Praktikum Grundlagen der Programmierung

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

Übungsblatt 10 Abgabefrist: 15.1.2018 um 05:00 Uhr

Σ Punkte: 20

Aufgabe 10.1 (P) Polymorphie

Diese Aufgabe stammt aus einer Klausur vergangener Jahre. Gegeben sind die folgenden beiden Klassen:

```
1  class A {
2    int x() { return 0; }
3    int y = 3;
4    static void f(A a) { System.out.print(a.x() + "-"); a.g(a); }
5    static void f(B b) { System.out.print(b.y + "-"); b.g(b); }
6    void g(A a) { System.out.println(a.y); }
7    void g(B b) { System.out.print(b.y + "-"); this.f(this); }
8  }
9
10 class B extends A {
11   int x() { return 1; }
12   int y = 9;
13   static void f(A a) { System.out.print(a.y + "-"); (new A()).g(a); }
14   static void f(B b) { System.out.print(b.y + "-"); ((B)b).f((A)b); }
15   void g(A a) { System.out.print(a.y + "-"); a.f((B)a); }
16   void g(B b) { System.out.print(b.x() + "-"); b.f(b); }
17 }
```

Welche der folgenden Aufrufe 1 – 5 können nicht kompiliert oder ausgeführt werden? Geben Sie jeweils an, ob es beim Kompilieren oder zur Laufzeit einen Fehler gibt. Welche Ausgabe liefern die Aufrufe, die kompiliert und ausgeführt werden können unter der Annahme, dass die Aufrufe, die nicht kompiliert und ausgeführt werden können, auskommentiert werden?

```
1  public class Poly {
2    public static void main (String[] args) {
3      A a1 = new A();
4      A a2 = new B();
5      B b = new B();
6
7      b.f(b);      // Aufruf 1
8      a1.g(b);     // Aufruf 2
9      a2.f(b);     // Aufruf 3
10     a2.g(a1);    // Aufruf 4
11     a2.f(a2);    // Aufruf 5
12   }
13 }
```

Aufgabe 10.2 (P) Heapsort

`Heapsort` ist eines der effizientesten Sortierverfahren überhaupt (bezogen auf die Worst-Case-Laufzeit). Zum Sortieren benutzt man dabei eine baumartige Datenstruktur, den *binären Heap*. Ein solcher Heap besteht aus *Knoten*, in denen die Elemente gespeichert sind. Jeder Knoten hat null bis zwei Nachfolger und - mit Ausnahme der *Wurzel* - genau einen Vorgänger. Die Elemente sind außerdem so angeordnet, dass sie nach folgendem Schema lückenlos in ein Array passen:

- Die Wurzel liegt am Index 0.
- Der Vorgänger eines Elements an Index i liegt am Index $\left\lfloor \frac{i-1}{2} \right\rfloor$.

Wir benutzen im Folgenden diese Array-Darstellung. Im Heap gilt die Invariante: Jeder Nachfolger eines Elements ist stets größer als oder gleich groß wie das Element selbst.

Ein minimales Element in einem (nicht leeren) Heap liegt also in der Wurzel. Die Idee von Heapsort ist nun, immer das minimale Element zu entfernen und die entstehende Lücke so zu schließen, dass danach die Invariante wieder gilt. Zudem muss der Heap zu Beginn erstellt werden. Der Einfachheit halber gehen wir von einer festen Anzahl an Elementen, alle vom Typ Integer, aus. Wir benötigen folgende Methoden:

1. `void down(int index, int stop)`:
   Lässt das Element am Index `index` *absinken*: Zuerst wird ermittelt, ob die Heap-Bedingung bzgl. den Nachfolgern erfüllt ist, d. h. ob das Element am Index `index` kleiner oder gleich allen direkten Nachfolgern ist. Falls nicht, wird das kleinste Element am Index `x` mit dem am Index `index` vertauscht und `down(x)` aufgerufen. Der Parameter `stop` gibt den letzten Index des betrachteten Bereichs im Array an: Durch das Entfernen von sortierten Elementen verkleinert sich dieser Bereich.
2. `void buildHeap()`: Baut den Heap mit Hilfe der Methode `down` auf. D. h. die Heap-Bedingung wird in einem anfangs ungeordneten Array hergestellt. Versuchen Sie, mit möglichst wenigen Aufrufen auszukommen.

