Institut für Softwaretechnik
Fachbereich 4

GReQL-Script

Sprachdefinition und Interpreterimplementation

Studienarbeit
im Studiengang Informatik

vorgelegt von:

Ildar Klassen
ildark@uni-koblenz.de
Matrikelnummer 203210016

Betreut von

Prof. Dr. Jürgen Ebert
Dr. Volker Riediger
Dipl. Inf. Daniel Bildhauer

Oktober 2007

Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Mit der Einstellung dieser Arbeit in die Bibliothek und der Veröffentlichung dieser Arbeit im Internet bin ich einverstanden.

Mendig, den 12. Oktober 2007

Kurzfassung

Im Rahmen dieser Studienarbeit wird eine Scriptsprache entworfen und ein Interpreter für diese implementiert.

Diese Scriptsprache ist der Nachfolger der von Bernt Kullbach entwickelten Scriptsprache *Command Line GReQL* (CLG) [Kul01] und soll die Arbeit mit *TGraphen* und der Graphanfragesprache *G*raph-*Re*pository *Q*uery *L*anguage 2 (GReQL 2), die in der Diplomarbeit vom Katrin Marchewka [Mar06] beschrieben wird, vereinfachen.

Inhaltsverzeichnis

1 Einleitung

In der *GUPRO* Umgebung [EKRW02], die das Analysieren von Programmquelltexten unterstützt, werden *TGraphen* in einem Repository gehalten, an das *GReQL*-Anfragen [Mar06] gestellt werden. Der *GReQL*-Interpreter [Bil06] operiert auf *TGraphen* mit Hilfe von *JGraLab* [Kah06], einer Java-Bibliothek zur Arbeit mit *TGraphen*.

TGraphen sind gerichtete Graphen, deren Knoten und Kanten attributiert und typisiert sein können, siehe Abbildung 1.1.

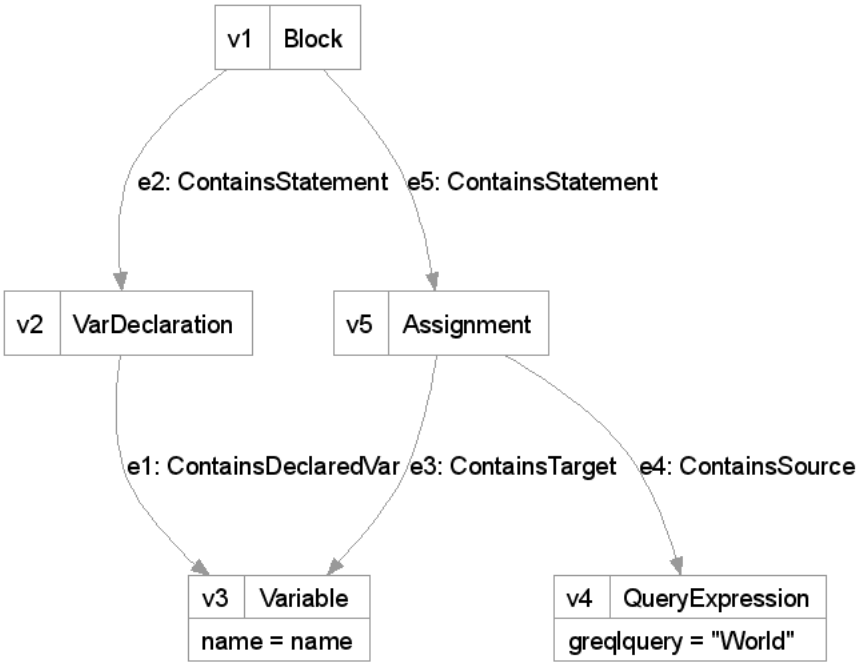

Abbildung 1.1: Beispiel-TGraph

Das Thema dieser Studienarbeit ist eine Scriptsprache, die den Umgang mit *GReQL*-Anfragen erleichtern soll. Als Beispiel liegt *Command Line GReQL* (CLG) [Kul01] vor, das aber nicht die nötige Mächtigkeit hat und auf den älteren Versionen von GraLab in C++ und GReQL1 beruht.

Nach dem Feststellen der Anforderungen wird die Syntax der Sprache definiert. Aus der Syntax wird ein Metamodell des abstrakten Syntaxgraphen hergeleitet. Nachfolgend wird ein Parser entwickelt, der einen abstrakten Syntaxgraph aus dem übergebenem Script generiert. Abschließend wird ein Interpreter implementiert, der den abstrakten Syntaxgraph als GReQL-Script ausführt.

Die Scriptsprache kann mittels *JGraLab*, einer von Steffen Kahle entwickelten Bibliothek zur Arbeit mit *TGraphen*, die in seiner Diplomarbeit [Kah06] beschrieben wird, Graphen erstellen, laden, modifizieren und speichern. Es können GReQL-Anfragen an die benutzten TGraphen gestellt werden. Die Anfrageergebnisse können in weiteren Anfragen benutzt, ausgegeben, gespeichert und geladen werden.

1.1 Notation

Im folgendem Text können folgende Elemente vorkommen, die hervorgehoben sind. Klassennamen sind in CamelCase und in einer Typewriter-Schrift geschrieben. z.B. `SymbolTable`. Dazu zählen auch die Knoten- und Kantennamen im abstrakten Syntaxgraph.

Regeln aus der GReQL-Script-EBNF sind ebenfalls in Typewriter-Schrift geschrieben. Dabei werden Lexer-Regeln und Parser-Regeln unterschieden. Lexer-Regeln sind komplett großgeschrieben, z.B. `GREQLQUERY`. Parser-Regeln sind komplett kleingeschrieben, z.B. `functiondeclaration`.

1.2 Entwicklungsumgebung

GReQL-Script ist in der Programmiersprache Java geschrieben. Für die Entwicklung wurde folgende Software verwendet:

- *GReQLEvaluator* [Bil06] - der GReQL-Anfragen-Auswerter.
- *JGraLab* [Kah06] - die Java-Bibliothek für TGraphen-Manipulation.
- *Enterprise Architect 6.5* (EA)[Ltd07] - UML-Modellierungswerkzeug (wurde als Studentenversion zur Verfügung gestellt).
- *Saxon 8.9* [Sax07] - XSLT-Processor zur Ausführung von `XMI2TGSchema.xsl`, das Teil von *JGraLab* ist. `XMI2TGSchema.xsl` generiert aus einer XMI-Datei, ein Schema für *JGraLab*.
- *Eclipse IDE 3.3* [ecl07a] - Programmier-Entwicklungsumgebung
- *ANTLR 2.7.6* [Pm05] - Parsergenerator
- *TPTP 4.4.0* [ecl07b] mit *JUnit 4* [jun07] - Testumgebung
- *ANT 1.7.0* [Apa06] - Buildtool

2 Anforderungen

In der Arbeitsgruppe Ebert wurde eine Anfragesprache entwickelt, die es erlaubt gezielte Anfragen an ein Graphrepository zu stellen, um Informationen aus diesem zu erhalten. Die Anfragesprache nennt sich GReQL (Graph Repository Query Language). Man könnte sie mit SQL vergleichen, nur dass GReQL speziell für die Arbeit mit Graphen entwickelt wurde. Bei diesen Graphen handelt es sich um TGraphen. Diese Graphen bestehen aus typisierten und attributierten Knoten und Kanten. Um Anfragen zu stellen, die Ergebnisse anderer Anfragen benötigen, wird bisher CLG (Command Line GReQL) benutzt. Die Mächtigkeit von CLG reicht nun nicht mehr aus und soll ausgebaut werden. Es soll eine neue Sprache und der dazugehöriger Interpreter entwickelt werden, die dem gleichen Zweck wie CLG dienen sollen, aber mehr Funktionalität bieten soll.

2.1 Anforderungen an die Sprache

2.1.1 Sprachaufbau

1. Die Sprache muss imperativ sein.
2. Ein GReQL-Script muss ausführbar sein.
3. Ein GReQL-Script muss aus Funktionen bestehen.
4. Scripte müssen Kommentare enthalten können.

2.1.2 GReQL-Anfragen

1. Man muss GReQL-Anfragen ausführen können.
2. GReQL Anfragen sollen direkt im Script oder aus einer Datei ausführbar sein.
3. GReQL-Anfragen müssen graphgebunden oder -ungebunden sein können.
4. GReQL Ergebnisse müssen in weiteren GReQL-Anfragen verwendet werden können.
5. GReQL Ergebnisse sollen in Variablen gespeichert werden können.

2.1.3 Kontrollanweisungen

Der Kontrollfluß soll mit Hilfe von Kontrollanweisungen gesteuert werden.

1. Es muss eine `if`-Anweisung mit einem optionalem `else`-Teil geben.
2. Es muss eine `while`-Schleife geben. (Abweisende Schleife, prüft vor dem Schleifendurchgang)
3. Es muss eine `do-while`-Schleife geben. (Nicht-Abweisende Schleife, prüft nach dem Schleifendurchgang)
4. Es muss eine `foreach`-Schleife geben. (Iteriert durch eine `JValueCollection` durch)
5. Es muss eine `switch`-Anweisung geben mit optionalem `default`-Teil, die nach Variablentyp entscheidet.
6. Innerhalb der Schleifen muss man einen Schleifendurchlauf mit `continue` abbrechen und mit dem nächsten Schleifendurchlauf fortfahren können.
7. Innerhalb der Schleifen muss man die Schleife mit `break` abbrechen und mit der nächsten Anweisung nach der Schleife fortfahren können.
8. Es muss eine `assert`-Anweisung nach dem Java5-Vorbild geben.

2.1.4 Arbeit mit Graphen

1. Man muss Graphen laden können.
2. Man muss Graphen erstellen können.
3. Man muss Graphen speichern können.

2.1.5 Arbeit mit `JValue`-Objekten

1. Man muss `JValue`-Objekte speichern können.
2. Man muss `JValue`-Objekte laden können.
3. Man muss auf die Attribute der attributierten Elemente mit dem Punkt (.) lesend und schreibend zugreifen können. Bsp. g.attr1=g.attr2

2.1.6 Arbeit mit Knoten

1. Man muss Knoten innerhalb eines Graphen erstellen können.
2. Man muss Knoten löschen können.

2.1.7 Arbeit mit Kanten

1. Man muss Kanten zwischen Knoten eines Graphen erstellen können.
2. Man muss Kanten löschen können.
3. Man soll alpha und omega der Kanten ändern können.

2.1.8 Funktionen

1. Man muss eigene Funktionen deklarieren können.
2. Man muss Funktionen anderer Scripte aufrufen können.
3. Es müssen Ausgaben auf der Standardausgabe möglich sein.
4. Es müssen Ausgaben auf der Standardfehlerausgabe möglich sein.
5. Die Parameterübergabe bei Funktionsaufrufen soll per call-by-value und per call-by-reference möglich sein.
6. Funktionen sollen sich selbst aufrufen können (Rekursion).

2.1.9 Sonstiges

1. Dateipfadangaben sollen absolut und relativ möglich sein.

2.2 Anforderungen an den Interpreter

2.2.1 Optionen

1. Es muss möglich sein Optionen an den Interpreter zu übergeben.
2. Die Standardausgabe und die Standardfehlerausgabe müssen durch Optionen auf Dateien umgelenkt werden können.
3. Die Assertions müssen durch Optionen ein- und ausschaltbar sein.

2.2.2 Sonstiges

1. Es könnte eine Fortschrittsmessung/-anzeige vorhanden sein.

3 Sprachbeschreibung

GReQL-Script ist eine *imperative Sprache*, das heißt sie besteht aus Anweisungen. Genauer gesagt besteht sie aus Funktionen, die die Anweisungen enthalten. Eine Anweisung gibt an, welche Aktion das GReQL-Script ausführen soll.

Als Vorbild für die Sprache diente Java, aber es gibt Anweisungen, die nicht Java-ähnlich funktionieren. In der Beschreibung werden Unterschiede oder Ähnlichkeiten zu Java aufgezeigt.

Ein GReQL-Script kann eine `main`-Funktion enthalten. Ähnlich wie bei Java wird diese Funktion beim Aufrufen des Scripts ausgeführt.

Im Folgenden werden die Sprachelemente beschrieben und erläutert.

3.1 Lexikalische Elemente

Ein GReQL-Script kann in folgende atomare Elemente (Token[1]) unterteilt werden.

- Schlüsselwörter
- Bezeichner
- GReQL-Anfragen
- Kommentare und Leerzeichen
- Delimiter und Operatoren.

Aus diesen Elementen und einigen weiteren Zeichen wie Klammern werden andere Strukturen gebildet.

3.1.1 Schlüsselwörter

Schlüsselwörter sind Wörter, die dem Markieren bestimmter Anweisungen oder Klauseln dienen, diese dürfen nicht als Bezeichner verwendet werden. Variablentypen, die im Abschnitt 3.2 behandelt werden, dürfen auch nicht als Bezeichner verwendet werden.

Reservierte Schlüsselwörter in GReQL-Script sind:

[1] Token - die kleinste, Sinn ergebende Einheit

```
assert  , break      , case   , continue   ,
default , do         , else   , foreach    ,
hastype , if         , import , importjava ,
return  , typeswitch , var    , while      .
```

Die Schlüsselwörter werden an dieser Stelle nur erwähnt und erst im Verlaufe der Sprachbeschreibung beschrieben.

3.1.2 Bezeichner

Variablennamen, Funktionsnamen, Schleifennamen (Label3.5.3) und Datentypen sind Bezeichner, damit sie eindeutig identifiziert werden können.

Ein Bezeichner muss mit einem Buchstaben („a“ bis „z“ und „A“ bis „Z“) anfangen, wonach beliebig viele Buchstaben, Ziffern („0“ bis „9“) oder der Unterstrich („_“) folgen können. Schlüsselwörter und Datentypen sind nicht als Bezeichner zugelassen.

Beispiele für gültige Bezeichner:

```
x
CamelCase
eine_variable
xyz123
```

Beispiele für ungültige Bezeichner:

```
_var1
13
$var1
x$
```

3.1.3 GReQL-Anfragen

GReQL-Anfragen können direkt in GReQL-Script vorkommen. Sie müssen mit Backquotes gekennzeichnet sein.

Gültige Beispiele für GReQL-Anfragen:

- `` `1` ``
- `` `false` ``
- `` `"string"` ``
- `` `list [1..10]` ``
- `` `from s:E{Street} with s.name = "Europabrücke" report omega(s) end` ``
  Diese Anfrage benötigt einen Graph.

GReQL-Anfragen werden ausführlicher im Abschnitt 3.4.1 behandelt.

3.1.4 Kommentare

Der Programmcode kann kommentiert werden. Damit kann man Text schreiben, der keinen Einfluss auf die Ausführung hat.

Durch Zeilenkommentare wird der Rest einer Zeile als Kommentar behandelt. Ein Zeilenkommentar beginnt mit zwei Schrägstrichen „//“ und endet am Ende einer Zeile. Durch Blockkommentare werden mehrere Zeilen auskommentiert. Ein Blockkommentar beginnt mit einem Schrägstrich gefolgt von einem Stern „/*“ und endet mit der umgekehrten Folge dieser Zeichen „*/“.

```
//Zeilenkommentar
/*
 * Blockkommentar
 */
```

Die Kommentare sind identisch mit den Java-Kommentaren, mit Ausnahme der JavaDoc-Kommentare, das heißt auch, dass sie nicht schachtelbar sind.

3.1.5 Delimiter und Operatoren

Klammern, Doppelpunkt und Semikolon werden als *Delimiter* (Trennzeichen) bezeichnet und werden dazu verwendet, um die Sprache zu strukturieren.

Es gibt den Zuweisungsoperator (= sh.3.3.2), den Punktoperator(`.`) und den `hastype`-Operator 3.4.3. Mit dem Punktoperator werden mehrere Aufgaben erledigt:

- Zugriff auf Attribute.
- Graphgebundene GReQL-Anfrage
- Graphgebundener Funktionsaufruf von `evaluateQuery(queryfile)`

Es werden keine weiteren Operatoren benötigt, da diese durch GReQL abgedeckt sind.

3.2 Datentypen

GReQL-Script arbeitet mit nur einem Datentyp: `JValue`. Da in GReQL-Anfragen `JValue`-Objekte als Eingabe- und Ausgabeparameter verwendet werden, deckt `JValue` alle nötigen Datentypen ab. Deshalb haben Variablen keine explizite Typangabe, sie sind alle vom Typ `JValue`.

In den `JValue`-Objekten sind aber andere Typen gekapselt. Diese Typen wurden aus JGraLab übernommen und werden hier nicht weiter erläutert. Das sind die Typen aus der `enum JValueType`, die in GReQL-Script verwendet werden können:

```
ATTRIBUTEDELEMENT
ATTRIBUTEDELEMENTCLASS
BOOLEAN
CHARACTER
DOUBLE
EDGE
ENUMVALUE
GRAPH
INTEGER
COLLECTION
RECORD
LONG
OBJECT
PATH
PATHSYSTEM
STRING
VERTEX
```

3.3 Anweisungen

Anweisungen geben an, was ausgeführt werden soll und kontrollieren den Programmablauf. Atomare Anweisungen enden mit einem Semikolon (`;`).

3.3.1 Variablendeklaration

Um eine Variable verwenden zu können, muss diese zuerst deklariert werden. Eine Variablendeklaration wird mit dem Schlüsselwort `var` markiert, gefolgt von einer Bezeichnerliste.

```
var x;
var y,z;
```

Die Sichtbarkeit der Variablen wird im Abschnitt 3.3.3 erläutert.

3.3.2 Zuweisung

Mit einer Zuweisung kann man einer Variable einen Wert zuweisen. Z.B. `x='13';` oder `x=y;`. Man muss die Variable vorher deklarieren, um sie verwenden zu können.

3.3.3 Block

Um mehrere Anweisungen zusammen zu fassen benutzt man Blöcke. Die Anweisungen werden von geschweiften Klammern eingeschlossen.

```
{
  var x,y;
  x=`1`;
  y=x;
}
```

Man kann Blöcke ineinander schachteln. Ein Block definiert auch die Sichtbarkeit der Variablen. Eine Variable ist im eigenen Block und in den darin enthaltenen Blöcken ab der Stelle, wo sie deklariert wurde, sichtbar. Wenn eine Variable zum ersten Mal verwendet wird, hat sie den Wert `null`.

```
{
  var x;
  x=`1`;
  {
    var y;
    y=x;
  }
  {
    var y;   //ist erlaubt, da kein y im "Vater"-Block
    var x;   //Fehler, da x bereits deklariert
  }
}
```

3.3.4 Zusicherungen

Zusicherungen verwendet man um sicher zu stellen, dass bestimmte Bedingungen erfüllt sind. Wenn eine Bedingung nicht erfüllt ist, wird die Ausführung abgebrochen. Damit man weiß, an welcher Stelle das Script gescheitert ist, kann man einen Ausdruck angeben, der bei Nichterfüllung ausgegeben wird. Zusicherungen können bei der Ausführung auch ausgeschaltet werden.

```
assert <Condition>;
```

oder

```
assert <Condition> : <Expression>;
```

Beispiel:

```
x=`2*2*2*2`;
assert `x=16` : `"x ist nicht 16"`;
//...
```

3.4 Ausdrücke

Überall wo eine Anweisung verwendet werden darf, darf auch ein Ausdruck verwendet werden. Also ist ein Ausdruck auch eine Anweisung. GReQL-Anfragen, Variablen und Funktionsaufrufe sind Ausdrücke.

