252-0027-00: Einführung in die Programmierung

Übungsblatt 13

Abgabe: 16. Dezember 2025, 23:59

Checken Sie die neue Übungs-Vorlage aus. Importieren Sie beide IntelliJ-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben). Vergessen Sie nicht, Tests zu schreiben!

Ab Woche 5, werden die TAs Ihnen nur dann Feedback geben, wenn Sie eine entsprechende Datei namens "requestfeedback.txt" in dem Projekt-Ordner hochladen. In der Text-Datei geben Sie an, für welche Aufgaben Sie Feedback erhalten möchten. Um eine neue Text-Datei in IntelliJ zu erstellen, rechts-klicken Sie den uXX Projekt-Ordner und wählen *New* → *File*. Im erscheinenden Fenster, benennen Sie die Datei mit "requestfeedback.txt" und drücken die Enter-Taste. Danach wählen Sie *Add*, falls IntelliJ Sie fragt, ob Sie die Datei zu git hinzufügen möchten.

Aufgabe 1: Maps

1. Implementieren Sie eine Methode `U12Map.arrayToMap(String[] A)`, die ein Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M` auf die Zahl 0 abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMap` für den Array `[one, two, three, one]` die Map `{one=0, two=0, three=0}` zurück.

2. Implementieren Sie eine Methode `U12Map.arrayToMapOne(String[] A)`, die ein Array `A` von Strings als Parameter akzeptiert und eine Map `M` von String zu Integer zurückgibt. Jeder String, der in `A` auftritt, soll in `M`, falls der String nur einmal vorkommt, auf die Zahl 0 abgebildet werden und, falls der String mehrfach vorkommt, auf die Zahl 1 abgebildet werden. Sie können davon ausgehen, dass `A` nicht `null` ist und dass kein String der empty (leere) String ist.

   Zum Beispiel gibt `arrayToMapOne` für `[one, two, three, one]` die Map `{one=1, two=0, three=0}` zurück und für `[one, two, three, one, one, four, two]` die Map `{four=0, one=1, two=1, three=0}`.

Aufgabe 2: Loop-Invariante

Gegeben ist die Methode `findLargestSmaller(int[] al, int value)`, die in einem *sortierten nicht-leeren* Array von `int`-Werten den grössten Wert findet, der strikt kleiner als `value` ist. `value` muss strikt grösser als `al[0]` sein. Wenn die Elemente `a[0]` .... `a[k]` des Arrays `a` sortiert sind, dann können Sie das mit `sorted(a, 0, k)` abkürzen.

Bestimmen Sie eine *nützliche Schleifeninvariante* für folgendes Programm.

```
static int findLargestSmaller(int[] al, int value) {
    // Precondition: al != null && al.length > 0 &&
        sorted(al, 0, al.length-1) && value > al[0]
    int candidate  = al[0];
    int next = 1;

    // Loop-Invariante:
    while (next < al.length && value > al[next] ) {
        candidate = al[next];
        next++;
    }

    // Postcondition: al != null && al.length > 0 && sorted(al, 0, al.length-1) &&
        ((next < al.length && value <= al[next] && candidate == al[next-1]) ||
        (next == al.length && candidate == al[al.length-1] && value > candidate))

    return candidate;
}
```

Schreiben Sie die Loop-Invariante in die Datei “LoopInvariante.txt”.

