Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 12

TECHNISCHE UNIVERSITÄT DARMSTADT

Entwurf

Achtung: Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

Hausübung 12

Gesamt: 32 Punkte

Automaten parsen

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h12` und ggf. `src/test/java/h12`.

Einleitung

In dieser Übung werden Sie sich mit dem Einlesen und Ausgeben von Daten befassen. Es handelt sich hierbei nicht um ein *neues* Thema. Vielmehr werden Sie die in den vorherigen Wochen gelernten Konzepte anwenden und dabei lernen, wie Sie Dateien ein- und auslesen können. Dies ist wichtig, denn selbst der nützlichste Algorithmus hat keinen Sinn, wenn die Eingabedaten nicht in den Algorithmus ein- bzw. Ausgabedaten nicht ausgegeben werden können. Wir werden dies anhand eines endlichen Automaten, wie er in der Digitaltechnik gelehrt wird, durchführen. Im Folgenden werden jedoch alle notwendigen Grundlagen eingeführt, es ist kein Vorwissen erforderlich.

Ein endlicher Automat ist ein abstraktes mathematisches Modell in der theoretischen Informatik, das dazu dient, bestimmte Arten von Berechnungen oder Prozessen zu repräsentieren. Der Begriff „endlich“ bezieht sich darauf, dass der Automat eine begrenzte Anzahl von Zuständen hat. Damit wäre auch schon die erste Komponente eines endlichen Automaten benannt: die *Zustände*. Ein Automat besitzt eine Menge an Zuständen und der Automat befindet sich zu jedem Zeitpunkt in genau einem dieser Zustände. Zwischen diesen Zuständen gibt es *Übergänge*, die definieren unter welcher Bedingung (Eingabesymbol) man von einem in einen anderen Zustand gelangt.

Es gibt verschiedene Untergruppen der endlichen Automaten, wobei in dieser Hausübung ausschließlich deterministische Moore-Automaten drankommen. Das „deterministisch“ gibt an, dass für jede Kombination von aktuellem Zustand und Eingabesymbol eindeutig ist, welcher Zustand der Folgezustand ist. Eine weitere Komponente eines Automaten sind die Ausgangssymbole, also die *Ausgabe* des Automaten. Moore-Automaten erzeugen Ausgaben basierend allein auf ihrem aktuellen Zustand und hängen nicht vom aktuellen Eingangssymbol ab.

Wir gehen einen Schritt weiter und konkretisieren die Aufgabe weiter. Und zwar werden wir nicht eine Menge an „fiktiven“ Eingabesymbolen verwenden, sondern einen bool'schen Ausdruck der Eingänge.

In Abbildung 1 sehen Sie ein Beispiel eines Automaten. Das Symbol `s` steht für *Set* und `r` für *Reset*. Ein *Set* sorgt dafür, dass der Automat in Zustand `ON` übergeht, und `Reset` dafür, dass er in `OFF` geht. Hierbei ist das `Reset` stärker, also falls beide Signale aktiv sind, geht der Automat in `OFF` über. Dies entspricht der Funktion eines RS-Latches. Oft werden außerdem Schlingen, also Übergänge bei denen Ausgangs- und Zielzustand identisch sind, weggelassen. Das bedeutet, dass für jede Eingangskombination eines Zustandes, die nicht spezifiziert ist, implizit ein Übergang auf sich selbst angenommen wird. In der Abbildung ist jedoch jeder Übergang explizit dargestellt. Eine Variable x wird direkt verwendet und eine Variable $'x$ negiert.

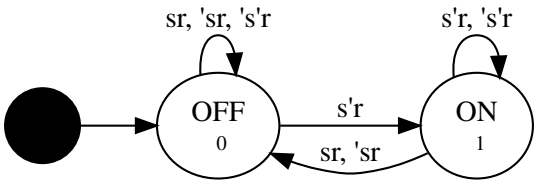

Abbildung 1: RS-Latch als endlicher Automat

Diese bool'schen Ausdrücke können nun auch anders formuliert werden. Dazu muss zunächst gelten, dass die Reihenfolge aller Signal für jeden Übergang die selbe Reihenfolge aufweisen. Dann kodiert man ein direktes Signal als `1` und ein negiertes Signal als `0`. Es wird also kodiert, welchen Zustand das Signal annehmen muss, damit es die Bedingung des Übergangs erfüllt. So wird zum Beispiel aus $s'r$ `1 0`, aus sr `1 1` und aus $'sr$ `0 1`. Den nächsten Schritt den man machen kann, ist, dass man die ausgehenden Übergänge eines Zustandes vereinfacht. Schaut man sich die Übergänge `1 1` und `0 1` des Zustands `ON` an, so fällt auf, das der Wert von s irrelevant für die Bedingung ist. Man kann sich auch für den bool'schen Ausdruck überlegen, dass $sr +' sr = r(s +' s) = r$ ist. Die Bedingung kann dadurch zu `- 1` umformuliert werden. Das `-` ist hierbei ein sogenanntes **Don't care**. Der vereinfachte Automat ist in Abbildung 2 dargestellt.

