

Die semantische Prüfung in Kombination mit einem typisierten Syntaxbaum (im Folgenden bezeichnet als “TypeCheck”) stellt sicher, dass der Quellcode semantisch korrekt ist. Dabei werden verschiedene Aspekte überprüft, wie die korrekte Deklaration und Verwendung von Variablen, die Ausführung von Operationen zwischen kompatiblen Typen, die Rückgabewerte und Parameter von Methoden sowie die Zuweisung von Ausdrücken zu Variablen. Während dieser Prüfung werden potenzielle Fehler wie unzulässige Typkonvertierungen, undefinierte Variablen und Funktionen, mehrdeutige oder widersprüchliche Typangaben sowie Gültigkeitsbereiche von Variablen identifiziert und behandelt. Die Typprüfung erweitert den Syntaxbaum zu einem typisierten Syntaxbaum (TAST), der zusätzliche Typinformationen in den Knoten enthält. Dieser TAST dient als Grundlage für die nachfolgenden Phasen der Codegenerierung. Diese Phase des TypeChecks hilft dabei, Fehler frühzeitig zu erkennen.

Implementierung

Im ersten Schritt wird die Methode "typeCheck()" auf der Program-Instanz aufgerufen, die den AST enthält. In dieser Methode wird ein Typ-Kontext und ein Methoden-Kontext erstellt.

- Der Typ-Kontext ist eine verschachtelte HashMap, die für jede Klasse alle Feldnamen sowie für jeden Feldnamen den entsprechenden Typ angibt.
- Der Methoden-Kontext ordnet jeder Klasse alle Methodennamen mit ihren Rückgabetypen und Parametern zu.

Anschließend wird die Methode typeCheck(/*...*/ methodContext, /*...*/ typeContext) für jede Klasse aufgerufen.
Diese Methode überprüft, dass Feld- und Methodennamen nicht doppelt vorkommen, und ruft die „typeCheck“-Methode dieser Elemente auf. Die „typeCheck“-Aufrufe, die sich im Call Stack über dem der „typeCheck“-Methode von „MethodDecl“ befinden, nutzen eine Liste der lokalen Variablen.
Die Interfaces aller relevanten Elemente ab dieser Ebene sind „IDatatype“, „IExpression“ und „IStatement“ und beinhalten eine Methode typeCheck(/*...*/ methodContext, /*...*/ typeContext, /*...*/ localVars). In den Implementierungen dieser Methoden liegt ein Großteil der Logik des TypeChecks.

Der TypeCheck wird durch eine wiederholte Aufrufstruktur implementiert, bei der die „typeCheck“-Methode auf die jeweils untergeordneten Elemente des AST angewendet wird. Beispielsweise wird für einen Codeblock der Typ Check auf jedes Statement angewendet. Dadurch wird sichergestellt, dass alle Statement-Elemente auf ihre

Typkorrektheit überprüft werden. Die Methode des „BlockStatement“ stellt dann z. B. sicher, dass alle Ausführungspfaden der richtige Typ zurückgeben.

Alle „typeCheck“-Methoden haben den Rückgabetyp „TypeCheckResult“. Der darin enthaltenen Typname ist ein String.
Das „TypeCheckResult“ wird außerdem in der abstrakten Klasse „AbstractType“ gespeichert, die von allen Klassen mit TypeCheck erweitert wird. Das ermöglicht z. B. den Zugriff durch die Codegenerierung.

Des Weiteren gibt es eine Hilfsklasse TypeCheckHelper. Sie enthält die statischen Methoden „String upperBound“ und „boolean typeExists“.

Wenn der TypeCheck Fehler im AST feststellt, wird eine „TypeCheckException“ geworfen, dessen Text den Fehler benennt. Werden ca. 20 solcher Fehler unterschieden.

Außerdem überprüft der TypeCheck ob es genau eine Main-Methode gibt.

Der TypeCheck erfüllt auch die Aufgabe, zu setzen in welcher Klasse sich das zu prüfende Element befindet. Dies war nicht an frühere Stelle im Kompilierungsprozess möglich. Da die Information jedoch nötig ist, wird der Name der aktuellen Klasse vor den Aufrufen der „typeCheck“-Methoden im entsprechenden Feld der folgenden zu prüfenden Instanz gesetzt.

