OpenDMAP Project

MANUAL

User Manual

OpenDMAP Project

Center for Computational Pharmacology

Version 1.1 3/15/08

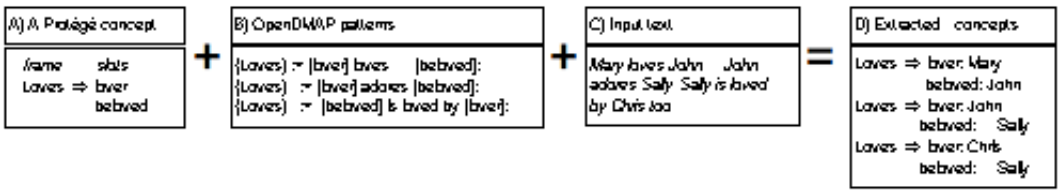
1. Introduction to OpenDMAP

OpenDMAP is an ontology-driven, rule-based concept analysis and information extraction system. It processes texts to recognize concepts and relationships from a knowledge base. OpenDMAP elevates the use of conceptually organized semantic information to be the primary organizing architecture of the system. Unlike traditional parsers, OpenDMAP does not have a lexicon that maps from words to all the possible meanings of these words. Rather, each concept is associated with phrasal patterns that are used to recognize that concept.

OpenDMAP uses Protégé knowledge bases (http://protege.stanford.edu/) to provide an object model for the possible concepts that might be found in a text. Protégé models concepts as frames that participate in abstraction and packaging hierarchies, and models relationships as frame-specific slots. OpenDMAP was developed for use in the domain of molecular biology. However, it is certainly not limited to this domain – it can be applied to any domain for which a knowledge base exists or can be developed.

OpenDMAP combines the use of patterns and an underlying ontology to extract information from text. For example, say that you have an ontology that has the concept frame *Loves* in it, along with *Loves*'s attending participants, (slots, in Protégé terminology), the *Lover* and the *Beloved* (Figure 1-A). Also, you write a set of OpenDMAP patterns that describes how the *Loves* concept might be realized in text, (Figure 1-B). Given an input text (Figure 1-C), OpenDMAP can extract *Loves* concepts and participants from the text and assign the participants to the ontology concept *Loves* frame (Figure 1-D).

Figure 1: What OpenDMAP does



OpenDMAP is not a standalone application; it is a library code base. Use of OpenDMAP requires the user to include OpenDMAP objects, call methods, and implement inherited classes. Optimal use of OpenDMAP is achieved by combining it with other applications such as entity-taggers, linguistic pre-processors, and applications that make use of the OpenDMAP output. There are two example application projects included with the OpenDMAP download that do not require the user to implement anything. This manual is intended to instruct a non-programming user on how to run the example projects provided with OpenDMAP, and to guide a programming user through the interface and execution of a customized OpenDMAP application.

OpenDMAP was developed by the Center for Computational Pharmacology at the University of Colorado School of Medicine (http://compbio.uchsc.edu/Hunter_lab/,

http://bionlp.sourceforge.net). The code base is licensed under the Mozilla Public License v1.1. See Section 6 for citation information upon your use of OpenDMAP.

2. Installing OpenDMAP and running the Example Projects

This section is predominantly a reiteration of the contents provided in the README file from the OpenDMAP download. However, there is additional information in Section 2.2 regarding the expected output of the GeneRIF-Transport and Love-Hate example applications.

2.1. Installation

2.1.1. Necessary external resources and configuration

As mentioned in the introduction, this manual covers two distinct interactions with OpenDMAP: installing and running the example OpenDMAP example project provided with the download, and executing a customized application using OpenDMAP. Different resources are needed for the different uses.

To run the example applications provided, the following resources are required to be installed on your system:

- Java 1.5
- Apache Ant
- Protégé (more details below)
- Xerces (more details below)

OpenDMAP is platform independent; however, the example applications provided are configured to run on a Unix system using shell scripts.

Java 1.5 is required to run OpenDMAP. Additionally, OpenDMAP depends on two external applications. The Xerces project is included in this distribution, (see the lib/ directory). OpenDMAP also relies on Protégé, which must be downloaded prior to use. Obtain the protege.jar and the looks-#.#.#.jar file by downloading Protégé from http://protege.stanford.edu/. Place the protege.jar file into the lib/ directory. Rename the looks-#.#.#.jar to looks.jar and place in the lib/ directory. If you do not want to rename this file, then you must update the classpath settings in runGenerifConsole.sh and runLoveHateConsole.sh in order for them to work properly.

