Programmierparadigmen

Sommerersemester 2017
Martin Wittiger, Felix Krause, Timm Felden

5. Übung

Abgabe bis 28. Juni um 4:44
Beachten Sie die Abgabehinweise auf der Vorlesungswebseite!

Aufgabe 5.1 ..... 5 Punkte

Dateiname: `Schachbrett.java` ★★★

Gegeben sei ein Schach-ähnliches Spiel, das auf einem 8×8 Brett gespielt wird. Ein Spieler führt die weißen Figuren, der andere die schwarzen. Der Spieler, der am Zug ist, muss eine der Figuren seiner Farbe gemäß ihrem Bewegungsmuster (siehe unten) bewegen, also auf ein anderes Feld als das, worauf sie bisher stand, setzen. Bewegt er sie dabei auf ein Feld, auf dem eine Figur des anderen Spielers steht, wird diese Figur *geschlagen*, also vom Brett genommen. Eigene Figuren können nicht geschlagen werden. Keine Figur kann sich über den Brettrand hinaus bewegen.

Kann eine gegnerische Figur gemäß ihrem Bewegungsmuster eine eigene Figur schlagen, so gilt jene Figur als *bedroht* – selbst dann, wenn die gegnerische Figur das Schlagen nicht ausführen könnte, weil dies eine ungültige Stellung erzeugen würde (s.u.). Ist die bedrohte Figur ein König, so steht dieser König im *Schach*.

Jeder Spieler hat genau einen *König* auf dem Brett. Es dürfen nur solche Züge getätigt werden, die eine gültige Stellung erzeugen. Eine gültige Stellung ist genau dann erreicht, wenn der König des Spielers, der gerade *nicht* am Zug ist, nicht im Schach steht. Beispielsweise darf man den eigenen König nicht auf ein Feld ziehen, das von einer gegnerischen Figur bedroht ist; ebensowenig darf man die Bedrohung des eigenen Königs ignorieren.

Die Richtung *Vorwärts* ist definiert als die vertikale Richtung hin zur gegenüberliegenden Grundreihe – für weiß also Richtung achte Reihe, für schwarz Richtung erste Reihe.

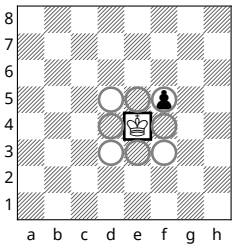
Auf der Webseite finden Sie ein Datenmodell für ein Spielbrett in der Datei `Schachbrett.java`. Implementieren Sie die Methode `moeglicheZuege`, die abhängig davon, ob weiß oder schwarz am Zug ist, berechnen soll, welche gültigen Züge der jeweilige Spieler machen darf. Sie dürfen die Klassen der einzelnen Schachfiguren anpassen.

Die beiliegende main-Methode in `SchachChecker.java` muss mit dem angepassten Code kompilieren und funktionieren. Verändern Sie also nicht die Konstruktordeklarationen der Klassen oder die Klassenhierarchie. `SchachChecker` verlangt als Eingabe eine Datei, in der ein Schachbrett codiert ist, sowie die Information, welcher Spieler am Zug ist (`w` oder `s`). Als Ausgabe liefert er die durch `moeglicheZuege` berechneten Züge. Auf der Webseite werden Testfälle bereitgestellt, die mit `SchachChecker.java` eingelesen werden können und die korrekte Ausgabe sowohl für weiß wie auch für

schwarz am Zug enthalten. In manchen Stellungen kann nur eine der beiden Parteien ziehen, da die Stellung ansonsten ungültig wäre. Gehen Sie in Ihrem Algorithmus davon aus, dass die gegebene Stellung gültig ist. Wie immer sei Ihnen nahegelegt, zusätzliche Testfälle zu erstellen und Ihren Code damit zu testen.