5.3 Bytecode-Generierung

Implementiert von Jochen Seyfried:

5.3.1 Übersetzung vom Abstrakten Syntaxbaum in Java-Bytecode mit ASM

Die Bytecode-Generierung ist der abschließende Schritt im Compiler, bei dem der Typisierte Abstrakte Syntaxbaum (TAST) in Java-Bytecode übersetzt wird. Dies wird hier mit dem Tool ASM bewerkstelligt, welches die Erstellung und Manipulation von Bytecode ermöglicht. De grundlegende Aufgabe der Bytecodegenerierung ist es, der JVM vorzugeben, wie diese ihren Runtime-Stack manipulieren soll, um das gewünschte Ergebnis zu erhalten, welches vom eingegebenen Code, bzw. dem AST festgelegt wird.

5.3.2 Ablauf der Bytecodegenerierung

Zur Generierung des Bytecodes werden die Knoten des ASTs durchlaufen. Dementsprechend wird mit der Initialisierung der Klassen begonnen. Dafür wird in ASM für jede Klasse ein ClassWriter-Objekt erstellt, welches den Bytecode für die Klasse beinhaltet. Zudem werden hierbei grundlegende Eigenschaften wie Name und Zugriffsmodifikatoren festgelegt.

In den einzelnen Klassen wird nun der Bytecode für die Felder und Methoden generiert, welche dies wiederrum die darin enthaltenen Statements tun. Das wird so lange

fortgeführt, bis alle Äste bis zu den Blättern abgearbeitet und sich die aufgebaute Struktur rückwärts zusammenfügt. Dies ist bei uns über einfache for-Schleifen gelöst:

```
1. for (RefType oneClass : classes) {
2.      oneClass.codeGen(cw, methodContext, typeContext);
3.  }
```

```
1. for (FieldDecl field : fieldDecls) {
2.     field.codeGen(cw);
3. }
```

```
1. for (MethodDecl method : methodDecls) {
2.     method.codeGen(cw, methodContext, typeContext, fieldDecls);
3. }
```

Vereinfachte Darstellung der AST Traversierung

Ein ähnliches Konzept wie das der Klassen gibt es auch für die Methoden, welche in einem MethodVisitor-Objekt gespeichert werden. Dabei wird initial der Kopf der Methode definiert. Die Instanz des MethodVisitor-Objektes muss nun in alle darin liegenden Operationen übergeben werden, um den dort generierten Bytecode in den MethodVisitor speichern zu können.

Hierbei werden oftmals Informationen benötigt, welche nicht auf dem Stack zu finden sind. Dazu gehören die Lokalen Variablen, die Methoden einer Klasse, die aktuelle Klasse, alle verfügbaren Klassen und deren zugehörige Parameter wie Rückgabetyp und Argumente. Diese werden in unserem Projekt in (Linked) Hash Maps gespeichert, welche in die verschiedenen codegen-Methoden übergeben werden.

Die Bytecodegenerierung der einzelnen Knoten erfolgt durch das Pushen von Parametern auf den Stack, dem Laden jener Parameter, dem Nutzen (und gleichzeitigen entfernen) der Parameter in arithmetischen und logischen Operationen und dem Aufrufen von Methoden.

Dabei ist sowohl der Zustand des Stacks vor als auch nach der Operation entscheidend. Sollten dabei inkorrekte Parameter auf dem Stack liegen treten Fehler auf. Dementsprechend muss sichergestellt werden, dass während der Ausführung eines Knotens, welcher selbst Kinder haben kann, die richtigen Werte für Folgeoperationen auf dem Stack zu haben. Andererseits muss der Stack nach Abschluss der Gesamtoperation möglicherweise leer sein, da Folgeoperationen sonst mit falschen Werten arbeiten.

Sind alle Knoten und Blätter durchlaufen und hat sich die Struktur fertig abgebaut, ist man am Ende einer Klasse und befindet sich wieder im ClassWriter-Objekt. Dieses kann nun abgeschlossen und der Bytecode für die Klasse in eine .class-Datei geschrieben werden, welche von der JVM ausgeführt werden kann.