Aufgabe 3: Generische Listen

In dieser Aufgabe implementieren Sie eine generische verkettete Liste. Anhang A zeigt eine generische Version eines Interfaces für Listen. Vervollständigen Sie die Klasse `MyListImpl`, sodass die Klasse das `MyList`-Interface implementiert. Dem Interface wurden zwei neue Methoden hinzugefügt. Die Methode `MyListNode getNode(int index)` gibt den Knoten zurück, welcher den Wert der Liste an Position `index` speichert. `MyListeNode` ist selber ein Interface (siehe Anhang B) mit Methoden, welche jeweils den gespeicherten Wert des Knoten, den nächsten Knoten, und ob es einen nächsten Knoten gibt, zurückgeben. Damit überprüfen wir, dass `MyListImpl` eine verkettete Liste implementiert. Die Methode `Iterator<T> iterator()` gibt einen Iterator für die Datenstruktur zurück. Implementieren Sie einen neuen Iterator, das heisst eine Klasse, welche das `Iterator`-Interface implementiert, und geben Sie nicht den Iterator einer anderen Datenstruktur zurück (zum Beispiel den Iterator einer `ArrayList`). Dies können wir in den Tests der Korrektur testen. Die Methode `void remove()` vom Iterator wird nicht benötigt. Die Methode `void addAll(MyList<T> other)` sollte eine konstante Laufzeit haben, sofern es sich bei `other` um eine Liste vom Typ `MyListImpl` handelt. Ein paar Tests finden Sie in `MyListTest`.

Aufgabe 4: Notenauswertung

Die Klasse `Service` stellt verschiedene Analysen für Prüfungsergebnisse von S Studierenden zur Verfügung. Die Liste von Ergebnissen besteht aus S Einträgen, also jeweils einem Eintrag pro Student/in. Jeder Eintrag besteht aus einer Zeile und enthält (in dieser Reihenfolge):

1. die Immatrikulationsnummer des Studierenden (ein identifizierender positiver `int`-Wert)
2. drei Noten (drei reelle Zahlen im Bereich von 1.0 bis 6.0, getrennt durch Leerzeichen)

Die drei Noten gehören zu den Fächern *Fach 1*, *Fach 2* und *Fach 3*. Zusätzliche Leerzeilen und -zeichen sollen ignoriert werden. Eine Beispiel für eine Liste für 3 Studierende ist:

```
111111004  5.0  5.0  6.0
111111005  3.75 3.0  4.0
111111006  4.5  2.25 4.0
```

Ihre Aufgabe ist es nun, die `Service`-Klasse und ihre Analysen zu implementieren. Die `Service`-Klasse hat einen Konstruktor, welcher alle Prüfungsergebnisse aus einem `Scanner` auslesen und damit das `Service`-Objekt initialisieren soll. Das Objekt soll so initialisiert werden, dass die vorgegebenen Methoden ihre Analysen durchführen können. Sie dürfen dabei Attribute und zusätzliche Methoden frei bestimmen.

1. Implementieren Sie nun die Methode `critical()`, welche die zwei Argumente `bound1` und `bound2` erwartet. Die Methode sucht alle "kritischen" Fälle und gibt eine Liste dieser Studierenden zurück. Ein Student darf maximal einmal in der Liste vorkommen. Die zurückgegebene Liste besteht aus den Immatrikulationsnummern dieser Studierenden (in beliebiger Reihenfolge).

   Ein/e Student/in gilt als kritisch, wenn die Note in *Fach 1* $\leq$ `bound1` und die Summe der Noten für *Fach 2* und *Fach 3* kleiner als `bound2` ist.

   Für das obige Beispiel gäbe `critical(4, 8)` eine Liste mit dem Element `111111005` zurück.

2. Implementieren Sie nun die Methode `top()`, welche die Studierenden mit den besten Ergebnissen zurückgeben soll. Der Parameter `limit` bestimmt die maximale Anzahl der zurückzugebenden Studierenden. Falls weniger Ergebnisse als `limit` existieren, sollen alle gefundenen zurückgegeben werden.

   Der Rückgabewert der Methode ist wieder eine Liste der Immatrikulationsnummern. Ein Student darf maximal einmal in der Liste vorkommen. Diese Liste soll absteigend nach der Leistung sortiert sein (der/die Student/in mit dem besten Ergebnis zuerst). Dabei gilt, dass ein Ergebnis A besser ist als ein Ergebnis B, wenn die Summe aller Noten von A grösser ist als die Summe der Noten von B. Sind die Summen gleich, sind die Ergebnisse gleich gut (und die Reihenfolge in der Liste somit egal).

