FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementierung der Graphdatenbankabfragesprache G-CORE in Gremlin-Groovy

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Rainer Zachmann
Matrikelnummer 01526652

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Ass. Dipl.-Ing. Dr.techn. Sebastian Skritek

Wien, 11. August 2019

Rainer Zachmann   Sebastian Skritek

Erklärung zur Verfassung der Arbeit

Rainer Zachmann

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. August 2019

Rainer Zachmann

Kurzfassung

Graphdatenbanken können zur Speicherung semistrukturierter Daten eingesetzt werden. Im Gegensatz zu relationalen Datenbanken gibt es bisher keine standardisierte Abfragesprache für jene Graphsysteme.

G-CORE ist eine neue, deklarative Graphdatenbankabfragesprache. In dieser Bachelorarbeit wird diese Sprache in Gremlin – einer funktionalen Abfragesprache – implementiert.

Zuerst wird G-CORE anhand mancher Beispiele analysiert. Dazu wird unter anderem ein neuer Beispielgraph vorgestellt. Auch bisherige Arbeiten werden genau studiert und berücksichtigt.

Mit dieser Arbeit konnte ein modulares Programm entwickelt werden, welches G-CORE-Abfragen einlesen, in einen abstrakten Syntaxbaum umwandeln und schließlich als Gremlin-Abfrage ausgeben kann.

Während der Implementierung wurden Inkompatibilitäten zwischen diesen Graphdatenbankabfragesprachen festgestellt und erörtert. Es wurden auch Probleme und Verbesserungsvorschlägen zur Sprache G-CORE gefunden.

Obwohl das Programm zu jeder gültigen Eingabe einen Syntaxbaum konstruiert, sind einige mögliche Fälle in der Gremlin-Übersetzung nicht implementiert worden.

Abstract

Graph databases are used to store semi-structured data. Contrary to relational databases there is no standard query language for those graph systems so far.

G-CORE is a new, declarative query language which operates on graph databases. It is implemented in Gremlin—a functional query language—within this Bachelor's thesis.

The G-CORE language is analysed and some examples are shown. Incidentally, a new example graph is presented. Also existing literature is studied thoroughly and taken into account.

In this thesis a modular computer program was developed, which reads in G-CORE queries, transforms them to an abstract syntax tree and finally outputs them as Gremlin queries.

During this implementation incompatibilities between these graph database query languages as well as problems and possible improvements in G-CORE were found and discussed.

Although the program constructs a syntax tree for every valid input, there are still several possible cases in which the Gremlin translation has not been implemented.

Inhaltsverzeichnis

Kurzfassung v

Abstract vii

Inhaltsverzeichnis ix

1 Einleitung 1
1.1 Motivation . . . 1
1.2 Aufgabenstellung . . . 2
1.3 Ziel der Arbeit . . . 2
1.4 Herangehensweise . . . 3
1.5 Strukturierung dieser Arbeit . . . 3

2 Graphdatenbanken und Abfragesprachen 5
2.1 Graphmodelle . . . 5
2.2 Abfragesprachen . . . 5
2.3 Gremlin-Groovy . . . 6
2.4 Abfragen in G-CORE . . . 7
2.5 Kritik an G-CORE . . . 8

3 Inkompatibilitäten und nicht implementierte Fälle 11
3.1 Graphmodellunterschiede . . . 11
3.2 Abfragen auf mehreren Graphen . . . 11
3.3 Ergebnisdatentypen . . . 12
3.4 Ausdrücke . . . 12
3.5 Nicht implementierte Fälle . . . 13

4 Implementierung 15
4.1 Gestaltung . . . 15
4.2 Lexer . . . 15
4.3 Parser . . . 16
4.4 Übersetzung in Gremlin . . . 16

5 Kritische Reflexion und Zusammenfassung 21

5.1 Vergleich mit verwandten Arbeiten 21
5.2 Diskussion offener Probleme 21
5.3 Zusammenfassung und Ausblick 22

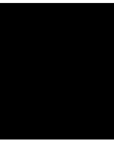
Abbildungsverzeichnis **23**

Tabellenverzeichnis **23**

Literaturverzeichnis **25**

Quelltext **27**
lexer.l 27
parser.y 29
common.h 49
common.c 50
ast.h 52
ast.c 55
gremlin.h 64
gremlin.c 64

KAPITEL 1



Einleitung

1.1 Motivation

Datenbanken werden in Computersystemen zur Speicherung verschiedenartiger Daten verwandt. Je nach Anwendungsfall und der Struktur dieser Daten sind unterschiedliche Datenbankmodelle dazu am besten geeignet.

Für semistrukturierte Daten – also solche, in denen nicht jeder Datensatz dieselben Eigenschaften hat – sind etwa relationale Datenbanken weniger geeignet, als dokumentenorientierte oder die hier eingesetzten graphbasierten Modelle.

Die Modellierung semistrukturierter Daten als Graph erlaubt Abfragen komplexer Zusammenhänge mittels bekannter Algorithmen. Obzwar die Knoten und Kanten sowie deren Eigenschaften auch in relationalen Tabellen gespeichert werden könnten, ermöglicht die graphorientierte Sichtweise eine relativ einfache Formulierung etwa von Zusammenhangskomponenten oder Pfaden.

Während für relationale Datenbanken mit SQL eine weitverbreitete standardisierte Abfragesprache in Gebrauch ist, nützen Graphdatenbanken bisher inkompatible, höchst unterschiedliche Sprachen, die auf verschiedenen Graphmodellen beruhen. In Tabelle 1.1 werden als Beispiel einfache Abfragen in SPARQL [13], Cypher [10], G-CORE [2] und Gremlin [11] gegenübergestellt.

Die Datenbankabfragen in der Tabelle geben jeweils eine Liste der Personen eines vordefinierten Graphen zurück – das heißt, all jene Knoten, denen der Bezeichner `Person` zugeordnet ist. Schon dieses Minimalbeispiel zeigt die diversen Ansätze zur Lösung desselben Problems.

G-CORE ist zwar nicht als verbindlicher syntaktischer Standard konzipiert. Die Sprache ist aber entworfen worden, um den Funktionsumfang von Graphdatenbankabfragesprachen

Tabelle 1.1: Einfache Beispiele in Graphdatenbankabfragesprachen

Sparql	`PREFIX : <http://example.org/graph/>`
	`SELECT ?p`
	`WHERE { ?p a :Person }`

Cypher	`MATCH (p:Person)` `RETURN p`
G-core	`CONSTRUCT (p)` `MATCH (p:Person)`
Gremlin	`g.V().hasLabel('Person').toList()`

zu vereinheitlichen. Diese Abfragesprache unterstützt die Ausgabe von Graphen als Ergebnis sowie das Speichern und Abfragen von Pfaden. [2]

Bisher ist diese Abfragesprache noch kaum im Einsatz. Durch diese und weitere Implementierungen von G-core kann die Sprache in Zukunft häufiger verwandt werden und der theoretische Ansatz kann in der Praxis erprobt werden.

Gremlin bietet dabei eine vielfach implementierte, funktionale Abfragesprache für Graphdatenbanken. Eine Standardimplementierung ist in der Programmiersprache Groovy verfügbar. [11]

1.2 Aufgabenstellung

In dieser Arbeit wird die Datenbankabfragesprache G-core teilweise implementiert und in Gremlin-Befehle übersetzt. Es soll also zu einfachen Abfragen in G-core jeweils eine Ausgabe in Gremlin-Groovy erfolgen – der Standardvariante von Gremlin.

Dieser ausgegebene Programmcode wird dann zusätzlich auf Ausführbarkeit und Richtigkeit getestet, wobei zum Vergleich die Ausgangsabfrage manuell nach der algebraischen Definition von G-core [1] ausgewertet wird.

Bei dieser Aufgabe sind besonders die zwischen G-core und Gremlin verschiedenen Graphmodelle Problemstellen. Die beiden Modelle gehen von unterschiedlichen Ergebnistypen und Grapheigenschaften aus. In der Zielsprache Gremlin ist es zum Beispiel nicht möglich Graphen auszugeben – nur Knoten oder Kanten können zurückgeliefert werden.

1.3 Ziel der Arbeit

Durch diese Arbeit wird einerseits die Implementierbarkeit der neuen Graphdatenbankabfragesprache in Gremlin gezeigt. Es müssen Entscheidungen getroffen werden, wie die Eigenschaften der Graphmodelle aufeinander abgebildet werden können.

Andererseits werden diverse Probleme während der Implementierung selbst sowie in der Übertragung auf das Modell der Zielsprache aufgezeigt. Es wird somit ermittelt, welche in G-CORE vorgesehene Abfragen kaum oder gar nicht in Gremlin übersetzt werden können.

1.4 Herangehensweise

Zu Beginn werden die vorhandenen Beispiele manuell in Gremlin übertragen und Beispielgraphen aus [2] werden in der Datenbank repliziert. Danach wird eine Anwendung entwickelt, die diesen Vorgang möglichst allgemein automatisiert.

Gleichzeitig werden die getroffenen Annahmen, die trotz der formalen Grammatik von G-CORE [1] nötig werden, sowie alle gefundenen Problembereiche protokolliert. Es wird ergründet, wie Probleme zu lösen oder warum sie nicht einfach zu lösen sind.

1.5 Strukturierung dieser Arbeit

Im folgenden Kapitel 2 werden die grundlegenden Erkenntnisse aus der Literatur vorgestellt und analysiert. Die Sprache G-CORE wird anhand von einfachen Beispielen demonstriert.

Die problematischen Unterschiede zwischen den betrachteten Abfragesprachen werden in Kapitel 3 dargestellt. Die Modelle, die G-CORE und Gremlin zu Grunde liegen werden hier verglichen. Außerdem werden am Schluss dieses Kapitels die nicht implementierten G-CORE-Abfragen besprochen.

Danach beschreibt Kapitel 4 die Implementierung dieser Arbeit als Computerprogramm. Die einzelnen Stufen der Übersetzung werden an einer beispielhaften Abfrage gezeigt.

Schließlich wird die eigene Arbeit in Kapitel 5 kritisch beleuchtet und zusammengefasst.

KAPITEL 2

Graphdatenbanken und Abfragesprachen

2.1 Graphmodelle

Verschiedene Graphdatenbanken setzen zuweilen sehr unterschiedliche Datenmodelle ein. Neben Knoten und verbindenden gerichteten Kanten besitzen diese Graphen zumindest einfache Bezeichner (Labels), welche die jeweilige Bedeutung der Kante angeben. Weiters können auch Knoten mit Labels ausgestattet sein und die Objekte des Graphen können oft noch zusätzliche Eigenschaften tragen. [3]

Das einfachste Modell ist der *Graph mit beschrifteten Kanten*, in dem nur die Kanten je ein Label tragen, während die Knoten keinerlei Bezeichner oder Eigenschaften tragen. [3]

Um eine zu große Anzahl an Kanten zu vermeiden, wurden *Eigenschaftsgraphen* entworfen. Dieses Modell erweitert das vorhergehende um Labels für Graphknoten sowie Eigenschaften in Form von Schlüssel-Wert-Paaren für Knoten und Kanten. [3]

Zuletzt wurde das Modell des *Pfad-Eigenschaftsgraphen* vorgestellt, das auch Pfade als Graphobjekte aufnimmt und diese mit Labels und Eigenschaften ausstatten lässt. [2]

2.2 Abfragesprachen

Sparql ist entwickelt worden, um Abfragen auf Rdf-Graphen zu ermöglichen [13]. Das *Resource Description Framework (RDF)* ist ein Standard für Datenaustausch im Web. Rdf-Graphen bestehen nach dem Datenmodell des *Graphen mit beschrifteten Kanten* lediglich aus Knoten und gerichteten Kanten. [12]

Cypher setzt eine textgraphische Syntax für Knoten und Kanten ein, die auch von G-core übernommen worden ist. Dabei wird in dieser Sprache das *Eigenschaftsgraphen*-Modell umgesetzt. [10]

Auch Gremlin nützt dieses Datenbankmodell mit Objekteigenschaften und -bezeichnern. Während G-core wie auch Sparql und Cypher eine deklarative Sprache ist, handelt es sich bei Gremlin um eine funktionale Sprache. [2, 3]

In Gremlin wird ein angegebener Graph nach der Vorgabe bestimmter Funktionsschritte durchwandert (traversiert). Diese Funktionen können in unterschiedliche Programmiersprachen eingebettet werden [11] – in dieser Arbeit wird die Gremlin-Groovy-Syntax verwandt.

Die Abfragesprache G-core wird in *G-CORE: A Core for Future Graph Query Languages* [2] anhand von fünfzehn Beispielen präsentiert. In der formalen Definition von G-core [1] wird eine algebraische Repräsentation dieser Sprache gezeigt und deren Auswertung beschrieben.

G-core basiert auf einem *Pfad-Eigenschaftsgraphen* mit gespeicherten Knoten, gerichteten Kanten und zusammengesetzten Pfaden, wobei alle diese Objekte beliebige Mengen von Labels sowie Eigenschaften tragen können. Die Abfragesprache unterstützt dabei Operationen auf mehreren solcher Graphen zugleich. [2]

Auf dieser Grundlage an Informationen haben die Entwickler des *Linked Data Benchmark Council* weitere Projekte veröffentlicht. Auf der Website *GitHub* ist in deren Bereich sowohl ein Parser [9] als auch eine Implementierung basierend auf dem *Apache-Spark-Framework* [5] verfügbar. Zu letzterer ist auch die ausführliche Masterarbeit *A G-CORE (Graph Query Language) Interpreter* [6] veröffentlicht worden.

2.3 Gremlin-Groovy

Die funktionale Sprache Gremlin ermöglicht es, Datenbankgraphen zu traversieren und währenddessen Daten auszugeben oder auch den Graphen zu verändern. In vielen Programmiersprachen wie beispielsweise Java oder Groovy ist Gremlin bereits implementiert worden. [11]

In Gremlin-Groovy gäbe eine Abfrage der Form

```
1 g.V().hasLabel('Person').
2   local(values('name').fold()).
3   toList()
```
(2.1)

eine Liste zurück, die zu jedem Knoten im Graphen zum Traversierungsobjekt `g` mit dem Label `Person` eine Liste von Werten der Eigenschaft `name` enthält.