Ausdrücke werden dort verwendet, wo Werte zugewiesen, abgefragt oder ausgegeben werden.

3.4.1 GReQL-Anfragen

GReQL-Anfragen sind die wichtigsten Ausdrücke in GReQL-Script. Mit Ihnen konstruiert man Werte, führt Operationen aus oder stellt Anfragen an Graphen. GReQL-Anfragen können sich über mehrere Zeilen erstrecken.

GReQL-Anfragen werden in Backquotes (`) notiert. Wenn die Anfrage einen Graphen benötigt, wird sie hinter einer Graphvariablen mit dem Punktoperator notiert. Einen Graph kann man mittels der Funktion `loadGraph()` laden. Die Funktionen werden im späteren Abschnitt erläutert.

```
x = `1`;   //Graph-lose Anfragen
g = loadGraphFromTG(`"graph1.tg"`);
   //Graph-gebundene Anfragen
omegaeb = g.`from s:E{Street}
             with s.name = "Europabrücke"
             report omega(s)
             end`;
```

Wenn eine Anfrage sehr groß ist, kann man sie in eine eigene Datei auslagern und mit der Funktion `evaluateQuery()` ausführen.

```
g = loadGraphFromTG(`"graph1.tg"`);
omegaeb = g.evaluateQuery(`"omegaeb.greql"`);
```

Die in GReQL-Script verwendeten GReQL-Anfragen dürfen keine `using`-Klausel verwenden, sonst wird eine Ausnahme geworfen.

3.4.2 Variablen und Attribute

Variablen und Attribute sind einfache Ausdrücke. In den GReQL-Anfragen kann man Variablen aus dem Script direkt verwenden.

```
var1 = `1`;
var2 = `var1 + 1`; //direkte Verwendung
```

Bei den attributierten Variablen kann man auf die Attribute lesend und schreiben zugreifen. Dies geschieht mit dem Punktoperator.

```
g = loadGraphFromTG(`"graph1.tg"`);
cpbag = g.`from cp:V{CarPark}
             with cp.name = "Löhrcenter"
             report cp
             end`;
cploehr = `toList(cpbag)[0]`;
temp = cploehr.capacity;  // Lesezugriff
cploehr.capacity = `999`; // Schreibzugriff
```

Wenn die Variable kein solches Attribut besitzt, wird das Script mit einem Fehler abgebrochen.

3.4.3 Typabfrage

Mit dem Schlüsselwort `hastype` kann geprüft werden, ob eine Variable einen bestimmten Typ hat. Diese Abfrage gibt einen booleschen Wert zurück. Sie gibt `true` zurück, wenn der Typ passt. Das heißt, dass `variable hastype ATTRIBUTEDELEMENT` auch `true` zurückgibt, wenn die Variable `variable` vom Typ `VERTEX` ist. Sie gibt `false` zurück, wenn der Typ nicht passt oder die Variable den Wert `null` hat.

```
if(value hastype LONG)
    value = `value+1`;
```

3.4.4 Funktionsaufrufe

Die letzte Art von Ausdrücken ist der Funktionsaufruf. Man benutzt Funktionen, indem man den Funktionsnamen und die Parameter angibt. Manche vordefinierten Funktionen werden mit dem Punktoperator hinter ein Ausdruck gesetzt, was nur bei bestimmten Rückgabetypen des Ausdrucks erlaubt ist. Alle Funktionen haben einen `JValue`-Rückgabetyp. Das bedeutet, man kann den Rückgabewert immer einer Variablen zuweisen, was natürlich nicht zwingend ist. Die Benutzung im Script sieht so aus:

```
    // eine statische Funktion
graph1 = loadGraphFromTG(`"graph1.tg"`);
    //eine Funktion auf einem Graph
vertex1 = createVertex(graph1, `"example.schema.CarPark"`);
vertex1.name = `"Uniparkplatz"`;
vertex1.capacity = `2000`;
```

3.4.5 print und errprint

Um Ausgaben auf dem Bildschirm zu erzeugen benutzt man die Funktionen `print()` und `println()`. Beide haben einen Eingabeparameter, den sie auf der Standardausgabe ausgeben. `println()` fügt noch einen Zeilenvorschub hinten an. Um die Ausgabe auf der Standarfehlerdausgabe auszugeben, benutzt man die Funktionen `errprint()` und `errprintln()`.

```
print(`"test"`);
//...
if(x){
  errprintln(`"schwerer Fehler!"`)
  return `false`;
}
```

Die vollständige Liste der vordefinierten Funktionen ist im Anhang A Funktionsreferenz zu finden.

3.5 Kontrollanweisungen

Die Kontrollanweisungen dienen der Kontrolle des Programmflusses. Bei den folgenden Anweisungen wird oft ein boolescher Ausdruck auf seinen Wahrheitswert überprüft. Dieser kann nicht nur `true`(wahr) oder `false`(falsch) sein, sondern auch `null`(nicht definiert). Wenn der Wert des Ausdrucks `null` ist, wird die Ausführung des Scripts mit einem Fehler unterbrochen.

3.5.1 Die `if`-Anweisung (Bedingte Ausführung)

Bei der `if`-Anweisung wird ein Teil nur dann ausgeführt, wenn eine Bedingung wahr ist. Wenn diese Bedingung falsch ist, wird entweder nach der `if`-Anweisung die Ausführung fortgeführt oder, wenn vorhanden, ein optionaler `else`-Teil ausgeführt. Die Bedingung muss ein boolescher Ausdruck sein.

Die `if`-Anweisung wird mit einem `if`-Schlüsselwort eingeleitet, danach folgt ein in runden Klammern eingeschlossener Ausdruck, der die Bedingung dar stellt. Nach den Klammern folgt dann eine Anweisung. Nach diesem Teil darf ein `else`-Teil folgen, der aus dem Schlüsselwort `else` und einer Anweisung besteht.

```
if(`b>10`)    //Bedingte
  b=`10`;     //Ausführung

if(`c<0`)     //Bedingte
  c=`-1`;     //Ausführung
else          //mit
  c=`1`;      //sonst-Teil
```

Bei der Schachtelung von `if`-Anweisungen sollte man darauf achten, dass der `else`-Teil stets zur letzt möglichen `if`-Anweisung gehört.

```
if(`a>0`) //1.if
  if(`a<10`) //2.if
    println(`"0<a<10"`);
  else // dieses else gehört zum 2.if
    println(`"a>=10"`);
```

Um Irrtümer zu vermeiden kann man Blöcke benutzen.

```
if(`a>0`){ //1.if
  if(`a<10`){ //2.if
    println(`"0<a<10"`);
  }
  else{ // dieses else gehört zum 2.if
    println(`"a>=10"`);
  }
}
```

3.5.2 Die *typeswitch*-Anweisung (Datentypunterscheidung)

Bei einer Datentypunterscheidung kann man abhängig vom Typ der Variablen verschiedene Anweisungen ausführen lassen. Bei der Variablen wird der Typ mittels der Funktion `getType()` überprüft und die passenden Fälle ausgeführt. Für jeden Fall (`case`) kann eine Liste mit Variablentypen angegeben werden. Ein Variablentyp darf nur in einer dieser Listen vorkommen. Es kann auch ein `default`-Fall angegeben werden. Bei jedem Fall wird geprüft, ob der Variablentyp sich in der Liste befindet, und wenn ja, dann wird dieser und die nachfolgenden Fälle, inklusive dem `default`-Fall, ausgeführt. Wenn keiner der Fälle eintritt, wird nur der `default`-Fall ausgeführt. Innnerhalb einer `typeswitch`-Anweisung darf die `break`-Anweisung benutzt werden. Mit der `break`-Anweisung kann man die `typeswitch`-Anweisung vorzeitig verlassen. Die `typeswitch`-Anweisung verhält sich wie die `switch`-Anweisung in Java, mit dem Unterschied, dass beim `case`-Teil eine Typliste angegeben wird.

Die Datentypunterscheidung beginnt mit dem Schlüsselwort `typeswitch`, gefolgt von einem booleschen Ausdruck in runden Klammern und danach dem `case`-Block. Der `case`-Block besteht aus mindestens einem `case`-Teil und einem optionalen `default`-Teil. Der `case`-Teil beginnt mit dem Schlüsselwort `case`, wonach eine Liste aus `JValueTypes` folgt, dann folgt hinter einem Doppelpunkt (`:`) die Anweisung oder der Anweisungsblock. Der `default`-Teil besteht aus dem Schlüsselwort `default`, dem Doppelpunkt und der Anweisung.

```
variable = evaluateQuery(`"queryfile.greql"`);
  //unbekannter typ
typeswitch(variable)
{
  case DOUBLE,
       LONG    : println(`"variable ist eine 8-byte Zahl."`);
  case INTEGER : println(`"variable ist eine Zahl."`); //wird auch bei
     LONG und DOUBLE ausgeführt
  default      : println(variable); //wird bei allen ausgeführt
}
```

3.5.3 Schleifen

Mit Schleifen kann man gleiche oder ähnliche Anweisungen mehrmals hintereinander wiederholen. In einigen Fällen will man eine Schleife vorzeitig verlassen, für solche Fälle gibt es `break` und `continue`.

While-Schleife

Die `while`-Schleife wiederholt eine Anweisung oder einen Anweisungsblock solange, wie eine Bedingung wahr ist. Die `while`-Schleife besteht aus dem Schlüsselwort `while`, einem von runden Klammern umschlossenen booleschen Ausdruck und einer Anweisung oder einem Anweisungsblock.

```
while(`b>10`)
  b=`b-1`;
```

Die `while`-Schleife prüft den Ausdruck, bevor ein Schleifendurchlauf ausgeführt wird. Die `do-while`-Schleife dagegen prüft den Ausdruck, nachdem ein Schleifendurchlauf ausgeführt wurde.

Die `do-while`-Schleife besteht aus dem Schlüsselwort `do`, einer Anweisung oder einem Anweisungsblock, dem Schlüsselwort `while` gefolgt von einem in runden Klammern umschlossenen booleschen Ausdruck. Die `do-while`-Schleife muss mit einem Semikolon abgeschlossen sein.

```
do
  b=`b-1`;
while(`b>10`);
```

Foreach-Schleife

Mit der `foreach`-Schleife kann man eine gezielte Anzahl von Schleifendurchläufen erzielen oder eine `JValueCollection` durchlaufen.

Die `foreach`-Schleife beginnt mit dem Schlüsselwort `foreach`. Danach folgt in Klammern eine Variable und nach einem Doppelpunkt ein Ausdruck, der eine `JValueCollection` sein muss. Dieser Variablen wird bei jedem Schleifendurchlauf ein Element dieser `JValueCollection` zugewiesen. Danach folgt die Anweisung oder der Anweisungsblock, die oder der ausgeführt werden soll. Es gibt also für jedes Element der `JValueCollection` einen Schleifendurchlauf.

```
//2 aus 10 Kombinationen
//i1 nimmt bei jedem Durchlauf eine Zahl von 1 bis 10 ein
foreach(i1: `list[1..10]`)
  foreach(i2:`list[i1..10]`){
    print(i1);
    print(`","`);
    println(i2);
  }
```

Schleifenbezeichner (label)

Wenn man mit mehreren geschachtelten Schleifen arbeitet, ist es manchmal nötig die Schleifen zu identifizieren. Dafür muss man vor die Schleife einen Bezeichner und einen Doppelpunkt setzen.

```
loop1:
foreach(i1: `list[1..10]`)
  loop2:
  foreach(i2:`list[i1..10]`){
    print(i1);
```

```
    print(`",");
    println(i2);
  }
```

break-Anweisung

Die `break`-Anweisung darf nur innerhalb von Schleifen und in `typeswitch`-Anweisungen verwendet werden. Die `break`-Anweisung bricht eine Schleife, bzw. eine `typeswitch`-Anweisung, komplett ab. Die `break`-Anweisung kann den Bezeichner der Schleife enthalten, die abgebrochen werden soll. Wenn kein Schleifenbezeichner angegeben ist wird die innerste Schleife verlassen.

```
loop1:
foreach(i1: `list[1..4]`)
  loop2:
  foreach(i2:`list[1..4]`){
    if(`i1=i2`) break; //verlässt loop2
    if(`i2=3`) break loop1; //verlässt loop1
    print(i1);
    print(`",");
    println(i2);
  }
```

continue-Anweisung

Die `continue`-Anweisung darf nur innerhalb von Schleifen verwendet werden. Die `continue`-Anweisung bricht einen Schleifendurchlauf ab. Die `continue`-Anweisung kann den Bezeichner der Schleife enthalten, deren Schleifendurchlauf abgebrochen werden soll. Wenn kein Schleifenbezeichner angegeben ist wird der Schleifendurchlauf der innersten Schleife abgebrochen.

```
loop1:
foreach(i1: `list[1..4]`)
  loop2:
  foreach(i2:`list[1..4]`){
    if(`i1=i2`) continue;
        //nächster Durchlauf von loop2
    if(`i2=3`) continue loop1;
        //nächster Durchlauf von loop1
    print(i1);
    print(`",");
    println(i2);
  }
```

3.6 Funktionen

In GReQL-Script kann man vordefinierte Funktionen benutzen und eigene Funktionen deklarieren. Die Benutzung von Funktionen wurde im Abschnitt 3.4.4 gezeigt. Ein Script muss die Funktion `main` enthalten, um ausführbar zu sein. Ein ausführbares Script mit einer `main`-Funktion kann so aussehen:

```
main(args){
  graph1 = loadGraphFromTG(`"graph1.tg"`);
  vertex1 = createVertex(graph1, `"example.schema.CarPark"`);
  vertex1.name = `"Uniparkplatz"`;
  vertex1.capacity = `2000`;
  saveGraphToTG(graph1, `"graph1.tg"`);
}
```

Eine Funktionsdeklaration besteht aus dem Funktionsnamen, den Parametern und dem Funktionsrumpf. Der Funktionsname stellt den Bezeichner dar, über den die Funktion aufgerufen werden kann. In der Parameterliste sind die verwendeten Parameter aufgelistet. Steht vor dem Parameternamen das Schlüsselwort `var`, so erfolgt die Parameterübergabe (wird im Abschnitt 3.6.2 erläutert) mit call-by-reference, andernfalls erfolgt die Parameterübergabe mit call-by-value. Der Funktionsrumpf enthält die Anweisungen, die ausgeführt werden. Darin kann auch der Aufruf der Funktion selbst vorkommen (Rekursion).

Beispiel für eine Funktionsdeklaration:

```
fak(y){
        if(`y<=0`)
                return `0`;
        if(`y=1`)
                return `1`;
        var f;
        f=fak(`y-1`);
        return `f*y`;
}
```

3.6.1 *return*-Anweisung

Alle Funktionen in GReQL-Script haben einen Rückgabewert. Mit der `return`-Anweisung gibt man an, was die Funktion zurückgeben soll. Der Code in der Funktion hinter einer `return`-Anweisung ist unerreichbar. Nach der `return`-Anweisung muss ein Ausdruck folgen, dessen Wert zurückgeben wird. Da nicht immer ein Rückgabewert sinnvoll ist, ist die `return`-Anweisung nicht zwingend. Wenn eine Funktion keine `return`-Anweisung hat, wird beim Verlassen der Funktion `null` zurück gegeben.

3.6.2 Parameterübergabe

Die Parameterübergabe in GReQL-Script kann auf zwei verschiedene Arten geschehen: call-by-value und call-by-reference.

call-by-value

In Java wird für die Parameterübergabe call-by-value verwendet. Beim Aufruf der Funktion wird eine flache Kopie des Objektes erstellt und an die Funktion übergeben. Damit wird ein Ändern des Wertes verhindert.

In GreQL-Script wird bei call-by-value nur ein Parameternamen angegeben.

```
main(args){
  var1=‘1‘;
  methodCBV(var1);
  println(var1); //gibt 1 aus
}
methodCBV(arg){ //Deklaration mit call-by-value
  arg=‘100‘;
  println(arg); //gibt 100 aus
}
```

call-by-reference

Falls man mehrere Rückgabewerte haben will, kann man call-by-reference benutzen. Beim Aufruf der Funktion wird eine Referenz auf das Objekt an die Funktion übergeben. Damit hat man vollen Zugriff auf die Variable.

Bei call-by-reference wird das Schlüsselwort `var` vor dem Parameternamen angegeben.

```
main(args){
  var1=‘1‘;
  methodCBR(var1);
  println(var1); //gibt 100 aus
}
methodCBR(var arg){ //Deklaration mit call-by-reference
  arg=‘100‘;
  println(arg); //gibt 100 aus
}
```

3.6.3 *import*-Klausel

Es gibt zwei Arten von import-Klauseln in GReQL-Script: `import` und `importjava`.

In der `import`-Klausel gibt man an, welche Scripte importiert werden sollen. Man gibt den Pfad des Scriptes als String an. Dieser Pfad kann absolut oder relativ zum eigenen Verzeichnis

sein. Man kann die Funktionen der importierten Scripte, wie die eigenen Funktionen benutzen. Wenn das Script über Funktionen, die die gleichen Signaturen wie Funktionen aus bereits geladenen Scripten haben, verfügt, dann gibt es eine Fehlermeldung und die Ausführung wird abgebrochen. Nach dem Importieren eines Scriptes, kann man die darin enthaltenen Funktionen verwenden, als wären sie in diesem Script deklariert. Solche Scripte müssen keine `main`-Funktion haben.

Mit der `importjava`-Klausel werden bestimmte Funktionen, der angegebenen Java-Klasse, importiert. Nach dem `importjava`-Schlüsselwort muss man einen Klassennamen als GReQL-String-Expression angeben. Es werden alle Java-Methoden importiert, die folgende Eigenschaften haben:

- der Rückgabetyp muss `JValue` sein
- alle Parametertypen müssen den Typ `JValue` haben
- die Methode muss als `public` deklariert sein
- die Methode muss als `static` deklariert sein

Die importierten Java-Methoden können wie GReQL-Script-Funktionen verwendet werden. Standardmässig wird bei jedem Script `greqlscript.lib.Standard` implizit importiert.

```
//scriptimport
import `"standardscriptfunktionen.greqlscript"`;
//javaimport
//hier explizit importiert
importjava `"greqlscript.lib.Standard"`;
main(args){
        var x,sqrt;
  x = `64`;
  // sqrt() sollte in
  // standardscriptfunktionen.greqlscript definiert sein
  sqrt = sqrt(x);
  //println is eine Java-Methode
  //aus greqlscript.lib.Standard
  println(sqrt);
}
```

4 Parser

Die Aufgabe des Parsers ist es, den Abstrakten Syntaxgraph zu erstellen.

Der Parser wird vom Parsergenerator ANTLR [Pm05] aus den Grammatikregeln generiert und besteht aus dem *Lexer*, in mancher Literatur auch Scanner genannt, und dem eigentlichen *Parser*.

An den Lexer wird eine Zeichenkette oder eine Datei übergeben. Daraus liest der Lexer die einzelnen Zeichen und fasst sie zu Token zusammen. Die Token helfen von den Zeichen zu abstrahieren. Das Gegenteil wäre wenn jeder Token immer aus nur einem Zeichen bestehen würde. Dann spricht man nur vom Parser, da man den Lexer nicht unbedingt braucht.