In dieser Hausübung werden Sie das Parsen des **kiss2**-Formats implementieren, welches ein häufig genutztes Format für solche endlichen Automaten ist. Die Darstellung aus Abbildung 2 enthält hierbei alle relevanten Informationen, um eine gültige kiss2-Datei für den Automaten zu schreiben. kiss-Dateien erlauben hierbei auch, dass Kanten auf sich selbst

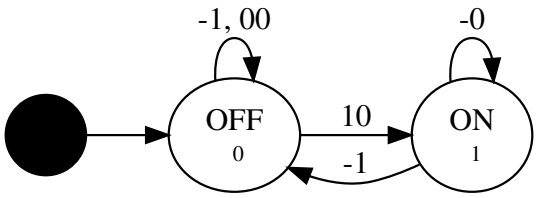

Abbildung 2: Vereinfachter Automat des RS-Latch aus 1

weggelassen werden können. Eine mögliche kiss2-Datei für das RS-Latch ist dabei in Codeblock gegeben.

RS-Latch in kiss2

```
.i 2
.o 1
.p 5
.s 2
.r OFF
-1 OFF OFF 0
00 OFF OFF 0
10 OFF ON  1
-0 ON  ON  1
-1 ON  OFF 0
```

Eine kiss2-Datei beginnt mit einem Header, der verschiedene Informationen enthält. Der Parameter `.i` ist eine natürliche Zahl und gibt an wie viele Bit das Eingangssignal breit ist. `.o` steht analog für die Breite des Ausgangssignals. Der Parameter `.p` steht dafür wie viele Übergänge in der Datei nach dem Header folgen. `.s` steht für die Anzahl der Zustände und `.r` definiert den Startzustand. Alle Headereinträge mit Ausnahme des Startzustandes sind verpflichtend und müssen in jeder gültigen kiss2-Datei enthalten sein.

Nach dem Header folgen die sogenannten *Terms* oder, wie bereits erwähnt, die möglichen Übergänge. Ein Term gliedert sich hierbei in vier Teilausdrücke, die jeweils durch beliebig viele Whitespace voneinander getrennt sind. Der erste Ausdruck ist das Eingabesignal, der zweite Ausdruck der Start- und der dritte Ausdruck der Zielzustand des Übergangs. Der letzte Ausdruck ist das Ausgabesignal.

Möglicherweise verwirrt es Sie, dass die Ausgabesignale nicht an den Zustand gebunden sind, sondern an den Übergang. In der Einleitung zu Automaten wurde eben erläutert, dass bei Moore-Automaten jeder Zustand eine Ausgabe definieren. Dem ist auch immer noch so, allerdings erlauben kiss2-Dateien es Moore-Automaten kompakter darzustellen, sodass verschiedene Übergänge in den selben Zielzustand zeigen, aber unterschiedliche Ausgabesignal definieren. Es ist einfach möglich diese Vereinfachung rückgängig zu machen, indem man jede Kombination des Zielzustandes mit dem jeweiligen Ausgabesignal des Zustands bildet und dafür einen Zustand einfügt. Sie müssen sich hierum nicht kümmern, es wurde jedoch der Vollständigkeit halber erwähnt.

H1: Dateien *lesbar* machen **2 Punkte**

Implementieren Sie in der Klasse `FileSystemIOFactory` im Package `h12.ioFactory` das Interface `IOFactory` und, dadurch die in diesem Interface definierten Methoden `supportsReader()`, `createReader(String ioName)`, `supportsReader()` und `createWriter(String ioName)`.

Die `FileSystemIOFactory` erlaubt im Vergleich zur `ResourceIOFactory`, welche in der Vorlage bereits implementiert ist und woran Sie keine Änderungen vornehmen sollen, sowohl lesenden als auch schreibenden Zugriff. Daher sollen die Methode `supportsReader()` und `supportsWriter()` jeweils `true` zurückliefern.

Die Methode `createReader(String ioName)` liefert ein `BufferedReader`-Objekt zurück. Ein `Reader` ist ein Objekt, welches es ermöglicht Zeichenströme zu verwalten. Der `BufferedReader` ist hierbei ein besonderer Reader, der unter anderem die Pufferung der Eingabe realisiert. Erzeugen Sie einen neuen `BufferedReader` und liefern Sie diesen zurück. Der `BufferedReader` basiert dabei auf einem `FileReader`, welchem Sie wiederum die Zieldatei `ioName` als Konstruktorargument übergeben müssen.

Analog implementieren Sie `createWriter(String ioStream)`, welche einen `BufferedWriter` basierend auf einem `FileWriter` für die Datei `ioName` zurückliefert.