Ausgehend von bestimmten Knoten können dann inzidente Kanten oder adjazente Knoten erreicht werden. Eine Auswertungsreihenfolge ist in Gremlin jedoch nicht vorgegeben. [11]

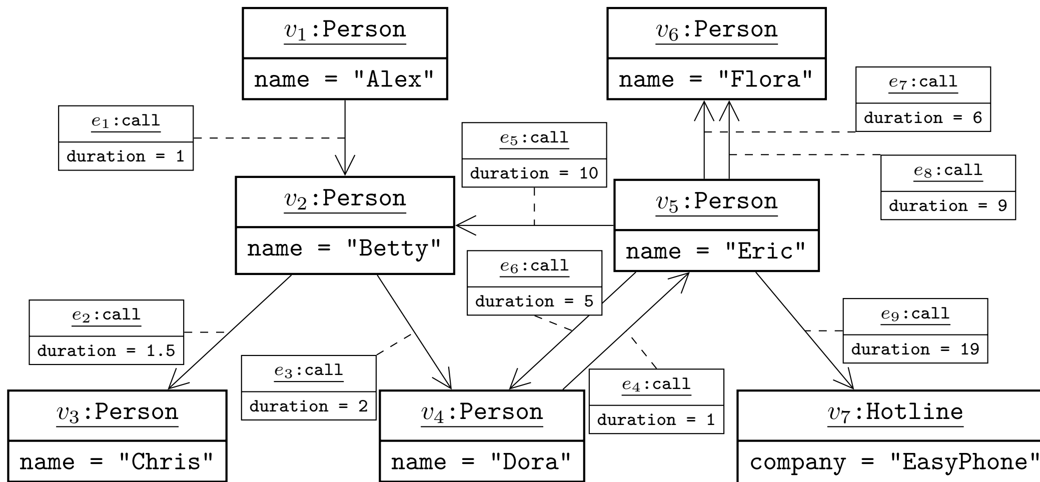


Abbildung 2.1: Der beispielhafte Datenbankgraph `call_graph` dargestellt als UML-Diagramm

Die erhaltenen Ergebnisse von Abfragen in Gremlin sind als Listen, Literale und Objektreferenzen derjenigen Programmiersprache, in welcher Gremlin implementiert worden ist. Dadurch wird die natürliche Einbindung in diverse Programmiersprachen möglich. [11]

2.4 Abfragen in G-core

Wie aus bestehenden Graphen der Datenbank ein neuer Graph konstruiert werden soll, wird durch G-core-Abfragen beschrieben. Jede Abfrage liefert als Ergebnis einen Graphen.

Die einfachsten Abfragen bestehen zumindest aus einer Construct-Klausel gefolgt von einer Match-Klausel [2]. Zum Beispiel erzeugt

```
1 CONSTRUCT (n)                                                    (2.2)
2 MATCH (n:Person) ON call_graph
```

einen Ergebnisgraphen, der nur aus unverbundenen Knoten besteht. Es werden genau diejenigen Knoten ausgewählt und mit sämtlichen ihrer Eigenschaften zurückgeliefert, die den Bezeichner `Person` als ein Label besitzen.

Dabei bezeichnet `call_graph` einen neuen, einfachen Beispielgraphen, der Personen und geführte Telefongespräche enthält. Der Graph ist in Abbildung 2.1 dargestellt. Die obige Abfrage 2.2 lieferte also alle Knoten außer v_7.

Weiters können auch Kanten oder Pfade als Muster angegeben werden, welche dann mit Teilen des Eingabegraphen verglichen und übereingestimmt werden. So findet etwa das Muster `(v)-[:call]->()-[:call]->(:Hotline)` alle Knoten `v`, die über genau zwei

Kanten mit dem Label `call` zu einem Knoten mit dem Bezeichner `Hotline` adjazent sind – also Personen, die mit einer Person telefoniert haben, die wiederum eine Hotline angerufen hat. In diesem Beispiel erhielte man also nur Dora v_4.

Die erhaltenen Übereinstimmungen können daraufhin mittels Where-Ausdrücken weiter eingeschränkt oder mit Optional-Klauseln erweitert werden. Auch in der Construct-Klausel kann das Ergebnis gefiltert werden – hierzu dient das Schlüsselwort When. [2, 1]

Darüber hinaus können mittels einer Art regulärer Ausdrücke Pfade variabler Länge abgefragt werden. Diese Pfade können in G-core als Graphobjekte mit Labels und Eigenschaften zurückgegeben werden. [2]

Beispielsweise ergibt die Abfrage

```
1 CONSTRUCT (a)-/@p :cycleCall {calls:=c}/->(a)
2 MATCH (a)-/SHORTEST p <:call*> COST c/->(a) ON call_graph
3 WHERE a.name = 'Eric'
```
(2.3)

einen Teilgraphen, der einen Zyklus als Pfad sowie die darin enthaltenen Knoten und Kanten enthält. Über dem Beispielgraphen ausgewertet, bekäme man somit ein neues Pfadobjekt mit dem Bezeichner `cycleCall`, der Eigenschaft `calls` mit dem Wert 3 sowie den Objekten v_5, e_5, v_2, e_3, v_4, e_4 und v_5 in dieser Reihenfolge.

Eine Besonderheit von G-core-Abfragen ist außerdem die Möglichkeit, mehrere Graphen mit den Schlüsselwörtern Union, Intersect und Minus miteinander zu kombinieren. Diese Mengenoperationen werden auf alle Graphobjekte angewandt. [2]

2.5 Kritik an G-core

Die Masterarbeit [6] zeigt einerseits, wie die Graphdatenbankabfragesprache auf nicht graphbasierten Systemen implementiert werden kann. Andererseits werden auch Probleme mit G-core anhand von Verbesserungsvorschlägen präsentiert.

Zuerst wird argumentiert, dass die Match-Klausel vor dem Construct stehen dürfen sollte. Die Reihenfolge der Beschreibung in der Grammatik [1] und die Implementierung [6] sowie auch die Reihenfolge, in der man als Benützer die Abfragen verfasst, entsprechen eben nicht der bisherigen Vorgabe von G-core.

Ebenfalls wird festgestellt, dass es in manchen Anwendungsfällen gar nicht wünschenswert ist, einen Graphen als Antwort auf jede Abfrage zurückzugeben. Andere Rückgabetypen könnten als Erweiterung unterstützt werden. [6]

Außerdem wird auch in [6] aufgezeigt, dass in manchen Abfragen mit On-Klauseln die Semantik der Zugehörigkeit dieser Ausdrücke zu einzelnen Mustern unklar erscheint. Man könnte also annehmen, dass in `MATCH (a), (b) ON g` auch `a` Knoten aus dem Graphen `g` auswählte, die On-Klausel erstreckt sich aber nicht über Beistriche hinweg.

Diese Definition der Gültigkeit von On-Klauseln hat den Vorteil, dass zugleich unterschiedliche Graphen als Quellen gewählt werden können (siehe Abfragen 4.4 oder 3.1).

Der Nachteil, dass nicht mehrere Muster dieselbe On-Klausel teilen können, wird deutlich, wenn die Entwickler der Sprache G-core selbst in [2] diese Angabe vergessen. In der Abfrage

```
1 CONSTRUCT (n)-[e:wagnerFriend {score:=COUNT(*)}]->(m)
2           WHEN e.score > 0
3   MATCH   (n:Person)-/@p:toWagner/->(), (m:Person)
4     ON    social_graph2
5     WHERE m = nodes(p)[2]
```
(2.4)

aus [2] muss das erste Muster vor dem Beistrich auf dem sogenannten Defaultgraphen `social_graph` angewandt werden, wodurch kein Ergebnis zustande kommt.

KAPITEL 3

Inkompatibilitäten und nicht implementierte Fälle

3.1 Graphmodellunterschiede

Obgleich sowohl in G-CORE- als auch in Gremlin-Graphen Eigenschaften zu Graphobjekten gespeichert werden können, gibt es in den beiden Graphmodellen grundlegende Unterschiede. Pfade zum Beispiel gehören zwar in G-CORE zu diesen Objekten, in Gremlin jedoch können sie weder als Objekt gespeichert werden, noch können sie Eigenschaften tragen. [2, 11]

Darüber hinaus werden Labels in Gremlin als spezielle Objekteigenschaft behandelt, die mit `hasLabel` zugesichert werden kann, wobei nur ein einziger Bezeichnertext pro Objekt verwendet werden kann. [11]

Demgegenüber steht in G-CORE die Festlegung einer Label-Menge pro Graphobjekt, wobei in Abfragen nur ein Label übereinstimmen muss. Ein Objekt mit den Bezeichnern `Person` und `Manager` kann demnach auch mit `:Person` allein abgefragt werden. [2]

Dieser Unterschied könnte in der Implementierung umgangen werden, indem statt der eigentlichen Labels in Gremlin eigene Eigenschaften (etwa `_label`) mit mehreren Werten verwendet würden. Allerdings könnte dann die `hasLabel`-Funktion nicht mehr verwendet werden – daher wurde diese Inkompatibilität beibehalten.

3.2 Abfragen auf mehreren Graphen

Die Funktionen UNION, INTERSECT und MINUS können in der Zielsprache Gremlin nicht abgebildet werden, weil Gremlin lediglich die Traversierung eines einzelnen Graphen unterstützt. [11]

Vereinigung, Durchschnitt oder Mengendifferenz der Objekte der Abfrageergebnisse – wie sie in G-core vorgesehen sind – könnten nur in Groovy selbst beziehungsweise der jeweiligen Umgebung von Gremlin ermittelt werden.

Außerdem gibt es in G-core die Möglichkeit, mehrere Graphen gleichzeitig in einer Abfrage zu traversieren. Eine Abfrage wie

```
1 CONSTRUCT (c)<-[:worksAt]-(n)
2   MATCH   (c:Company) ON company_graph,
3           (n:Person)  ON social_graph
4     WHERE c.name IN n.employer
```
(3.1)

aus [2] wäre in Gremlin nicht mit getrennten Graphen `company_graph` und `social_graph` möglich.

Deswegen werden sämtliche G-core-Graphen in einem Gremlin-Graphen (in der Implementierung `_full_graph` genannt mit dem Traversierungsobjekt `_full_graph_t`) als Komponenten gespeichert und der ursprüngliche Graph wird zu allen Graphobjekten als zusätzliche Eigenschaft `_on` eingetragen.

3.3 Ergebnisdatentypen

Während in G-core zu jeder Abfrage stets ein Graph mit seinen etwaigen Knoten, Kanten und Pfaden zurückgeliefert wird, können in Gremlin verschiedene andere Werte wie Zahlen, Zeichenketten und Listen oder auch Knoten und Kanten zurückgegeben werden. [2, 11]

Daher liefert diese Implementierung der Übersetzung die Graphobjekte der möglichen Variablenbelegungen der Construct-Klausel einer Abfrage als Liste. Gesamte Graphen, die etwa über den Graphbezeichner abgefragt werden, können in Gremlin nicht in einer einzigen Abfrage ausgegeben werden. Deshalb können Gremlin-Abfragen (anders als in G-core) nicht rekursiv aufeinander aufbauen.

3.4 Ausdrücke

Die Graphdatenbankabfragesprache G-core zählt in [1] einige Beispiele für ein- und zweistellige Operatoren in Ausdrücken auf. Implementierungen sollen demnach sowohl arithmetische als auch boolesche Operatoren sowie Aggregationen unterstützen.

Das Manuskript [1] sieht hier zumindest die Wörter And, Or und Not; Addition, Subtraktion, Multiplikation und Division; Vergleichsoperatoren inklusive Subset Of und In sowie die Aggregationen Count, Min, Max, Sum, Avg vor.

In Gremlin hingegen werden statt dieser Ausdrücke Prädikatsfunktionen bevorzugt. Während boolesche Operatoren als Funktionen `and` und `or` direkt in die Traversierung

eingefügt werden können, sollen die arithmetischen durch Prädikate wie beispielsweise `gt` (größer als) oder `neq` (ungleich) innerhalb anderer Funktionsaufrufe (`has`, `where` usw.) repräsentiert werden. [11]

Hierbei sind die beiden Sprachen dergestalt unterschiedlich, dass eine vollständige Übertragung in die jeweils andere Abfragesprache sehr aufwändig wäre.

3.5 Nicht implementierte Fälle

Die nicht implementierten Fälle umfassen Abfragen über Pfade, rekursive Abfragen, komplizierte Ausdrücke sowie When-, Set-, Remove- Path- Graph- und Optional-Klauseln. Das heißt, diese Konstrukte werden zwar korrekt erkannt und in den Syntaxbaum umgewandelt, sie können aber mit dieser partiellen Implementierung nicht in Gremlin ausgegeben werden.

Abfragen über Pfade wurden in der Gremlin-Übertragung nicht implementiert, weil diese in der Zielsprache sehr ausführlich und unübersichtlich wären. Eine relativ einfache Klausel wie `MATCH (n)-/SHORTEST p<:knows*> COST c/->(m)` [2] könnte zum Beispiel als

```
1  _full_graph_t.withSack(0).V().as('n').                         (3.2)
2    repeat(
3        outE().hasLabel('knows').
4        sack(sum).by(constant(1)).
5        inV().as('m').
6        group('_min').
7          by(select('n', 'm')).
8          by(sack().min()).
9        filter(
10             project('_sack', '_x').
11               by(sack()).
12               by(select('_min').select(select('n', 'm'))).
13             where('_sack', eq('_x'))
14           ).
15        aggregate('_p').
16          by(project('n', 'm', 'p', 'c').
17            by(select('n')).
18            by(select('m')).
19            by(path()).
20            by(sack()))
21      ).
22    cap('_p').unfold().
23    dedup().
24      by(select('n', 'm').select(values).order(local).by(id))
```

übersetzt werden [4]. Solch lange Abfragen wären im Allgemeinen fehleranfällig und schwierig zu testen.

Rekursive Abfragen wie `ON (`⟨Abfrage⟩`)` oder Exists können nicht direkt in Gremlin umgesetzt werden, weil Gremlin-Abfragen eben keine Graphen zurückliefern können.

Ausdrücke, welche ineinander verschachtelt vorkommen, wie boolesche Operatoren wurden ebenfalls nicht implementiert. Es werden die Schlüsselwörter In und Subset Of sowie die Operatoren `==`, `<>`, `<=`, `>=`, `<` und `>` unterstützt.