To execute a customized application, the same resources listed above, (with the exception of Apache Ant), and steps to integrate them with OpenDMAP are required. Additionally, depending on your involvement with the code base, these resources may also be necessary:

- JavaCC
- JJTree

See Section 2.1.4 for more details on how these two resources are used.

While Ant is a not required resource for executing a custom application, there are OpenDMAP compile, test, and package tasks provided in the Ant build.xml file (that comes with the OpenDMAP download) that make development much easier if you chose to use Ant.

Links to all resources listed here are provided in Section 7.

2.1.2. Directory Structure

src/	Contains the OpenDMAP source code.
ant/	Notes: - Use the build.xml file to compile, package, and run the OpenDMAP test suites. - Also, two example applications (described below) are included in this distribution and can be started using the runGenerifConsole.sh and runLoveHateConsole.sh scripts in this directory.
build/	Contains the - .class files after build.xml has been run (classes/). - opendmap.jar file (distribution/).
docs/	Contains the - JavaDocs for the OpenDMAP classes (api/). - OpenDMAP Pattern manual (pattern-manual/), which is a manual describing the construction of OpenDMAP patterns. Also included in this directory are files containing the example patterns, text, and ontologies used in the pattern manual. - OpenDMAP Testing Manual (testing-manual/), which describes the testing architecture that is also distributed in this release. - This OpenDMAP User Manual (user-manual/), which describes the installation, use, and extension of OpenDMAP.
lib/	Contains external libraries. Notes: - The xercesImpl.jar file is included here in this distribution. - You must download the protege.jar file and add the looks.jar, (as described in Section 2.1.1 above), and place it in the lib/ directory in order to use the example applications.
license/	Contains the license for OpenDMAP as well as the Xerces project.
logs/	This directory is created by Protégé for storing log files.
test/	Contains six XML files containing tests for OpenDMAP conforming to the testing architecture included in this distribution. Notes: - See build.xml for a demonstration of executing these tests. - By default, all tests are executed during the build procedure. - Refer to the Testing Manual in /docs for further details
test/projects/	Contains the example pattern files and Protégé projects used by - the GenerifConsole application (generif/) - the LoveHateConsole application (people-loving-people/) - the test suites (newspaper/ and test/)

2.1.3. Building the Project

If you modify the source files, the ant/build.xml file can be used to compile and repackage the project, as well as run tests on OpenDMAP. The runBuild.sh script in the ant/ directory will allow you to build this project from the command line. In order to use build.xml, you will need to have Ant installed on your system (http://ant.apache.org/).

Edit the JAVA_HOME and ANT_HOME variables within the script according to your system prior to running runBuild.sh.

2.1.4. Generating the Pattern Syntax Classes

The OpenDMAP Pattern Manual provides a detailed explanation of the pattern syntax used to create concept-based patterns. (Note that "syntax" here does not mean linguistic syntax, but rather, the regular-expression language used to make patterns).This syntax is defined in the pattern syntax file src/edu/uchsc/ccp/opendmap/pattern/DMAPPatternParser.jjt. Altering OpenDMAP pattern syntax requires knowledge of JavaCC, a Java parser-generator, and JJTree, a syntax-tree generator that uses JavaCC parsers. The pattern classes are generated using JavaCC and JJTree. If you alter the pattern syntax file DMAPPatternParser.jjt, the pattern classes can be recompiled using the create-pattern-files task located in Ant file build.xml. To create these classes you will need to download JavaCC (see Section 7) and set the javacc.home property in build.xml to reflect the appropriate path on your system.

2.2. Example Projects

Two example applications are included in this distribution. Interacting with the applications and examining the source code are both good places to start if you are interested in creating your own OpenDMAP application.

2.2.1. The GenerifConsole Example Application

The GenerifConsole is a skeleton system designed to extract protein transport relations from GeneRIFs (see http://www.ncbi.nlm.nih.gov/projects/GeneRIF/GeneRIFhelp.html for a definition) or from sentences. This application works in a limited capacity, as the protein names that it will recognize have been hard-coded within the OpenDMAP pattern file test/projects/generif/generif-entity.patterns.