   Für das obige Beispiel gäbe `top(2)` entweder die Liste `[111111004, 111111006]` oder die Liste `[111111004, 111111005]` zurück (beide wären richtig).

In der Klasse `ServiceTest` finden Sie einen ersten kleinen JUnit-Test als Starthilfe. Ausserdem dürfen Sie folgende Annahmen machen: Der Parameter `limit` ist immer grösser als 0 und die beiden Parameter für `critical()` sind immer im Bereich von 0.0 bis 100.0.

Aufgabe 5: Interpreter

In der letzten Übung implementierten Sie einen Evaluator für mathematische Ausdrücke. In dieser Aufgabe erweitern Sie ihn so, dass er statt einzelnen Ausdrücken einfache Programme mit mehreren Anweisungen auswertet. Das nennt man auch *interpretieren* und ein solches Programm entsprechend *Interpreter*.

digit ← `0` | `1` | ... | `9`
char ← `A` | `B` | ... | `Z` | `a` | `b` | ... | `z`
num ← *digit* { *digit* } [`.` *digit* { *digit* }]
var ← *char* { *char* }
func ← *char* { *char* } `(`
op ← `+` | `-` | `*` | `/` | `^`
atom ← *num* | *var*
term ← `(` *expr* `)` | *func* *expr* `)` | *atom*
expr ← *term* [*op* *term*]
stmt ← *var* `=` *expr* `;`
prog ← { *stmt* }

Abbildung 1: **EBNF-Beschreibung** von *prog*

```
alpha = i * ((2*PI) * (1 / 6.05));
size = (0.25 * cos(t/2)) + 0.75;

x = cos(alpha + (0.3 * t)) * size;
y = sin((1.5 * alpha) + t) * size;

r = (cos(alpha + (2 * t)) + 1) / 2;
g = (sin(alpha + (2 * t)) + 1) / 2;
b = (cos(alpha +  (PI/2)) + 1) / 2;
```

Abbildung 2: Beispiel-Programm

Abbildung 1 zeigt die EBNF-Beschreibung für Programme (*prog*). Es gibt nur eine Art von Anweisung (*stmt*), nämlich eine Zuweisung eines Ausdrucks zu einer Variable. Beachten Sie, dass die Beschreibung zugunsten der Lesbarkeit nicht alle Tokenizer-Regeln explizit enthält: die RHS der Parser-Regeln enthalten teilweise auch direkt Buchstaben (`blau`). Der aktualisierte `Tokenizer` in der Übungsvorlage stellt aber auch für die Zeichen `( ) = ;` die benötigten Methoden zur Verfügung.

In der Vorlage befindet sich die `Interpreter`-Klasse, welche (abgesehen von Klassen- und Methodennamen) dem fertigen `ExprEvaluator` der letzten Übung entspricht. Die `Interpreter`-Klasse befindet sich in einem *Paket* (engl. package) namens `language`. Platzieren Sie alle Klassen, welche Sie für diese Aufgabe erstellen, ebenfalls in diesem Paket.

1. Ändern Sie den `Interpreter` so, dass er Programme, die der Beschreibung in Abbildung 1 entsprechen, interpretiert. Die Semantik von Zuweisungen soll dieselbe sein wie in Java.

   Aufgrund der Möglichkeit, Variablen zu definieren, reicht die einfache rekursive Struktur des `ExprEvaluators` nicht aus, um Programme zu interpretieren. Der Interpreter muss sich zusätzlich den Zustand des Programms, d.h. die definierten Variablen und deren Werte, merken. Dafür eignet sich eine `Map`, oder genauer, da wir eine Abbildung von Variablen-Namen auf Werte brauchen, eine `Map<String,Double>`. Erstellen Sie eine solche im `Interpreter`-Konstruktor und verwenden Sie sie wo nötig. Neu könnte der Konstruktor auch eine `Map` von Variablennamen und -werten entgegennehmen anstatt dem einzelnen Wert für die *x*-Variable.