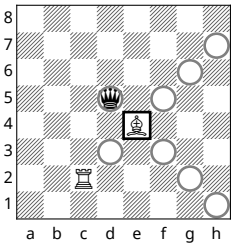
Alle Testdateien haben die Dateiendung `.chess`. Die Testfälle benutzen die Anfangsbuchstaben der Figurennamen, um die Figuren zu identifizieren. Ein Großbuchstabe bedeutet *weiße Figur*, ein Kleinbuchstabe *schwarze Figur*. Als Werkzeug, um Testeingaben besser studieren (und neue generieren) zu können, sei Ihnen der Online-Editor von `lichess.org` empfohlen: `https://lichess.org/editor`

König ♔



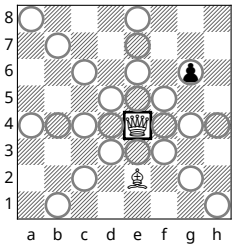
Der König darf einen Schritt in jede Richtung (horizontal, vertikal, diagonal) machen und dabei auch Figuren schlagen.

Läufer ♗



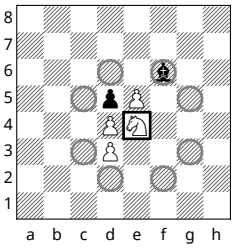
Der Läufer darf beliebig viele Schritte diagonal machen und dabei auch Figuren schlagen, aber keine Figuren überspringen.

Dame ♕



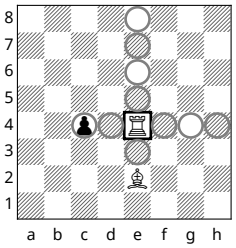
Die Dame darf beliebig viele Schritte in *eine* beliebige Richtung (horizontal, vertikal, diagonal) machen und dabei auch Figuren schlagen, aber keine Figuren überspringen.

Springer ♘



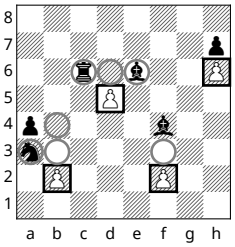
Der Springer macht immer genau zwei Schritte in eine Richtung (horizontal oder vertikal) und dann einen Schritt orthogonal dazu. Dabei kann er Figuren überspringen und die Figur auf dem Zielfeld schlagen.

Turm ♖



Der Turm darf beliebig viele Schritte horizontal oder vertikal machen und dabei auch Figuren schlagen, aber keine Figuren überspringen.

Bauer ♙



Der Bauer kann nur einen Schritt vorwärts ziehen und zusätzlich einen zweiten Schritt, sofern er anfangs in der zweiten (weiß) bzw. siebten (schwarz) Reihe steht. Vorwärts kann er weder Figuren überspringen noch schlagen. Er schlägt diagonal nach vorne.

Abbildung 1: Bewegungsmuster der Figuren. Die Figur am Zug ist mit einem schwarzen Quadrat markiert, die möglichen Zielfelder mit grauen Kreisen.

Hinweis für Schachbegeisterte: Die komplexeren Züge wie die Rochade, das Schlagen *en passant* und die Bauernumwandlung sind nicht Teil dieser Aufgabe.

Aufgabe 5.2 ..... 4 Punkte

Dateiname: `Subtypes.java` ★★★

In den Materialien finden Sie eine Java-Klasse `Subtypes.java`. Diese definiert acht innere Typen `A` bis `I`. Ändern Sie diese Typen ab, so dass die nachfolgend gelisteten Bedingungen erfüllt werden. Wäre beispielsweise die Bedingung *`W`, `Y` und `Z` sind Klassen, wobei `Y` ein Subtyp von `X` ist und `Z` ein Subtyp von `Y`* durch folgenden Code umgesetzt:

```
1 public static interface X {}
2 public static class Y implements X {}
3 public static class Z extends Y {}
4 public static class W {}
```

Es müssen folgende Bedingungen erfüllt sein:

