

Complete User Guide (Draft 4.1.0)

This document is an incomplete draft, and is under active revision. Please check back for updates. The [Cypher Developers mailing list](#) provides a forum for discussion and questions.

This file is also available in [PDF form](#).

Table of Contents

[1 Introduction](#)

[1.1 What is Cypher?](#)

[1.2 Fun and Easy](#)

[2 Installation](#)

[2.1 Requirements](#)

[2.2 Set up](#)

[3 Getting Started](#)

[3.1 Startup Properties](#)

[3.2 Output and Reports](#)

[3.3 Running the Web Service](#)

[3.4 API](#)

[4 Hello World Example](#)

[4.1 The Dataset](#)

[4.1.1 Pattern files and index.xml](#)

[4.1.2 Con Element \(The Word/Phrase List\)](#)

[4.1.3 Pattern Element \(The Grammar\)](#)

[4.1.4 Container Element](#)

[4.1.5 mso:Sense Class \(The Vocabulary\)](#)

[4.1.6 Frame Semantics \(The Dictionary\)](#)

[5 Additional Topics](#)

[5.1 Paraphrases](#)

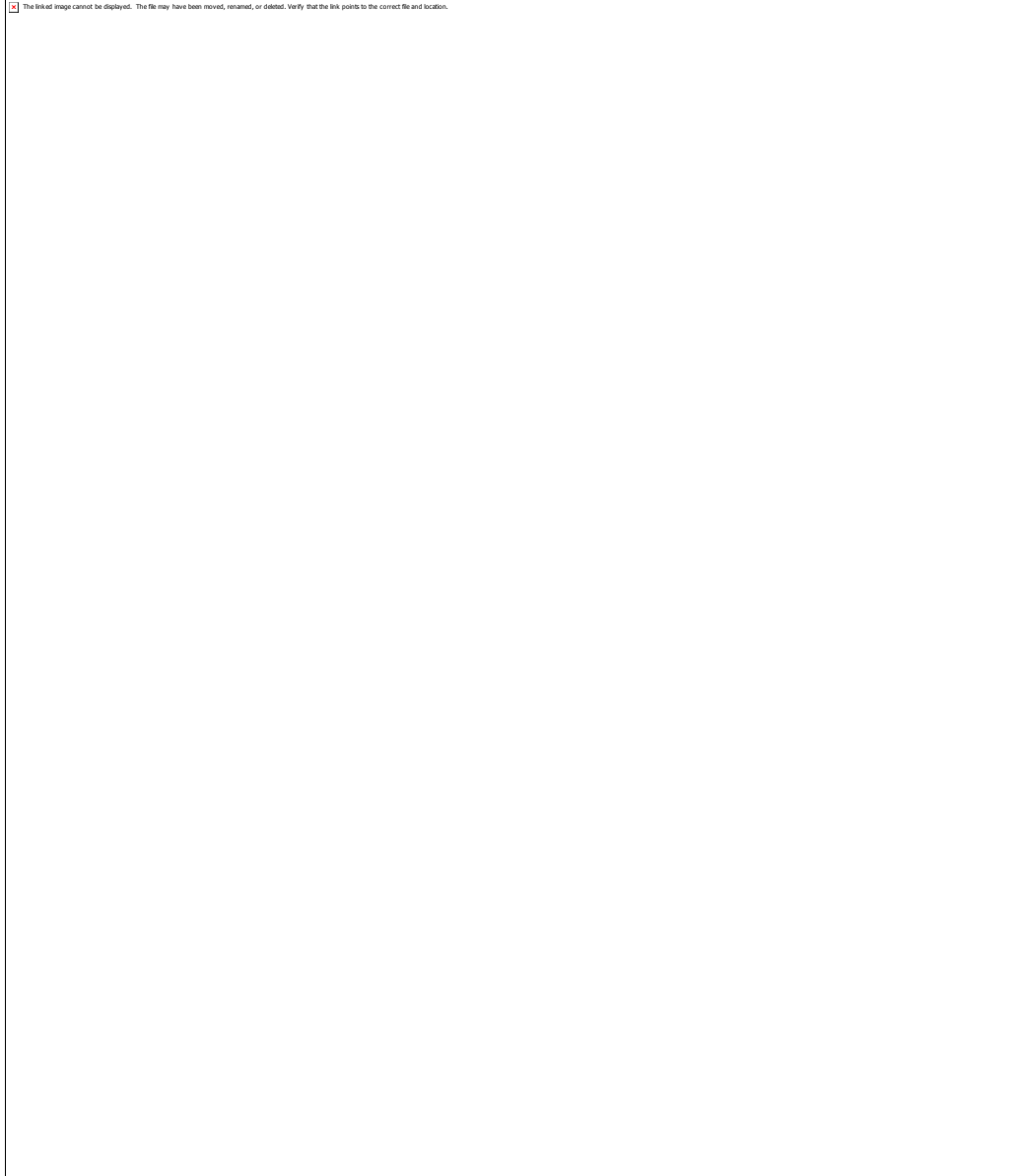
[6 FAQ](#)

Introduction

What is Cypher?

Welcome to Cypher! Cypher is one of the first software program available which generates the RDF graph and SeRQL query representations of a natural language input. The Cypher framework provides a set of robust definition languages, which can be used to extend and create grammars and lexicons. The Cypher specifications are designed to allow a novice to quickly and easily build transcoders for processing highly complex sentences and phrases of any natural language, and to cover any vocabulary.

Cypher is a 100% Pure Java application, and runs on any machine that supports Java. The specification languages used to describe the grammar, lexicon, and framenet is all XML.



Example Input/Output using English Language Package

FOAF Triples

Input 1 *The Semantic Web is one of Chris Walker's hobby interests*

Input Type English Declarative Clause

Abbreviated Output `mso:CWalker foaf:interest mso:SemanticWeb`

Output Type `.rdf` File

Dataset Download (coming soon)

Input 2	<i>Chris Walker's interest</i>
Input Type	Relational Noun Phrase
Abbreviated Output	<code>select distinct node0 from {mso:CWalker} foaf:interest {node0}</code>
Output Type	.serql File
Dataset	Download (coming soon)

Frame Triples

Input 3	<i>The Terminal stars Tom Hanks and Catherine Zeta-Jones</i>
Input Type	English Declarative Clause
Abbreviated Output	<code>mso:frame0 rdf:type mso:MovieRoleFrame mso:frame0 mso:actor mso:TJeffreyHanks-Actor mso:frame0 mso:actor mso:CZetaJones-Actress mso:frame0 mso:movie mso:TheTerminal-Movie</code>
Output Type	.rdf File
Dataset	Download (coming soon)

