Praktikum Grundlagen der Programmierung

Prof. Dr. Harald Räcke, Prof. Dr. Felix Brandt, J. Kranz, A. Reuss WS 2017/18

Übungsblatt 11 Abgabefrist: 22.1.2018 um 05:00 Uhr

Σ Punkte: 20 | Σ Bonuspunkte: 3

Aufgabe 11.1 (P) Abstrakte geometrische Körper

Weil wir mit ihr so viel Spaß hatten, betrachten wir erneut die Klassenhierarchie zur Modellierungsaufgabe geometrischer Körper. Diskutieren Sie, warum die Klasse `Grundflaeche` besser als abstrakte Klasse definiert werden sollte, und insbesondere, welche Methoden in der Klasse `Grundflaeche` direkt implementiert und welche lediglich deklariert werden sollten.

Passen Sie die Implementierung der Klasse `Grundflaeche` entsprechend an.

Aufgabe 11.2 (P) Erweiterung geometrischer Formen

In dieser Aufgabe sollen die Klassen der geometrischen Formen von früheren Aufgabenblättern (Blatt 9) erweitert werden.

- Die Klassen vom Typ `Prisma` und `Grundflaeche` sollen untereinander vergleichbar gemacht werden, indem die Klassen das Interface `Comparable<Prisma>` bzw. `Comparable<Grundflaeche>` implementieren [1].
  - Prismen werden nach Größe des Volumens verglichen;
  - `Grundflaechen` werden nach Größe der Fläche verglichen.
  - Zur Vereinfachung ignorieren wir Abweichungen durch Gleitkommaarithmetik und gehen davon aus, dass solche nicht vorkommen. D. h. die Flächen zweier Grundflächen sind gleich, wenn die Differenz exakt 0 ist. Die gleiche Annahme treffen wir für das Volumen von Prismen.
- Implementieren Sie dazu die Methode `public int compareTo(Prisma o)` bzw. `public int compareTo(Grundflaeche o)` in der Klasse `Prisma` bzw. `Grundflaeche`. Halten Sie sich dabei an die Vorgaben der Java-Dokumentation für das Interface `Comparable<T>` unter `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`. Beachten Sie insbesondere, dass ggf. eine `NullPointerException` geworfen wird.
- Testen Sie die Implementierung, indem Sie jeweils eine `LinkedList` mit `Grundflaeche`-Objekten und eine LinkedList mit `Prisma`-Objekten mittels `Collections.sort(List<T> list)` sortieren.
  Beispiel: `List<Prisma> pf = new LinkedList<Prisma>(); Collections.sort(pf);`
- Als zweiten Schritt sollen die Methoden `istQuadrat` und `zuQuadrat` in ein Interface `Quadrierbar` ausgelagert werden, welches nur von solchen `Grundflaechen` implementiert werden soll, die auch ein Quadrat darstellen können.
- Zuletzt erstellen wir ein weiteres Interface `Polygon` mit nur einer Methode `int getEckenAnzahl()`, welche die Zahl der in einer Grundfläche enthaltenen Ecken zurückgibt. Implementieren Sie das Interface `Polygon` für alle Grundflächen, die eine endliche Anzahl von Ecken besitzen.
- Testen Sie `Quadrierbar` und `Polygon`, indem Sie über eine `LinkedList` des Typen `Object`, die verschiedene `Prisma`- und `Grundflaeche`-Objekte enthält, iterieren und

[1] siehe dazu `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html`

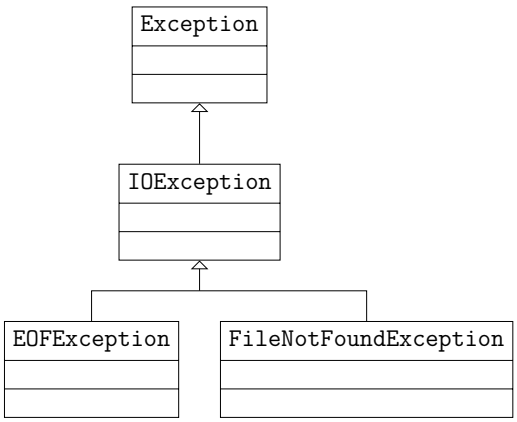
mittels `instanceof` die Verfügbarkeit beider Interfaces abfragen und, falls vorhanden, die aus diesen Interfaces ermittelbaren Informationen ausgeben.
- Deklarieren Sie alle Objektvariablen als `final`, wo das möglich ist.

