Hermes Architecture and User Guide

GengoAI

Version v1.0, February 26, 2020: Initial Release

Table of Contents

1. Overview

Hermes is a Natural Language Processing framework for Java inspired by the Tipster Architecture and licensed under the Apache License, Version 2.0 making it free for all uses. The goal of Hermes is to ease the development and use of NLP technologies by providing easy access to and construction of linguistic annotations on documents using multiple cores or multiple machines (using Apache Spark). Hermes is designed to aid users in analyzing large textual data sources making it easy define and perform complex workflows to extract, analyze, aggregate, and distill information and knowledge. Conceptually, text in Hermes is represented as a HString (short for Hermes String) which is a *CharSequence* that provides access to the:

- Overlapping or enclosed annotations.
- Attributes defining aspects of the text.
- Relations to other HStrings in the same document.
- Character offsets within the document
- Spatial relations (e.g. overlaps, encloses) with other HStrings.

HStrings can be easily manipulated in a fluent manner or by using Lyre a robust extraction and transformation language that facilitates extraction, filtering, feature generation, counting, and transformation into other objects. Additionally, Hermes provides extraction based on:

- Lexicons
- Token-based regular expressions
- Machine Learning
- Trigger-based matching via the Caduceus relation-extraction relation.

Throughout Hermes data can be stored and processed in-memory, on-disk, or distributed. This combination facilitates working with corpora of all sizes. Additionally, Hermes is tightly integrated with GengoAI's Apollo machine learning framework allowing easy training of ml models, including word embeddings (e.g. Glove and Word2Vec), topics (Latent Dirichlet Allocation), and supervised classification of attributes (e.g. sentiment, part-of-speech).

2. Installation

Hermes requires Java 11+ and is available via the maven central repository at:

```
<dependency>
     <groupId>com.gengoai</groupId>
     <artifactId>hermes</artifactId>
     <version>1.0</version>
</dependency>
```

Additionally, you can download a Hermes distribution, which provides easy access to a number of command line and gui applications for processing, annotating, and performing analytics over documents and corpora. We provide two distributions:

Local Mode - No Spark	http://download.com
Run Hermes local to one computer or bring your own Spark cluster.	
Local Mode - No Spark	
Run Hermes local to one computer or with Spark in Standalone mode. Also have the option of running on your own Spark Cluster.	

As part of these distributions there are a series of scripts to aid in running Hermes applications (listed in section Hermes Applications) and for running within a distributed Spark environment.

Hermes stores its data in a resources directory defined in configuration via `hermes.resources.dir`. By default this will be set to the `hermes` directory under the user's home directory, e.g. `/home/user/hermes/`.

3. Core Classes

The core classes in Hermes consist of *AnnotatableType*, *AttributeType*, *AnnotationType*, *RelationType*, *HString*, *Annotation*, *Document*, *Relation*, *Attribute*, and *Corpus*. How the core clases are composed and inherit from one another is depicted in the following diagram.

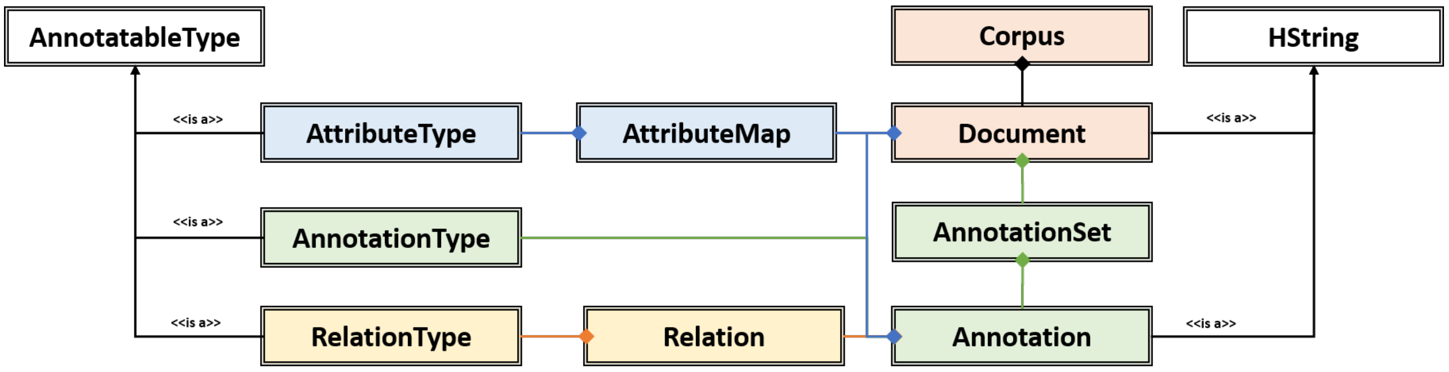

Figure 1. Diagram of Herme's Core Class

3.1. AnnotatableType

An annotatable type is a type added to documents through the act of annotation. Annotation can be performed on a corpus of documents or a single document. Hermes supports the following Annotatable Types:

3.1.1. AttributeType

An AttributeType defines a **named** Attribute that can be added to an HString. Each AttributeType has an associated value type which defines the class of value that the attribute accepts and is specified using Java Generics as follows:

```
AttributeType<String> AUTHOR = AttributeType.make("AUTHOR", String.class);
AttributeType<Set<BasicCategories>> CATEGORIES = AttributeType.make("CATEGORIES", parameterizedType(
Set.class,BasicCategories.class))
```

Annotating for AttributeType adds the attribute and value to an annotation or document. For example, when annotating for the AttributeType PART_OF_SPEECH, each token annotation has a POS value set for its PART_OF_SPEECH attribute of. Many AnnotationType will include attributes when being annotated, e.g. token annotations provide TOKEN_TYPE and CATEGORY attributes.

3.1.2. AnnotationType

An AnnotationType defines an Annotation, which is a **typed** (e.g. token, sentence, phrase chunk) span of text on a document having a defined set of attributes and relations. AnnotationTypes are hierarchical meaning that each type has a parent (*ANNOTATION* by default) and can have subtypes. Additionally, each AnnotationType has an associated Tag attribute type, which represents the central attribute of the annotation type (e.g. entity type for entities and part-of-speech for tokens.). By default, an annotation's tag type is inherited from the parent or defined as being a StringTag. The following code snippet illustrates creating a simple AnnotationType with the default parent and a and an AnnotationType whose parent is *ENTITY*.

```
/* Assume that SENSE_TAG is a predefined AttributeType */
AnnotationType WORD_SENSE = AnnotationType.make("WORD_SENSE", SENSE_TAG);
/* MY_ENTITY will be a type of ENTITY and have an ENTITY_TYPE tag attribute inherited from ENTITY  */
AnnotationType MY_ENTITY = AnnotationType.make(ENTITY, "MY_ENTITY");
```

3.1.3. RelationType

A RelationType defines the type of arbitrary link, i.e. relation, between two HStrings. Relation types can define such things as co-reference and syntactic and semantic structure. Defining a RelationType is performed as follows:

```
RelationType AGENT = RelationType.make("AGENT");
```

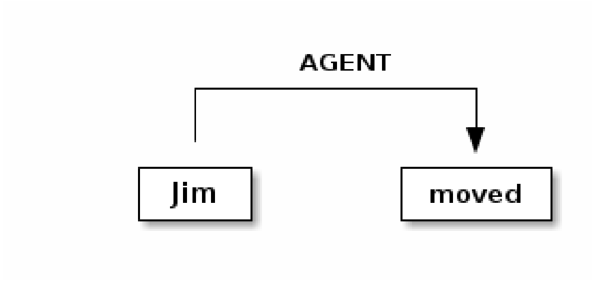

Figure 2. Pictorial example of annotating an AGENT relation from Jim to moved.

Annotating for RelationType adds a Relation object on the source and target annotation as an *outgoing* and *incoming* annotation respectively. For example, if we had a token *Jim* as the agent of the verb *moved*, and annotated for the *AGENT* RelationType we would add an outgoing *AGENT* relation on *JIM* with *moved* as the target and an incoming *AGENT* relation on *moved* with *JIM* as the source. Pictorial this would be represented as follows:

3.1.4. Annotators

Annotator(s) satisfy, i.e. provide, one or more AnnotatableType (AnnotationType, AttributeType, or RelationType) by processing a document and adding a new or modifying an existing annotation. In order to provide the new AnnotatableType an annotator may require one or more AnnotatableType to be present on the document. For example, an annotator providing the PHRASE_CHUNK AnnotationType would require the presence of the TOKEN AnnotationType and PART_OF_SPEECH AttributeType. When annotation is complete, the AnnotatableType is marked as complete on the document and an annotator provided version is associated with the type.

Sentence Level Annotators: Sentence level annotators work on individual sentences. They have a minimum requirement of SENTENCE and TOKEN AnnotationType. Additional types can be specified by overriding the `furtherRequires` method. Sentence level annotators are a convenience when creating annotators that work over or within single sentences.

Sub Type Annotators: In certain cases, such as Named Entity Recognition, there may exist a number of different methodologies which we want to combine to satisfy a parent AnnotationType. In these situations a SubTypeAnnotator can be used. A SubTypeAnnotator satisfies an AnnotationType by calling multiple other annotators that satisfy one or more of its sub types. For example, the EntityAnnotator provides the ENTITY AnnotationType, by using sub annotators which could be a combination of rule-based and machine learning-based methods.

Annotator Configuration: Annotators are not normally created and used directly, but instead are automatically constructed and used when making a call to the `annotate` methods either on a document or corpus. AnnotatableType define the annotator that should be constructed as follows:

1. Check if a configuration setting is defined for the type

- TYPE.LANGUAGE.LABEL.annotator
- TYPE.LABEL.annotator
- TYPE.LABEL.annotator.LANGUAGE

where *TYPE* is one of `Annotation`, `Attribute`, `Relation`, *LANGUAGE* is the language of the document either in its full form, or ISO code, and *LABEL* is the label (name) of the type. Examples of each are as follows:

- Annotation.ENGLISH.ENTITY.annotator = com.mycompany.EntityAnnotator
- Annotation.ENTITY.annotator = com.mycompany.EntityAnnotator
- Annotation.ENTITY.ENGLISH.annotator = com.mycompany.EntityAnnotator

2. Check for default implementations

- com.gengoai.hermes.LANGUAGE_CODE[LowerCase].LANGUAGE_CODE[UpperCase] + LABEL[CamelCase] + "Annotator"
- com.gengoai.hermes.annotator."Default" + LANGUAGE_NAME[CamelCase] + LABEL[CamelCase] + "Annotator"
- com.gengoai.hermes.annotator."Default" + LABEL[CamelCase] + "Annotator"

where *LANGUAGE_CODE* is the ISO code of the document language, *LANGUAGE_NAME* is the name of the document's language, and *LABEL* is the label (name) of the type. Examples of each are as follows:

- com.gengoai.hermes.en.ENEntityAnnotator
- com.gengoai.hermes.annotator.DefaultEnglishDependencyAnnotator
- com.gengoai.hermes.annotator.DefaultDependencyAnnotator

An example configuration defining a *REGEX_ENTITY* AnnotationType is as follows:

```
Annotation {
     REGEX_ENTITY {
          ENGLISH = @{ENGLISH_ENTITY_REGEX} ①
          JAPANESE = @{JAPANESE_ENTITY_REGEX} ②
     }
}
```

① Points to a Java Bean named *ENGLISH_ENTITY_REGEX* defined in configuration.

② Points to a Java Bean named *JAPANESE_ENTITY_REGEX* defined in configuration.

For more information on **Configuration** in Hermes, see the Configuration Section of the Mango User Document.

3.2. HString

An HString (Hermes String) is a Java String on steroids. It represents the base type of all Hermes text objects. Every HString has an associated span denoting its starting and ending character offset within the document. HStrings implement the CharSequence interface allowing them to be used in many of Java's builtin String methods and they have similar methods as found on Java Strings. Importantly, methods not modifying the underlying string, e.g. substring and find, return an HString whereas methods that modify the string, e.g. toLowerCase, return a String object. The String-Like operations are as follows:

Type	Method	Description
`char`	`charAt(int)`	Returns the character at the given index in the HString.
`boolean`	`contains(CharSequence)`	Returns true if the given CharSequence is a substring of the HString.
`boolean`	`contentEquals(CharSequence)`	Returns true if the given CharSequence is equal to the string form of the HString.
`boolean`	`contentEqualsIgnoreCase(CharSequence)`	Returns true if the given CharSequence is equal to the string form of the HString regardless of case.
`boolean`	`endsWith(CharSequence)`	Returns true if the HString ends with the given CharSequence.
`Language`	`getLanguage()`	Gets the Language that the HString is written in.
`int`	`length()`	The length in characters of the HString
`HString`	`find(String)`	Finds the given text in this HString starting from the beginning of this HString. If the document is annotated with tokens, the match will extend to the token(s) covering the match.
`HString`	`find(String, int)`	Finds the given text in this HString starting from the given start index of this HString. If the document is annotated with tokens, the match will extend to the token(s) covering the match.
`Stream<HString>`	`findAll(String)`	Finds all occurrences of the given text in this HString.
`Matcher`	`matcher(String | Pattern)`	Returns a Java regular expression over the HString for the given pattern.
`boolean`	`matches(String)`	Returns true if the HString matches the given regular expression.

Type	Method	Description
`String`	`replace(CharSequence, CharSequence)`	Replaces all substrings of this HString that matches the given string with the given replacement.
`String`	`replaceAll(CharSequence, CharSequence)`	Replaces all substrings of this HString that matches the given regular expression with the given replacement.
`String`	`replaceFirst(CharSequence, CharSequence)`	Replaces the first substring of this HString that matches the given regular expression with the given replacement.
`HString`	`context(AnnotationType, int)`	Generates an HString representing the given window size of annotations of the given type on both the left and right side without going past sentence boundaries.
`HString`	`context(int)`	Generates an HString representing the given window size of tokens on both the left and right side without going past sentence boundaries.
`HString`	`rightContext(AnnotationType, int)`	Generates an HString representing the given window size of annotations of the given type to the right of the end of this HString without going past the sentence end.
`HString`	`rightContext(int)`	Generates an HString representing the given window size of tokens to the right of the end of this HString without going past the sentence end.
`HString`	`leftContext(AnnotationType, int)`	Generates an HString representing the given window size of annotations of the given type to the left of the end of this HString without going past the sentence start.
`HString`	`leftContext(int)`	Generates an HString representing the given window size of tokens to the left of the start of this HString without going past the sentence start.
`List<HString>`	`split(Predicate<? super Annotation>)`	Splits this HString using the given predicate to apply against tokens.
`boolean`	`startsWith(CharSequence)`	Returns true if the HString starts with the given CharSequence.
`HString`	`subString(int, int)`	Returns a new HString that is a substring of this one.
`char[]`	`toCharArray()`	Returns a character array representation of this HString.

