

Natural Language Understanding for Healthcare Queries

Vivek Raghuram



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-35

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-35.html>

May 9, 2018

Copyright © 2018, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Natural Language Understanding for Healthcare Queries

by Vivek Raghuram

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Jerome Feldman
Research Advisor

Date

Professor Nelson Morgan
Second Reader

Date

Abstract

Natural Language Understanding for Healthcare Queries

by

Vivek Raghuram

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Jerome Feldman, Chair

One of the goals of search query understanding is to extract as much useful information about a query as possible. Natural language understanding (NLU) systems, which are designed to provide detailed information on domain-constrained user input, are well-suited to this task. We demonstrate an implementation of the ECG System [10], an application independent NLU system, towards query understanding on a healthcare wiki. In our implementation we take into account all the constraints imposed by integrating with an existing wiki maintained by The Hesperian Society. In doing so we test the flexibility of the ECG framework, show the information extracted can be used to improve queries and develop several useful improvements to the ECG System.

Contents

Contents	i
1 Introduction	1
2 Background	3
2.1 Embodied Construction Grammar (ECG)	3
2.2 ECG System	4
2.3 MediaWiki	6
3 Hesperian Wiki	7
3.1 Hesperian Wiki Version Information	7
3.2 Upgrading the Wiki	9
4 The Hesperian Product	10
4.1 User Interface	10
4.2 Language Side	14
4.3 Application Side	25
4.4 Example Queries	28
5 Related Work	31
6 Conclusion	33
6.1 Contribution and Limitations	33
6.2 Future Work	33
6.3 Concluding Thoughts	34
A Wiki Upgrade Instructions	35
B Selected Queries	39
C Sample ActSpec	43
References	46

Acknowledgments

First, I'd like to thank my research advisor, Professor Jerome Feldman, for all his wisdom and guidance over the last three years. Thanks to him I've had the opportunity to engage with a number of fascinating topics, nurturing my interest in language understanding and cognitive science. I'm grateful for Professor Feldman's instruction and advice which has made me a better computer scientist. I would also like to thank Professor Nelson Morgan for his helpful feedback and being on my Masters committee. I'd also like to thank everyone at ICSI who has helped me with this project over the years. That includes Greta Huang, Kavi Mehta, Sid Oderberg, Kelly Shen and Sean Trott. I'd like to especially mention, Ethan Goldberg, without whose contributions this work would not have been possible. Finally, I'd like to thank my family for their constant encouragement and support.

Chapter 1

Introduction

With the abundance of information on the Internet, the primary method of information retrieval is through a search engine. However, search engines generally operate on surface characteristics of user queries, with only limited knowledge of the underlying meaning. This is because natural language is inherently messy. It follows a broad and often inconsistent set of rules operating at both the level of syntax and semantics which can make interpreting the meaning of an utterance a challenge for a computer system. This imposes significant limitations on the capabilities of information retrieval systems (and dialogue systems, Q&A systems, etc.). We attempt to address this problem by adding a natural language understanding component to a search engine in order to understand and rewrite user queries.

The ECG System is a natural language understanding system using embodied construction grammars (ECG) [3, 11], to identify the structure of utterances and extract their meaning. The analysis of input is based on a specified ECG grammar, which encodes information through form-meaning pairs with a focus on embodied meaning. The system is designed to be easily adapted to work with a target application and domain. It has been previously applied toward robotic control and real-time strategy video gaming [10]. This project applies the ECG System towards query understanding on a healthcare wiki.

The wiki in question belongs The Hesperian Society, a non-profit organization involved in the dissemination of healthcare information for situations when a medical professional is not available. Their original publication, Where There Is No Doctor, expanded into numerous books covering a range of issues from women's health, childcare, common infections, etc. These books in turn were published in a wiki during a previous collaboration with the International Computer Science Institute (ICSI) [30]. The wiki now gets thousands of visitors seeking healthcare information that may not be freely available in their countries or communities. As a result user privacy is an important concern and prevents them from maintaining identifiable user information. Additionally, legal requirements dictate that although the wikis can maintain a repository of medical information, they cannot be perceived to be giving medical advice.

The current search engine, while generally capable, is inadequate when dealing with a specialized domain. By understanding the user queries within this context, we hypothesize

we can extract important details about the users and their needs. This information can then be used to rewrite the user query to improve its performance in the existing search engine. The Hesperian wiki is particularly well-suited to testing this hypothesis due to the inherently detailed and specific nature of medical information.