Tipp: Nutzen Sie Hilfsmethoden für wiederkehrende Teilaufgaben.

Aufgabe 10.3 (P) Generische Prioritätswarteschlange

Wir implementieren eine Prioritätswarteschlange mit Hilfe eines binären Heaps. Unsere Prioritätswarteschlange hat die die folgenden Methoden (Der Typ `ComparableThing` wird weiter unten erklärt.):

1. `public int size()`:
   Gibt die Anzahl an Elementen in der Prioritätswarteschlange zurück.
2. `public void clear()`:
   Entfernt alle Elemente aus der Prioritätswarteschlange.
3. `public void add(ComparableThing newElement)`:
   Fügt das Element in die Prioritätswarteschlange ein.
4. `public ComparableThing poll()`:
   Gibt ein Element der Prioritätswarteschlange mit der höchsten Priorität zurück und entfernt es aus ihr. Ist die Prioritätswarteschlange leer, wird `null` zurückgegeben.

5. `public ComparableThing peek()`:
   Gibt ein Element mit der höchsten Priorität zurück, ohne es zu entfernen. Ist die Prioritätswarteschlange leer, wird `null` zurückgegeben.

Da wir sehr schlau sind und nicht für jeden Typen eine extra Implementierung schreiben wollen, erstellen wir eine generische Variante der Prioritätswarteschlange. Der Typ wird `ComparableThing` genannt und implementiert das Interface `Comparable`, um Prioritäten vergleichen zu können.

Wie Sie aus der Vorlesung wissen, leistet sich Java den Scherz, Arrays als kovariante Objekte zu behandeln, sodass alle möglichen Probleme im Zusammenhang mit generischer Programmierung entstehen. Wir behelfen uns hier mit einer auf *Raw Types* basierenden Lösung, die in der Angabe bereits gegeben ist. In der Vorlage `PriorityQueue.java` sind bereits enthalten:

1. Alle nötigen Attribute
2. Konstruktoren, die einen Heap mit einer bestimmten Anfangsgröße anlegen
3. Methoden zum Anpassen der Arraygröße

Noch zu implementieren:

1. `private void up(int index)` und `private void down(int index)`:
   Die Methode `down` lässt ein Element wie in der Heapsort-Aufgabe beschrieben in den Heap einsinken. Ein zweiter Parameter ist nicht mehr nötig, da wir das Attribut `count` haben. Das Pendant `up` lässt ein Element an die korrekte Stelle im Heap aufsteigen. Sie wird beim Hinzufügen (`add`) eines neuen Elements benötigt.
2. Alle möglichen Hilfsmethoden
3. Die oben genannten Prioritätswarteschlangenmethoden
4. Ein kleines Testprogramm

Denken Sie daran, die Anzahl gespeicherter Elemente bei Bedarf anzupassen, und rufen Sie die benötigten Methoden an den entsprechenden Stellen auf.

Aufgabe 10.4 (P) Bettelschipp

Wir wollen die folgende Variante des Spiels *Schiffe-Versenken* implementieren. Gespielt wird auf einem quadratischen Spielfeld der Seitenlänge 10, d. h. es gibt 100 einzelne Felder. Die Felder eines Spielfeldes werden über ihre Koordinaten nach rechts (0 bis 9) sowie nach oben (0 bis 9) identifiziert. Das Feld (0,0) liegt also links unten in der Ecke. Zu Beginn markiert der Computer auf dem Spielfeld *zufällig* Bereiche, die entweder aus horizontal benachbarten Feldern bestehen oder aus vertikal benachbarten Feldern. Diese bezeichnen wir als *Schiffe*. Zwei Schiffe dürfen sich nicht überschneiden oder aneinander angrenzen (auch nicht diagonal). Von den Längen 2, 3 und 4 soll jeweils ein Schiff platziert werden. Beachten Sie, dass bei einer zufälligen Platzierung keine zulässige (d. h. regelkonforme) Möglichkeit von vornherein ausgeschlossen werden darf.