5.4 Testen

Implementiert von Julian Murek:

5.4.1 Tests zur Token Generierung

Um zu überprüfen ob aus dem Java-Quellcode die richtigen Tokens generiert werden wird die klasse “JavaLexerTest” verwendet. Diese Klasse vergleicht die vom Lexer generierten Tokens mit handgemachten Tokens, welche in einer Datei gespeichert wurden.

Die Klasse erwartet bei der Initialisierung ein Basisverzeichnis, in welchem sich die gespeicherten Java-Quelldateien und Token-Dateien befinden, dies dient dazu Schreibarbeit zu verringern und die Refaktorisierung zu erleichtern. Hat man eine “JavaLexerTest” Klasse angelegt, so kann man die “testTokens” Funktion, mit jeweils einem Pfad zu der Java-Quelldatei und Token-Datei als Parameter, aufrufen der Pfad ist dabei relativ zu dem angegebenen Basisverzeichnis.

Diese Methode liest dann den Quellcode ein und generiert, mit dem JavaLexer, die entsprechenden Tokens, diese werden in einer Liste gespeichert. Anschließend werden die erwarteten Tokens aus der angegebenen Token-Datei ausgelesen (zuvor spezifiziert mit dem Token-Pfad als Funktions-Parameter). Das Auslesen geschieht mit einer Schleife, in welcher bei jedem Durchgang der eine Zeile mit einem Token eingelesen wird und mit dem Token aus der vorherig erstellten Liste verglichen wird. Stimmen “Token-Type" oder “Token-Text” nicht miteinander überein, so schlägt ein “assertEqual” fehl und die Methode bricht mit einer Fehlermeldung ab. Diese Fehlermeldung gibt Auskunft an welcher Stelle der Vergleich fehgeschlagen ist, welcher Token erwartet wurde und welcher Token tatsächlich eingelesen wurde.

Auszug aus Token-File:

```
Class: "class"
Identifier: "IfElseIfStatementWithOneReturn"
OpenCurlyBracket: "{"
Int: "int"
Identifier: "foo"
OpenRoundBracket: "("
Int: "int"
Identifier: "i"
ClosedRoundBracket: ")"
OpenCurlyBracket: "{"
Int: "int"
Identifier: "result"
Assign: "="
IntValue: "0"
Semicolon: ";"
If: "if"
OpenRoundBracket: "("
Identifier: "i"
ComparisonOperator: "=="
IntValue: "1"
ClosedRoundBracket: ")"
OpenCurlyBracket: "{"
Identifier: "result"
Assign: "="
IntValue: "10"
Semicolon: ";"
ClosedCurlyBracket: "}"
```

Beispielhafte Fehlermeldung:

```
org.junit.ComparisonFailure: Token mismatch at index 25
Expected :ClosedCurlyBracket: "}"
Actual   :Semicolon: ";"
<Click to see difference>

<1 internal line>
    at Tokens.JavaLexerTest.testTokens(JavaLexerTest.java:45)
    at Tokens.TestAll.testIfElseIfStatementWithOneReturn(TestAll.java:72) <25 internal lines>
```

5.4.2 Tests zur Überprüfung des Abstrakten Syntaxbaums

Um zu überprüfen, ob der generierte Abstrakte-Syntax-Baum/Abstract-Syntax-Tree (AST) mit dem erwarteten AST übereinstimmt, wurde folgender Ansatz gewählt: Bei jeder Klasse des ASTs wurde die “equals” Methode überschrieben. Normalerweise würde diese Methode bei einem Aufruf, mit z.B. zwei ASTs als Parameter überprüfen, ob die zwei übergebenen Objekte tatsächlich identisch sind, nicht ob die lokalen Variablen

die gleichen Werte beinhalten. Die überschriebene Methode ruft allerdings rekursiv die Equals-Methode auf den relevanten Lokalen-Variablen der jeweiligen Klasse auf. Dies geschieht so lange, bis einfache Datentypen erreicht werden, welche bei einem Vergleich “true” oder “false” zurückgeben, die Ergebnisse der Methoden-Aufrufe werden schlussendlich verundet (&&) und zurückgegeben.