Through building this Hesperian Product ¹, we also test the robustness of the ECG System design. We specifically approach the problem as a system integration challenge with an existing application, taking into account all the constraints imposed by that application. To that end, we maintain user privacy and use the search engine to return pages, rather than returning specific answers, to avoid giving medical advice.

In this work we show how incorporating natural language understanding can be useful for better understanding queries within a specialized domain. We also demonstrate the flexibility and extensibility of the ECG System towards addressing a new domain and target application entirely different from previous ECG products.

The structure of this report is as follows. In chapter 2 we present background on this work, including more detailed information on ECG and the ECG System. Chapter 3 discusses the technical details of the Hesperian wiki and the steps undertaken to prepare it for this project. Chapter 4 covers the various features of the solution, the Hesperian Product. Chapter 5 covers related work. Finally, chapter 6 discusses contributions and limitations, future work and concludes the report.

¹Our code is available at www.github.com/ICSI-Berkeley/ecg_hesperian

Chapter 2

Background

In this section we describe the necessary background on the ECG System and Mediawiki. Previous papers have described both the ECG System [10] and Embodied Construction Grammar (ECG) [13] in considerably more detail. Additionally code and tutorials for everything we describe are available on our Github Wiki [31]¹. The Github Wiki also includes further background on ECG, code for other ECG products and links to plenty of other previous work concerning ECG. That said, this section should provide enough background to understand the remainder of this paper.

2.1 Embodied Construction Grammar (ECG)

Embodied construction grammar is a formalism for representing language usage and meaning [3, 11]. It is based on both the Neural Theory of Language (NTL) [12] and extensive cognitive science research. Like other construction grammars, ECG represents linguistic content through pairings of form and meaning. In the case of ECG, meaning is represented by embodied schemas that, taken together, form a lattice of embodied semantic knowledge. Crucially, more abstract schemas are composed of primitive embodied schemas, reflecting beliefs on how humans construct meaning from conceptual primitives [12]. The individual schemas are activated by grammatical constructions which bind grammatical constituents to roles on the schemas. Schemas and constructions taken together constitute a grammar.

Thanks to extensive work done using ECG grammars, we have developed various grammar packages that are assembled into a “core” grammar [31]. The core grammar largely consists of relatively primitive constructions and schemas that most grammars will need such as prepositions and containment schemas. Developing a new ECG grammar consists of importing the appropriate core packages before adding additional constructions and schemas for the particular domain.

ECG is computationally implemented using a best-fit construction parser called the Analyzer [4]. The Analyzer takes a fully defined ECG grammar and produces semantic analyses

¹github.com/icsi-berkeley/ecg_homepage/wiki

of input utterances.

There are a few important features of ECG that are necessary for understanding parts of this paper.

1. ECG’s vocabulary for open class words is implemented using a type-token mechanism, where the grammar contains constructions for a general type and then a separate tokens file contains entries for the specific tokens of that type. For example, a `SymptomType` construction might have numerous tokens such as “cramp” and “period”. These tokens are defined with different values on their roles (e.g. the patient gender for “period” will be female). However, importantly, all these different symptoms behave the same way grammatically and so are represented by the same grammatical construction.
2. ECG contains the CELEX [1] morphology which contains broad coverage for the English language and is used in conjunction with the tokens to recognize morphological variations of words in utterances.
3. ECG uses an ontological lattice to classify and constrain units of meaning. For example, “cramp” might have an ontology entry that is a subtype of “pain” and “symptom”. Ontological constraints are an important part of ECG analyses.

To aid in development of ECG grammars, we use an integrated development environment (IDE) called the ECG Workbench [16]. It is built atop the Eclipse Rich Client Platform and includes both the Analyzer as well as a number of tools to aid in grammar development. For this project the Analyzer and Workbench were updated to use Java 8 and Eclipse 4.7 Oxygen. Some of the tools available in the Workbench include a Token Editor [34] to easily add tokens to the grammar and a Sentence Test Runner to quickly attempt parses of a large number of sentences. Additionally, when the Analyzer parses a sentence, it produces a data structure called a Semantic Specification or SemSpec. The Workbench provides a convenient interface to view SemSpecs (Figure 2.1).

2.2 ECG System

The ECG System refers to the full natural language understanding system that utilizes ECG as the underlying grammar and is made of multiple modular components. The main components of the system are in the bottom half of Figure 2.2: the Analyzer, Specializer and Problem Solver. As a generalizable framework, the ECG System is designed to be retargeted to different application domains [10]. Many of the components of the ECG System have “core” versions, that are extended using Python’s inheritance feature to produce domain specific versions.