Input 4	<i>Actresses who played in movies with Tom Hanks</i>
Input Type	English Anaphora Noun Phrase
Abbreviated Output	<code>select distinct node0 from {mso:frame0} rdf:type {mso:MovieRoleFrame} {mso:frame0} mso:actor {mso:TomHanks-Actor} {mso:frame0} mso:actor {node0} {mso:frame0} mso:movie {node1} {node0} rdf:type {mso:Actress} {node1} rdf:type {mso:Movie}</code>
Output Type	.serql File
Dataset	Download (coming soon)

Fun and Easy

There are only three steps required to produce the above output: 1) describe the phrase structure of each sentence, enter the words in Cypher's lexicon (dictionary), and create the RDF schema (if needed). Cypher is designed to allow developers to build on the works of others. All Cypher data constructs are pluggable and reusable objects. Once you've described a `PresentTenseVerbPhrase`, it can be used by you or by others to create more complex phrases. And, a third-party developed lexicon can be added to your existing Cypher installment to instantly add new words to its vocabulary without any further modification to Cypher's grammar. Cypher's vocabulary effects the amount and quality of the semantic output. Programming with Cypher is fun to do, and with only three XML elements you must learn to write a grammar: `<con/>`, `<pattern/>`, `<mso:sense/>`, Cypher has a smooth learning curve. This technology is the result of over five years of research and development. The work has turned out to be, what is in our opinion, a novel and extremely elegant yet powerful approach to a very old problem in computer science. Our vision for Cypher is to create an open framework to facilitate the development of datasets which cover the majority of phrase patterns and vocabularies for the majority of languages; this is a gargantuan, but finite undertaking, and it all begins with you. So without further adieu, we present Cypher. Enjoy.

Installation

Requirements

- [Java](#) JDK 1.5 or higher; **note: be sure to install the full JDK, not just the JRE, because Cypher will need access to the javac process**
- [Sesame](#) 2 or higher
- 1 GB RAM (Recommended)
- Operating System: Windows, Linux, OSX

Set up

- Download and decompress the [Cypher distribution file](#) for your operating system
- Set the `CYPHER_HOME` environment variable to the Cypher root directory. For example, if you unzipped the `cypher.zip` file to `C:/`, set `CYPHER_HOME` to `C:/cypher`
- Run Sesame and create four repositories for Cypher to use: semantic output, internal use, lexicon, and framenet dictionary;
- Browse to the `<cypher home>/data/configuration/config.ini` directory
- Set the Sesame repository names, the repository url and the repository login information in the, use the default settings as an example:
 - `rdf.repository.url=http://localhost/openrdf-sesame`
 - `rdf.repository.username=foo`
 - `rdf.repository.password=bar`
 - `rdf.repository.lexicon=mySesameLexicon`
- Set the location of the input file containing the text to be transcoded, and the destination output file where the output will be written. Example:

- cypher.input=C:/cypher/in
 - cypher.output=C:/cypher/out
- Run `run.bat` (Windows) or `run.sh` (Linux)
- The console will report when the input has been transcoded. When all input has been processed, the program ends.
- Note: there are generated [several report files and four output file types](#):
 - .xml - contains the grammar parser output
 - .rdf - contains the RDF statements (e.g. representing a clause)
 - .sparql / .serql - contains a RDF query (e.g. representing a noun phrase)

Getting Started

Cypher conforms to a process called Transcography. The main focus of Transcography is to provide a minimalist framework for NLP, i.e. it is averse to first order logic, predicate calculus, neural nets, etc. It produces quality output by resolving ambiguity at multiple levels of processing, and allowing the interaction of processes to improve parse judgments.

Transcography organizes and modulates the tasks in NLP in such a way as to “corner the beast” of ambiguity almost solely into a lexicon which is based on frame semantics. Once the phrase grammar has been produced for the input, five main techniques are used to produce the semantic output: **Symbolic Reference**, **Node Expansion**, **Subcategorization**, **Identity Transfer**, and **Inference**.

Symbolic Reference – Each constituent of a phrase must resolve to a concept, referenced either by description (e.g. *the blue bird*) or unique identifier (i.e. *Henry Ford*). Cypher, therefore, produces either a URI or BNode which represents the phrase, plus a set of triples representing the description given by the phrase.

Node Expansion – The set of triples produced by each child node of a phrase is included in the parent phrase’s output.

Subcategorization – Transcography conforms to the theory that verbs and other units of language subcategorize for their arguments, and that this information is specified in the lexicon. Cypher accommodates this with the [mso:Sense](#) class’ properties: [mso:allows](#), [mso:restricts](#), [mso:requires](#).

Identity Transfer – The human language processor produced semantic output by consulting a dictionary, and retrieving an entry for each word encountered in the input. The entry contains the description of an anonymous entity, and this description is transferred to the instance concept. Cypher accommodates this with the [mso:Sense](#) class’ property [mso:ref](#).

Inference – Each phrase and clause in natural language expresses information not explicit in the phrase. The human language processes makes use of a dictionary which provides a semantic map, linking the explicit description provided by the phrase, to

implicit descriptions inferred from the phrase. Cypher accommodates this by using [Frame Semantics](#).

Startup Properties

The program startup properties are located in `<cypher home>/data/configuration/config.ini`. Following is a description of each property:

Property Name	Description	Value	Required
rdf.repository.*	<p>Properties of this pattern specify a configuration string for a Sesame Repository. This allows for the use of SAIL and Repository implementations from third-parties.</p> <p>The syntax is: <code><classname>,<args>...</code></p> <p>where <code><classname></code> is the fully qualified name of a class in the classpath which is an implementation of the org.openrdf.repository.Repository. The .jar file containing this class should be placed in the <code><cypher home>/lib</code> directory. The <code><args></code> is a list of Strings which will be passed (in the order in which they are listed) to the constructor method of the Repository class.</p>	string	N/A
rdf.repository.output	The repository used to store semantic output	string	YES
rdf.repository.lexicon	The repository used to store the lexical data	string	YES
rdf.repository.dictionary	The repository used to store the framenet data	string	YES
rdf.repository.internal	The repository used for internal processing	string	YES
cypher.language	The directory under <code><cypher home>/data/language</code> containing Cypher's language definition files	string	YES
cypher.input.files	Whether the files in cypher.input.dir will be loaded as input	boolean	YES
cypher.input.dir	The absolute path to the directory containing input	path string	YES
cypher.output.dir	The absolute path to the directory to which the output is written. If the Cypher web service is running, then set the <code>cypher.output.dir</code> to the <code><cypher-ws home>/out</code> directory, where <code><cypher-ws home></code> is the location where the	path string	YES