1. Einer Variablen von Typ `A` kann ein Wert vom Typ `G` zugewiesen werden.
2. `C` ist eine abstrakte Klasse, die zwei Interfaces direkt implementiert und zwei Subtypen hat.
3. Alle Werte außer solcher vom Typ `E` können einer Variablen vom Typ `I` zugewiesen werden.
4. `G` ist ein Subtyp von `B`, `C` ist keiner.
5. Variablen aller Klassentypen sind zuweisbar zu einer Variablen vom Typ `E`.
6. Einer Variablen vom Typ `H` kann kein Wert ungleich **`null`** zugewiesen werden. Variablen jeden anderen Typs kann ein Wert ungleich **`null`** zugewiesen werden.
7. `A` implementiert zwei Interfaces und ist instantiierbar.
8. `F` ist ein Subtyp von `D` und hat keine Subtypen.
9. `E` und `I` haben keine Vererbungsbeziehung zueinander.

Hinweise Gehen Sie davon aus, dass keine Typen außer `A` bis `I` existieren. Sie dürfen den Modifier **`interface`** der gegebenen Typen ändern, die Modifier **`public static`** jedoch nicht (sonst kann Ihre Abgabe nicht korrigiert werden). Fügen Sie keine Typen hinzu und ändern Sie die Namen der gegebenen Typen nicht.

Aufgabe 5.3 ..... 2 Punkte

Dateiname: `dope.pdf` ★☆☆

Typen, die eine zur Laufzeit festgelegte Größe haben, wie beispielsweise Strings, benötigen zusätzlichen Speicherplatz, um einen sogenannten *Dope* zu speichern. In dieser Aufgabe schauen wir uns an, wie der Ada-Compiler diesen Dope implementiert. Gegeben sei folgender Ada-Code:

```
1  procedure Dope is
2     type String_Access_1 is access String;
3
4     type String_Access_2 is access String;
5     for String_Access_2'Size use Standard'Address_Size;
6
7     use Ada.Text_IO;
8
9     SA_1_Size : constant Integer := String_Access_1'Size / System.Storage_Unit;
10    SA_2_Size : constant Integer := String_Access_2'Size / System.Storage_Unit;
11
12    String_1 : String_Access_1 := new String'(3 => 'a', 4 => 'b', 5 => 'c');
13    String_2 : String_Access_2 := new String'(10 => 'c', 11 => 'd');
14 begin
15    Put_Line ("Positive'Size:" & Positive'Size'Img);
16    Put_Line ("String_1 range: (" &
17      String_1'First'Img & " .." & String_1'Last'Img & ')');
18    Put_Line ("String_2 range: (" &
19      String_2'First'Img & " .." & String_2'Last'Img & ')');
20
21    Put_Line ("String_Access_1'Size:" & SA_1_Size'Img);
22    Put_Line ("String_Access_2'Size:" & SA_2_Size'Img);
23 end Dope;
```

Die Ausgabe dieses Codes auf einem 64-bit Betriebssystem wie z.b. `marvin` ist:

```
String_1 range: ( 3 .. 5)
String_2 range: ( 10 .. 11)
String_Access_1'Size: 16
String_Access_2'Size: 8
```

Es werden zwei unterschiedliche Pointer-Typen für den Typ `String` definiert. Für den zweiten Pointer-Typ definieren wir in Zeile 5 explizit, dass er genauso groß wie eine Speicheradresse des Betriebssystems sein soll (bei 64-bit also 8 Byte). Wir allokieren dann zwei Strings auf dem Heap und lassen uns einige Größen ausgeben. Beantworten Sie folgende Fragen:

1. Welche Informationen muss der Compiler zusätzlich zu dem Inhalt für die beiden String-Typen speichern, um in den ersten beiden Ausgabezeilen zu ermöglichen, die korrekten Indizes auszugeben, so wie sie bei der Werterstellung angegeben wurden?