   Schreiben Sie die fehlenden `interpret*(...)`-Methoden und ändern Sie `interpret(String)` entsprechend. Beachten Sie, dass Anweisungen (*stmt*s) [1] im Gegensatz zu Ausdrücken (*expr*s)

[1] in unserer verinfachten Sprache

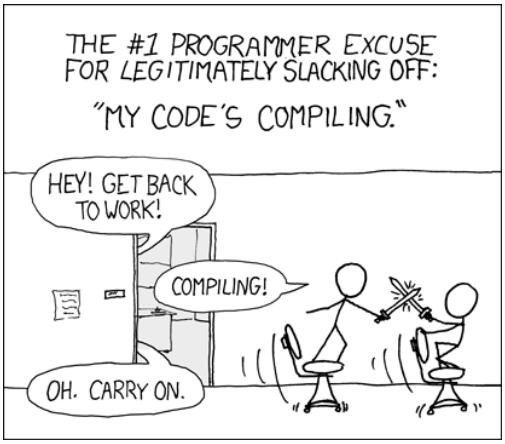
selbst keinen Wert haben, sondern nur einen Effekt auf den Programmzustand (z.B. die Änderung eines Wertes einer Variable). Ein Programm als Ganzes hat ebenfalls keinen Wert, also können Sie den Rückgabetyp von `interpret(String)` auf `void` ändern.

2. Schreiben Sie ein Java-Programm `Repl` (im Paket `language`), welches eine *REPL* implementiert. "REPL" steht für *read–eval–print loop* und bezeichnet ein (Java-)Programm, welches wiederholt Anweisungen von der Konsole liest (*read*), diese auswertet (*eval*) und das Resultat ausgibt (*print*). Der Programmzustand (d.h. die Werte von Variablen) wird über mehrere Anweisungen hinweg mitgeführt und durch das Interpretieren von Anweisungen verändert.

   Sie können sich an der `EvaluatorApp` der letzten Übung orientieren. Da Programme selber keinen Wert haben, gibt es kein explizites Resultat zum Ausgeben. Geben Sie stattdessen nach jeder Ausführung alle definierten Variablen und deren Werte aus. Dafür brauchen Sie Zugriff auf die Variablen-`Map` des Interpreters. Achten Sie darauf, dass die `ProgramException` richtig behandelt wird.

Aufgabe 6: Compiler

Das Interpretieren von Quellcode ist ineffizient und langsam. Deshalb werden Java-Programme auch zuerst *kompiliert*, bevor sie ausgeführt werden. Kompilieren heisst, den Quellcode in eine Form zu übersetzen, die vom Computer direkt(er) ausgeführt werden kann. In dieser Übung schreiben Sie einen einfachen Compiler, der den Quellcode von Programmen von Aufgabe 5 in eine Serie von Instruktionen übersetzt, die effizient ausgeführt werden können.



xkcd: Compiling by Randall Munroe (CC BY-NC 2.5)

Die Programmiersprache in Aufgabe 5 hat eine rekursive Struktur: Ausdrücke können mehrere Ausdrücke enthalten, welche wiederum mehrere Ausdrücke enthalten können, usw. Um eine solche Struktur in eine lineare Folge von Instruktionen umzuwandeln, verwenden wir einen *Operanden-Stack*. Dies ist ein `Stack` (wie der Call-Stack, den Sie in der Vorlesung gesehen haben), der Zwischenresultate von Berechnungen speichert. Instruktionen können Werte auf den Stack "pushen" oder Werte ab dem Stack "poppen" und verwenden. Es gibt folgende Arten von Instruktionen:

`CONST` c	Pusht den konstanten Wert c auf den Stack
`LOAD` v	Lädt den Wert der Variable v und pusht ihn auf den Stack
`STORE` v	Poppt einen Wert vom Stack und speichert ihn in der Variable v
`OP` $\oplus$	Poppt zwei Werte r und l vom Stack (zuerst r, dann l) berechnet $l \oplus r$ und pusht das Resultat zurück auf den Stack
`FUNC` f	Poppt einen Wert x vom Stack, berechnet $f(x)$ und pusht das Resultat zurück auf den Stack