Es wurden auch keine Indexzugriffe auf Listenelemente der Form `list[i]` implementiert – nicht zuletzt deshalb, weil die Definition in [2] hier widersprüchlich ist. Dort steht zuerst, der Zugriff wäre null-indiziert, dann aber, dass obiger Ausdruck das Element an der Stelle $i - 1$ (statt $i + 1$) bedeutete.

KAPITEL 4

Implementierung

4.1 Gestaltung

Die Implementierung besteht aus einem Lexer, einem Parser, einer Darstellung als abstrakter Syntaxbaum sowie einer Übersetzung in Gremlin-Groovy. Diese Programmteile wurden in der Programmiersprache C verfasst. Der Lexer wurde mit Flex 2.6.4 [8] erzeugt, während der Parser mittels GNU Bison 3.1 [7] generiert wurde.

4.2 Lexer

Zunächst muss der Text der Eingabe in eine maschinenlesbare Form überführt werden. Dazu gibt es keine Vorgaben in der Präsentation von G-CORE [2]. Anhand der dort angeführten Beispiele wurde aber ein einfacher Lexer entwickelt.

Der Lexer registriert die gesammelten Schlüsselwörter der Abfragesprache unabhängig von deren Groß- und Kleinschreibung. Weißraum bestehend aus Leerzeichen, Tabulatoren und Zeilenumbrüchen wird ignoriert, während einfache Literale wie Zeichenfolgen unter Anführungszeichen oder auch Zahlwerte erkannt werden.

Die Abfrage

```
1 CONSTRUCT (n)
2   MATCH   (n:Person)
3     ON    social_graph
4     WHERE n.employer = 'Acme'
```
(4.1)

aus [2] verarbeitet der Lexer also gleich wie zum Beispiel

```
1   construct ( n ) Match ( n: Person) On
2 social_graph where n .employer= 'Acme'
```
(4.2)

zu der Folge erkannter Wörter (mit deren Dateninhalt in eckigen Klammern)

```
CONSTRUCT '(' IDENTIFIER[n] ')' MATCH '(' IDENTIFIER[n] ':'
IDENTIFIER[Person] ')' ON IDENTIFIER[social_graph] WHERE IDENTIFIER[n]
'.' IDENTIFIER[employer] '=' STRING[Acme] .
```

Diese identifizierten Wörter und deren Bedeutung liefert der Lexer dann an den Parser.

4.3 Parser

Gemäß der Grammatik in [1] sowie [9] wurden Regeln für einen LR(1)-Parser geschrieben. Während dieser die Eingabe verarbeitet, wird ein abstrakter Syntaxbaum erzeugt, der dieselbe Abfrage repräsentiert.

Die Struktur dieses Baumes bestehend aus definierten Knotentypen und jeweils erlaubten Kindknoten ist vorab festgelegt worden. Ein Knoten des abstrakten Syntaxbaumes ist hierbei ein strukturiertes Objekt mit seinem Knotentyp, einem je nach Typ verwandten Datenzeiger sowie einer Liste von Kindern.

Bei diesen Typen von Knoten wird zwischen fünf Arten von Literalen und 49 Arten innerer Knoten unterschieden. Damit kann G-core vollständig beschrieben werden.

Graphknoten der Abfrage werden – wie in [2] beschrieben – nach der Syntax in Cypher [10] beispielweise als `(v:Label {key=value})` notiert, wobei hier ein Gleichheitszeichen statt des Doppelpunktes in Cypher steht. In Eigenschaftszuweisungen der Construct-Klauseln wird in G-core der Operator `:=` verwandt.

Die Syntax, mit der Kanten `e` und Pfade `p` beschrieben werden sollen, folgt den Beispielen in [2] als `(v)-[e]->(w)` beziehungsweise `(v)-/p/->(w)`. Im Gegensatz zu Cypher werden hier bisher keine ungerichteten Kanten implementiert.

Der Parser erzeugt zum Beispiel für die Abfrage 4.1 (oder genauso für 4.2) den Syntaxbaum in Abbildung 4.1. Diese Datenstruktur wird dann an die Funktion zur Übersetzung in Gremlin-Befehle übergeben.

4.4 Übersetzung in Gremlin

Zuletzt wird also der erhaltene Syntaxbaum ausgehend von der Wurzel traversiert. Erzeugter Gremlin-Quellcode wird dabei in die Standardausgabe des Programms geschrieben.

Für jede Abfrage aus Construct und Match wird eine Gremlin-Abfrage erzeugt. Weil in Gremlin keine Graphen zurückgegeben werden können, werden für jede Abfrage in einer Mengenoperation (Union, Intersect, Minus) die Teilabfragen nacheinander, unabhängig ausgeführt.

Vor dem Construct wird die Match-Klausel übersetzt. Für Knoten- und Kanten-Abfragemuster werden äquivalente Konstrukte in Gremlin erzeugt. Etwa wird aus dem

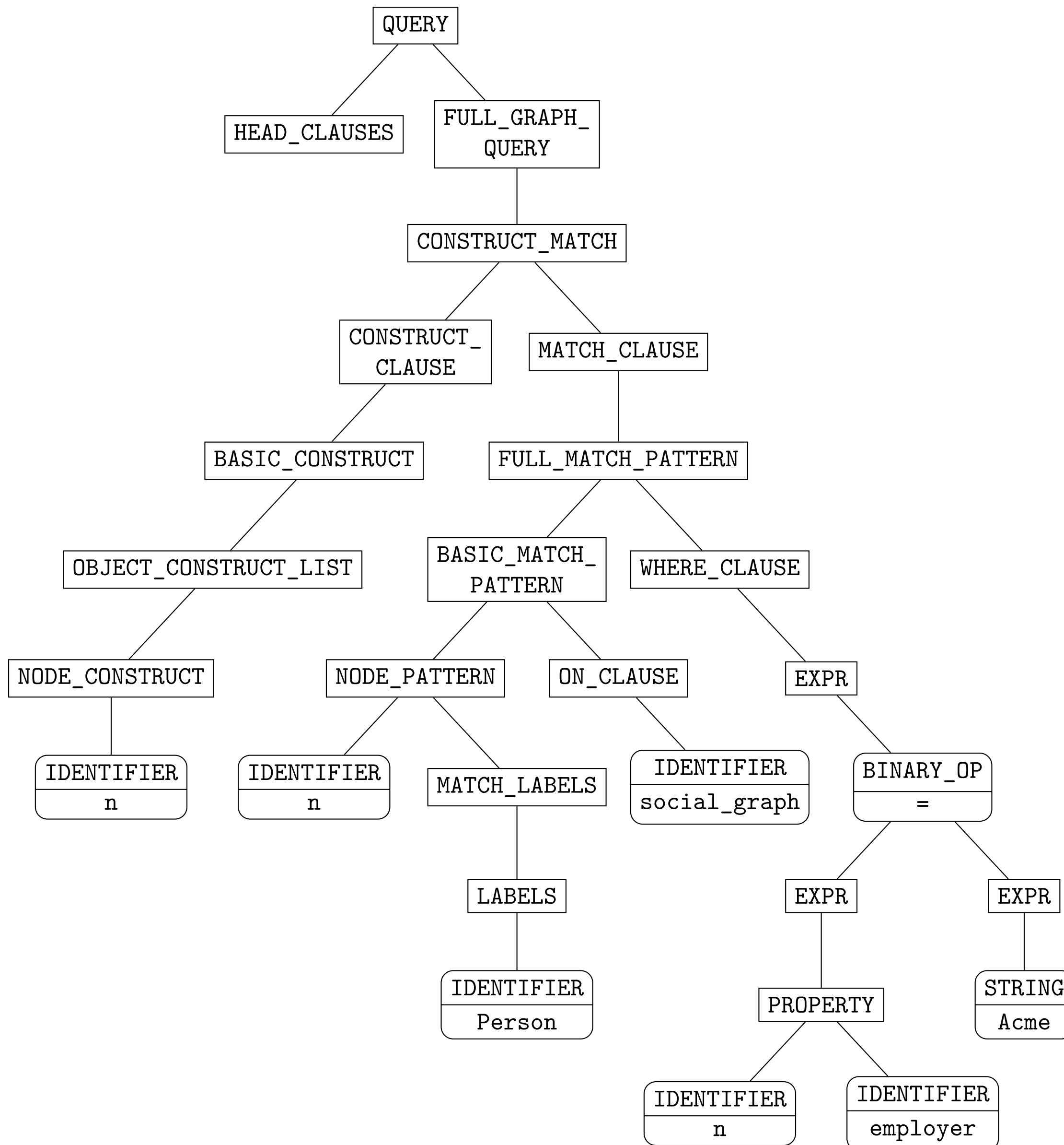


Abbildung 4.1: Abstrakter Syntaxbaum zur Abfrage 4.1

Muster (n:Person) der Aufruf .V().as('n').hasLabel('Person'), während Kanten über .outE() und .inV() traversiert werden.

Einschränkungen mit Where-Ausdrücken werden in die where-Aufrufe gefolgt von by-Schritten übertragen. Danach werden die in der Construct-Klausel abgefragten Objekte mit einem select-Befehl ausgewählt.

Der abstrakte Syntaxbaum des Beispiels in Abbildung 4.1 erzeugt insgesamt die Ausgabe

```
1 _full_graph_t.withSideEffect('_', 0).                                  (4.3)
2 V().as('n').hasLabel('Person').has('_on', 'social_graph').
3 where('n', test(gcoreEq, '_')).
4   by(values('employer').fold()).
5   by(constant('Acme')).
6 select('n').
7   by(valueMap().with(WithOptions.tokens, WithOptions.labels)).
8 toList()
```

(jedoch ohne die Einrückungen und Zeilenumbrüche).

Anfangs wird hier dem Traversierungsobjekt _full_graph_t der Seiteneffekt _ (Unterstrich) hinzugefügt, damit dann im where-Aufruf Konstanten gleich wie Eigenschaften behandelt werden können. Die Zugehörigkeit der Knoten zum Graphen social_graph wird über die Eigenschaft _on dargestellt.

In where-Ausdrücken werden dann eigene Prädikatsfunktionen wie gcoreEq eingesetzt, die den jeweiligen Vergleich unter Berücksichtigung der Forderung, dass einelementige Mengen in G-core mit ihrem Element gleichgesetzt werden [2], durchführen.

Ein komplexeres Beispiel beschreibt die G-core-Abfrage

```
1 CONSTRUCT (c)<-[:worksAt]-(n)                                          (4.4)
2   MATCH   (c:Company)             ON company_graph,
3           (n:Person {employer=e}) ON social_graph
4     WHERE c.name = e
5 UNION social_graph
```

aus [2] wegen der Variablenbelegung der Eigenschaft employer, die bei einem Knoten des Beispielgraphen mehrfach vorkommt.

Hierbei werden die Variablenbelegungen für c und n berechnet, aber es werden keine Kanten erzeugt und die Vereinigung mit dem Graphen social_graph wird ignoriert, weil in Gremlin keine Graphen ausgegeben werden können (siehe Abschnitt 3.3).

Im Ergebnis wird eine Belegung pro Wert der Eigenschaft zurückgeliefert, wie es auch die übersetzte Abfrage

```
1 _full_graph_t.withSideEffect('_', 0).                                    (4.5)
2 V().as('c').hasLabel('Company').has('_on', 'company_graph').
3 V().as('n').hasLabel('Person').as('_tmp').
4   coalesce(values('employer'), constant(null)).as('e').
5 select('_tmp').has('_on', 'social_graph').
6 where('c', test(gcoreEq, 'e')).
7   by(values('name').fold()).
8   by().
9 select('n', 'c').
10   by(valueMap().with(WithOptions.tokens, WithOptions.labels)).
11 toList()
```

(wiederum wurden hier Einrückungen und Zeilenumbrüche zur Darstellung ergänzt) vermag.

Der temporäre Bezeichner `_tmp` dient hierbei dazu, einen Einschub vorzunehmen und danach wieder zurückzukehren. In diesem Einschub werden die Werte der Eigenschaft `employer` als Variable `e` gebunden, wobei fehlende Werte – wie in [2] beschrieben – durch `null` ersetzt werden.

KAPITEL 5

Kritische Reflexion und Zusammenfassung

5.1 Vergleich mit verwandten Arbeiten

Gegenüber der Masterarbeit [6], die G-CORE in SQL übersetzt, kann mit dieser Arbeit eine Übersetzung in Gremlin durchgeführt werden. Die Ergebnisse sind daher sehr unterschiedlich und bei der Gremlin-Implementierung werden zusätzliche Einschränkungen in Kauf genommen (siehe Kapitel 3).

Auch die Masterarbeit fordert gewisse Einschränkungen für das Graphmodell. Es werden nämlich ebenfalls nur einfache Labels statt Mengen unterstützt. Zudem kann dort zu jeder Eigenschaft nur ein Wert gespeichert werden, während in G-CORE (und Gremlin) mehrere Werte pro Eigenschaft erlaubt sind. [6, 2]

Allerdings vermag jene SQL-Übersetzung, wesentlich mehr G-CORE-Funktionalitäten zu implementieren. Dabei werden vorhandene Bibliotheken wie *Apache Spark* eingesetzt. [6] Dadurch konnten unter anderem Pfadabfragen einfacher implementiert werden.

Während SQL wie G-CORE eine deklarative Abfragesprache ist, konnte in dieser Arbeit hier eine Übertragung auf ein anderes Paradigma – nämlich auf die funktionale Sprache Gremlin – durchgeführt werden. [2, 11]

5.2 Diskussion offener Probleme

Der erzeugte Gremlin-Code beruht auf einer speziellen Graphdefinition mit der Eigenschaft `_on` zu jedem Objekt. Außerdem werden in Groovy definierte Prädikatsfunktionen zur Ausführung der Abfragen benötigt. Daher kann der Code nicht zu beliebigen Gremlin-Graphen kompatibel gestaltet werden.

Auch der Einsatz des nicht direkt verwendeten Seiteneffektes _ (Unterstrich) sowie die unformatierte, einzeilige Ausgabe führen zu schlecht lesbaren Ergebnissen, welche dem Stil manueller Gremlin-Groovy-Abfragen widersprechen.