Im Verlauf des Spiels rät der Spieler wiederholt *ein Feld*, auf dem er ein Schiff vermutet. Sind alle Felder eines Schiffs geraten („getroffen“), gilt das Schiff als *gesunken*. Das Spiel

endet, wenn alle Schiffe gesunken sind. Das Programm gibt nach jedem Raten das Spielfeld aus und markiert darauf, welche Felder einen Treffer ergaben, welche Felder bereits Teil eines gesunkenen Schiffs sind und welche Felder bisher ohne Treffer (Schuss ins Wasser) geraten wurden.

Zum Testen des Programms soll es während des Spielverlaufs alternativ zum Raten eine Möglichkeit geben, sich die Platzierung der Schiffe anzeigen zu lassen. Zum Beispiel kann diese Ausgabe bei Eingabe von ungültigen Koordinaten (außerhalb vom Spielfeld) anstelle eines Fehlerhinweises erfolgen.

Implementieren Sie das Spiel objektorientiert! Verwalten Sie das Spielfeld, die Spielfeldkoordinaten und die Schiffe jeweils in einer eigenen Klasse.

Hinweis: Nennen Sie Ihr Programm `Battleship.java`. Die Praktikums-Webseite bietet eine Vorlage, die Sie nutzen können und bereits eine Spielfeldausgabe und eine Methode zum Erzeugen von Zufallszahlen enthält.

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; erstellen Sie jedoch nicht mehr als ein Package pro Aufgabe. 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 10.5 (H) Polymorphsefpsdofhksdlfbhkl [3 Punkte]

Betrachten Sie die Statements 1 bis 8 im folgenden Programm.

- Welches Statement kompiliert nicht? Wieso kompiliert es nicht?
- Welches Statement kompiliert und wirft eine Exception zur Laufzeit? Wieso wird hier eine Exception geworfen?
- Was ist die Ausgabe eines jeweiligen Statements, wenn dieses kompiliert und keine Exception wirft?

Nehmen Sie jeweils für ein Statement `i` an, dass alle anderen Statements `j` mit $j \neq i$ auskommentiert sind, d. h. Statement `i` ist der **einzige** Aufruf im Code.

```java
class A {
    void m(A a) { System.out.println("A.A+"); }
    void m(B b) { System.out.println("A.B+"); }
    void m(C c) { System.out.println("A.C+"); }
}

class B extends A {
    void m(A a) { System.out.println("B.A+"); }
    void m(B b) { System.out.println("B.B+"); }
    void m(C c) { System.out.println("B.C+"); }
}

class C extends B { }

public class Polymorphie {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();

        a.m((A) b);  // Statement 1
        a.m((B) c);  // Statement 2
        a.m(c);      // Statement 3
        b.m(a);      // Statement 4
        b.m((A) b);  // Statement 5
        b.m((A) c);  // Statement 6
        b.m((B) a);  // Statement 7
        b.m(c);      // Statement 8
    }
}
```

Statement 1: _____

Statement 2: _____

Statement 3: _____

Statement 4: _____

Statement 5: _____

Statement 6: _____

Statement 7: _____

Statement 8: _____

Aufgabe 10.6 (H) Pinguin in Seenot [6 Punkte]

Ein kleines, noch unerfahrenes Pinguinkind hat sich im gefährlichen Eismeer verirrt. Die Pinguinmutter hatte nur einen kleinen Moment nicht aufgepasst und es aus den Augen verloren. Nun ist sie – verständlicherweise – in heller Aufregung und möchte ihr Kleines so schnell wie möglich erreichen, indem sie von Eisscholle zu Eisscholle schwimmt.[1] Das Pinguinkind hat inzwischen seinen Fehler eingesehen. Es wartet auf seiner Eisscholle und hofft, dass Mutti rechtzeitig ankommt, bevor es von den herannahenden Seeleoparden verspeist wird.