	cypher-ws.war file was deflated.		
cypher.output.format	The RDF format for output. Options are rdfxml, turtle, n3, ntriples, trix, trig	string	NO
cypher.output.commit	Whether to commit RDF output to the cypher.repository.output repository	boolean	NO
cypher.output. entailments	Whether to output the frame entailments	boolean	NO
cypher.report	Whether to write reports of the transcoding. Generally, the more reports which are turned on, the greater the burden on processing due to I/O. This option, and all those below it, are false by default.	boolean	NO
cypher.report.debug	Whether to output the pattern matching debug info	boolean	NO
cypher.report.grammar	Whether to output the grammar parse info	boolean	NO
cypher.report.rdf	Whether to output any RDF statements that result from transcoding	boolean	NO
cypher.report.sparql	Whether to output the SPARQL form of the query resulting from transcoding	boolean	NO
cypher.report.serql	Whether to output the SERQL form of the query resulting from transcoding	boolean	NO
cypher.report.failed	Whether to output the lists of inputs which failed to transcode	boolean	NO
cypher.report.lexical	Whether to output the list of words which failed lexical lookup	boolean	NO
cypher.report.query-rdf	Whether to output the RDF representation of the query that results from transcoding	boolean	NO
cypher.report.html	Whether to output HTML reports. The starting point is the index.html file located at the root of the cypher.output.dir directory. Enabling this option slows down parse time considerably.	boolean	NO
cypher.report.paraphrase	Whether to output paraphrases of input	boolean	NO
cypher.report.refresh	The refresh interval for HTML interface; the format is minute:hour	string	YES
cypher.report.bulk	Whether to write reports to one file. If true, reports will be written to the <code><cypher.output.dir>/bulk</code> directory	boolean	NO
		path string	
cypher.http.base	Used by the Cypher web service. This should be the URL of the Cypher service (e.g. http://localhost:8080/cypher-ws). If the Cypher web service is running, then set the cypher.output.dir to the <code><cypher-ws home>/out</code> directory, where <code><cypher-ws</code>		

	home> is the location where the cypher-ws.war file was deflated. This is also used as the namespace for all URI's minted during transcoding.		
logger.config.file	The name of the file containing the Log4j configuration. File name must be relative to the <cypher home>/data/configuration directory	string	YES

Output and Reports

Cypher recursively loads input from the [cypher.input.file](#) directory, writes the parser results to the [cypher.output.file](#) specified in the startup properties file. The format is as follows:

- input dir
 - file-a.txt
 - file-b.html
- output dir
 - file-a
 - The first sentence
 - 1
 - query.serql
 - query.sparql
 - statements.rdf
 - grammar.xml
 - 2
 - n
 - transcode.failed.txt
 - report.csv
 - Another sentence
 - report.csv
 - file-b
 - Yet another sentence

Running the Web Service

Using the Cypher Web Service, you can run Cypher as a RESTful web service. The steps to installing the web service are:

1. Download the [cypher-ws.war](#) file
2. Install the cypher-ws.war file: Place the .war file in your web server's context (e.g. in the .../webapps folder under Tomcat). Besure the 'expand' option for your web server is set to true, as Cypher will be writing to files under it's docBase. If your web server doesn't

have a deflate option, then decompress the cypher-ws.war file and point the web server to the target directory.

3. Set `cypher.http.base=http://<yourdomain>/cypher-ws`
4. Set `cypher.output.dir=<cypher-ws home>/out`, where `<cypher-ws home>` is the directory where the cypher-ws.war file was deflated
5. Set `cypher.report=true` and `cypher.report.html=true`
6. Start the web server then browse to `<cypher.http.base>/api`

Parameter	Description
i	The input to be transcoded
o	The content type of the output. Options are 'html' and 'xml'. This can also be specified in the 'accept' POST request header (text/html and text/xml are accepted).

Sample Request: `http://localhost:8080/cypher/api?i=My+name+is+Simon&o=xml`

Sample Output

```
<msg>Results will appear at this url as they become available</msg>
<results>
<url>http://ec2.monrai.com/cypher/out/user-15/My name is
Simon/report.xml</url>
</results>
```

The URL then points to the output below (once the output is available)

```
<msg>The URIs containing the transcoder results (i.e. triples and
queries)</msg>
<results>
<result
  rdf_format="turtle"
  total_rdf="11"
  total_sparql="0"
  total_serql="0"
  path=http://localhost:8080/cypher/out/user-15/Kingsley knows Tim/1/
  rdf_path="http://localhost:8080/cypher/out/user-15/Kingsley knows
Tim/1/statements.rdf"
  rdf_assert_path=http://localhost:8080/cypher/out/user-15/Kingsley knows
Tim/1/statements.assert.rdf
  sparql_path="http://localhost:8080/cypher/out/user-15/Kingsley knows
Tim/1/query.sparql"
  serql_path="http://localhost:8080/cypher/out/user-15/Kingsley knows
Tim/1/query.serql"
  grammar_path="http://localhost:8080/cypher/out/user-15/Kingsley knows
Tim/1/grammar.xml"
  paraphrase="the entity called Kingsley perceives the entity called Tim as
familiar"/>
</results>
```

API

The Java API allows you to connect to Cypher using the Java Programming Language. To use, ensure the *.jar files in the <cypher home>/lib/* directories are in the classpath. Create a PatternMatcher object and register the appropriate listeners. Listeners include PattenMatchedListener, PatternTranscodedListener, and LexicalMatchedListener. Below is a code example of how to send text to Cypher and process the output results:

```
import com.monrai.cypher.start.Cypher;
import com.monrai.cypher.lang.nl.matcher.cfg.CFGPatternMatcher_EN;
import java.io.File;

...

// parse an input
public void parse(String sentence){

    Cypher.main(new String[0]);
    PatternMatcher matcher = new CFGPatternMatcher_EN(0, Console.stx, false);
    File out = new File("./example.out.txt");
    matcher.setOutputFile(out);
    matcher.addPatternTranscodedListener(this);
    matcher.addPatternMatchedListener(this);
    matcher.addLexicalMatchListener(this);
    Thread t = new Thread(matcher);
    matcher.loadInput(sentence);
    Cypher.getWorkQueue().execute(t);

}

// events

public void patternMatched(Pattern pattern) {
    // process syntax pattern match output
    boolean expandContainers = true;
    Constituents cons = pattern.getConstituents(expandContainers);
    for(int i = 0; i < cons.length; i++){
        Constituent con = cons[i];
        URI function = getFunction();
        if(con instanceof Pattern){
            // do something with the Pattern
        }
        else if(con instanceof Token){
            if(con instanceof Word_EN){
            }
            else if(con instanceof Symbol){
            }
            else if(con instanceof Number){
            }
        }
    }
}
}
```

```

public void patternMatchFailed(String input) {
    // handle syntatic pattern match failure
}

public void patternTranscoded(Transcoder transcoder) {
    // process semantic pattern match output
    Pattern pattern = transcoder.getPattern();
    boolean queryOutput = false;

    if(transcoder instanceof TranscoderV2_RDF) {
        TranscoderV2_RDF transcoder_rdf = ((TranscoderV2_RDF)transcoder);

        // was a query output?
        QueryFactory query = transcoder_rdf.getQuery();
        if(qf != null && qf.getProjection().length > 0) {
            // process the query output
        }
        else {
            // get string representation of the triple output
            // transcoder.getEntailmentGraph() returns the description of the
            // symbolic reference resource of transcoder.getPattern()
            // transcoder.getAuxiliaryGraph() returns the description of the resource
            // specified by mso:assert
            Graph[] ga = new Graph[] { ((TranscoderV2_RDF)
transcoder).getEntailmentGraph(), ((TranscoderV2_RDF)
transcoder).getAuxiliaryGraph()) };
            String triples = RDFRegistry.serializeGraphToRDF(ga);
            // do something with string or object representation of triples
        }
    }
}

public void patternTranscodeFailed(Pattern pattern) {
    // handle semantic pattern match failure
}

public SenseV2 unknownLexemeEncountered(Constituent token, Pattern context) {
    // handle lexical match failure
    // return a SenseV2 back to the pattern matcher
}

public Vector ambiguousLexemeEncountered(Constituent token, Pattern context,
SenseV2[] senses) {
    // handle polysemous lexical match
    // using the token and context, return a list of disambiguated Sensev2 back
to the pattern matcher
}