Eine vollständige Implementierung der Abfragesprache G-core wäre in Gremlin auf Grund der beschriebenen Inkompatibilitäten nicht möglich. Eventuell könnten aber weitere Behelfslösungen angewandt werden. Darunter litte jedoch die Verständlichkeit des Gremlin-Codes umso mehr.

Ein weiteres Problem beim manuellen Testen der Implementierung ist, dass bisher nur wenige Testfälle verfügbar sind und kaum Grenzfälle oder dergleichen abgedeckt werden. Eine größere Testsammlung wäre auch für künftige Entwicklungen nützlich.

5.3 Zusammenfassung und Ausblick

Verschiedene Abfragesprachen für Graphdatenbanken setzen auf unterschiedliche Modelle, Syntaxen und Paradigmen. Sparql, Cypher, G-core und Gremlin wurden dahingehend verglichen. Gremlin wurde als Zielsprache für die Implementierung von G-core gewählt.

Die Graphdatenbankabfragesprache G-core konnte in dieser Arbeit implementiert werden. Es wurde gezeigt, wie ein abstrakter Syntaxbaum zu jeder gegebenen Abfrage erzeugt werden kann. Die wesentlichen Unterschiede zu Gremlin konnten aufgezeigt werden, während die anderen Funktionalitäten programmatisch in Gremlin-Abfragen übersetzt wurden.

Die implementierten Fälle wurden in Gremlin-Groovy ausgeführt und getestet. Dabei konnten die gewünschten Ergebnisse erhalten werden. Einige Fälle von Eingaben können mit dieser partiellen Implementierung jedoch nicht übertragen werden.

Die Programmteile des Lexers, Parsers sowie die Definition des abstrakten Syntaxbaumes können unabhängig voneinander ausgetauscht und wiederverwandt werden. Auch die Übersetzungsfunktion nach Gremlin kann relativ einfach erweitert oder verändert werden.

Weiters bleibt noch anzumerken, dass sich die generierten Gremlin-Ausgaben stilistisch deutlich von selbst geschriebenem Gremlin-Code unterscheiden.

Die Implementierung kann für bestimmte G-core-Funktionen in Zukunft noch ergänzt werden. Unter dem alleinigen Einsatz von Gremlin, wird jedoch keine vollständige Umsetzung möglich sein.

Abbildungsverzeichnis

2.1 Der beispielhafte Datenbankgraph `call_graph` dargestellt als UML-Diagramm 7
4.1 Abstrakter Syntaxbaum zur Abfrage 4.1 17

Tabellenverzeichnis

1.1 Einfache Beispiele in Graphdatenbankabfragesprachen 2

Literaturverzeichnis

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest und Hannes Voigt: *A formal definition of G-CORE [2]*, 2017. http://arxiv.org/abs/1712.01550, zuletzt abgerufen am 18. Februar 2019, Anhang A im Online-Manuskript.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest und Hannes Voigt: *G-CORE: A Core for Future Graph Query Languages*. In: *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, Seiten 1421–1432. Association for Computing Machinery, Juni 2018.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter und Domagoj Vrgoč: *Foundations of Modern Query Languages for Graph Databases*. ACM Computing Surveys, 50(5): 68:1–68:40, September 2017.

[4] Apache Software Foundation: *Recipes. Shortest Path*. http://tinkerpop.apache.org/docs/3.4.0/recipes/#shortest-path, zuletzt abgerufen am 4. Mai 2019.

[5] Peter Boncz und Georgiana Diana Ciocîrdel: *G-CORE interpreter on Spark*. https://github.com/ldbc/gcore-spark, zuletzt abgerufen am 27. Juni 2019, Quellcode zur Arbeit [6].

[6] Georgiana Diana Ciocîrdel: *A G-CORE (Graph Query Language) Interpreter*. Masterarbeit, Vrije Universiteit Amsterdam, August 2018.

[7] Akim Demaille und Paul Eggert: *GNU Bison*. https://www.gnu.org/software/bison/, zuletzt abgerufen am 8. August 2019.

[8] Will Estes: *flex 2.6.4*. https://github.com/westes/flex/releases/tag/v2.6.4, zuletzt abgerufen am 8. August 2019.

[9] George Fletcher, Oskar van Rest und Hannes Voigt: *G-CORE Grammar and Parser*. https://github.com/ldbc/ldbc_gcore_parser, zuletzt abgerufen am 27. Juni 2019.

[10] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer und Andrés Taylor: *Cypher: An Evolving Query Language for Property Graphs*. In: *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, Seiten 1433–1445. Association for Computing Machinery, Juni 2018.

[11] Marko A. Rodriguez: *The Gremlin Graph Traversal Machine and Language*. In: *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, Seiten 1–10. Association for Computing Machinery, Oktober 2015.

[12] W3C RDF Working Group: *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf11-concepts/, zuletzt abgerufen am 27. Juli 2019.

[13] W3C SPARQL Working Group: *SPARQL 1.1 Overview*. https://www.w3.org/TR/sparql11-overview/, zuletzt abgerufen am 21. Juni 2019.

Quelltext

lexer.l

```
%{
#include <stdlib.h>

#include "ast.h"
#include "common.h"
#include "parser.h"

#define YY_USER_ACTION                                    \
  yylloc->first_line = yylloc->last_line = yylineno; \
  yylloc->first_column = yycolumn + 1;               \
  yylloc->last_column = yycolumn + yyleng;           \
  yycolumn += yyleng;
%}

%option bison-bridge bison-locations
%option case-insensitive
%option noyywrap yylineno warn nodefault reentrant
%option header-file="lexer.h"
%option noinput nounput

%%

\n                  yycolumn = 0;
[[:space:]]{-}[\n]+  /* ignore white space */

"ALL" return TOK_ALL;
"AND" return TOK_AND;
"AS" return TOK_AS;
"CASE" return TOK_CASE;
"CONSTRUCT" return TOK_CONSTRUCT;
"COST" return TOK_COST;
"ELSE" return TOK_ELSE;
"END" return TOK_END;
"EXISTS" return TOK_EXISTS;
"FALSE" return TOK_FALSE;
```

```
"GRAPH" return TOK_GRAPH;
"GRAPH"[[:space:]]{-}[\n]+"VIEW" return TOK_GRAPH_VIEW;
"GROUP" return TOK_GROUP;
"IN" return TOK_IN;
"INTERSECT" return TOK_INTERSECT;
"IS" return TOK_IS;
"MATCH" return TOK_MATCH;
"MINUS" return TOK_MINUS;
"NOT" return TOK_NOT;
"NULL" return TOK_NULL;
"OF" return TOK_OF;
"ON" return TOK_ON;
"OPTIONAL" return TOK_OPTIONAL;
"OR" return TOK_OR;
"PATH" return TOK_PATH;
"REMOVE" return TOK_REMOVE;
"SET" return TOK_SET;
"SHORTEST" return TOK_SHORTEST;
"SUBSET" return TOK_SUBSET;
"THEN" return TOK_THEN;
"TRUE" return TOK_TRUE;
"UNION" return TOK_UNION;
"WHEN" return TOK_WHEN;
"WHERE" return TOK_WHERE;

":=" return TOK_COLON_EQUAL;
"<-" return TOK_EDGE_LEFT;
"->" return TOK_EDGE_RIGHT;
">=" return TOK_GREATER_EQUAL;
"<=" return TOK_LESS_EQUAL;
"<>" return TOK_NOT_EQUAL;

[_[:alpha:]][_[:alnum:]]*  {
  char *string = cstrdup (yytext);
  *yylval = ast_new_node (L_IDENTIFIER, string, 0, NULL);
  return TOK_IDENTIFIER;
}

[[:digit:]]*"."[[:digit:]]+  {
  double *decimal = cmalloc (sizeof *decimal);
  *decimal = atof (yytext);
  *yylval = ast_new_node (L_FLOAT, decimal, 0, NULL);
  return TOK_FLOAT;
}

[[:digit:]]+  {
  int *integer = cmalloc (sizeof *integer);
  *integer = atoi (yytext);
  *yylval = ast_new_node (L_INTEGER, integer, 0, NULL);
```

```
  return TOK_INTEGER;
}

"'"[^'\n]*"'"  {
  yytext[yyleng - 1] = '\0';
  char *string = cstrdup (yytext + 1);
  *yylval = ast_new_node (L_STRING, string, 0, NULL);
  return TOK_STRING;
}

.  return *yytext;

%%
```