Zum Glück bewahren Pinguine auch in gefährlichen und stressigen Situationen einen kühlen Kopf. Bevor sie losschwimmt, berechnet die Pinguinmutter also den kürzesten Weg zur Eisscholle, auf die sich der putzige kleine Kerl verschwommen hat. Die Schwimmdistanzen benachbarter Eisschollen (inklusive Aufenthaltsdauer auf der Startscholle) sind bekannt. Der kürzeste Weg von `A` nach `E` im abgebildeten Beispiel ist `A`⇝`B`⇝`D`⇝`E` (Distanz 24).

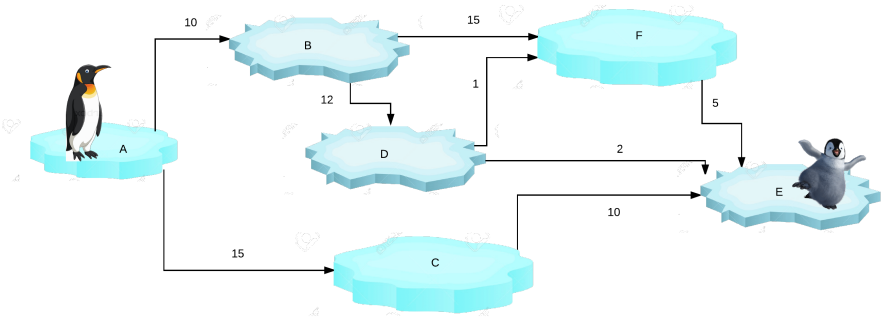

Abbildung 1: Pinguin in Seenot

Bei der Berechnung verwaltet die Pinguinmutter zwei Mengen von Eisschollen:

- `Bekannt`: Einen kürzesten Weg zu diesen Eisschollen hat sie bereits herausgefunden.
- `Vermutet`: Im Moment kennt sie bereits Wege zu diesen Eisschollen, ist sich aber nicht sicher, ob nicht *noch kürzere Wege* existieren.

Mütter sind sehr effizient. Entsprechend geht die Pinguinmutter systematisch vor:

[1] Die Zwischenhalte auf den Eisschollen sind notwendig, um das Pinguinkind durch Rufe und Blickkontakt zu beruhigen.

1. Initialisierung:

   (a) Setze die *Distanz* zur ersten Eisscholle e_0 auf 0 (Mama ist bereits dort).

   (b) Setze die Distanz für alle anderen Eisschollen auf *unendlich*. (Wir verwenden dazu `Integer.MAX_VALUE`.)

   (c) Setze den *Vorgänger* aller Eisschollen auf `null`.

   (d) Füge e_0 zu der Menge der `vermuteten` Eisschollen hinzu.

   Die *Distanz* gibt die Schwimmdistanz von e_0 zu dieser Eisscholle an. Der *Vorgänger* einer Eisscholle `e` wird später eine andere Eisscholle `e'` sein, sodass ein kürzester Weg von e_0 nach `e` die Schwimmung `e'`$\rightsquigarrow$`e` beinhaltet.

   Berechnung:[2]

2. Solange die Menge der `vermuteten` Eisschollen nicht leer ist, tue Folgendes:

   (a) Wähle aus dieser Menge eine Eisscholle `e` mit der niedrigsten Distanz aus. Diese Scholle ist nun `bekannt`.