Der Parser versucht mittels der Produktionsregeln, Token zu größeren Fragmenten zusammenzufassen, bis er ein Startsymbol, das Script, erkennt. Die erkannte Information kann in einem Syntaxbaum gespeichert werden. Überflüssige Informationen, z.B. Semikolon oder Schlüsselwörter, die für die Weiterverarbeitung nicht benötigt werden, werden weggelassen. Dann spricht man nicht von (exakten) Syntaxbaum sondern vom abstrakten Syntaxbaum.

Elemente, die identisch sind, aber an mehreren Stellen vorkommen können zusammengefasst werden. In der Abbildung 4.1 sieht man einen abstrakten Syntaxbaum und den dazu äquivalenten abstrakten Syntaxgraph.

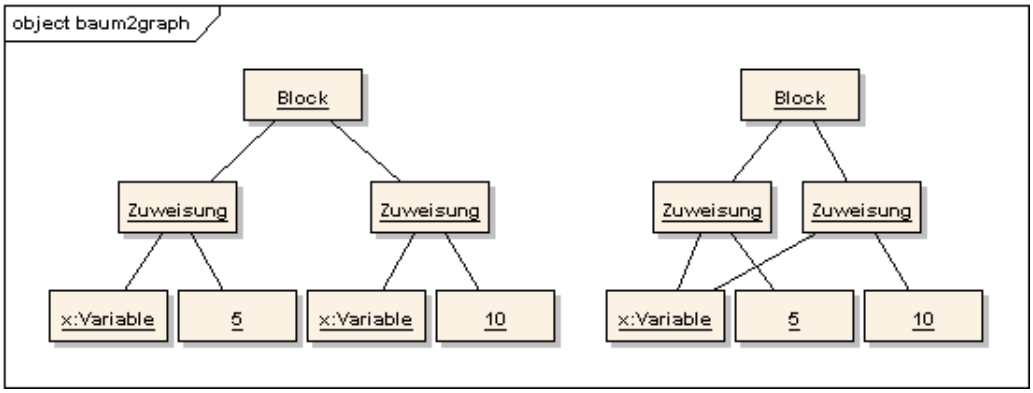

Abbildung 4.1: Abstrakter Syntaxbaum (links), Abstrakter Syntaxgraph (rechts)

4.1 Architektur

GReQLScript besitzt eine Pipe-Filter-Architektur. Dabei wird die Eingabe aus einer Datei oder einer Zeichenkette an den Lexer übergeben, der die einzelnen Zeichen in `Token` zusam-

menfasst und an den Parser weitergibt. Der Parser erzeugt den ASG und reicht ihn an den Interpreter weiter. Der Interpreter interpretiert den ASG und macht Ausgaben, die umgelenkt werden können.

Dies wird in der Abbildung 4.2 verdeutlicht.

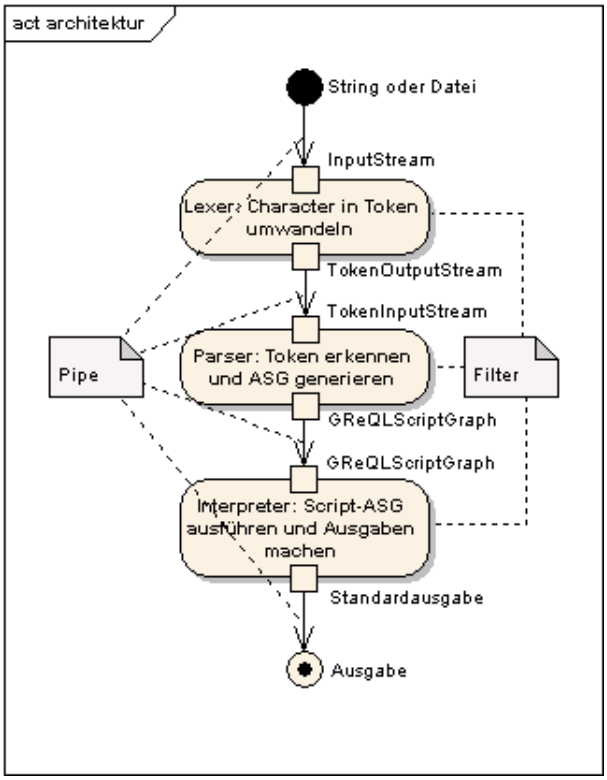

Abbildung 4.2: Pipe-Filter-Architektur

4.2 Abstrakter Syntaxgraph

An dieser Stelle wird spezifiziert, wie der Abstrakte Syntaxgraph aussehen soll. Dies wird mit einem Metamodell des Graphen erledigt. Im Folgenden wird Abstrakter Syntaxgraph mit ASG abgekürzt.

Das Metamodell des ASGs kann aus den GReQL-Script-EBNF-Regeln aus dem Anhang B abgeleitet werden. Im Folgenden werden einige Regeln vorgestellt, nach denen man die EBNF-Regeln in das Metamodell überführen kann.

4.2.1 Konvertierungsregeln

An dieser Stelle werden einige Faustregeln vorgestellt, nach denen die EBNF-Regeln in ein Metamodell des ASG umgewandelt werden können Man kann sich nicht strikt an diese Re-

geln halten, diese geben nur die Richtung vor, wie die EBNF-Regeln im Metamodell des ASG aussehen könnten. Lexikalische Elemente werden als terminal angesehen, d.h. man kann sie nicht weiter zerlegen. Diese werden wenn sie irrelevant sind weggelassen, z.B. wird das Schlüsselwort `if` nach der Erkennung nicht mehr benötigt und somit nicht in den ASG mit aufgenommen. Die relevanten Terminalsymbole werden als Attribute der Knoten gespeichert, z.B. wird der Variablenname im Attribut `name` des Knotens `Variable` gehalten.

Aggregation

Generell könnte man alle Regeln als Aggregation modellieren. Alle Unterelemente der Regel werden als Aggregation mit dem Regelnamen als Oberelement verknüpft.

Beispiel:

```
ifstatement =
  "if" '(' booleanexpression ')' statement ["else" statement] ;
```

Hier besteht das `ifstatement` aus mehreren anderen Elementen.

Eine Unterscheidung zwischen Aggregation und Komposition ist nicht notwendig.

Generalisierung

Spezielle Regeln, die genau ein Vorkommen eines nichtterminalen Elementes erlauben, können als Generalisierung modelliert werden. Diese Regeln erkennt man an einer Veroderung.

Beispiel:

```
statement =
  block
  | variabledeclaration ';'
  | assignment
  ...
```

Hier kann `block`, `variabledeclaration`, `assignment` usw. von `statement` abgeleitet werden.

Gemischt

Es könnten Regeln folgender Form vorkommen:

```
elA =
  elB | elC elD
```

Dann sollte man die Regeln zerlegen in eine Veroderung einzelner Elemente. Folgendes sollte resultieren

```
elA =
  elB | elE
elE =
  elC elD
```

Danach kann man wieder die vorherigen Regeln anwenden können. In den angegebenen EBNF-Regeln wurden alle solchen Regeln zerlegt.

Multiplizitäten

Die Multiplizitäten ergeben sich ebenfalls aus den Regeln.

Regeln mit optionalem Teil haben auf der Kind-Seite der Beziehung eine 0 als Minimum. Wenn das Unterelement vorhanden sein muss, steht auf der Kind-Seite der Beziehung eine 1 als Minimum. Wenn das Unterelement maximal ein Mal vorkommen kann, steht auf der Kind-Seite der Beziehung eine 1 als Maximum. Bei erlaubtem mehrfach vorkommenden Teil, steht auf der Kind-Seite der Beziehung ein * als Maximum. Als Beispiel kann man in der Abbildung 4.3 an der Beziehung ContainsImport erkennen, dass 0..* für optionales mehrfaches Vorkommen steht.

Auf der Eltern-Seite der Beziehung steht eine 1 als Minimum, wenn das Unterelement in keiner anderen Regel vorkommt. Andernfalls eine 0, weil das Element mit anderen Elementen verbunden werden kann und nicht mit diesem. Bei Maximum auf der Eltern-Seite der Beziehung steht eine 1, wenn das Unterelement mit maximal einem Elternelement dieser Sorte verbunden ist, z.B. eine `ParameterDeclaration` ist mit maximal einer `ScriptFunction` verbunden. Wenn ein Unterelement mit mehreren Eltern-Knoten einer Klasse verbunden ist, dann ist das Maximum *, z.B. eine `Variable` kann in mehreren `IfStatement`s enthalten sein.

4.2.2 Metamodell

In diesem Abschnitt wird das Metamodell des ASG mit Klassendiagrammmen beschrieben.

Scripte

Der Script-Knoten ist der erste Knoten des Graphen. Er enthält die Importanweisungen und die Funktionsdeklarationen. Die Abbildung 4.3 enthält folgende Regeln:

```
script =
  {importline} functiondeclaration {functiondeclaration} ;

importline =
  ("import" | "importjava") pathgreqlquery ';' ;

pathgreqlquery =
  queryexpression ;

queryexpression =
  GREQLQUERY ;
```

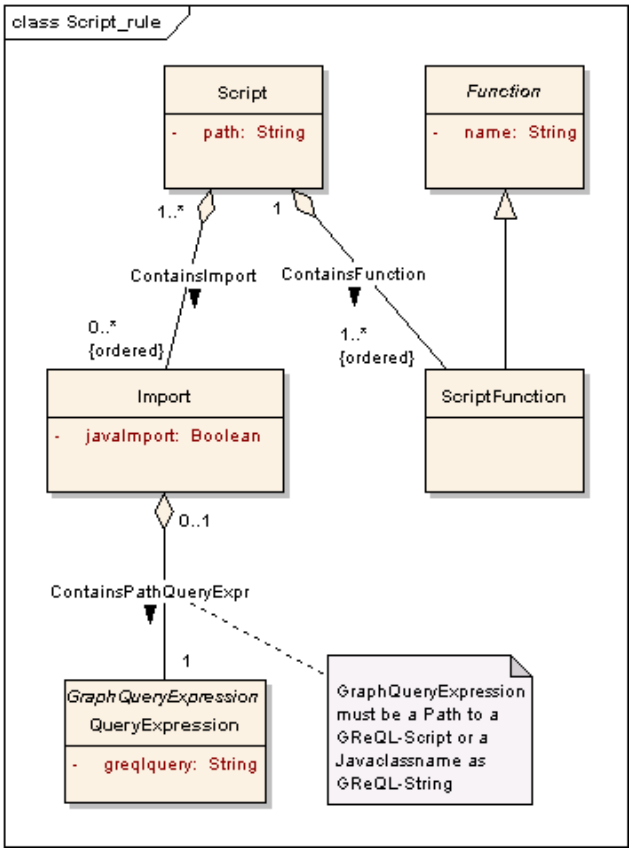

Abbildung 4.3: Scripte

In der Regel `script` sind keine Veroderungen enthalten, also kann diese Regel mit Aggregationen modelliert werden. In der zweiten Regel ist eine Veroderung enthalten, man könnte also eine Generalisierung einbauen. Aber da die Veroderung nur aus zwei terminalen Symbolen besteht, ist ein boolesches Attribut geeigneter. Die Regel `pathgreqlquery` ist nur eine Zwischenregel, die in der EBNF den Zweck hat, dass man erkennt, dass die `queryexpression` einen Pfad zurückgeben sollte. Man würde nicht zur Parse-Zeit prüfen ob die GReQLQuery einen Pfad zurück gibt, deshalb ist es ausreichend, wenn anstatt `pathgreqlquery` die `queryexpression` verwendet wird. In den EBNF-Regeln ist diese Regel zur besseren Lesbarkeit eingebaut. Diese Zwischenregel ist somit im ASG überflüssig. Die `queryexpression` enthält die eigentliche GReQLQuery.

Funktionen

Die Abbildung 4.4 enthält folgende Regeln:

```
functiondeclaration =
  functionid '(' [parameterdeclaration {',' parameterdeclaration}] ')'
     block ;

functionid =
  IDENT ;
```

```
parameterdeclaration =
  ["var"] variable ;

variable =
  IDENT ;
```

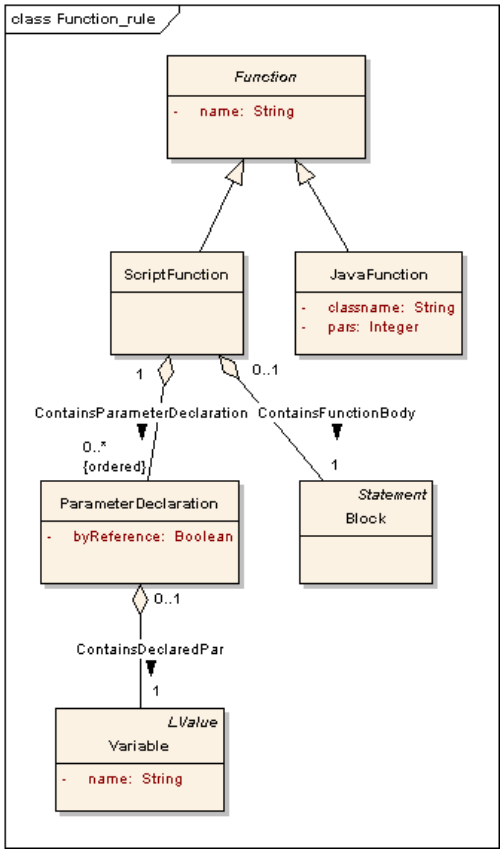

Abbildung 4.4: Funktionen

Da man `JavaFunctions` verwenden kann, müssen sie auch im ASG enthalten sein. Die Knoten-Klasse `Function` wurde als Oberklasse gewählt, damit im `FunctionCall` nicht zwischen `JavaFunction` und `ScriptFunction` unterschieden werden muss. Die Knoten-Klasse `Function` ist abstract, aber enthält den Bezeichner. `JavaFunctions` haben zusätzliche Informationen um identifiziert zu werden. Eine `ScriptFunction` besteht aus Parameterdeklarationen und einem Funktionsrumpf. Das entspricht der `functiondeclaration`-Regel aus den EBNF-Regeln. Der Funktionsrumpf besteht aus einem `Block`, dieser ist eine Anweisung. Bei der `ParameterDeclaration` gibt das Schlüsselwort `var` an, ob es sich um einen call by reference Parameter handelt, dieses wird als boolescher Attribut gespeichert. Der Knoten `Variable` enthält nur den Bezeichner der Variablen.

Anweisungen

Die Abbildung 4.5 enthält folgende Regeln:

```
block =
  '{' {statement} '}' ;

statement =
  block
  | variabledeclaration ';'
  | assignment
  | ifstatement
  | labeledstatement
  | returnstatement
  | breakstatement
  | continuestatement
  | assertstatement
  | expression ';'
  | emptystatement ;

labeledstatement =
  [label ':'] (loopstatement | typeswitchstatement) ;

label =
  IDENT ;
```

In der Abbildung 4.5 sind alle möglichen Anweisungen vorhanden. Der Block enthält beliebig viele Anweisungen und ist selbst eine Anweisung, nach dem Composite-Design-Pattern aus [GHJV96]. So können Blöcke ineinander geschachtelt werden. An der `statement`-Regel kann man sehen, dass eine Veroderung mit der Generalisierung am besten zu modellieren ist. Beim `LabeledStatement` wird dei Veroderung ebenfalls mit einer Generalisierung modelliert. Der Knoten `LabeledStatement` kann ein Label haben, das bei `break`- und `continue`-Anweisungen angegeben werden kann. Der detailliere Aufbau einiger Knoten-Klassen wird in weiteren Abbildungen verfeinert.

Variablen Deklarationen

Die Abbildung 4.6 enthält folgende Regeln:

```
variabledeclaration =
  singlevariabledeclaration {',' variable} ;

singlevariabledeclaration =
  "var" variable ;
```

In der Abbildung 4.6 sieht man, dass man einige Regeln zusammenfassen kann. Eine Variablendeklaration deklariert eine oder mehrere Variablen. Die `singlevariabledeclaration` wird bei der `Foreach`-Schleife benötigt. In der EBNF muss die `Foreach`-Schleife nur eine Variable deklarieren, im ASG ist diese Spezialisierung nicht notwendig.

Zuweisungen

Die Abbildung 4.7 enthält folgende Regeln:

```
assignment =
```

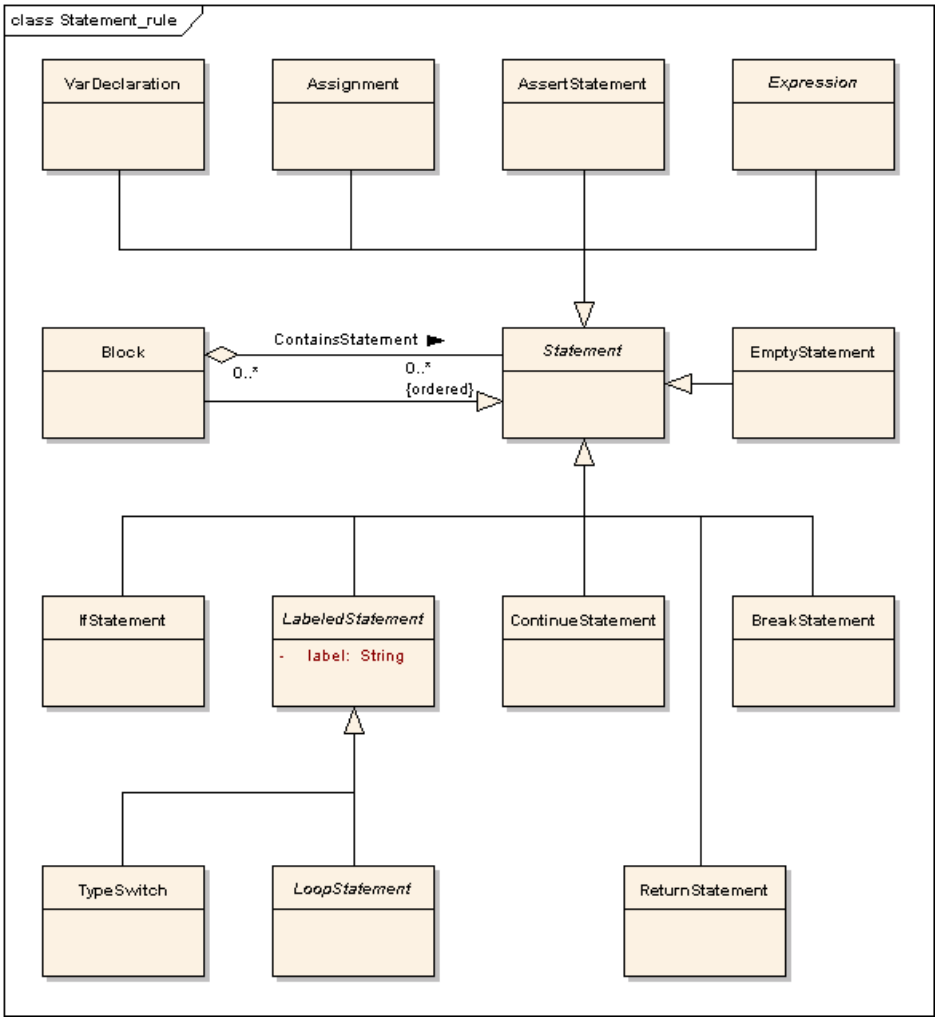

Abbildung 4.5: Anweisungen

```
  lvalue '=' expression ';' ;

lvalue =
  variable
  | attributeaccess ;

attributeaccess =
  expression '.' attributename ;

attributename =
  IDENT ;
```

Eine Zuweisung besteht aus dem Ziel, einer Variablen oder einem Attributzugriff, dem ein Wert zugewiesen werden soll, und dem Ausdruck, dessen Wert zugewiesen werden soll. Der `AttributeAccess` besteht aus dem Ausdruck, auf dessen Attribut zugegriffen werden soll, und dem Attributnamen des Attributs, auf das zugegriffen werden soll.