parser.y

```
%code {

#include <errno.h>
#include <error.h>
#include <getopt.h>
#include <locale.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "common.h"
#include "gremlin.h"
#include "lexer.h"

}

%code requires {

#include "ast.h"

typedef void *yyscan_t;

}

%define api.pure         full
%define api.token.prefix {TOK_}
%define parse.lac        full
%define parse.error      verbose
%define parse.trace      true
%locations
%defines
```

```
%verbose

%lex-param   {yyscan_t yyscanner}
%parse-param {yyscan_t yyscanner}

%define api.value.type {struct ast_node *}

%token ALL
%token AND
%token AS
%token CASE
%token CONSTRUCT
%token COST
%token ELSE
%token END
%token EXISTS
%token FALSE
%token GRAPH
%token GRAPH_VIEW
%token GROUP
%token IN
%token INTERSECT
%token IS
%token MATCH
%token MINUS
%token NOT
%token NULL
%token OF
%token ON
%token OPTIONAL
%token OR
%token PATH
%token REMOVE
%token SET
%token SHORTEST
%token SUBSET
%token THEN
%token TRUE
%token UNION
%token WHEN
%token WHERE

%token COLON_EQUAL ":="
%token EDGE_LEFT "<-"
%token EDGE_RIGHT "->"
%token GREATER_EQUAL ">="
%token LESS_EQUAL "<="
%token NOT_EQUAL "<>"
```

```
%token IDENTIFIER STRING INTEGER FLOAT

%left OR
%left AND
%precedence NOT
%left '=' "<>" '>' '<' ">=" "<=" IN SUBSET
%left '+' '-'
%left '*' '/'
%precedence IS '['
%precedence UMINUS

%precedence R_CONCAT R_UNION
//%precedence R_KLEENE

%code provides {

void yyerror (YYLTYPE *yylloc, yyscan_t yyscanner, const char *msg);

}

%code {

static struct ast_node *
list_append_node (enum ast_type type, struct ast_node *list,
                  struct ast_node *node);

static struct ast_node *
node_append_list (enum ast_type type, struct ast_node *node,
                  struct ast_node *list);

static struct ast_node *
node_append_list_node (enum ast_type type, struct ast_node *nodea,
                       struct ast_node *list, struct ast_node *nodeb);

static struct ast_node *
node_append_list_list_node (enum ast_type type, struct ast_node *nodea,
                            struct ast_node *lista, struct ast_node *listb,
                            struct ast_node *nodeb);

static void
node_prepend_two (struct ast_node *node, struct ast_node *one,
                  struct ast_node *two);

struct ast_node *root = NULL;

}

%%
```

```
stmt : graphView { $$ = root = $1; }
     | query     { $$ = root = $1; }
     ;

graphView : GRAPH_VIEW gid AS '(' fullGraphQuery ')'
          { $$ = ast_v_new_node (N_GRAPH_VIEW, 2, $gid, $fullGraphQuery); }
          ;

query : headClauses fullGraphQuery
      { $$ = ast_v_new_node (N_QUERY, 2, $headClauses, $fullGraphQuery); }
      ;

gid : IDENTIFIER
    ;

fullGraphQuery : basicGraphQuery
               {
                 $$ = ast_v_new_node (N_FULL_GRAPH_QUERY, 1,
                                      $basicGraphQuery);
               }
               | fullGraphQuery[l] setOp fullGraphQuery[r]
               {
                 $$ = ast_v_new_node (N_FULL_GRAPH_QUERY, 1,
                                      ast_v_new_node (N_SETOP, 2, $l, $r));
                 $$->children[0]->data = $setOp->data;
                 free ($setOp);
               }
               ;

headClauses : %empty { $$ = ast_v_new_node (N_HEAD_CLAUSES, 0); }
            | headClauses pathClause
            { $$ = list_append_node (N_HEAD_CLAUSES, $1, $pathClause); }
            | headClauses graphClause
            { $$ = list_append_node (N_HEAD_CLAUSES, $1, $graphClause); }
            ;

basicGraphQuery : constructClause matchClause
                {
                  $$ = ast_v_new_node (N_CONSTRUCT_MATCH, 2, $constructClause,
                                       $matchClause);
                }
                | gid
                ;

setOp : UNION
      {
        enum ast_setop_type *type = cmalloc (sizeof *type);
        *type = SETOP_UNION;
        $$ = ast_new_node (0, type, 0, NULL);
```

```
      }
      | INTERSECT
      {
        enum ast_setop_type *type = cmalloc (sizeof *type);
        *type = SETOP_INTERSECT;
        $$ = ast_new_node (0, type, 0, NULL);
      }
      | MINUS
      {
        enum ast_setop_type *type = cmalloc (sizeof *type);
        *type = SETOP_MINUS;
        $$ = ast_new_node (0, type, 0, NULL);
      }
      ;

pathClause : PATH IDENTIFIER[pname] '=' walkPattern optGraphPattern
             optWhereClause optCostClause
           {
             $$ = ast_v_new_node (N_PATH_CLAUSE, 5, $pname, $walkPattern,
                                  $optGraphPattern, $optWhereClause,
                                  $optCostClause);
           }
           ;

graphClause : GRAPH gid AS '(' fullGraphQuery ')'
            {
              $$ = ast_v_new_node (N_GRAPH_CLAUSE, 2, $gid,
                                   $fullGraphQuery);
            }
            ;

constructClause : CONSTRUCT fullConstruct { $$ = $fullConstruct; }
                ;

matchClause : MATCH fullGraphPatternCondition optionalClauses
            {
              $$ = node_append_list (N_MATCH_CLAUSE,
                                     $fullGraphPatternCondition,
                                     $optionalClauses);
            }
            ;

optWhereClause : %empty { $$ = NULL; }
               | WHERE booleanCondition
               { $$ = ast_v_new_node (N_WHERE_CLAUSE, 1, $booleanCondition); }
               ;

optCostClause : %empty { $$ = NULL; }
              | COST expr
```

```
              { $$ = ast_v_new_node (N_COST_CLAUSE, 1, $expr); }
              ;

booleanCondition : expr
                 ;

fullConstruct : basicConstruct
              {
                $$ = ast_v_new_node (N_CONSTRUCT_CLAUSE, 1,
                                     $basicConstruct);
              }
              | fullConstruct ',' basicConstruct
              {
                $$ = list_append_node (N_CONSTRUCT_CLAUSE, $1,
                                       $basicConstruct);
              }
              ;

fullGraphPatternCondition : fullGraphPattern optWhereClause
                          {
                            $$ = list_append_node (N_FULL_MATCH_PATTERN,
                                                   $fullGraphPattern,
                                                   $optWhereClause);
                          }
                          ;

optionalClauses : %empty { $$ = NULL; }
                | optionalClauses OPTIONAL fullGraphPatternCondition
                { $$ = list_append_node (0, $1, $fullGraphPatternCondition); }
                ;

optGraphPattern : %empty { $$ = NULL; }
                | ';' graphPattern { $$ = $graphPattern; }
                ;

basicConstruct : objectConstructList setClauses removeClauses optWhenClause
               {
                 $$ = node_append_list_list_node (N_BASIC_CONSTRUCT,
                                                  $objectConstructList,
                                                  $setClauses, $removeClauses,
                                                  $optWhenClause);
               }
               ;

fullGraphPattern : basicGraphPatternLocation
                 { $$ = ast_v_new_node (0, 1, $basicGraphPatternLocation); }
                 | fullGraphPattern ',' basicGraphPatternLocation
                 {
                   $$ = list_append_node (0, $1, $basicGraphPatternLocation);
```

```
                 }
                 ;

walkPattern : relationshipPattern
            { $$ = ast_v_new_node (N_WALK_PATTERN, 1, $relationshipPattern); }
            | walkPattern ',' relationshipPattern
            {
              $$ = list_append_node (N_WALK_PATTERN, $1,
                                     $relationshipPattern);
            }
            ;

graphPattern : basicGraphPattern
             { $$ = ast_v_new_node (N_GRAPH_PATTERN, 1, $basicGraphPattern); }
             | graphPattern ',' basicGraphPattern
             {
               $$ = list_append_node (N_GRAPH_PATTERN, $1,
                                      $basicGraphPattern);
             }
             ;

objectConstructList : gid
                    | objectConstruct
                    {
                      $$ = ast_v_new_node (N_OBJECT_CONSTRUCT_LIST, 1,
                                           $objectConstruct);
                    }
                    | objectConstructList objectConstruct
                    {
                      $$ = list_append_node (N_OBJECT_CONSTRUCT_LIST, $1,
                                             $objectConstruct);
                    }
                    ;

setClauses : %empty { $$ = NULL; }
           | setClauses SET property ":=" expr
           {
             enum ast_setrem_type *type = cmalloc (sizeof *type);
             *type = SETREM_PROPERTY;
             struct ast_node *node = ast_v_new_node (N_SET_CLAUSE, 2,
                                                     $property, $expr);
             node->data = type;
             $$ = list_append_node (0, $1, node);
           }
           ; /* labels not yet supported */

removeClauses : %empty { $$ = NULL; }
              | removeClauses REMOVE property
              {
```

```
                enum ast_setrem_type *type = cmalloc (sizeof *type);
                *type = SETREM_PROPERTY;
                struct ast_node *node = ast_v_new_node (N_REMOVE_CLAUSE, 1,
                                                        $property);
                node->data = type;
                $$ = list_append_node (0, $1, node);
              }
              ; /* labels not yet supported */

optWhenClause : %empty { $$ = NULL; }
              | WHEN booleanCondition
              { $$ = ast_v_new_node (N_WHEN_CLAUSE, 1, $booleanCondition); }
              ;

basicGraphPatternLocation : basicGraphPattern optOnClause
                          {
                            $$ = ast_v_new_node (N_BASIC_MATCH_PATTERN, 2,
                                                 $basicGraphPattern,
                                                 $optOnClause);
                          }
                          ;

basicGraphPattern : nodePattern
                  | relationshipPattern
                  ;

objectConstruct : nodeConstruct
                | relationshipConstruct
                ;

property : IDENTIFIER[var] '.' IDENTIFIER[key]
         { $$ = ast_v_new_node (N_PROPERTY, 2, $var, $key); }
         ;

optOnClause : %empty { $$ = NULL; }
            | ON location
            { $$ = ast_v_new_node (N_ON_CLAUSE, 1, $location); }
            ;

location : gid
         | '(' fullGraphQuery ')' { $$ = $fullGraphQuery; }
         ;

nodePattern : '(' optIdentifier optMatchLabels optMatchProperties ')'
            {
              $$ = ast_v_new_node (N_NODE_PATTERN, 3, $optIdentifier,
                                   $optMatchLabels, $optMatchProperties);
            }
            ;
```

```
nodeConstruct : '(' optIdentifier optClone optGroupClause optDefLabels
                optDefProperties ')'
              {
                $$ = ast_v_new_node (N_NODE_CONSTRUCT, 5, $optIdentifier,
                                     $optClone, $optGroupClause,
                                     $optDefLabels, $optDefProperties);
              }
              ;

relationshipPattern : nodePattern[l] '-' edgeOrPathPattern "->" nodePattern[r]
                    {
                      $$ = $edgeOrPathPattern;
                      node_prepend_two ($$, $l, $r);
                    }
                    | nodePattern[l] "<-" edgeOrPathPattern '-' nodePattern[r]
                    {
                      $$ = $edgeOrPathPattern;
                      node_prepend_two ($$, $r, $l);
                    }
                    ;

relationshipConstruct : nodeConstruct[l] '-' edgeOrPathConstruct "->"
                        nodeConstruct[r]
                      {
                        $$ = $edgeOrPathConstruct;
                        node_prepend_two ($$, $l, $r);
                      }
                      | nodeConstruct[l] "<-" edgeOrPathConstruct '-'
                        nodeConstruct[r]
                      {
                        $$ = $edgeOrPathConstruct;
                        node_prepend_two ($$, $r, $l);
                      }
                      ;

edgeOrPathPattern : %empty
                  { $$ = ast_v_new_node (N_EDGE_PATTERN, 0, NULL); }
                  | '[' optIdentifier optMatchLabels optMatchProperties ']'
                  {
                    $$ = ast_v_new_node (N_EDGE_PATTERN, 3, $optIdentifier,
                                         $optMatchLabels,
                                         $optMatchProperties);
                  }
                  | '/' optQuantifier '@' optIdentifier optRegex
                    optMatchLabels optMatchProperties optCostClause '/'
                  {
                    enum ast_path_type *type = cmalloc (sizeof *type);
                    *type = PATH_OBJ;
```

```
                  $$ = ast_v_new_node (N_PATH_PATTERN, 6, $optQuantifier,
                                       $optIdentifier, $optRegex,
                                       $optMatchLabels,
                                       $optMatchProperties, $optCostClause);
                  $$->data = type;
                }
                | '/' optQuantifier optIdentifier optRegex
                  optCostClause '/'
                {
                  enum ast_path_type *type = cmalloc (sizeof *type);
                  *type = PATH_VIRT;
                  $$ = ast_v_new_node (N_PATH_PATTERN, 4, $optQuantifier,
                                       $optIdentifier, $optRegex,
                                       $optCostClause);
                  $$->data = type;
                }
                ;

edgeOrPathConstruct : %empty
                    { $$ = ast_v_new_node (N_EDGE_CONSTRUCT, 0, NULL); }
                    | '[' optIdentifier optClone optGroupClause
                      optDefLabels optDefProperties ']'
                    {
                      $$ = ast_v_new_node (N_EDGE_CONSTRUCT, 5,
                                           $optIdentifier, $optClone,
                                           $optGroupClause, $optDefLabels,
                                           $optDefProperties);
                    }
                    | '/' IDENTIFIER '/'
                    {
                      enum ast_path_type *type = cmalloc (sizeof *type);
                      *type = PATH_VIRT;
                      $$ = ast_v_new_node (N_PATH_CONSTRUCT, 1, $IDENTIFIER);
                      $$->data = type;
                    }
                    | '/' '@' optIdentifier optClone optDefLabels
                      optDefProperties '/'
                    {
                      enum ast_path_type *type = cmalloc (sizeof *type);
                      *type = PATH_OBJ;
                      $$ = ast_v_new_node (N_PATH_CONSTRUCT, 4,
                                           $optIdentifier, $optClone,
                                           $optDefLabels, $optDefProperties);
                      $$->data = type;
                    }
                    ;

optIdentifier : %empty { $$ = NULL; }
              | IDENTIFIER
```

```
              ;

optQuantifier : %empty { $$ = NULL; }
              | ALL
              {
                enum ast_quantifier_type *type = cmalloc (sizeof *type);
                *type = QUANTIFIER_ALL;
                $$ = ast_new_node (N_QUANTIFIER, type, 0, NULL);
              }
              | optNatural SHORTEST
              {
                enum ast_quantifier_type *type = cmalloc (sizeof *type);
                *type = QUANTIFIER_SHORTEST;
                $$ = ast_v_new_node (N_QUANTIFIER, 1, $optNatural);
                $$->data = type;
              }
              ;

optRegex : %empty { $$ = NULL; }
         | '<' regex '>' { $$ = $2; }
         ;

optClone : %empty { $$ = NULL; }
         | '=' IDENTIFIER
         { $$ = ast_v_new_node (N_CLONE_REFERENCE, 1, $IDENTIFIER); }
         ;

optMatchLabels : %empty { $$ = NULL; }
               | optMatchLabels ':' labelDisjunction
               {
                 $$ = list_append_node (N_MATCH_LABELS, $1,
                                        $labelDisjunction);
               }
               ;

optDefLabels : %empty { $$ = NULL; }
             | optDefLabels ':' IDENTIFIER
             { $$ = list_append_node (N_LABELS, $1, $IDENTIFIER); }
             ;

optMatchProperties : %empty { $$ = NULL; }
                   | '{' optMatchPropertyList '}'
                   { $$ = $optMatchPropertyList; }
                   ;

optDefProperties : %empty { $$ = NULL; }
                 | '{' optDefPropertyList '}'
                 { $$ = $optDefPropertyList; }
                 ;
```

```
optGroupClause : %empty { $$ = NULL; }
               | GROUP expr
               { $$ = ast_v_new_node (N_GROUP_CLAUSE, 1, $expr); }
               ;

labelDisjunction : IDENTIFIER
                 { $$ = ast_v_new_node (N_LABELS, 1, $IDENTIFIER); }
                 | labelDisjunction '|' IDENTIFIER
                 { $$ = list_append_node (N_LABELS, $1, $IDENTIFIER); }
                 ;

optMatchPropertyList : %empty { $$ = NULL; }
                     | matchPropertyList
                     ;

optDefPropertyList : %empty { $$ = NULL; }
                   | defPropertyList
                   ;

matchPropertyList : matchProperty
                  { $$ = ast_v_new_node (N_PROPERTIES, 1, $matchProperty); }
                  | matchPropertyList ',' matchProperty
                  {
                    $$ = list_append_node (N_PROPERTIES, $1, $matchProperty);
                  }
                  ;

matchProperty : IDENTIFIER[key] '=' expr
              { $$ = ast_v_new_node (N_KEY_VALUE, 2, $key, $expr); }
              ;

defPropertyList : defProperty
                { $$ = ast_v_new_node (N_PROPERTIES, 1, $defProperty); }
                | defPropertyList defProperty
                { $$ = list_append_node (N_PROPERTIES, $1, $defProperty); }
                ;

defProperty : IDENTIFIER[key] ":=" expr
            { $$ = ast_v_new_node (N_KEY_VALUE, 2, $key, $expr); }
            ;

regex : '(' regex ')'
      { $$ = ast_v_new_node (N_REGEX, 1, $2); }
      | ':' labelDisjunction
      { $$ = ast_v_new_node (N_REGEX, 1, $labelDisjunction); }
      | '~' IDENTIFIER[pname]
      { $$ = ast_v_new_node (N_REGEX, 1, $pname); }
      | regex '*'       /*%prec R_KLEENE*/
```

```
      {
        $$ = ast_v_new_node (N_REGEX, 1,
                             ast_v_new_node (N_RE_KLEENE, 1, $1));
      }
      | regex regex     %prec R_CONCAT
      {
        $$ = ast_v_new_node (N_REGEX, 1,
                             ast_v_new_node (N_RE_CONCAT, 2, $1, $2));
      }
      | regex '+' regex %prec R_UNION
      {
        $$ = ast_v_new_node (N_REGEX, 1,
                             ast_v_new_node (N_RE_UNION, 2, $1, $3));
      }
      ;

expr : IDENTIFIER
     { $$ = ast_v_new_node (N_EXPR, 1, $IDENTIFIER); }
     | IDENTIFIER[var] '.' IDENTIFIER[key]
     {
       $$ = ast_v_new_node (N_EXPR, 1,
                            ast_v_new_node (N_PROPERTY, 2, $var, $key));
     }
     | IDENTIFIER[var] ':' IDENTIFIER[lbl]
     {
       $$ = ast_v_new_node (N_EXPR, 1,
                            ast_v_new_node (N_LABEL, 2, $var, $lbl));
     }
     | prefixOp
     { $$ = ast_v_new_node (N_EXPR, 1, $prefixOp); }
     | postfixOp
     { $$ = ast_v_new_node (N_EXPR, 1, $postfixOp); }
     | binaryOp
     { $$ = ast_v_new_node (N_EXPR, 1, $binaryOp); }
     | IDENTIFIER '(' exprList ')'
     {
       $$ = ast_v_new_node (N_EXPR, 1,
                            node_append_list (N_FUNCTION, $IDENTIFIER,
                                              $exprList));
     }
     | EXISTS '(' query ')'
     { $$ = ast_v_new_node (N_EXPR, 1, $query); }
     | CASE caseBody END
     { $$ = ast_v_new_node (N_EXPR, 1, $caseBody); }
     | literal
     { $$ = ast_v_new_node (N_EXPR, 1, $literal); }
     | '(' expr ')'
     { $$ = ast_v_new_node (N_EXPR, 1, $2); }
     | '[' optExprList ']'
```

```
    { $$ = ast_v_new_node (N_EXPR, 1, $optExprList); }
    | expr[l] '[' expr[r] ']'
    {
      $$ = ast_v_new_node (N_EXPR, 1,
                           ast_v_new_node (N_GETELEM, 2, $l, $r));
    }
    ;

optExprList : %empty
            { $$ = ast_v_new_node (N_LIST, 0); }
            | exprList
            ;

exprList : expr
         { $$ = ast_v_new_node (N_LIST, 1, $expr); }
         | exprList ',' expr
         { $$ = list_append_node (N_LIST, $1, $expr); }
         ;

prefixOp : NOT expr
         {
           enum ast_unaryop_type *type = cmalloc (sizeof *type);
           *type = UNARYOP_NOT;
           $$ = ast_v_new_node (N_UNARY_OP, 1, $expr);
           $$->data = type;
         }
         | '-' expr %prec UMINUS
         {
           enum ast_unaryop_type *type = cmalloc (sizeof *type);
           *type = UNARYOP_MINUS;
           $$ = ast_v_new_node (N_UNARY_OP, 1, $expr);
           $$->data = type;
         }
         ;

postfixOp : expr IS NULL
          {
            enum ast_unaryop_type *type = cmalloc (sizeof *type);
            *type = UNARYOP_IS_NULL;
            $$ = ast_v_new_node (N_UNARY_OP, 1, $expr);
            $$->data = type;
          }
          | expr IS NOT NULL
          {
            enum ast_unaryop_type *type = cmalloc (sizeof *type);
            *type = UNARYOP_IS_NOT_NULL;
            $$ = ast_v_new_node (N_UNARY_OP, 1, $expr);
            $$->data = type;
          }
```

```
          ;

binaryOp : expr[l] '=' expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_EQUAL;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] "<>" expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_NOT_EQUAL;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] "<=" expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_LESS_EQUAL;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] ">=" expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_GREATER_EQUAL;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] '<' expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_LESS;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] '>' expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_GREATER;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
           $$->data = type;
         }
         | expr[l] SUBSET OF expr[r]
         {
           enum ast_binaryop_type *type = cmalloc (sizeof *type);
           *type = BINARYOP_SUBSET_OF;
           $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
```

```
  $$->data = type;
}
| expr[l] IN expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_IN;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] AND expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_AND;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] OR expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_OR;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] '+' expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_PLUS;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] '-' expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_MINUS;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] '*' expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_MULT;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
  $$->data = type;
}
| expr[l] '/' expr[r]
{
  enum ast_binaryop_type *type = cmalloc (sizeof *type);
  *type = BINARYOP_DIV;
  $$ = ast_v_new_node (N_BINARY_OP, 2, $l, $r);
```

```
            $$->data = type;
          }
          ;

caseBody : optExpr whenList optElse
          {
            $$ = node_append_list_node (N_CASE, $optExpr, $whenList, $optElse);
          }
          ;

optExpr : %empty { $$ = NULL; }
        | expr
        ;

whenList : %empty { $$ = NULL; }
          | whenList WHEN expr[w] THEN expr[t]
          {
            $$ = list_append_node (0, $1,
                                   ast_v_new_node (N_WHEN_THEN, 2, $w, $t));
          }
          ;

optElse : %empty { $$ = NULL; }
        | ELSE expr { $$ = $expr; }
        ;

optNatural : %empty { $$ = NULL; }
           | INTEGER
           ;

literal : INTEGER
        | FLOAT
        | STRING
        | TRUE
        {
          bool *boolean = cmalloc (sizeof *boolean);
          *boolean = true;
          $$ = ast_new_node (L_BOOLEAN, boolean, 0, NULL);
        }
        | FALSE
        {
          bool *boolean = cmalloc (sizeof *boolean);
          *boolean = false;
          $$ = ast_new_node (L_BOOLEAN, boolean, 0, NULL);
        }
        ;

%%
```

```
int main (int argc, char **argv)
{
  yyscan_t yyscanner;
  int retval, opt;
  bool verbose = false;

  setlocale (LC_ALL, "");

  if (yylex_init (&yyscanner) != 0)
    error (EXIT_FAILURE, errno, "Could not initialize lexical scanner");

  while ((opt = getopt (argc, argv, "dv")) != -1)
    switch (opt)
    {
      case 'd':
        yydebug = 1;
        break;
      case 'v':
        verbose = true;
        break;
      default:
        fprintf (stderr, "Usage: %s [-dv]\n", program_invocation_name);
        exit (EXIT_FAILURE);
    }

  retval = yyparse (yyscanner);

  if (yylex_destroy (yyscanner) != 0)
    error (EXIT_FAILURE, errno, "Could not free lexical scanner");

  if (root == NULL)
  {
    if (verbose)
      fputs ("No AST nodes found.\n", stdout);
  }
  else
  {
    if (verbose)
    {
      if (yydebug)
        fputs ("\n", stdout);

      ast_debug_print (root);
      fputs ("\n", stdout);
    }

    gcore_to_gremlin (root);

    ast_free (root);
```

```
  }
  return retval;
}

void yyerror (YYLTYPE *yylloc, __attribute__ ((unused)) yyscan_t yyscanner,
              const char *msg)
{
  if (yylloc->first_line == yylloc->last_line)
    if (yylloc->first_column == yylloc->last_column)
      error (0, 0, "Error at line %d, column %d: %s\n",
             yylloc->first_line, yylloc->first_column, msg);
    else
      error (0, 0, "Error at line %d, columns %d..%d: %s\n",
             yylloc->first_line, yylloc->first_column,
             yylloc->last_column, msg);
  else
    error (0, 0, "Error at %d,%d..%d,%d: %s\n",
           yylloc->first_line, yylloc->first_column,
           yylloc->last_line, yylloc->last_column, msg);
}

static struct ast_node *
list_append_node (enum ast_type type, struct ast_node *list,
                  struct ast_node *node)
{
  if (list == NULL)
    return ast_v_new_node (type, 1, node);

  if (node == NULL)
  {
    list->type = type;
    return list;
  }

  size_t nmemb = list->nmemb + 1;
  struct ast_node **children = crealloc (list->children,
                                         nmemb * sizeof *children);
  children[list->nmemb] = node;
  free (list);
  return ast_new_node (type, NULL, nmemb, children);
}

static struct ast_node *
node_append_list (enum ast_type type, struct ast_node *node,
                  struct ast_node *list)
{
  if (list == NULL)
    return ast_v_new_node (type, 1, node);
```

```
  size_t nmemb = 1 + list->nmemb;
  struct ast_node **children = cmalloc (nmemb * sizeof *children);
  children[0] = node;
  memcpy (children + 1, list->children, list->nmemb * sizeof *children);
  free (list->children);
  free (list);
  return ast_new_node (type, NULL, nmemb, children);
}

static struct ast_node *
node_append_list_node (enum ast_type type, struct ast_node *nodea,
                       struct ast_node *list, struct ast_node *nodeb)
{
  size_t nmemb = 0, na = 0, nl = 0;
  if (nodea != NULL)
    nmemb += (na = 1);
  if (list != NULL)
    nmemb += (nl = list->nmemb);
  if (nodeb != NULL)
    nmemb++;

  struct ast_node **children = cmalloc (nmemb * sizeof *children);
  if (nodea != NULL)
    children[0] = nodea;
  if (list != NULL)
  {
    memcpy (children + na, list->children, nl * sizeof *children);
    free (list->children);
    free (list);
  }
  if (nodeb != NULL)
    children[na + nl] = nodeb;

  return ast_new_node (type, NULL, nmemb, children);
}

static struct ast_node *
node_append_list_list_node (enum ast_type type, struct ast_node *nodea,
                            struct ast_node *lista, struct ast_node *listb,
                            struct ast_node *nodeb)
{
  size_t nmemb = 1, nla = 0, nlb = 0;
  if (lista != NULL)
    nmemb += (nla = lista->nmemb);
  if (listb != NULL)
    nmemb += (nlb = listb->nmemb);
  if (nodeb != NULL)
    nmemb++;
```

```
  struct ast_node **children = cmalloc (nmemb * sizeof *children);
  children[0] = nodea;
  if (lista != NULL)
  {
    memcpy (children + 1, lista->children, nla * sizeof *children);
    free (lista->children);
    free (lista);
  }
  if (listb != NULL)
  {
    memcpy (children + 1 + nla, listb->children, nlb * sizeof *children);
    free (listb->children);
    free (listb);
  }
  if (nodeb != NULL)
    children[1 + nla + nlb] = nodeb;

  return ast_new_node (type, NULL, nmemb, children);
}

static void
node_prepend_two (struct ast_node *node, struct ast_node *one,
                  struct ast_node *two)
{
  size_t nmemb = node->nmemb + 2;
  struct ast_node **children = cmalloc (nmemb * sizeof *children);

  children[0] = one;
  children[1] = two;
  memcpy (children + 2, node->children, node->nmemb * sizeof *children);
  free (node->children);

  node->nmemb = nmemb;
  node->children = children;
}
```