2. Basierend auf den Ausgabezeilen 3 und 4, finden Sie eine schlüssige Erklärung, wo der Compiler in beiden Fällen diese Zusatzinformation speichert. Bedenken Sie dabei, dass beide String-Pointer-Typen auf jeden Fall eine Adresse enthalten müssen, die den Startpunkt des Strings auf dem Heap definiert. Ihre Antwort muss nicht unbedingt die tatsächliche Implementierung beschreiben, solange sie eine gangbare Lösung darstellt.

Aufgabe 5.4 ..... 4 Punkte

Dateiname: `dispatching.adb` ★★☆

Im Folgenden sind einige Klassen in einer C++ artigen Sprache gegeben, in der das Schlüsselwort **`virtual`** und das Fehlen desselben dieselbe Bedeutung wie in C++ haben. Gehen Sie davon aus, dass die gegebenen Klassen gültige Deklarationen darstellen und eine Vererbungshierarchie gemäß den üblichen Regeln der Objektorientierung darstellen. Gehen Sie weiterhin davon aus, dass die gegebenen Aufrufe gültig sind und eine Klassenmethode gemäß den üblichen Konzepten der Objektorientierung aufrufen und beantworten Sie für jeden unten aufgeführten Aufruf folgende zwei Fragen:

1. Auf welche Methode *bindet* der angegebene Aufruf; das heißt, auf welche Methode löst der Compiler zur Compilezeit den angegebenen Methodennamen auf?
2. Welche Methode wird zur Laufzeit *aufgerufen*, wenn der gegebene Aufruf ausgeführt wird?

Referenzieren Sie eine Methode immer mit der Nummer der Zeile, in der sie deklariert wurde. Tragen Sie Ihre Antworten in die bereitgestellte Datei ein und geben Sie diese ab. Stellen Sie sicher, dass die geänderte Datei zusammen mit `print_dispatching.adb` kompiliert und Ihre Lösung auf der Kommandozeile ausgibt.

Klassendefinitionen:

```
class A {
  virtual void m1();
  void m1(int);
  virtual void m2();
  void m3(int);
};
class B : A {
  virtual void m1();
  virtual void m2(int);
  void m3();
  virtual void m4();
};
class C : B {
  void m1(int);
  virtual void m2();
};
class D {
  virtual void m1(int,int);
  virtual void m4(int);
  void m5();
};
class E : A, D {
  virtual void m1(int);
  virtual void m1(int,int);
};
class F : E {
  virtual void m1();
  void m5();
};
```

Aufrufe:

```
A* a = new C;
B* b = new B;
D* d = new E;
F* f = new F;
a.m1();
a.m1(1);
a.m2();
b.m1();
b.m1(1);
b.m2();
b.m3(1);
d.m1(1,2);
d.m4(42);
d.m5();
f.m1();
f.m1(1);
f.m1(1,2);
f.m2();
f.m4(3);
f.m5();
```

