Aufgabe 8.1 (P) Variablen, Variablen, Variablen

Im Folgenden werden wir uns mit den unterschiedlichen Begrifflichkeiten von Variablen in Java auseinandersetzen.

```java
class Foo {
    int x;
    static int y;

    void f(int n) {
        int m;
    }
}
```

In der Klasse `Foo` finden wir 4 unterschiedliche Variablen vor. Die Variablen `x` und `y` werden unter dem Oberbegriff *Membervariable* (im Englischen oft auch *field variable*) geführt. Es wird aber noch zwischen diesen beiden Variablen unterschieden. Die Variable `x` kommt in jeder Instanz der Klasse `Foo` genau einmal vor, d. h. in jedem Objekt der Klasse `Foo`. Variablen mit dieser Eigenschaft werden in Java als *Objektvariablen* bezeichnet. Die Variable `y` mit dem Schlüsselwort `static` hingegen kommt genau einmal in dem gesamten Programm vor, egal, wie viele Instanzen/Objekte der Klasse `Foo` existieren. Daher werden Variablen mit dieser Eigenschaft als *Klassenvariablen* bezeichnet. Von der Variable `n` wiederum gibt es so viele Instanzen, wie es Aufrufe von `f` gibt. Die Variable ist auch nur innerhalb der Methode `f` gültig. Solche Variablen werden in Java als *Parameter* bzw. *Parametervariablen* bezeichnet. Zu guter Letzt haben wir noch die Variable `m`. Variablen, die innerhalb einer Methode definiert sind, werden als *lokale Variablen* bezeichnet. Von diesen gibt es - wie für Parametervariablen - so viele Instanzen, wie es Methodenaufrufe gibt.

Aufgabe 8.2 (P) Stundenplan

In dieser Aufgabe wollen wir einen Stundenplan, in dem Termine eingefügt und wieder gelöscht werden können, simulieren. Ein Termin beginnt an einem Wochentag zu einer bestimmten Uhrzeit (Stunde und Minute) und hat eine bestimmte Dauer (in Minuten). Ein Termin ist also nur an einen Wochentag, nicht an ein konkretes Datum gebunden. Der Stundenplan enthält Termine für eine (generelle) Woche (Montag bis Sonntag).

Schreiben Sie eine öffentliche Klasse `Date`, die die folgenden Objektvariablen und Methoden enthält.

Date
- weekday : int - starthour : int - startmin : int - duration : int - title : String
+ Date(weekday : int, starthour : int, startmin : int, duration : int, title : String) + getWeekday() : int + getStarthour() : int + getStartmin() : int + getDuration() : int + getTitle() : String + toString() : String

Der Konstruktor soll die Objektvariablen entsprechend der Parameter initialisieren. Überlegen Sie sich eine geeignete Abbildung von Wochentagen (Montag bis Sonntag) auf `Integer` und setzen diese um. Sie können davon ausgehen, dass nur gültige Parameter im Konstruktor übergeben werden. Das bedeutet jedoch nicht, dass ein Termin nicht länger als Sonntag Mitternacht oder länger als eine Woche gehen kann. Die `get`'er-Methoden sollen die korrespondierenden Objektvariablen zurückliefern. Die `toString`-Methode soll eine `String`-Repräsentation des Objekts, die alle Objektvariablen enthält, zurückliefern. In dieser `String`-Repräsentation sollen die Wochentage im Wortlaut angegeben werden und Startzeit und Dauer mit der entsprechenden Einheit (Tage, Stunden, Minuten) angegeben werden. Die genaue Formatierung kann frei gewählt werden.

Schreiben Sie eine öffentliche Klasse `Timetable`, die die folgenden Objektvariablen und Methoden enthält.

Timetable
- dates : DateList
+ Timetable() + addDate(newDate : Date) : boolean + deleteDate(date : Date) : boolean + toString() : String

Schreiben Sie dafür zunächst eine private *innere Klasse* `DateList` mit den folgenden Objektvariablen und Methoden, die eine Liste `Date`-Elementen repräsentiert.

DateList
- info : Date - next : DateList
+ DateList(info : Date) + toString() : String

Der Konstruktor soll die Membervariable `info` mit dem Paramter `info` initialisieren. Die Methode `toString()` soll eine Repräsentation der `Date`-Elemente in der Reihenfolge, wie sie in der Liste enthalten sind, zurückliefern. Sie dürfen in der inneren Klasse `DateList` weitere Methoden hinzufügen, jedoch keine weiteren Membervariablen.