   (b) Kalkuliere für jede Nachbarscholle `n` von `e` die Distanz d_n als Summe aus der Distanz bis `e` plus der Distanz der Schwimmung `e`$\rightsquigarrow$`n`.
   Falls d_n kürzer ist als die bisherige Distanz für `n`, setze den Vorgänger auf `e` und die Distanz bis zu dieser Eisscholle auf d_n.

   (c) Nimm alle Nachbarschollen, für die die kürzeste Distanz noch nicht bekannt ist, in die Menge der `vermuteten` Eisschollen auf.

   Ermittlung des kürzesten Weges:

3. Gehe von der Ziel-Eisscholle rückwärts bis zur Start-Eisscholle entlang den Vorgänger-Einträgen. Dies ist der kürzeste Weg in umgekehrter Reihenfolge.

Um die einzelnen Schritte zu verdeutlichen, führen wir den Algorithmus einmal für unser Beispielszenario oben durch. Die Pinguinmutter beginnt auf Eisscholle `A` und will möglichst schnell zur Eisscholle `E`.

Tabelle 1: Vorgänger und Distanzen im Verlauf der Berechnung

Schritt	Vermutet	Bekannt	`e`	A	B	C	D	E	F
Init.	A			0	X (∞)	X (∞)	X (∞)	X (∞)	X (∞)
1	A		A	0	A (10)	A (15)	X (∞)	X (∞)	X (∞)
2	B, C	A	B	0	A (10)	A (15)	B (22)	X (∞)	B (25)
3	C, F, D	A, B	C	0	A (10)	A (15)	B (22)	C (25)	B (25)
4	D, E, F	A, B, C	D	0	A (10)	A (15)	B (22)	D (24)	D (23)
5	E, F	A, B, C, D	F	0	A (10)	A (15)	B (22)	D (24)	D (23)
6	E	A, B, C, D, F	E	0	A (10)	A (15)	B (22)	D (24)	D (23)
Ergebnis		Alle		0	A (10)	A (15)	B (22)	D (24)	D (23)

Die mitgelieferte Vorlage enthält schon die benötigten Klassen. Ihnen obliegt es, diese wie folgt zu einer mustergültigen Lösung zu erweitern:

[2] Nach einer Idee von Edsger Wybe Dijkstra. Grundidee: Man findet in jedem Durchlauf den bzw. einen nächstgrößeren kürzesten Weg vom Ausgangspunkt zu einem anderen Punkt.

1. Klasse `Eisscholle`:

```
public class Eisscholle {
   public static final int UNBEKANNT = 0;
   public static final int VERMUTET = 1;
   public static final int BEKANNT = 2;

   private int distance;
   private Eisscholle vorgaenger;
   private final String name;
   private int state = UNBEKANNT;
}
```

   Die Klasse hat die folgenden Methoden:

   (a) `public Eisscholle(String name)`: Der Konstruktor bekommt den Namen der Eisscholle und setzt die Attribute auf die Werte der Initialisierungsphase.

   (b) `Getter` und `Setter` für alle Attribute, soweit möglich. Der Wert für die Distanz muss stets ≥ 0 sein. Wird ein ungültiger Wert für die Distanz übergeben, so soll der Wert auf `Integer.MAX_VALUE` gesetzt werden.

   (c) Überschreiben Sie die Methode `public boolean equals(Object o)`, sodass die Gleichheit von `Eisschollen` durch die Gleichheit ihrer Namen gegeben ist.

   (d) Überschreiben Sie die Methode `public String toString()` auf sinnvolle Weise.

2. Die Klasse `Seeweg` stellt die Distanz zwischen zwei Eisschollen dar:

```
public class Seeweg {
   private int distance;
   private Eisscholle from;
   private Eisscholle to;
}
```

   Die Klasse hat die folgenden Methoden:

   (a) `public Seeweg(int distance, Eisscholle from, Eisscholle to)`: Der Konstruktor benötigt Werte für alle Attribute und setzt die Attribute entsprechend.