Type	Method	Description
`String`	`toLowerCase()`	Returns a lowercased version of this HString.
`String`	`toUpperCase()`	Returns an uppercased version of this HString.
`HString`	`trim(Predicate<? super HString>)`	Trims the left and right ends of the HString removing tokens matching the given predicate.
`HString`	`trimLeft(Predicate<? super HString>)`	Trims the left end of the HString removing tokens matching the given predicate.
`HString`	`trimRight(Predicate<? super HString>)`	Trims the right end of the HString removing tokens matching the given predicate.
`HString`	`union(HString)`	Constructs a new HString that has the shortest contiguous span that combines all of the tokens in this HString and the given HString.
`HString`	`union(HString, HString, HString…)`	Static method that constructs a new HString that has the shortest contiguous span that combines all of the tokens in all given HStrings.
`HString`	`union(Iterable<? extends HString>)`	Static method that constructs a new HString that has the shortest contiguous span that combines all of the tokens in all given HStrings.
`List<HString>`	`charNGrams(int)`	Extracts character n-grams of the given order from the HString
`List<HString>`	`charNGrams(int,int)`	Extracts character n-grams ranging from the given minimum to given maximum order from the HString

HStrings store attributes using an **AttributeMap** which maps **AttributeType** to values. HStrings provide Map-like access to their attributes through the following methods:

Type	Method	Description
`T`	`attribute(AttributeType<T>)`	Gets the value of the given attribute associated with the HString or null if the attribute is not present.
`T`	`attribute(AttributeType<T>, T)`	Gets the value of the given attribute associated with the HString or the given default value if the attribute is not present.
`boolean`	`attributeEquals(AttributeType<T>, Object)`	Returns **true** if the attribute is present on the HString and its value is equal to given value.

Type	Method	Description
`boolean`	`attributeIsA(AttributeType<T>, Object)`	Returns **true** if the attribute is present on the HString and its value is equal to given value or is an instance of the given value if the AttributeType's value is an instance of **Tag**.
`boolean`	`hasAttribute(AttributeType<T>)`	Returns **true** if the HString has a value for the given AttributeType.
`void`	`removeAttribute(AttributeType<T>)`	Removes any associated value for the given AttributeType from the HString.
`T`	`put(AttributeType<T>, T)`	Sets the value of the given AttributeType returning the old value or null if there was not one.
`void`	`putAdd(AttributeType<T>, Iterable<E>)`	Adds the given values to the given attribute which represents a Collection of values.
`void`	`putAll(HString)`	Copies the attributes and values from the given HString
`void`	`putAll(Map<AttributeType<?>,?)`	Copies all attributes and values from the given Map
`T`	`putIfAbsent(AttributeType<T>, T)`	Sets the value of the given attribute to the given value if the HString does not already have a value for the attribute.
`T`	`computeIfAbsent(AttributeType<T>, Supplier<T>)`	Sets the value of the given attribute to the given value if the HString does not already have a value for the attribute.
`POS`	`pos()`	Returns the `PART_OF_SPEECH` attribute for the HString or calculates the best part-of-speech if the attribute is not present.

Look at the **GettingStarted.java** and **HStringIntroduction.java** in the examples project for more information on handling Attributes.

The power of HStrings is fast access to the Annotation that they overlap and/or enclose. The following methods define the basic annotation API:

Type	Method	Description
`List<Annotation>`	`annotations()`	Gets all annotations overlapping with this HString.

Type	Method	Description
`List<Annotation>`	`annotations(AnnotationType)`	Gets all annotations of the given type overlapping with this HString.
`List<Annotation>`	`annotations(AnnotationType, Predicate<? super Annotation>)`	Gets all annotations of the given type overlapping with this HString that evaluate to true using the given Predicate.
`Stream<Annotation>`	`annotationStream()`	Gets a java Stream over all annotations overlapping this HString.
`Stream<Annotation>`	`annotationStream(AnnotationType)`	Gets a java Stream over all annotations of the given type overlapping this HString.
`Annotation`	`asAnnotation()`	Casts this HString as Annotation if it already is one otherwise creates a dummy annotation.
`Annotation`	`asAnnotation(AnnotationType)`	Casts this HString as Annotation as the given type if it is an instance of that type otherwise creates a dummy annotation.
`List<Annotation>`	`enclosedAnnotations()`	Gets all annotations enclosed by this HString
`List<Annotation>`	`enclosedAnnotations(AnnotationTYpe)`	Gets all annotations of the given type enclosed by this HString
`Annotation`	`first(AnnotationType)`	Gets the first annotation of the given type overlapping with this HString or an empty Annotation if there is none.
`Annotation`	`firstToken()`	Gets the first token overlapping with this HString or an empty Annotation if there is none.
`void`	`forEach(AnnotationType, Consumer<? super Annotation>)`	Convenience method for processing annotations of a given type.
`boolean`	`hasAnnotation(AnnotationType)`	Returns **true** if an annotation of the given type overlaps with this HString.

Type	Method	Description
`List<Annotation>`	`interleaved(AnnotationType…)`	Returns the annotations of the given types that overlap this string in a maximum match fashion. Each token in the string is examined and the annotation type with the longest span on that token is chosen. If more than one type has the span length, the first one found will be chosen, i.e. the order in which the types are passed in to the method can effect the outcome.
`boolean`	`isInstance(AnnotationType)`	Returns **true** if this HString is an instance of the given AnnotationType.
`Annotation`	`last(AnnotationType)`	Gets the last annotation of the given type overlapping with this HString or an empty Annotation if there is none.
`Annotation`	`lastToken()`	Gets the last token overlapping with this HString or an empty Annotation if there is none.
`Annotation`	`next(AnnotationType)`	Gets the annotation of a given type that is next in order (of span) to this HString.
`Annotation`	`sentence()`	Gets the first sentence overlapping with this HString or an empty Annotation if there is none.
`List<Annotation>`	`sentences()`	Gets all sentences overlapping with this HString.
`Stream<Annotation>`	`sentenceStream()`	Gets all sentences overlapping with this HString as a Java stream.
`List<Annotation>`	`startingHere(AnnotationType)`	Gets all annotations of the given type with the starting character offset as this HString.
`Annotation`	`tokenAt(int)`	Gets the token at the given index relative to the HString (i.e. 0 for the first token, 1 for the second token, etc).
`List<Annotation>`	`tokens()`	Gets all tokens overlapping with this HString.
`Stream<Annotation>`	`tokenStream()`	Gets all tokens overlapping with this HString as a Java stream.

Look at the **GettingStarted.java** and **CustomAnnotator.java** in the examples project for more information on handling Annotations.

Finally, HStrings provide access to the incoming and outgoing Relation directly annotated on them and in their overlapping annotations.

Type	Method	Description
`void`	`add(Relation)`	Adds an outgoing relation to the object
`void`	`addAll(Iterable<Relation>)`	Adds multiple outgoing relations to the object.
`RelationGraph`	`annotationGraph(Tuple, AnnotationType…)`	Constructs a relation graph with the given relation types as the edges and the given annotation types as the vertices.
`List<Annotation>`	`children()`	Gets all child annotations, i.e. those annotations that have a dependency relation pointing this HString.
`List<Annotation>`	`children(String)`	Gets all child annotations, i.e. those annotations that have a dependency relation pointing this HString, with the given dependency relation.
`Tuple2<String,Annotation>`	`dependency()`	Get dependency relation for this annotation made up the relation and its parent.
`RelationGraph`	`dependencyGraph()`	Creates a RelationGraph with dependency edges and token vertices.
`RelationGraph`	`dependencyGraph(AnnotationType…)`	Creates a RelationGraph with dependency edges and vertices made up of the given types.
`boolean`	`dependencyIsA(String…)`	Returns **true** if the dependency relation equals any of the given relations
`boolean`	`hasIncomingRelation(RelationType)`	Returns **true** if an incoming relation of a given type is associated with the HString (includes sub-annotations)
`boolean`	`hasIncomingRelation(RelationType, String)`	Returns **true** if an incoming relation of a given type with the given value is associated with the HString (includes sub-annotations)

Type	Method	Description
`boolean`	`hasOutgoingRelation(RelationType)`	Returns **true** if an outgoing relation of a given type is associated with the HString (includes sub-annotations)
`boolean`	`hasOutgoingRelation(RelationType, String)`	Returns **true** if an outgoing relation of a given type with the given value is associated with the HString (includes sub-annotations)
`HString`	`head()`	Gets the token that is highest in the dependency tree for this HString
`List<Annotation>`	`incoming(RelationType)`	Gets all annotations that have relation with this HString as the target where this HString includes all sub-annotations.
`List<Annotation>`	`incoming(RelationType, boolean)`	Gets all annotations that have relation with this HString as the target, including sub-annotations if the given boolean value is **true**.
`List<Annotation>`	`incoming(RelationType, String)`	Gets all annotations that have relation with this HString as the target where this HString includes all sub-annotations.
`List<Annotation>`	`incoming(RelationType, String, boolean)`	Gets all annotations that have relation with this HString as the target where this HString, including sub-annotations if the given boolean value is **true**.
`List<Relation>`	`incomingRelations()`	Gets all incoming relations to this HString including sub-annotations.
`List<Relation>`	`incomingRelations(boolean)`	Gets all incoming relations to this HString including sub-annotations if the given boolean is **true**
`List<Relation>`	`incoming(RelationType)`	Gets all relations of the given type targeting this HString or one of its sub-annotations.
`List<Relation>`	`incoming(RelationType, boolean)`	Gets all relations of the given type targeting this HString or one of its sub-annotations if the given boolean is **true**.

Type	Method	Description
`List<Annotation>`	`outgoing(RelationType)`	Gets all annotations that have relation with this HString as the source where this HString includes all sub-annotations.
`List<Annotation>`	`outgoing(RelationType, boolean)`	Gets all annotations that have relation with this HString as the source, including sub-annotations if the given boolean value is **true**.
`List<Annotation>`	`outgoing(RelationType, String)`	Gets all annotations that have relation with this HString as the source where this HString includes all sub-annotations.
`List<Annotation>`	`outgoing(RelationType, String, boolean)`	Gets all annotations that have relation with this HString as the source where this HString, including sub-annotations if the given boolean value is **true**.
`List<Relation>`	`outgoingRelations()`	Gets all outgoing relations to this HString including sub-annotations.
`List<Relation>`	`outgoingRelations(boolean)`	Gets all outgoing relations to this HString including sub-annotations if the given boolean is **true**
`List<Relation>`	`outgoing(RelationType)`	Gets all relations of the given type originating from this HString or one of its sub-annotations.
`List<Relation>`	`outgoing(RelationType, boolean)`	Gets all relations of the given type originating from this HString or one of its sub-annotations if the given boolean is **true**
`Annotation`	`parent()`	Gets the dependency parent of this HString
`void`	`removeRelation(Relation)`	Removes the given Relation.

Look at the **DependencyParseExample.java** and **SparkSVOExample.java** in the examples project for more information on handling Relations.

3.3. Annotation

An annotation is an HString that associates an AnnotationType, e.g. token, sentence, named entity, to a specific span of characters in a document, which may include the entire document. Annotations typically have attributes, e.g. part-of-speech, entity type, etc, and relations, e.g. dependency and co-reference, associated with them. Annotations are assigned a *long* id when attached to a document, which uniquely identifies it within that document. Annotations provide the following extra methods to the standard set of HString methods:

Type	Method	Description
`long`	`getId()`	Gets the unique long id assigned to the Annotation when attached to a document.
`AnnotationType`	`getType()`	Returns the AnnotationType associated with this Annotation
`Tag`	`getTag()`	Returns the Tag value associated with this annotation (see the Tags section more information on Tags)
`boolean`	`hasTag()`	Returns **true** if the annotation has a value associated with its Tag attribute.
`boolean`	`tagEquals(Object)`	Returns **true** if the annotation has a tag value and the tag value is equal to the given tag (Note that the method parameter will be decoded into a Tag)
`boolean`	`tagIsA(Object)`	Returns **true** if the annotation has a tag value and the tag value is an instance of to the given tag (Note that the method parameter will be decoded into a Tag)
`void`	`attach()`	Attaches, i.e. adds, the annotation to its document.

3.3.1. Creating Annotations

The primary way of creating an annotation is through an *AnnotationBuilder* on a Document. An AnnotationBuilder provides the following methods for constructing an annotation:

Type	Method	Description
`AnnotationBuilder`	`attribute(AttributeType, Object)`	Sets the value of the given AttributeType on the new Annotation to the given value.
`AnnotationBuilder`	`attributes(Map<AttributeType<?>,?>)`	Copies the AttributeTypes and values from the map into the new annotation.

Type	Method	Description
`AnnotationBuilder`	`attributes(HString)`	Copies the AttributeTypes and values from the given HString into the new annotation.
`AnnotationBuilder`	`bounds(Span)`	Sets the bounds (start and end character offset) of the annotation to that of the given span.
`AnnotationBuilder`	`start(int)`	Sets the start character offset of the annotation in the document.
`AnnotationBuilder`	`end(int)`	Sets the end character offset of the annotation in the document.
`AnnotationBuilder`	`from(HString)`	Conveinince method for calling `bounds(HString)`, `attributes(HString)`, and `relations(HString)`.
`AnnotationBuilder`	`relation(Relation)`	Adds the given relation to the new Annotation as an outgoing relation.
`AnnotationBuilder`	`relation(Iterable<Relation>)`	Adds all of the given relation to the new Annotation as an outgoing relations.
`Annotation`	`createAttached()`	Creates and attaches the annotation to the document.
`Annotation`	`createDetached()`	Creates the annotation but does not attach it to the document.

As an example of creating Annotations, let's assume we want to add ENTITY annotations to all occurrences of GengoAI in a document. We can do this as follows:

```
Document doc = ...;

int startAt = 0;
HString mention;
while( !(mention=doc.find("GengoAI", startAt)).isEmpty() ){ ①
     doc.annotationBuilder(Types.ENTITY) ②
        .bounds(mention)
        .attribute(Types.ENTITY_TYPE, Entities.ORGANIZATION) ③
        .createAttached();
     startAt = mention.end(); ④
}
```

① Continue while we have found a mention of "GengoAI" from the *startAt* position.