Hinweis: Zur Verwendung der genannten Klassen und Methoden aus der Java-Standardbibliothek benötigen Sie die folgenden Imports.

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
```

Aufgabe 11.3 (P) Exceptions

Betrachten Sie folgenden Ausschnitt aus der Klassenhierarchie von Exceptions in Java und folgende Java-Implementierung der Klasse `ExceptionTest`:



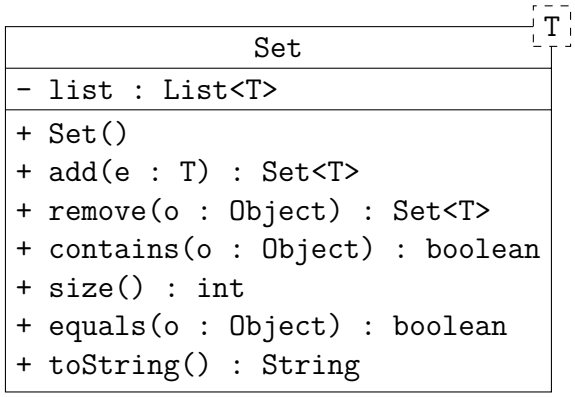

```
import java.io.*;
public class ExceptionTest {
   public static void main (String[] args) {
      try {
        // ...
      }
      catch (EOFException e) {
         System.out.println("EOFException");
      }
      catch (IOException e) {
         System.out.println("IOException");
      }
      catch (Exception e) {
         System.out.println("Exception");
      }
      System.out.println("ENDE");
   }
}
```

1. An der durch „...“ gekennzeichneten Stelle im `try`-Block stehe ein Programmstück, durch das Exceptions vom Typen `EOFException`, `IOException` oder aber `FileNotFoundException` geworfen werden können.
   Was wird bei Ausführung der `main`-Methode ausgegeben, falls dabei im `try`-Block
   i) als erstes eine Ausnahme vom Typ `EOFException` geworfen wird,
   ii) als erstes eine Ausnahme vom Typ `FileNotFoundException` geworfen wird, oder
   iii) gar keine Ausnahme geworfen wird?
2. Was wird bei Ausführung der `main`-Methode ausgegeben, falls dabei im `try`-Block als erste Ausnahme eine Division durch 0 auftritt?

Aufgabe 11.4 (P) Unveränderliche Mengen

Implementieren Sie eine *unveränderliche* Datenstruktur, die das Verhalten einer Menge abbildet. Unveränderlich bedeutet, dass jede Membervariable `final` sein muss. Möchte man also ein Element `e` der Menge `E` hinzufügen, so darf/kann nicht die Menge `E` selber verändert werden, sondern es muss ein neues Objekt `E'` erzeugt werden, welches das neue

Element `e` beinhaltet sowie alle alten Elemente der Menge `E`, d. h. `E' = E` $\cup$ {`e`}. Um die Elemente einer Menge zu repräsentieren, verwenden wir eine Liste. Da wir gefordert hatten, dass die Menge unveränderlich sein muss, fordern wir auch, dass die Liste unveränderlich sein muss. D. h. alle Membervariablen der Liste müssen **`final`** sein. Für die Listenimplementierung können Sie sich an Ihren eigenen Listenimplementierungen sowie an den Lösungsvorschlägen aus vorherigen Aufgaben orientieren. Die Menge selber und somit natürlich auch die Liste soll Elemente von einem generischen Typ `T` enthalten.



Im Folgenden sei `s` ein `Set`-Objekt von einem generischen Typ `T`:

- Der parameterlose Konstruktor erstellt ein Objekt, welches eine leere Menge repräsentiert.
- Die Methode `s.add(T e)` gibt ein `Set`-Objekt zurück, das das Element `e` enthält sowie alle Elemente, die in `s` enthalten sind. Ist das Element `e` schon in `s` enthalten, dann soll `s` zurückgegeben werden. Ist `e` gleich **`null`**, dann soll eine `NullPointerException` geworfen werden.
- Die Methode `s.remove(Object o)` gibt ein `Set`-Objekt zurück, das alle Element von `s` enthält außer dem Element `o`. Ist `o` gleich **`null`**, dann soll eine `NullPointerException` geworfen werden.
- Die Methode `s.contains(Object o)` gibt **`true`** zurück, wenn das Element `o` in der Menge enthalten ist, und andernfalls **`false`**.