```

Hello World

As a first exposure to the Cypher program and definition languages, this section will describe how to make Cypher output a query whose result set is the [URI](#) resource for *Hello World*. The remaining sections will describe each definition construct in detail. The steps to producing the *Hello World* output are as follows:

- Load the Cypher data set (grammar, lexicon, framenet and ontology) given in Section 3.1. This Hello World dataset is loaded by default when the program starts. Other sets may be used by resetting the configuration properties in `<cypher home>/data/configuration/config.ini`
- Enter these lines into the Cypher input file:
 - *Cypher told me Hello World* [semantic](#) | [grammar](#)
 - *The greeting which a program said to a person* [semantic](#) | [grammar](#)
- Run Cypher (simply open the `run.bat` file in the root of the Cypher installation directory). Wait for the transcoded output to appear in the output file.

At least four files should have been generated in the output directory, two files for each input. Here are the output files and their descriptions:

File extension	Description
<code>.rdf</code>	RDF graph, produced from sentences and clauses
<code>.serql</code> <code>.sparql</code>	SeRQL and SPARQL query, produced from questions and phrases
<code>.xml</code>	The grammar output of phrase structure parser

Each file's name matches the corresponding input. Execute the query in the `.serql` or `.sparql` file corresponding to *The greeting which a program said to a person* and the result set should contain the URI for *Hello World*. If you have access to Sesame through the web interface, login and navigate to the repository you created for Cypher's semantic output, then click the [Query](#) link on the left-side panel. The page provides an interface for executing SPARQL and SeRQL queries against the Cypher repository.

Next, try entering the following lines into the input files, and executing the resulting SPARQL and SeRQL queries:

- *The thing that a computer program said* [semantic](#) | [grammar](#)
- *The people who the program spoke to* [semantic](#) | [grammar](#)
- *A person who heard the greeting* [semantic](#) | [grammar](#)

Exercise: Try extending the grammar to cover:

- *The person who heard the computer program that said Hello World*
- *Things that talked to me*
- *The first thing that Cypher told me was Hello World*

New! Number Pattern Example Dataset

To test the [new example dataset](#), try using a number pattern as a `directObject` in a sentence.

- Enter: *Tom is seventeen billion two hundred thirty eight million five hundred twenty nine thousand four hundred five.* [semantic](#) | [grammar](#)

In the RDF output file, the English number will be represented as an integer wrapped in a RDF literal.

The first mistake people make when interacting with Cypher is that the program is based on some keyword-matching trick. In order to gain understanding of the true novelty of Cypher, and the process which underlies it (called Transcography), it is suggested that the reader studies the Hello World example files (see `<cypher home>/data/language/` and `<cypher home>/data/repository/`), and take a stab at writing a sample program using the Cypher protocols.

The Dataset

The Hello World dataset is included in the Cypher release, under the examples directory. Any custom data set will need to follow the default dataset file structure. This sub-section walks through the dataset.

Con Element

An example syntax for the grammar is located in the following files:

```
<cypher home>/data/language/en/grammar/hw-cons.xml
```

```
<cypher home>/data/language/en/grammar/hw-patterns.xml
```

These files describe the phrase structure of the clause *The first thing a computer program said to me was Hello World*, allowing the program to determine which words form noun phrases and verb clusters. Each word and sub-phrase in the clause is called a constituent of the clause. A constituent is the atom of a clause or phrase pattern. The file `hw-cons.xml` contains the `<con/>` declarations, which describes all the possible string values, part of speech, and syntax role of each constituent. Syntax role are those parts of the sentence we all learned in school, for example: direct object, simple predicate, modifier, and subject. In the sentence above, the phrase *The first thing* acts as the simple subject of the sentence. The parser tags each phrase and sub phrase with the functional role specified by the `<con/>` element. Here is an example from the `hw-cons.xml` file:

```
<con label="3singNoun" function="NounPhrase/head" form="$nn"/>
```

When this `<con/>` appears in a pattern, it means that any word whose part of speech is `$nn` will be tagged with the functional role `NounPhrase/head`. Then the tagged word or phrase can be [used in the lexicon](#). The `<con/>` has three attributes: label, function, and form.

Label

The label used to reference this con element in pattern definitions. The label can contain any alphanumeric characters, dash (-), or underscore (_), and should be descriptive.

Pattern Files and index.xml

The pattern files are located at `<cypher home>/data/language/cypher.language/grammar` directory. Each directory under this directory must contain an index.xml file. The index.xml file contains the schema definitions for the patterns used in the grammar, and a list of pattern files which are to be loaded (specified by the `<load/>` directive). The order of the `<load/>` directives determines the order the files will be loaded. Files containing `<con/>` elements must be loaded before the files which reference them (i.e. pattern files). Each pattern file should also contain a `xml:base` attribute in the root `<cypher/>` element. This namespace prefix will be used to generate URIs for both the patterns and functions defined in it. These URIs are used to uniquely reference patterns and functions in the [grammar output](#) and [lexicon definitions](#).