Exkurs:

Vielleicht wundern Sie sich wofür die `IOFactory` genutzt wird. Die `IOFactory` ist eine sogenannte *abstrakte Fabrik*. Es handelt sich hierbei um ein grundlegendes Entwurfsmuster der Softwareentwicklung. Ein Entwurfsmuster ist eine bewährte allgemeine Lösung für ein wiederkehrendes Problem in der Softwareentwicklung. Entwurfsmuster können dann wie Vorlagen auf ein Entwurfsproblem gelegt werden, um dieses zu lösen. Die abstrakte Fabrik fällt konkret in die Kategorie der Erzeugungsmuster und dient dazu, die Erstellung von Objekten zu abstrahieren. Sie werden einige Entwurfsmuster im weiteren Laufe Ihres Studiums kennenlernen, besonders in der Veranstaltung Software Engineering.

Im Folgenden werden wir das Entwurfsmuster kurz am Beispiel der `IOFactory` erläutern. Falls Sie mehr darüber erfahren möchten, konsultieren Sie zum Beispiel `https://refactoring.guru/design-patterns/abstract-factory`.

Im Wesentlichen ermöglicht die Abstract Factory das Gruppieren von Objekten zu Familien, ohne dabei die konkreten Klassen bzw. Informationen der einzelnen Objekte offenzulegen. Die `IOFactory`, welches die abstrakte Fabrik darstellt, definiert so zum Beispiel `createReader(String ioName)`. Jede implementierende Klasse muss den Vertrag für die Methode `createReader` einhalten. Der Vertrag für `createReader` ist, dass ein `BufferedReader` erzeugt und zurückgegeben wird, der auf die Ressource `ioName` verweist. Das besondere ist jetzt, die `IOFactory` von der Erzeugung der eigentlichen Objekte abstrahiert. So erzeugt die `createReader`-Methode der `FileSystemIOFactory` einen `BufferedReader` der eine Datei des Dateisystems öffnet und die der `ResourceIOFactory` einen `BufferedReader` basierend auf einer Ressource. Eine weitere Möglichkeit wäre eine Implementierung die auf ein Datenbank im Internet zugreift. Das relevante hierbei ist, dass egal welche Implementierung genutzt wird, eine `IOFactory` als statischer Typ genutzt werden kann. Das bedeutet, dass man dem eigentlichen Algorithmus eine `IOFactory` übergibt. Der Algorithmus ist jedoch unabhängig davon, von welcher Datenquelle die Daten stammen, respektive welcher `BufferedReader` genutzt wird.

H2: Kommentare entfernen 4 Punkte

In dieser Aufgabe werden Sie die Klasse `CommentFreeReader` des Packages `h12.parse` implementieren. Die Klasse macht genau das, was man durch den Namen schon erahnen lässt: Es werden die Kommentare eines Zeichenstroms entfernt. In Kiss2 werden Kommentare mit einem `//` eingeleitet und enden am Ende der Zeile. Ein Kommentar darf auch eine ganze Zeile einnehmen, also direkt am Anfang der Zeile beginnen.

Die Klasse ist bereits definiert, sie müssen lediglich die `public`-Methoden `hasNext()`, `read()` und `peek()` und die `private`-Methode `lookAhead()` implementieren. Die Klasse bekommt im Konstruktor einen `BufferedReader` übergeben, der in der Klasse als Attribut `reader` gespeichert ist. Über diesen `BufferedReader` wird der Eingabe-Zeichenstrom **zeilenweise** eingelesen. Die zu implementierende Klasse `CommentFreeReader` stellt einen **von Kommentaren befreiten** `char`-**Zeichenstrom** bereit.

Der `CommentFreeReader` soll aktuell eingelesene Daten in einem bereits gegebenen `String`-Attribut `lookAheadString` speichern. Dadurch wird es möglich eine `peek`-Methode zu realisieren, die das nächste Zeichen des Zeichenstroms zurückliefert, aber noch nicht aus dem Strom entfernt. Ist der `lookAheadString` zu einem Zeitpunkt, in der **nicht** die `read`-Methode aktiv ist, leer, also `""`, so ist das Ende des Eingabe-Zeichenstroms erreicht. Ihnen sollte während der Implementierung von `read` klar werden warum dies der Fall ist.

H2.1: Lookahead 2 Punkte

In dieser Teilaufgabe werden Sie sich darum kümmern, dass der `lookAheadString` befüllt wird. Dazu implementieren Sie die `ürivate`-Methode `lookAhead`.

Zum Lesen einer ganzen Zeile des `BufferedReader`s nutzen Sie die Methode `readLine()`. Diese liefert entweder die nächste Zeile oder, falls das Ende des Eingabe-Stroms erreicht ist, `null` zurück. Falls das Ende der Eingabe erreicht ist, soll der Puffer auf dem leeren `String` gesetzt werden.

Wurde jedoch eine Zeile gelesen, dann soll aus dieser Zeile, falls vorhanden, der Kommentar entfernt werden. Ein Kommentar wird hierbei durch das Kommentar-Token `//` eingeleitet uns alle nachfolgenden Symbole ungleich `'\n'` gehören zum Kommentar. Der übrige Kommentar-freie Inhalt soll in den Puffer `lookAheadString` geschrieben werden. Dabei muss diesem Inhalt das Newline-Symbol `'\n'` am Zeilenende jeder gelesenen Zeile eingefügt werden.