- Die Methode `size` gibt die Kardinalität der Menge zurück.
- Die Methode `s.equals(Object o)` erfüllt die üblichen Eigenschaften, die der Java-Standard fordert. Des Weiteren wird **`true`** zurückgegeben, wenn die Mengen `s` und `o` die gleichen Elemente enthalten. Andernfalls wird **`false`** zurückgegeben. Beachten Sie die üblichen Eigenschaften einer Menge wie z. B. $e \in E \implies E = E \cup \{e\}$ oder $e \notin E \implies E = E \setminus \{e\}$.
- Die Methode `s.toString()` gibt einen String der Form $\{x_1, \ldots, x_n\}$ zurück, wenn die Menge `s` die Elemente x_i, $1 \leq i \leq n$, enthält und n die Kardinalität der Menge ist. Der zurückgegebene String ist also eine String-Repräsentation aller enthaltener Elemente der Menge.

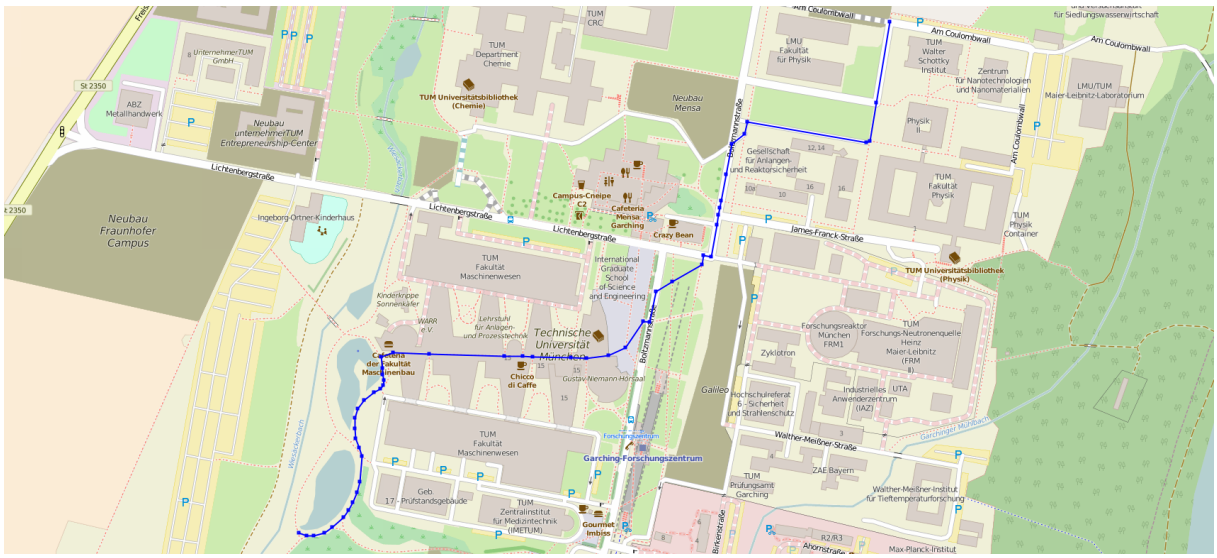
Zusatzaufgabe: Implementieren Sie einen Iterator für die Menge, d. h. die Klasse `Set<T>` muss das Interface `Iterable<T>` implementieren (die Membervariablen des Iterators müssen nicht **`final`** sein).

Hinweis 1: Zusätzliche Members (Attribute, Konstruktoren, Methoden) sind erlaubt.

Hinweis 2: Eine Datenstruktur ist unveränderlich, wenn es keine Möglichkeit gibt, diese nach der Erstellung zu verändern. Mit `final` kann sichergestellt werden, dass eine Variable nach der Initialisierung nicht mehr verändert werden kann – die Variable ist also konstant. Um eine unveränderliche Datenstruktur zu bekommen, muss also nur noch sichergestellt werden, dass jedes Objekt, auf das eine Variable aus der Datenstruktur zeigt, nicht mehr verändert werden kann. Wenn dies induktiv sichergestellt wurde, dann ist die Datenstruktur unveränderlich.

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Wenn Sie Ihre Dateien als Archiv abgeben möchten, verwenden Sie bitte ausschließlich `.zip`-Dateien. Sie dürfen Packages verwenden. Achten Sie darauf, dass Ihr Code kompiliert. Bitte vermerken Sie aus Datenschützgründen nicht Ihren Namen oder Ihre Matrikelnummer im Code. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 11.5 (H) Nogivan [7 Punkte + 1 Bonuspunkte]



Ziel dieser Aufgabe ist es, den Algorithmus von Dijkstra in einem praktischen Anwendungsbeispiel einzusetzen. Wir werden dazu eine Route minimaler Länge in einer Karte von OpenStreetMap[2] berechnen. OpenStreetMap ist ein Projekt, welches daran arbeitet, mithilfe der Community eine quelloffene Karte für die gesamte Welt zu erstellen. In Deutschland sind die Kartendaten schon sehr präzise, vielerorts sogar besser als kommerzielle Angebote.

Die Karte wird in mehreren Formaten angeboten. Da wir kürzeste Wege berechnen wollen, können wir mit Bilddaten nichts anfangen; wir benötigen stattdessen den Graphen, dessen Kanten die Straßen repräsentieren. Mit diesem Blatt wird ein kleiner Ausschnitt der Karte vom Campus Garching verteilt. Da die OpenStreetMap-Daten sehr umfangreich sind, müssen Sie sich größere Ausschnitte der Karte selbst herunterladen. Dies können Sie tun, indem Sie auf `http://www.openstreetmap.org` auf *Export* klicken und einen Teil der Karte wählen. Nutzen Sie anschließend den Button *Overpass API*, der auch den Download größerer Ausschnitte erlaubt.