   (b) `Getter`- und `Setter`-Methoden für alle Attribute

3. `Seerettung`:

```
public class Seerettung {
   private static PriorityQueue<Eisscholle> nachbarschollen;
}
```

   `PriorityQueue<Eisscholle>` ist eine Prioritätswarteschlange, die benutzt werden soll, um eine Eisscholle mit der niedrigsten Distanz aus der Menge der `vermuteten` Eisschollen auszuwählen. Sie bietet folgende Methoden:

   (a) `int size()`: Gibt die Anzahl an Elementen in der Warteschlange zurück.

(b) `boolean add(Eisscholle e)`: Fügt das Element in die Warteschlange ein.

(c) `public Eisscholle poll()`: Gibt ein Element der Warteschlange mit der höchsten Priorität zurück und entfernt es aus ihr. Ist die Warteschlange leer, wird `null` zurückgegeben.

(d) Auf https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html finden Sie weitere Methoden.

Die Klasse `Seerettung` hat folgende Methode:

(a) `public static List<Eisscholle> findeWeg(Eisscholle[] eisschollen, List<Seeweg> seewege, int startIndex, int endIndex)`: Diese Methode soll den oben beschriebenen Algorithmus anwenden. Sie bekommt als Parameter ein Array von Eisschollen, eine Liste `List<Seeweg>` mit allen Verbindungen, die zwischen den Eisschollen existieren, sowie die Indizes der Startscholle und der Endscholle. Sie gibt eine `List<Eisscholle>` zurück, in der die `Eisschollen` des kürzesten Weges in **richtiger** Reihenfolge enthalten sind.

4. Die Klasse `EisschollenComparator` stellt die der Prioritätswarteschlange zugrundeliegende Eisschollenvergleichsfunktion bereit.

5. Die Klasse `SeerettungTest` beinhaltet das oben berechnete Beispiel als Testfall.

Aufgabe 10.7 (H) Compiler für *MiniJava mit Funktionen* [6 Punkte]

Auf Blatt 8 haben wir den Anfang eines Parsers entwickelt, der ein MiniJava-Programm auf Korrektheit prüft. Auf diesem Blatt soll der Parser derart weiterentwickelt werden, dass er zum Übersetzen von Programmen in *MiniJava mit Funktionen* nach Steck-Assembly verwendet werden kann. Die Methoden des Parsers sollen dazu so verändert werden, dass sie jeweils einen Knoten des Syntaxbaums zurückliefern. Es sollen hierzu die Klassen verwendet werden, die auf Blatt 9 definiert wurden. Sie dürfen Ihre Lösung entweder auf Ihren eigenen Lösungen der letzten Blätter oder auf den offiziellen Musterlösungen aufbauen.

Wir erweitern zunächst die Grammatik von MiniJava um Funktionen:

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

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

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

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

<type>    ::= int

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

<expr>    ::= <number>
            | <name>
            | <name> ( (ϵ | <expr>(, <expr>)*) )
            | (<expr>)
            | <unop> <expr>
            | <expr> <binop> <expr>

<unop>    ::= -

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

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

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

<bunop>   ::= !

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

Gehen Sie nun wie folgt vor:

1. Wandeln Sie alle Methoden des Parsers in Objektmethoden um. Fügen Sie Attribute für das Token-Array und den Index `from` dem Parser hinzu; schreiben Sie auch einen passenden Konstruktor. Modifizieren Sie die Methoden derart, dass sie mit dem Attribut `from` statt dem entsprechenden Parameter arbeiten. Die Methode `parseStatement` soll nun z.B. die Signatur `private void parseStatement()` haben.

2. Erweitern Sie den Parser um die Erzeugung der Knoten des Syntax-Baumes. Jede Methode soll dabei den von ihr erzeugen Knoten zurückliefern. Die Methode `parseStatement` soll nun z.B. die Signatur `private Statement parseStatement()` haben. Schlägt das Parsen fehl, soll jeweils `null` zurückgeliefert werden.