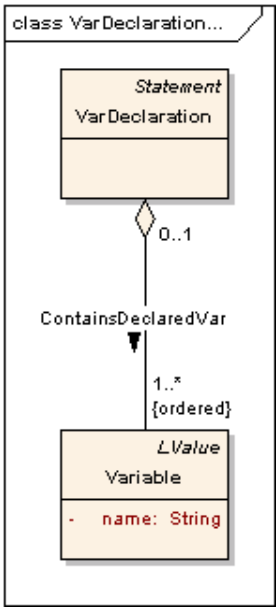

Abbildung 4.6: Variablen Deklarationen

`if`-Anweisungen

Die Abbildung 4.8 enthält folgende Regeln:

```
ifstatement =
  "if" '(' booleanexpression ')' statement ["else" statement] ;

booleanexpression =
  expression ;
```

Das `IfStatement` besteht aus dem Ausdruck, der einen booleschen Wert zurückgeben soll, der entscheidet welche Alternative ausgeführt wird, dem `then`-Teil und dem optionalen `else`-Teil. In der Abbildung 4.8 kann man die Multiplizitäten der `Statement`-Beziehungen vergleichen. Bei `ContainsThenPart` ist eine 1 auf der Kind-Seite, d. h. es muss genau ein Statement als `ThenPart` verbunden sein. Bei `ContainsElsePart` ist 0..1 auf der Kind-Seite angegeben, d. h. es darf ein Statement als `ElsePart` verbunden sein.

`typeswitch`-Anweisungen

Die Abbildung 4.9 enthält folgende Regeln:

```
typeswitchstatement =
  "typeswitch" '(' expression ')' '{' switchcase {switchcase} [defaultcase
     ] '}' ;

switchcase =
  "case" jvaluetype {',' jvaluetype} ':' statement ;

jvaluetype =
  "ATTRIBUTEDELEMENT"
  | "ATTRIBUTEDELEMENTCLASS"
```

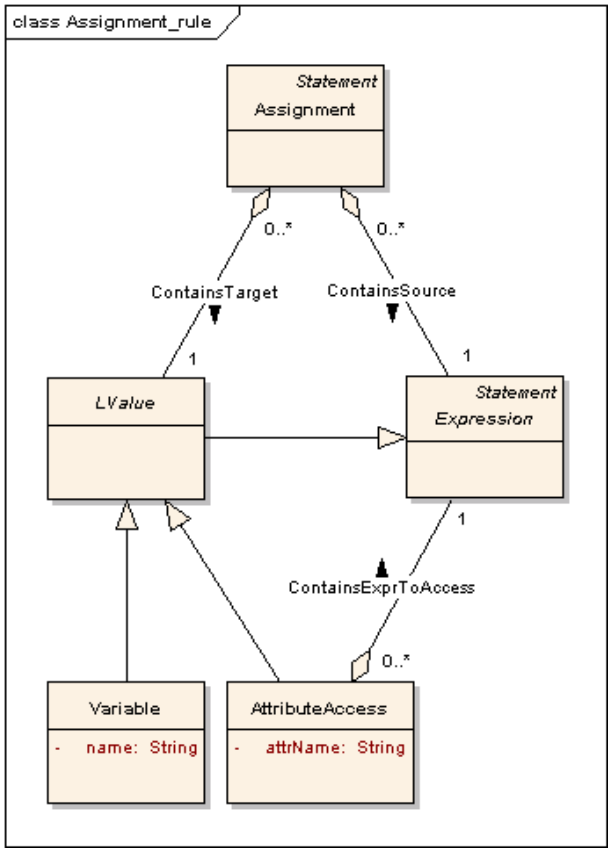

Abbildung 4.7: Zuweisungen

```
  | "BOOLEAN"
  | "CHARACTER"
  | "DOUBLE"
  | "EDGE"
  | "ENUMVALUE"
  | "GRAPH"
  | "INTEGER"
  | "COLLECTION"
  | "RECORD"
  | "LONG"
  | "OBJECT"
  | "PATH"
  | "PATHSYSTEM"
  | "STRING"
  | "VERTEX" ;

defaultcase =
  "default" ':' statement ;
```

Abbildung 4.9 zeigt dem Aufbau eines `TypeSwitch`s. Ein `TypeSwitch` enthält eine `Expression` und mindestens einen `SwitchCase`. Ein `TypeSwitch` kann auch mehrere `SwitchCases` enthalten. Optional kann ein `TypeSwitch` auch einen `DefaultCase` ha-

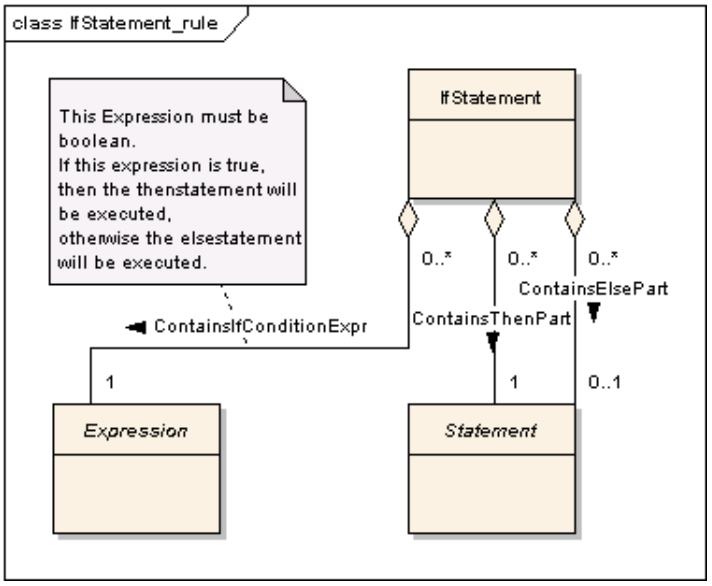

Abbildung 4.8: `if`-Anweisungen

ben. Jeder `SwitchCase` muss mindestens einen `JValueType` und ein `Statement` enthalten. Ein `DefaultCase` muss nur ein `Statement` enthalten.

Schleifen

Die Abbildung 4.10 enthält folgende Regeln:

```
loopstatement =
  (foreachloop | whileloop | dowhileloop) ;

foreachloop =
  "foreach" '(' singlevariabledeclaration ':' collectionexpression ')'
     statement ;

whileloop =
  "while" '(' booleanexpression ')' statement ;

dowhileloop =
  "do" statement "while" '(' booleanexpression ')' ';' ;

collectionexpression =
  expression ;

booleanexpression =
  expression ;
```

In der Abbildung 4.10 sieht man alle Schleifen, die es in GReQLScript gibt. An dieser Stelle weicht die EBNF von dem Metamodell etwas ab, weil es Gemeinsamkeiten gibt, die in der EBNF nicht mehr zerlegt werden können, aber in dem Metamodell durch Generalisierung verbessert werden können. Alle Schleifen haben einen Schleifenrumpf, dieser ist ein Statement und

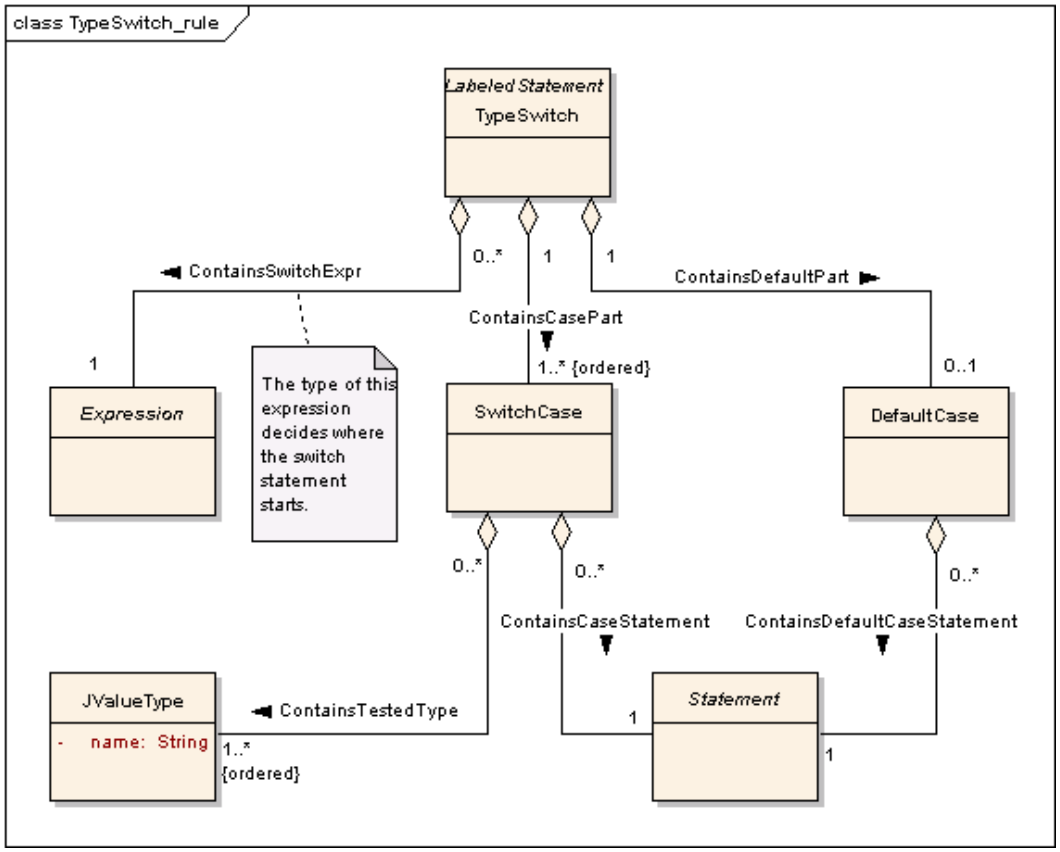

Abbildung 4.9: `typeswitch`-Anweisungen

ist in `LoopStatement` enthalten. Im ASG haben die `While`-Schleife und die `DoWhile`-Schleife den gleichen Aufbau. Der Unterschied wird erst bei der Ausführung bemerkbar. Beide Schleifen bestehen aus den Schleifenrumpf und einer `Expression`, deren Auswertung einen booleschen Wert haben muss. Diese Schleifen sind als `ConditionalLoop` generalisiert. Der Unterschied dieser Schleifen liegt in deren Ausführung. Die `Foreach`-Schleife dagegen hat außer dem Schleifenrumpf eine Variablendeklaration und eine `Expression`, deren Auswertungwert eine `Collection` sein muss.

Unterbrechungsanweisungen

Die Abbildung 4.11 enthält folgende Regeln:

```
returnstatement =
  "return" [expression] ';' ;

breakstatement =
  "break" [label] ';' ;

continuestatement =
  "continue" [label] ';' ;
```

In der Abbildung 4.11 sieht man die unterbrechenden Anweisungen. Das

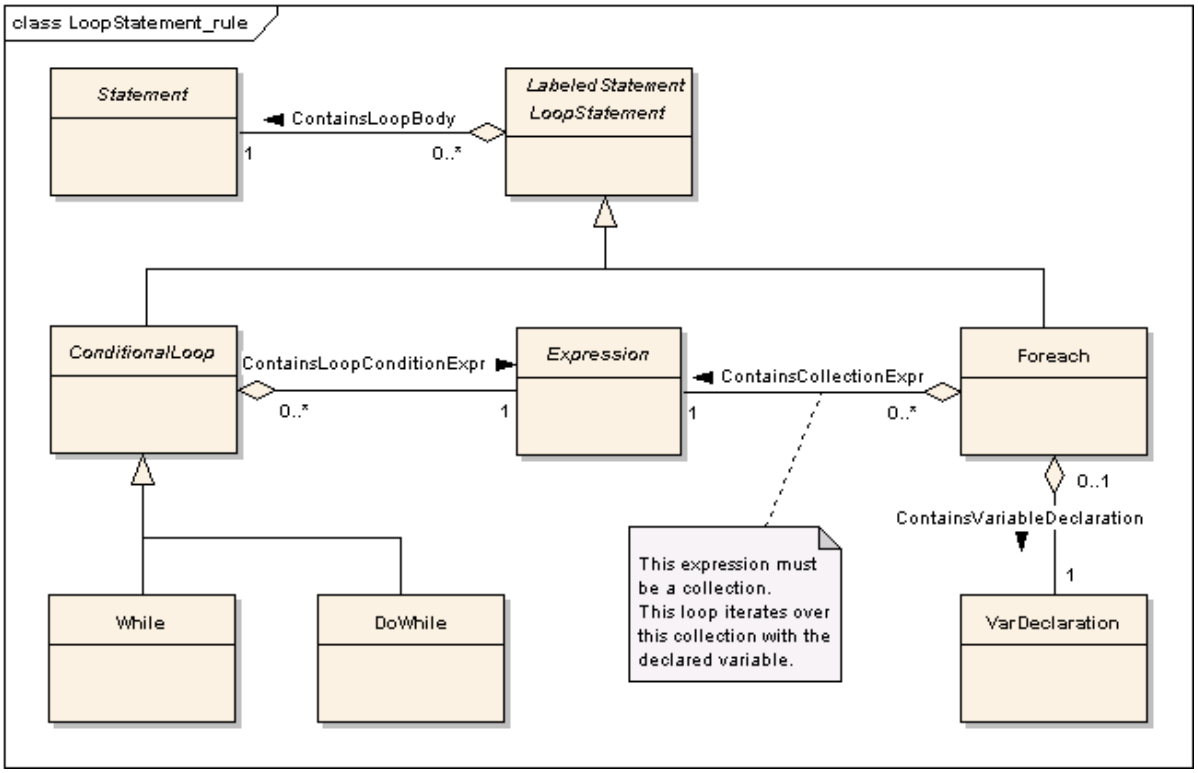

Abbildung 4.10: Schleifen

`ReturnStatement` kann allein oder mit einer `Expression`, deren Wert eine Funktion zurückgeben soll, vorkommen. `BreakStatement` und `ContinueStatement` können ein Label der Schleife enthalten, die sie unterbrechen sollen. Das Label bei `BreakStatement` kann auch eines `Typeswitch`es sein.

Zusicherungen

Die Abbildung 4.12 enthält folgende Regeln:

```
assertstatement =
  "assert" booleanexpression [':' stringexpression]';' ;

stringexpression =
  expression ;
```

Der Aufbau der Assert-Anweisung aus der Abbildung 4.12 ist der gleiche wie bei Java 5.0. Sie enthält zwei `Expressions` von denen eine, die einen booleschen Wert zurückgeben soll, vorhanden sein muss und die andere optional ist, dessen Wert wird als Fehlermeldung ausgegeben.

Ausdrücke

Die Abbildung 4.13 enthält folgende Regeln:

```
expression =
  graphqueryexpression
```

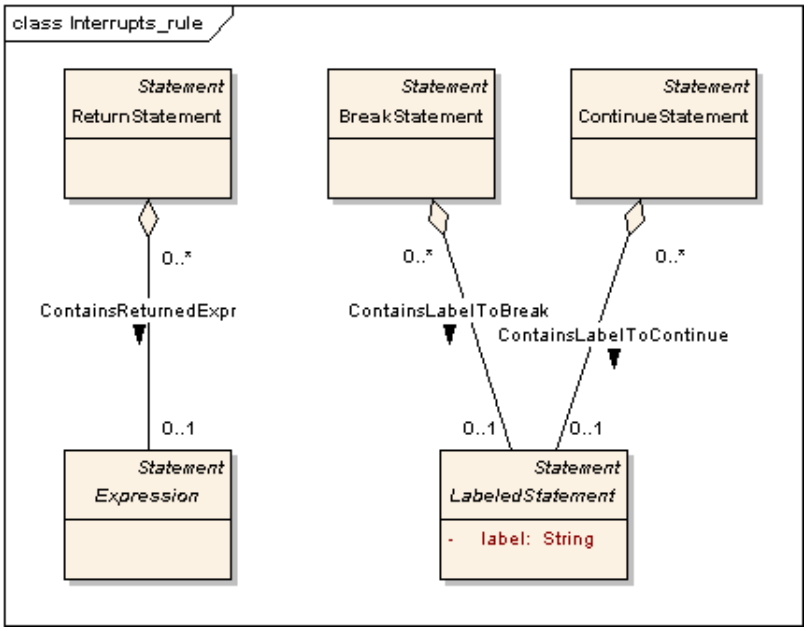

Abbildung 4.11: Unterbrechungsanweisungen

```
  | lvalue
  | functioncall
  | hastypeexpression ;

graphqueryexpression =
  [graph '.'] (queryexpression | evaluatequeryfunction) ;

graph =
  expression ;

queryexpression =
  GREQLQUERY ;

evaluatequeryfunction =
  "evaluateQuery" '(' pathgreqlquery ')' ;

functioncall =
  functionid '(' [expression {',' expression}] ')' ;

hastypeexpression =
  expression "hastype" jvaluetype ;
```

In der Abbildung 4.13 sind alle Ausdrücke enthalten. Es gibt vier Arten von Ausdrücken: GReQL-Anfragen, LValues, Funktionsaufrufe und Hastype-Ausdrücke. GReQL-Anfragen können direkt im Code oder aus einer Datei mit der `evaluateQuery()`-Funktion verwendet werden. Beide können einen Graphen verwenden. Die `LValue`-Regel wurde bereits in der Abbildung 4.7 erwähnt. Beim Funktionsaufruf wird der Funktionsbezeichner und die Parameterausdrücke angegeben. Die `HastypeExpression` enthält einen Ausdruck und einen

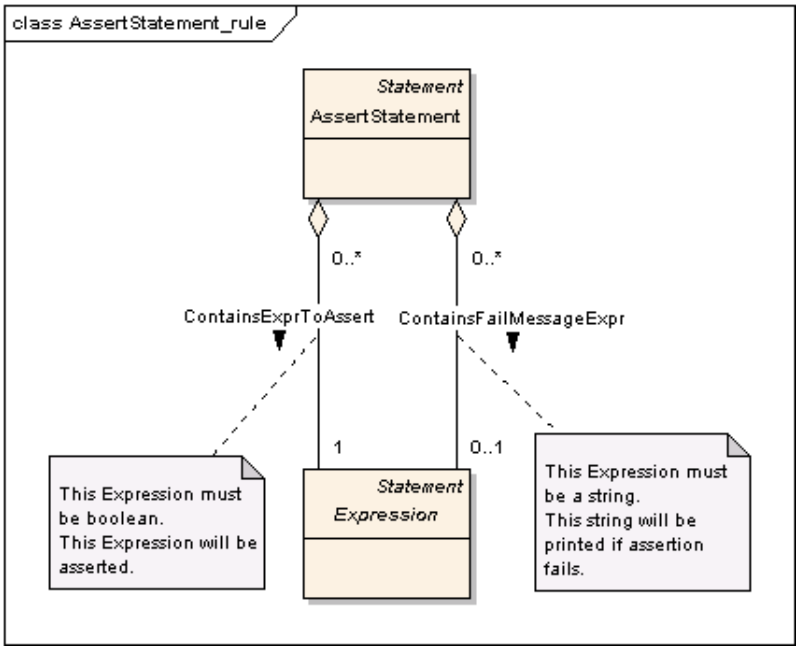

Abbildung 4.12: Zusicherungen

Typ.

