Höhere Programmierkonzepte
Praktikum IV

Prof. Dr. Nikolaus Wulff

8. – 17. Dez 2020

In diesem Praktikum stellen Sie die Praktika I – III auf das JDK1.9 Modulsystem um. Ferner entwickeln Sie eine Skriptsprache, so dass Ausdrücke wie $0.5+4.25/7$, $C = A+B$, $f(x,y) = x*y$ oder $y = A*x$ als Zeichenketten in einem String-Array von *Axela* berechnet werden können.

1 Umstellung von *Axela* auf Java Module

```
1  module Axela.Engine {
2    requires transitive Axela.Core;
3    // export the engine package
4    exports de.lab4inf.axela.engine;
5    // uses Axela for our own AxelaEngine test via reflection
6    uses de.lab4inf.axela.core.Axela;
7    // open our implementation to other modules for reflection
8    provides de.lab4inf.axela.core.Axela
9        with de.lab4inf.axela.engine.AxelaEngine;
10 }
```

Listing 1: module-info.java für das Axela.Engine Projekt.

Alles was sie benötigen, um Ihre Projekte auf das Modulsystem umzustellen ist – neben einem JDK ab 1.9 –, eine Datei `module-info.java` analog zu Listing (1), welche im Source Verzeichnis in das unbenannte *default* Paket gehört. In dieser Datei sind alle Im- und Exporte des jeweiligen Moduls hinterlegt. In Zeile 2 wird z.B. vermerkt, dass Axela.Engine von Axela.Core abhängt, selber das *de.lab4inf.axela.engine* Paket für andere lesend öffnet (Zeile 4) und die `Axela` Schnittstelle mit der `AxleaEngine` implementiert (Zeile 8 & 9) und diese dann auch per Reflection gefunden wird.[1] Dies wird z. B. in der statischen `Axela.getEngine()` verwendet und soll spätestens ab jetzt funktionieren und mit einer Testmethode überprüft werden.

[1] Voraussetzung ist AxelaEngine besitzt einen POJO Konstruktor und/oder eine statische `provider` Fabrik Methode.

Sobald Sie in jedem der Teilprojekte eine entsprechende Modulinformation hinterlegt haben werden diese vermutlich in der Eclipse nicht mehr übersetzen. Sie müssen nun sauber den Test- und den Produktivcode hinsichtlich Module- und Classpath trennen. Bislang lagen alle übersetzten Klassen und (Teil)projekte auf dem Classpath, nun gehört der Produktivcode zum Modulepath. Dies muss in den Eclipse Projekt Einstellungen entsprechend hinterlegt werden. Voraussetzung für das Gelingen ist, dass Sie nirgendwo in ihren Projekten zyklische Abhängigkeiten haben, weshalb Sie in den vorherigen Praktika I – III darauf achten sollten.

Hinweis

JUnit selber wird nur für die Tests verwendet und gehört daher nicht als Abhängigkeit in die Modulbeschreibung sondern verbleibt auf dem Classpath – obwohl JUnit selber eine `modul-info` besitzt und somit modular ist. **Produktivcode darf nie eine Abhängigkeit zum Testcode haben!**

2 Symbolisches Rechnen

Beginnen Sie mit diesem zweiten Teil des Praktikums erst nach einer vollständigen Umstellung auf das Java1.9 Modulsystem sowohl in der Eclsipe IDE und dem *gradle* CI/CD git Pipline, damit Sie nicht zwei Probleme gleichzeitig haben!

```
1  statements := statement (';' statement)*
2  statement  := definition | assignment | expr
3
4  definition  := ID '(' exprlist ')' '=' expr    # function definition
5  assignment  := ID '=' expr | vector | matrix   # variable assignment
6
7  vector      := '{' exprlist '}'                # vector definition
8  matrix      := '{' vector (',' vector)* '}'    # matrix definition
9
10 expr        := ID '(' exprlist ')'             # function call
11              | '(' expr ')'                    # expression bracket
12              | expr '^' expr                   # power operation
13              | expr '*'|'/' expr               # multiplicative operation
14              | expr '+'|'-' expr               # additive operation
15              | ID                              # symbolic variable
16              | NUMBER                          # numeric constant
17
18 exprlist    := expr (','expr)*                 # list of expressins
```

Listing 2: Erster EBNF Entwurf der Axela Grammatik.

In Praktikum IV und V werden Sie an der AxelaScriptEngine arbeiten, die es gestattet algebraische Operationen wie Addition oder Multiplikationen auch mittels Symbolen anstatt nur mit reelen Zahlen durchzuführen.