To run the GenerifConsole, first edit ant/runGenerifConsole.sh and set your JAVA_HOME variable. Then execute runGenerifConsole.sh from the ant/ directory. Try the following sentence as input to see a demonstration of OpenDMAP output (other example input texts can be found in the test/projects/generif/generifs.txt file):

Lim-kinase 1 is translocated by Src from the nucleus to the cytoplasm.

You will see output to screen similar to Figure 2. The first block of information directly underneath the line you typed shows a concept that was matched to one of the protein patterns. The first line gives the frame that was matched: {i-lim-kinase}. The pattern that matched this text is in file /test/projects/generif/language/generif-entity.patterns, and is shown here:

{i-lim-kinase} := LIM-kinase 1;

The second line starts with the concept that was found, followed by span information in parentheses. Span refers to the location of the concept in text. The information provided in the output in Figure 2 corresponds to *(token start position – token end position – number of tokens ignored*). The *number of tokens ignored* refers to tokens that may have been skipped over during the matching process if a wildcard element in a pattern allowed. The third line shows the matched text in quotes.

In the second block of information, the first line again shows the frame of the pattern that matched, with the frame slots filled in with the extracted information. The pattern that matched is in the file /test/projects/generif/language/generif.patterns, and is shown here:

```
{c-transport} := [patient] is|are|was|were [action c-action-transport-passive] @
                 (by the? [agent]) @ (from the? [source]) @
                 (to|toward|towards|into the? [destination]);
```

The second line lists the matched concept and its span information, followed then by the text that matched this pattern. What follows next is analogous information about the three slots – *action*, *source*, and *destination* – that were filled in matching the {c-transport} pattern.

Figure 2: Sample output from runGenerifConsole.sh

```
Welcome, please enter a GeneRIF:
> lim-kinase 1 is translocated by Src from the nucleus to the cytoplasm.

{i-lim-kinase}                                                    \
i-lim-kinase (0-1-0) Matches=2 Span=2                              |   →1st block of information
  "lim-kinase 1"                                                  /

{c-transport [action]={c-action-transport-two-way} [source]={c-nucleus} [destination]={c-cytoplasm}}
c-transport (2-8-0) Matches=7 Span=7                               \
  "translocates from the nucleus to the cytoplasm"                  \
  action = c-action-transport-two-way (2-2-0)                        |   → 2nd block of information
  source = c-nucleus (5-5-0)                                        /
  destination = c-cytoplasm (8-8-0)                                /
```

2.2.2. The LoveHateConsole Example Application

The LoveHateConsole is an example application to illustrate how OpenDMAP references can be processed to extract their meaning and generate an appropriate response.
It implements a very simplistic processor for 'Loves' and 'Hates' relationships as defined in the 'people-loving-people' sample Protégé project.

To run the LoveHateConsole, first edit ant/runLoveHateConsole.sh and set your JAVA_HOME variable. Then execute runLoveHateConsole.sh from the ant/ directory. When prompted by the application, enter sentences similar to the following:

> Does Mary love John?
> Does John hate Pat?

OpenDMAP processes the language of the question you typed in by matching patterns from the love-hate-query.patterns file (in path /test/projects/people-loving-people). Then based on knowledge in the people-loving-people ontology (same path, people-loving-people.pprj), OpenDMAP finds the information to answer the question, and a natural language response is generated and output to screen.

3. How OpenDMAP works – a walk through the GeneRIFConsole transport example

Presented here is a run-time walk-through of how OpenDMAP works, using the GeneRIF Console Example that is distributed with OpenDMAP. The purpose of this walk-through is to introduce key concepts necessary to understand the steps to implement a custom OpenDMAP project.

When you run the runGenerifConsole.sh script, a `GenerifConsole` (from package edu.uchsc.ccp.opendmap.example) is invoked and is passed the name of a configuration file, in this case test/examples/generif/configuration.xml. The configuration file tells the system where to find the ontology and pattern files. The `GenerifConsole` invokes a `GenerifProcessor` (same package), which in turn invokes a `Parser` (package edu.uchsc.ccp.opendmap) that is passed a `DMAPTokenizer`.

The `Parser` is the key workhorse component of OpenDMAP. It loads into memory the patterns from the pattern files (generif-entity.patterns, generif.patterns, and ptrans.patterns in this example) and frames from the ontology file (generif.pprj). Frames are structures of data, sometimes called *classes*, which have *slots* that represent properties of the frame. For more information about frame-based representation, refer to www.protege.stanford.org. The `Parser` communicates with the `DMAPTokenizer` to get the input string.