② We will create an AnnotationBuilder with type ENTITY and assume the bounds of the mention match.

③ Set the ENTITY_TYPE attribute to the value ORGANIZATION.

④ Increment the next start index.

The difference between an attached and detached annotation is attached annotations (1) have an assigned id, (2) are accessible through the HString annotation methods, and (3) can be the target of relations. Detached annotations are meant to be used as intermediatory or temporary annotations often constructed by an Annotator which uses a global document context to filter or combine annotations.

3.3.2. Tags

Every AnnotationType has an associated Tag attribute type. The Tag defines the central attribute of the annotation type. For example, Hermes defines the PART_OF_SPEECH tag to be the central attribute of tokens and the ENTITY_TYPE tag as the central attribute of entities. An annotation's Tag attribute can be accessed through the `getTag()` method on the annotation or through the `attribute(AttributeType<?>)` method, note that an annotation's tag is assigned to the specific AttributeType (e.g. PART_OF_SPEECH) but is also accessible through the TAG AttributeType.

Tags have the following properties:

name	The name of the tag, e.g. PART_OF_SPEECH. For tags which are hierarchical the name is the full path without the root, e.g. ORGANIZATION$POLITICAL_ORGANIZATION$GOVERNMENT.
label	The label of the tag, which for hierarchal tags is the leaf level name, i.e. for ORGANIZATION$POLITICAL_ORGANIZATION$GOVERNMENT the label would be GOVERNMENT.
parent	The parent tag of this one, where *null* means the tag is a root. Note all non-hierarchical tags have a null parent.

Names and labels must be unique within in a tag set, i.e. an entity type tag set can only contain one tag with the label *QUANTITY* meaning you are not allowed to define a *MEASUREMENT$QUANTITY* and *NUMBER$QUANTITY*.

3.3.3. Core Annotations

Hermes provides a number of annotation types out-of-the-box and the ability to create custom annotation types easily from lexicons and existing training data. Here, we discuss the core set of annotation types that Hermes provides.

`TOKEN`	Tokens represent, typically, the lowest level of annotation on a document. Hermes equates a token to mean a word (this is not always the case in other libraries depending on the language). A majority of the attribute and relation annotators are designed to enhance (i.e. add attributes and relations) to tokens. For example, the part-of-speech annotator adds part-of-speech information to tokens and the dependency annotator provides dependency relations between tokens.

SENTENCE	Sentences represent a set of words typically comprised of a subject and a predict. Sentences have an associated INDEX attribute that denote the index of the sentence in the document.
PHRASE_CHUNK	Phrase chunks represent the output of a shallow parse (sometimes also referred to as a light parse). A chunk is associated with a part-of-speech, e.g noun, verb, adjective, or preposition.
ENTITY	The entity annotation type serves as a parent for various named entity recognizers. Entities are associated with an EntityType, which is a hierarchy defining the types of entities (e.g. a entity type of MONEY has the parent NUMBER).

Take a look at **CustomAnnotator.java**, **LexiconExample.java**, and **GettingStarted.java** in the Hermes examples project to see examples of using annotations and creating custom annotation types.

3.4. Relation

Relations provide a mechanism to link two Annotations. Relations are directional, i.e. they have a source and a target, and form a directed graph between annotations on the document. Relations can represent any type of link, but often represent syntactic (e.g. dependency relations), semantic (e.g. semantic roles), or pragmatic (e.g. dialog acts) information. Relations, like attributes, are stored as key value pairs with the key being the RelationType and the value being a String representing the label. Relations are associated with individual annotations (i.e. tokens for dependency relations, entities for co-reference). Methods on HString allow for checking for and retrieving relations for *sub-annotations* (i.e. ones which it overlaps with), which allows for analysis at different levels, such as dependency relations between phrase chunks.

3.4.1. Dependency Relations

Dependency relations are the most common relation and connect and label pairs of words where one word represents the head and the other the dependent. The assigned relations are syntactic, e.g. *nn* for noun-noun, *nsubj* for noun subject of a predicate, and *advmod* for adverbial modifier, and the relation points from the dependent (source) to the head (target). Because of their wide use, Hermes provides convenience methods for working dependency relations. Namely, the `parent` and `children` methods on HString provide access to the dependents and heads of a specific token and the `dependencyRelation` method provides access to the head (parent) of the token and the relation between it and its head.

3.4.2. Relation Graphs

In some cases it is easier to work with annotations and relations as a real graph. For these cases, Hermes provides the `dependencyGraph` and `annotationGraph` methods on HString. These methods construct a Mango Graph![1] with which you can render to an image, perform various clustering algorithms, find paths between annotations, and score the annotations using methods such as PageRank.

3.5. Document

A Document is represented as a text (HString) and its associated attributes (metadata), annotations, and relations between annotations. Every document has an id associated with it, which should be unique within a corpus. Documents provide the following additional methods on top of the ones inherited from HString:

Type	Method	Description
`void`	`annotate(AnnotatableType…)`	Annotates the document for the given types ensuring that all required AnnotatableTypes are also annotated.
`Annotation`	`annotation(long)`	Retrieve an Annotation by its unique id.
`void`	`attach(Annotation)`	Attaches the given annotation to the document assigning it a unique annotation id.
`Set<AnnotatableType>`	`completed()`	Returns the set of AnnotatableType that have been annotated or marked as being annotated on this document.
`String`	`getAnnotationProvider(AnnotatableType)`	Returns the name and version of the annotator that provided the given AnnotatableType.
`boolean`	`isCompleted(AnnotatableType)`	Returns **true** if the given AnnotatableType has been annotated or marked as being annotated on this document.
`int`	`numberOfAnnotations()`	Returns the number of Annotation on the document.
`boolean`	`remove(Annotation)`	Removes the given annotation returning **true** if it was successfully removed.
`void`	`removeAnnotationType(AnnotationType)`	Removes all annotations of the given type and marks that type as incomplete.
`void`	`setCompleted(AnnotatableType,String)`	Sets the given AnnotatableType as being complete with the given provider.
`Document`	`fromJson(String)`	Static method to deserialize a Json string into a Document.
`String`	`toJson()`	Serializes the document into Json format.

3.5.1. Creating Documents

Documents are created using a DocumentFactory, which defines the preprocessing (e.g whitespace and unicode normalization) steps (TextNormalizers) to be performed on raw text before creating a document and the default language with which the documents are written. The default DocumentFactory has its default language and TextNormalizers specified via configuration as follows:

```
hermes {

  ## Set default language to English
  DefaultLanguage = ENGLISH

  #By default the document factory will normalize unicode and white space
  preprocessing {
    normalizers = hermes.preprocessing.UnicodeNormalizer
    normalizers += "hermes.preprocessing.WhitespaceNormalizer"
    normalizers += "hermes.preprocessing.HtmlEntityNormalizer"
  }

}
```

The default set of TextNormalizers includes:

1. A UnicodeNormalizer which normalizes Strings using NFKC normalization (Compatibility decomposition, followed by canonical composition).
2. A WhitespaceNormalizer which collapses multiple whitespace and converts newlines to linux (\n) format.
3. A HtmlEntityNormalizer which converts named and hex html entities to characters.

The following snippet illustrates creating a document using the default DocumentFactory.

```
Document document = DocumentFactory.getInstance().create("...My Text Goes Here...");
```

For convenience a document can also be created using static methods on the document class, which will use the default DocumentFactory as follows:

```
Document d1 = Document.create("...My Text Goes Here..."); ①
Document d2 = Document.create("my-unique-id", "...My Text Goes Here..."); ②
Document d3 = Document.create("Este es un documento escrito en español.", Language.SPANISH); ③
Document d4 = Document.create("...My Text Goes Here...", ④
                              Maps.of($(Types.SOURCE, "The document source"),
                                      $(Types.AUTHOR, "A really important person")));
```

① Creation of a document specifying only the content.

② Creation of a document specifying its unique id and its content.

③ Creation of a document specifying the language the document is written in.

④ Creation of a document specifying a set of attributes associated with it.

DocumentFactories provide additional methods for constructing documents from pre-tokenized text (`fromTokens`) and to force the factory to ignore the string preprocessing (`createRaw`).

3.5.2. Working with Documents

Annotations are spans of text on the document which have their own associated set of attributes and relations. Annotations are added to a document using a AnnotationPipeline. The pipeline defines the type of annotations, attributes, and relations that will be added to the document. However, Document and Corpora provide a convenience method `annotate(AnnotatableType…)` that takes care of constructing the pipeline and calling its annotation method. The following snippet illustrates annotating a document for TOKEN, SENTENCE, and PART_OF_SPEECH:

```
Document d1 = Document.create("...My Text Goes Here...");
d1.annotate(Types.TOKEN, TYPES.SENTENCE, TYPES.PART_OF_SPEECH) ①
```

① The **Types** class contains a number of pre-defined AnnotatableType

Ad-hoc annotations are easily added using one of the `createAnnotation` methods on the document. The first step is to define your AnnotationType:

```
AnnotationType animalMention = Types.type("ANIMAL_MENTION");
```

Now, let’s identify animal mentions using a simple regular expression. Since Document extends HString we have time saving methods for dealing with the textual content. Namely, we can easily get a Java regex Matcher for the content of the document by:

```
Matcher matcher = document.matcher("\\b(fox|dog)\\b");
```

With the matcher, we can iterate over the matches and create new annotations as follows:

```
while (matcher.find()) {
   document.createAnnotation(animalMention,
                             matcher.start(),
                             matcher.end());
}
```

More complicated annotation types would also provide attributes, for example entity type, word sense, etc. Once annotations have been added to a document they can be retrieved using the `annotations(AnnotationType)` method.

```
document.get(animalMention)
        .forEach(a -> System.out.println(a + "[" + a.start() + ", " + a.end() + "]"));
```

In addition, convenience methods exist for retrieving tokens, `tokens()`, and sentences, `sentences()`.

```
document.sentences().forEach(System.out::println);
```

A document stores its associated annotations using an AnnotationSet. The default implementation uses an interval tree backed by a red-black tree, which provides O(n) storage and average O(log n) for search, insert, and delete operations.

3.6. Document Collections and Corpora

A collection of documents in Hermes is represented using either a *DocumentCollection* or *Corpus*. The difference between the two is that a *Corpus* represents a **persistent** collection of documents whereas a *DocumentCollection* is a **temporary** collection used for ad-hoc analytics or to import documents into a corpus. The figure below, shows a typical flow of data in which: (1) A document is collection is created by reading files in a given format (e.g. plain text, html, pdf, etc.); (2) The files are imported into a Corpus for processing; (3) Operations, e.g. annotation, are performed over the corpus which allows these operations to be persisted; and (4) Optionaly, the documents in the corpus in are exported to a set of files in a given format (e.g. CoNLL).

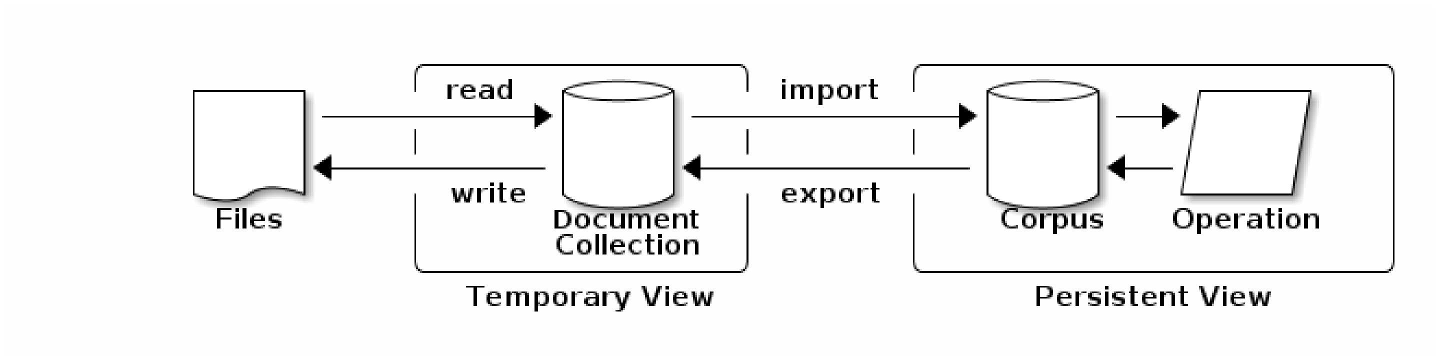

Figure 3. Typical flow of documents from Raw input to Corpus creation.

Hermes provides the ability to easily create, read, write, and analyze document collections and corpora locally and distributed. Both makes it easy to annotate documents with a desired set of annotations, attributes, and relations, query the documents using keywords, and perform analyses such as term extraction, keyword extraction, and significant n-gram extraction.

3.6.1. Document Formats

Hermes provides a straightforward way of reading and writing documents in a number of formats, including plain text, csv, and json. In addition, many formats can be used in a "one-per-line" corpus where each line represents a single document in the given format. For example, a json one-per-line corpus has a single json object representing a document on each line of the file. Each document format has an associated set of *DocFormatParameters* that define the various options for reading and writing in the format. By default the following parameters can be set:

`defaultLanguage`	The default language for new documents. (default calls `Hermes.defaultLanguage()`)
`normalizers`	The class names of the text normalizes to use when constructing documents. (default calls `TextNormalization.configuredInstance().getPreprocessors()`)
`distributed`	Creates a distributed document collection when the value is set to **true** (default `false`).
`saveMode`	Whether to *overwrite*, *ignore*, or *throw an error* when writing a corpus to an existing file/directory (default `ERROR`).

The following table lists the included document formats with their added format parameters and read/write capabilities:

Table 1. Document formats included with Hermes

Format Name	Read	Write	Support OPL	Description
`TEXT`	✓	✓	✓	Plain text documents.
• Standard Document Format Parameters Only				
`PTB`	✓			Penn Treebank bracketed (.mrg) files
• Standard Document Format Parameters Only				
`HJSON`	✓	✓	✓	Hermes Json format.
• Standard Document Format Parameters Only				
`CONLL`	✓	✓		CONLL format.