Die Daten befinden sich im OSM-Format, welches den XML-Standard verwendet; der genaue Aufbau wird unter `http://wiki.openstreetmap.org/wiki/OSM_XML` beschrieben. Nutzen Sie einen Texteditor, um sich mit dem Format vertraut zu machen. Wollen Sie sich eine Karte im OSM-Format ansehen bzw. sie verändern, können Sie das Programm JOSM[3] verwenden.

Gehen Sie bei Ihrer Implementierung vor wie folgt:

[2] `https://www.openstreetmap.org`
[3] `https://josm.openstreetmap.de/`

1. Implementieren Sie zunächst die Methode `parseFile(String fileName)` der Java-Klasse `MapParser`, die eine Instanz der Klasse `MapGraph` aus einer Datei im OSM-Format erstellt. Die Klasse `MapGraph` repräsentiert den Graphen der Straßen und Wege, auf dem Sie später minimale Wege suchen möchten. Sie können zum Einlesen der Datei einen XML-Parser verwenden – ein Tutorial dazu finden Sie unter `http://www.tutorialspoint.com/java_xml/`. Achten Sie unbedingt darauf, den *SAX Parser* zu verwenden (etwas weiter hinten im Tutorial beschrieben), da die OpenStreetMap-Daten extrem umfangreich sind (Oberbayern z.B. ist fast 3GB groß). Achten Sie beim Einlesen darauf, möglichst nicht benötigte Knoten (z.B. solche, die zu Gebäuden gehören) zu ignorieren, um Arbeitsspeicher zu sparen. Sie erkennen derartige Knoten daran, dass sie zu keinem Weg mit gesetztem `highway`-Attribut gehören.

2. Implementieren Sie nun die Methode `int distance()` der Klasse `MapPoint`, die den Abstand zweier Punkte auf der Karte berechnet. Eine Beschreibung, die erklärt, wie Sie aus den Werten von Längengrad und Breitengrad den Abstand ermitteln, können Sie z.B. unter `http://www.movable-type.co.uk/scripts/latlong.html` finden.

3. Implementieren Sie anschließend eine Methode `OSMNode closest(MapPoint p)` der Klasse `MapGraph`. Die Methode ermittelt zu einem gegebenen Kartenpunkt p denjenigen Knoten des Graphens, der dem Punkt am nähesten ist. Gibt es mehrere Knoten mit dem gleichen kleinsten Abstand zu p, so wird derjenige Knoten von ihnen zurückgegeben, der die kleinste Id hat. Implementieren Sie außerdem die Methode `boolean hasEdge(OSMNode from, OSMNode to)`, die zurückgibt, ob eine bestimmte Kante im Graphen vorhanden ist.

4. Implementieren Sie die Methode `RoutingResult route(MapPoint from, MapPoint to)` der Klasse `MapGraph`. Nutzen Sie den Algorithmus von Dijkstra, um alle kleinsten Entfernungen von dem Knoten, der dem Kartenpunkt `from` am nähesten ist, zu berechnen. Die Methode gibt ein Objekt des Typs `RoutingResult` zurück, welches die Entfernung der zwischen Start- und Endknoten (entlang des Weges) und den entsprechenden Weg enthält. Kann kein Weg gefunden werden, so soll `null` zurückgegeben werden. Testen Sie anschließend Ihre Implementierung mit **kleinen** Beispielen. **Hinweis**: Sie können für die Prioritätswarteschlange die mitgelieferte Klasse `BinomialHeap` verwenden, die weiter unten beschrieben ist.

5. Implementieren Sie eine Methode `void write(RoutingResult rr)` der Java-Klasse `GPXWriter`, die eine Route im GPX-Format ausgibt. Nutzen Sie dazu die vorgegebene Methode `void writeLine(String line)`, die den übergebenen String in die Datei schreibt, die die Klasse im Konstruktor übergeben bekommt. Eine Beschreibung des GPX-Formates lässt sich z.B. unter `https://de.wikipedia.org/wiki/GPS_Exchange_Format` finden. Nutzen Sie Ihre Implementierung anschließend, um eine kürzeste Route in eine Datei zu schreiben und zu betrachten. Sie können eine GPX-Datei z.B. mit dem Programm GpsPrune[4] öffnen.