Ein Sonderfall stellt ein Kommentar dar, der die ganze Zeile einnimmt, also kein Inhalt vor dem Kommentar steht. In diesem Fall soll die Zeile komplett übersprungen werden und die nächste Zeile eingelesen werden.

H2.2: Zeichenstrom nutzen 2 Punkte

In dieser Teilaufgabe werden Sie sich mit den Methoden befassen, die nach außen bereitgestellt werden, also `hasNext()`, `read()` und `peek()`.

Liefern Sie in der Methode `hasNext` genau dann `true` zurück, wenn der `lookAheadString` nicht leer ist.

Die `peek`-Methode liefert den `char` des Puffers zurück, der an Index 0 liegt, also das vorderste Zeichen.

Die Methode `read` liefert, genau wie `peek` auch, das vorderste Zeichen des `lookAheadString`s zurück, jedoch besitzt `read` im Gegensatz zu `peek` noch einen zustandsändernden Seiteneffekt. Konkret bedeutet das, dass die `read`-Methode das vorderste Symbol nicht nur zurückliefert, sondern ganz aus dem Puffer entfernt. Hierbei kann es dazu kommen, dass das letzte Zeichen des Puffers entfernt wird und der Puffer nach einem `read`-Aufruf leer wäre, obwohl noch Zeichen im Eingabe-Strom sind. Prüfen Sie daher nach dem `read`-Aufruf, ob der Puffer leer ist, und, falls ja, rufen sie `lookAhead` auf.

H3: *Wörter* erkennen — 4 Punkte

Durch die vorherige Teilaufgabe haben wir die Möglichkeit auf einen Zeichenstrom zuzugreifen, in dem die Kommentare entfernt sind. In dieser Teilaufgabe werden Sie aus diesen einzelnen Zeichen zusammenhängende *Wörter*, sogenannte *Token*, bilden. Die Klasse `Token` des Packages `h12.parse` ist bereits vorgegeben. Sie haben Ihre Arbeit ausschließlich in Klasse `Scanner` des Packages `h12.parse` zu verrichten.

Wir definieren ein Wort einfach als einen zusammenhängenden Zeichenstrom, der nicht durch ein Whitespace unterbrochen ist.

Der `Scanner` stellt die `public`-Methoden `scan()` und `hasNext()` bereit. Da eine `hasNext`-Methode realisiert werden soll, wird, wie in der vorherigen Teilaufgabe, ein Puffer benötigt wird, um `Token` bereits *vorher* einzulesen. Wie Sie diesen Puffer realisieren ist Ihnen in dieser Aufgabe freigestellt.

Die `hasNext`-Methode soll genau dann `true` zurückliefern, wenn es ein nächstes `Token` im Zeichenstrom gibt, andernfalls `false`.

Die `scan`-Methode soll das nächste `Token` zurückliefern und den Puffern neu befüllen. Sie können ein neues `Token` durch den Aufruf des Konstruktors erzeugen, der ein Argument des Typs `String` erwartet. Übergeben Sie dem `Token` hierbei den Inhalt des Wortes als `String`.

H4: Parser — 7 Punkte

In dieser Teilaufgabe wird es darum gehen, wie man aus einem `Token`-Strom die Struktur des Automaten enthält. Hierbei ist nun der genaue Aufbau einer Kiss2-Datei relevant. Sollten Sie diesen noch nicht verstanden haben, lesen Sie sich am Besten vorher nochmal die Einleitung durch. Implementieren werden Sie diese Aufgabe in der Klasse `FsmParser` in Package `h12.parse`.

In diesem Rahmen werden Ihnen nun zunächst `Token`s näher erläutert. In der vorherigen Teilaufgabe haben Sie `Token` erstellt, in dieser Aufgabe werden Sie sie verwenden. Ein `Token` kann hierbei jedes beliebige Wort sein, dass sich aus `char` bauen lässt. Jedoch gibt es verschiedene Typen von `Token`s die im Rahmen von kiss2 wichtig sind. Diese sind:

Type	Erklärung	Beispiel
`KEYWORD_INPUT_WIDTH`	Schlüsselwort, das den Headereintrag für die Eingangssignal-Breite einleitet	{.i}
`KEYWORD_OUTPUT_WIDTH`	Schlüsselwort, das den Headereintrag für die Ausgangssignal-Breite einleitet	{.o}
`KEYWORD_NUMBER_OF_TERMS`	Schlüsselwort, das den Headereintrag für die Anzahl der Terms einleitet	{.p}
`KEYWORD_NUMBER_OF_STATES`	Schlüsselwort, das den Headereintrag für die Anzahl an Zuständen einleitet	{.s}
`KEYWORD_INITIAL_STATE`	Schlüsselwort, das den Headereintrag für die Startzustand des Automaten einleitet	{.r}
`NUMBER`	Eine gültige natürliche Zahl	{0, 1, 2, ...}
`IDENTIFIER_STATE`	Ein gültiger Bezeichner für einen Zustand	{fop, s1, 22, 0101, ...}
`BITFIELD`	Gültige Eingangs- bzw Ausgangssignale	{- - -, - 1, 0 0, ...}

Diese Typen sind als Enumeration `Type` in der Klasse `Token` definiert. Ein `Token` stellt die bereits implementierte Methode `is(Type type)` bereit, welche genau dann `true` zurückliefert, wenn das `Token` ein gültiges `Token` für diesen `Type` ist.