Format Name	Read	Write	Support OPL	Description

- `docPerSentence=[true|false]`: One document per sentence when **true** (default: **true**).
- `fields=<list of fields>`: list of string denoting the field names (default: ["WORD", "POS", "CHUNK"]]).
- `fs=<String>`: Field separator (default: "\\s+")
- `overrideSentences=[true|false]`: Override the CONLL sentence boundaries with Hermes boundaries when **true** (default: **false**)

The following fields are supported:

- INDEX - The index of the word in the sentence.
- WORD - The word.
- LEMMA - The lemmatized form of the word.
- UPOS - The universal part-of-speech tag of the word.
- POS - The part-of-speech tag of the word.
- CHUNK - IOB annotated Phrase Chunks.
- ENTITY - IOB annotated Named Entities.
- HEAD - The index of this word's syntactic head in the sentence.
- DEP_REL - The dependency relation of this word to its head.
- IGNORE - Ignores the field.

Format Name	Read	Write	Support OPL	Description
`CSV`	✓	✓		Delimited separated files (e.g. CSV and TSV) with each row representing a document.

- `columns=<list of column names>`: The list of column names when file does not have a header (default: empty).
- `content=<String>`: Name of the content column (default: "content").
- `id=<String>`: Name of the document id column (default: "id").
- `language=<String>`: Name of the language column (default: "language").
- `comment=<Character>`: The character used for comments in the file (default: '#').
- `delimiter=<Character>`: The character used for delimiting columns in the file (default: ',').
- `hasHeader=[true|false]`: The file has a header naming the columns when **true** (default: false).

Note that columns name will be autogenerated as C0, C1, …, CN when no column names are given and there is no header in the file. Additional columns in the file not assigned to "id", "language", or "content" will be treated as document level attributes.

Format Name	Read	Write	Support OPL	Description
`TWITTER_SEARCH`	✓			Twitter API Search result
• Standard Document Format Parameters Only				
`POS`	✓	✓	✓	Format with words separated by whitespace and POS tags appended with an underscore, e.g. The_DT brown_JJ.
• Standard Document Format Parameters Only				
`TAGGED`	✓	✓	✓	Format with words separated by whitespace and sequences labeled in SGML like tags, e.g. <TAG>My text</TAG>.
• `annotationType=<String>`: The annotation type that sequences are an instance of (default: ENTITY).				

The **Format Name** is used to identify the document format to read and to use the format with one-per-line, you can append "_opl" to the format name.

3.6.2. Document Collection Creation

The *DocumentCollection* class provides the following methods to create a document collection from a series of documents:

Type	Method	Description
`DocumentCollection`	`create(String)`	creates a document collection from documents stored in the format and at the location specified by the given specification.
`DocumentCollection`	`create(Specification)`	creates a document collection from documents stored in the format and at the location specified by the given specification.
`DocumentCollection`	`create(Document…)`	Creates a document collection in memory containing the given documents.
`DocumentCollection`	`create(List<Document>)`	Creates a document collection in memory containing the given documents.
`DocumentCollection`	`create(MStream<Document>)`	Creates a document collection from the given Mango stream where the corpus will be distributed if the given Mango stream is also distributed and a streaming corpus otherwise.

Type	Method	Description
`DocumentCollection`	`create(Stream<Document>)`	Creates a stream-based corpus containing the given documents.

The following is an example of creating a document collection from Twitter data:

```
DocumentCollection twitter = DocumentCollection.create("twitter_search::/data/twitter_search_results/"
);
```

A more complex example is creation from CSV files:

```
DocumentCollection csv = DocumentCollection.create(
"csv::/data/my_csv.csv;columns=id,content,language,author,source");
```

3.6.3. Working with Document Collections and Corpora

The Hermes *Corpus* and *DocumentCollection* class provides a variety of different methods for accessing, analyzing, and manipulating its documents.

Accessing Documents

Document collections and corpora allow for the following access to their collection of documents:

Type	Method	Description
`Iterator<Document>`	`iterator()`	Gets an iterator over the documents in the corpus.
`MStream<Document>`	`stream()`	Returns a Mango stream over the documents in the corpus.
`MStream<Document>`	`parallelStream()`	Returns a parallel Mango stream over the documents in the corpus.

In addition to the methods above, corpora allow for access to individual documents using `get(String)` method where the string parameter is the document id.

Manipulating the Corpus and its Documents

The main method for manipulation of a collection is through the `update(SerializableConsumer<Document>)` method, which processes each document using the given consumer. For document collections this method acts as a map whereas for corpora the update persists to the underlying storage.

Corpora also allow for individual documents to be udpated via the `update(Document)` method.

Additionally, documents can be added and removed from corpora using the following set of methods:

Type	Method	Description
`void`	`add(Document)`	Adds a document to the corpus.
`void`	`addAll(Iterable<Document>)`	Adds the given documents to the corpus.
`void`	`importDocuments(String)`	Imports documents from the given document collection specification.
`boolean`	`remove(Document)`	Returns **true** if the given Document was successfully removed from the corpus.
`boolean`	`remove(String)`	Returns **true** if the document with the given document id was successfully removed from the corpus.

Querying

Hermes provides a simple boolean query language to query documents. The query syntax is as follows:

Operator	Description
`AND`	Requires the queries, phrases, or words on the left and right of the operator to both be present in the document. (AND is case insensitive)
`OR`	Requires for one of the queries, phrases, or words on the left and right of the operator to be present in the document. (OR is case insensitive)
`-`	Requires the query, phrase, or word on its right hand side to **not** be in the document.
`$ATTRIBUTE='VALUE'`	Requires the value of the document attribute describe after the $ to equal the value in the parenthesis.
`'PHRASE'`	Searches for the phrase defined between the single quotation marks. (note if the phrase includes a single quote it can be escaped using the backslash character.)
`WORD`	Searches for the word (note the word cannot start or end with parenthesis and cannot have whitespace)

Multiword phrases are expressed using quotes, e.g. `'United States'` would match the entire phrase whereas `United AND States` only requires the two words to present in the document in any order. The default operator when one is not specified is `OR`, i.e. `United States` would be expanded to `United OR States`.

```
Corpus corpus = ...;
SearchResults results = corpus.query("'United States' AND 'holiday'");
System.out.println("Query: " + results.getQuery());
System.out.println("Total Hits: " + results.size());
for( Document document : results ){
    System.out.println(document.getTitle());
}
```

As shown in the code snippet above, querying a corpus results in a *SearchResults* which retains the query that generated results and a document collection view of the results.

Frequency Analysis

A common step when analyzing a corpus is to calculate the term and document frequencies of the words in its documents. In Hermes, the frequency of any type of annotation can be calculated across a corpus using the `termCount(Extractor)` method. The analysis is defined using an *Extractor* object, which provides a fluent interface for defining annotation type, conversion to string form, filters, and how to calculate the term values (see Text Mining for more information on Extractors). An example is as follows:

```
Corpus corpus = ...;
Extractor spec = TermExtractor.builder() ①
                              .toLemma()
                              .ignoreStopwords()
                              .valueCalculator(ValueCalculator.L1_NORM);
Counter<String> tf = corpus.termCount(spec); ②
```

① Shows creation of the *TermExtractor* which defines the way we will extract terms. Here we specify that we want lemmas, will ignore stopwords, and want the returning counter to have its values L1 normalized.

② Shows the calculating of term frequencies over the entier corpus.

By default, the TermExtractor will specify TOKEN annotations which will be converted to a string form using the toString method, all tokens will be kept, and the raw frequency will be calculated.

In a similar manner, document frequencies can be extracted using the `documentCount(Extractor)` method. An example is as follows:

```
Corpus corpus =...;
Extractor spec = TermExtractor.builder()
                              .toLemma()
                              .ignoreStopwords();
Counter<String> tf = corpus.documentCount(spec);
```

Both the *termCount* and *documentCount* methods take an *Extractor*, which can include any type of extraction technique (discussed in Text Mining).

Extracting N-Grams

While n-grams can be extracted using the `termCount` and `documentCount` feature, Hermes provides the `nGramCount(NGramExtractor)` method for calculating document-based counts of n-grams where the n-gram is represented as *Tuple* of string. An example of gathering bigram counts from a corpus is as follows:

```
Corpus corpus = ...;
NGramExtractor extractor = NGramExtractor.bigrams() ①
                                         .toLemma()
                                         .ignoreStopWords()
                                         .valueCalculator(ValueCalculator.L1_NORM);
Counter<Tuple> tf = corpus.nGramCount(extractor); ②
```

① Shows creation of the n-gram extractor which defines the way we will extract n-grams. Here we specify that we want to extract unigrams, bigrams, and trigrams and that will convert to lemma form, ignore stopwords, and want the returning counter to have its values L1 normalized.

② Shows the calculating of n-gram frequencies over the entier corpus.

By default, the NGramExtractor will specify TOKEN annotations which will be converted to a string form using the toString method, all tokens will be kept, and the raw frequency will be calculated.

In addition, Hermes makes it easy to mine "significant bigrams" from a corpus using the `significantBigrams(NGramExtractor, int, double)` and `significantBigrams(NGramExtractor, int, double, ContingencyTableCalculator)` methods. Both methods take an `NGramExtractor` to define how the terms should be extracted (note that the min and max order is ignored), a (int) minimum count required to consider a bigram, and a (double) minimum score for a bigram to be considered significant. Additionally, a *ContingencyTableCalculator* can be given which is used to calculate the score of a bigram (by default `Association.Mikolov` is used which is the calculation used within word2vec to determine phrases). Both methods return a *Counter<Tuple>* containing the bigrams and their score. The following example illustrates finding significant bigrams using Normalized Pointwise Mutual Information (NPMI):

```
Corpus corpus = ...;
NGramExtractor extractor = NGramExtractor.bigrams()
                                         .toLemma()
                                         .ignoreStopWords()
                                         .valueCalculator(ValueCalculator.L1_NORM);
Counter<Tuple> bigrams = corpus.significantBigrams(extractor, 5, 0, Association.NPMI); ①
```

① Extract significant bigrams which have a minimum count of 5 and a minimum NPMI of 0.

Sampling

Often times we only want to use a small portion of a corpus to test for analysis in order to test it out. The corpus class provides a means for performing reservoir sampling on the corpus using the following two methods:

```
sample(int size)
sample(int size, Random random)
```

Both return a new corpus and take the sample size as the first parameter. The second method takes an additional parameter of type *Random* which is used to determine inclusion of a document in the sample. Note that for non-distributed corpora the sample size must be able to fit into memory.

Grouping

The Corpora class provides a `groupBy(SerializableFunction<? super Document, K>)` method for grouping documents by an arbitrary key. The method returns a *Multimap<K, Document>* where *K* is the key type and takes a function that maps a *Document* to *K*. The following code example shows where this may of help.

```
Corpus corpus = ...;
corpus.groupBy(doc -> doc.getAttributeAsString(Types.SOURCE)); ①
```

① Group documents by their source.

Note that because this method returns a Multimap, the entire corpus must be able to fit in memory.

4. Text Mining

The goal of Text Mining is to turn unstructured data into high-quality structured information. Hermes provides a variety of tools to perform text mining over corpora, some of which were described in the Document Collections and Corpora section. Fundamental to text mining in Hermes is the concept of a *Extractor* and the *Extraction* it produces. Extractors are responsible for taking an *HString* as input and producing an *Extraction* as output via the `Extraction extract(@NonNull HString hString)` method. The class hierarchy for Extractors is as follows (note names in Yellow represent abstract classes or interfaces):

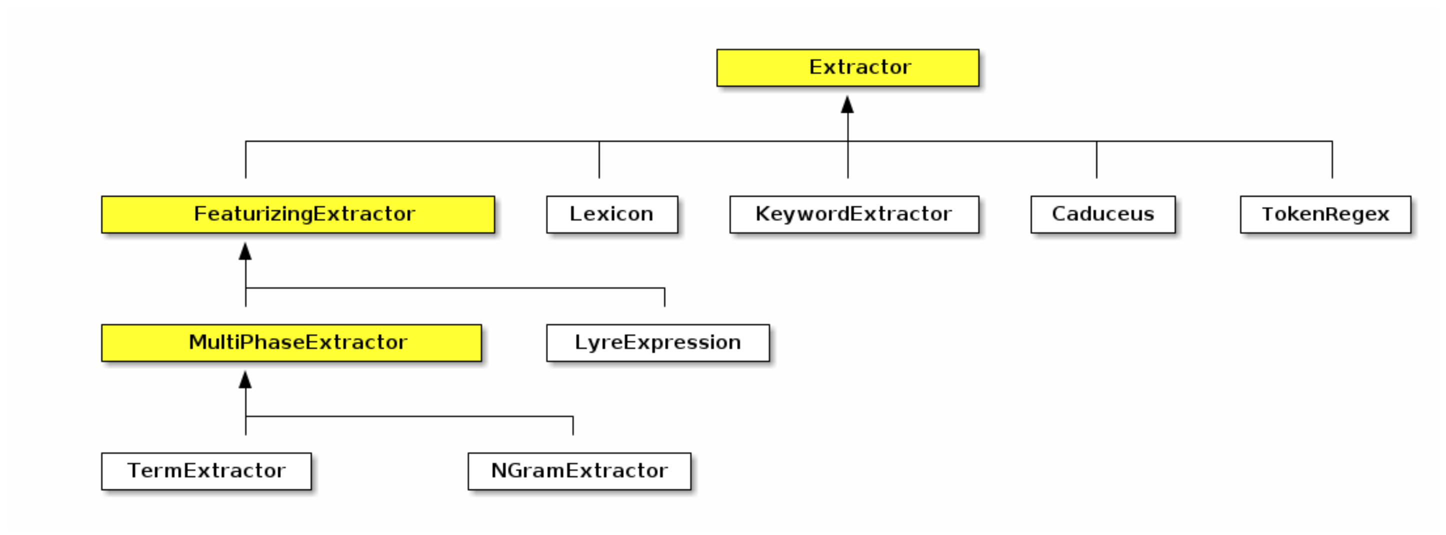

Figure 4. Inheritance hierarchy for extractors.

The *Lexicon* extractor uses a lexicon to match terms in an *HString* and described in detail in Lexicons. The *KeywordExtractor* extracts key phrases from an HString based on a defined algorithm and described in detail in Keyword Extraction. The *FeaturizingExtractor* combines an extractor with a *Featurizer* allowing for the output of the extractor to be directly used as features for machine learning.

