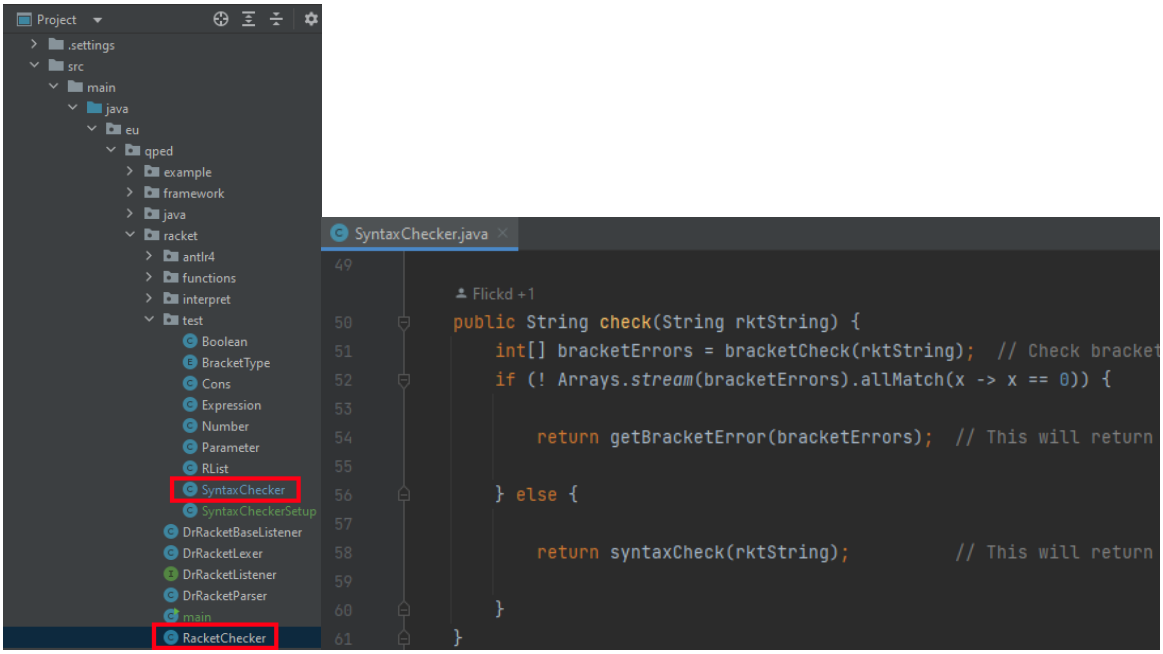
Syntax-Checker Entwickler Notizen

Kurzbeschreibung der Funktionsweise des Syntax-Checkers:

Im Mass-Checker Project wird der Syntax-Check vor innerhalb der Methode RacketChecker aufgerufen. So wird vor der Evaluation des Racket-Programms dieses auf Syntax- und Kompilierungsfehler überprüft, um eine reibungslose Evaluation ohne Fehler in der Evaluationsmethode zu garantieren. Nach einem bestandenen Syntax-Check kann also davon ausgegangen werden, dass das vorliegende Racket-Programm kompilierbar und ohne Syntax-Fehler ist, dabei ist es erstmal egal, was das Racket-Programm am Ende berechnet.

Die beschriebenen Methoden finden sich hier:



Die Methode check innerhalb der Klasse SyntaxChecker kombiniert verschiedene Teilaspekte des SyntaxChecks. Das Prinzip ist dabei immer ähnlich: Die Rückgabe der check-Methoden ist ein String mit einer Fehlermeldung. Wenn das eingegebene Racket-Programm keine Syntax-Fehler enthält, dann ist der zurückgegebene String leer. Wenn allerdings ein Syntax-Fehler im Racket-Programm vorliegt, beispielsweise fehlt eine schließende Klammer, steht eine entsprechende Fehlermeldung im zurückgegebenen String.

BracketCheck:

Diese Methode überprüft den eingegebenen String auf Klammerfehler. Dabei wird die Anzahl schließender und die Anzahl öffnender Klammern miteinander vergleichen. Die Anzahl wird als Integer mit 0 initialisiert und bei einer öffnenden Klammer um +1 erhöht und bei einer schließenden Klammer um -1 verringert. Ist der Counter am Ende positiv, fehlt eine oder mehrere schließende Klammern, ist der Counter negativ, fehlt eine oder mehrere öffnende Klammern. Nur wenn der Counter genau 0 ist, gibt es keinen Klammerfehler. Außerdem werden nur Klammern gezählt, die außerhalb von Anführungszeichen “ liegen. Desweiteren werden die Klammertypen (), [] und {} einzeln überprüft.