Die Methode `boolean addDate(Date newDate)` der Klasse `Timetable` fügt `newDate` zur Liste `dates` (Objektvariable) hinzu, falls es keine zeitlichen Übereinschneidungen mit bereits vorhanden Terminen in `dates` gibt. Die Methode liefert `true` zurück, wenn `newDate` hinzugefügt werden konnte; andernfalls wird `dates` nicht verändert und `false` zurückgeliefert.

Die Methode `boolean deleteDate(Date date)` der Klasse `Timetable` überprüft, ob es ein `Date`-Element in der Liste `dates` (Objektvariable) gibt, bei dem die Werte der Objektvariablen mit denen des Parameters `date` übereinstimmen. Falls ein solches `Date`-Element gefunden wird, wird dieses aus `dates` entfernt und `true` zurückgeliefert; andernfalls wird `dates` nicht verändert und `false` zurückgeliefert.

Die Methode `toString()` der Klasse `Timetable` liefert eine `String`-Repräsentation der Objektvariablen `dates`, die die Repräsentation der `Date`-Elemente in chronologischer Reihenfolge enthält. Die genaue Formatierung kann frei gewählt werden.

Überlegen Sie in Gruppen oder im gesamten Tutorium, wie mit Terminen, die länger als Sonntag Mitternacht gehen, umgegangen werden kann, und setzen Sie mindestens eine dieser Lösungen um.

Aufgabe 8.3 (P) Unveränderbare Zeichenkettenmengen

Ziel dieser Aufgabe ist es, eine Klasse `ImmutableSet` zur Repräsentation von *unveränderbaren* Mengen von Strings zu entwickeln. Implementieren Sie diese Klasse mit Hilfe eines Arrays, in dem jedes Element der repräsentierten Menge **genau einmal** vorkommt. Die Klasse `ImmutableSet` soll folgende Methoden bereitstellen:

- Einen Konstruktor zur Erzeugung einer **leeren** Menge.
- Eine Methode `boolean isElement(String s)`, die überprüft, ob der durch `s` repräsentierte String in der Menge enthalten ist.
- Eine Methode `boolean superset(ImmutableSet subset)`, die überprüft, ob alle Elemente der Menge `subset` in der Menge enthalten sind, die die Methode bereitstellt.
- Eine Methode `boolean isEqual(ImmutableSet other)`, die überprüft, ob die Menge `other` genau die gleichen Elemente enthält wie die Menge, die die Methode bereitstellt.
- Eine Methode `ImmutableSet add(String s)`, die eine um den String `s` erweiterte Menge zurückgibt. Das heißt, der Aufruf `m1.add(s)` gibt eine Menge `m2` zurück, die `s` und alle Elemente von `m1` enthält.
  - Falls der String `s` bereits in der alten Menge enthalten ist, können Sie die alte Menge zurückgeben.
  - Für den anderen Fall erstellen Sie einen zweiten Konstruktor, diesmal einen privaten Konstruktor, der einen String `s` und eine Menge `old` bekommt und daraus eine neue Menge erstellt, die alle Elemente enthält.
- Eine Methode `String toString()`, die eine String-Darstellung der Menge erzeugt und zurückliefert.

Hinweise: Verwenden Sie bei der Implementierung nach Möglichkeit von Ihnen bereits implementierte Methoden aus anderen Teilaufgaben. Testen Sie Ihre Klasse mit einer sinnvollen `main`-Methode. Diese sollte alle Methoden testen und auch mögliche Interaktionen zwischen den einzelnen Methoden berücksichtigen.

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). Geben Sie Ihren Code **nicht** als Archiv (z.B. als `.zip`-Datei) ab. Nutzen Sie **keine** Ordner in Moodle. Nutzen Sie **keine** Packages. 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 8.4 (H) Videosammlung [5 Punkte]

Ziel dieser Aufgabe ist es, eine Videosammlung zu simulieren.