The *LyreExpression* extractor is based on Hermes's Lyre Expression Language. The *MultiPhaseExtractor* is the base for *TermExtractor* and *NGramExtractor* which we looked at in the <<#fa> section on corpora. Multi-phase extractors define a series of steps to transforming an HString into an Extraction, which include the annotation types to extract, filters to apply on the extracted annotations, methodology for trimming the extracted annotations, methodology for converting the annotations into Strings, and a prefix for when the extraction is used as a machine learning feature.

Every extractor produces an *Extraction*. Extractions can provide their results as an *Iterable* of *HString* or *String* or a *Counter*<*String*> via the following methods:

Type	Method	Description
`int`	`size()`	The number of items extracted.
`Iterable<String>`	`string()`	Returns the extracted items as an Iterable of String.
`Counter<String>`	`count()`	Returns the extracted items as a Counter of String.
`Iterator<HString>`	`iterator()`	Returns an Iterator of the extracted HString (Note that if the extractor does not support HString it will generate a fragment).

Note that how the results are constructed are dependent on the extraction technique. For example, some extractions only provide fragments (i.e. non-attached) *HString* due to the way extraction is performed.

4.1. Lexicons

A traditional approach to information extraction incorporates the use of lexicons, also called gazetteers, for finding specific lexical items in text. Hermes's *Lexicon* classes provide the ability to match lexical items using a greedy longest match first or maximum span probability strategy. Both matching strategies allow for case-sensitive or case-insensitive matching and the use of constraints (using the Lyre expression language), such as part-of-speech, on the match.

Lexicons are managed using the *LexiconManager*, which acts as a cache associating lexicons with a name and a language. This allows for lexicons to be defined via configuration and then to be loaded and retrieved by their name (this is particularly useful for annotators that use lexicons).

Lexicons are defined using a *LexiconSpecification* in the following format:

```
lexicon:(mem|disk):name(:(csv|json))*::RESOURCE(;ARG=VALUE)*
```

The schema of the specification is "lexicon" and the currently supported protocols are: * mem: An in-memory Trie-based lexicon. * disk: A persistent on-disk based lexicon.

The name of the lexicon is used during annotation to mark the provider. Additionally, a format (csv or json) can be specified, with json being the default if none is provided, to specify the lexicon format when creating in-memory lexicons. Finally, a number of query parameters (ARG=VALUE) can be given from the following choices:

- `caseSensitive=(true|false)`: Is the lexicon case-sensitive (**true**) or case-insensitive (**false**) (default **false**).
- `defaultTag=TAG`: The default tag value for entry when one is not defined (default null).
- `language=LANGUAGE`: The default language of entries in the lexicon (default `Hermes.defaultLanguage()`)

CSV lexicons allow for the additionaly following parameters:

- `lemma=INDEX`: The index in the csv row containing the lemma (default 0).
- `tag=INDEX`: The index in the csv row containing the tag (default 1).
- `probability=INDEX`: The index in the csv row containing the probability (default 2).
- `constraint=INDEX`: The index in the csv row containing the constraint (default 3).
- `language=LANGUAGE`: The default language of entries in the lexicon (default `Hermes.defaultLanguage()`)

As an example, we can define the following lexicons in our configuration:

```
person.lexicon =  lexicon:mem:person:json::<hermes.resources.dir:ENGLISH>person.lexicon ①
huge.lexicon  =  lexicon:disk:everything:<hermes.resources.dir:ENGLISH>huge.lexicon ②
csv.lexicon   = lexicon:mem:adhoc:csv::/data/test/import.csv;probability=-
1;constraint=2;caseSensitive=true;tagAttribute=ENTITY_TYPE;defaultTag=PERSON ③
```

① Defines an in-memory lexicon stored in json format named "person".

② Defines a disk-based lexicon named "everything".

③ Defines an in-memory lexicon stored in csv format named "adhoc" that is case-sensitive, has a tag attribute of ENTITY_TYPE with a default tag of PERSON, does not use probabilities, and the constraint is stored in the second (0-based) column.

Note that we can use <hermes.resources.dir:ENGLISH> to specify that file is located in the ENGLISH directory of the Hermes resources, which is defined in the config option `hermes.resources.dir`. The language name can be omitted when the lexicon is in the default resources.

We can retrieve a lexicon from the *LexiconManager* as follows:

```
Lexicon lexicon = LexiconManager.getLexicon("person.lexicon"); ①
Lexicon undefined = LexiconManager.getLexicon("undefined.lexicon"); ②
```

① Retrieve the person lexicon we defined previously in our config file.

② Attempt to retrieve a lexicon that has not been defined via configuration. In this case, it will try to find a json formatted lexicon with the named "undefined.lexicon.json" in one of the resource directories Hermes knows about.

The lexicon manager allows for lexicons to be manually registered using the register method, but please note that this registration will not carry over to each node in a distributed environment.

Take a look at **LexiconExample.java** in the Hermes examples project to see examples of constructing and using lexicons.

4.1.1. Reading and Writing Lexicons

The *LexiconIO* class provides static methods for reading and writing in-memory lexicons. The primary format of a Hermes lexicon is Json and is described as follows:

```
{
  "@spec": { ①
    "caseSensitive": false,
    "tagAttribute": "ENTITY_TYPE",
    "language": "ENGLISH
  },
  "@entries": [ ②
    {
      "lemma": "grandfather",
      "tag": "GRANDPARENT"
    },
    {
      "lemma": "mason",
      "tag": "OCCUPATION",
      "probability": 0.7,
      "constraint": "!#NNP"
    },
    {
      "lemma": "housewife",
      "tag": "OCCUPATION"
    }
  ]
}
```

① The "@spec" section defines the specification of the lexicon.

② The "@entries" section is where the lexical entries are specified.

As seen in the snippet the json file starts with a specification section, "@spec", in which the valid parameters are:

- **caseSensitive**: Is the lexicon case-sensitive (**true**) or case-insensitive (**false**) (default **false**).
- **language**: The language of the entries in the lexicon (default `Hermes.defaultLanguage()`).
- **tag**: The default tag value for entry when one is not defined (default null).

The "@entries" section defines the individual lexicon entries in the lexicon with the following valid parameters:

- **lemma**: The lexical item to be matched (no default must be set).
- **tag**: The tag value associated with the lemma that the *tagAttribute* will be set to (default lexicon default tag).
- **probability**: The probability of the lexical item associated with its tag (default 1.0).
- **constraint**: The constraint (using a Lyre expression) that must be satisfied for the lexical match to take place (default null).
- **tokenLength"** Optional parameter the defines the number of tokens in the entry (default calculated based on the lexicon language).

Additionally, csv based lexicons can be imported using `LexiconIO.importCSV(Resource, Consumer<CSVParameters>)` where the Resource defines the location of the CSV file and the Consumer is used to specify the lexicon parameters. The CSVParameters defines the columns for lemmas, tags, probabilities, and constraints as well as the standard lexicon information of case-sensitive or insensitive matching, tag attribute, and default tag.

4.1.2. Word Lists

Word lists provide a set like interface to set of vocabulary items. Implementations of *WordList* may implement the *PrefixSearchable* interface allowing prefix matching. Word lists are loaded from plain text files with "#" at the beginning of a line denoting a comment. Whereas lexicons provide a robust way to match and label HStrings, _WordList_s provide a simple means of determining if a word/phrase is defined. Note that convention states that if the first line of a word list is a comment stating "case-insensitive" then loading of that word list will result in all words being lower-cased.

4.2. Lyre Expression Language

Lyre (Linguistic querY and extRaction languagE) provides a means for querying, extracting, and transforming HStrings. A *LyreExpression* represents a series of steps to perform over an input HString which can be used for querying (i.e. acting as a Java *Predicate*) and extracting and transforming (i.e. like a Java *Function*) using the following methods:

Type	Method	Description
`String`	`apply(HString)`	Applies the expression returning a String value.
`double`	`applyAsDouble(HString)`	Applies the expression returning a double value or *NaN* if the return value is not convertible into a double.
`double`	`applyAsDouble(Object)`	Applies the expression returning a double value or *NaN* if the return value is not convertible into a double.
`List<Feature>`	`applyAsFeatures(HString)`	Applies the expression returning a list of *Feature* for machine learning.
`HString`	`applyAsHString(HString)`	Applies the expression returning it is an HString using `HString.toHstring(Object)`.
`List<Object>`	`applyAsList(Object)`	Applies the expression returning it is a list of Object.
`List<T>`	`applyAsList(Object, Class<T>)`	Applies the expression returning it is a list of type T.
`Object`	`applyAsObject(Object)`	Applies the expression.

Type	Method	Description
`String`	`applyAsString(Object)`	Applies the expression returning it as a String value.
`Counter<String>`	`count(HString)`	Applies the expression returning a count over the string results.
`boolean`	`test(HString)`	Returns **true** if the expression evaluates to true.
`boolean`	`testObject(HString)`	Returns **true** if the expression evaluates to true.

A LyreExpression can be created by parsing a string representation using `Lyre.parse(String)` or by using the `LyreDSL` class to programmatically build up the expression.

```
import static LyreDSL.*;

LyreExpression l1 = Lyre.parse("map(filter(@TOKEN, isContentWord), lower)");
LyreExpression l2 = map(filter(annotation(Types.TOKEN), isContentWord), lower);
```

The code snippet illustrated above gives an example of creating the same expression using both the String representation and the DSL methods. The constructed expression extracts all TOKEN annotations from the HString input filtering them to keep only the content words (i.e. non-stopwords) with the resulting list of filtered tokens mapped to a lowercase resulting a list of string.

4.2.1. Lyre Syntax

Lyre expressions attempt to process and convert input and output types in an intelligent manner. For example, a method that transforms an HString into a String will apply itself to each HString in List. Note that to make these operations more explicit, you can use the `map` and `filter` commands. Lyre is comprised of the following types of expressions (defined in `com.gengoai.hermes.extraction.lyre.LyreExpressionType`):

PREDICATE
A predicate expression evaluates an Object or HString for a given condition returning **true** or **false**. When the object passed in is a collection, the predicate acts as a filter over the items in the collection.
HSTRING
An HString expression evaluates an Object or HString returning an HString as the result. If the resulting object is not already an HString, `HString.toHString(Object)` is called for conversion.
STRING
A string expression evaluates an Object or HString returning a String as the result.
FEATURE

A feature expression evaluates an Object or HString returning a machine learning Feature as the result.
OBJECT
An object expression evaluates an Object or HString returning an object as the result (this is used for Lists).
NUMERIC
A numeric expression evaluates an Object or HString returning a numeric result.
COUNTER
A counter expression evaluates an Object or HString returning a Counter result.

This

The `$_` (or this) operator represents the current object in focus, which by default is the object passed into one of the LyreExpression's apply methods. Note that one-argument methods in Lyre (e.g. lower, isUpper, etc.) have an implied `$_` argument if none is given.

Literals

String Literals: Lyre allows for string literals to be specified using single quotes ('). The backslash character can be use to escape a single quote if it is required in the literal.

```
'Orlando'
'\'s'
```

Numeric Literals: Lyres accepts numerical literal values in the form of ints and doubles and allows for scientific notation. Additionally, negative and positive infinity can be expressed as `-INF` and `INF` respectively and NaN as `NaN`.

```
12
1.05
1e-5
```

Null: Null values are represented using the keyword `null`.

```
$_ = null
```

Boolean Literals: Boolean values are represented as `true` and `false`.

```
isStopWord = true
```

Lists

A list of literals or expressions can be defined as follows:

```
[1.0, 2.0, 3.0]
['Orlando', 'Dallas', 'Phoenix']
[lower, upper, lemma]
```

Note when a list is the return type and the returned list would have a single item the single item is returned instead. For example, if a method generated the list `[1]`, the value `1` would be returned instead of the list.

Length: The length of a list is determined using the `llen' method as follows:

```
llen( @ENTITY )
```

where we are returning the length of the list of entities on the object in focus.

List Accessors: Lyre provides three methods for accessing a list of items:

- `first(LIST)`: Return the first element of a list expression or null if none.
- `last(LIST)`: Return the last element of a list expression or null if none.
- `get(LIST, INDEX)`: Gets the i-th element in the given list or null if the index is invalid.

The following code snippet illustrates using these three accessor methods:

```
first( @ENTITY ) ①
last( @ENTITY ) ②
get(@TOKEN, 10) ③
```

① Returns the first entity overlapping the object in focus.

② Returns the last entity overlapping the object in focus.

③ Get the 10th token overlapping the object in focus.

List Selectors: Lyre provides two methods for selecting the best item in a list:

- `max(LIST, INDEX)`: Return the annotation in the list expression with maximum confidence as obtained via the *CONFIDENCE* attribute or null if none.
- `longest(LIST, INDEX)`: Return the longest (character length) element of a list expression or null if none.

The following code snippet illustrates using these two selection methods:

```
max( @ENTITY ) ①
longest( @ENTITY ) ②
```

① Gets the entity with maximum confidence overlapping the object in focus.

② Gets the entity with longest character length overlapping the object in focus. Note that unlike `max` the entity returned from `longest` may not be the one they system is most confident in, but instead is the one that covers the most amount of text.

List Transforms: Lyre provides three methods of transforming a list:

- `map(LIST, EXPRESSION)`: The map operator applies the given expression to each element of the given list.
- `filter(LIST, EXPRESSION)`: The filter operator retains items from the given list for which the given expression evaluates to **true**.
- `flatten(LIST)`: Flattens all elements in a list recursively.

Note that Lyre will create a one-item list if the list item passed in is not a collection. The following code snippet illustrates using these three transform methods:

```
map(@PHRASE_CHUNK, lower) ①
filter(@TOKEN, isContentWord) ②
flatten( map(@TOKEN, [ 'p1=' + $_[:-1], 'p2=' + $_[:-2] ] )  ) ③
```

① Lower cases each phrase chunk overlapping the current object in focus. (Note this is the same as `lower(@PHRASE_CHUNK)`)

② Keeps only the tokens overlapping the current object in focus which are content words. (Note this is the same as `isContentWord(@TOKEN)`)

③ Create a flattened list of unigram and bigram prefixes of all tokens on the current HString.

List Predicates: Lyre provides three methods for testing a list based on its items:

- `any(LIST, EXPRESSION)`: Returns **true** if any item in the given list evaluates to **true** for the given predicate expression.
- `all(LIST, EXPRESSION)`: Returns **true** if all items in the given list evaluates to **true** for the given predicate expression.
- `none(LIST, EXPRESSION)`: Returns **true** if none of the items in the given list evaluates to **true** for the given predicate expression.