In der Klasse `MapGraph` sind bereits Attribute zur Speicherung des Graphen vorgegeben:

- Das Attribut `nodes` bildet die ID eines Knotens auf eine Instanz der Klasse `OSMNode` ab.

[4] `http://activityworkshop.net/software/gpsprune/`

- Das Attribut `edges` bildet die ID eines Knotens auf eine Menge von Kanten ab. Eine Kante enthält den Zielknoten und das jeweilige `OSMWay`-Objekt.

Abbildungen lassen sich in Java mit sog. *HashMaps* realisieren. Sie müssen nicht verstehen, wie diese Datenstrukturen genau funktionieren; wichtig ist hier nur, dass Sie sie verwenden können, um Schlüssel auf Werte abzubilden. Fügt man ein Pärchen mittels `put()` in eine solche Abbildung ein, kann man später mittels `get()` herausfinden, auf welchen Wert ein Schlüssel abbildet. Mengen dagegen werden über *HashSets* realisiert. Sie können z.B. testen, ob ein Element in einer Menge vorhanden ist, oder über alle Elemente einer Menge iterieren. Für eine genaue Referenz der Klassen `HashMap` und des `HashSet` sei auf die Java-Dokumentatio[5] [6] verwiesen.

Zur Auswahl von Verbindungen kürzester Distanz im Dijkstra-Algorithmus können Sie den Binomialheap verwenden, der bereits im Package `heap` in der Klasse `BinomialHeap` implementiert ist und die folgenden Operationen bietet:

- `public BinomialHeap()`:
  Der Konstruktor initialisiert den Binomialheap mit 0 Elementen.
- `public int getSize()`:
  Gibt die Anzahl enthaltener Elemente zurück.
- `public T poll()`:
  Gitb das kleinste Element (bzgl. der `compareTo`-Methode des Typs `T`) zurück und entfernt es aus dem Binomialheap. `T` ist der generische Typ der gespeicherten Elemente.
- `public T peek()`:
  Gibt das kleinste Element zurück, ohne es aus dem Binomialheap zu entfernen.
- `public Object insert(T element)`:
  Fügt das übergebene Element in den Binomialheap ein. Die Methode darf *nicht* mit bereits enthaltenen Elementen aufgerufen werden. Das zurückgegebene Objekt identifiziert die Speicherstelle und wird bei der folgenden Operation genutzt.
- `public void replaceWithSmallerElement(Object handle, T elementNew)`:
  Ersetzt das Element, für welches beim Einfügen das Objekt `handle` zurückgegeben wurde, mit dem übergebenen Element. Das übergebene Element muss kleiner sein als dasjenige, bei dessen Einfügen `handle` zurückgegeben wurde.

Bonusaufgabe: Testen Sie Ihre Implementierung nun mit etwas größeren Beispielen. Sie werden feststellen, dass die Suche nach dem kürzesten Weg unangemessen langwierig ist – insbesondere für nah beieinander liegende Kartenpunkte[7]. Überlegen Sie sich, wie Sie den Algorithmus beschleunigen können. Nutzen Sie dabei folgende Denkanstöße:

- Überlegen Sie sich, wie Sie zusätzlich den Luftlinien-Abstand zweier Punkte sinnvoll nutzen können.

[5] `https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html`
[6] `https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html`
[7] Trennen Sie die Laufzeit für das Einlesen von der Laufzeit für die Pfadsuche. Die Laufzeit für das Einlesen ist in unserem Fall unrealistisch lang, da das OSM-Format nicht auf diesen Anwendungszweck optimiert ist.