4.3 ANTLR

ANTLR ist ein Parsergenerator, der einen rekursiv absteigenden prädiktiven LL(k)-Parser generiert. Rekursiv absteigend bedeutet hier, dass der Parser für jedes Nichtterminalsymbol eine Methode besitzt. Prädiktiv bedeutet, dass der Methodenaufruf vom Lookahead der Größe k bestimmt wird. Diese Methoden können Aktionen in Form von Code enthalten, mit denen der ASG in `GReQLScriptParser` generiert wird.

In GReQL-Script wurde die Version 2.7.6 [Pm05] verwendet.

4.3.1 Grammatik

Eine ANTLR-Grammatik besteht aus Produktionsregeln, wie auch die EBNF. Beim Erstellen der ANTLR-Grammatik kann fast alles aus der EBNF übernommen werden. Es gibt jedoch Unterschiede. Die Syntax der ANTLR-Grammatik ist ähnlich der EBNF. In der Tabelle 4.1 kann man die Unterschiede der Syntax sehen.

So wird aus der EBNF-`script`-Regel

```
script =
  {importline} functiondeclaration {functiondeclaration} ;
```

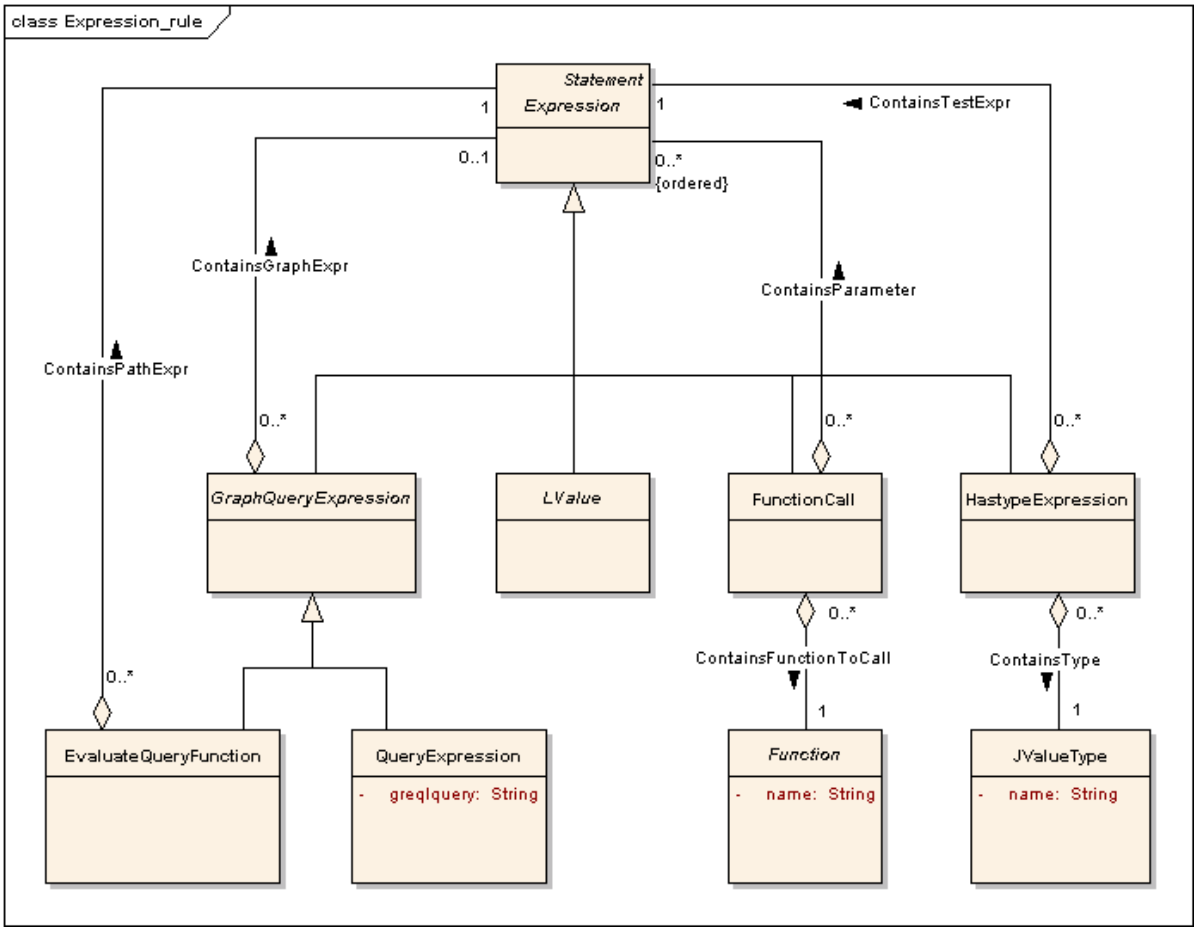

Abbildung 4.13: Ausdrücke

die ANTLR-`script`-Regel (ohne Aktionen)

```
script
  : (importline)*
    (functiondeclaration)+ ;
```

4.3.2 Aktionen

Mit Hilfe der Aktionen wird der ASG generiert.

An dieser Stelle kann nicht die gesamte ANTLR-Grammatik erläutert werden. Es wird folgende Struktur bei den Parser-Regeln verwendet. Die Begriffe in spitzen Klammern werden weiter unten erläutert.

```
<modifier> <parserrulename>
[<parameter>]
returns [<resulttype> <resultname>]
throws <exceptions>
```

EBNF	Bedeutung	ANTLR
`X= ... ;`	X ist ... ;	`X: ... ;`
`[X]`	0 oder 1 (optionales) Vorkommen von X	`(X)?`
`{X}`	kein oder beliebig viele Vorkommen von X	`(X)*`
`X{X}`	ein oder beliebig viele Vorkommen von X	`(X)+`
`X|Y`	X oder Y	`X|Y`
`'x'`	x ist ein Zeichen in der Eingabe	`'x'`
`"xy"`	ist eine Zeichenfolge in der Eingabe	`"xy"`
`(...)`	zur Klammerung von ...	`(...)`
	Nicht in der originalen EBNF enthalten	
`.`	ein beliebiges Zeichen in der Eingabe	`.`
`(.)\('x'|'y')`	ein beliebiges Zeichen außer x oder y	`~('x'|'y')`

Tabelle 4.1: EBNF-ANTLR-Vergleich

```
{ <initactions> }
  : { <anyactions> }
    <varname> = <parserrulename2>
    { <anyactions> }
    <anyvarname> : <lexerrulename>
    { <anyactions> }
  ;
```

- `<modifier>` enthält die Java-Modifier der Methode. Ist in GReQLScript bei allen Regeln `protected`, da die Regeln nur vom Parser aufgerufen werden.
- `<parserrulename>` gibt den Regelnamen bzw. Methodennamen an.
- `<parameter>` Parameterliste in eckigen Klammern.
- `<resulttype>` Rückgabetyp der Methode.
- `<resultname>` Rückgabevariable der Methode, wird benötigt da keine `return`-Anweisung benutzt werden darf.
- `<exceptions>` Ausnahmen die geworfen werden können.
- `<initactions>` Javacode, der am Anfang der Methode stehen wird. Enthält Initialisierungen, z.B. Variablen die man außer der *result*-Variable benötigt. Aktionen werden in geschweiften Klammern gesetzt.
- `<anyactions>` Javacode als Aktion. Wird ausgeführt wenn erreicht wird.
- `<varname>` Variable, entweder die Rückgabevariable oder eine, die in `<initactions>` initialisiert wurde. Dieser Variablen wird der Rückgabewert der Regel/Methode `<parserrulename2>` zugewiesen.
- `<anyvarname>` Variable, die vorher undefiniert war. Dieser Variablen wird das Token der LexerRegel/Methode `<lexerrulename>` zugewiesen.

4.3.3 ASG-Generierung

Der Parser erkennt nicht nur die Grammatik, sondern er generiert dabei den Abstrakten Syntaxgraph. Man kann in der ANTLR-Grammatik nach dem Erkennen eines Teils eine Aktion ausführen lassen. Diese Aktionen sind Java-Code, der ausgeführt wird, wenn eine Stelle erreicht wurde, also ein bestimmter Teil erkannt wurde. Die Regeln sind im Parser Methoden und werden rekursiv aufgerufen, angefangen mit der `script`-Regel. Beim Aufruf einer Produktionsregel wird der passende Knoten im ASG mittels JGraLab erzeugt. Dann werden die untergeordneten Regeln aufgerufen, diese liefern fertige Knoten. Zum Schluss werden die von untergeordneten Regeln bzw. Methoden erstellten Knoten und der Knoten aus der aktuellen Regel verbunden und nach oben weiter gereicht. So wird beim Aufruf der `Script`-Methode der gesamte ASG rekursiv erzeugt.

Am Beispiel von Block kann man erkennen, dass zuerst ein Knoten im ASG-Graph erzeugt wird. Danach wird eine öffnende geschweifte Klammer (LCURLY) erkannt, die im Lexer definiert ist. Nach der Klammer sollen `Statement`s erkannt werden. Wenn ein Statement erkannt wird, wird die dazugehörige Aktion ausgeführt. Diese Aktion verbindet den `Statement`-Knoten, der von der `statement`-Regel zurückgegeben wurde, mit dem erzeugten `Block`-Knoten. Wenn die schließende geschweifte Klammer (RCURLY) erkannt wurde, wird der `Block`-Knoten nach oben in der Aufrufhierarhie gereicht zu der Regel, die ihn braucht, z.B. zu der `while`-Regel.

```
block returns [Block b = null] {
        b = graph.createBlock();
    Statement s = null;
} : LCURLY
    (s=statement {
            graph.createContainsStatement(b,s);
    })*
    RCURLY
  ;
```

Einige Beispiele aus der GReQL-Script ANTLR-Grammatik:

Bei der `AttributeName`-Regel wird ein Attributname erkannt und als String zurückgegeben.

EBNF-Regel:

```
attributename =
  IDENT ;
```

ANTLR-Regel:

```
protected attributename returns [String an = null]
  : id:IDENT
    {
        an=id.getText();
    }
  ;
```

Die `Variable`-Regel sieht folgendermaßen aus.

EBNF-Regel:

```
variable =
  IDENT ;
```

Die `Variable`-Regel wurde zur einfacheren Behandlung in zwei Regeln aufgeteilt. Die `variabledef`-Regel wird an den Stellen verwendet, wo Variablen definiert werden. Die `variableuse`-Regel wird dagegen an den Benutzungsstellen verwendet. Eine Alternative wäre eine Regel mit einem booleschem Parameter.

Die `variabledef`-Regel gibt einen neuen `Variable`-Knoten zurück, wenn er noch nicht deklariert wurde, andernfalls wird eine `DuplicateSymbolException` geworfen. Die `variableuse`-Regel gibt einen existierenden `Variable`-Knoten zurück, wenn er bereits deklariert wurde, andernfalls wird eine `UndefinedSymbolException` geworfen.

Die Überprüfung auf Fehler erfolgt durch die `SymbolTable variables`. Symboltabellen werden im Abschnitt 4.3.4 erklärt. Das Abfangen des Fehlers dient seiner Lokalisierung im Quelltext.

ANTLR-Regel:

```
protected variabledef
returns [Variable vid = null]
throws DuplicateSymbolException
{
        SourcePosition sp = null;
}
  : {sp=getCurrentSourcePosition();}
    id:IDENT
    {
                //this is called only from declarations
                //->create Variable node
                vid = graph.createVariable();
                vid.setName(id.getText());
                try{
                        variables.put(id.getText(), vid);
                } catch(DuplicateSymbolException e){
                        e.setFilename(path);
                        e.setSourcePosition(sp);
                        throw e;
                }
    }
  ;

protected variableuse
returns [Variable vid = null]
throws UndefinedSymbolException
{
        SourcePosition sp = null;
}
  : {sp = getCurrentSourcePosition();}
```

```
    id:IDENT
    {
        //this is called by var uses
        //get an existing var, in this block
                try {
                        vid=variables.get(id.getText());
                } catch (UndefinedSymbolException e) {
                        e.setFilename(path);
                        e.setSourcePosition(sp);
                        throw e;
                }
    }
  ;
```

Die `IfStatement`-Regel sieht folgendermaßen aus.

EBNF-Regel:

```
ifstatement =
  "if" '(' booleanexpression ')' statement ["else" statement] ;
```

Diese Regel gibt einen `IfStatement`-Knoten zurück. Dabei können `Exceptions` auftreten. Im Initialisierungsblock werden der `IfStatement`-Knoten und einige Hilfsvariablen erzeugt. Im Laufe der Erkennung werden die Hilfsvariablen belegt und dann in Aktionen mit dem erzeugten `IfStatement`-Knoten verbunden.

ANTLR-Regel:

```
protected ifstatement
returns [IfStatement is = null]
throws DuplicateSymbolException, UndefinedSymbolException,
    GReQLScriptSemanticException
{
        is = graph.createIfStatement();
        Expression expr = null;
        Statement s = null;
        SourcePosition sp = null;
}
  : "if" LPAREN ({sp = getCurrentSourcePosition();}
                expr=expression {
                createEdge(ContainsIfConditionExpr.class,is,expr,sp);
    }) RPAREN ({sp = getCurrentSourcePosition();}
        s=statement {
        createEdge(ContainsThenPart.class,is,s,sp);
    }) ({LA(1)==LITERAL_else}? "else" ({sp = getCurrentSourcePosition();}
        s=statement {
        createEdge(ContainsElsePart.class,is,s,sp);
    }))?
  ;
```

Damit der ASG kein Baum, sondern ein Graph wird, um mehrfache Knoten von einer Variablen oder ähnlichen Knoten zu vermeiden, werden Symboltabellen eingesetzt.

4.3.4 Symboltabellen

Eine Symboltabelle wird benötigt um eine Verwendung von nicht definierten Variablen oder die doppelte Deklaration einer Funktion zu erkennen. Mittels Symboltabellen wird z.B. ein Knoten einer deklarierten `Variable` nicht neu erzeugt, sondern an mehreren Stellen verbunden. Symboltabellen erleichtern die Verwaltung von `Imports`, `Functions`, `Variables` und `Labels`.

Man könnte `HashMaps` als Symboltabelle verwenden. Dies würde zwar ausreichen, aber es würde viel Code-Duplikate geben, die auf Fehler prüfen. Deshalb wird dafür eine eigene Klasse verwendet, die damit verbundene Aufgaben übernimmt.

In der Abbildung 4.14 sieht man die Schnittstelle der `SymbolTable`, die im GReQLScript-Parser verwendet wird. Diese Klasse ist parametrisiert. Der Parameter `T`, der den Typ der zugewiesenen Elemente angibt, wird bei der Erzeugung angegeben.

- `put(String symbol, T element): T` die Methode `put()` weist dem Symbol `symbol` ein `Object element` vom Typ `T` zu. Diese Methode wirft eine Ausnahme, wenn diesem Symbol bereits ein `Object` zugewiesen ist. Das zugewiesene `Object element` wird zurückgegeben.
- `get(String symbol): T` die Methode `get()` gibt das zugewiesene `Object` zurück oder wirft eine Ausnahme, wenn dem Symbol kein `Object` zugewiesen wurde.
- `containsSymbol(String symbol): boolean` die Methode `containsSymbol()` gibt `true` zurück, wenn dem Symbol `symbol` ein `Object` zugewiesen ist.
- `removeSymbol(String symbol): T` die Methode `removeSymbol()` hebt die Zuweisung auf und gibt das bisher zugewiesene `Object` zurück oder wirft eine Ausnahme, wenn dem Symbol kein `Object` zugewiesen wurde.
- `openBlock(): void` die Methode `openBlock()` startet einen neuen Unterblock.
- `closeBlock(): void` die Methode `closeBlock()` schließt einen Unterblock.
- `isEmpty(): boolean` die Methode `isEmpty()` gibt `true` zurück, wenn in der `SymbolTable` keine Symbolzuweisungen enthalten sind.
- `values(): boolean` die Methode `values()` gibt alle zugewiesenen `Objects` zurück.

In GReQL-Script werden 6 Symboltabellen verwendet.

- für Scripte
- für deklarierte Funktionen
- für undeklarierte Funktionen
- für Variablen
- für Label
- für `JValueType` aus GReQL

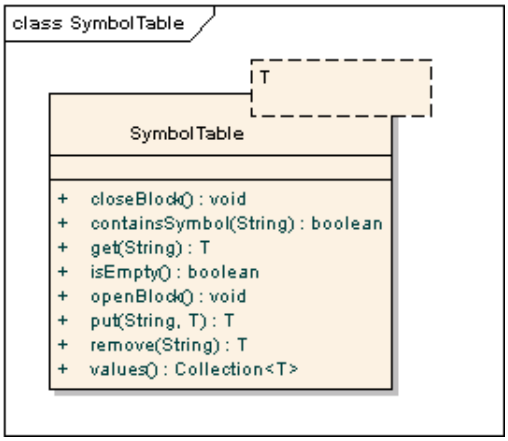

Abbildung 4.14: SymbolTable

Eine `SymbolTable` wird dazu verwendet, um Scripte nicht mehrfach zu importieren. Beim Aufruf der Import-Regel wird geprüft, ob das angegebene Script bereits erkannt wurde bzw. in Erkennung ist. Wenn der Aufruf von `containsSymbol(scriptname)` `false` zurück gibt, wird es importiert.

Die Klasse `FunctionSymbolTable` verwaltet zwei interne `SymbolTables`, eine für deklarierte Funktionen, eine für nicht deklarierte Funktionen und zusätzlich gibt es einen `GraphMarker` für Java-Methoden-Referenzen. Um Funktionen mit gleichen Namen aber mit unterschiedlicher Parameteranzahl zu unterstützen, wird eine Kombination aus dem Funktionsnamen und der Parameteranzahl als Symbol bei den `SymbolTables` verwendet.

Zuerst werden alle Java Funktionen aus den Importierten Java-Klassen im ASG deklariert. Bei einem Import einer Java-Klasse, wird für jede enthaltene Methode geprüft ob sie in GReQLScript verwendet werden kann und wenn ja, dann wird mittels der Methode `createJavaFunction()` ein `JavaFunction`-Knoten im ASG erzeugt. Dann wird für jede erkannte Funktion im Script die Methode `createScriptFunction()` aufgerufen, die eine `ScriptFunction` generiert. Dabei prüfen beide Methoden, ob keine `Function` mit diesem Namen und Parameterzahl bereits deklariert wurde.

Beim Erkennen eines `FunctionCalls` wird die Methode `getFunction()` benutzt. Diese gibt die deklarierte `Function` zurück, dabei muss nicht zwischen `ScriptFunction` und `JavaFunction` unterschieden werden. Bei einer Kreis-Abhängigkeit der Scripte kann es passieren, dass eine Funktion verwendet wird ohne vorher deklariert worden zu sein. In diesem Fall erzeugt die Methode `getFunction()` eine neue `ScriptFunction` und vermerkt diese als undeklariert. Später wird `createScriptFunction()` für diese Funktion aufgerufen und deklariert. Falls die undeklarierte `Function` nach dem Erkennen des Scripts immer noch undeklariert ist, wird eine Ausnahme geworfen.