Die `FsmParser`-Klasse besitzt ein Attribut `currentToken` des Typs `Token` und die Methoden `consumeToken()` und `consumeAndCheckToken()`. Diese sind bereits implementiert und nehmen Ihnen die Verwaltung des `Scanners` bzw. aktuellen `Tokens` ab. Die `consumeToken()`-Methode liest das nächste `Token` des `Scanners` ein und speichert es in `currentToken` ab. Die `consumeAndCheckToken(Token.Type type)`-Methode macht dasselbe, prüft jedoch vorher, ob der Typ des aktuellen `Tokens` mit `type` übereinstimmt.

Der Parser hat die Aufgabe die Struktur der Datei zu erkennen und relevante Informationen aus dieser Struktur zu lesen. Die eigentliche Konstruktion werden wir an das Interface `FsmBuilder` auslagern. Der `FsmParser` muss daher nicht wissen wie eine `Fsm` erzeugt wird, stattdessen werden jeweilige Methoden des `FsmBuilders` aufgerufen. Die Methoden `setInputSize(int inputSize)`, `setOutputSize(int outputSize)`, `setNumberOfTerms(int numberOfTerms)`, `setNumberOfStates(int numberOfStates)` und `setInitialState(String initialStateIdentifier)` sollen aufgerufen werden, wenn ein entsprechender Wert des Headers eingelesen wurde. Nachdem der Header abgeschlossen wurde, was dadurch erkenntlich ist, wenn das erste Mal ein Term erkannt wird, soll `finishHeader()` aufgerufen werden. Für jeden Term in der kiss2-Datei soll die Methode `addTerm(BitField inputField, String inputStateIdentifier, String nextStateIdentifier, BitField outputField)` des `FsmBuilder` aufgerufen werden. Die letzte Methode, `finishFSM()` soll genau dann aufgerufen werden, wenn das Ende der Datei erreicht ist.

H4.1: Header parsen — 4 Punkte

Implementieren Sie zunächst die Methoden `parseOutputWidth`, `parseNumberOfTerms`, `parseNumberOfStates` und `parseInitialState`, sodass die einzelnen Header-Parameter korrekt eingelesen werden können. Sie können sich hierbei an Methode `parseInputWidth` orientieren, welche bereits vorgegeben ist.

Nutzen Sie die vorgestellten Methoden `consumeToken` bzw. `consumeAndCheckToken`, sowie `is` von Klasse `Token`. Sie können die `String`-Repräsentation eines `Tokens` mit der Methode `getValue` erhalten.

Falls eine natürliche Zahl erwartet ist, diese aber nicht das `Token` ist, werfen Sie eine `BadNumberException` die in Package `h12.template.errors` definiert ist. Übergeben Sie dabei den aktuellen `Token` dem Konstruktor. Falls ein Zustandsname das nächste `Token` sein soll, aber nicht erkannt werden kann, werfen Sie analog eine `BadIdentifierException` aus selbigem Package.

Zuletzt implementieren Sie die Methode `parseHeader`. Diese Methode dient dazu, dass auf Basis des aktuellen Tokens entschieden wird um welchen Header-Parameter es sich handelt. Diese ist bereits für `Token` des Typs `KEYWORD_INPUT_WIDTH` vorgegeben. Ergänzen Sie die Methode nach diesem Muster um die restlichen vier Header-Parameter. Überlegen Sie sich auch, welche Methode des `FsmBuilders` Sie gegebenenfalls noch aufrufen müssen.

H4.2: Term parsen — 2 Punkte

Implementieren Sie zunächst die Methode `parseTerm`, die einen einzelnen Term, also einen Übergang des Automaten parst. Orientieren Sie sich hierfür an den `parse`*-Methoden des Headers, die Sie in der vorherigen Teilaufgabe implementiert haben. `BitField` stellt einen Konstruktor bereit, der ein Eingabe- bzw. Ausgabesignal einer kiss2-Datei als `String` erwartet. Rufen Sie `addTerm(BitField inputField, String inputStateIdentifier, String nextStateIdentifier, BitField outputField)` des `FsmBuilder` mit den richtigen Werten auf.

Anschließend müssen Sie noch Methode `parseTerms` implementieren. Diese Methode soll solange `parseTerm` aufrufen bis das Ende der Datei erreicht ist, also bis der `Scanner` kein nächstes `Token` mehr besitzt.