1. Implementieren Sie die Klasse `Video` mit den privaten Attributen `String titel`, `int id` und `String[] genres`, sowie den Getter-Methoden `String getTitel()`, `int getId()`, `String[] getGenres()` und dem Konstruktor `Video(String titel)`. Die ID des ersten Filmes soll 0 sein, die folgenden werden aufsteigend durchnummeriert. **Dies funktioniert ähnlich wie bei der Klasse `Count` in der Vorlesung.**

2. Erweitern Sie die Klasse `Video` um die öffentliche Methoden `int addGenre(String genre)`. Diese fügt den Genres ein weiteres hinzu, wobei ein Video maximal nur fünf Genres angehören kann. Die Methode soll zudem verhindern, dass das selbe Genre mehrfach hinzugefügt wird. Die Methode gibt bei erfolgreichem Hinzufügen des Genres die gesamte Anzahl der Genres für dieses Video zurück und `-1`, falls das Genre nicht hinzugefügt werden konnte.

3. Implementieren Sie die Klasse `Videosammlung`, die eine Videosammlung mit maximal n Videos repräsentiert. Dabei soll die Größe n der Videosammlung im Konstruktor mit übergeben werden.

4. Fügen Sie der Klasse `Videosammlung` eine Methode `int addVideo(Video v)` hinzu, bei der das Video `v` die erste freie Stelle der Videosammlung belegt. Die Methode gibt dabei die Position der Stelle zurück. Sollte die Videosammlung voll sein, so gibt die Methode `-1` zurück.

5. Erweitern Sie die Klasse `Videosammlung` um die Methoden `Video verkaufen(int index)` und `Video verkaufen(String titel)`. Beide Methoden löschen jeweils ein Video aus der Sammlung, wobei `index` dem Index entspricht und `titel` dem Titel des Videos, das gelöscht werden soll. Gibt es einen Titel mehrfach in der Sammlung, so wird nur die erste Instanz gelöscht. Die Methode gibt das verkaufte Video zurück, bzw. `null`, falls kein Video gelöscht wurde.

6. Fügen Sie eine Objektvariable `private int verbleibende` ein, in der die verbleibenden freien Plätze gespeichert werden. Der Wert dieser Variablen soll bei jedem Aufruf der `addVideo` und `verkaufen` Methoden sinnvoll modifiziert werden.

7. Fügen Sie anschließend der Klasse `Videosammlung` eine Objektmethode `public int getVerbleibende()` hinzu, welche die Anzahl der verbleibenden Plätze zurückgibt, ohne dabei die einzelnen Plätze zu überprüfen.

8. Erweitern Sie die Klasse `Videosammlung` um eine Methode `String[] videosInGenre (String genre)`. Die Methode gibt ein String-Array mit dem Titel aller Videos, die dem übergebenen `String genre` entsprechen, zurück.

Hinweis: In der Klasse `Videotest` finden Sie einige Tests, um Ihre Implementierung zu überprüfen.

Aufgabe 8.5 (H) Worstorage [7 Punkte]

In dieser Aufgabe geht es darum, die (un)bekannte Datenstruktur Worstorage zu implementieren, die (un)bekanntlich vergleichbare Objekte (in unserem Fall Pinguine, die nach ihrer Knuffigkeit geordnet sind) besonders ineffizient speichern kann.

Ein Worstorage besteht intern (private Objekt-Attribute) aus einem Array `ps` der Größe $n = 2^k - 1$ ($k \geq 1$) mit den Referenzen auf die gespeicherten Objekte, welches in k *Ebenen* unterteilt ist, sowie einem `int`-Array `count` der Größe k, in dem für jede Ebene die *Anzahl an Einträgen* festgehalten wird.

Wir nummerieren die Positionen im Array mit 1 (Arrayindex 0) bis n (Index $n-1$). Der *direkte* linke Nachfolger eines Elements an Position p, falls vorhanden, ist das Element an Position $2p$, der *direkte* rechte Nachfolger, falls vorhanden, ist das Element an Position $2p+1$. Nachfolger allgemein sind dann wie folgt induktiv definiert. Linke Nachfolger sind der direkte linke Nachfolger sowie alle seine Nachfolger. Analog dazu sind die rechten Nachfolger alle Nachfolger des rechten Nachfolgers sowie dieser selbst. Die erste Ebene besteht nur aus der Position 1. Die Anzahl der dort gespeicherten Elemente (0 oder 1) steht an Index 0 im Array `count`. Alle weiteren Ebenen e enthalten genau alle direkten Nachfolger von Elementen der Ebene $e-1$.