Note that Lyre will create a one-item list if the item passed in is not a collection. The following code snippet illustrates using these three predicate methods:

```
any(@TOKEN, isStopWord) ①
all(@TOKEN, isContentWord) ②
none(@TOKEN, isContentWord) ③
```

① Returns **true** if any token overlapping the object in focus is a stopword, e.g. it would evaluate to true when being tested on "the red house" and false when tested on "red house".

② Returns **true** if all tokens overlapping the object in focus are content words, e.g. it would evaluate to true when being tested on "red house" and false when tested on "the red house".

③ Returns **true** if none of the tokens overlapping the object in focus are content words, e.g. it would evaluate to true when being tested on "to the" and false when tested on "to the red house".

Operators

Logical Operators: Lyre provides a set of logical operators for and (`&&`), or (`||`), and xor (`^`) that can be applied to two predicate expressions. Note that if a non-predicate expression is used it will evaluated as a predicate in which case it will return **false** when the object being tested is null and **true** when not null with the following checks for specific types of the expression being treated as a predicate:

1. Collection: **true** when non-empty, **false** otherwise.
2. CharSequence: **true** when not empty or null, **false** otherwise.
3. Lexicon: **true** when the item being tested is in the lexicon, **false** otherwise.
4. Number: **true** when the number is finite, **false** otherwise.
5. Part of Speech: **false** when the part-of-speech is "ANY" or null, **true** otherwise.

Negation: Lyre uses `!` to denote negation (or not) of a predicate, e.g. `!isLower` negates the the string predicate testing for all lowercase letters, returning **true** if the string passed in has any non-lowercase letter.

Relational Operators: Lyre provides the standard set of relational operators, `=`, `<`, `<=`, `>`, `>=`, and `!=`. How the left-hand and right-hand sides are compared is dependent on their type. The following table lists the comparison rules.

LHS Type	RHS Type	Comparison
`null`	`ANY`	equality and inequality perform a reference check and all other operations return false.
`ANY`	`null`	equality and inequality perform a reference check and all other operations return false.
`NUMBER`	`NUMBER`	double-based numeric comparison.
`TAG`	`TAG`	equality and inequality check based on `isInstance( Tag )` all other operations perform comparison based on the `name` of the tags.

LHS Type	RHS Type	Comparison
`TAG`	`TAG`	equality and inequality check based on `isInstance( Tag )` all other operations perform comparison based on the `name` of the tags.
`instanceOf(RHS)`	`instanceOf(LHS)`	Standard object-based comparison.
`CharSequence`	`CharSequence`	string-based comparison.
`ANY`	`NOT CharSequence`	Tries to convert the LHS into the type of the RHS and reapplies the rules.
`NOT CharSequence`	`ANY`	Tries to convert the RHS into the type of the LHS and reapplies the rules.

Pipe Operators: Lyre provides two pipe operators. The first is the And-pipe operator `&>` which sequentially processes each expression with the output of the previous expression or the input object for the first expression. All expression are evaluated regardless of whether or not a null value is encountered. The second is the Or-pipe operator `|>` which sequentially processes each expression with the input object, returning the result of the first expression that evaluates to a non-null, non-empty list, or finite numeric value.

```
map(@TOKEN, lower &> s/\d+/#/g) ①
map(@TOKEN, filter($_, isContentWord) |> 'STOPWORD') ②
```

① Maps the tokens overlapping the object in focus first to lowercase and then for each lowercase token replaces all digits with "#".

② Maps the tokens overlapping the object in focus to themselves when they are content words and to the literal value 'STOPWORD' when they are not content words.

Plus: The plus operator, `+`, can be used to concatenate strings, perform addition on numeric values, or append to a list. Which operation is performed depends on the LHS and RHS type as follows in order:

LHS Type	RHS Type	Comparison
`Collection`	`ANY`	Add the RHS to the collection unless the RHS is null.
`null`	`Collection`	Return the RHS.
`HString`	`HString`	Perform a union of the two Hstring.
`NUMBER`	`NUMBER`	Add the two numeric values together.
`null`	`null`	Return an empty list.
`null`	`ANY`	Return the RHS.
`ANY`	`null`	Return the LHS.
`ANY`	`ANY`	Return the concatenation of the two objects' string representation.

Membership Operators: Lyre provides to membership operators the `in` and the `has` operator. The in operator, `LHS in RHS`, checks if the left-hand object is "in" the right-hand object, where in means "contains". Lyre is able to handle collections, lexicons, and CharSequence as the right-hand object.

```
'a' in 'hat' ①
'dog' in ['cat', 'dog', 'bird'] ②
```

① Returns true if the character 'a' is the string 'hat'.

② Returns true if the string 'dog' is in the given list.

The has operator, `LHS has RHS`, checks if any annotations on the LHS HString evaluates to true using the right-hand expression.

```
$_ has #NP(@PHRASE_CHUNK)
```

The code snippet above checks if the current HString in focus has any phrase chunks whose part-of-speech is NP (Noun Phrase).

Slice Operator: Performs a slice on Strings and Collections where a slice is a sub-string or sub-list. Slices are defined using the square brackets, `[` and `]`, with the starting (inclusive) and ending (exclusive) index separated by a colon, e.g. `[0:1]`. The starting or ending index can be omitted, e.g. `[:1]` or `[3:]`, where the implied starting index is *0* and the implied ending index is the length of the object. Additionally, the ending index can be given as a relative offset to the end of the item, e.g. `[:-2]` represents a slice starting at 0 to item length -2. An example of the slice operator is as follows:

```
$_[:-1] ①
$_[2:] ②
['A', 'B', 'C'][0:2] ③
['A', 'B', 'C'][0:4] ④
['A', 'B', 'C'][40:] ⑤
```

① Creates a substring starting at 0 and ending at the length of the string - 1.

② Creates a substring starting at 2 and ending at the length of the string

③ Creates a sub-list starting at index 0 and ending at index 2 (exclusive).

④ Creates a sub-list starting at index 0 and ending at index 4(exclusive). Note that the list is of length 3 and therefore will return a copy of the entire list.

⑤ Creates a sub-list starting at index 40 and ending at the last item in the list. Note that the list is of length 3 and therefore will return an empty list as there is no 40th item.

Length: The length in characters of the string representation of an object or the number of items in a list can be determined using the `len` method.

Conditional Statements

If: The if-then ,`if( PREDICATE, TRUE_EXPRESSION, FALSE_EXPRESSION )`, method performs a given true or false expression based on a given condition. The following example snippets checks if the object in focus is a digit and when it is returns the string literal `'[:digit:]` and when it is not returns the item.

```
if(isDigit, '[:digit:]', $_)
```

When: The when, `when( PREDICATE, TRUE_EXPRESSION )`, performs the given expression when the given condition is true and returns **null** when the condition is false. The following example snippets checks if the length of the item in focus is greater than three and when it is returns the concatenation of the string literal `'s3='` and the substring/sublist starting at index 3 to the end of the item.

```
when( len > 3, 's3=' + $_[-3:] )
```

Not Null: The not null,`nn( EXPRESSION, DEFAULT_VALUE_EXPRESSION )`, returns the result of the given expression when not null and the result of the default value expression when null. The following example snippets checks for the existence of entities on the object in focus and when there are none (the empty list is automatically converted into a null value) will return the string literal `'non-entity'`.

```
nn( @ENTITY, 'non-entity' )
```

Predicates

Match All: The all predicate, `~`, returns **true** for all input.

Exists: The exists predicate, `exists( OBJECT )` checks if the Object exists meaning it is not null and not a blank CharSequence or empty list.

```
exists(@ENTITY)
```

The code snippet above checks if the object in focus has at least one overlapping entity returning **true** if it does and **false** otherwise.

Look Behind: The positive, `(?< ⋯)`, and negative, `(?!< ⋯ )`, look behind predicates determine if the previous annotation matches (positive) or does not match (negative) the given expression. Note that positive and negative look behinds should be done on single HStrings or used in a list operator.

```
(?< /M[rs]\\./g ) #NNP ①
filter(@TOKEN, (?< /M[rs]\\./g ) #NNP) ②

(?!< /M[rs]\\./g ) #NNP ③
filter(@TOKEN, (?< /M[rs]\\./g ) #NNP) ④
```

① Positive look behind returning **true** if the HString in focus is a proper noun and is preceded by a Mr. os Ms.

② Positive look behind returning a list of tokens which are proper nouns and are preceded by Mr. or Ms.

③ Negative look behind returning **true** if the HString in focus is a proper noun and is **not** preceded by a Mr. os Ms.

④ Negative look behind returning a list of tokens which are proper nouns and are **not** preceded by Mr. or Ms.

Look Ahead: The positive, `(?> …)`, and negative, `(?!> … )`, look ahead predicates determine if the next annotation matches (positive) or does not match (negative) the given expression. Note that positive and negative look aheads should be done on single HStrings or used in a list operator.

```
#NNP (?> $_ = 'inc.') ①
filter(@TOKEN, #NNP (?> $_ = 'inc.')) ②

#NNP (?!> $_ = 'inc.') ③
filter(@TOKEN, #NNP (?!> $_ = 'inc.')) ④
```

① Positive look ahead returning **true** if the HString in focus is a proper noun and is followed by the word "inc.".

② Positive look behind returning a list of tokens which are proper nouns and are followed by the word "inc.".

③ Negative look behind returning **true** if the HString in focus is a proper noun and is **not** followed by the word "inc.".

④ Negative look behind returning a list of tokens which are proper nouns and are **not** followed by the word "inc.".

String Matching: Lyre provides the following predicates for matching strings:

- `isLower( OBJECT )`: Returns **true** if the string version of the object is all lowercase.
- `isUpper( OBJECT )`: Returns **true** if the string version of the object is all uppercase.
- `isWhitespace( OBJECT )`: Returns **true** if the string version of the object is all whitespace.
- `isLetter( OBJECT )`: Returns **true** if the string version of the object contains only letters.
- `isDigit( OBJECT )`: Returns **true** if the string version of the object contains only digits.

- `isAlphaNumeric( OBJECT )`: Returns **true** if the string version of the object contains only alphanumeric characters.
- `isPunctuation( OBJECT )`: Returns **true** if the string contains all punctuation characters.
- `isContentWord( OBJECT )`: Returns **true** if the string / HString is a non-stopword.
- `isStopWord( OBJECT )`: Returns **true** if the string / HString is a stopword.
- `hasStopWord( OBJECT )`: Returns **true** if the string / HString contains a stopword, i.e. any token in the HString is a stopword.
- `/PATTERN/: Returns *true` if the string / HString matches the given regular expression where the regex can have the options `i` for case-insensitive and `g` for matching the full span.

Note that for all methods listed above, the `OBJECT` is optional and will default to `$_` if not specified.

Tag Matching: The tag predicate, `#TAG_VALUE( OBJECT )`, checks if the Tag on the object in focus is of the given Tag value. Note that object in focus must be an annotation and the tag value needs to be convertible to that annotation's tag type.

```
filter(@TOKEN, #NNP)①

filter(@ENTITY, #PERSON) ②
```

① Filter all tokens on the object in focus only retaining those that have a NNP part-of-speech.

② Filter all entities on the object in focus only retaining those with an entity type of PERSON.

Transforms

String transforms: Lyre provides the following methods for transforming objects into a string representation:

- `string( OBJECT )`: Transforms the input into a string by calling the `toString()` method.
- `lower( OBJECT )`: Transforms the input into a lowercase string.
- `upper( OBJECT )`: Transforms the input into an uppercase string.
- `lemma( OBJECT )`: Transforms the input into the lemmatized version.
- `stem( OBJECT )`: Transforms the input into the stemmed version.
- `s/pattern/replacement/[ig]*`: Transforms the input by performing a regular expression substitution with where the `i` option indicates a case-insensitive match and `g` indicates replace-all. Note that this method is accessible via the LyreDSL through the `rsub` set of methods.

All of the methods listed above can specified by their name only, e.g. `string`, when taking `$_` (this) as the argument.

Apply transform: Lyre uses `LHS ~= RHS` to denote that the RHS expression is to be applied to the LHS expression. This is especially useful for regular expression matching and substitution.

```
$_ ~= /[Oo]rlando/; ①
@PHRASE_CHUNK ~= [lower, upper] ②
```

① Applies the regular expression match as a predicate to the current object in focus.

② Applies the list of Lyre expressions to each phrase chunk in the object in focus returning a List of two-element lists where the first element is the lowercase version and the second element the uppercase version of the phrase chunk.

Context Expansion: The context method, `cxt( HSTRING, CONTEXT_SIZE )`, returns a contextual (previous or next) token for the given HString at the given position (relative) where a positive value for the context size represents a next contextual token and a negative value a previous contextual token.

```
cxt( $_, -1) ①
cxt( $_, 10) ②
```

① Returns the token left of the current HString.

② Returns the token 10 to the right of the current HString.

String Padding:

- `lpad( OBJECT, MINIMUM_LENGTH, PADDING_CHARACTER )`: Transforms the string version of the input object to ensure it is of the minimum length by prepending the padding character.
- `rpad( OBJECT, MINIMUM_LENGTH, PADDING_CHARACTER )`: Transforms the string version of the input object to ensure it is of the minimum length by appending the padding character.

Note that in both `lpad` and `rpad` if the given PADDING_CHARACTER has a length > 1 only the first character is used.

```
lpad('ab', 4, '^'); ①
rpad('ab', 3, '$') ②
```

① Pads the given input literal `ab` to be of length 4 by prepending `^` resulting in `^^ab`.

② Pads the given input literal `ab` to be of length 3 by appending `$` resulting in `ab$`.