Aufgabe 5.5 ..... 3 Punkte

Dateiname: `kreuzung.hpp`

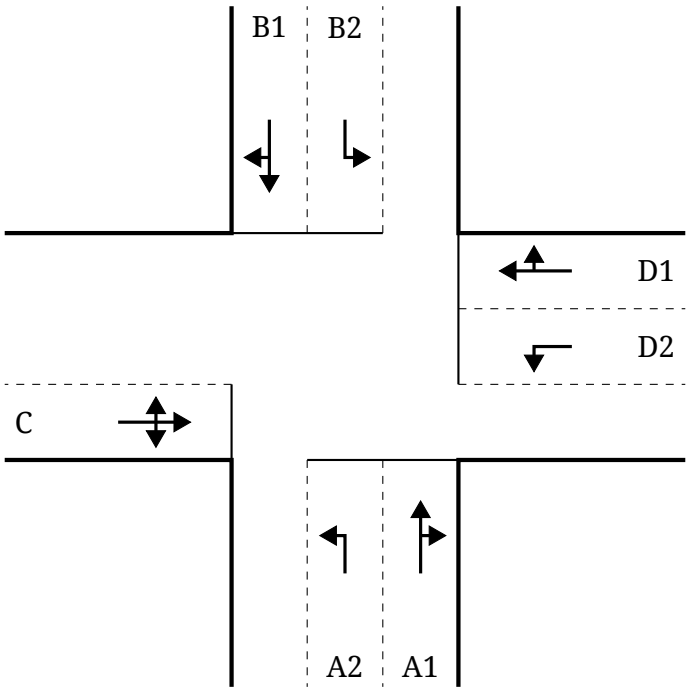


Abbildung 2: Kreuzungsanlage

Gegeben sei die Kreuzung in Abb. 2 (Kreuzungsanlage). Die Ampelanlage schaltet zyklisch durch folgende sechs Phasen:

1. Die Spuren A1 und B1 haben 25 Sekunden grün.
2. Alle Spuren haben 5 Sekunden rot, damit die Kreuzug geräumt wird.
3. Die Spuren A2 und B2 haben 15 Sekunden grün.
4. Alle Spuren haben 5 Sekunden rot, damit die Kreuzug geräumt wird.
5. Die Spuren C, D1 und D2 hat 20 Sekunden grün.
6. Alle Spuren haben 5 Sekunden rot, damit die Kreuzug geräumt wird.

Durchschnittlich gelten für Autos, die über die Kreuzung fahren, folgende Zeitwerte:

- Ein Auto benötigt 2 Sekunden, um die Kreuzung geradeaus zu überqueren.
- Ein Auto benötigt 3 Sekunden, um rechts abzubiegen.
- Ein Auto benötigt 4 Sekunden, um links abzubiegen.

Implementieren Sie die Routine `simuliere` in C++, die die Fahrzeuge über die Kreuzung fahren lässt und berechnet, nach wie vielen Sekunden alle gegebenen Autos die Kreuzung fertig überquert haben. Gehen Sie dabei von folgenden Voraussetzungen aus:

- Alle Autos bewegen sich gemäß den oben gegebenen Durchschnittswerten. Ein Auto fährt erst dann in die Kreuzung ein, wenn das vorhergehende Auto auf derselben Spur die Kreuzung verlassen hat.
- Die Linksabbieger der Richtungen C und D müssen den Gegenverkehr durchfahren lassen, bevor sie abbiegen dürfen. Ein Linksabbieger darf nur dann fahren, wenn während seiner Zeit auf der Kreuzung kein Rechtsabbieger oder Geradeausfahrer aus der Gegenrichtung die Kreuzung belegt. Die schließt die Situation ein, dass das Auto der Gegenrichtung nach dem Linksabbieger losfährt!
- Ein Auto fährt los, solange die Ampel grün ist – selbst wenn die Ampel umschalten würde, bevor es die Kreuzung verlässt.
- Die Simulation fängt immer in der ersten Ampelphase an.

Hinweise:

- In den Materialien finden Sie bereits eine Teilimplementierung der Funktion sowie der nötigen Datenstrukturen. Versuchen Sie, keine zusätzlichen Datenstrukturen in die Klasse `kreuzung` hinzuzufügen; die bestehenden reichen völlig aus.
- Ihre Abgabe muss zusammen mit `kreuzung.cpp` kompilieren. Sie finden Testfälle, die von `kreuzung.cpp` eingelesen werden können, bei den Materialien. Beachten Sie, dass die in den Testfällen gelisteten Spuren so strukturiert sind, dass das ganz rechte Fahrzeug als nächstes fährt. Dies erleichtert Ihnen die Implementierung (siehe auch Kommentare in der gegebenen Datei).
- Die Dateiendung `.hpp` ist nur eine von vielen, die für C++ Header verwendet werden. Lassen Sie sich nicht davon irritieren, dass Sie oder Ihr Editor eine andere gewohnt sind.