- Ein entscheidender Grund für die schlechte Performance des Algorithmus sind die Aufrufe der `replaceWithSmallerElement()`-Operation. Versuchen Sie, die Anzahl der Aufrufe zu reduzieren!

Sie sollten nun in der Lage sein, kürzeste Wege auf der Karte von Oberbayern[8] hinreichend effizient zu berechnen.

Aufgabe 11.6 (H) Ein Besucher von Format [8 Punkte]

In dieser Aufgabe geht es darum, Programme zu formatieren. Schreiben Sie dazu eine Klasse `FormatVisitor`, die mithilfe des Visitor-Patterns eine `String`-Darstellung für jede Klasse des Syntaxbaumes liefert. Der Visitor soll das Ergebnis der Formatierung über die Methode `public String getResult()` anbieten.

Ihr `FormatVisitor` sollte das Programm an den Google Style Guide angelehnt formatieren. Sie müssen nicht jede Regel umsetzen, die Formatierung sollte jedoch lesbar und konsistent sein.

Gehen Sie bei dieser Aufgabe in den folgenden Schritten vor, die Ihnen jeweils bereits Teilpunkte bringen.

1. Implementieren Sie den `FormatVisitor` zunächst derart, dass er Subausdrücke von Ausdrücken und Bedingungen vollständig klammert.

2. Fügen Sie **wo nötig** Implementierungen der Methode `public String toString()` den Klassen des Syntaxbaums hinzu, sodass **jeder** Knoten formatiert ausgegeben werden kann.

3. Passen Sie den Visitor derart an, dass Ausdrücke und Bedingungen *sinnvoll* geklammert werden. Insbesondere dürfen nicht überall Klammern stehen, obwohl diese *in Java*[9] oft nicht benötigt werden. Orientieren Sie sich an den Tests in der Klasse `FormatTest`. Sie dürfen im Rahmen dieser Aufgabe und zur Umsetzung der Klammerung in den Klassen `Expression`, `Condition` sowie allen Unterklassen die Methode `public int firstLevelPriority()`, welche eine Operatorenpriorität auf der obersten Ebene des Ausdrucks zurückliefert, implementieren. Weisen Sie den Operatoren sinnvolle Prioritäten zu und nutzen Sie diese, um auf unnötige Klammern in der Stringdarstellung zu verzichten.

Hinweis: In der folgenden Aufgabe werden Sie die Klassen des Syntaxbaums erweitern. Ihr `FormatVisitor` darf diese Erweiterung ebenfalls formatieren können, muss dies aber nicht. Die Aufgaben bauen also nicht aufeinander auf.

Aufgabe 11.7 (H) Arrastasias Horrorhaufen [5 Punkte + 2 Bonuspunkte]

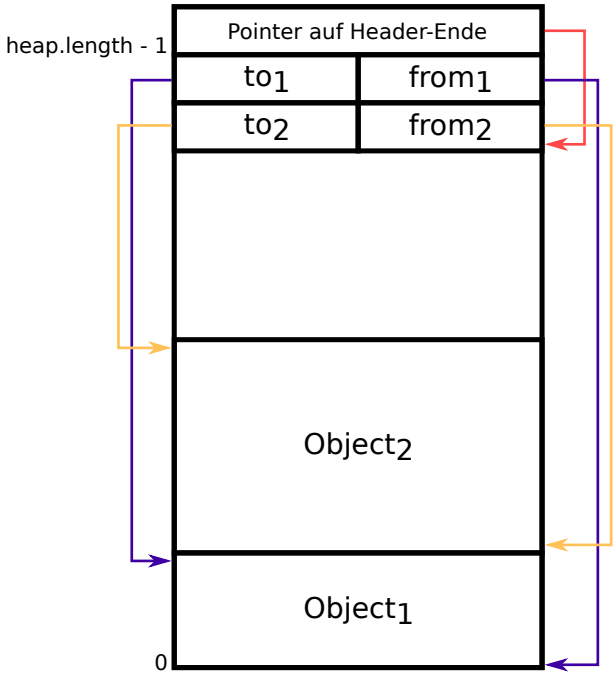
In dieser Aufgabe werden wir die Steckmaschine um Heap-Speicher erweitern. Dies erlaubt es uns, den Compiler auf die Sprache *MiniJava+* zu erweitern, welche zusätzlich

[8] `http://download.geofabrik.de/europe/germany/bayern.html`

[9] Richtig, der MiniJava-Parser kennt keine Operator-Prioritäten; wir gehen hier aber von echtem Java aus.

zu *MiniJava mit Funktionen* Unterstützung für Arrays bietet. Wie üblich dürfen Sie Ihre Implementierung entweder auf Ihrer eigenen Lösung oder offziellen Musterlösung des vorherigen Blattes aufbauen.

Wir erweitern zunächst die Steckmaschine um einen Heap. Der Heap ist grundlegend nach folgender Abbildung aufgebaut:



Prinzipiell ist der Heap ein weiteres Array aus Integern; wir wählen hier eine Größe von 128. Er verfügt über zwei Teile - im oberen Teil findet sich eine Management-Datenstruktur (hier als *Header* bezeichnet), während die eigentlichen Objekte im unteren Teil angelegt werden. Der Header beginnt in der obersten Zelle mit einem Pointer auf die letzte aktuell allokierte Management-Zelle. Jede Management-Zelle hat eine Breite von 32 Bit und ist in zwei gleich große Teile aufgeteilt - einen Pointer auf den Anfang und einen Pointer auf das Ende des jeweils beschriebenen Objekts. Wir allokieren in dieser Aufgabe nur Speicher, den wir niemals wieder freigeben. Ihre Aufgabe ist es zunächst, diese Allokation sowie den Zugriff auf Objekte als neue Steck-Instruktionen zu implementieren.

Implementieren Sie die folgenden zusätzlichen Instruktionen in der Steckmaschine:

Instruktion	**Immediate**	**Beschreibung**
`LDH`	keins	Lädt eine Heap-Zelle von Offset o_2 des von o_1 referenzierten Heap-Objekts auf den Stack
`STH`	keins	Schreibt das Stack-Element o_3 in das von o_1 referenzierte Heap-Objekt an Offset o_2
`ALLOCH`	keins	Allokiert ein Objekt der Größe o_1 auf dem Heap und hinterlässt die Heap-Referenz oben auf dem Stack

Es sei hier wiederum o_1 das oberste Stack-Element, o_2 das zweit-oberste Stack-Element usw. Die Elemente o_i werden – wie immer – vom Stack entfernt. Die Instruktion `ALLOCH` legt eine weitere Zeile im Management-Teil des Heaps an, welche das neue Objekt beschreibt. Der Index dieser neuen Zeile stellt die Heap-Referenz dar, die anschließend

auf den Stack geladen wird. Achten Sie bei Ihrer Implementierung der neuen Instruktionen darauf, Fehler abzufangen. Nutzen Sie die bereits bekannte Methode `private void error("Fehlertext");` und ersetzen Sie *Fehlertext* durch eine verständliche Beschreibung des aufgetretenen Fehlers.

Wir erweitern nun die Grammatik von *MiniJava mit Funktionen* (vgl. Blatt 10) zu *MiniJava+* entsprechend der Abbildung 1. Die Grammatik wurde hier entsprechend der Syntax in Java um Arrays erweitert. Im Unterschied zu Java wird jedoch auf die Länge eines Arrays `array` über `length(array)` statt über `array.length` zugegriffen. Erweitern Sie nun den Parser und den Code-Generator um Unterstützung für Arrays. Gehen Sie dabei wie folgt vor:

1. Erweitern Sie die Menge an Klassen des Syntaxbaums derart, dass sie der neuen Grammatik entspricht.
2. Erweitern Sie den Parser, sodass dieser die neue Grammatik akzeptiert.
3. Erweitern Sie den Code-Generator um Unterstützung für Arrays. **Verzichten** Sie dabei zunächst auf eine Unterstützung der Längenbestimmung von Arrays. Allokieren Sie für ein Array der Länge l ein Heap-Objekt der Größe l und legen Sie je ein Array-Element pro Heap-Zelle ab.
4. **(Anspruchsvolle Bonusaufgabe)** Implementieren Sie schließlich Unterstützung für die Bestimmung der Länge eines Arrays. Denken Sie sich dazu ein passendes Format für Arrays im Heap aus, welches die Längenbestimmung erlaubt.

Hinweis: Der Compiler unterstützt nun unterschiedliche Typen von Variablen und Ausdrücken. Sie müssen in dieser Aufgabe die Typkorrektheit des Programms nicht sicherstellen. Das Verhalten von typ-inkorrekten Programmen ist undefiniert.

Hinweis: Testen Sie jeden Schritt Ihrer Implementierung. Nutzen Sie dazu die um Tests für Arrays erweiterte Klasse `CompilerTest`, die mit diesem Blatt verteilt wird.