H4.3: Fsm parsen 1 Punkt

Zuletzt implementieren Sie noch die Methode `parseFSM`. Diese Methode soll das parsen einer ganzen kiss2-Datei ermöglichen, also Header und alle Terms. Überlegen Sie sich selbstständig wie dies zu realisieren ist.

H5: Builder 6 Punkte

In dieser Aufgabe werden Sie nun das bereits kennengelernte Interface `FsmBuilder` in der Klasse `FsmBuilderImpl` aus Package `h12.parse` implementieren.

Wie bereits in der vorherigen Aufgabe eingeleitet, sorgt der `FsmBuilder` dafür, dass aus den strukturieren Daten, die der `FsmParser` generiert hat, eine Instanz einer `Fsm` zu generieren. Die Klasse `FsmBuilderImpl` besitzt daher einerseits ein Attribut `fsm` des statischen Typs `Fsm` und `stateFactory` des Typs `StateFactory`. Das `Fsm`-Objekt wird direkt zu Beginn erzeugt und durch die verschiedenen `FsmBuilder`-Methoden mit den relevanten Informationen befüllt. Die `StateFactory` hat den Zweck die Umwandlung der Identifier der Zustände in Form von `String`s auf `State`s zu verwalten und zu puffern. Mehrfacher Aufruf der `get(String identifier)`-Methode mit gleichem Zustands-Identifier liefern dasselbe `State`-Objekt zurück. Zudem lässt sich die Anzahl der Zustände insgesamt durch den Aufruf von `getNumberOfStates()` bestimmen.

H5.1: Header verstehen 2 Punkte

In dieser Teilaufgabe sollen Sie die Methoden `setOutputSize(int outputSize)`, `setNumberOfTerms(int numberOfTerms)`, `setNumberOfStates(int numberOfStates)`, `setInitialState(String initialState)` implementieren. Dies sind die Methoden, die die Informationen des Headers verarbeiten. Die Methode `setInputSize(int inputSize)` ist bereits vorgegeben, damit Sie sich an dieser orientieren können.

Für jeden Header-Parameter ist ein Attribut `header*` vorgegeben, das genutzt werden soll, um den Parameterwert des Headers zu speichern. Wird ein Wert mehr als einmal geschrieben, dann soll eine `ParameterAlreadySpecifiedException` aus Package `h12.template.errors` mit dem jeweiligen Typ als `HeaderParameter` geworfen werden.

H5.2: Terms verarbeiten 2 Punkte

In dieser Aufgabe werden Sie die Methode

```
addTerm(BitField inputField, String inputStateIdentifier, String
            nextStateIdentifier, BitField outputField)
```

implementieren.

Diese Methode soll zunächst prüfen, ob die übergebenen `Bitfield`s `inputField` bzw. `oututField` in ihrer Breite mit den im Header definierten Werten für Eingangs- und Ausgangssignalbreite übereinstimmen. Die Klasse `Bitfield` stellt Ihnen dazu eine Methode `width()` bereit, die die Bitbreite zurückliefert. Ist das für ein `Bitfield` nicht der Fall, so soll eine `SizeMismatchException` aus Package `h12.template.errors` mit entsprechendem Typ als `HeaderParameter` geworfen werden.

Andernfalls, also falls beide `Bitfield`s in ihrer Größe korrekt sind, so soll die `Transition` zur `fsm` hinzugefügt werden. Nutzen Sie dazu die Methode `add(State state)` und `setTransition(Transition transition)` der Klasse `Fsm` bzw. `State`. Die Klasse `Transition` stellt einen Konstruktor bereit, der ein Eingabesymbol des Typs

`Bitfield`, ein Folgezustand des Typs `State` und ein Ausgangssymbol des Typs `Bitfield` erwartet, den Sie nutzen sollen, um Objekte der Klasse zu erzeugen. Vergessen Sie auch nicht die `StateFactory` zu nutzen.

Zählen Sie außerdem den Wert des Attributs `numberOfTermsCounter` um eins hoch. Dies werden Sie für die nächste Teilaufgabe benötigen.

H5.3: finishHeader und finishFsm — 1 Punkt

Die beiden `finish`*-Methoden dienen zur Prüfung der bisherigen Daten.

Die Methode `finishHeader` überprüft, ob alle notwendigen Header-Parameter gesetzt sind. Ist dies nicht der Fall, so soll eine `ParameterNotSpecifiedException` des Packages `h12.template.errors` mit entsprechendem Typ als `HeaderParameter` geworfen werden, um so zu signalisieren, dass die kiss2-Datei ein Fehler hat.

Die `finishFsm` besitzt auch eine prüfende Funktionalität. Und zwar soll überprüft werden, dass die Anzahl der Zustände und die Anzahl der Übergänge mit der jeweiligen Anzahl übereinstimmt, die im Header spezifiziert wurde. Ist das für einen Parameter nicht der Fall, so soll eine `SizeMismatchException` mit entsprechendem Parameter-Typen geworfen werden.