Function

Each pattern has an associated list of functional roles borne by its constituents bare. Thus, the pattern's list of functional roles determines which `<con/>` elements can be used in the pattern. These functional roles are listed in the index.xml file using the `<function/>`. New pattern schema definitions may be added to this file. [TODO: Document *transcoder* attribute] Here is an example pattern schema definition for Date:

```
<pattern label="Date" use="Pattern">
<function label="pointer"/>
<function label="month"/>
<function label="dayOfMonth"/>
<function label="weekDay"/>
<function label="year"/>
<function label="monthDaySeperator"/>
<function label="dayYearSeperator"/>
<function label="monthYearSeperator"/>
<function label="dateComplementMarker"/>
<function label="hour"/>
<function label="minute"/>
<function label="second"/>
<function label="hourMinuteSeparator"/>
<function label="minuteSecondSeparator"/>
<function label="amPmIndicator"/>
<function label="hourIndicator"/>
</pattern>
```

When referencing a function in the `<con/>` function attribute, use the syntax `PatternName/functionName`:

```
<con label="3SingNoun" function="NounPhase/head form="$nn"/>
```

The above `<con/>` may be used only in patterns whose [use](#) element is Nounphrase.

Form

Part of Speech Forms

The form attribute of the `<con/>` element specifies the possible part of speech and/or string values of the constituent. The simplest `<con/>` form is a part of speech category. The `3singNoun` specifies `$nn`, which is the part of speech category for singular nouns.

```
<con label="3SingNoun" function="NounPhase/head" form="$nn"/>
```

The part of speech with which a word will be tagged is determined by the `<cypher home>/data/grammar/words.index` file. This is an archive file, so renaming it `.zip` will allow you to open it and access the Cypher word taxonomies. This archive contains text files; each file's name is two letters, which match the first two letters of the words in that file. The file contains a list of words and all possible part of speech candidates for the word. To add a word to Cypher's vocabulary, find the appropriate file, and add the word and **all** its available part of speech categories to the file (preferably in its alphabetical location), with emphasis is placed on **ALL**. Cypher doesn't use part of speech taggers like many NLP programs. Instead, it leverages the patterns to choose the correct part of speech. Thus, Cypher uses a knowledge-based approach (vs. a statistical approach) to such NLP task like POS tagging and WSD, which is inline with how we believe humans process language. [\[TODO: blog entry on Cypher pos tagging techniques\]](#). Use the existing lists as an example. The number after the `@` symbol is irrelevant in the beta release of Cypher. After you're done, rename the file back to `words.index` and restart Cypher. For a list of all available part of speech categories, see the file `<cypher home>/data/grammar/en/taxonomy/taxonomy.types.ini`. New part of speech categories can be created by simply adding it to the `taxonomy.types.ini` file.

In addition to specifying a part of speech for the form attribute of `<con/>`, a string value can also be specified. If a string value is given, both the part of speech category and string must match. The syntax for specifying a string value is

```
<con label="3SingNoun" function="NounPhase/head" form="{ $nn:dog }"/>
```

A list of strings can also be specified.

```
<con label="3SingNoun" function="NounPhase/head" form="{ $nn:dog,cat,house,bird }"/>
```

Only alphanumeric characters are allowed as string values. The `Non3singObjectivePronoun` uses the `$np` (proper noun) part of speech category, and also specifies that the pronoun noun has to equal the string value "me".

A list of form values can be specified using a comma to delimit the values:

```
<con label="3SingNoun" function="NounPhase/head" form="$nns, { $nn:dog,cat,house,bird }, $pn"/>
```

The `<cypher home>/data/language/en/taxonomy/taxonomy.types.ini` file contains a list of all available part of speech categories.

Pattern Forms

Pattern references can also be used as a constituent form value, which enables indefinite nesting of patterns to be specified.

```
<con label="3SingNoun" function="NounPhase/verbalModifier"
form="$GerundClause"/>
```

Here, a gerund clause may be used in a noun phrase pattern.

Character Groups

The reserved part of speech category `$character_group` will match any string of alphanumeric characters. For example:

```
<con label="Noise" function="TextFilePattern/randomNoise"
form="$character_group"/>
```

Numbers and Range Forms

The part of speech category `$da` matches any positive integer. A range of integers can be specified with the following syntax:

```
<con label="AreaCode" function="PhoneNumber/areaCode" form="{ $da:100-999 }"/>
```

This would match any number between 100 and 999 inclusive.

Symbols

Cypher tokenizes symbols (non-alphanumeric characters) as separate tokens. Thus, the string `http://` will be processed as `http : //`. Therefore, parsing `http://` will require at least three `<con/>`, one to match `http`, one to match the colon, and one to match the forward slash (which would be used twice in the [pattern](#)):

```
<con label="Schema" function="URI/schema" form="{ $character_group:http }"/>
<con label="SchemaColon" function="Uri/schemaDelimiter" form="$colon"/>
<con label="Delimit1" function="Uri/pathDelimiter1" form="$fslash"/>
<con label="Delimit2" function="Uri/pathDelimiter2" form="$fslash"/>

<!-- matches http : / / -->
<pattern use="URI" label="ProtocolPattern">
<li>$Schema $SchemaColon $Delimit1 $Delimit2</li>
</pattern>
```

Pattern Element

The hlw-patterns.xml file contains the <pattern/> elements which describe the phrase structure grammar used to parse and annotate user input. The simplest pattern is a list of one or more <con/> references, i.e. the <con/> label attribute value with a dollar sign (\$) prepended. The DeclarativeName pattern in the example file specifies two constituents:

```
<pattern use="Name" label="DeclarativeName">
<li>$FirstName $LastName</li>
</pattern>
```

The element is a required child of the <pattern /> element.

Optional Constituents

Place a list of mandatory constituents in brackets to specify one or more constituents:

```
<pattern use="Name" label="DeclarativeName">
<li>[$FirstName $LastName,$FirstName]</li>
</pattern>
```

Place a list of optional constituents in parentheses to specify zero or more constituents:

```
<pattern use="Name" label="DeclarativeName">
<li>$FirstName ($LastName)</li>
</pattern>
```

Recursion

There are two types of recursion for phrase definitions. The first is self-reference, i.e. referencing a pattern as it's own constituent. This can be accomplished simply by creating a <con/> whose form is patternA, and using that <con/> in the definition of patternA. The shorthand for doing this is to use the `this` keyword syntax:

```
<pattern use="Name" label="DeclarativeProperName">
<li>$ProperName ($this:misc)</li>
</pattern>
```

Here, `this` is a keyword which references the pattern this construct appears in. Following the colon is the function for the constituent, excluding the pattern type prefix. Here, we use the `misc` function. [TODO: describe shortcut syntax]

Example

Input: *Henry Marshall Foundation*

Output:

```
<pattern>
<con type="ProperName" value="Henry"/>
```

```
<pattern>
<con type="ProperName" value="Marshall"/>
<pattern>
<con type="ProperName" value="Foundation"/>
</pattern>
</pattern>
</pattern>
```

The other type of recursion is repetition. Placing the + symbol after a constituent specifies that constituent may repeat.

```
<pattern use="Name" label="DeclarativeProperName">
<li>$ProperName ($this:misc+)</li>
</pattern>
```

Example

Input: *Henry Marshall Foundation*

Output:

```
<pattern>
<con type="ProperName" value="Henry"/>
<con type="ProperName" value="Marshall"/>
<con type="ProperName" value="Foundation"/>
</pattern>
```

As the examples demonstrate, the difference between recursion and repetition is that the former specifies nesting, and the latter specifies listing. Be sure to always place repeating constituents (those which use the + marker) inside parenthesis, to allow the pattern to escape, otherwise you'll have a pattern which requires a constituent list of infinite length in order to match (i.e. a pattern which can never match).