Um schnell auf die gespeicherten Objekte zugreifen zu können, gilt die folgende Invariante: Alle linken Nachfolger sind stets kleinere, alle rechten Nachfolger größere oder gleich große Elemente. Zudem darf im Speicher keine *Lücke* bestehen: Hat ein Element keinen direkten linken (bzw. rechten) Nachfolger, darf es überhaupt keine linken (bzw. rechten) Nachfolger geben. Nutzen Sie die `compareTo`-[1]Funktion der Klasse `Penguin` für den Vergleich zweier Pinguine. Diese ist bereits im Codegerüst `Worstorage.java` implementiert.

Liegen keine Elemente im Array (alle Einträge `null`), ist obige Invariante bereits erfüllt. Die zu implementierenden Methoden müssen dafür sorgen, dass sie immer erhalten bleibt. Außerdem muss `count` auch nach der Ausführung der Methoden weiterhin die korrekten Werte enthalten.

Im Einzelnen sollen folgende Objekt-Methoden in dem Codegerüst `Worstorage.java` in der Klasse `Worstorage` implementiert werden:

1. `public void add(Penguin penguin)`: Fügt einen Pinguin ein, falls dieser noch nicht enthalten ist. Reicht der Speicher nicht aus, soll die Anzahl Einträge auf die doppelte Anzahl plus 1 steigen. Die Anzahl Ebenen wird entsprechend um 1 vergrößert, d. h. `count` referenziert nach der Operation ein um 1 größeres Array als zuvor.
2. `public boolean find(Penguin penguin)`: Gibt `true` genau dann zurück, wenn das Argument enthalten ist. Zum Auffinden des Tierchens soll die Invariante genutzt werden. Es soll also nicht im gesamten Array gesucht werden.
3. `public void remove(Penguin penguin)`: Entfernt den Argumentpinguin, falls enthalten. Schließen Sie ggf. entstehende Lücken, so dass wieder ein konsistenter Zustand erreicht wird. Es dürfen dazu nur **Nachfolger** des Elements, das gelöscht wurde, ihre Speicherposition ändern. Alle übrigen Elemente bleiben an ihrem Platz. Denken Sie auch daran, `count` zu aktualisieren. Gibt es in der untersten Ebene

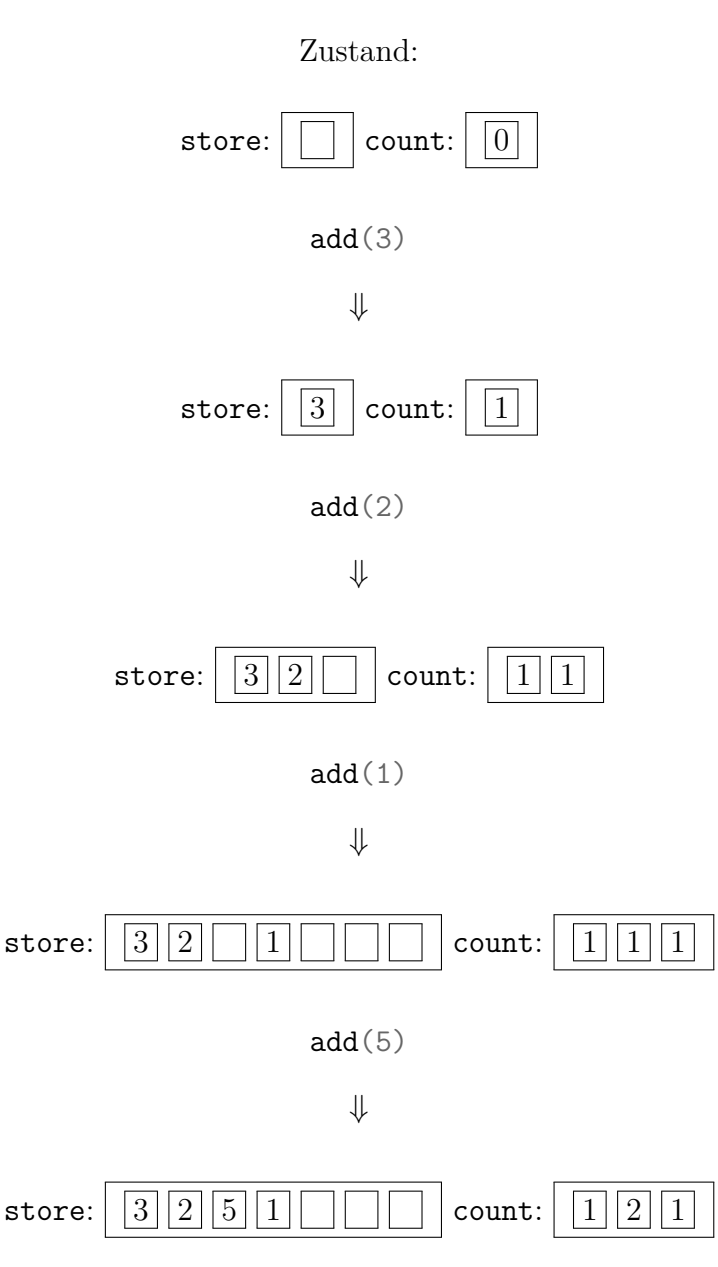
[1] `https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-`

