Übung 1: Grundlagen

TECHNISCHE UNIVERSITÄT DARMSTADT

Software Engineering
WS 2018/19 - Dr. Michael Eichberg

Abgabe

Die Übung wird als *sbt*-Projekt (`http://www.scala-sbt.org`) bereitgestellt. *sbt* kann verwendet werden, um Java (und Scala) Projekte einfach auszuführen und zu testen. Installieren Sie `sbt` auf Ihrem Rechner und stellen Sie sicher, dass ein Java SDK (min. Java 8) installiert ist. Überprüfen Sie, dass der Java Compiler `javac` auf der Kommandozeile ausführbar ist, auch wenn Sie sich nicht im Verzeichnis mit der ausführbaren Datei befinden. Wenn nicht, passen Sie Ihren `PATH` entsprechend an, Hinweise wie dies bei Ihrem Betriebssystem möglich ist, finden Sie im Internet.
IntelliJ erlaubt das Importieren von *sbt*-Projekten direkt, dafür muss allerdings das Scala Plugin installiert sein. Um ein Eclipse-Projekt zu erzeugen, kann `sbt eclipse` im Projektverzeichnis (das Verzeichnis, dass die Datei build.sbt enthält) ausgeführt werden, danach kann das Projekt mit `Datei > Importieren > Vorhandene Projekte in Arbeitsbereich` importiert werden. Wenn Sie eine `main`-Methode geschrieben haben, können Sie Ihr Programm mit `sbt run` ausführen, Tests können Sie unter `src/test/java/` anlegen und mit `sbt test` ausführen.
Das Ausführen des Kommandos `sbt` im Projektverzeichnis startet den interaktiven Modus von *sbt*. Im interaktiven Modus können Kommandos ausgeführt werden, ohne jedes Mal `sbt` eingeben zu müssen.
Um Ihre Lösung abzugeben, melden Sie sich zunächst unter

`https://submission.st.informatik.tu-darmstadt.de/course/se18`

an und erzeugen Sie ein *submission token*. Wenn Sie den interaktiven Modus von *sbt* verwenden, führen Sie dann folgendes Kommando aus:

`submit <ihreTUId> <submissionToken>`

wobei *<ihreTUId>* Ihre eindeutige Identifikationsnummer an der TU Darmstadt (**nicht Ihre Matrikelnummer!**) und *<submissionToken>* das zuvor generierte Token ist. Wenn Sie nicht den interaktiven Modus verwenden, muss das Kommando in Anführungszeichen gesetzt werden, also `sbt "submit <ihreTUId> <submissionToken>"`. Sie können (innerhalb der Abgabefrist) beliebig oft eine Lösung einreichen, allerdings wird nur die zuletzt eingereichte Lösung bewertet. Die letzte Lösung, die ein Gruppenmitglied eingereicht hat, wird zur Bewertung für die ganze Gruppe herangezogen. Koordinieren Sie sich daher in Ihrer Gruppe, wer Ihre gemeinsame Lösung einreicht.
Stellen Sie sicher, dass Sie das zur Verfügung gestellte Template nutzen und die Namen und Signaturen der vorgegeben Klassen und Methoden nicht verändern sowie dass von Ihnen hinzugefügte Klassen und Methoden die geforderten Namen und Signaturen verwenden. Ändern Sie außerdem nichts an der vorgegebenen Datei `build.sbt`. Andernfalls wird das System Ihre Lösung nicht bewerten können. Beachten Sie, dass der Zugriff auf die Abgabeplattform nur im **internen Netz der Universität** möglich ist. Für einen Zugriff von außerhalb benötigen Sie daher eine VPN-Verbindung.

Einführung

In dieser Übung werden Sie eine einfache Bibliothek zum Verarbeiten boolescher Ausdrücke in Postfixnotation[1] implementieren. Ein boolescher Ausdruck in Postfixnotation ist eine Zeichenfolge in der, von Leerzeichen getrennt, die Operatoren `&` (*And*), `|` (*Or*) und `!` (*Not*) sowie Variablen (beliebige sonstige Zeichenketten ohne Leerzeichen, die jedoch nicht mit einem Operatorsymbol beginnen) vorkommen. Dabei stehen die Operatoren nicht wie üblich zwischen (bzw. bei *Not* vor) den Termen, die sie verknüpfen, sondern dahinter. So sind keine Klammern nötig, um eine eindeutige Darstellung zu erhalten. Beispielsweise wird der (in üblicher Infixnotation angegebene) boolesche Ausdruck

```
a & (!b | (c & d))
```

folgendermaßen in Postfixnotation dargestellt:

```
a b ! c d & | &
```

[1] https://de.wikipedia.org/wiki/Umgekehrte_polnische_Notation

Bearbeiten Sie zur Lösung der Übung die folgenden sechs Teilaufgaben:

Problem 1 Klassenhierarchie 2P

Legen Sie im *Package* `ex01` ein *öffentliches Interface* `BooleanExpression` an (die dafür benötigte Datei ist im Template bereits vorhanden). Legen Sie im selben Package die *Klassen* `And`, `Or`, `Not` und `Var` an, die dieses Interface implementieren. Die Klassen `And` und `Or` sollen jeweils zwei, die Klasse `Not` einen Operanden vom Typ `BooleanExpression` verwalten. Die Klasse `Var` verwaltet den Namen einer Variable als `String`. Verwenden Sie dazu *finale* Felder. Auf die Operanden soll mit den *öffentlichen Methoden* `getLeftOp` und `getRightOp` (für `And` und `Or`) und `getOp` (für `Not`) zugegriffen werden können, auf den Namen von Variablen mit der *öffentlichen Methode* `getName`.

Problem 2 Parsing 4P

Um boolesche Ausdrücke einlesen zu können, legen Sie im Interface `BooleanExpression` die *statische Methode* `parseExpression` an, die ein Argument vom Typ `String` entgegennimmt und deren Rückgabewert vom Typ `BooleanExpression` ist. Diese Methode liest das Argument (ein boolescher Ausdruck in Postfixnotation) ein und wandelt es in einen äquivalente Baum bestehend aus den von Ihnen implementierten Klassen um. Um einen booleschen Ausdruck in Postfixnotation umzuwandeln, können Sie, unter Verwendung eines Stapels (*Stack*) von booleschen Ausdrücken, wie folgt vorgehen:

- Beginnen Sie mit der Betrachtung des Strings beim ersten Zeichen
- Wenn das aktuell betrachtete Zeichen ein Leerzeichen ist, gehen Sie zum nächsten Zeichen über.
- Wenn das aktuelle Zeichen ein binärer Operator (`&` oder `|`) ist, nehmen Sie die obersten beiden Ausdrücke vom Stapel und erzeugen Sie ein Objekt vom Typ `And` bzw. `Or` mit diesen beiden Ausdrücken als Teilausdrücke. Legen Sie dieses Objekt auf den Stapel.
- Wenn das aktuelle Zeichen `!` ist, nehmen Sie den obersten Ausdruck vom Stapel und erzeugen Sie ein Objekt vom Typ `Not` mit diesem Ausdruck als Teilausdruck. Legen Sie dieses Objekt auf den Stapel.
- Andernfalls erzeugen Sie ein Objekt vom Typ `Var`, dessen Name der Teilstring zwischen der aktuellen Position und der des nächsten Leerzeichens ist. Legen Sie dieses Objekt auf den Stapel und fahren Sie an der Position des Leerzeichens fort.
- Wenn Sie am Ende des Strings ankommen, geben Sie den Ausdruck auf dem Stapel zurück.
- Wenn nicht genügend Ausdrücke auf dem Stapel liegen, um eine Operation durchzuführen oder wenn Sie am Ende des Strings ankommen und nicht genau ein Ausdruck auf dem Stapel liegt, werfen Sie eine `IllegalArgumentException`.

Problem 3 Ausgabe 2P

Fügen Sie dem Interface `BooleanExpression` eine *virtuelle Methode* `toPostfixString` hinzu, die keine Argumente entgegennimmt und einen String zurückgibt. Implementieren Sie diese Methode in den Implementierungen von `BooleanExpression` so, dass sie den booleschen Ausdruck in einen String in Postfixnotation umwandelt. Der Aufruf dieser Methode auf dem Rückgabewert von `parseExpression` sollte einen String zurückgeben, der mit dem ursprünglich eingelesenen String identisch ist, jedoch keine überflüssigen Leerzeichen enthält (d.h. zwischen zwei Operaden und/oder Operatoren steht genau ein Leerzeichen).

Problem 4 Auswertung 2P

Fügen Sie dem Interface `BooleanExpression` eine *virtuelle Methode* `evaluate` hinzu, die ein Argument vom Typ `Map<String, Boolean>` entgegennimmt und deren Rückgabewert vom Typ `boolean` ist. Implementieren Sie diese Methode in den Implementierungen von `BooleanExpression` so, dass Sie den aktuellen Ausdruck (entsprechend der üblichen Definition der booleschen Algebra) auswertet, wobei für Variablen der Wert der Map entnommen wird.