Programmteil	Instruktionen
`b`	`LOAD b`
`1`	`CONST 1`
`b + 1`	`LOAD b  CONST 1  OP +`
`(b + 1)`	`LOAD b  CONST 1  OP +`
`2`	`CONST 2`
`c`	`LOAD c`
`2 * c`	`CONST 2  LOAD c  OP *`
`sin(2 * c)`	`CONST 2  LOAD c  OP *  FUNC sin`
`(b + 1) / sin(2 * c)`	`LOAD b  CONST 1  OP +  CONST 2  LOAD c  OP *  FUNC sin  OP /`
`a = (b + 1) / sin(2 * c);`	`LOAD b  CONST 1  OP +  CONST 2  LOAD c  OP *  FUNC sin  OP /  STORE a`

Tabelle 1: Kompilieren der Anweisung `a = (b + 1) / sin(2 * c);`

Unten sehen Sie ein kleines Programm, das aus solchen Instruktionen besteht. Es lädt zuerst den Wert der Variable `x` und dann einen konstanten Wert 2 auf den Stack. Die nächste Instruktion holt sich die beiden Werte vom Stack, multipliziert sie und pusht das Resultat zurück. Die letzte Instruktion schliesslich holt diesen Wert vom Stack und speichert ihn zurück in die Variable `x`:

```
LOAD x
CONST 2
OP *
STORE x
```

Sie sollen nun einen `Compiler` schreiben, welcher ein Programm in eine Liste solcher Instruktionen kompiliert. Der `Compiler` geht grundsätzlich gleich vor wie der `Interpreter`: er parst das Programm rekursiv und berechnet gleichzeitig ein Resultat. Im Gegensatz zum `Interpreter` berechnet er aber keine Werte, sondern generiert Listen von Instruktionen.

Um zu verstehen, wie diese Instruktionen genau generiert werden, betrachten Sie Tabelle 1, welche zeigt, wie der Ausdrucks `a = (b + 1) / sin(2 * c);` kompiliert wird. In der linken Spalte stehen die Programmteile (in der Reihenfolge, in der sie auch schon vom Interpreter fertig geparst werden) und in der rechten Spalte die entsprechenden Instruktionen. Zum Beispiel sehen Sie, dass der `Compiler` für die Zahl `1` im Quellcode die Instruktion `CONST 1` generiert. Oder, dass er für einen binären Ausdruck zuerst die Instruktionen des linken Teils, dann die Instruktionen des rechten Teils und schliesslich eine `OP`-Instruktion generiert.

1. Erstellen Sie zuerst einige Klassen `ConstInstr`, `LoadInstr`, usw. für die verschiedenen Arten von Instruktionen. Erstellen Sie dazu am besten ein Unterpaket `language.instructions` und deklarieren Sie die `*Instr`-Klassen als `public`. Für die `OP`- und `FUNC`-Instruktionen können Sie entscheiden, ob Sie jeweils eine einzige Klasse verwenden möchten oder mehrere Subklassen, eine für jeden unterstützten Operator, bzw. für jede Funktion.

   Alle Instruktionen sollen ein Interface `Instr` (ebenfalls im `language.instructions`-Paket zu erstellen) mit einer `execute()`-Methode implementieren. Diese Methode nimmt als Argumente den Operanden-`Stack` und die Variablen-`Map` und führt die Instruktion aus. Falls ein Fehler auftritt (z.B., wenn eine Variable nicht definiert ist oder wenn der Stack nicht die erwartete Grösse hat), soll die Methode eine Ausnahme der Klasse `ExecutionException` (die Sie erstellen) werfen. Diese Art von Exception soll *checked* sein.

2. Erstellen Sie eine `Program`-Klasse, welche eine Liste von Instruktionen enthält und eine `execute()`-Methode hat, welche diese Instruktionen ausführt. Diese Methode nimmt eine Variablen-`Map` entgegen, die vom Programm verwendet und aktualisiert wird.