Kommt es in `finishFsm` zu keinem Verstoß, so ist die `Fsm` korrekt gebaut und gültig. In diesem Fall kann das `bool`-Attribut `buildFinished` auf `true` gesetzt werden. Wozu dieses Attribut benutzt wird, sehen Sie in der nächsten Teilaufgabe.

H5.4: getFsm — 1 Punkt

Solange die `Fsm` nicht korrekt fertiggestellt wurde, was sich durch die Fertigstellung von `finishFsm()` ohne geworfene Fehlermeldung ergibt, darf die `fsm` nicht zurückgegeben werden. Konkret also genau dann, wenn `buildFinished` gesetzt ist. Wird `getFsm()` aufgerufen, obwohl der Automat noch nicht fertig gebaut ist, so soll eine `KissParserException` mit einer von Ihnen sinnvoll angelegten Nachricht geworfen werden.

H6: Kiss2 Exporter 4 Punkte

Implementieren Sie die Methode `export(Fsm fsm)` der Klasse `KissExporter` des Packages `h12.export`. Diese Methode soll einen übergebenen Automaten `fsm` in eine gültige `kiss2`-Datei umwandeln und abspeichern. Dazu besitzt die Klasse ein Attribut `writer` des Typs `BufferedWriter` auf den schreibend zugegriffen werden kann, um in die Ausgabedatei zu schreiben.

Denken Sie daran, dass alle Werte des Headers vor dem Beginn der Terms in der Datei enthalten sein müssen. Sie können die Methode `getInitialState()` der Klasse `Fsm` nutzen, um sich den Startzustand des Automaten zurückliefern zu lassen. Ist kein Startzustand definiert, dann liefert die Methode `null` zurück. Der Header-Eintrag, der den Startzustand festlegt, soll nur enthalten sein, falls die `fsm` einen Startzustand definiert hat.

Zur Erinnerung an das Dateiformat ist in Codeblock H6 eine kiss2-Datei dargestellt.

Struktur einer kiss2-Datei

```
1  .i 2
2  .o 1
3  .p 5
4  .s 2
5  .r OFF
6  -1 OFF OFF 0
7  00 OFF OFF 0
8  10 OFF ON  1
9  -0 ON  ON  1
10 -1 ON  OFF 0
```

Alle Headereinträge

Der Body enthält je Zeile einen Übergang

Hinweis:

Die Klasse `Fsm` implementiert das Interface `Iterable<State>` und `State Iterable<Transition>`. Es ist Ihnen freigestellt wie Sie die nötigen Informationen aus der `Fsm` erhalten, jedoch stellen die Iteratoren einen guten Einstiegspunkt dar.

Hinweis:

Sie können sich die Klasse `DotExporter` anschauen, um Inspirationen für einen `FsmExporter` zu sehen.

H7: System Verilog Exporter 5 Punkte

In dieser Teilaufgabe wird es darum gehen den `SystemVerilogExporter` aus Package `h12.export` zu implementieren, welcher einen eingelesenen Automaten in eine äquivalente Verilog-Repräsentation umwandelt. Sie müssen sich keine Sorgen machen, es sind keine Verilog-Kenntnisse nötig, um diese Aufgabe zu erfüllen.

Die folgenden Teilaufgabe implementieren Sie, falls nicht anders angegeben, in der Methode `export(Fsm fsm)` in chronologischer Reihenfolge.

H7.1: 1 Punkt

Header der Verilog-Datei Die erste Sache, die in Verilog relevant ist, ist eine korrekte Definition des Moduls. Dies besteht aus einem Header zur Definition der Schnittstelle und einem Footer.

Den Header können Sie mit einem Aufruf von `generateModuleHeader(int inputBitWidth, int outputBitWith)` generieren. Der erste Parameter gibt hierbei die Breite der Eingangssymbole und der zweite Parameter die der Ausgangssymbol an. Greifen Sie geeignet auf die übergebene `Fsm fsm` zu, um die relevanten Informationen zu erhalten.

Anschließend rufen Sie `generateModuleFooter()` auf, die den Footer des Verilog-Moduls generiert.

Alle folgenden Teilaufgaben werden Sie nun *zwischen* beiden, in dieser Teilaufgabe realisierten, Aufrufen bearbeiten. Damit ist sichergestellt, dass die Modulheader und -footer den Inhalt des Moduls einschließen.

H7.2: 1 Punkt

Definitionen realisieren Anschließend sind verschiedene Definitionen nötig. Zum Beispiel zum Speichern des aktuellen Zustandes.

Es gibt hierzu die Methode `generateVariable(String variableName, int variableBitWidth)`. Der erste Parameter gibt den Namen der Variable an und der zweite Parameter die Breite dieser in Bit.