When the prompt comes up and you type in the string of text, “Lim-kinase 1 is translocated by Src from the nucleus to the cytoplasm,” the `Parser` examines the input string, looking for matches to the patterns. It does this by creating a `Reference` for each token (and pre-tagged entity, if there are any) that the `DMAPTokenizer` gives it,

as well as a Reference for each pattern element. A Reference is a record of something the Parser has recognized in the input string or is looking for from the pattern files. Input text References and pattern References are compared, and when elements from the two types of References match, the Parser creates yet another Reference that indicates a matched pattern.

Continuing our example, let's delve a little deeper to examine the Parser behavior. Given a string of text, the Parser reads in the first token, *Lim-kinase*, and creates a Reference for this token. It creates a list of patterns in which this Reference could play a role. This list is called the prediction list. The Parser recognizes that the Reference created for *Lim-kinase* could play a role in the {i-lim-kinase} concept pattern (in generif-entity.patterns),

{i-lim-kinase} := LIM-kinase 1;

and therefore adds this pattern to the prediction list. Now the Parser reads in the second token, *1*, makes a Reference for this token, and recognizes that this new Reference, combined with the previous Reference for *Lim-kinase*, fulfills a pattern in the prediction list, namely, the pattern for the concept {i-lim-kinase}. A new Reference is made for the recognized concept {i-lim-kinase}. The Parser further recognizes this {i-lim-kinase} concept Reference as a potential filler for the [patient] slot in a {c-transport} pattern (in generif.patterns)

{c-transport} := [patient] is|are|was|were [action c-action-transport-passive]
@ (by the? [agent]) @ (from the? [source])
@ (to|toward|towards|into the? [destination]);

so the parser adds this pattern to the prediction list. As the Parser reads in more words from the text you typed in, it recognizes those words to be elements of a pattern in the prediction list, and updates the prediction list by removing patterns that conflict with the text, or adding other patterns that the text could play a role in. In this case, there is only one possible pattern, and the words the Parser is reading conform to that pattern, so the prediction list of frames remains the same until the end of reading in this string.

At this point OpenDMAP has stored the concepts it found – in this case the {c-transport} concept and its slot fillers - in Reference objects, and OpenDMAP's job is done. Now an external application can access the information stored in the References. In this example application, the information is accessed and printed to screen.

4. Getting your own project off the ground

4.1. Necessary Ingredients

Running OpenDMAP requires these components to be in place:

- Protégé Ontology: frame-based representation of the knowledge to be extracted
- Pattern file: file(s) that contains the patterns OpenDMAP will find
- Configuration file: xml file that informs OpenDMAP where the resources are

- DMAPTokenizer: object that returns a sequence of DMAPTokens for input to the Parser. Use either the DefaultTokenizer that comes with OpenDMAP or a customized DMAPTokenizer that you build
- Parser: object responsible for executing the matching mechanisms
- An external application that uses the extracted information

4.2. Protégé Ontology

OpenDMAP uses Protégé ontologies to provide object models for the possible concepts that might be found in text. For each concept (i.e. *class*) of interest in an ontology, the user creates patterns that OpenDMAP uses to find those concepts in text. The ontology file(s) can be placed anywhere in your directories. In the configuration file (see Section 4.4) you provide OpenDMAP the information about where to find your ontology file(s). For more information about frame-based representation and Protégé in particular, refer to the Protégé website at www.protege.sourceforge.net. For more information about how the patterns correspond to the Protégé ontology, refer to the OpenDMAP Pattern Syntax Manual.

4.3. Pattern Files

The user creates patterns for concepts of interest in the ontology. The pattern file(s) can be placed anywhere in your directories. In the configuration file (see Section 4.4) you provide OpenDMAP the information about where to find your pattern file(s). OpenDMAP patterns use a particular syntax. Please refer to the OpenDMAP Pattern Manual for instruction on writing patterns. For more information about how the patterns correspond to the Protégé ontology, refer to the OpenDMAP Pattern Syntax Manual.

4.4. Configuration File

The OpenDMAP configuration file informs the `Parser` (see Section 4.5) of the configuration variables. It contains XML declarations that describe these variables to the `Parser`. The syntax for the configuration file is shown in Figure 2. An explanation of each of the configuration file components follows.