common.h

```
#pragma once

#include <stddef.h>

/**
 * Checked malloc.
 * @param size bytes to allocate
 * @return pointer to allocated memory
 */
void *
```

```
cmalloc (size_t size);

/**
 * Checked realloc.
 * @param ptr pointer to original memory
 * @param size bytes to be eventually allocated
 * @return pointer to new allocated memory
 */
void *
crealloc (void *ptr, size_t size);

/**
 * Checked strdup.
 * @param s string to duplicate
 * @return allocated string
 */
char *
cstrdup (const char *s);

/**
 * Concatenate two strings.  ‘a’ must not be and ‘b’ should be finally freed.
 * @param a first string to be appended to
 * @param b second string appended to the first
 * @return allocated concatenation
 */
char *
strconcat (char *a, const char *b);

/**
 * Checked asprintf.
 * @param fmt format string
 * @param ... arguments to be formatted
 * @return allocated string
 */
char *
casprintf (const char *fmt, ...);
```

common.c

```
#include <errno.h>
#include <error.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "common.h"
```

```
void * __attribute__((malloc))
cmalloc (size_t size)
{
  void *rv = malloc (size);
  if (rv == NULL)
    error (EXIT_FAILURE, errno, "Could not allocate memory");

  return rv;
}

void *
crealloc (void *ptr, size_t size)
{
  void *rv = realloc (ptr, size);
  if (rv == NULL)
    error (EXIT_FAILURE, errno, "Could not reallocate memory");

  return rv;
}

char * __attribute__((malloc))
cstrdup (const char *s)
{
  char *rv = strdup (s);
  if (rv == NULL)
    error (EXIT_FAILURE, errno, "Could not allocate memory");

  return rv;
}

char *
strconcat (char *a, const char *b)
{
  size_t lena = strlen (a);
  size_t lenb = strlen (b);
  char *rv = crealloc (a, lena + lenb + 1);

  memcpy (rv + lena, b, lenb + 1);
  return rv;
}

char * __attribute__((malloc)) __attribute__((format (printf, 1, 2)))
casprintf (const char *fmt, ...)
{
  char *rv;
  va_list ap;

  va_start (ap, fmt);
  errno = 0;
```

```
  if (vasprintf (&rv, fmt, ap) == -1)
    error (EXIT_FAILURE, errno, "Could not allocate memory");
  va_end (ap);

  return rv;
}
```

ast.h

```
#pragma once

#include <stddef.h>

/* enum ast_type {{{ */

enum ast_type
{
  _INVALID_AST_NODE_TYPE,

  L_IDENTIFIER,
  L_STRING,
  L_INTEGER,
  L_FLOAT,
  L_BOOLEAN,

  N_BASIC_CONSTRUCT,
  N_BASIC_MATCH_PATTERN,
  N_BINARY_OP,
  N_CASE,
  N_CLONE_REFERENCE,
  N_CONSTRUCT_CLAUSE,
  N_CONSTRUCT_MATCH,
  N_COST_CLAUSE,
  N_EDGE_CONSTRUCT,
  N_EDGE_PATTERN,
  N_EXPR,
  N_FULL_GRAPH_QUERY,
  N_FULL_MATCH_PATTERN,
  N_FUNCTION,
  N_GETELEM,
  N_GRAPH_PATTERN,
  N_GRAPH_CLAUSE,
  N_GRAPH_VIEW,
  N_GROUP_CLAUSE,
  N_HEAD_CLAUSES,
  N_KEY_VALUE,
  N_LABEL,
  N_LABELS,
```

```
  N_LIST,
  N_MATCH_CLAUSE,
  N_MATCH_LABELS,
  N_NODE_CONSTRUCT,
  N_NODE_PATTERN,
  N_OBJECT_CONSTRUCT_LIST,
  N_ON_CLAUSE,
  N_PATH_CLAUSE,
  N_PATH_CONSTRUCT,
  N_PATH_PATTERN,
  N_PROPERTIES,
  N_PROPERTY,
  N_QUANTIFIER,
  N_QUERY,
  N_REGEX,
  N_REMOVE_CLAUSE,
  N_RE_CONCAT,
  N_RE_KLEENE,
  N_RE_UNION,
  N_SETOP,
  N_SET_CLAUSE,
  N_UNARY_OP,
  N_WALK_PATTERN,
  N_WHEN_CLAUSE,
  N_WHEN_THEN,
  N_WHERE_CLAUSE,
};

/* }}} */

struct ast_node
{
  enum ast_type type;
  void *data;
  size_t nmemb;
  struct ast_node **children;
};

/* enums ast_*_type {{{ */

enum ast_setop_type
{
  SETOP_UNION,
  SETOP_INTERSECT,
  SETOP_MINUS,
};

enum ast_path_type
{
```

```
  PATH_OBJ,
  PATH_VIRT,
};

enum ast_quantifier_type
{
  QUANTIFIER_ALL,
  QUANTIFIER_SHORTEST,
};

enum ast_setrem_type
{
  SETREM_PROPERTY,
  SETREM_LABEL,
};

enum ast_unaryop_type
{
  UNARYOP_NOT,
  UNARYOP_MINUS,
  UNARYOP_IS_NULL,
  UNARYOP_IS_NOT_NULL,
};

enum ast_binaryop_type
{
  BINARYOP_EQUAL,
  BINARYOP_NOT_EQUAL,
  BINARYOP_LESS_EQUAL,
  BINARYOP_GREATER_EQUAL,
  BINARYOP_LESS,
  BINARYOP_GREATER,
  BINARYOP_SUBSET_OF,
  BINARYOP_IN,
  BINARYOP_AND,
  BINARYOP_OR,
  BINARYOP_PLUS,
  BINARYOP_MINUS,
  BINARYOP_MULT,
  BINARYOP_DIV,
};

/* }}} */

/**
 * Allocate a new AST node with the specified children array.
 * @param type node type
 * @param data node data or ‘NULL’
 * @param nmemb number of children
```

```
 * @param children array of child node pointers
 * @return constructed node
 */
struct ast_node *
ast_new_node (enum ast_type type, void *data, size_t nmemb,
              struct ast_node **children);

/**
 * Allocate a new AST node with the specified children.  Those child pointers
 * set to ‘NULL’ are ignored and ‘nmemb’ is reduced accordingly.
 * @param type node type
 * @param nmemb number of children
 * @param ... pointers to the child nodes
 * @return constructed node
 */
struct ast_node *
ast_v_new_node (enum ast_type type, size_t nmemb, ...);

/**
 * Recursively free a specified AST.
 * @param node root of the AST
 */
void
ast_free (struct ast_node *node);

void
ast_debug_print (struct ast_node *node);
```