Problem 5 Vorbereitung für Disjunktive Normalform 1P

Als Vorbereitung für die letzte Teilaufgabe fügen Sie dem Interface `BooleanExpression` eine *Standardmethode* (*default method*) `disjunctiveTerms` hinzu, die keine Argumente entgegennimmt und deren Rückgabewert eine Liste von `BooleanExpression` ist, die den aktuellen Ausdruck in einer Liste zurückgibt. Überschreiben Sie diese Methode in der Klasse `Or` so, dass Sie eine Liste zurückgibt, die die Verkettung der Ergebnisse des Aufrufs von `disjunctiveTerms` auf beiden Teilausdrücken darstellt. Es soll also für die Klassen `And`, `Not` und `Var` eine Liste der Länge 1 zurückgegeben werden, die das aktuelle Objekt enthält, z.B. $\mathrm{dT}(!A) = \mathrm{List}(!A)$. Für die Klasse `Or` soll hingegen eine Liste mit Teilausdrücken, die selbst nicht vom Typ `Or` sind, zurückgegeben werden, z.B. $\mathrm{dT}(((A\&B) \mid C) \mid (D \mid !(E|F))) = \mathrm{List}(A\&B, C, D, !(E|F))$.

Problem 6 Disjunktive Normalform 5P

Fügen Sie dem Interface `BooleanExpression` eine *virtuelle Methode* `toDNF` hinzu, die keine Argumente entgegennimmt und deren Rückgabewert vom Typ `BooleanExpression` ist. Implementieren Sie diese Methode in den Implementierungen von `BooleanExpression` so, dass sie einen zum aktuellen Ausdruck equivalenten Ausdruck zurückgibt, für den gilt:

- Der Teilausdruck in einem `Not` muss ein Objekt vom Typ `Var` sein.
- Die Teilausdrücke in einem `And` dürfen niemals vom Typ `Or` sein.

Dazu können Sie wie folgt vorgehen:

- Für `Var` geben Sie den aktuellen Ausdruck zurück: DNF(a) = a
- Für `Not` prüfen Sie den enthaltenen Teilausdruck:
  - Im Falle von `Var` geben Sie den aktuellen Ausdruck zurück: DNF(!a) =!a
  - Im Falle von `Not` geben Sie das Ergebnis des Aufrufs von `toDNF` auf dem Teilausdruck in diesem zweiten `Not` zurück: DNF(!!A) = DNF(A).
  - Im Falle von `And` und `Or` geben Sie ein neues Objekt vom Typ `Or` bzw. `And` zurück (beachten Sie das vertauschen von `And` und `Or`!), dessen Teilausdrücke Sie erhalten, indem Sie `toDNF` jeweils auf einem neuen Objekt vom Typ `Not` aufrufen, dessen Teilausdruck der jeweilige Teilausdruck des `And` bzw. `Or` ist: DNF(!(A&B)) = DNF(!A)|DNF(!B) sowie DNF(!(A|B)) = DNF(!A)&DNF(!B)
- Für `And` rufen Sie für beide Teilausdrücke `toDNF` auf. Überprüfen Sie, ob eines der beiden Ergebnisse (oder beide) vom Typ `Or` ist. Wenn nicht, geben Sie ein neues Objekt vom Typ `And` mit diesen Teilausdrücken zurück. Ist jedoch wenigstens eines der Ergebnisse vom Type `Or`, rufen Sie auf beiden Ergebnissen `disjunctiveTerms` auf und erzeugen Sie für jede mögliche Kombination eines beliebigen Ausdrucks der ersten Ergebnisliste mit einem beliebigen Ausdruck der zweiten Ergebnisliste (also für jedes Element des Kreuzprodukts beider Listen) ein neues Objekt vom Typ `And`. Erzeugen Sie aus all diesen Objekten einen Baum (oder eine verkettete Liste), indem Sie sie mit neuen Objekten vom Typ `Or` verknüpfen. Geben Sie den Wurzelknoten dieses Baums (bzw. den Anfang der Liste), der vom Typ `Or` ist, zurück. Sei also DNF(A) = $A1|A2|\ldots$ und DNF(B) = $B1|B2|\ldots$, dann ist DNF(A&B) = ($A1$&$B1$)|($A2$&$B1$)|…|($A1$&$B2$)|($A2$&$B2$)|…
- Für `Or` geben Sie ein neues Objekt vom Typ `Or` zurück, dessen Teilausdrücke Sie erhalten, indem Sie `toDNF` auf den beiden Teilausdrücken des aktuellen Ausdrucks aufrufen: DNF(A|B) = DNF(A)|DNF(B)