Figure 2: The configuration file format

```
<Configuration>

      <RetainCase> boolean </RetainCase>

      <ProtegeProject name=projectName>
            <ProjectFile> filename </ProjectFile>
            <PatternFile> filename </PatternFile>
            <PatternSlot root=frameName> slotName </PatternSlot>
      </ProtegeProject>

      <Association type="isa">
            <Parent project=projectName1 frame=frameName1 />
            <Child project=projectName2 frame=frameName2 />
      </Association>

</Configuration>
```

The parser configuration must be wrapped in `Configuration` tags.

The `RetainCase` XML element tells the `Parser` whether it should be case-sensitive when matching. To cause the parser to match in a case-sensitive way, set the Boolean parameter to `true`; for case-insensitivity, set it to `false`. The default value for `RetainCase` is `false`.

The `ProtegeProject` XML element tells the `Parser` information about the Protégé Project and pattern files it will be using. The `name` attribute is required. It is used to assign this project definition a name within the configuration file, which can then be used in `Association` elements (see below) to create links between Protégé projects. At least one `ProtegeProject` element must be defined in the configuration file; there may be multiple `ProtegeProject` elements defined.

Within the `ProtegeProject` element, at least one `ProjectFile` element must be defined. This element specifies the location of the Protégé project .pprj file that will be loaded. The location is defined with a relative path in relation to the location of the configuration file. It is an error for this element to be missing.

`PatternFile` elements specify the location of pattern files to be loaded with the Protégé project. There can be one or more `PatternFile` elements defined per `ProtegeProject` element. Like the `ProjectFile`, the path is relative in relation to the configuration file. The concepts modeled as patterns in the pattern file(s) must correspond to concepts in the ontology project file(s) within this `ProtegeProject`. It is an error for a pattern file to reference a concept that does not exist in the ontology project file(s) for this `ProtegeProject`.

`PatternSlot` elements specify the names of Protégé slots that hold OpenDMAP patterns. There can be zero, one, or multiple `PatternSlot` elements defined per `ProtegeProject`. The `root` attribute is specified by equating it to a frame name (i.e. class name) in the Protégé project. OpenDMAP will look for slot patterns on and under that frame. If the `root` is not specified, it defaults to the entire Protégé project, meaning OpenDMAP will look for the specified slot on all frames. (See the GeneRIFConsole example.)

The `Association` element tells the parser about relationships between frames in separate Protégé projects. Associations allow separate Protégé projects to act as a single project within OpenDMAP. The `type` attribute is required and currently must take the value `isa`, meaning the association link between two Protégé projects' frames is an is-a relationship.

The `Parent` and `Child` are required elements that specify the parent and child frames in the is-a relationship. They both have the same two required attributes: `project`, which names one of the `ProtegeProject` elements in the configuration file (defined by the `ProtegeProject name` attribute), and `frame`, which names one of the frames in the respective Protégé project. Once an association between two frames is created, OpenDMAP will treat the frames as if they were part of a single Protégé project class hierarchy.

For an example configuration file, see the test/projects/generif/configuration.xml file. This file tells OpenDMAP to configure a case-insensitive `Parser`, which will then load two Protégé projects, one called "language" whose Protégé project file is located at language/generif.pprj, and the second called "ptrans" whose Protégé project file is located at ptrans/ptrans_cellular_component.pprj. The `Parser` will get patterns for the language project from the files generif-entity.patterns and generif.patterns, both in the language/ directory. It will get the ptrans project pattern file from ptrans/ptrans.patterns. In addition, the association tells the parser to treat all c-cellular-component frames in the ptrans project as if they were also c-cell-component frames in the language project. This means that whenever OpenDMAP recognizes a c-cellular-component frame (or one of its children) using patterns from the ptrans project, it will also recognize that frame as a c-cell-component frame that can match patterns in the language project.

4.5. Parser

As mentioned previously, the `Parser` is the workhorse of OpenDMAP: it is the component that drives the matching mechanism. Pseudocode for instantiating and using a `Parser` is provided in Appendix A. The `ParserFactory.newParser (configFilename, TRACELEVEL)` method instantiates a new `Parser` (line 13). The `configFilename` variable is the path to the configuration file (see Section 4.4). The `TRACELEVEL` variable is responsible for setting the level of tracing messages that get output while OpenDMAP is running. The possible values range from *OFF* (no messages) to *FINEST* (many messages, including a list of the patterns added to the prediction list). Refer to the class edu.uchsc.ccp.opendmap.DMAPLogger for the full list of `TRACELEVEL` values.