ast.c

```
#include <assert.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "ast.h"
#include "common.h"

struct ast_node *
ast_new_node (enum ast_type type, void *data, size_t nmemb,
              struct ast_node **children)
{
  struct ast_node *node = cmalloc (sizeof *node);

  node->type = type;
  node->data = data;
```

```
  node->nmemb = nmemb;
  node->children = children;

  return node;
}

struct ast_node *
ast_v_new_node (enum ast_type type, size_t nmemb, ...)
{
  va_list ap;
  struct ast_node **children = cmalloc (nmemb * sizeof *children);

  va_start (ap, nmemb);
  for (size_t i = 0; i < nmemb; )
  {
    struct ast_node *child = va_arg (ap, struct ast_node *);
    if (child == NULL)
      nmemb--;
    else
      children[i++] = child;
  }
  va_end (ap);

  return ast_new_node (type, NULL, nmemb, children);
}

void
ast_free (struct ast_node *node)
{
  if (node == NULL)
    return;

  for (size_t i = 0; i < node->nmemb; i++)
    ast_free (node->children[i]);

  free (node->data);
  free (node->children);
  free (node);
}

/* static helpers {{{ */

static void
ast_print_setop_data (const char *t, enum ast_setop_type *type)
{
  const char *op;

  switch (*type)
  {
```

```
    case SETOP_UNION:
      op = "UNION";
      break;
    case SETOP_INTERSECT:
      op = "INTERSECT";
      break;
    case SETOP_MINUS:
      op = "MINUS";
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_path_data (const char *t, enum ast_path_type *type)
{
  const char *op;

  switch (*type)
  {
    case PATH_OBJ:
      op = "OBEJCTIFIED";
      break;
    case PATH_VIRT:
      op = "VIRTUAL";
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_quantifier_data (const char *t, enum ast_quantifier_type *type)
{
  const char *op;

  switch (*type)
  {
    case QUANTIFIER_ALL:
      op = "ALL";
      break;
    case QUANTIFIER_SHORTEST:
```

```
      op = "SHORTEST";
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_setrem_data (const char *t, enum ast_setrem_type *type)
{
  const char *op;

  switch (*type)
  {
    case SETREM_PROPERTY:
      op = "PROPERTY";
      break;
    case SETREM_LABEL:
      op = "LABEL";
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_unaryop_data (const char *t, enum ast_unaryop_type *type)
{
  const char *op;

  switch (*type)
  {
    case UNARYOP_NOT:
      op = "NOT";
      break;
    case UNARYOP_MINUS:
      op = "MINUS";
      break;
    case UNARYOP_IS_NULL:
      op = "IS NULL";
      break;
    case UNARYOP_IS_NOT_NULL:
      op = "IS NOT NULL";
```

```
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_binaryop_data (const char *t, enum ast_binaryop_type *type)
{
  const char *op;

  switch (*type)
  {
    case BINARYOP_EQUAL:
      op = "=";
      break;
    case BINARYOP_NOT_EQUAL:
      op = "<>";
      break;
    case BINARYOP_LESS_EQUAL:
      op = "<=";
      break;
    case BINARYOP_GREATER_EQUAL:
      op = ">=";
      break;
    case BINARYOP_LESS:
      op = "<";
      break;
    case BINARYOP_GREATER:
      op = ">";
      break;
    case BINARYOP_SUBSET_OF:
      op = "SUBSET OF";
      break;
    case BINARYOP_IN:
      op = "IN";
      break;
    case BINARYOP_AND:
      op = "AND";
      break;
    case BINARYOP_OR:
      op = "OR";
      break;
    case BINARYOP_PLUS:
      op = "+";
      break;
```

```
    case BINARYOP_MINUS:
      op = "-";
      break;
    case BINARYOP_MULT:
      op = "*";
      break;
    case BINARYOP_DIV:
      op = "/";
      break;

    default:
      assert (0);
  }

  printf ("%s (%s)\n", t, op);
}

static void
ast_print_type_data (struct ast_node *node)
{
  const char *t, *fmt = "%s\n";

  switch (node->type)
  {
    case L_IDENTIFIER:
      t = "IDENTIFIER";
      fmt = "%s (%s)\n";
      break;
    case L_STRING:
      t = "STRING";
      fmt = "%s (%s)\n";
      break;
    case L_INTEGER:
      printf ("INTEGER (%d)\n", *(int *) node->data);
      return;
    case L_FLOAT:
      printf ("FLOAT (%f)\n", *(double *) node->data);
      return;
    case L_BOOLEAN:
      printf ("BOOLEAN (%s)\n", *(bool *) node->data ? "true" : "false");
      return;

    case N_BASIC_CONSTRUCT:
      t = "BASIC_CONSTRUCT";
      break;
    case N_BASIC_MATCH_PATTERN:
      t = "BASIC_MATCH_PATTERN";
      break;
    case N_BINARY_OP:
```

```
    ast_print_binaryop_data ("BINARY_OP", node->data);
    return;
  case N_CASE:
    t = "CASE";
    break;
  case N_CLONE_REFERENCE:
    t = "CLONE_REFERENCE";
    break;
  case N_CONSTRUCT_CLAUSE:
    t = "CONSTRUCT_CLAUSE";
    break;
  case N_CONSTRUCT_MATCH:
    t = "CONSTRUCT_MATCH";
    break;
  case N_COST_CLAUSE:
    t = "COST_CLAUSE";
    break;
  case N_EDGE_CONSTRUCT:
    t = "EDGE_CONSTRUCT";
    break;
  case N_EDGE_PATTERN:
    t = "EDGE_PATTERN";
    break;
  case N_EXPR:
    t = "EXPR";
    break;
  case N_FULL_GRAPH_QUERY:
    t = "FULL_GRAPH_QUERY";
    break;
  case N_FULL_MATCH_PATTERN:
    t = "FULL_MATCH_PATTERN";
    break;
  case N_FUNCTION:
    t = "FUNCTION";
    break;
  case N_GETELEM:
    t = "GETELEM";
    break;
  case N_GRAPH_PATTERN:
    t = "GRAPH_PATTERN";
    break;
  case N_GRAPH_CLAUSE:
    t = "GRAPH_CLAUSE";
    break;
  case N_GRAPH_VIEW:
    t = "GRAPH_VIEW";
    break;
  case N_GROUP_CLAUSE:
    t = "GROUP_CLAUSE";
```

```
    break;
  case N_HEAD_CLAUSES:
    t = "HEAD_CLAUSES";
    break;
  case N_KEY_VALUE:
    t = "KEY_VALUE";
    break;
  case N_LABEL:
    t = "LABEL";
    break;
  case N_LABELS:
    t = "LABELS";
    break;
  case N_LIST:
    t = "LIST";
    break;
  case N_MATCH_CLAUSE:
    t = "MATCH_CLAUSE";
    break;
  case N_MATCH_LABELS:
    t = "MATCH_LABELS";
    break;
  case N_NODE_CONSTRUCT:
    t = "NODE_CONSTRUCT";
    break;
  case N_NODE_PATTERN:
    t = "NODE_PATTERN";
    break;
  case N_OBJECT_CONSTRUCT_LIST:
    t = "OBJECT_CONSTRUCT_LIST";
    break;
  case N_ON_CLAUSE:
    t = "ON_CLAUSE";
    break;
  case N_PATH_CLAUSE:
    t = "PATH_CLAUSE";
    break;
  case N_PATH_CONSTRUCT:
    ast_print_path_data ("PATH_CONSTRUCT", node->data);
    return;
  case N_PATH_PATTERN:
    ast_print_path_data ("PATH_PATTERN", node->data);
    return;
  case N_PROPERTIES:
    t = "PROPERTIES";
    break;
  case N_PROPERTY:
    t = "PROPERTY";
    break;
```

```
    case N_QUANTIFIER:
      ast_print_quantifier_data ("QUANTIFIER", node->data);
      return;
    case N_QUERY:
      t = "QUERY";
      break;
    case N_REGEX:
      t = "REGEX";
      break;
    case N_REMOVE_CLAUSE:
      ast_print_setrem_data ("REMOVE_CLAUSE", node->data);
      return;
    case N_RE_CONCAT:
      t = "RE_CONCAT";
      break;
    case N_RE_KLEENE:
      t = "RE_KLEENE";
      break;
    case N_RE_UNION:
      t = "RE_UNION";
      break;
    case N_SETOP:
      ast_print_setop_data ("SETOP", node->data);
      return;
    case N_SET_CLAUSE:
      ast_print_setrem_data ("SET_CLAUSE", node->data);
      return;
    case N_UNARY_OP:
      ast_print_unaryop_data ("UNARY_OP", node->data);
      return;
    case N_WALK_PATTERN:
      t = "WALK_PATTERN";
      break;
    case N_WHEN_CLAUSE:
      t = "WHEN_CLAUSE";
      break;
    case N_WHEN_THEN:
      t = "WHEN_THEN";
      break;
    case N_WHERE_CLAUSE:
      t = "WHERE_CLAUSE";
      break;

    default:
      assert (0);
  }

  printf (fmt, t, node->data);
}
```

```
static void
ast_debug_print_indent (struct ast_node *node, unsigned int indent)
{
  for (unsigned int i = 0; i < indent; i++)
    printf ("  ");

  ast_print_type_data (node);

  for (size_t i = 0; i < node->nmemb; i++)
    ast_debug_print_indent (node->children[i], indent + 1);
}

/* }}} */

void
ast_debug_print (struct ast_node *node)
{
  ast_debug_print_indent (node, 0);
}
```

gremlin.h

```
#pragma once

#include "ast.h"

/**
 * Transform the specified G-CORE AST into Gremlin and print the result to
 * standard output.
 * @param root AST root node
 */
void
gcore_to_gremlin (struct ast_node *root);
```