The `<pattern/>` element has one mandatory child: ``, and three attributes: `label`, `use`, and `store`.

Li

A list of patterns definitions may be given using the `` element

Label

The label used to reference this element. The label can contain any alphanumeric characters, dash (-), or underscore (_), and should be descriptive.

Use

The use attribute specifies which pattern function may be used in the pattern definition. If the pattern type is *XYZ*, then only `<con/>` whose function attribute is `XYZ.function` may be used in the pattern definition.

Container Element

The `<container/>` element is syntactically identical to the `<pattern/>` element. Containers provide a way to reference a list of constituents. Like macros, containers are used to prevent repetition of code.

mso:Sense Class

Now that we have defined a phrase structure, we can reuse the grammar to generate RDF graphs and queries for other input with the same phrase structure, for example:

This is a picture of my dog playing in the park

The sales meeting is today

This is done by extending the lexicon. An example lexicon for the grammar is located in the following files:

```
<cypher home>/data/repository/lexicon/hw.rdf
```

The lexicon is an RDF document which contains the program's vocabulary, and is used to lookup the semantic output for a word or word group. When a word is encountered in the user input, the lexical information for that word is retrieved and used to generate the output. A word sense entry contains the information for a particular usage of a given lemma. Among the types of metadata associated with a word sense are lemma form, allowed syntactic arguments, required syntactic arguments, restricted syntactic arguments, and semantic output.

rdfs:label

The `rdfs:label` property is used to reference this element. The label can contain any alphanumeric characters, dash (-), or underscore (_), and should be descriptive.

mso:lemma

The `mso:lemma` is the canonical representation of a word. Generally, the root form of the word should be used. The lemma is used by Cypher to look up the word. Cypher determines the lemma of a word by transforming the word into its root form using the morphological processor located in `<cypher home>/data/language/cypher.language/grammar`.

mso:pos

The `mso:pos` property specifies the part of speech to which this lexeme belongs. Only words whose root form matches the lexeme's lemma, and whose part of speech matches this property may use this lexeme.

mso:Sense

An instance of `mso:Sense` corresponds to a particular usage of the lemma. Word senses place restrictions on the lexeme's syntactic arguments, and contain the semantic output templates needed to generate the RDF graphs, SeRQL, and SPARQL queries. Linguistics refers to this as describing the "subcategorization" of the lexeme. The three types of restrictions are allows, requires, and restricts. Each syntactic argument restriction has the same structure. For example, the *know* lexeme in the example file has the following sense element:

```
<mso:Sense rdf:about="&mso;know">
<rdfs:label>Know</rdfs:label>
<mso:lemma>know</mso:lemma>
<mso:pos>V</mso:pos>
<mso:requires rdf:resource="&mso-terms;Clause/subject" />
<mso:allows rdf:resource="&mso-terms;Clause/directObject"/>
<mso:ref rdf:nodeID="knowStatement" />
</mso:Sense>

<rdf:Statement rdf:nodeID="knowStatement">
<rdf:subject rdf:resource="&mso-terms;Clause/subject"/>
<rdf:predicate rdf:resource="&foaf;knows"/>
<rdf:object rdf:resource="&mso-terms;Clause/directObject"/>
</rdf:Statement>
```

Notice the URIs representing the syntax functions for subject and direct object. These URIs are produced using the pattern `<namespace><pattern label>/<function label>`. The namespace is given by the `xml:base` specified in the [index.xml](#) file describing the Clause schema. This word sense specifies that constituents whose function is `Clause/subject` must appear in the immediate syntactic context of the token *know* (i.e. it must be a sister node of *know* in the grammar parse tree), and that `Clause/directObject` is allowed to appear but is not mandatory. A word lookup will fail if one of the following is true about the word's syntactic context:

- a restricted constituent appears
- a required constituent does not appear
- a constituent appears which is neither allowed nor required

mso:Sense Output

The `mso:Sense` class also describes a function. Once an input token has been matched to a correct word sense, the associated output is used to generate the RDF, or SeRQL and SPARQL for the input. The `mso:Sense` class may specify one of two types of output: `mso:ref`, and `mso:assert`.

mso:ref

The `mso:ref` specifies an RDF resource (URI or BNode) referenced by the word sense. In the case of a URI, the output for the word sense instance will be the URI given. In the case of a BNode, a new resource will be created to represent the sense instance, and the description of the given BNode (i.e. all statements in which the BNode is a subject or object) will be transferred to the newly created resource. For example, to say that the word *something* means `rdf:Thing`, use the following lexeme:

```
<mso:Sense rdf:about="something" pos="n">
<rdfs:label>Something</rdfs:label>
<mso:lemma>something</mso:lemma>
<mso:ref rdf:Resource="aThing"/>
</mso:Sense>

<rdf:Thing rdf:nodeID="aThing">
</rdf:Thing>
```

This allows for descriptions of arbitrary complexity to be transferred to any phrase in the input. The novelty of this technique is the ability to describe verb clusters using `rdf:Statement` URIs and [semantic frames](#).