To begin the pattern matching process, a string of text is passed to a `DMAPTokenizer` (line 24, and see Section 4.6 re `DMAPTokenizers`). This `DMAPTokenizer` is then passed to the `Parser.parse(DMAPTokenizer)` method (line 27). Information about concepts in text that match patterns is deposited into `Reference` objects by the `Parser`.

4.6. DMAPTokenizer

The input to an OpenDMAP `Parser` is a `DMAPTokenizer`. The role of the tokenizer is to convert words (and other entities, if marked up in preprocessing) from raw text into `DMAPToken` objects. `DMAPTokens` minimally hold information about the text and span of the word or entity. Additionally, they can store information about class membership, slot information, additional span data, etc. The tokenizer's purpose is to provide these `DMAPTokens` to the parser in a particular order (see below for more discussion on the proper order of `DMAPTokens`). Be aware that the term 'token' as used in the code library is overloaded to mean both single words normally described as tokens, as well as multiword entities that, for the purposes of OpenDMAP, act as single entities. For clarity's sake, in this manual 'token' is defined as any single word or multiple-word item that will be passed to OpenDMAP as a single `DMAPToken` (see Figure for an example of a multiword token).

OpenDMAP comes with a simple tokenizer called the `DefaultTokenizer` (package edu.uchsc.ccp.opendmap) that tokenizes text on whitespace and punctuation, with the following rules:

- The period character (.) is treated as a delimiter if it follows an alpha character, and is removed. In all other environments, (following a digit, between alpha or digit strings, etc.), the tokenizer does not recognize the period as a delimiter and leaves it in place.
- The forward slash (/) and n dash (-) characters are never recognized as tokenization delimiters and are left in place.
- The parentheses characters (( )) and the single quotation character (') are always recognized as tokenization delimiters, but are not removed. Rather, they are recognized as single character tokens in and of themselves.
- All other punctuation characters are recognized as delimiting characters during tokenization and are removed.

The `DefaultTokenizer` does not convert pre-tagged entities into `DMAPToken`s.

Most likely you will want to include more than just strings of tokens in your OpenDMAP patterns, for instance, genes, species mentions, part-of-speech tags, syntactic information, etc. This can be accomplished in two ways: create OpenDMAP patterns to recognize such entities of interest, or pre-process the text with appropriate tools such as named entity taggers, a species extractor, a part-of-speech tagger, a syntactic parser, etc. (These two methods are not mutually exclusive). Recall that the `DefaultTokenizer` does not convert pre-tagged entities into `DMAPToken`s. If you use pre-processors of any kind you will need to implement a custom tokenizer that both tokenizes the input text and converts tokenized words as well as the pre-tagged entities into `DMAPToken`s.

To implement a custom tokenizer, extend the abstract `DMAPTokenizer` class. Pseudocode that lays a framework for completing this is provided in Appendix B. Implement the inherited methods `hasNext()`, `next()`, and `iterator()`. The `hasNext()` method (lines 12-15) returns `true` if there is another token left in the

Figure 3: Correct order of text tokens returned by `next()` method. The final character offset value of the token ‘the’ is 3, so that token is returned first. The final offset value of the token ‘endoplasmic’ is 15; it is the second token returned. The final offset value of the token ‘reticulum’ is 25; it is returned third. The final offset value of the multiword token ‘endoplasmic reticulum’ is also 25. Since this value is not less than the final offset value of the token ‘reticulum’, ‘endoplasmic reticulum’ is returned fourth. Finally, the token ‘showed’ is returned last.

Text:	The endoplasmic reticulum showed …			
Tokenized words:	`The`	`endoplasmic`	`reticulum`	`showed`
Character offsets:	`000` `123`	`00000111111` `56789012345`	`111222222` `789012345`	`222333` `789012`
Token offsets:	0	1	2	3
Tagged entities:		`endoplasmic reticulum`		
`next()` return order:	1-[The] 2-[endoplasmic] 3-[reticulum] 4-[endoplasmic reticulum] 5-[showed] …			

input string to pass to OpenDMAP. The `next()` method (lines 17-31) is responsible for converting the text, whether single words or multiple word entities, into `DMAPToken` objects in the correct order. This order is based on the final character offset value of the tokens being returned by `next()`: if the final character offset of $token_i$ is less than the final character offset of $token_j$, then return $token_i$ first. For an example, see Figure 3.