Ruft man also zum Beispiel die Equals-Methode auf der “Program” Klasse auf, so ruft diese wiederrum die Equals-Methode auf der Liste von Klassen “RefTypes” auf. In den einzelnen RefTypes wird die gleiche Methode für die lokalen Variablen “name”, “fieldDecls”, “methodDecls” und “hasMain” aufgerufen. Der Aufruf auf “name” und “hasMain” liefert dabei direkt ein Ergebnis zurück (da dies primitive Datentypen sind), während “fieldDecls” und “methodDecls” wiederrum weiter rekursive Aufrufe tätigen. Am Ende werden die Ergebnisse der vier Methodenaufrufe verundet und zurückgegeben.

Um den Testprozess so einfach wie zu möglich zu gestalten wurde eine Helferklasse angelegt, diese nimmt zwei Parameter. Einen Pfad zu einer Java-Quelldatei, aus welcher ein AST generiert wird und einen AST selbst (Program-Klasse). Der AST bzw. Die program-Klasse muss dabei per Hand erstellt werden. Die Helferklasse “AstComparer” generiert aus der Javaquelldatei ein AST und führt auf diesem einen Vergleich mit dem übergebenen AST durch. Es ist auch zu bemerken das in jeder Equals-Methode ausgegeben wird, ob die Methode “true” oder “false” zurückgibt, dies dient zur leichteren Fehlererkennung

5.4.3 Tests zur Überprüfung der Semantikprüfung

Um die Typüberprüfung, des Compilers, zu testen, wurde die Klasse “TypeChecker” angelegt. Diese enthält drei statische Methoden:

- “assertTypeCheckResult” Diese Methode nimmt als Eingabe: einen boolean, welcher angibt ob erwartet wird, dass der Typcheck erfolgreich ist, ein Program welches den generierten AST repräsentiert, einen AST welcher den korrekten und typisierten AST darstellt und einen boolean, ob überprüft werden soll, dass es genau eine Main-Methode gibt. Führt man diese Methode durch schaut diese, ob der Typcheck, auf dem zu testenden AST, fehlschlägt (dabei wird differenziert ob nach einer Main-Methode geprüft werden soll oder nicht) und vergleicht dieses Ergebnis dann mit dem erwarteten Resultat. Daraufhin wird der generierte TypContext sowie der generierte MethodContext mit dem Typ und Method-Context des korrekten typisierten ASTverglichen. Zuletzt gilt es noch die beiden Klassen-Listen der ASTs miteinander zu vergleichen, da der Typcheck bestimmte Variablen innerhalb des ASTs verändert, z.B. den “returnType” des Codeblocks. Des weiteren wurde in die Equals-Methode an Stellen, welche für die Codegenerierung notwendig sind, eingebaut, dass auch die

"TypeCheckResult"-Ergebnisse mit dem typisierten AST verglichen werden (ein Beispiel hierfür wäre die Klasse "MethodCallStatementExpression".

- "assertTypeCheckResult" stellt die zweite (überladene) Methode dar, lässt man den boolean "testForMain" aus, so wird die vorherige "assertTypeCheckResult" Methode mit "true" aufgerufen.

- "assertTypeCheckResultNoMain" ruft "assertTypeCheckResult" mit dem "testForMain" boolean "false" auf.

5.4.4 Typprüfungstests

Es wurden mehrere Tests angelegt, welche überprüfen, ob TypeContext, MethodContext, Variablen und TypeCheckResults korrekt angelegt oder verändert werden, und ob der Typcheck bei diesen semantisch korrekten Programmen auch durchläuft.

5.4.5 Fehlererkennungstests

Es wurden auch Tests angelegt welche überprüfen ob der TypCheck fehlschlägt wenn ein Programm semantisch inkorrekt ist und z.B. einer Integervariable in Bool zuweist. Die korrekte Erstellung des TypCoontext und MethodContext wird hier auch überprüft.

5.4.6 Tests zur Überprüfung der Bytecode-Generierung

Da das Testen des Bytecodes eine eher komplexere Aufgabe ist wurden mehrere Klassen angelegt.