Eine der `SymbolTables` ist für die Verwaltung der Variablen zuständig. Mit den Methoden `openBlock()` und `closeBlock()` wird die Schachtelung der Blöcke kontrolliert.

Für die Verwaltung von Labels wird die Klasse `LabelSymbolTable` verwendet, die intern eine `SymbolTable` enthält.

Da die Funktionsnamen und Variablennamen in unterschiedlichen Symboltabellen verwaltet werden, kann eine Variable den gleichen Namen wie eine Funktion haben, z.B. „sqrt“.

4.3.5 Probleme

ANTLR generiert einen LL(k)-Parser aus der ANTLR-Grammatik. Der Nachteil des *rekursiv absteigenden* Parsers ist, dass er Linksrekursionen der Grammatikregeln nicht erkennen kann, und damit in einer Endlosschleife geraten würde.

Die GReQLScript-EBNF enthält eine Linksrekursion, die durch `Expression` verursacht wird. Z.B. fängt eine `HastypeExpression` ebenfalls mit einer `Expression` an. Das Problem kann umgangen werden, indem die `Expression` anders beschrieben wird. In [ASU88] wird ein Algorithmus vorgestellt, der Linksrekursionen auflöst.

Zuerst müssen die betroffenen Regeln identifiziert werden. In diesem Fall sind es die folgenden fünf Regeln.

```
expression
  : graphqueryexpression
  | identifier
  | functioncall
  | hastypeexpression
  ;

graphqueryexpression
  : (expression DOT)? (queryexpression | evaluatequeryfunction)
  ;

identifier
  : variableid
  | attributeaccess
  ;

attributeaccess
  : expression DOT attributename
  ;

hastypeexpression
  : expression HASTYPE_KW JVALUETYPE
  ;
```

Danach werden die Regeln in eine Regel zusammengefasst. Dazu ersetzt man die Vorkommen `graphqueryexpression`, `identifier`, `attributeaccess`, `hastypeexpression` durch ihre Produktionen. Man erhält folgende Regel.

```
expression
  : (expression DOT)? (queryexpression | evaluatequeryfunction)
  | variableid
  | expression DOT attributename
  | functioncall
  | expression HASTYPE_KW JVALUETYPE
```

```
  ;
```

Dann werden die Klammern aufgelöst und man erhält Produktionen mit und ohne `expression` am Anfang.

```
expression
  : expression DOT queryexpression
  | expression DOT evaluatequeryfunction
  | expression DOT attributename
  | expression HASTYPE_KW JVALUETYPE
  | queryexpression
  | evaluatequeryfunction
  | variableid
  | functioncall
  ;
```

Die Produktionen ohne `expression` können als terminierend angesehen werden, d.h. keine Rekursion. Diese werden in einer Regel `exprleftpart` zusammen gefasst. Die Produktionen mit `expression` am Anfang erweitern die `expression` nach rechts. Diese Produktionen ohne werden ohne `expression` als `exprrightpart` zusammen gefasst. Die Regel `expression` besteht nun aus dem terminalen Teil und beliebig vielen optionalen erweiternden Teilen. Diese Regeln sind nun linksrekursionsfrei, aber schlecht strukturiert.

```
expression
  : exprleftpart exprrightpart*
  ;

exprleftpart
  : queryexpression
  | evaluatequeryfunction
  | variableid
  | functioncall
  ;

exprrightpart
  : DOT queryexpression
  | DOT evaluatequeryfunction
  | DOT attributename
  | HASTYPE_KW JVALUETYPE
  ;
```

Nach einigen Umstrukturierungen resultieren folgende Regeln.

```
expression
  : exprleftpart
  ( graphqueryexpressionaddon
  | attributeaccessaddon
  | hastypeexpressionaddon)*
  ;

exprleftpart
  : graphqueryexpression
  | variableid
```

```
  | functioncall
  ;

graphqueryexpression
  : queryexpression
  | evaluatequeryfunction
  ;

graphqueryexpressionaddon
  : DOT queryexpression
  | DOT evaluatequeryfunction
  ;

attributeaccessaddon
  : DOT attributename
  ;

hastypeexpressionaddon
  : HASTYPE_KW JVALUETYPE
  ;
```

5 Auswerter

Der Auswerter bekommt den ASG-Graph vom Parser. Der ASG-Graph enthält alle nötigen Informationen um das Script auszuführen. Die Ausführung erfolgt ähnlich dem Command-Pattern nach GoF [GHJV96].

5.1 Design-Patterns

Am effektivsten wäre es, wenn die Knoten-Klassen mit einer `execute()`-Methode ausgestattet wären und die gemeinsam genutzte Funktionalität in den abstrakten Superklassen zusammengefasst wäre.

Dies ist in *JGraLab* nur bedingt möglich. In der Abbildung 5.1 ist ein Auszug zu sehen, welche Klassen von *JGraLab* generiert werden. Links ist das Metamodel der Knoten. Hier sind `Statement` und `Expression` abstrakt und `IfStatement` und `FunctionCall` konkret. Rechts sind die aus dem Metamodel generierten Schema-Interfaces für *alle* Knoten und Knoten-Klassen, zu erkennen am `Impl`, nur für konkrete Knoten. Diese Knoten-Klassen können, durch Vererbung, mit Methoden erweitert werden, z.B. `execute()`. Leider können abstrakte Knoten nicht mit Methoden erweitern, weil es für sie keine Knoten-Klassen gibt. So müsste der ähnliche Code bei einigen Klassen doppelt auftreten. Z.B. bei allen Schleifen würden sich Code-Fragmente in der `execute()`-Methode wiederholen.

Eine andere Möglichkeit, die in GReQL-Script verwendet wurde, ist die Folgende. Für jede Knoten-Klasse, die ausführbar sein soll, wird eine Interpreter-Klasse erstellt. Im Falle von GReQL-Script ist es jede Knoten-Klasse in der `Statement`-Hierarchie. Für abstrakte Knoten-Klassen werden abstrakte Interpreter-Klassen erstellt. So wird für den Knoten `<NODE>` eine Klasse `<NODE>Interpreter` erstellt. In der Abbildung 5.2 ist ein Teil der `Statement`-Hierarchie mit der dazugehöriger Interpreter-Klassen-Hierarchie zu sehen.

Um die Knoten mit den Interpreter-Klassen zu verbinden, wird eine Vermittlerklasse `Interpreter` verwendet, d.h. über diese Klasse werden Knoten ausgeführt bzw. ausgewertet. Dies entspricht dem *Interpreter-Pattern* in Abbildung 5.3 von GoF [GHJV96]. Dabei wird als Kontext der aktuell bearbeitete ASG-Knoten und das `Interpreter`-Objekt, das den restlichen Kontext enthält, übergeben. Somit ist die `Interpreter`-Klasse gleichzeitig der `Client` und `Context`. In der Abbildung 5.2 sieht man die Klasse `Interpreter` mit den Methoden, `interpret()` und `getResult()`, die weiter an die Interpreter-Klassen delegiert werden. Diese Klasse verwaltet außerdem den Kontext. Kontext ist hier der aktuelle Zustand des `Interpreters`, der momentan bearbeitete Anweisung und die momentanen

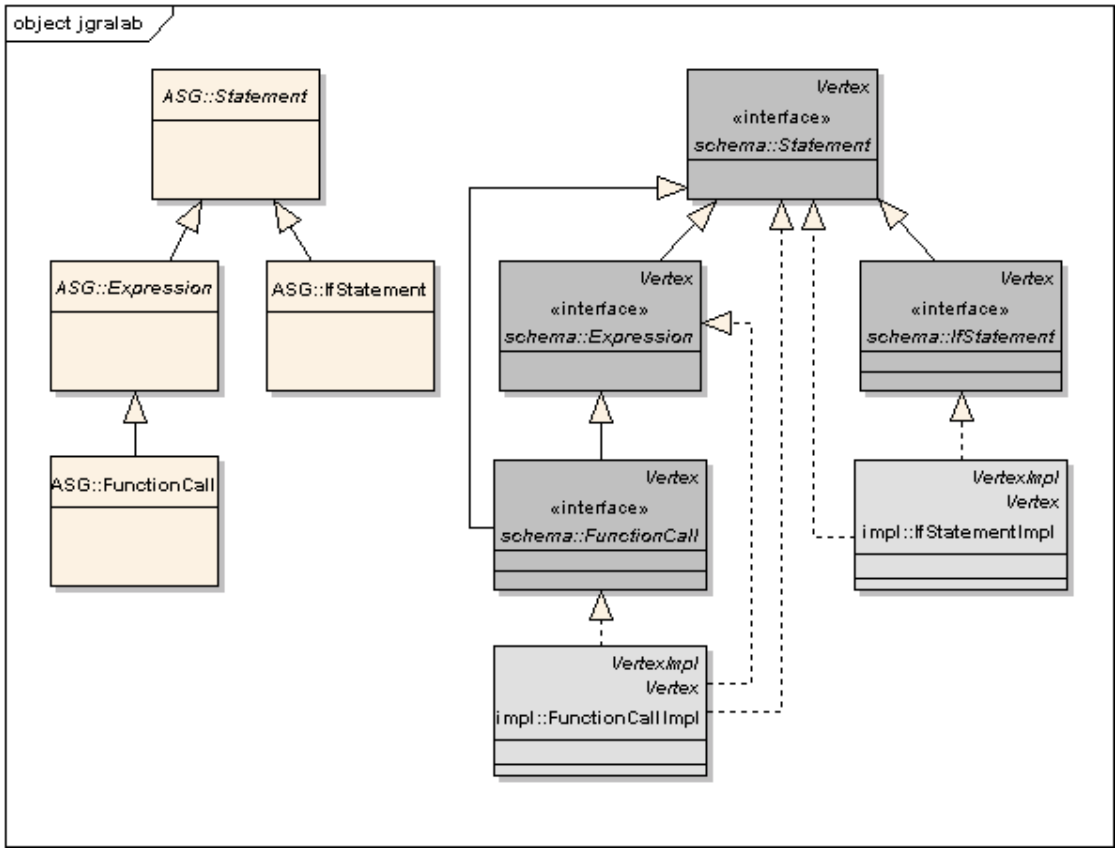

Abbildung 5.1: Von JGraLab generiertes Schema (Auszug)

Variablenwerte. Der Kontext besteht aus den Variablendaten, die im Abschnitt 5.5 behandelt werden, und den Unterbrechungsmarkierungen, über die im Abschnitt 5.3 Näheres zu finden ist.

Die abstrakte Superklasse der Interpreter-Klassen-Hierarchie ist `StatementInterpreter`. Darin ist eine abstrakte Methode `interpret(Statement,Interpreter):void` deklariert. Das `Statement`-Objekt das übergeben wird muss der Statement-Knoten, der ausgeführt werden soll, sein und das `Interpreter`-Objekt beinhaltet alle Daten auf denen operiert wird. Diese Methode wird nicht direkt sondern über das `Interpreter`-Objekt aufgerufen. Die `Interpreter`-Klasse enthält die Methode `interpret(Statement):void`. Ein Knoten wird über diese Methode ausgeführt.

Eine weitere wichtige Klasse in der Interpreter-Klassen-Hierarchie ist die abstrakte Klasse `ExpressionInterpreter`. Darin ist die abstrakte Methode `getResult(Expression, Interpreter):JValue` deklariert. Der Unterschied zwischen der `interpret()`-Methode und der `getResult()`-Methode ist, dass die `getResult()`-Methode ein Rückgabewert vom Typ `JValue` hat. Die `interpret()`-Methode enthält nur den Aufruf von `getResult()` und ist als `final` deklariert. Das Ergebnis von `getResult()` innerhalb von der Methode `interpret()`

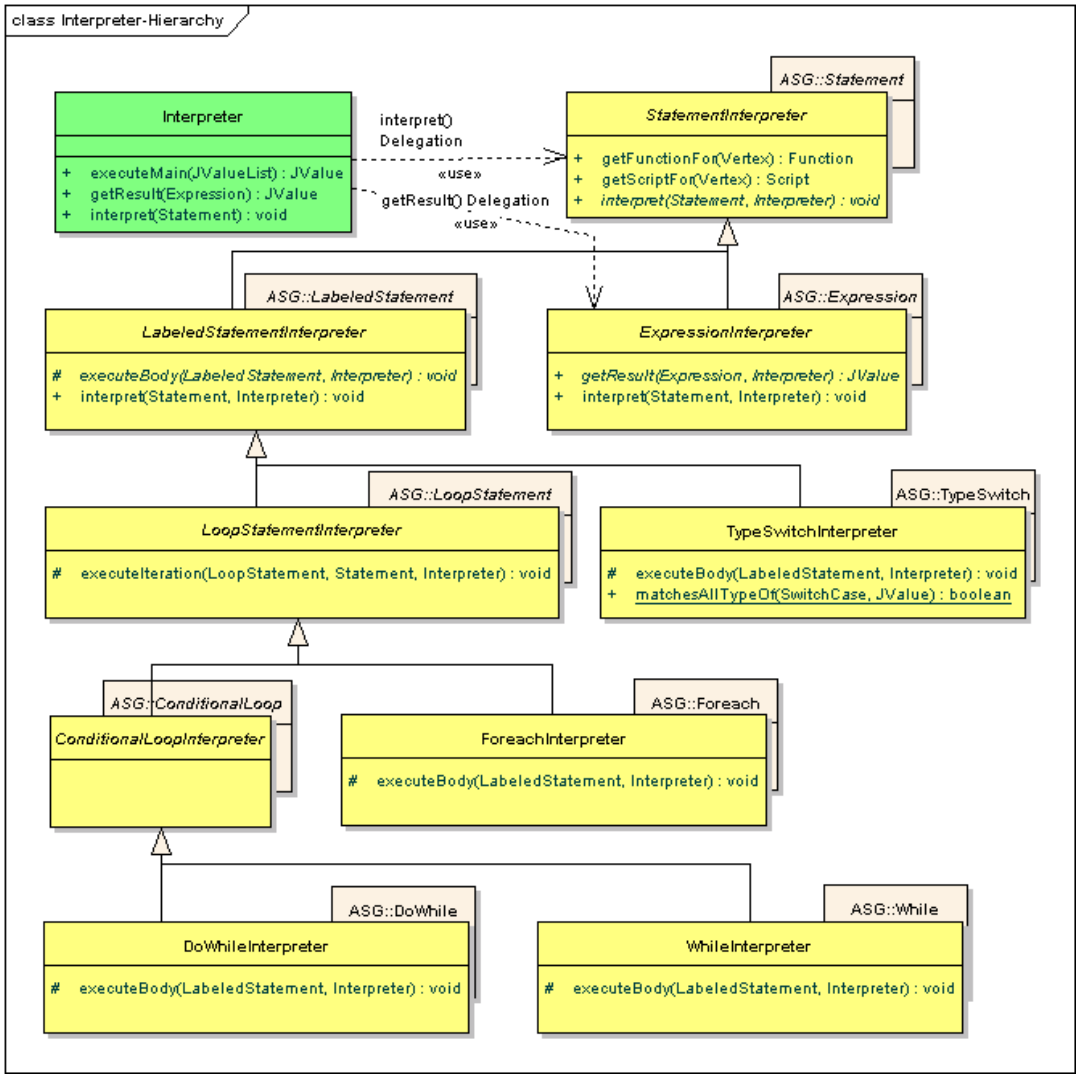

Abbildung 5.2: Interpreter Hierarchie Ausschnitt

in `ExpressionInterpreter` wird verworfen und ist für die Verwendung von `Expressions` als `Statements` vorgesehen, z.B. bei `FunctionCall`. So müssen die Unterklassen von `ExpressionInterpreter` die Methode `getResult()` implementieren und nicht die `interpret()`-Methode. Die `Interpreter`-Klasse enthält die Methode `getResult(Expression):JValue`. Ein `Expression`-Knoten wird über diese Methode ausgewertet.

Die abstrakten Klassen enthalten Code, der sich in den unteren Klassen wiederholen würde. So kümmert sich die Klasse `LabeledStatementInterpreter` um die Behandlung von `breaks`. Wobei die `continues` von der Klasse `LoopStatementInterpreter` behandelt werden.

Im Sequenzdiagramm in der Abbildung 5.4 ist der Anfang der Ausführung zu sehen. Als Erstes holt sich der `Interpreter` den „main“-`Function`-Knoten. Mit die-

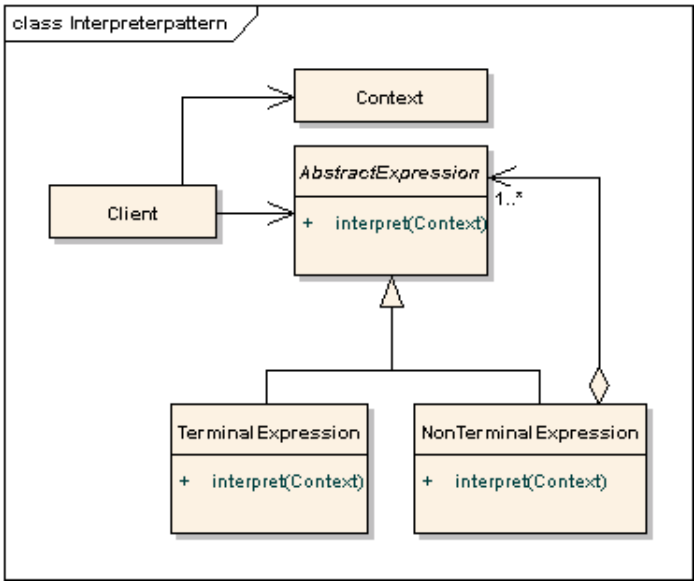

Abbildung 5.3: GoF Interpreter Pattern

sem Knoten verbindet er einen neu erzeugten `FunctionCall`-Knoten. Dann führt der `Interpreter` die eigene Methode `getResult(Expression):JValue` mit dem `FunctionCall` als Parameter auf. Diese Methode delegiert den Aufruf an den `FunctionCallInterpreter`. Ab dieser Stelle wird rekursiv absteigend die Methode `interpret(Statement)` für die enthaltenen `Statement`-Knoten ausgeführt. So wertet ein `FunctionCallInterpreter` zuerst die Parameter aus und führt dann die `interpret(Statement)`-Methode für den enthaltenen `Block` aus. Als weiteres würde der `BlockInterpreter` die `interpret(Statement)`-Methode für die enthaltenen `Statements` aufrufen. Bei `Expressions` wird statt `interpret(Statement)` die Methode `getResult(Expression)` verwendet.

5.2 VariableStack

Bei einer Sprache, die keine Rekursion unterstützt, wäre es ausreichend jedem `Variable`-Knoten einen Wert bzw. eine Referenz zuzuweisen. In GReQL-Script wird Rekursion unterstützt und die Handhabung von Variablen wird in einem etwas komplexerem Stack verwaltet, in der `VariableStack` Klasse. Diese Klasse enthält für jede angefangene Funktion eine weitere Struktur als Stackelement, die die Variablenzuweisungen verwaltet. So werden Variablen der selben Funktion bei einem rekursiven Aufruf nicht überschrieben. In der Abbildung 5.5 ist das Klassendiagramm von `VariableStack` enthalten.