For the OpenDMAP `Parser` to keep track of where it is in the input text, each `DMAPToken` must have its ordinal token position values set via the `setStart(int)` and `setEnd(int) DMAPToken` methods (lines 22-23). For `DMAPToken`s that are one word long, the start and end position values will be the same. For instance, using the snippet text from Figure 3, the start and end position for the token "The" are both 0. The start and end position for the token "endoplasmic" are both 1. For the token "endoplasmic reticulum", the start position is 1 and the end position is 2.

4.7. External application that uses the extracted information

The `Parser` stores information about all the concepts that matched patterns in `Reference` objects. There are two kinds of `Reference` objects: subsumed and unsubsumed. Subsumed references are those matched concepts that are themselves parts of larger matched concepts. Unsubsumed references are those matched concepts that do form a part of another larger concept. For instance, take again the example from Figure 2:

Lim-kinase 1 is translocated by Src from the nucleus to the cytoplasm.

Table 1 shows some examples of subsumed and unsubsumed `References` taken from the output of this GeneRIF processed with OpenDMAP. Please note that

Table 1 does not list every subsumed `Reference` of the GeneRIF.

Table 1: Subsumed versus Unsubsumed `References`.

	Text	Concept
Subsumed	lim-kinase 1	{i-lim-kinase}
	is translocated	{c-action-passive}
	cytoplasm	{c-cytoplasm}
	to the cytoplasm	{destination}
Unsubsumed	Lim-kinase 1 is translocated by Src from the nucleus to the cytoplasm	{c-transport}

All `References` are available to an external application by having it call the `Parser.getReferences()` method. Information stored in the `References` is

available by standard `getter` methods. For an example of an external application accessing the information in `References`, see Appendix A lines 29-35.

5. Integration into UIMA: a good idea

OpenDMAP can be used in isolation, as described in this manual. However, it is strongly suggested you consider adopting a middleware architecture into which you plug OpenDMAP, the various text-preprocessing applications you may be using, and post-processing applications that make use of OpenDMAP output. While any middleware architecture will work, we suggest adoption of the Unstructured Information Management Architecture (UIMA), available at http://incubator.apache.org/uima/downloads.html. The reason for this endorsement is there are extant biomedical text processing UIMA resources available at our BioNLP-UIMA Component Repository SourceForge site http://bionlp-uima.sourceforge.net/, and more planned for future development. Included in the repository is a wrapper to plug OpenDMAP in to UIMA. Also available are wrappers for biological applications like gene and mutation taggers, and linguistic applications like sentence detection and tokenization. Additionally, other groups have and are developing UIMA resources for text mining tasks (see Section 7 for a list of publications and resources).

6. Citing OpenDMAP

Please site the following publication when referring to your use of OpenDMAP:

Hunter, L; Lu, Z; Firby, JR; Baumgartner, WA, Jr.; Johnson, HL; Ogren, PV and Cohen, KB. OpenDMAP: **An open-source, ontology-driven concept analysis engine, with applications to capturing knowledge regarding protein transport, protein interactions and cell-type-specific gene expression**. BMC Bioinformatics. 2008, 9:78.

Please refer to the list of relevant publications and other resources in Section 7 that addresses OpenDMAP performance and other related resources.

7. Bibliography and Resources

Publications about OpenDMAP applications:

Hunter, L; Lu, Z; Firby, JR; Baumgartner, WA, Jr.; Johnson, HL; Ogren, PV and Cohen, KB. OpenDMAP: **An open-source, ontology-driven concept analysis engine, with applications to capturing knowledge regarding protein transport, protein interactions and cell-type-specific gene expression**. BMC Bioinformatics. 2008, 9:78.

Baumgartner, WA Jr.; Lu, Z; Johnson, HL; Caporaso, JG; Paquette, J; Lindemann, A; White, EK; Medvedeva, O; Cohen, KB and Hunter, L **Concept recognition for**

extracting protein interactions from biomedical text. Genome Biology BioCreative II Special Issue (in press).

Baumgartner, WA Jr.; Lu, Z; Johnson, HL; Caporaso, JG; Paquette, J; Lindemann, A; White, EK; Medvedeva, O; Cohen, KB and Hunter, L. **An integrated approach to concept recognition in biomedical text**. In BioCreative workshop proceedings. Madrid, Spain. 2007.