Thus the description of the `<mso:ref />` resource is treated as a template. Functional role names specified by the `<mso:allows>` and `<mso:requires />` elements may be used in these templates to reference arguments from the word's context. The functional role URI in the template will be replaced by the argument's symbolic reference, i.e. the [Resource which Cypher generated](#) to represent the argument. This process is called Identity Transfer. Here is an example word sense definition for a usage of the word *talk*:

```
<mso:Sense rdf:about="TALK">
<mso:pos>V</mso:pos>
<rdfs:label>Talk</rdfs:label>
<mso:lemma>talk</mso:lemma>
<mso:requires rdf:resource="&mso-terms;Clause/subject"/>
<mso:requires rdf:resource="&mso-terms;Clause/toComplement"/>
<mso:ref rdf:nodeID="commFrame"/>
</mso:Sense>

<mso:Communicate rdf:nodeID="commFrame">
<mso:agent rdf:resource="&mso-terms;Clause/subject"/>
<rdf:theme rdf:resource="&mso-terms;Clause/toComplement"/>
</mso:Communicate>
```

Now, the following semantic output will be generated when *I talked to Dennis Cooper* is transcoded:

Grammar Parser Output:

```
<pattern>
<con value="I" function="&mso-terms;Clause/subject" />
```

```

<con value="talk" function="Clause/predicate" />
<con value="to" function="Clause/toComplementMarker" />
<pattern type="Name" function="&mso-terms;Clause/toComplement">
<con value="Dennis" function="Name/firstName" />
<con value="Copper" function="Name/lastName" />
</pattern>
</pattern>

```

Semantic Output:

```

{bnode0} rdf:type {mso:communicate}
{bnode0} mso:agent {myonto:HenryWilliamsSr}
{bnode0} mso:theme {myonto:DennyJC}

```

Now, if I want to know who had a conversation with Dennis Cooper, I just query the `mso:agent` value of all `mso:Communicate` frames whose `mso:theme` equals `myonto:DennyJC`. As we shall soon show, these simple semantic constructs can go a very long way.

When a resource of `rdf:type rdf:Statement` is given as the `mso:ref`, Cypher will assert the triples the `rdf:Statement` represents. In the know example above, the following triple will be asserted in the output (the functional role URIs will be replaced by their symbolic references):

```

{&mso-terms;Clause/subject} &foaf;knows {&mso-terms;Clause/directObject}

```

mso:assert

The `mso:assert` property takes a resource (i.e. URI or BNode) as a value. The description of the given resource (i.e. all triples where the resource is a subject or object) will be included in the triples returned by the sense element. This allows for the inclusion of statements about arbitrary resources to be included in the output of a word sense. Like `mso:ref`, functional roles appearing in the statements will be swapped for their symbolic references.

Base Senses

It is convenient to specify a sense which has rules and output shared by all word sense of a particular part-of-speech. To achieve this, create an instance of a `mso:Sense` and omit the `mso:lemma`. The metadata specified by this word sense will be included in the metadata for all word senses which matching its `mso:pos`.

Frame Semantics

Cypher uses frame semantics to represent the semantic content of words like verbs and relational nouns (e.g. mother, lawyer). It is good lexical design practice to make the symbolic reference of verbs and relational nouns be of type `mso:Frame` or `rdf:Statement`.

The frame net definition is located in the following files:

```
<cypher home>/data/repository/dictionary/hw.rdfs
```

In a nutshell, frame semantics represent the semantic content of words like verbs as scenes. Each scene (frame) has actors that play roles in the scene, these roles are called frame elements. By knowing that Henry plays the `receiver` role of the `Give` frame, we know he has something, and that he got that something from someone. So the query *Who has something*, should return *Henry*, even though it wasn't explicitly stated that Henry has something. Frame semantics was chosen mostly for its simplicity, we've found that when frame elements are linked to one another, a powerful yet elegant semantic network results.

The `mso:Frame` class encapsulates a very simple frame definition. Each `mso:Frame` has a label, a description (used for paraphrasing), a list of frame element (i.e. the roles which are played in the frame scene), and a list of sub-frames. Each sub-frame lists one or more frame instances, and frame elements from the parent frame are passed to a child frame. In this way, each frame acts as a dictionary entry, with links pointing into and out from the frame.

```
<!-- COMMUNICATE FRAME -->
<owl:Class rdf:about="Communicate">
<rdfs:subClassOf rdf:resource="&mso;Frame"/>
<rdfs:label>communicate</rdfs:label>
<dc:description>&mso;speaker conveys information to &mso;addressee, the
information conveyed is &mso;message</dc:description>
<mso:subFrame rdf:nodeID="conveyFrame"/>
</owl:Class>

<mso:ConveyFrame rdf:nodeID="conveyFrame">
<mso:conveyor rdf:resource="&mso;speaker"/>
<mso:audience rdf:resource="&mso;addressee"/>
<mso:thought rdf:resource="&mso;message"/>
</mso:ConveyFrame>

<owl:ObjectProperty rdf:about="speaker">
<rdfs:label>speaker</rdfs:label>
<rdfs:domain rdf:resource="&mso;Communicate"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="addressee">
<rdfs:label>addressee</rdfs:label>
<rdfs:domain rdf:resource="&mso;Communicate"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="message">
<rdfs:label>message</rdfs:label>
<rdfs:domain rdf:resource="&mso;Communicate"/>
</owl:ObjectProperty>
```

Frame semantics add an important dimension to RDF. If a program wants to know *Which women ran for president*, it would only need to execute the following query:

```
select president from
{frame} rdf:type mso:Run-ForOfficeFrame
{office} mso:office {mso:USPresident}
{frame} mso:agent {president}
{president} rdf:type {mso:Woman}
```

Proper Names

Proper names are not entered into the lexicon using the `<mso:Sense/>` class. Instead, proper names are resolved by searching the FOAF statements. When a capitalized word appears in a nominal context, Cypher identifies the word(s) as a Name pattern, and attempts to retrieve the URI for the word from the FOAF graph. Currently, Cypher only acknowledges `foaf:surname`, `foaf:firstname`, `foaf:family_name` and `foaf:givenname`. For the example, we treated Hello World as a person whose first name is *Hello* and last name is *World*. There is a FOAF resource for Hello World in the `<cypher home>/data/repository/lexicon/hw.foaf.rdf` file.

Additional Topics

Paraphrases

It is useful to ask Cypher to paraphrase input as a measure to validating transcoder results. This is controlled by the [cypher.report.paraphrase](#) startup property. The `dc:description` property is used to determine the paraphrase for resources. Consider the following word sense for the term *dog*:

```
<mso:Sense rdf:about="&mso-terms;term-dog">
<mso:lemma>dog</mso:lemma>
<mso:pos>N</mso:pos>/'
<rdfs:label>dog</rdfs:label>
<mso:requires rdf:resource="&mso-terms;NounPhrase/pointer"/>
<mso:ref rdf:nodeID="aDog"/>
</mso:Sense>

<mso:Canine rdf:nodeID="aDog">
<dc:description>a male animal of the family Canidae</dc:description>
</mso:Canine>
```