```
<program> ::= <function>*

<function> ::= <type> <name> ( <params> ) { <decl>* <stmt>* }

<params>  ::= ϵ | (<type> <name>)(, <type> <name>)*

<decl>    ::= <type> <name> (, <name> )* ;

<type>    ::= int
            | int[]

<stmt>    ::= ;
            | { <stmt>* }
            | <name> = <expr>;
            | <expr> [ <expr> ] = <expr>;
            | <name> = read();
            | write( <expr> );
            | if (<cond>) <stmt>
            | if (<cond>) <stmt> else <stmt>
            | while(<cond>) <stmt>
            | return <expr>;

<expr>    ::= <number>
            | <name>
            | new int [ <expr> ]
            | <name> [ <expr> ]
            | <name> ( (ϵ | <expr>(, <expr>)*) )
            | length ( <expr> )
            | (<expr>)
            | <unop> <expr>
            | <expr> <binop> <expr>

<unop>    ::= -

<binop>   ::= - | + | * | / | %

<cond>    ::= true | false
            | (<cond>)
            | <expr> <comp> <expr>
            | <bunop> (<cond>)
            | <cond> <bbinop> <cond>

<comp>    ::= == | != | <= | < | >= | >

<bunop>   ::= !

<bbinop>  ::= && | ||
```

Abbildung 1: Grammatik der Sprache MiniJava+