   Das Ausführen der Instruktionen ist denkbar einfach: Erst wird ein Operanden-Stack erstellt, und dann wird eine Instruktion nach der anderen über ihre `execute()`-Methode ausgeführt.

3. Schreiben Sie jetzt die `Compiler`-Klasse. Sie können sie analog zur `Interpreter`-Klasse entwerfen, mit `compile*()`- statt `interpret*()`-Methoden. Jede dieser Methoden soll eine Liste zurückgeben, welche die Instruktionen enthält, die dem geparsten Programm-Teil entsprechen. Die Haupt-Methode `compile(String)`, schliesslich, soll alle Instruktionen in eine `Program`-Instanz packen und diese zurückgeben.

Aufgabe 7: EBNF-Baum (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

In diesem Kurs haben Sie bereits gelernt, was EBNF ist. In verschiedenen Aufgaben durften Sie bestimmen, ob eine Zeichenfolge legal ist für eine EBNF-Beschreibung. In dieser Aufgabe werden Sie einen Schritt weiter gehen. Sie werden eine Methode implementieren, die für eine gegebene EBNF-Beschreibung alle legalen Zeichenfolgen bis zu einer bestimmten Länge generiert. Wenn die EBNF-Beschreibung Ihnen als String übergeben würde, so müssten Sie zuerst diese einlesen und in eine brauchbare Form bringen. Das ist aber ein Thema für eine andere Vorlesung. Stattdessen erhalten Sie die EBNF-Beschreibung in Form eines Baumes von verschiedenen Klassen. In der ersten Teilaufgabe werden Sie solche EBNF-Bäume in einen String umwandeln, der die EBNF-Beschreibung in der gelernten Art und Weise repräsentiert. In der zweiten Teilaufgabe implementieren Sie dann die Logik, um alle legalen Zeichenfolgen bis zu einer bestimmten Länge zurückzugeben.

Die Klasse `EBNFNode` ist die abstrakte Superklasse der Terminals, Nonterminals und aller Kontrollformen. Sie ermöglicht uns, die rechte Seite von EBNF-Regeln als Bäume zu repräsentieren. Die dafür benötigten Unterklassen sind in Tabelle 2 beschrieben.

Sie dürfen davon ausgehen, dass `Literal` nur Buchstaben oder Zahlen enthält und der Name von `RuleName` nur aus Buchstaben besteht. Sie müssen sich also nicht um Sonderzeichen kümmern. Des Weiteren dürfen Sie davon ausgehen, dass die gegebenen `EBNFNode`-Instanzen azyklisch miteinander verlinkt sind und jede `EBNFNode`-Instanz nur als Unterknoten (also `child`, `left`, `right`, `first` und `second`) von höchstens einer anderen `EBNFNode`-Instanz auftritt. Eine `Option`-Instanz kann also zum Beispiel nicht sich selbst als `child` haben. Sie dürfen auch annehmen, dass `name` von `RuleName` und die Unterknoten aller `EBNFNodes` nie `null` sind.

Mithilfe der oben genannten Klassen kann bisher nur die rechte Seite von EBNF-Regeln ausge-

Klasse	EBNF
`Epsilon()`	ϵ
`Literal(char literal)`	literal
`RuleName(String name)`	<name>
`Brackets(EBNFNode child)`	(child)
`Option(EBNFNode child)`	[child]
`Repetition(EBNFNode child)`	{child}
`Sequence(EBNFNode left,EBNFNoderight)`	left right
`Alternative(EBNFNode first,EBNFNodesecond)`	first|second

Tabelle 2: Die verschiedenen Klassen für den EBNF-Baum. Der blaue Text in der EBNF-Spalte ist jeweils ein Platzhalter für ein Zeichen (bei Literal), ein String (bei RuleName) oder ein Terminal, Nonterminal oder eine Kontrollstuktur (bei den verbleibenden Klassen).