For the input *I have a dog*, the paraphrase output will be *I have a dog (male animal of the family Canidae)*. URIs for property values occurring in the `dc:description` will be replaced by the paraphrase of its value. For example, consider the word sense for the term *like*:

```
<mso:Sense rdf:about="&mso-terms;like">
<rdfs:label>like</rdfs:label>
<mso:lemma>like</mso:lemma>
<mso:pos>V</mso:pos>
<rdfs:label>like</rdfs:label>
<mso:requires rdf:resource="&mso-terms;Clause/subject"/>
```

```

<mso:requires rdf:resource="&mso-terms;Clause/directObject"/>
<mso:ref rdf:nodeID="interestStatement"/>
</mso:Sense>

<rdf:Statement rdf:nodeID="interestStatement">
<rdf:subject rdf:resource="&mso-terms;Clause/subject"/>
<rdf:predicate rdf:resource="&foaf;interest"/>
<rdf:object rdf:nodeID="&mso-terms;Clause/directObject"/>
<dc:description>&rdf;subject is interested in &rdf;object</dc:description>
</rdf:Statement>

<foaf:Person rdf:nodeID="johnny">
<foaf:givenname>Johnny</foaf:givenname>
<dc:description>Katherine's older brother</dc:description>
</foaf:Person>

```

Here, for the input *Johnny likes dogs*, the paraphrase output will be *Johnny (Katherine's older brother) is interested in dogs (male animals of the family Canidae)*. Note that the case of the grammatical number of the nouns in the declared paraphrase matches the grammatical number of the noun it paraphrases in the input. In addition, Cypher ensures subject-verb agreement, and that the case of the nouns in the `dc:description` match the case of the nouns they paraphrase.

Users should ensure that the description is comprehensive, and is a single complete phrase.

FAQ

- [How well would Cypher perform on random input and standard corpora?](#)
- [Is there an online demo or evaluation for Cypher?](#)
- [How does Cypher tie into similar systems?](#)
- [Is Cypher proprietary technology?](#)
- [Is there any technical information about Cypher's accuracy and coverage?](#)
- [I can't seem to start the program.](#)
- [The inputs didn't produce an output file.](#)
- [The generated SeRQL query did not produce any result set](#)
- [I get "Connection Refused" errors](#)

How well would Cypher perform on random input and/or public corpora; the Brown Corpus, or the Wall Street Journal corpus, for example?

Although Cypher embodies very advanced natural language processing techniques, it is not a tool intended for creating a general purpose natural language processor. Anyone is free to attempt this with the framework, however. Cypher was developed to perform two very specific tasks very well: 1) generate syntactic output which describes the structure of natural language descriptions (*this is a picture of me in Vienna with Dan, his three brothers and my dog*) and phrases (*movies that star Tom Hanks playing an officer*), and 2) to produce RDF triples by leveraging a lexicon containing explicit templates mapping the syntactic output to semantic frame and RDF triple slots. That said, Cypher's phrase

description language is powerful enough to address the various linguistic phenomenon encountered in most English corpora, however, it is important to note that first the phrase description patterns must be written, and that a lexicon must exist for each word encountered in the corpora. Cypher is designed to parse a very minuscule subset of the "territory of language": noun phrases and descriptive clauses. We believe this sliver of coverage represents the vast majority of input required to build a plain language interface to the semantic web.

Is there an online demo or evaluation for Cypher?

Cypher and its documentation are available free for anyone to use. Currently there is no online demo available. Any online demo which would do Cypher justice should allow users to upload and test their own datasets, since it is the data set author who knows best what output to expect. We're currently working on such a demo. In the meantime, there is an example data set, the [Hello World example](#). Users are encouraged to study and modify the Hello World example as an entry point into understanding the potential of the Cypher framework. The English data set contains the Monrai developed grammar patterns for a wide variety of clauses and patterns, which, along with a user created lexicon, can process literally millions of natural language descriptions and phrase patterns.

How does Cypher tie into similar systems?

Much of the technology upon which Cypher is based already exist in one form or another in separate systems, i.e. constituent phrase structure parsers, semantic frame networks, lexical databases, RDF generators, etc. Cypher is novel in the sense that it brings these disparate technologies and techniques together for the goal of making the semantic web easier for average users to use. It does this by offering plain language as a tool for users to create metadata.

Is Cypher proprietary technology?

The protocols and theory Cypher uses to produce its output are open for users to examine. These are available as the Cypher definition languages for the [phrase grammar](#), [lexicon](#), [framenet](#), and the W3C recommended [RDF\(S\) specifications](#). Anyone with the appropriate programming skills can (and are encouraged to) develop an interpreter for these protocols and in effect create their own Cypher implementation. Although the current Monrai implementation of Cypher is proprietary, we have and will continue to offer it free (as in beer), much like Sun's model for its JVM technology. We will consider opening the implementation as the technology matures.

Is there any technical information about Cypher's accuracy and coverage (e.g. precision, recall, fallout)?

Accuracy and coverage statistics should be applied to the data set, not to the Cypher implementation. Data sets are created by the Cypher user. One can say

that Cypher has 100% precision and recall againsts the input set for which its data set is designed to handle. Theoretically, a large enough grammar and lexicon could allow Cypher to approach 100% accuracy for random descriptions and noun phrases. Cypher's performance is always relative to the grammar and lexicon in it's data set.

I can't seem to start the program.

The `run.bat` file in the root of the install directory starts the program. Cypher is command-line based. Once the `IN>` prompt appears, the program is ready for input. The `run.bat` script assumes that Cypher was unzipped to the `C:/` directory. Please edit the `run.bat` script to match your installation root. Creating a `.sh` script for your Linux system should be simple using the `run.bat` as a guide.

The inputs didn't produce an output file.

The output will be generated to the `<cypher_home>/output` folder. The name of the file matches the input. The Hello World example uses a very sparse dataset, so be sure you entered the example inputs correctly, as these input patterns and words are the only ones covered by the example. If all else fails, study the example source files to make sure your input pattern is covered by the grammar. Then modify the file or your input appropriately.

The generated SeRQL query did not produce any result set

Some queries may require you to run an inferencer with the Sesame database. For example, the input *The thing that a computer program said* will query a resource of `rdf:type mso:Thing`. The default dataset states that Cypher (which you stated has said a greeting) is of type `mso:Computer_Program`, and `mso:Computer_Program` is of type `mso:Thing`. An inferencer will allow Sesame to infer that Cypher is also a `mso:Thing`. The alternative is to explicitly add the triple

```
mso:cypher rdf:type mso:Thing
```

to the dataset (simple place the file containing the triples into the `<cypher_home>/repository/output/directory`). The [OWLIM](#) inferencer is a decent implementation. Visit openrdf.org for a comprehensive list.

I get "Connection Refused" errors when trying to start Cypher

In order for Cypher to run, it needs to be able to connect to a [Sesame](#) database. In addition, the database you will need to create two repositories for Cypher to use. Set the repository names in the `<cypher_home>/data/config/engine.properties` file, for example:

```
rdf.repository.output=rep1  
rdf.repository.internal=rep2
```

Modify the `rdf.repository.url` property so that it points to your instance of Sesame (the default Cypher setting points to `http://localhost/openrdf-sesame/`).