HString Trimming: The trim, `trim(HSTRING, PREDICATE)`, method removes tokens from the left and right of the input HString if they evaluate to true with the given predicate. The following code snippet trims token from the input HString when they are a stopword or have a length less than three characters.

```
trim( $_, isStopWord || len < 3 )
```

AnnotatableType Accessors

Annotation Accessors: Lyre provides the following methods for accessing the annotations on an HString:

- `@ANNOTATION_TYPE`: Return a list of the annotation for the given `ANNOTATION_TYPE` overlapping the HString in focus.
- `@ANNOTATION_TYPE{TAG}`: Return a list of the annotation for the given `ANNOTATION_TYPE` overlapping the HString in focus filtered to only those whose tag is an instance of the given tag value.
- `interleave('ANNOTATION_TYPE', …, 'ANNOTATION_TYPE')`: Return a list of the annotation for the given `ANNOTATION_TYPE` overlapping the HString in focus using the `HString.interleave(AnnotationType…)` method. Note that the AnnotationType names are given as string literals.
- `@>RELATION_TYPE`: Retrieves a list of the annotations that are reachable via an outgoing relation for the given `RELATION_TYPE`.
- `@>RELATION_TYPE{'RELATION VALUE'}`: Retrieves a list of the annotations that are reachable via an outgoing relation for the given `RELATION_TYPE` and having the given `RELATION_VALUE`.
- `@>`: Retrieves a list of the annotations that are reachable via an outgoing dependency relation.
- `@>{'DEPENDENCY_TYPE'}`: Retrieves a list of the annotations that are reachable via an outgoing dependency relation of given `DEPENDENCY_TYPE`.
- `@<RELATION_TYPE`: Retrieves a list of the annotations that are reachable via an incoming relation for the given `RELATION_TYPE`.
- `@<RELATION_TYPE{'RELATION VALUE'}`: Retrieves a list of the annotations that are reachable via an incoming relation for the given `RELATION_TYPE` and having the given `RELATION_VALUE`.
- `@<`: Retrieves a list of the annotations that are reachable via an incoming dependency relation.
- `@<{'DEPENDENCY_TYPE'}`: Retrieves a list of the annotations that are reachable via an incoming dependency relation of given `DEPENDENCY_TYPE`.

Note that in call cases if the returned list has a single annotation, the single annotation will be returned instead.

The following code snippet shows examples of extracting annotations in Lyre.

```
@ENTITY ①
@TOKEN( @ENTITY ) ②
@>COREFERNCE{'pronominal'} ③
@<dep{'nsubj'} ④
@ENTITY{PERSON} ⑤
```

① Returns a list of annotations (or single annotation if only one) of the entities overlapping the object in focus, `$_`, realized as an HString using `HString.toHString(Object)`.

② For each Entity annotation overlapping the object in focus, extract the tokens as a list, which results in a list of object where the object is a single annotation or list of annotations.

③ Get all annotations reachable via an outgoing COREFERENCE relation from the object in focus where the COREFERENCE relation has the value "pronominal".

④ Get all annotations reachable via an incoming dependency relation from the object in focus where the dependecy relation has the type "nsubj".

⑤ Returns a list of annotations of the PERSON entities overlapping the object in focus.

Attributes: Lyre provides the following methods for accessing the attributes on an HString:

- `$ATTRIBUTE_TYPE( OBJECT )`: Retrieves the value of the given `ATTRIBUTE_TYPE` on the object in focus.
- `pos( OBJECT )`: Gets the part-of-speech tag on the object in focus.
- `upos( OBJECT )`: Gets the universal part-of-speech tag on the object in focus.

*Token Length: To determine the number of tokens in an object in focus the `tlen` method is available.

Lexicons and Word Lists

Lexicons: The lexicon function, `%LEXICON_NAME`, Returns the lexicon with the given `LEXICON_NAME` using Hermes's LexiconManager. The returned lexicon can then be used to check of the existence of words. The following snippet gives two examples of checking if the object in focus is in a lexicon named `person`:

```
$_ in %person
%person
```

As can be seen in the snippet above, the lexicon acts as a predicate checking for existence.

Word Lists: The word list function, `wordList( LIST_EXPRESSION )`, will create a temporary word list for use. The following code snippet shows an example of constructing a temporary word list made up of the strings "dog", "cat", and "bird" and using the constructed word list as a container to check if the object in focus's string representation if one of the items in the word list.

```
$_ in wordList(['dog', 'cat', 'bird'])
```

Feature and Count Generators

Feature Generator: Lyre provides the following methods for easily constructing lists of features:

- `binary{'PREFIX'}( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into binary features prefixed with the given `PREFIX`.
- `binary( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into binary features.
- `frequency{'PREFIX'}( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into

features prefixed with the given `PREFIX` whose values are number of times it occurred in the list.

- `frequency( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into features prefixed whose values are number of times it occurred in the list.
- `L1{'PREFIX'}( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into features prefixed with the given `PREFIX` whose values are number of times it occurred in the list divided by the total number of items in the list.
- `L1( EXPRESSION )`: Converts the list of values returned by the given `EXPRESSION` into features whose values are number of times it occurred in the list divided by the total number of items in the list.
- `iob( 'ANNOTATION_TYPE' )`: Generates IOB-formatted tags for the given `ANNOTATION_TYPE` overlapping the object in focus.
- `iob( 'ANNOTATION_TYPE', OBJECT )`: Generates IOB-formatted tags for the given `ANNOTATION_TYPE` overlapping the given OBJECT.

```
binary{'WORD'}(@TOKEN)①
frequency{'WORD'}(@TOKEN)②
K1{'WORD'}(@TOKEN)③
iob('PHRASE_CHUNK', @SENTENCE) ④
```

① Generates a list of binary features containing the string form of the tokens overlapping the object in focus with a 'WORD' prefix.

② Generates a list of features with frequency counts containing the string form of the tokens overlapping the object in focus with a 'WORD' prefix.

③ Generates a list of features with L1 normalized frequency counts containing the string form of the tokens overlapping the object in focus with a 'WORD' prefix.

④ Generates iob formatted chunk part-of-speech tags over the sentences overlapping the object in focus.

Count Generator: Lyre provides the following methods for easily creating Counters over lists of items:

- `count( EXPRESSION, 'VALUE_CALCULATOR' )`, counts the items in the list obtained from given `EXPRESSION` converting the values using the *ValueCalculator* corresponding to the given literal value `'VALUE_CALCULATOR'`.
- `count( EXPRESSION )`, counts the frequency of the items in the list obtained from given `EXPRESSION`.

4.3. Keyword Extraction

Keyword extraction is the processing of identify key phrases or concepts that a document or collection of document is discussing. There are a number of methods defined for extracting keywords in the NLP literature that range from simple phrase counting to machine learning models that extract phrases from a controlled vocabulary. In general, the various keyword extraction methodologies have a combination of the follow characteristics:

- **Unsupervised** or **Supervised**: Much like machine learning keyword extraction be done in an supervised fashion, often learning a mapping between documents and a controlled vocabulary, or an unsupervised fashion, often in which phrase and corpus statistics are used to determine the importance of phrases.
- **Single Document** or **Corpus**: Many of the early approaches to keyword extraction relied on corpus level statistics and thus keywords could only be generated given a corpus. Newer approaches allow for keyword extraction from a single document where corpus level statistics are not required.

In Hermes all keyword extraction methods implement the *KeywordExtactor* interface which is an *Extractor* with the addition of a `fit(corpus)` method. The `fit` method is used when corpus level statistics are needed in order to perform keyword extraction or when keyword extraction is performed in a supervised manner.

Hermes provides the following keyword extractor implementations:

`Class Name`	**Requires Fit**
`TermKeywordExtractor`	
Extracts a set of keywords based on term frequency using an underlying *FeautrizingExtractor*, such as a Lyre expression.	
`TFIDFKeywordExtractor`	✓
Extracts a set of keywords based on term frequency inverse document frequency (TFIDF) using an underlying *FeautrizingExtractor*, such as a Lyre expression. The call to the `fit(corpus)` method is used to calculate the document frequencies of the terms in the corpus	
`RakeKeywordExtractor`	
An implementation of the RAKE algorithm *Rose, S., Engel, D., Cramer, N., & Cowley, W. (2010). Automatic Keyword Extraction from Individual Documents. In M. W. Berry & J. Kogan (Eds.), Text Mining: Theory and Applications: John Wiley & Sons.*. Rake extracts contiguous spans of text which do not contain stopwords as candidate keywords and scores the candidates based on their order and frequency.	
`NPClusteringKeywordExtractor`	
An implementation of the keyword extractor defined in *Bracewell, David B., Yan, Jiajun, and Ren, Fuji, (2008), Single Document Keyword Extraction For Internet News Articles, International Journal of Innovative Computing, Information and Control, 4, 905—913*. The algorithm extracts noun phrases as candidate keywords and then clusters the noun phrases into semantically coherent groups. These groups are scored and the central noun phrase of the group is then used as a keyword with the group's score as its own.	

4.4. Token-Based Regular Expressions

Hermes provides a token-based regular expression engine that allows for matches on arbitrary annotation types, relation types, and attributes, while providing many of the operators that are possible using standard Java regular expressions. As with Java regular expressions, the token regular expression is specified as a string and is compiled into an instance of of *TokenRegex*. The *TokenRegex* class has many of the same methods as Java's regular expression, but returns a *TokenMatcher* instead of Matcher. The *TokenMatcher* class allows for iterating of the matches, extracting the match or named-groups within the match, the starting and ending offset of the match, and conversion into a *TokenMatch* object which records the current state of the match. Token regular expressions can act as extractors where the extraction generates the HStrings matched for the default group. An example of compiling a regular expression, creating a match, and iterating over the matches is as follows:

```
TokenRegex regex = TokenRegex.compile(pattern);
TokenMatcher matcher = regex.matcher(document);
while (matcher.find()) {
    System.out.println(matcher.group());
}
```

4.4.1. Regular Expression Syntax

The syntax for token-based regular expressions borrows from the Lyre Expression Language where possible. Token-based regular expressions differ from Lyre in that they work over sequences of HStrings whereas Lyre is working on single HString units. As such, there are differences in the syntax between Lyre.

Content Matching

Content can be matched in the following manner:

- **Any**: The `.` operator can be used to match any HString.
- **Case-Sensitive String Match**: Matches based on a case-sensitive match to the string form of the HString, expressed using double quotes, e.g. `"ABC"`.
- **Case-Insensitive String Match**: Matches based on a case-insensitive match to the string form of the HString, expressed using single quotes, e.g. `'ABC'`.
- **Lemmatized String Match**: Matches based on a case-insensitive match to the lemmatized form of the HString, expressed using the less than and greater signs, e.g. `<ABC>`.
- **Lexicon Match**: Matches against a lexicon can be made by expressing the name of the lexicon to match against using `%LEXICON_NAME`.

```
"Orlando" ①
'orlando' ②
<fly> ③
```

① Case-sensitive match in which only HStrings containing only the word "Orlando" are matched.

② Case-insensitive match in which all capitalization variations of the word "orlando" are matched.

③ Case-insensitive match in which all HString whose lemmatized is "fly" will be matched (e.g., "flies", "flew", and "flown").

Word Classes

The following word classes are defined:

- `AlphaNumeric`: Matches HStrings whose string form contains only alphanumeric characters.
- `ContentWord`: Matches HStrings which are content words, i.e. non-stopwords.
- `HasStopWord`: Matches HStrings which contain on or more stopwords.
- `StopWord`: Matches HStrings which are stopwords.
- `Upper`: Matches HStrings whose string form contains only uppercase letters.
- `UpperInitial`: Matches HStrings whose first character is an uppercase letter.
- `Lower`: Matches HStrings whose string form contains only lower letters.
- `LowerInitial`: Matches HStrings whose first character is a lowercase letter.
- `Letter`: Matches HStrings whose string form contains only letters.
- `Digit`: Matches HStrings whose string form contains only digits.
- `Number`: Matches HStrings which represents numbers as ascertained by their `TOKEN_TYPE` or `PART_OF_SPEECH` attribute or whose string form contains only digits.
- `Punctuation`: Matches HStrings whose string form contains only punctuation.
- `/PATTERN/[ig]*`: Matches HStrings whose string form matches the given regular expression where the regex can have the options `i` for case-insensitive and `g` for matching the full span.

Attributes

Token-based regular expressions allow for matching of attribute values including numeric comparisons for number-valued attributes. The first set of attribute-based matching will match non-numeric attributes, such as strings and tags.

```
$PART_OF_SPEECH = 'NOUN' ①
$PART_OF_SPEECH != 'NOUN' ②
$COLLECTION ~ 'VALUE' ③
```

① Equality check where tag-valued attributes are match if the value of the attribute is an instance of the tag defined on the right-hand side and non-tag valued attributes are match if the value of the left-hand side is equal to the value on the right-hand side.

② Inequality check where tag-valued attributes are match if the value of the attribute is **not** an instance of the tag defined on the right-hand side and non-tag valued attributes are match if the value of the left-hand side is **not** equal to the value on the right-hand side.

③ Containment check where a match is made when the value on the right-hand side is in the collection of values associated with the attribute or is a substring of the string value of the left-hand side.

Additionally, the full range of numeric comparisons are available:

```
$CONFIDENCE = 0.0
$CONFIDENCE != 0.0
$CONFIDENCE >= 0.50
$CONFIDENCE > 0.50
$CONFIDENCE <= 0.50
$CONFIDENCE < 0.50
```

Tags: Tag attributes can be easily matched using `#TAG_VALUE`, e.g. `#NOUN` for tokens would match all tokens whose part-of-speech is a noun.

Categories: Whether or not an HString or one of its sub-spans contains a given category value can be expressed using `$CATEGORY ~ 'VALUE'` or `$CAT ~ 'VALUE'` where the VALUE must be predefined *BasicCategory*.

Annotations

Matches can include the existence of annotations including optional constraints on the match. Annotations are declared using `@ANNOTATION_TYPE` and constraints can be specified using parenthesis, i.e. `@ANNOTATION_TYPE(…)`. The following code snippet gives examples:

```
@ENTITY( $CONFIDENCE >= 0.90 ) ①
@PHRASE_CHUNK( #NP ) @PHRASE_CHUNK( #VP ) @PHRASE_CHUNK( #NP ) ②
```

① Match all entities with a confidence value of 0.90 or more.

② Match sequences of phrase chunks in the form NP VP NP

That the syntax of matching with annotations is backwards compared to Lyre. In Lyre, the first example in the snippet above would be expressed as `$CONFIDENCE(@ENTITY) >= 0.90`.

Relations

• `@<RELATION_TYPE`: Retrieves a the annotations that are reachable via an incoming relation of the

given type.