ClassFileLoader

Die ClassFileLoader-Klasse erweitert die ClassLoader Klasse welche verwendet wird um Klassen aus einer Datei zu laden. Der Konstruktor der Klasse nimmt einen Dateipfad als Parameter, welcher auf eine Datei verweist (.class Datei), die den Bytecode der zu ladenden Klasse enthält.

Die Methode "findClass" nimmt nun einen Klassennamen als String und sucht nach dieser Klasse in der, im Konstruktor, angegebenen Datei und gibt bei Erfolg ein "Class<?>" Objekt zurück.

JarFileLoader

"JarFileLoader" ist ebenfalls eine Erweiterung der Klasse "ClassLoader", sie ist das equivalent zu "ClassFileLoader" für ".jar" Dateien. D.h. dass sie die Klassen nicht aus einem ".class" file liest sondern aus einer Jar-Datei. Dies ist hilfreich da beim Testen von abhängigen Klassen nicht mehrere ".class" files geladen werden müssen sondern nur eine Jar-Datei.

CompareByteSyntax

Die Klasse `CompareByteCodeSyntax` bietet Methoden zum Vergleich von zwei Klassen hinsichtlich ihrer Signatur, Methoden, Felder und Annotationen. Ihr Hauptziel ist es, zu überprüfen, ob zwei Klassen die identische Syntax besitzen. Dies wird durch folgende Methoden erreicht:

`haveSameBehavior`: Diese Methode ist der zentrale Einstiegspunkt und führt verschiedene Vergleiche durch. Sie ruft nacheinander Methoden zum Vergleich der Klassensignaturen, Methoden, Felder und Annotationen auf. Gibt eine der Vergleiche `false` zurück, wird die gesamte Methode ebenfalls `false` zurückgeben, was darauf hinweist, dass die beiden Klassen nicht dasselbe Verhalten haben.

`compareClassSignatures`: Diese Methode vergleicht die Interfaces und Superklassen der beiden Klassen.

compareMethods: Diese Methode vergleicht die Methoden der beiden Klassen. Sie überprüft, ob die Anzahl der Methoden in beiden Klassen gleich ist und ob jede Methode in der einen Klasse eine entsprechende Methode in der anderen Klasse hat.

compareMethod: Diese Methode vergleicht zwei Methoden direkt miteinander. Sie prüft, ob Rückgabetypen, Parametertypen, Exceptiontypen und Annotationen der beiden Methoden übereinstimmen.

compareFields: Diese Methode vergleicht die Felder der beiden Klassen. Sie überprüft, ob die Anzahl der Felder in beiden Klassen gleich ist und ob jedes Feld in der einen Klasse ein entsprechendes Feld in der anderen Klasse hat.

compareField: Diese Methode vergleicht zwei Felder direkt miteinander. Sie prüft, ob die Feldtypen und Annotationen der beiden Felder übereinstimmen.

compareAnnotations: Diese Methode vergleicht die Annotationen zweier Elemente (Klassen, Methoden oder Felder). Sie überprüft, ob die Anzahl der Annotationen gleich ist und ob jede Annotation in dem einen Element eine entsprechende Annotation in dem anderen Element hat.

CompareByteCodeBehaviour

Die Klasse CompareByteCodeBehaviour dient dazu, die Verhaltensweise von Methoden zweier Klassen zu vergleichen. Sie enthält zwei Listen (methodArray1 und methodArray2), die Methoden der beiden zu vergleichenden Klassen speichern.

Beim Erstellen einer Instanz der Klasse werden die beiden Listen initialisiert. Es gibt eine Methode clearMethods(), die diese Listen leert, um sicherzustellen, dass keine alten Daten verwendet werden.

Die Methode functionAlreadyAdded() überprüft, ob eine bestimmte Methode bereits in den Listen enthalten ist, um doppelte Einträge zu vermeiden.

Die Methode sortMethods() durchsucht die Methoden beider Klassen nach Übereinstimmungen und fügt diese den Listen hinzu, sofern sie noch nicht vorhanden sind.