keinen Eintrag mehr (`count` hat am letzten Index den Wert 0), so soll die Anzahl Speicherplätze von `x` auf `(x-1)/2` verkleinert und die Anzahl Ebenen (Länge von `count`) um 1 verringert werden. Es soll aber immer mindestens eine Ebene und einen Speicherplatz geben.

4. `public String toString()`: Gibt das interne Pinguin-Array **in folgendem Format** zurück: Alle Einträge werden nacheinander, beginnend mit Index 0, mit Kommas getrennt ausgegeben. `null`-Einträge werden dabei durch die leere Zeichenkette dargestellt, Pinguine durch ihre Knuffigkeit, also den entsprechenden `int`-Wert. Nichts sonst darf zurückgegeben werden. Beispiel: `new Penguin[]{p1, null, p2, null, null, p3, null}` wird genau dann als `"-1,,5,,,3,"` zurückgegeben, wenn die Pinguine `p1`, `p2`, `p3` die Knuffigkeiten -1, 5 und 3 haben.

Der Konstruktor soll zu Beginn die Größe derart festlegen, dass es genau eine Ebene und dementsprechend genau einen Speicherplatz für Pinguine gibt.

Beispiel (mit Zahlen statt Pinguinen):



⇓

store: | 2 | 1 | 5 | count: | 1 | 2 |

`add(4)`

⇓

store: | 2 | 1 | 5 | | | 4 | | count: | 1 | 2 | 1 |

`add(8)`

⇓

store: | 2 | 1 | 5 | | | 4 | 8 | count: | 1 | 2 | 2 |

`add(3)`

⇓

store: | 2 | 1 | 5 | | | 4 | 8 | | | | | 3 | | | |

count: | 1 | 2 | 2 | 1 |

`add(4)` (bereits enthalten)

⇓

store: | 2 | 1 | 5 | | | 4 | 8 | | | | | 3 | | | |

count: | 1 | 2 | 2 | 1 |

`remove(5)`

⇓

store: | 2 | 1 | 4 | | | 3 | 8 | count: | 1 | 2 | 2 |

`add(7)`

⇓

store: | 2 | 1 | 4 | | | 3 | 8 | | | | | | | 7 | |

count: | 1 | 2 | 2 | 1 |

Aufgabe 8.6 (H) Parsy McParseface [8 Punkte]

Unter *parsen* versteht man die Umwandlung von Programmtext in ein Format, welches man anschließend in weiteren Schritten derart verarbeiten kann, dass am Ende ein Maschinenprogramm herauskommt (z.B. für unsere Steckmaschine vom letzten Blatt). Im Allgemeinen ist mit der Implementierung von Parsern eine reichhaltige Theorie verbunden[2] – wir werden uns in dieser Aufgabe jedoch mit einem möglichst *einfachen* sog. *Recursive Descent Parser* beschäftigen, der ausschließlich MiniJava-Programme vararbeiten kann. Unser Parser soll außerdem hier zunächst nur für ein gegebenes MiniJava-Programm ausgeben, ob dieses syntaktisch korrekt ist.