drückt werden. Um ganze EBNF-Regeln und -Beschreibungen auszudrücken, brauchen wir zusätzlich die Klasse `EBNFRules`. Diese Klasse hat ein Feld `rules` vom Typ `Map<String, EBNFNode>`. Ein Eintrag mit Schlüssel `key` und Wert `value` bedeutet dabei, dass es eine EBNF-Regel mit Namen `key` gibt und die rechte Seite dieser Regel ist durch `value` gegeben. Sie dürfen davon ausgehen, dass `key` nur aus Buchstaben besteht und dass `value` nie `null` ist und die Map von `EBNFRules` und dessen Inhalt (also auch alle EBNF-Bäume) von uns nach dem Konstruktor-Aufruf nicht mehr verändert werden.

a) Die Klassen `EBNFNode` und `EBNFRules` implementieren das Interface `EBNFString`. Implementieren Sie dafür die Methode `String toEBNFString()` der Klasse `EBNFRules` und der Unterklassen von `EBNFNode`, welche die EBNF-Bäume und die EBNF-Regel-Map als "normalen" EBNF-String zurückgeben. Die zweite Spalte der Tabelle 2 zeigt dabei, wie der zurückgegebene String der jeweiligen `EBNFNode`-Unterklasse aussehen soll. Der zurückgegebene String soll keine Leerzeichen enthalten. Damit Sie keine Sonderzeichen benötigen, gibt diese Methode in der Klasse `Epsilon` einfach einen leeren String zurück. In der Methode der Klasse `Sequence` müssen sie die zusätzlich die EBNF-Strings der Unterklassen `left` und/oder `right` in Klammern setzen, wenn es sich bei diesen um eine `Alternative` handelt, da die Sequenz eine höhere Präzedenz hat. Das letzte Beispiel verdeutlicht dieses Szenario.

- `EBNFNode node1 = new Option(new Literal('A'));`
  `node1.toEBNFString()` soll folgenden String zurückgeben: `[A]`
- `EBNFNode node2 = new Sequence(new Literal('B'), new Literal('C'));`
  `node2.toEBNFString()` soll folgenden String zurückgeben: `BC`
- `EBNFNode node3 = new Alternative(new RuleName("ab"), new Literal('D'));`
  `node3.toEBNFString()` soll folgenden String zurückgeben: `<ab>|D`
- `EBNFNode node4 = new Repetition(new Brackets(new Epsilon()));`
  `node4.toEBNFString()` soll folgenden String zurückgeben: `{()}`
- `EBNFNode node5 = new Sequence(new Literal('E'),`
  `new Alternative(new Literal('F'), new Literal('G')));`
  `node5.toEBNFString()` soll folgenden String zurückgeben: `E(F|G)`

Obwohl der EBNF-Baum hier keine Klammern enthält, müssen diese hier für den String hinzugefügt werden.

Bei der Methode `toEBNFString()` von `EBNFRules` soll der zurückgegebene String jede EBNF-Regel in der Map auf einer eigenen Zeile enthalten. Ein Zeilenumbruch in einem String wird durch `\n` repräsentiert. Die Regeln sollen dabei in lexikographischer Reihenfolge aufgeführt werden (die Java-Library sortiert Strings standardmässig mit dieser Ordnung). Das folgende Beispiel verdeutlicht das gewünschte Format:

- ```
  Map<String,EBNFNode> map1 = new TreeMap<String,EBNFNode>();
  map1.put("ab", node1);
  map1.put("a", node2);
  map1.put("aa", node3);
  map1.put("b", node4);
  EBNFRules rules1 = new EBNFRules(map1);
  ```

  `rules1.toEBNFString()` soll die folgenden vier Zeilen als einen String zurückgeben:

  ```
  <a><-BC
  <aa><-<ab>|D
  <ab><-[A]
  <b><-{()}
  ```

Hinweis: Bedenken Sie, dass die an `EBNFRules` übergebene Map auch eine `HashMap` sein kann.