Java enthält seid dem JDK1.6 im Paket *javax.scripting* eine API für eine `ScriptEngine` die auf JavaScript basiert. Allerdings wurde diese im JDK1.11 auf `@Deprecated` gesetzt und im JDK1.15 entfernt. Höchste Zeit also eine eigene AxelaScriptEngine zu implementieren!

Einen Großteil der Arbeit wurde Ihnen bereits abgenommen und Sie erhalten einen Parser sowie eine ScriptEngine zur Verfügung gestellt. Dieser Parser generiert einen abstrakten Syntax Baum (AST) basierend auf der Grammatik (2) und hinterlegt das Ergebnis in entsprechenden Knotenklassen. Um diese Knotenhierarchie travestieren zu können ist eine entsprechende `NodeVistor` Schnittstelle[2] definiert, deren Implementierungen ohne *instanceof* Casts alle Knoten besuchen und geeignete Manipulationen oder Operationen durchführen können. Knoten für die sich ein Besucher nicht interessiert lässt er einfach "links liegen".

Aufgabe

Ihre Aufgabe wird es sein den `ValueVistor` zu vervollständigen, der einfache algebraische Berechnungen erlaubt. Die Implementierung ist mit einem zugehörigen `ValueVisitorTest` zu überprüfen.

```
1  project('Axela.Script') {
2    apply plugin: 'com.intershop.gradle.javacc'
3    javacc {
4      // configuration container for all javacc configurations
5      configs {
6        javaCCVersion = '7.0.10'
7        main {
8          outputDir = file('build/generated-src/javacc')
9          jjtree {
10           inputFile = file('src/main/java/de/lab4inf/axela/script/javacc/
                 JavaccParser.jjt')
11           packageName = 'de.lab4inf.axela.script.javacc'
12         }
13       }
14     }
15   }
16
17     dependencies {
18      api('net.java.dev.javacc:javacc:7.0.10')
19        implementation project(':Axela.Core')
20        implementation project(':Axela.Engine')
21        implementation project(':Axela.Math')
22     }
23 }
```

Listing 3: AxelaScript Dependency für JavaCC in der build.gradle.

[2]Das Besuchermuster: https://de.wikipedia.org/wiki/Besucher_(Entwurfsmuster)

1. Erstellen Sie ein weiteres Teilprojekt *Axela.Script*. Das Projekt benötigt eine Dependency auf JavaCC siehe Listing (3). Binden Sie das Projekt in Ihre *gradle* CI/CD im Git und der Eclipse ein.

2. Die `AxelaScriptEngine` wird als `Axela.Plugin` in der AxelaEngine registriert. Sofern Sie die entsprechenden Plugin-Erweiterungen noch nicht vorgenommen haben finden Sie eine Anleitung im ILIAS Forum zum Thema "Plugin".

3. Implementieren Sie die fehlenden Operationen des `ValueVistor`s. Dieser Besucher gestattet es aus einem geparsten AST einen Wert zu berechnen. Für dieses Praktikum genügt es wenn einfache Ausdrücke mit konkreten Zahlen – also ohne Variablen oder Funktionsaufrufe – bewertet werden können. **Vergessen Sie den `ValueVistorTest` nicht!** Versuchen Sie das zugrundeliegende Muster und die Knotenhierarchie zu verstehen, damit Sie zukünftig in der Lage sind weitere Besucher im Praktikum V zu implementieren.

4. Der getestete `ValueVistor` soll als ein weiterer *Iris* Service für *Axela* registriert werden, was in der Plugin `init` Methode zu geschehen hat.

5. Sobald diese Punkte vollständig abgearbeitet und alle möglichen Übersetzungsfehler und Abhängigkeiten bereinigt sind, da ihre API möglicherweise etwas abweichend zu den Vorgaben aus Praktikum I–III ist, soll der `Praktikum4Test` des *Axela.Client* Projekts als externer Black-Box Test ohne weitere Modifikationen zufallsgenerierte Testfälle parsen und berechnen und Ausgaben wie unten zu sehen erzeugen und natürlich '"grün" sein: D.h. Abtestat!

```
found ScriptEngine: Axela-ScriptEngine  Extension [axela]
found ScriptEngine: Oracle Nashorn  Extension [js]

submit task: 0.75 * -1.25
submit task: -4.91 * 6.01 + -4.97
submit task: 2.21 - -4.76 * 2.08
submit task: 0.75 / -1.25
submit task: -4.78 + -1.81 * -8.54 / 3.50 + -6.00
submit task: ((0.50 + 1.30
submit task: ((0.50 + )) 1.30
submit task: (0.50 + )) (1.30
```

Testausgabe