Auch der größte Speicher läuft voll, wenn ein ein rekursiver Aufruf in eine Endlosschleife gerät. Weil es schwierig ist gute Exceptions zu werfen, wenn eine `StackOverflowError`-Ausnahme gefangen wird, wurde die maximale Funktionsaufruftiefe beschränkt, dies

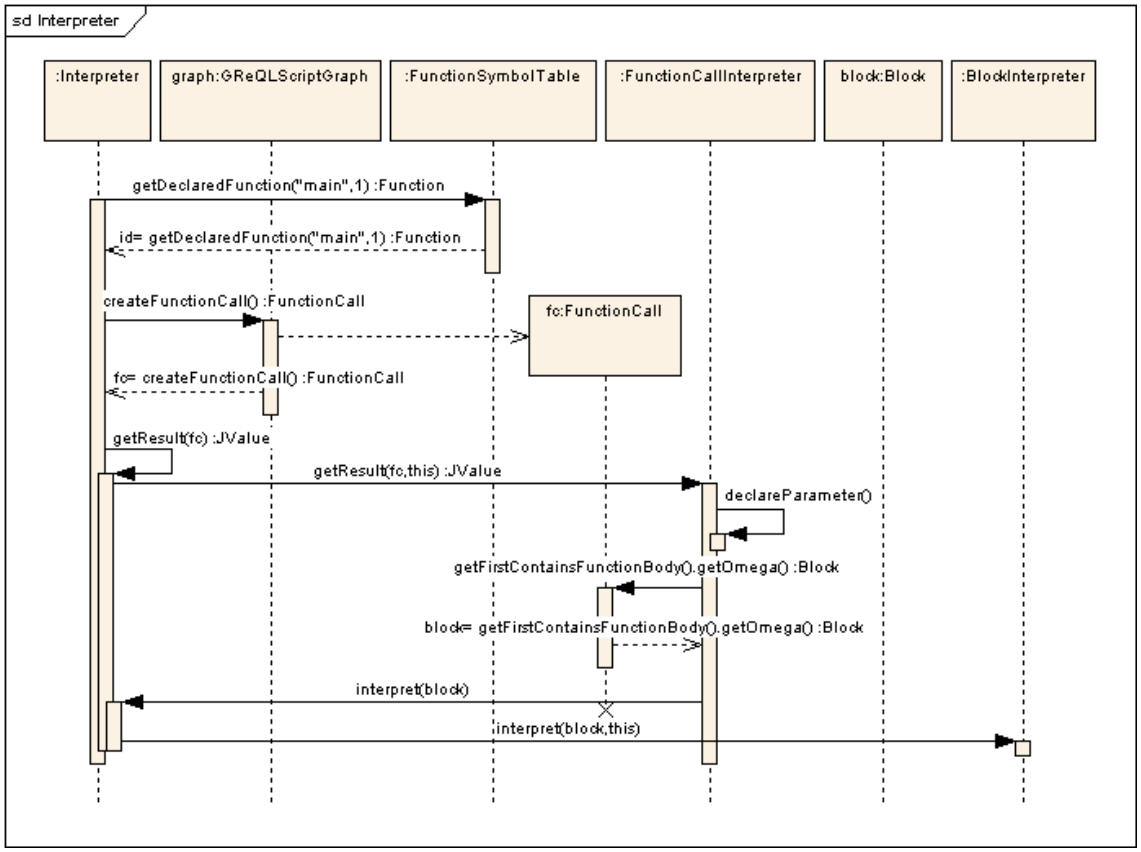

Abbildung 5.4: Interpreter Sequenzdiagramm

wird durch `Variablestack.MAX_DEEP` reguliert. Wenn die Funktionsaufruftiefe diesen Wert erreicht wird eine `StackOverflowException` geworfen. Der Unterschied zu `StackOverflowError` ist, dass bei der `StackOverflowException` die Fehlerposition im Script angezeigt wird und nicht im Interpreter.

GReQL implementiert call-by-reference auf folgende Weise. Im `VariableStack` werden den Variablen `JValueReference`-Objekte zugewiesen. Nach dem Initialisieren kann eine Referenz nicht mehr geändert werden. Es kann nur noch der Wert geändert werden, so wird sicher gestellt dass eine Änderung an einem mit call-by-reference übergebenem Parameter auch an der übergebenen Variablen durchgeführt wird. Bei der Initialisierung kann auch ein `AttributeReference`-Objekt übergeben werden. In diesem Fall sollte man wissen welchen Typ das Attribut annehmen kann. Das Verhalten ist in einem Objektdiagramm in Abbildung 5.6 verdeutlicht. GReQL-Script arbeitet mit zwei Arten von Referenzen: `JValueReference` und `AttributeReference` siehe Abbildung 5.7.

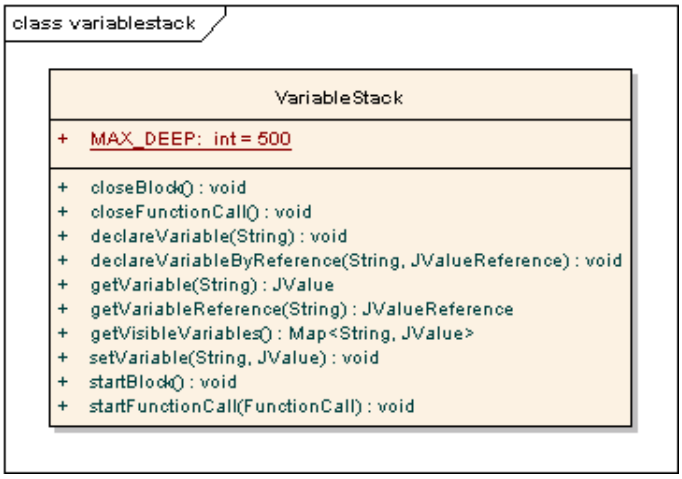

Abbildung 5.5: VariableStack - Klassendiagramm

5.3 Interrupts

Die unterbrechenden Anweisungen `break`, `continue` und `return` markieren den Kontext als unterbrochen. Dazu stellt die Klasse `InterruptContext`, die in Abbildung 5.8 zu sehen ist, folgende Methoden zur Verfügung:

- `setBreakLabel(LabeledStatement):void`: Diese Methode setzt die Markierung auf `BREAK`. Der `LabeledStatement`-Knoten wird für die Abfrage gespeichert, es kann auch `null` übergeben werden.
- `setContinueLabel(LabeledStatement):void`: Diese Methode setzt die Markierung auf `CONTINUE`. Der `LabeledStatement`-Knoten wird für die Abfrage gespeichert, es kann auch `null` übergeben werden.
- `getLabeledStatementToInterrupt():LabeledStatement`: Der gespeicherte `LabeledStatement`-Knoten wird zurückgegeben. Diese Methode setzt die Markierung auf `NORMAL`.
- `setReturnInterrupt(JValue):void`: Diese Methode setzt die Markierung auf `RETURN`. Der `JValue`-Parameter wird für die Abfrage gespeichert, es kann auch `null` übergeben werden.
- `getReturnValue():JValue`: Der gespeicherte `JValue`-Wert wird zurückgegeben. Diese Methode setzt die Markierung auf `NORMAL`.
- `isInterrupted():boolean`: Diese Methode gibt `false` zurück, wenn die Markierung auf `NORMAL` gesetzt ist, sonst gibt sie `true` zurück.
- `getInterruptMark():InterruptState`: Diese Methode gibt den Wert der Markierung zurück.

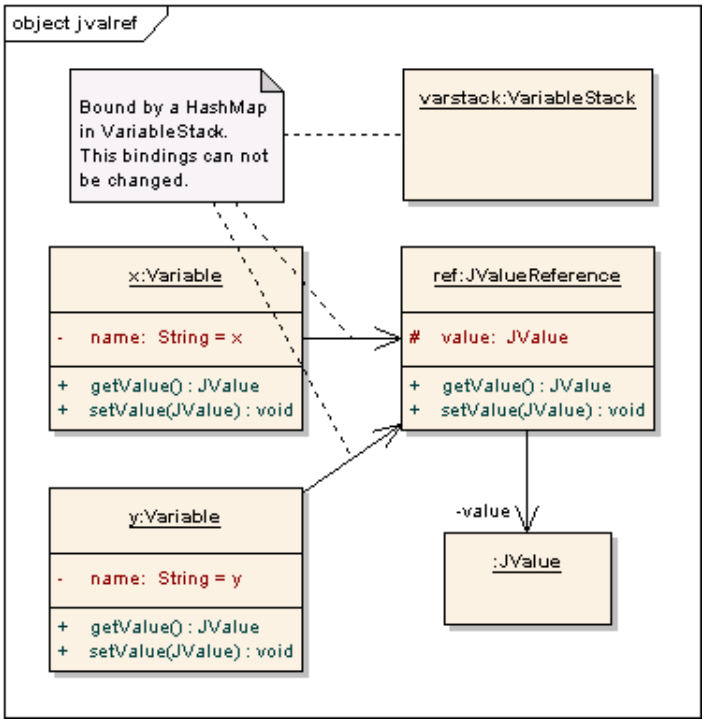

Abbildung 5.6: JValueReference - Objektdiagramm

5.3.1 Auslösen des Interrupted Zustandes

Der Zustand von `InterruptContext` wird von den drei unterbrechenden Anweisungen gesetzt.

- `BreakStatementInterpreter`: ruft die Methode `setBreakLabel(LabeledStatement):void` auf. Als Parameter wird das mit dem `BreakStatement` verbundene `LabeledStatement` übergeben oder `null`, wenn kein `LabeledStatement` vorhanden.
- `ContinueStatementInterpreter`: ruft die Methode `setContinueLabel(LabeledStatement):void` auf. Als Parameter wird das

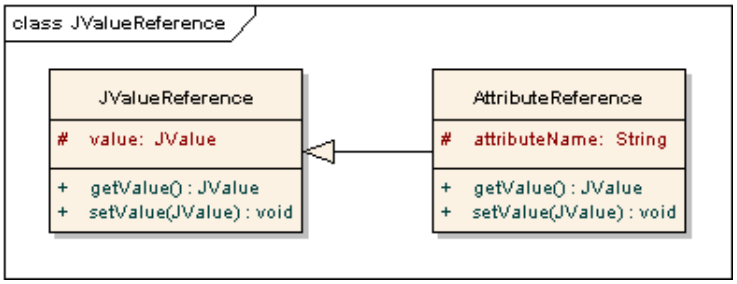

Abbildung 5.7: JValueReference - Klassendiagramm

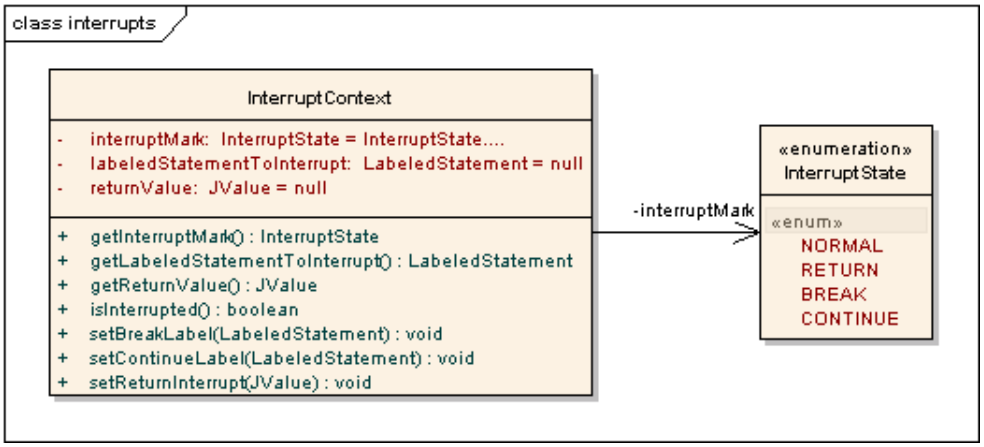

Abbildung 5.8: InterruptContext

mit dem `ContinueStatement` verbundene `LabeledStatement` übergeben oder `null`, wenn kein `LabeledStatement` vorhanden.

- `ReturnStatementInterpreter`: ruft die Methode `setReturnInterrupt(JValue):void` auf. Als Parameter wird das Auswertungsergebnis der `Expression` übergeben, die mit dem `ReturnStatement` verbunden ist, oder `null`, wenn keine `Expression` vorhanden.

Alle drei diese Methoden setzen den Zustand des `InterruptContext` auf Unterbrechen.

5.3.2 Überprüfung des Interrupted Zustandes

Alle `Statements`, die mehr als ein Unter-`Statement` enthalten, müssen vor der Ausführung der Unter-`Statements` prüfen ob die Ausführung nicht unterbrochen werden soll. In folgenden Klassen ist diese Überprüfung mit der Methode `isInterrupted():boolean` eingebaut:

- DoWhileInterpreter
- ForeachInterpreter
- WhileInterpreter
- TypeSwitchInterpreter
- BlockInterpreter

Diese Klassen überprüfen den Zustand des `InterruptContexts` verändern ihn aber nicht.

5.3.3 Auflösen des Interrupted Zustandes

Der Zustand des `InterruptContexts` wird von folgenden Anweisungen auf `NORMAL_MARK` gesetzt.

- `LabeledStatementInterpreter`: ruft die Methode `getLabeledStatementToInterrupt():LabeledStatement` auf. Das zurückgegebene `LabeledStatement` wird mit dem ausgeführtem `LabeledStatement` verglichen. Das ausgeführte `LabeledStatement` wird unterbrochen und wenn das Label ungleich war, wird erneut `setBreakLabel(LabeledStatement):void` aufgerufen, damit das darüber liegende `LabeledStatement` unterbrochen wird.
- `LoopStatementInterpreter`: ruft die Methode `getLabeledStatementToInterrupt():LabeledStatement` auf. Das zurückgegebene `LabeledStatement` wird mit dem ausgeführtem `LabeledStatement` verglichen. Ein Schleifendurchlauf des ausgeführten `LabeledStatement` wird unterbrochen und wenn das Label ungleich war, wird erneut `setContinueLabel(LabeledStatement):void` aufgerufen, damit ein Schleifendurchlauf des darüber liegenden `LabeledStatement`s unterbrochen wird.
- `FunctionCallInterpreter`: ruft die Methode `getReturnValue():JValue` auf. Das zurückgegebene `JValue`-Objekt wird von der `getResult(Expression,Interpreter):JValue` zurückgegeben.

5.3.4 Alternative mit Throwable

In einem Block und in den Schleifen muss mit der oben gezeigten Methode nach jeder Anweisung geprüft werden, ob die Ausführung nicht unterbrochen werden soll. Diese Überprüfung ist schnell, aber mit einer anderen Methode wäre sie nicht nötig. Wenn die Java-Exceptions dafür verwendet werden, muss nur ein `catch`-Block an der richtigen Stelle sein und mittels der `Exception`s bzw. mit `Throwable` kann man auch Objekte, wie z.B. einen `return`-Wert, gekapselt nach oben in der Aufrufhierarchie weiter reichen. Beispiel: Der `ReturnInterpreter` wertet seine `Expression` aus und kapselt sie in ein Objekt, das von `Throwable` erbt, und wirft dieses als `Exception`. Im `FunctionCallInterpreter` ist die Ausführung des `Block`s in einem `try-catch`-Block eingebettet. Dort wird das geworfene Objekt gefangen und der darin enthaltene Wert als Rückgabewert zurückgegeben.

Vorteile:

- Die anderen Klassen sind kaum betroffen, sie müssen nur ein `try-finally`-Block verwenden, wenn sie noch etwas nach der Ausführung der inneren Element erledigen müssen.
- Es muss nicht nach jeder Anweisung geprüft werden ob unterbrochen werden soll.
- Es muss weniger Quellcode geschrieben werden.

Nachteile:

- Es muss beim Werfen des `Throwable`-Objektes der Stacktrace eingesammelt werden.

Welche Methode effektiver ist hängt zum einen davon ab wie viele Anweisungen bis zu einem Interrupt ausgeführt werden und zum anderen in welcher Aufruftiefe der Interrupt ausgelöst wird, das beeinflusst die Zeit der Stackeinsammlung bei `Throwable`. Nach einem einfachen Test stellte sich heraus, dass bei einer Aufruftiefe von 10 Methoden das Werfen eines `Throwable`-Objektes in etwa so lange dauert wie 700 boolesche Vergleiche. Das bedeutet diese Methode würde sich nur dann lohnen, wenn durchschnittlich mehr als 700 Anweisungen bis zu einem Interrupt ausgeführt werden. In GReQL-Script ist es eher nicht der Fall. Deshalb wird die Methode mit dem Booleschen Testen verwendet.

6 Testen

In diesem Abschnitt wird erläutert, wie GReQL-Script getestet wurde. Man sollte Grundkenntnisse von JUnit [jun07] besitzen um den Folgenden Abschnitt verstehen zu können.

Für das Testen wurde *T*est & *P*erformance *T*ools *P*latform (TPTP) [ecl07b] mit JUnit [jun07] verwendet.

6.1 Teststrategie

Es wird automatisiert nach dem *Blackbox*-Verfahren getestet. Es wird ein Script geschrieben, das möglichst nur die zu testenden Anweisungen verwendet. Wenn nach der Spezifikation erwartet wird, dass dieses Script fehlerfrei ist, dann wird eine Datei mit der erwarteten Ausgabe erstellt. Falls dieses Script nicht ausführbar sein soll, wird in der Testmethode, die `Exception` definiert, die erwartet wird. Für jedes solches Script wird eine Test-Methode, die das Script ausführt. Bei den Scripts, die ausgeführt wurden, wird die tatsächliche Ausgabe mit der erwarteten Ausgabe verglichen. Wenn eine `Exception` erwartet wurde, wird genau diese abgefangen und alles Andere führt zum Scheitern des Tests.

Es ist in manchen Fällen nicht möglich die Ausgabe exakt zu beschreiben, z.B. bei Ausgabe des Graphen in eine Bild-Datei. In diesen Fällen werden manuelle Tests durchgeführt. Dabei wird ein Testscript automatisch ausgeführt und eine zusätzliche Information ausgegeben, die dann überprüft werden muss.

7 Verwendung von GReQL-Script

In diesem Abschnitt wird gezeigt, wie GReQL-Scripte ausgeführt werden. Hierbei sollte Java auf alle nötigen Klassen zugreifen können, z.B. per *CLASSPATH*-Variable.

7.1 Kommandozeile

In der Kommandozeile kann ein Script nur von einer Datei ausgeführt werden. Der Befehl dazu lautet:

```
java GReQLScript [options] -f greqlscriptfile [scriptargs]
```

oder wenn man die *jar*-datei verwendet

```
java -jar greqlscript.jar [options] -f greqlscriptfile [scriptargs]
```

Wobei `[options]` die Optionen sind und `[scriptargs]` die Scriptargumente. Folgende Optionen können verwendet werden.

Option	**Auswirkung**
`-ea`	schaltet Assertions ein Voreinstellung: aus
`-md <size>`	setzt die maximale Funktionsaufruftiefe Voreinstellung: 500
`-ostd <file>`	leitet die Standardausgabe in eine Datei um
`-oerr <file>`	leitet die Standardfehlerausgabe in eine Datei um
`-sg`	speichert den ASG erzeugt die Dateien ¡filename¿.tg, ¡filename¿.dot und ¡filename¿.png

7.2 Ausführen

Beliebige Dateien, deren Inhalt der GReQL-Script Syntax entspricht können ausgeführt werden. Ein “Hello, World!”-Script sieht folgendermaßen aus.