b) Implementieren Sie die Methode `Set<String> getShortestWords(String name, int limit)` der Klasse `EBNFRules`, so dass sie ein Set mit allen Zeichenfolgen mit höchstens `limit` Zeichen zurückgibt, welche für die EBNF-Beschreibung mit der Startregel `name` legal sind. Falls `EBNFRules` keine EBNF-Regel mit Name `name` enthält oder sich in der Sprachdefinition von `name` ein EBNF-Regelname befindet, welcher in der gegebenen `EBNFRules`-Instanz nicht definiert ist, soll eine `IllegalEBNFDescriptionException` geworfen werden. Beispiele:

```
Map<String,EBNFNode> map2 = new HashMap<String,EBNFNode>();
map2.put("xrule", new Alternative(new Literal('X'), new Literal('Q')));
map2.put("yrule", new Repetition(new Literal('Y')));
map2.put("zrule", new Sequence(new Literal('Z'), new RuleName("yrule")));
map2.put("qrule", new RuleName("house"));
EBNFRules rules2 = new EBNFRules(map2);
```

- `rules2.getShortestWords("xrule", 1)` soll ein Set mit den folgenden zwei Strings zurückgeben: `"X"` und `"Q"`.
- `rules2.getShortestWords("yrule", 3)` soll ein Set mit den folgenden vier Strings zurückgeben: `""`, `"Y"`, `"YY"` und `"YYY"`.
- `rules2.getShortestWords("zrule", 4)` soll ein Set mit den folgenden vier Strings zurückgeben: `"Z"`, `"ZY"`, `"ZYY"` und `"ZYYY"`.
- `rules2.getShortestWords("tree", 5)` soll eine `IllegalEBNFDescriptionException` werfen, da keine Regel mit diesem Namen existiert.
```

- `rules2.getShortestWords("qrule", 7)` soll eine `IllegalEBNFDescriptionException` werfen, da in der Definition die Regel `<house>` verwendet wird, welche jedoch in `rules2` nicht existiert.

Hinweis: Wir haben in der Klasse `EBNFNode` bereits eine Methode `Set<String> getShortestWords(EBNFRules rules, int limit)` definiert. Wir empfehlen Ihnen, diese Methode in den Unterklassen zu überschreiben und diese Methode dann für die Implementation der Methode `getShortestWords` von `EBNFRules` zu nutzen.

Anhang B: MyList-Interface

```
public interface MyList<T> {

        /**
         * Return the value at position 'index'.
         * Throws a NoSuchElementException if the argument exceeds the list size.
         * */
        public T get(int index);

        /**
         * Return the list node at position 'index'.
         * Throws a NoSuchElementException if the argument exceeds the list size.
         * */
        public MyListNode<T> getNode(int index);

        /** Set the value at position 'index' to 'value'. */
        public void set(int index, T value);

        /** Returns whether the list is empty (has no values). */
        public boolean isEmpty();

        /** Returns the size of the list. */
        public int getSize();

        /** Inserts 'value' at position 0 in the list. */
        public void addFirst(T value);

        /** Appends 'value' at the end of the list. */
        public void addLast(T value);

        /** Appends the 'other' list to the end of the list. */
        public void addAll(MyList<T> other);

        /**
         * Removes and returns the first value of the list.
         * Throws a NoSuchElementException if the List is empty.
         * */
        public T removeFirst();

        /**
         * Removes and returns the last value of the list.
         * Throws a NoSuchElementException if the List is empty.
         * */
        public T removeLast();

        /** Removes all values from the list, making the list empty. */
        public void clear();

        /** Returns an iterator to the data structure. */
        public Iterator<T> iterator();
```

```
}
```

Anhang C: MyListNode Interface

```
public interface MyListNode<T> {

  /** Returns the value stored in the node. */
  public T value();

  /** Sets the value stored in the node. */
  public void setValue();

  /** Returns false iff this is the last node of the list. */
  public boolean hasNext();

  /** Returns next node. */
  public MyListNode<T> next();

  /** Sets the next node. */
  public void setNext();
}
```