Die zentrale Methode compareMethodBehaviour() vergleicht dann das Verhalten der Methoden beider Klassen. Zunächst werden Instanzen der beiden Klassen erstellt. Dann werden die Methoden, die in den Listen gespeichert sind, verglichen, indem sie aufgerufen und ihre Ergebnisse überprüft werden. Dabei werden Beispielargumente verwendet, die anhand der Parameter der Methoden generiert werden. Wenn die Ergebnisse der Methodenaufrufe unterschiedlich sind, wird false zurückgegeben; andernfalls true.

Zusätzlich gibt es Hilfsmethoden wie getTestInputs(), die Testeingaben für die Methoden generieren, und getDefaultArguments(), die Standardargumente für die Konstruktoren der Klassen bereitstellen.

ByteCodeTester

Die Klasse ByteCodeTester enthält Methoden zum Testen und Vergleichen von Bytecode in Java. Sie verwendet und abstrahiert die Klasse CompareByteCodeBehaviour und CompareByteCodeSyntax zum Vergleichen der Syntax und des Verhaltens von Bytecode.

Die Methode testByteCodeFromAst führt Typprüfungen und Codegenerierung auf einem gegebenen abstrakten Syntaxbaum (AST) durch und vergleicht den resultierenden Bytecode mit einer Referenzklasse. Die Methode testByteCodeFromTypedAst überspringt die Typprüfung und generiert direkt den Bytecode aus dem gegebenen AST, um ihn anschließend zu vergleichen. testClassFileFromScratch erstellt den AST aus einer Java-Datei, führt Typprüfung und Codegenerierung durch und vergleicht dann den Bytecode.

Private Methoden wie typeCheckWithoutmain und generateCode führen Typprüfungen und Codegenerierungen aus und fangen mögliche Ausnahmen ab. Die Methode testAST lädt die angegebenen Klassen aus den Bytecode-Dateien und vergleicht sie sowohl syntaktisch als auch im Verhalten.

Die Methode generateAST generiert den AST aus einer gegebenen Java-Datei, indem sie Lexer und Parser verwendet. compareJarFilesFromAST und

compareJarFilesFromScratch erzeugen Bytecode aus einem AST bzw. einer Java-Datei und vergleichen ihn mit Bytecode in JAR-Dateien. compareJarFiles vergleicht die Bytecodes von Klassen in zwei JAR-Dateien, sowohl hinsichtlich der Syntax als auch des Verhaltens.

Zusammengefasst, bietet die Klasse die Möglichkeit Bytecode aus einer Java-Quelldatei, einem AST oder einem getypten AST mittels “CompareByteCodeBehaviour” und “CompareByteCodeSyntax” mit anderem Bytecode aus einer “.class” oder “.jar” zu vergleichen. Alle Vorgänge sind automatisiert und man benötigt als Eingabeparameter lediglich AST oder Pfad zu einer Quelldatei, mindestens einen Namen einer Klasse und eine Bytecode-Datei welche korrekt ist bzw. mit welcher der generierte Bytecode verglichen werden soll.

6. Installation

Das Projekt nutzt Maven 3.9.5 mit Java 22.

Die Main-Methode befindet sich in der Klasse „Compiler“. Diese kann z. B. in der IDE IntelliJ ausgeführt werden. Die passenden CLI-Argumente können in den Run/Debug Configurations gesetzt werden.

Maven generiert in der Phase „ package “ 2 JAR-Dateien:

- NichtHaskell.jar
- NichtHaskellCompiler-jar-with-dependencies.jar

Beide Archive sind in target-Ordner zu finden.
Die zweite dieser Dateien stellt den ausführbaren Compiler dar. Eine aktuelle Version ist im Repository im Ordner „ jar “ zu finden.

Der Compileraufruf erfolgt mit :

```
java -jar NichtHaskellCompiler-jar-with-dependencies.jar <file_path> [--suppress-details]
```

Wenn die Option „--suppress-details“ genutzt wird, dann ist die Ausgabe des Compilers auf dem Terminal weniger ausführlich. Fehler oder ein erfolgreicher Kompilierungsprozess werden immer angegeben.

Kompilierte Ergebnisse werden im aktuelle Arbeitsverzeichnis gespeichert.