3. Erweitern Sie den Parser, sodass er statt *MiniJava* Programme der obigen Grammatik von *MiniJava mit Funktionen* akzeptiert.

4. Implementieren Sie die Methode `public Program parse()`, die das gesamte Programm parst und zurückliefert. Kann das Programm nicht geparst werden, soll die Methode `null` zurückliefern.

5. Implementieren Sie die Klasse `Compiler`, die eine statische Methode `public static int[] compile(String code)` enthält, welche ein gegebenenes *MiniJava-mit-Funktionen*-Programm in Steck-Assembly übersetzt. Schlägt das Parsen oder die Erzeugung des Assemblys fehl, soll ein Fehler erzeugt werden. Erzeugen Sie einen Fehler durch das Statement `throw new RuntimeException("Fehlertext");`, wobei Sie eine geeignete Fehlermeldung statt *Fehlertext* übergeben können.

Hinweis: MiniJava kennt keine Operatorprioritäten. Daher ist die Reihenfolge, in der Ausdrücke mit mehrere Operatoren auf der gleichen Ebene ausgewertet werden, nicht definiert (also Ihnen überlassen). So ist z.B. offen, ob der Ausdruck `3 + 1 / 2` zu 2 oder 3 evaluiert. Wenn Sie selbst Testprogramme für Ihren Parser schreiben, müssen Sie Subausdrücke daher klammern, z.B. also `3 + (1 / 2)` verwenden.

Hinweis: Nutzen Sie die Klasse `CompilerTest` zum Testen Ihrer Implementierung.

Aufgabe 10.8 (H) Schwanzrufoptimierung [5 Punkte]

In der Vorlesung haben wir gelernt, *Endrekursion* normaler Rekursion vorzuziehen; und zwar deswegen, weil der Compiler bei endrekursiven Funktionen die sog. *Tail Call Optimization* anwenden kann, die aus einem rekursiven Aufruf einen Sprung macht. Der alte Stack-Frame wird hier beibehalten, der Kontrollfluss geht lediglich zurück an den Anfang der Funktion (prinzipiell ist dies auch möglich, wenn eine andere Funktion aufgerufen wird, solange der Stack-Frame kompatibel ist). Das *Recyclen* des aktuellen Stack-Frames ist nur bei endrekursiven Aufrufen korrekt; es muss nach dem rekursiven Aufruf also direkt eine `return`-Anweisung erfolgen.

Die Vorteile von Tail Call Optimization lassen sich in Java nur schwer beobachten, da der Java-Compiler die Optimierung nicht anwendet[3]. Um den Effekt der Optimierung dennoch studieren zu können, werden die Steckmaschine um Tail Call Optimization erweitern.

Implementieren Sie hierzu die Methode `public static void optimize(int[] program)` der Klasse `TailCallOptimization`. Die Methode erhält ein Programm und optimiert es, indem sie endrekursive Aufrufe durch passende Sprünge ersetzt. Testen Sie Ihre Implementierung mittels der mitgelieferten Klasse `TailCallTest`. Passen Sie dazu die größe des Stacks der Steckmaschine auf 32 an, sodass die Berechnung der Fakultät von 12 ohne Optimierung zu einem Überlauf führt. Die endrekursive Version der Fakultätsfunktion sollte mit eingeschalteter Optimierung nun allerdings dennoch funktionieren.

[3]Nach offizieller Java-Philosophie ist eine Exception einem funktionierenden Programm stets vorzuziehen.

Hinweis: Es ist leider nötig, die Code-Generierung anzupassen. Stellen Sie insbesondere sicher, dass Sie den Anfang einer Funktion im Assembly erkennen können und bei einem Aufruf genug Platz haben, die jeweiligen Instruktionen einzufügen.

Zusatzüberlegungen: Wäre es einfacher bzw. besser gewesen, die Optimierung direkt in der Code-Generierung zu machen? Welches Problem tritt hierbei auf? Welche dritte Möglichkeit hätte es gegeben?