Deren Counter werden in dem Rückgabe-Array der Methode BracketChecker hinterlegt, die Methode getBracketError formuliert dann auf der Basis der Counter eine Fehlermeldung.

SyntaxCheck

Der SyntaxCheck orientiert sich an der formalen Sprache der Racket-BSL.

```
     program = def-or-expr ...

 def-or-expr = definition
             | expr
             | test-case
             | library-require

  definition = (define (name variable variable ...) expr)
             | (define name expr)
             | (define name (lambda (variable variable ...) expr))
             | (define-struct name (name ...))

        expr = (name expr expr ...)
             | (cond [expr expr] ... [expr expr])
             | (cond [expr expr] ... [else expr])
             | (if expr expr expr)
             | (and expr expr expr ...)
             | (or expr expr expr ...)
             | name
             | 'name
             | '()
             | number
             | boolean
             | string
             | character
```

Das eingegebene Racket-*programm* muss laut der Sprach-Definition eine *definition* oder *expression* sein. *Test-case* und *library-require* haben wir außenvor gelassen, da diese in dem Modul Deklarative Programmierung keine große Rolle spielen. *Definition* und *expression* werden jeweils noch einmal feiner aufgeteilt. Bei dem SyntaxCheck kann also wie folgt vorgegangen werden: Das eingegebene racket-Programm kann von vorne bis hinten durchlaufen werden. Handelt es sich bei dem ersten erkannten Token um eine öffnende Klammer? Wenn ja, kann das Programm weiterlaufen und die das nächste Token wird betrachtet. Wenn nein, muss es sich um ein Literal handeln. Wenn dies nicht der Fall ist, wurde an dieser Stelle schon ein Syntax-Fehler erkannt. Alle Tokens des Racket-Programms müssen dieser Grammatik folgen und sobald eine Abweichung gefunden wird, wurde dort ein Syntax-Fehler gefunden.

Der Baum der formalen Sprache findet sich somit fast genauso im Code wieder:

```
public String syntaxCheck(String rktString) {...}
```

```
public String defAndExprCheck(Element defOrExpr) {...}
```

```
public String expressionCheck(Element expression) {...}
```

```
public String definitionCheck(Element definition) {...}
```

defAndExprCheck ruft dabei entweder expressionCheck oder definitionCheck auf, und wird selber so oft wie nötig von syntaxCheck aufgerufen.

Die Methode syntaxCheck(String rktString) erhält als Argument einen String, der das gesamte Racket-Programm enthält. Dieser String wird in das XML Format überführt, da so leichter Tokens identifiziert werden können und das Programm als Baum vorliegt, mit dem sehr einfach children und parents erkennbar sind.

Bsp.:

```
String rktString = "(+ 1 1)";
```

```
<?xml version="1.0" encoding="UTF-8"?>
<drracket>
   <paren line="1" type="round">
      <terminal line="1" type="Name" value="+"/>
      <terminal line="1" type="Number" value="1"/>
      <terminal line="1" type="Number" value="1"/>
   </paren>
</drracket>
```

Bei diesem Beispiel handelt es sich offensichtlich um ein gültiges Racket-Programm. syntaxCheck erkennt <drracket> als root-Node, was bedeutet, dass alle Kinder von <drracket> eine *definition* oder *expression* sein müssen. In diesem Fall wird eine „runde“ Klammer gefunden. Sowohl eine *definition*, als auch eine *expression* kann mit einer runden Klammer starten, daher muss nun das erste Kindelement der Klammer betrachtet werden. Dieses ist ein „Name“ mit dem Wert „+“. Das „+“ wird als vordefinierte Funktion wiedererkannt, daher weiß der syntaxChecker nun, dass es sich um eine *expression* handelt, und die Methode expressionCheck wird aufgerufen, mit der runden Klammer als Wurzelelement. expressionCheck überprüft nun, ob die Argumente passen oder andere Fehler innerhalb der Klammer vorliegen. In diesem Fall sind die Argumente 1 und 1 gültig, und es wird keine Fehlermeldung zurückgegeben.