Listing 7.1: helloworld.greqlscript

```
main(args){
  var name;
```

```
  name=`"World"`;
  if(`count(args)>0`)
    name = `args[0]`;
  println(`"Hello, "+name+"!"`);
}
```

Die Ausführung des obigen Scripts mit Parameter erfolgt mit diesem Befehl:

```
java -jar greqlscript.jar -f helloworld.greqlscript GReQL-World
```

Die Ausgabe des Scripts ist:

```
Hello, GReQL-World!
```

7.3 Java

Scripte können auch direkt in Java ausgeführt werden. In Java wird zuerst ein `GReQLScriptOptions`-Objekt benötigt. Dieses kann mit einer Fabrikmethode erzeugt werden. Als Initialparameter muss das Script entweder als ein `String` oder als ein `File` übergeben werden. Dann kann man weitere Optionen mit den Methoden einstellen.

Methode	**Auswirkung**
`setAssertionsEnabled(boolean)`	schaltet Assertions ein/aus
`setMaxDeep(int)`	setzt die maximale Funktionsaufruftiefe
`setStdOutToFile(String)`	leitet die Standardausgabe in eine Datei um
`setStdErrToFile(String)`	leitet die Standardfehlerausgabe in eine Datei um
`setSaveGraph(boolean)`	speichert den ASG

Danach erstellt man ein `GReQLScript`-Objekt, mit den Optionen als Parameter, und führt es mit der Methode `execute()` oder `execute(String[])` aus. Die Ausführung in Java sieht dann so aus:

Listing 7.2: JavaCode für GReQLScript

```
import greqlscript.parser.*;
import greqlscript.exceptions.*;
...
GReQLScriptOptions opts = GReQLScriptOptions
                .getOptionsForString(
"main(args){"+
"  var name;"+
"  name=`"World"`;"+
"  if(`count(args)>0`)"+
"    name = `args[0]`;"+
"  println(`"Hello, "+name+"!"`);"+
```

```
"}"
);
GReQLScript scr = new GReQLScript(opts);
try {
    String scrargs = {"GReQL-World"};
        scr.execute(scrargs);
} catch (Exception e) {
        e.printStackTrace();
}
```

8 Fazit

In dieser Arbeit wurden die gestellten Anforderungen vom Interpreter oder von den definierten aufrufbaren Funktionen erfüllt und getestet.

Die Scriptsprache kann verwendet werden und ist durch Funktionen erweiterbar. Der Interpreter verwendet die *Java-Reflection-API* um Funktionen aufrufen zu können. Auf diese Weise ist es einfacher neue Funktionen zu definieren. Falls GReQL-Script öfter verwendet wird oder die Performanz nicht mehr ausreicht, wird die Implementierung eines Zwischencompilers empfohlen, da GReQL-Scripte seltener geändert werden müssen im Gegensatz zu GReQL-Anfragen. Der Zwischencompiler würde aus einem GReQL-Script eine `java`-Datei erzeugen und diese weiter zu `class`-Datei compilieren. Dann ist eine Benutzung von *Java-Reflection-API* nicht notwendig und das Script kann effizienter ausgeführt werden.

In GReQL-Script sind nur wenige Operatoren, `hastype`-Operator und der Punktoperator (`.`), vorhanden, weil die Operatoren in GReQL-Anfragen verwendet werden. Diese Operatoren haben die gleiche *Operatorpräzedenz*. Wenn weitere Operatoren eingeführt werden sollen, muss auch eine Präzedenzhierarchie eingeführt werden.

Das Ziel der Studienarbeit war die Arbeit mit GReQL-Anfragen zu erleichtern und eine einfach zu benutzende Scriptsprache zu entwickeln. Dieses Ziel wurde nach der Meinung des Autors erreicht.

A Funktionsreferenz

Hier werden die vordefinierten Funktionen dokumentiert. Wie man eigene Funktionen deklariert, wird im Abschnitt 3.6 erklärt. Der Aufbau der Funktionsdokumentation sieht folgendermaßen aus.

Die Überschrift ist der Funktionsname. Danach folgen Signatur, Parameterliste, Rückgabe und die Beschreibung.

Signatur:	Die Signatur besteht aus dem Funktionsnamen und der Parameterliste.
Parameter:	In diesem Abschnitt werden die typisierten Parameter erläutert.
Rückgabe:	Hier wird der Rückgabetyp und der Rückgabewert erläutert.

Die Funktionsbeschreibung beschreibt das, was die Funktion macht.

A.1 Besondere Funktionen

A.1.1 evaluateQuery

Signatur:	`evaluateQuery(queryfile)`
Parameter:	`STRING queryfile` - Datei mit der GReQL-Anfrage
Rückgabe:	`JVALUE` - gibt das Ergebnis der Anfrage zurück

führt die GReQL-Anfrage aus der Datei, in Verbindung mit einem optionalem Graph, aus und gibt das Ergebnis der Anfrage zurück oder gibt einen Fehler aus bei Misserfolg. Z.B. `graph.evaluateQuery(`"fwr1.greql"`);`

A.2 greqlscript.lib.Standard

Diese Klasse stellt einige Funktionen, die in GReQLScript verwendet werden können, zur Verfügung.

A.2.1 isNull

Signatur:	`isNull(arg)`
Parameter:	`JVALUE arg` - die Variable die geprüft wird
Rückgabe:	`BOOLEAN` - gibt `true` zurück, wenn `arg` `null` ist

gibt `true` zurück, wenn das übergebene Argument `arg` `null` ist, sonst wird `false` zurück gegeben.

A.2.2 print

Signatur:	`print(arg)`
Parameter:	`JVALUE arg` - die Variable die auszugeben ist
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt die Variable `arg` auf der Standardausgabe aus und gibt `null` zurück

A.2.3 println

Signatur:	`println(arg)`
Parameter:	`JVALUE arg` - die Variable die auszugeben ist
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt die Variable `arg` mit nachfolgendem Zeilenumbruch auf der Standardausgabe aus und gibt `null` zurück

A.2.4 println

Signatur:	`println()`
Parameter:	
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt einen Zeilenumbruch auf der Standardausgabe aus und gibt `null` zurück

A.2.5 errprint

Signatur:	`errprint(arg)`
Parameter:	`JVALUE arg` - die Variable die auszugeben ist
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt die Variable `arg` auf der Standardfehlerausgabe aus und gibt `null` zurück

A.2.6 errprintln

Signatur:	`errprintln(arg)`
Parameter:	`JVALUE arg` - die Variable die auszugeben ist
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt die Variable `arg` mit nachfolgendem Zeilenumbruch auf der Standardfehlerausgabe aus und gibt `null` zurück

A.2.7 errprintln

Signatur:	`errprintln()`
Parameter:	
Rückgabe:	`NULL` - gibt immer `null` zurück

gibt einen Zeilenumbruch auf der Standardausgabe aus und gibt `null` zurück

A.2.8 loadGraphFromTG

Signatur:	`loadGraphFromTG(filename)`
Parameter:	`STRING filename` - Dateiname des zu ladenden Graphen
Rückgabe:	`GRAPH` - gibt den geladenen Graph zurück

lädt einen Graph aus der Datei `filename` oder gibt `null` zurück, wenn nicht erfolgreich.

A.2.9 createGraphFromSchema

Signatur:	`createGraphFromSchema(schemaClassValue, graphClassValue, instanceName, vMax, eMax)`
Parameter:	`STRING schemaClassValue` - Name der Schemaklasse des zu erzeugenden Graphen `STRING graphClassValue` - Name der Graphklasse des zu erzeugenden Graphen `STRING instanceName` - Name der erzeugenden Grapheninstanz `INTEGER vMax` - maximal erlaubte Anzahl an Knoten im Graph `INTEGER eMax` - maximal erlaubte Anzahl an Kanten im Graph
Rückgabe:	`GRAPH` - gibt den erzeugten Graph zurück

erzeugt aus dem Schema `schemaClassValue` einen Graph der Klasse `graphClassValue` mit dem Instanznamen `instanceName` oder gibt `null` zurück, wenn nicht erfolgreich.

A.2.10 saveGraphToTG

Signatur:	`saveGraphToTG(filename, graph)`
Parameter:	`STRING filename` - Dateiname wo der Graph gespeichert werden soll
Rückgabe:	`BOOLEAN` - gibt bei erfolgtem Speichern `true` zurück

speichert den Graph `graph` in der Datei `filename` und gibt `true` zurück oder gibt `false` zurück, wenn nicht erfolgreich.

A.2.11 createVertex

Signatur:	`createVertex(graph, vertexClass)`
Parameter:	`GRAPH graph` - Graph in dem der Knoten erzeugt werden soll `STRING vertexClass` - Name der Knotenklasse des zu erzeugenden Knoten
Rückgabe:	`VERTEX` - gibt den erzeugten Knoten zurück

erzeugt einen Knoten der Klasse `vertexClass` im Graph `graph` oder gibt `null` zurück, wenn nicht erfolgreich.

A.2.12 createEdge

Signatur:	`createEdge(graph, edgeClass, alphaVertex, omegaVertex)`
Parameter:	`GRAPH graph` - Graph in dem die Kante erzeugt werden soll `STRING edgeClass` - Name der Kantenklasse der zu erzeugenden Kante `VERTEX alphaVertex` - der Anfangsknoten der zu erzeugenden Kante `VERTEX omegaVertex` - der Endknoten der zu erzeugenden Kante
Rückgabe:	`EDGE` - gibt die erzeugte Kante zurück

erzeugt eine Kante der Klasse `edgeClass` im Graph `graph` vom Knoten `alphaVertex` zum Knoten `omegaVertex` oder gibt `null` zurück, wenn nicht erfolgreich.

A.2.13 getType

Signatur:	`getType(jValue)`
Parameter:	`JVALUE jValue` - Variable deren Typ ermittelt werden soll
Rückgabe:	`STRING` - gibt den Typ von `jValue` als String zurück

gibt den Typ vom übergebenem Parameter `jValue` als String zurück.

A.2.14 deleteVertex

Signatur:	`deleteVertex(vertex)`
Parameter:	`VERTEX vertex` - Knoten, der gelöscht werden soll
Rückgabe:	`NULL` - gibt immer `null` zurück.

löscht den angegebenen Knoten `vertex` im Graph und gibt `null` zurück.

A.2.15 deleteEdge

Signatur:	`deleteEdge(edge)`
Parameter:	`EDGE edge` - Kante, die gelöscht werden soll
Rückgabe:	`NULL` - gibt immer `null` zurück.

löscht die angegebene Kante `edge` im Graph und gibt `null` zurück.

A.2.16 printhtml

Signatur:	`printhtml(value, pathToHTMLFile)`
Parameter:	`JVALUE value` - JValue, das ein Serialiseable Objekt enthält `STRING pathToHTMLFile` - Dateiname wo das gekapselte Objekt gespeichert werden soll
Rückgabe:	`BOOLEAN` - gibt bei erfolgter Speicherung `true` zurück.

speichert das JValue-Objekt `value` in der Datei `pathToHTMLFile` im HTML-Format und gibt `true` zurück oder gibt `false` zurück, wenn nicht erfolgreich.

A.2.17 store

Signatur:	`store(value, pathToXMLFile, graph)`
Parameter:	`JVALUE value` - JValue, das gespeichert werden soll `STRING pathToXMLFile` - Dateiname wo das gekapselte Objekt gespeichert werden soll `GRAPH graph` - Graph dem alle Objekte im JValue `value` zugehören
Rückgabe:	`BOOLEAN` - gibt bei erfolgter Speicherung `true` zurück.

speichert das JValue-Objekt `value`, das dem Graph `graph` zugehört, in der Datei `pathToXMLFile` und gibt `true` zurück oder gibt `false` zurück, wenn nicht erfolgreich.

A.2.18 load

Signatur:	`load(pathToXMLFile, graph)`
Parameter:	`STRING pathToXMLFile` - Dateiname woher das Objekt geladen werden soll `GRAPH graph` - Graph in den das JValue geladen wird
Rückgabe:	`JVALUE` - gibt das geladene JValue-Objekt zurück.

ladet ein JValue-Objekt aus der Datei `pathToXMLFile` in den Graph `graph` und gibt es zurück oder gibt `null` zurück, wenn nicht erfolgreich.

A.2.19 exception

Signatur:	`exception(message)`
Parameter:	`STRING message` - Nachricht der Exception
Rückgabe:	

wirft eine Exception mit der `message` als Nachricht.

A.2.20 basedir

Signatur:	`basedir()`
Parameter:	
Rückgabe:	`STRING` - das aktuelle Verzeichnis zurück.

gibt das aktuelle Verzeichnis als `STRING` zurück.

A.2.21 scriptdir

Signatur:	`scriptdir()`
Parameter:	
Rückgabe:	`STRING` - das aktuelle Verzeichnis zurück.

gibt das aktuelle Verzeichnis als `STRING` zurück.

A.2.22 setAlpha

Signatur:	`setAlpha(edge, vertex)`
Parameter:	`NULL` - gibt immer `null` zurück
Rückgabe:	

setzt den `alpha`-Knoten der Kante `edge` auf den Knoten `vertex` und gibt null zurück.

A.2.23 setOmega

Signatur:	`setOmega(edge, vertex)`
Parameter:	`NULL` - gibt immer `null` zurück
Rückgabe:	

setzt den `omega`-Knoten der Kante `edge` auf den Knoten `vertex` und gibt null zurück.

B EBNF

An dieser Stelle wird die Syntax von GReQL-Script in einer erweiterten EBNF beschrieben. Lexer-Regeln sind großgeschrieben. Parser-Regeln sind kleingeschrieben.

```
MetaSyntax:
X = ... ;      X ist ...
[ X ]          0 oder 1 Vorkommen von X
{ X }          kein oder beliebig viele Vorkommen von X
X | Y          X oder Y
'x'            x ist ein Zeichen in der Eingabe
 .             ein beliebiges Zeichen in der Eingabe
(.)\('x'|'y') ein beliebiges Zeichen in der Eingabe außer x oder y
"xy"           ist eine Zeichenfolge in der Eingabe
( )            zur Klammerung
```

```
1  WHITESPACE = ' ' | '\t' | '\f' | '\r' | '\n' ;
2
3  COMMENT = "//" {(.)\('\r'|'\n')} ('\r'|'\n') ;
4
5  MLCOMMENT = "/*" {(.)\("*/")} "*/" ;
6
7  GREQLQUERY = '`' {(.)\('`')} '`' ;      //` als Delimiter
8
9  IDENT = (('a'..'z')|('A'..'Z')) {('a'..'z')|('A'..'Z')|('0'..'9')|'_'} ;
10
11 script =
12   {importline} functiondeclaration {functiondeclaration} ;
13
14 importline =
15   ("import" | "importjava") pathgreqlquery ';' ;
16
17 functiondeclaration =
18   functionid '(' [parameterdeclaration {',' parameterdeclaration}] ')'
       block ;
19
20 functionid =
21   IDENT ;
22
23 parameterdeclaration =
24   ["var"] variable ;
25
26 variable =
27   IDENT ;
28
29 block =
```

```
30   '{' {statement} '}' ;
31
32 statement =
33   block ;
34   | variabledeclaration ';'
35   | assignment
36   | ifstatement
37   | labeledstatement
38   | returnstatement
39   | breakstatement
40   | continuestatement
41   | assertstatement
42   | expression ';'
43   | emptystatement ;
44
45 labeledstatement =
46   [label ':'] (loopstatement | typeswitchstatement) ;
47
48 label =
49   IDENT ;
50
51 variabledeclaration =
52   singlevariabledeclaration {',' variable} ;
53
54 singlevariabledeclaration =
55   "var" variable ;
56
57 assignment =
58   lvalue '=' expression ';' ;
59
60 lvalue =
61   variable
62   | attributeaccess ;
63
64 attributeaccess =
65   expression '.' attributename ;
66
67 attributename =
68   IDENT ;
69
70 ifstatement =
71   "if" '(' booleanexpression ')' statement ["else" statement] ;
72
73 typeswitchstatement =
74   "typeswitch" '(' expression ')' '{' switchcase {switchcase} [defaultcase
       ] '}' ;
75
76 switchcase =
77   "case" jvaluetype {',' jvaluetype} ':' statement ;
78
79 jvaluetype =
80   "ATTRIBUTEDELEMENT"
```

```
81    | "ATTRIBUTEDELEMENTCLASS"
82    | "BOOLEAN"
83    | "CHARACTER"
84    | "DOUBLE"
85    | "EDGE"
86    | "ENUMVALUE"
87    | "GRAPH"
88    | "INTEGER"
89    | "COLLECTION"
90    | "RECORD"
91    | "LONG"
92    | "OBJECT"
93    | "PATH"
94    | "PATHSYSTEM"
95    | "STRING"
96    | "VERTEX" ;
97
98  defaultcase =
99    "default" ':' statement ;
100
101 loopstatement =
102   (foreachloop | whileloop | dowhileloop) ;
103
104 foreachloop =
105   "foreach" '(' singlevariabledeclaration ':' collectionexpression ')'
        statement ;
106
107 whileloop =
108   "while" '(' booleanexpression ')' statement ;
109
110 dowhileloop =
111   "do" statement "while" '(' booleanexpression ')' ';' ;
112
113 returnstatement =
114   "return" [expression] ';' ;
115
116 breakstatement =
117   "break" [label] ';' ; //nur in labeledstatement erlaubt
118
119 continuestatement =
120   "continue" [label] ';' ; //nur in loopstatement erlaubt
121
122 assertstatement =
123   "assert" booleanexpression [':' stringexpression]';' ;
124
125 expression =
126   graphqueryexpression
127   | lvalue
128   | functioncall
129   | hastypeexpression ;
130
131 graphqueryexpression =
```

```
132   [graph '.'] (queryexpression | evaluatequeryfunction) ;
133
134 queryexpression =
135   GREQLQUERY ;
136
137 evaluatequeryfunction =
138   "evaluateQuery" '(' pathgreqlquery ')' ;
139
140 functioncall =
141   functionid '(' [expression {',' expression}] ')' ;
142
143 hastypeexpression =
144   expression "hastype" jvaluetype ;
145
146 emptystatement =
147   ';' ;
148
149 graph =
150   expression ;
151
152 pathgreqlquery =
153   queryexpression ;
154
155 booleanexpression =
156   expression ;
157
158 collectionexpression =
159   expression ;
160
161 stringexpression =
162   expression ;
```

Literaturverzeichnis

[Apa06] Apache. ANT 1.7.0, 2006. Software.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey Ullman. *Compilerbau Teil 1*. Addison-Wesley (Deutschland) GmbH, 1988.

[Bil06] Daniel Bildhauer. Ein Interpreter für GReQL 2 - Entwurf und prototypische Implementation. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[ecl07a] eclipse.org. Eclipse 3.3, 2007. Software.

[ecl07b] eclipse.org/tptp. TPTP 4, 2007. Software.

[EKRW02] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO — Generic Understanding of Programs An Overview. Electronic Notes in Theoretical Computer Science 72 No.2, 2002.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster*. Addison-Wesley (Deutschland) GmbH, 1996.

[jun07] junit.org. JUnit 4, 2007. Software.

[Kah06] Steffen Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[Kul01] Bernt Kullbach. Command Line GReQL (CLG). Benutzerhandbuch. CLG-Version 1.0. Projektbericht 3/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

[Ltd07] Sparx Systems Ltd. Enterprise architect 6.5, 2007. Software.

[Mar06] Katrin Marchewka. GReQL 2. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.

[Pm05] Terence Parr and many more. *ANTLR Reference Manual*. www.antlr2.org, 2005.

[Sax07] Saxonica. Saxon-B 8.9, 2007. Software.