Lu, Zhiyong. **Text Mining on GeneRIFs**. PhD thesis, Computational Bioscience Program, University of Colorado School of Medicine, CO, USA, 2007.

Publications about external resources used with OpenDMAP:

Noy, NF; Crubézy, M; Fergerson, RW; Knublauch, H; Tu, SW; Vendetti, J and Musen, MA. **Protege-2000: an open-source ontology-development and knowledge acquisition environment**. AMIA Annual Symposium Proceedings. 2003.

Ferrucci, D; Lally, A. **UIMA: an architectural approach to unstructured information processing in the corporate research**. Natural Language Engineering. 2004, 10(304).

Links to resources mentioned in this manual:

Ant:	http://ant.apache.org/index.html
Java 1.5:	http://java.sun.com/javase/downloads/index_jdk5.jsp
JavaCC:	https://javacc.dev.java.net/
Java CC JJTree doc:	https://javacc.dev.java.net/doc/JJTree.html
OpenDMAP:	http://opendmap.sourceforge.net
Protégé:	http://protégé.sourceforge.net
UIMA:	http://incubator.apache.org/uima/
UIMA Components for BioNLP:	http://bionlp-uima.sourceforge.net
Other BioNLP resources:	http://bionlp.sourceforge.net

Appendix A: Pseudocode for instantiating and using a `Parser`

```
1   Public class runOpenDMAP {
2
3       Public static void main(String[] args) {
4
5           String line = //line of raw text
6           String configFilename = //path to configuration file
7
8           while ((line != null) {
9               // instantiate a parser
10              Level TRACELEVEL = Level.OFF;
11              Parser parser = null;
12              try {
13                  parser = ParserFactory.newParser(configFilename, TRACELEVEL);
14              } catch (ConfigurationException ce) {
15                  // Bad configuration
16                  ce.printStackTrace();
17                  return;
18              }
19
20              // Reset the parser
21              parser.reset();
22
23              //instantiate a tokenizer, pass it the line of text,
24              DefaultTokenizer dt = new DefaultTokenizer(line, true);
25
26              // Parse the utterance
27              parser.parse(dt);
28
29              // An external application accesses the concept
30              // matches from the References
31              Collection<Reference> all = parser.getReferences();
32              System.out.println("\nAll references:");
33              for (Reference r: all) {
34                  System.out.println("ALL: " + r.getStart() + ".." + r.getEnd() + " "
35                                                  + r.getReferenceString());
36              }
37          }
38      }
39  }
```

Appendix B: Pseudocode for a Custom `DMAPTokenizer`

```
1  public class CustomTokenizer extends DMAPTokenizer {
2          // instantiate variables String inputText, String tokenText, int charBegin, int charEnd
3
4      public CustomTokenizer(String inputText) {
5          //set all token span values to 0
6          //check for the existence of the first (word) token
7          if ( ! findToken() ) {
8              tokenText = null;
9          }
10     }
11
12     // is there another 'token' to return?
13     public boolean hasNext() {
14         return ( tokenText != null );
15     }
16
17     // return next token loaded into a DMAPToken object
18     public DMAPToken next() {
19         if (tokenText != null) {
20             DMAPItem dmapItem = new TextItem(tokenText);
21             DMAPToken dmapToken = new DMAPToken(dmapItem, charBegin, charEnd);
22             dmapToken.setStart(tokenStart);
23             dmapToken.setEnd(tokenEnd);
24             // reset token count span info for next token
25             tokenStart = tokenEnd + 1;
26             if ( ! findToken() ) {
27                 tokenText = null;
28             }
29         }
30         return dmapToken;
31     }
32
33     // the method where the tokenizing gets done
34     private boolean findToken() {
35         /* Implement code that checks for the next valid "token" in the text. This "token" could be
36          * the next word-token in the input text, or it could be the next word or multiword string
37          * that has been tagged by a preprocessor.
38          * if a valid "token" is not found,
39          *    return false;
40          * else
41          *    A "token" string has been found. Set the charBegin and charEnd values.
42          *    tokenText = inputText.substring(charBegin, charEnd);
43          *    DMAPToken dmapToken = new DMAPToken(tokenText, charBegin, CharEnd)
44          *
45          *    return true;     */
46     }
47 }
```