Es gibt drei Variablen, die Sie generieren müssen. Das sind:

- `state`: Die Breite, mit der die Zustände kodiert werden. Hierzu stellt das Attribut `stateEncoding` eine Methode `getWidth()` bereit, die genutzt werden soll.
- `nextState`: Analog zu state
- `nextOut`: Die Breite des Ausgangssymbols

Wie Sie an die nötigen Bit-Breiten kommen ist Ihnen überlassen. Gehen Sie jedoch wie in der vorherigen Teilaufgabe vor.

H7.3: **1 Punkt**

Zustandsübergang In dieser Teilaufgabe wird der sogenannte *posedge*-Block realisiert. Dies geschieht mit Methode `generatePosedgeBlock(State initialState)`. Der übergebene `State` ist der Startzustand des Automaten. Diesen erhalten Sie, indem Sie eine Instanz des Automaten erzeugen. Dies geht durch den Aufruf `createInstance()` für `fsm`. Rufen Sie nach der Erzeugung der `FsmInstance` die Methode `getCurrentState()` auf, um den aktuellen Zustand zurückzugeben. Da dies direkt nach der Erzeugung der Instanz aufgerufen wurde, entspricht der zurückgegeben Zustand dem Startzustand des Automaten.

H7.4: **2 Punkte**

Alle Bedingungen kodieren In dieser Teilaufgabe implementieren Sie zuletzt, dass alle Übergänge korrekt nach Verilog abgebildet werden.

Zunächst rufen Sie `generateConditionsHeader(int inputBitWidth)`, wobei der Parameter die Bit-Breite der Eingangssymbole ist (erinnern Sie sich an die vorherigen Teilaufgaben zurück), gefolgt vom Aufruf der Methode `generateConditionsFooter()`. Dies stellt wieder den Rahmen dar. Daher implementieren Sie den Rest wieder `zwischen` diesen Aufrufen.

Sie sollen nun für jeden Übergang des Automaten die Methode `generateCondition(State startState, BitField event, State endState, BitField output)` aufrufen. Die Methode generateCondition benötigt vier Parameter: den Ausgangszustand, das Eingangssymbol, den Zielzustand und das Ausgangssymbol. Wie in den vorherigen Aufgaben auch müssen Sie sich selbst überlegen wie Sie an die nötigen Infos kommen.

Zuletzt sollen Sie nun noch die `generateCondition(...)` Methode selbst implementieren. In Codeblock H7.4 finden Sie einen Übergang, den Sie ausgeben sollen.

Beispiel einer Condition

```
1 {4'b??10,10'b0000000001} : begin nextState = 10'b0000000001; nextOut = 2'b00;
  ↪  end
```

Die Formatierung an sich und die Bezeichner, also `nextState` und `nextOut`, nutzen Sie so wie angegeben. Die vier Bitfelder müssen Sie aus den übergebenen Argumenten bestimmen. Bei Bitfeldern ist relevant, dass diese ausgegeben werden als: die Breite in Bit gefolgt von `'b` und anschließend den Wert des `BitField`. `BitField` stellt die Methode `toString()` und `toString(chat dcSymbol)` bereit, die ein `BitField` in binärer Darstellung als `String` umwandelt.

- `4'b??10`: Dies ist das übergebene `event`
- `10'b0000000001`: Dies ist der `startState`, der durch einen Aufruf von `stateEncoding.encode(startState)` in ein `BitField` mit entsprechender Kodierung umgewandelt werden kann
- `10'b0000000001`: Dies ist der `endState`, auch wie `startState` mittels des `stateEncoding` umgewandelt.
- `2'b00`: Ist das Feld `output`

Zur Implementierung können Sie sich an allen anderen `generate`*-Methoden der Klasse orientieren.

H8: Testen (unbewertet)

Damit Sie Ihre Implementierung testen können, wurde ein weiterer `FsmExporter` implementiert, der `DotExporter`. Dieser konvertiert einen Automaten in eine gültige DOT-Datei. DOT ist eine Beschreibungssprache zur Darstellung von Graphen. Wir nutzen außerdem ein Online Viewer für DOT-Dateien, der unter `https://dreampuf.github.io/GraphvizOnline` gefunden werden kann.

Sie können einfach die `main`-Methode der Klasse `Main` aus Package `h12` ausführen. Zunächst wird der Automat entsprechend der, in dieser Hausübung realisierten, Pipeline geparsed und anschließend in einen Graphen als DOT-Datei exportiert. Die `main`-Methode gibt einen URL aus, auf den Sie klicken können, der Sie auf die oben abgegebene Seite leitet. Anschließend dauert es einen kleinen Moment und Sie können auf der rechten Seite den gerenderten Graphen sehen. Haben Sie alles richtig gemacht, sollten Sie ein zu den in der Einleitung gezeigten Graphen äquivalentes Bild sehen können.

Möchten Sie noch weiter mit Ihrem Parser experimentieren, können Sie Beispiele für kiss2-Dateien von `https://automata.cs.ru.nl/BenchmarkCircuits/Kiss` herunterladen.