- @<RELATION_TYPE{'RELATION_VALUE'}: Retrieves a the annotations that are reachable via an incoming relation of the given type and having the given relation value.
- @>RELATION_TYPE: Retrieves a the annotations that are reachable via an outgoing relation of the given type.
- @>RELATION_TYPE{'RELATION_VALUE'}: Retrieves a the annotations that are reachable via an outgoing relation of the given type and having the given relation value.
- @<: Retrieves a the annotations that are reachable via an incoming dependency relation.
- @<{'DEPENDENCY_TYPE'}: Retrieves a the annotations that are reachable via an incoming dependency relation and having the given DEPENDENCY_TYPE.
- @>: Retrieves a the annotations that are reachable via an outgoing dependency relation.
- @>{'DEPENDENCY_TYPE'}: Retrieves a the annotations that are reachable via an outgoing dependency relation and having the given DEPENDENCY_TYPE.

```
@PHRASE_CHUNK & @>{'nsubj'} ①
@<ROLE{'AGENT'}( @ENTITY( #PERSON ) ) ②
```

① Match all phrase chunks that have the 'nsubj' dependency relation.

② Match all HString which have an incoming "ROLE" relation with value "AGENT" and whose "AGENT" is an entity of type PERSON

Greedy Qualifiers

The following set of greedy qualifiers are supported:

- **Kleene Star**: The * unary operator will match zero or more of the previously defined expression, e.g. 'abc'* will match zero or more HString whose string form is a case-insensitive match to "abc".
- **Kleene Plus**: The + unary operator will match one or more of the previously defined expression, e.g. 'abc'+ will match one or more HString whose string form is a case-insensitive match to "abc".
- **Optional Marker**: The ? unary operator denotes that previous expression is optional, i.e. matches zero or one, , e.g. 'abc'? will match if the previous HString is empty or whose string form is a case-insensitive match to "abc".
- **Range Operator**: A variable number of matches can be expressed using the range operator, denoted using {minimum,maximum}, where the maximum can be omitted to denote matching an exact number of times, have the value "*" denoting any number of maximum matches, or given numeric value expression the maximum number of matches.

Logical Operators

Hermes’s token-based regular expressions also supports the following logical operators:

- **Negation**: The `^` operator denotes negation matching only when the next expression evaluates to true.
- **Alternations**: Alternations can be expressed using `|` and act as an **or** on the matching, e.g. `('cat'|'dog')` would match either "cat" or "dog".
- **And**: Ands can be expressed using `&` and require both the left-hand and right-hand sides to evaluate to true for the current HString in order for it to match, e.g. `@ENTITY(#PERSON & $CONFIDENCE >0.6)` would match entities which are only of type PERSON and have a confidence greater than 0.6.
- **Non-Capturing Groups**: Non-capturing groups are used to force the order of operation and are denoted using parenthesis, e.g. `^('man' 'of')` matches any two HStrings which whose sequence is not equal to "man of".

Special Constructs

The following special constructs are supported:

Look Behind: The positive, `(?< …)`, and negative, `(?!< … )`, look behind predicates determine if the previous annotation sequences matches (positive) or does not match (negative) the given expression.

```
(?< 'manager' 'of' ) @PHRASE_CHUNK(#NOUN)
(?!< #DT | #ADJECTIVE ) <bank>
```

The first expression, from the snippet above, matches noun phrases which are preceded by the phrase "manager of". The second example, matches all HStrings whose lemmatized form is "bank" and are not preceded by a determiner or adjective.

Look Ahead: The positive, `(?> …)`, and negative, `(?!> … )`, look ahead predicates determine if the next annotation sequence matches (positive) or does not match (negative) the given expression.

```
PHRASE_CHUNK( #NOUN ) (?> PHRASE_CHUNK(#VERB) )
Digit (?!> Punctuation )
```

The first expression, from the snippet above, matches noun phrases which are followed by verb phrases. The second example, matches all HStrings which represent Digits and are not followed by punctuation.

Named Groups: Named groups can specified using `(?<NAME> …)`. The groups captured HString is accessed using `TokenMatcher.group(String)` where the String is the group name. The following example illustrates a named group:

```
(?<PERSON> @ENTITY(#PERSON))
```

Backreference: Backreference to named groups is possible using `\GROUP_NAME`.

```
(?<PERSON> @ENTITY(#PERSON)) .* PHRASE_CHUNK(#VERB & @<{'nsubj'}(\PERSON))
```

The example snippet above matches all sequences in which a person entity is followed by a distant verb phrase for which the the nsubj is the matched person entity. Note that the backreference will match when there is overlap of between the HString in the named group and the HString being examined for the backreference.

Take a look at **TokenRegexExample.java** in the Hermes examples project to see example patterns.

4.5. Caduceus

Caduceus, pronounced **ca·du·ceus**, is a rule-based information extraction system. Caduceus programs consist of a list of rules for extracting arbitrary spans of text to define annotations (e.g. entities and events) and relations (e.g. event roles). Each rule starts with a unique name declared in square brackets, e.g. `[my_rule]`. Following the rule name is the **trigger**, which is a Token-Based Regular Expressions that captures the text causing the rule to fire. Example triggers are as follows:

```
trigger: (<man> | <woman> | <child> | <baby>) ①
trigger: (?<PERSON> @ENTITY(#PERSON)) (?<ACTION> @PHRASE_CHUNK(#VERB)) (?<OBJECT> @PHRASE_CHUNK(#NOUN))
②
```

① A simple trigger that fires for HStrings whose lemmatized form match one of the given lemmas.

② A trigger that requires a PERSON entity followed by verb phrase followed by a noun phrase, which each item being a named group.

Rules construct annotations and/or relations based on the matched trigger. A rule may have define zero or more annotations to be constructed. Each annotation is defined using `annotation:` and requires the following options to be specified:

- `capture=(*|GROUP_NAME)`: The text span which will make up the annotation, where `\*` represents the full trigger match and `GROUP_NAME` represents a named group from the trigger match.
- `type=ANNOTATION_TYPE`: The name of the annotation type to construct.

Additionally, attributes can be defined using as follows:

```
$ATTRIBUTE_NAME = VALUE
```

An example of a rule to create ENTITY annotations of type PERSON and BODY_PART is as follows:

```
//###################################################################################
// Entity: PERSON - Identifies PERSON entities from Alice in Wonderland
//###################################################################################
[person]
trigger: ('alice' | 'rabbit' | ('white' 'rabbit') | ('mad' 'hatter') )
annotation: capture=*
            type=ENTITY
            $ENTITY_TYPE=PERSON
            $CONFIDENCE=1.0
```

```
//###################################################################################
// Entity: BODY_PART - Identifies body part entities.
//###################################################################################
[body_parts]
trigger: ( (<eye> | <ear> | <arm> | <leg> | <head>) && #NOUN )
annotation: capture=*
            type=ENTITY
            $ENTITY_TYPE=BODY_PART
            $CONFIDENCE=1.0
```

Comments are specified using `//`.

A rule may have define zero or more relations to be constructed. Each relation is defined using `relation: NAME`, where NAME is unqiue in the rule. Relations require the following options to be specified:

- `type=RELATION_TYPE`: The name of the relation type to construct.
- `value=STRING`: The value of the relation type.
- `@>`: The source of the relation specified using a Lyre Expression Language with an optional named matching group defined like `@>{GROUP_NAME}`
- `@<`: The target of the relation specified using a Lyre Expression Language with an optional named matching group defined like `@>{GROUP_NAME}`

Additionally, a relation can be defined as *bidirectional* by specifying `bidirectional=true`. An example of a rule to create a HAS_A relation between a PERSON and BODY_PART is as follows:

```
//####################################################################################
// Relation: HAS_A - Relates a BODY_PART to a PERSON
//####################################################################################
[body_part_attributes]
trigger: (?<PERSON> @ENTITY(#PERSON)) "with" .{1,3} (?<BODY_PART> @ENTITY( #BODY_PART ))
relation:  has_a
           type=ATTRIBUTE
           value= HAS_A
           @>{PERSON}=@ENTITY
           @<{BODY_PART}=@ENTITY
```

In the example rule listed above, we define a trigger that matches a PERSON entity followed by the word "with" followed by between one to three tokens, and finally followed by a BODY_PART entity. The relation definition for HAS_A is named `has_a` and defines an ATTRIBUTE relation with value HAS_A where the source of the relation is the PERSON entity and the target of the relation is the BODY_PART entity.

In order to prevent superfluous matches, annotations and relations can have requirements on each other specified using a `requires` statement. The use of requires can be seen in the following example rule:

```
//###################################################################################
// Murder - The killing of one person by another
// Roles:
//    KILLER - The PERSON performing the murder
//    VICTIM - The PERSON being murdered
//    MANNER - The way in which the murder was performed
//###################################################################################
[murder]
trigger: (?<EVENT> <murder> | <kill> | <shoot> | <stab> | <poison> )
annotation: capture=*
                type=EVENT
                $TAG=MURDER
                $CONFIDENCE=1.0
                requires=VICTIM
                requires=KILLER
relation: KILLER
             type=ROLE
             value=KILLER
             @>=@ENTITY{PERSON}(@<dep{'nsubj'})
             @<=$_
relation: VICTIM
             type=ROLE
             value=VICTIM
             @>=@ENTITY{PERSON}(@<dep{'dobj'})
             @<=$_
relation: MANNER
             type=ROLE
             value=MANNER
             @>=@<dep{'advmod'}
             @<=$_
```

As can be seen in the rule listed above, the MURDER event will only be created when a KILLER role and VICTIM role can be specified.

4.5.1. Rule Processing and Execution

Rules are processed sequentially starting at the first defined rule in a Caduceus program file. This allows rules to be split apart with each being as simple as possible. Please note that Caduceus is not designed for blazing performance, but for ease of use.

Caduceus programs can be executed over corpora or single documents as the following code snippet illustrates.

```
CaduceusProgram events = CaduceusProgram.read(Resources.from("/data/caduceus/events.cg"));

Corpus corpus = ...;
corpus.update(events); ①

Document document = ...;
events.execute(document); ②
```

① Caduceus programs can be executed over a corpus using the corpus `update(CaduceusProgram)` method.

② Individual documents can be processed by calling the `execute(Document)` method on the CaduceusProgram.

Take a look at **CaduceusExample.java** in the Hermes examples project to see a complete example.

5. Machine Learning

Machine learning is commonly used for providing annotations and relations, determining the value of an attribute for a document or annotation, or determining the topics discussed in a corpus. Training of these types of machine learning models is done from a corpus. In the case of supervised learning the corpus contains the gold standard, or correct, annotations, relations, or attributes. Herme's Corpus class makes it easy to construct (see asLabeledStream, asClassificationDataSet, asRegressionDataSet, and asSequenceDataSet in the Javadoc) an Apollo dataset which various machine learning algorithms can be trained or applied.

Take a look at **GettingStarted.java**, **CorpusExample.java**, **MLExample.java**, and **SparkExample.java** in the Hermes examples project to see a complete example.

6. Workflows

A workflow represents a set of *actions* to perform on an document collection. Actions fall into one or more of the following three categories:

1. Modify - The action modifies the documents in the collection, e.g. adds new annotations or attributes.
2. Compute - The action generates information that is added to the *Context* for use by downstream actions.
3. Output - The action generates an output for consumption by external processes and/or downstream actions.

Actions share a common key-value memory store, called a *Context*, in which information they require or they generate can be added.

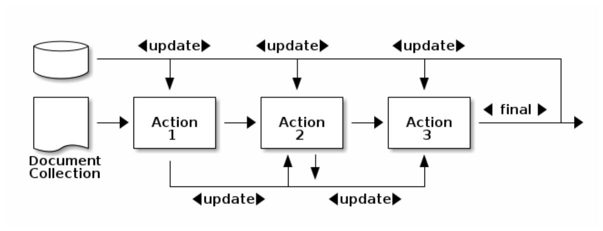

Figure 5. Example of Sequential Workflow

The figure illustrated above gives an example of a sequential workflow where an document collection is passed through a series of three actions each of which update (Modify) the collection and the context. The Workflow Runner the can save the final output to file.

6.1. Contexts

Contexts are a specialized map that act as a shared memory for a Workflow. The context will retrieve values from its internal storage and also fallback to checking for values in the global configuration.

6.2. Actions

An action defines a processing step to perform on a *DocumentCollection* with a given *Context* which results in either modifying the corpus or the context. Action implementations can persist their state to be reused at a later time including across jvm instances and runs. This is done by implementing the `loadPreviousState(DocumentCollection,Context)` method to modify the corpus and/or context based on its saved state. An action can ignore its state and reprocess the corpus when either the config setting `processing.override.all` is set to true or the config setting `className.override` is set tp true.

6.3. Defining a Workflow

Workflows are defined using Json. The follow is an example definition:

```
{
  "type"  : "Sequential",
  "context": {
    "alpha": {"@class": "double", "@value": 0.2}
  },
  "beans" : {
    "ANNOTATOR" : {
        "@singleton": true,
        "@class": "com.gengoai.hermes.workflow.actions.Annotate",
        "types": "${CORE_ANNOTATIONS}"
      }
  },
  "actions": [
      "@ANNOTATOR",
      {
        "@class": "com.gengoai.hermes.workflow.actions.TermCounts",
        "extractor": "lower(filter(@TOKEN,isContentWord))",
        "documentFrequencies": true
      }
  ]
}
```

Each workflow json must specify the workflow type (currently only Sequential workflows are supported). An initial set of key-value pairs can be specified in the "context" section. Note that its require to specify values as json objects in the format `{"@class":"java class name", "@value": value}`. The json format allows for reusable beans to be defined in the "beans" section. The bean format uses the bean name as the key for a json object which is made up of a "@class" property to define the fully qualified class name, "@singleton" to optionally define the object as a singleton, and all setters are defined as properties in the object, e.g. "types" relates to the `setTypes` method of the Annotate action. Finally the set of actions are defined as an array within the "actions" object.

7. Hermes Applications

7.1. Workflow Runner

7.2. annotate

7.3. csv2Lexicon

The **csv2Lexicon** application converts a csv file containing lexical information into a Hermes lexicon. The application has the following command line options:

- **csv**: The csv lexicon specification.
- **lexicon**: The output lexicon specification.

An example command line is as follows:

```
csv2Lexicon --csv lexicon:mem:adhoc:csv::/data/test/import.csv;probability=-
1;constraint=2;caseSensitive=true;tagAttribute=ENTITY_TYPE;defaultTag=PERSON --lexicon
lexicon:disk:adhoc::/data/test/adhoc.lexicon
```

[1] See the Mango User Guide for details on the Graph data structure.