gremlin.c

```
#include <assert.h>
#include <error.h>
#include <stdio.h>
#include <stdlib.h>

#include "common.h"
#include "gremlin.h"

static void
```

```
gcore_match_disj_labels (struct ast_node *label_disjunction)
{
  assert (label_disjunction->type == N_LABELS);
  assert (label_disjunction->nmemb > 0);

  printf ("hasLabel('%s'", (char *) label_disjunction->children[0]->data);
  for (size_t i = 1; i < label_disjunction->nmemb; i++)
    printf (", '%s'", (char *) label_disjunction->children[i]->data);
  fputs (").", stdout);
}

static void
gcore_match_labels (struct ast_node *match_labels)
{
  assert (match_labels->type == N_MATCH_LABELS);

  for (size_t i = 0; i < match_labels->nmemb; i++)
    gcore_match_disj_labels (match_labels->children[i]);
}

static void
gcore_match_key_value (struct ast_node *key, struct ast_node *value)
{
  assert (key->type == L_IDENTIFIER);
  assert (value->type == N_EXPR);
  assert (value->nmemb == 1);

  struct ast_node *node = value->children[0];
  switch (node->type)
  {
    case L_IDENTIFIER:
      printf ("as('_tmp').coalesce(values('%s'), "
              "constant(null)).as('%s').select('_tmp').",
              (char *) key->data, (char *) node->data);
      break;
    default:
      error (EXIT_FAILURE, 0,
             "{key=VAL} for VAL not a new identifier not yet implemented");
  }
}

static void
gcore_match_properties (struct ast_node *properties)
{
  assert (properties->type == N_PROPERTIES);

  for (size_t i = 0; i < properties->nmemb; i++)
  {
    struct ast_node *kv = properties->children[i];
```

```
    assert (kv->type == N_KEY_VALUE);
    assert (kv->nmemb == 2);
    gcore_match_key_value (kv->children[0], kv->children[1]);
  }
}

static void
gcore_node_pattern (struct ast_node *identifier,
                    struct ast_node *match_labels,
                    struct ast_node *properties)
{
  if (identifier != NULL)
  {
    assert (identifier->type == L_IDENTIFIER);
    printf ("as('%s').", (char *) identifier->data);
  }

  if (match_labels != NULL)
    gcore_match_labels (match_labels);

  if (properties != NULL)
    gcore_match_properties (properties);
}

static void
gcore_basic_match_node (struct ast_node *node_pattern)
{
  assert (node_pattern->type == N_NODE_PATTERN);
  assert (node_pattern->nmemb <= 3);
  struct ast_node *id = NULL, *lbls = NULL, *props = NULL;

  for (size_t i = 0; i < node_pattern->nmemb; i++)
  {
    struct ast_node *node = node_pattern->children[i];
    switch (node->type)
    {
      case L_IDENTIFIER:
        assert (id == NULL);
        id = node;
        break;
      case N_MATCH_LABELS:
        assert (lbls == NULL);
        lbls = node;
        break;
      case N_PROPERTIES:
        assert (props == NULL);
        props = node;
        break;
      default:
```

```
      assert (0);
    }
  }

  gcore_node_pattern (id, lbls, props);
}

static void
gcore_edge_pattern (struct ast_node *node_pattern_from,
                    struct ast_node *node_pattern_to,
                    struct ast_node *identifier,
                    struct ast_node *match_labels,
                    struct ast_node *properties)
{
  gcore_basic_match_node (node_pattern_from);
  fputs ("outE().", stdout);
  gcore_node_pattern (identifier, match_labels, properties);
  fputs ("inV().", stdout);
  gcore_basic_match_node (node_pattern_to);
}

static void
gcore_basic_match_edge (struct ast_node *edge_pattern)
{
  assert (edge_pattern->type == N_EDGE_PATTERN);
  assert (edge_pattern->nmemb >= 2 && edge_pattern->nmemb <= 5);
  struct ast_node *id = NULL, *lbls = NULL, *props = NULL;

  for (size_t i = 2; i < edge_pattern->nmemb; i++)
  {
    struct ast_node *node = edge_pattern->children[i];
    switch (node->type)
    {
      case L_IDENTIFIER:
        assert (id == NULL);
        id = node;
        break;
      case N_MATCH_LABELS:
        assert (lbls == NULL);
        lbls = node;
        break;
      case N_PROPERTIES:
        assert (props == NULL);
        props = node;
        break;
      default:
        assert (0);
    }
  }
```

```
  gcore_edge_pattern (edge_pattern->children[0], edge_pattern->children[1],
                      id, lbls, props);
}

static void
gcore_path_obj_pattern (struct ast_node *node_pattern_from,
                        struct ast_node *node_pattern_to,
                        struct ast_node *quantifier,
                        struct ast_node *identifier,
                        struct ast_node *regex,
                        struct ast_node *match_labels,
                        struct ast_node *properties,
                        struct ast_node *cost_clause)
{
  error (EXIT_FAILURE, 0, "Path objects not implemented");
}

static void
gcore_path_virt_pattern (struct ast_node *node_pattern_from,
                         struct ast_node *node_pattern_to,
                         struct ast_node *quantifier,
                         struct ast_node *identifier,
                         struct ast_node *regex,
                         struct ast_node *cost_clause)
{
  error (EXIT_FAILURE, 0, "Virtual paths not yet implemented");
}

static void
gcore_basic_match_path (struct ast_node *path_pattern)
{
  assert (path_pattern->type == N_PATH_PATTERN);
  assert (path_pattern->nmemb >= 2 && path_pattern->nmemb <= 8);
  struct ast_node *quant = NULL, *id = NULL, *re = NULL, *lbls = NULL,
                  *props = NULL, *cost = NULL;

  for (size_t i = 2; i < path_pattern->nmemb; i++)
  {
    struct ast_node *node = path_pattern->children[i];
    switch (node->type)
    {
      case N_QUANTIFIER:
        assert (quant == NULL);
        quant = node;
        break;
      case L_IDENTIFIER:
        assert (id == NULL);
        id = node;
```

```
        break;
      case N_REGEX:
        assert (re == NULL);
        re = node;
        break;
      case N_MATCH_LABELS:
        assert (lbls == NULL);
        lbls = node;
        break;
      case N_PROPERTIES:
        assert (props == NULL);
        props = node;
        break;
      case N_COST_CLAUSE:
        assert (cost == NULL);
        cost = node;
        break;
      default:
        assert (0);
    }
  }

  switch (*(enum ast_path_type *) path_pattern->data)
  {
    case PATH_OBJ:
      gcore_path_obj_pattern (path_pattern->children[0],
                              path_pattern->children[1], quant, id, re, lbls,
                              props, cost);
      break;
    case PATH_VIRT:
      assert (path_pattern->nmemb <= 6);
      assert (lbls == NULL && props == NULL);
      gcore_path_virt_pattern (path_pattern->children[0],
                               path_pattern->children[1], quant, id, re,
                               cost);
      break;
    default:
      assert (0);
  }
}

static void
gcore_basic_match (struct ast_node *pattern, struct ast_node *on_clause)
{
  fputs ("V().", stdout);

  switch (pattern->type)
  {
    case N_NODE_PATTERN:
```

```
      gcore_basic_match_node (pattern);
      break;
    case N_EDGE_PATTERN:
      gcore_basic_match_edge (pattern);
      break;
    case N_PATH_PATTERN:
      gcore_basic_match_path (pattern);
      break;
    default:
      assert (0);
  }

  if (on_clause != NULL)
  {
    assert (on_clause->type == N_ON_CLAUSE);
    assert (on_clause->nmemb == 1);

    switch (on_clause->children[0]->type)
    {
      case L_IDENTIFIER:
        printf ("has('_on', '%s').", (char *) on_clause->children[0]->data);
        break;
      case N_FULL_GRAPH_QUERY:
        error (EXIT_FAILURE, 0, "ON (fullGraphQuery) not yet implemented");
        break;
      default:
        assert (0);
    }
  }
}

static void
gcore_where_expr (struct ast_node *node, char **by_steps)
{
  switch (node->type)
  {
    char *by;
    const char *cby;

    case L_IDENTIFIER:
      printf ("'%s'", (char *) node->data);
      cby = "by().";
      if (*by_steps == NULL)
        *by_steps = cstrdup (cby);
      else
        *by_steps = strconcat (*by_steps, cby);
      break;

    case N_PROPERTY:
```

```
  assert (node->nmemb == 2);
  printf ("'%s'", (char *) node->children[0]->data);
  by = casprintf ("by(values('%s').fold()).",
                  (char *) node->children[1]->data);
  if (*by_steps == NULL)
    *by_steps = by;
  else
  {
    *by_steps = strconcat (*by_steps, by);
    free (by);
  }
  break;

case N_LABEL:
  assert (node->nmemb == 2);
  error (EXIT_FAILURE, 0, "WHERE label expression not yet implemented");
  break;

case N_UNARY_OP:
  assert (node->nmemb == 1);
  switch (*(enum ast_unaryop_type *) node->data)
  {
    case UNARYOP_NOT:
      fputs ("__.not(", stdout);
      gcore_where_expr (node->children[0]->children[0], by_steps);
      fputs (")", stdout);
      break;
    case UNARYOP_MINUS:
      fputs ("-", stdout);
      gcore_where_expr (node->children[0]->children[0], by_steps);
      break;
    case UNARYOP_IS_NULL:
      gcore_where_expr (node->children[0]->children[0], by_steps);
      fputs (", eq(null)", stdout);
      break;
    case UNARYOP_IS_NOT_NULL:
      gcore_where_expr (node->children[0]->children[0], by_steps);
      fputs (", neq(null)", stdout);
      break;
    default:
      assert (0);
  }
  break;

case N_BINARY_OP:
  assert (node->nmemb == 2);
  const char *test;
  switch (*(enum ast_binaryop_type *) node->data)
  {
```

```
    case BINARYOP_EQUAL:
      test = "gcoreEq";
      break;
    case BINARYOP_NOT_EQUAL:
      test = "gcoreNeq";
      break;
    case BINARYOP_LESS_EQUAL:
      test = "gcoreLte";
      break;
    case BINARYOP_GREATER_EQUAL:
      test = "gcoreGte";
      break;
    case BINARYOP_LESS:
      test = "gcoreLt";
      break;
    case BINARYOP_GREATER:
      test = "gcoreGt";
      break;
    case BINARYOP_SUBSET_OF:
      test = "gcoreSubset";
      break;
    case BINARYOP_IN:
      test = "gcoreIn";
      break;
    default:
      error (EXIT_FAILURE, 0, "WHERE clause not yet fully implemented");
      test = "";
  }
  gcore_where_expr (node->children[0]->children[0], by_steps);
  printf (", test(%s, ", test);
  gcore_where_expr (node->children[1]->children[0], by_steps);
  fputs (")", stdout);
  break;

case L_INTEGER:
  fputs ("'_'", stdout);
  by = casprintf ("by(constant(%d)).", *(int *) node->data);
  if (*by_steps == NULL)
    *by_steps = by;
  else
  {
    *by_steps = strconcat (*by_steps, by);
    free (by);
  }
  break;

case L_FLOAT:
  fputs ("'_'", stdout);
  by = casprintf ("by(constant(%f)).", *(double *) node->data);
```

```
      if (*by_steps == NULL)
        *by_steps = by;
      else
      {
        *by_steps = strconcat (*by_steps, by);
        free (by);
      }
      break;

    case L_STRING:
      fputs ("'_'", stdout);
      by = casprintf ("by(constant('%s')).", (char *) node->data);
      if (*by_steps == NULL)
        *by_steps = by;
      else
      {
        *by_steps = strconcat (*by_steps, by);
        free (by);
      }
      break;

    default:
      error (EXIT_FAILURE, 0, "WHERE clause not yet fully implemented");
  }
}

static void
gcore_where (struct ast_node *expr)
{
  assert (expr->type == N_EXPR);
  assert (expr->nmemb == 1);
  struct ast_node *node = expr->children[0];
  char *by_steps = NULL;

  fputs ("where(", stdout);
  gcore_where_expr (node, &by_steps);
  fputs (").", stdout);

  if (by_steps != NULL)
  {
    fputs (by_steps, stdout);
    free (by_steps);
  }
}

static void
gcore_match (struct ast_node *full_match_pattern)
{
  assert (full_match_pattern->type == N_FULL_MATCH_PATTERN);
```

```
  assert (full_match_pattern->nmemb > 0);
  assert (full_match_pattern->children[0]->type == N_BASIC_MATCH_PATTERN);

  for (size_t i = 0; i < full_match_pattern->nmemb; i++)
  {
    struct ast_node *node = full_match_pattern->children[i];
    switch (node->type)
    {
      case N_BASIC_MATCH_PATTERN:
        assert (node->nmemb == 1 || node->nmemb == 2);
        gcore_basic_match (node->children[0],
                           node->nmemb == 2 ? node->children[1] : NULL);
        break;
      case N_WHERE_CLAUSE:
        assert (i + 1 == full_match_pattern->nmemb);
        assert (node->nmemb == 1);
        gcore_where (node->children[0]);
        break;
      default:
        assert (0);
    }
  }
}

static void
gcore_optional (struct ast_node *full_match_pattern)
{
  assert (full_match_pattern->type == N_FULL_MATCH_PATTERN);
}

static void
gcore_construct_object (struct ast_node *node)
{
  switch (node->type)
  {
    case N_NODE_CONSTRUCT:
      if (node->nmemb > 0 && node->children[0]->type == L_IDENTIFIER)
        printf ("'%s'", (char *) node->children[0]->data);
      break;
    case N_EDGE_CONSTRUCT:
    case N_PATH_CONSTRUCT:
      assert (node->nmemb >= 2);
      gcore_construct_object (node->children[0]);
      if (node->nmemb > 2 && node->children[2]->type == L_IDENTIFIER)
      {
        fputs (", ", stdout);
        printf ("'%s'", (char *) node->children[2]->data);
      }
      fputs (", ", stdout);
```

```
      gcore_construct_object (node->children[1]);
      break;
    default:
      assert (0);
  }
}

static void
gcore_construct (struct ast_node *basic_construct)
{
  assert (basic_construct->type == N_BASIC_CONSTRUCT);
  assert (basic_construct->nmemb > 0);
  struct ast_node *node = basic_construct->children[0];

  switch (node->type)
  {
    case N_OBJECT_CONSTRUCT_LIST:
      assert (node->nmemb > 0);
      gcore_construct_object (node->children[0]);
      for (size_t i = 1; i < node->nmemb; i++)
      {
        fputs (", ", stdout);
        gcore_construct_object (node->children[i]);
      }
      break;
    case L_IDENTIFIER:
      fputs ("'_'", stdout);
      break;
    default:
      assert (0);
  }

  for (size_t i = 1; i < basic_construct->nmemb; i++)
  {
    node = basic_construct->children[i];
    switch (node->type)
    {
      case N_SET_CLAUSE:
      case N_REMOVE_CLAUSE:
        break;
      case N_WHEN_CLAUSE:
        assert (i + 1 == basic_construct->nmemb);
        error (EXIT_FAILURE, 0, "WHEN clause not yet implemented");
        break;
      default:
        assert (0);
    }
  }
}
```

```
static void
gcore_construct_match (struct ast_node *construct_clause,
                       struct ast_node *match_clause)
{
  assert (construct_clause->type == N_CONSTRUCT_CLAUSE);
  assert (match_clause->type == N_MATCH_CLAUSE);

  assert (match_clause->nmemb > 0);
  gcore_match (match_clause->children[0]);

  for (size_t i = 1; i < match_clause->nmemb; i++)
    gcore_optional (match_clause->children[i]);

  assert (construct_clause->nmemb > 0);
  fputs ("select(", stdout);
  gcore_construct (construct_clause->children[0]);

  for (size_t i = 1; i < construct_clause->nmemb; i++)
  {
    fputs (", ", stdout);
    gcore_construct (construct_clause->children[i]);
  }

  fputs (").by(valueMap().with(WithOptions.tokens, WithOptions.labels)).",
         stdout);
}

static void
gcore_full_graph_query (struct ast_node *node)
{
  switch (node->type)
  {
    case N_SETOP:
      assert (node->nmemb == 2);
      gcore_full_graph_query (node->children[0]->children[0]);
      gcore_full_graph_query (node->children[1]->children[0]);
      break;
    case N_CONSTRUCT_MATCH:
      assert (node->nmemb == 2);
      gcore_construct_match (node->children[0], node->children[1]);
      break;
    case L_IDENTIFIER:
      break;
    default:
      assert (0);
  }
}
```

```
static void
gcore_query (struct ast_node *head_clauses, struct ast_node *full_graph_query)
{
  assert (head_clauses->type == N_HEAD_CLAUSES);
  assert (full_graph_query->type == N_FULL_GRAPH_QUERY);

  if (head_clauses->nmemb != 0)
    error (EXIT_FAILURE, 0, "Head clauses (PATH, GRAPH) not yet implemented");

  assert (full_graph_query->nmemb == 1);
  gcore_full_graph_query (full_graph_query->children[0]);
}

void
gcore_to_gremlin (struct ast_node *root)
{
  fputs ("_full_graph_t.withSideEffect('_', 0).", stdout);

  switch (root->type)
  {
    case N_GRAPH_VIEW:
      error (EXIT_FAILURE, 0, "GRAPH VIEW not yet implemented");
      break;
    case N_QUERY:
      assert (root->nmemb == 2);
      gcore_query (root->children[0], root->children[1]);
      break;
    default:
      assert (0);
  }

  fputs ("toList()\n", stdout);
}
```