Es sei zunächst folgendes Gerüst für Ihre Implementierung gegeben:

```java
import java.util.Scanner;

public class MiniJavaParser {
  public static String readProgramConsole() {
    @SuppressWarnings("resource")
    Scanner sin = new Scanner(System.in);
    StringBuilder builder = new StringBuilder();
    while (true) {
      String nextLine = sin.nextLine();
      if (nextLine.equals("")) {
        nextLine = sin.nextLine();
        if (nextLine.equals(""))
          break;
      }
      if (nextLine.startsWith("//"))
        continue;
      builder.append(nextLine);
      builder.append('\n');
    }
    return builder.toString();
  }

  public static String[] lex(String program) {
    return null; // Todo
  }

  public static int parseNumber(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseName(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseType(String[] program, int from) {
```

[2] Siehe z.B. `http://www2.in.tum.de/hp/Main?nid=218`

```
    return -1; // Todo
  }

  public static int parseDecl(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseUnop(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseBinop(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseComp(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseExpression(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseBbinop(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseBunop(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseCondition(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseStatement(String[] program, int from) {
    return -1; // Todo
  }

  public static int parseProgram(String[] program) {
    return -1; // Todo
  }

  public static void main(String[] args) {
    // Todo
  }

}
```

Im Gerüst findet sich für jedes Nichtterminal der MiniJava-Grammatik eine Methode.

Jede dieser Methoden erwartet ein Array bestehend aus *Token* als Parameter sowie einen Startindex. Ein Token ist eine logisch zusammenhängende Einheit des Eingabeprogramms, z.B. das Schlüsselwort `read`. Die Methode soll ihr Nichtterminal parsen und den Index des nächsten Tokens hinter dem Nichtterminal zurückliefern. Schlägt das Parsen fehl, soll eine negative Zahl zurückgegeben werden.

Bevor ein Programm geparst werden kann, muss es zunächst in *Token* zerlegt werden. In unserem Fall ist ein Token

- eine Klammer (sowohl rund als auch geschweift) bzw. ein Komma,
- ein Operator,
- eine Zahl oder
- ein Name (hierzu gehören auch *Schlüsselwörter* wie `read`, `write` oder `int`).

Leerzeichen und Zeilenumbrüche werden während des Lexens verworfen. Das MiniJava-Programm

```
1 int sum, n, i;
2 n = read();
3 while (n < 0) {
4   n = read();
5 }
```

wird daher in das Token-Array `[int, sum, ,, n, ,, i, ;, n, =, read, (, ), ;, while, (, n, <, 0, ), {, n, =, read, (, ), ;, }]` zerlegt. Gehen Sie in Ihrer Implementierung wie folgt vor:

1. Implementieren Sie die Methode `String lex(String program)`, die ein Programm in ein Array von Token umwandelt.
2. Implementieren Sie die übrigen Methoden des Parsers. Es kann nützlich sein, weitere Hilfsmethoden hinzuzufügen.
3. Implementieren Sie ein Hauptprogramm, welches den Benutzer um Eingabe eines Programms bittet und anschließend ausgibt, ob dieses Programm korrekt geparst werden konnte. Nutzen Sie die bereits vom letzten Blatt bekannte Methode `String readProgramConsole()`.

Hinweis: Sie dürfen Ihr Token-Array *padden*, also leere Strings an das Ende anfügen, sofern Ihnen das hilft, weniger Fälle innerhalb des Parsers unterscheiden zu müssen. Insbesondere kann so vermieden werden, dass häufig auf das Ende der Eingabe geprüft werden muss.

Hinweis: Wenn Sie die in der Einleitung erwähnte Umwandlung in ein Format vermissen, haben Sie bitte Geduld bis zu einem der folgenden Übungsblätter.

Hinweis: Testen Sie Ihre Implementierung. Nutzen Sie z.B. das folgende Programm:

```
int sum, n, i;
n = read();
while (n < 0) {
  n = read();
}

sum = 0;
i = 0;
while (i < n) {
  {
    if (i % 3 == 0 || i % 7 == 0) {
      sum = sum + i;
      if (i % 3 == 0 || i % 7 == 0) {
        sum = sum + i;
      } else
        sum = 99;
    }
    i = i + 1;
  }
}

write(sum);
```

Dieses Programm ist ein korrektes MiniJava-Programm. Ändern Sie das Programm, um zu testen, ob ihr Parser falsche Programme zurückweist.