The Analyzer and Specializer, which are encapsulated in the UI-Agent, encompass the “Language Side” of the system. The job of the Analyzer is to produce a SemSpec which the Specializer then crawls to extract task specific information in JSON structures called Action

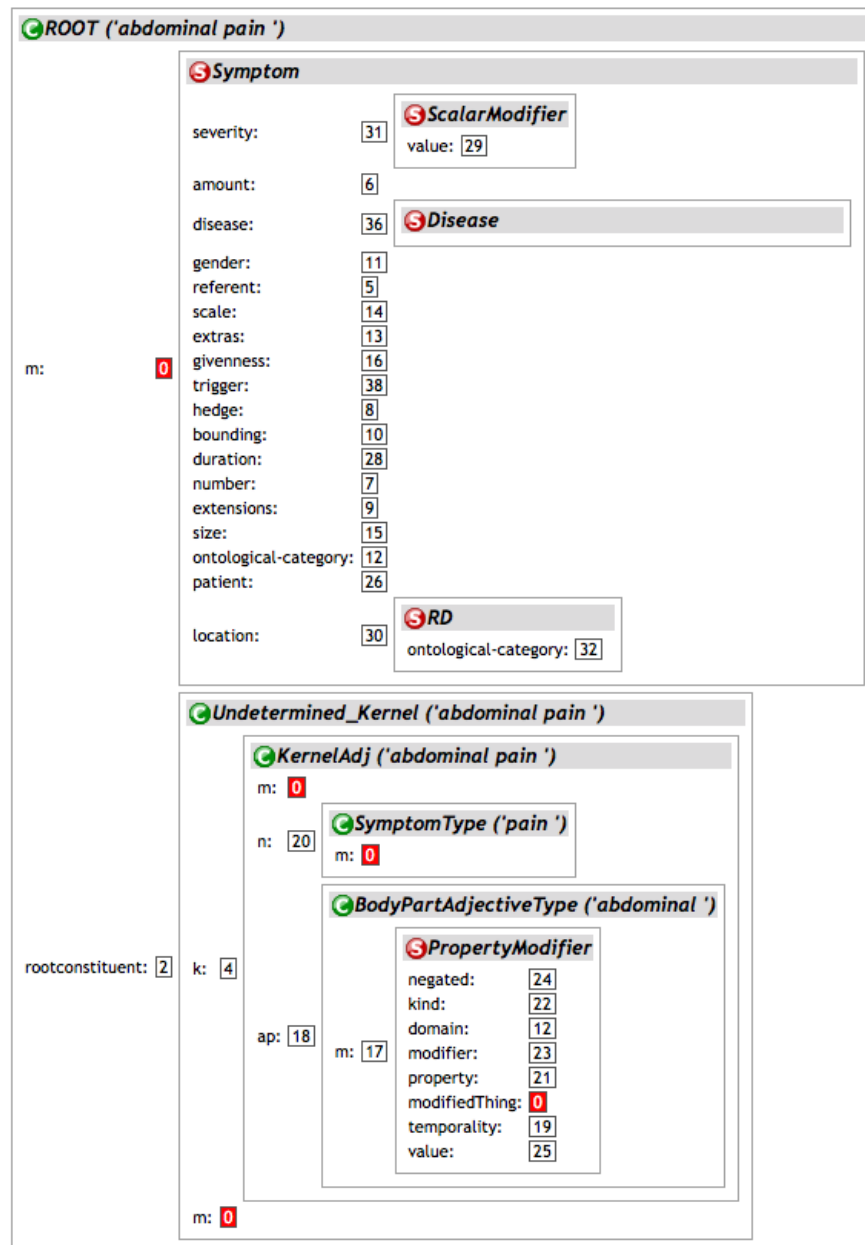


Figure 2.1: SemSpec for the noun phrase, "abdominal pain"

Specifications (ActSpecs). Since the provided grammar fully parameterizes the parsing behavior for a particular domain, the Analyzer almost never changes from app to app. The Specializer only requires minimal changes since most of its behavior is guided by the declaration of ActSpec templates. These templates specify the structure of the ultimate ActSpec and what information should fill its fields. The UI-Agent mediates communication between the two pieces and in some applications also takes the user input. Because the UI-Agent

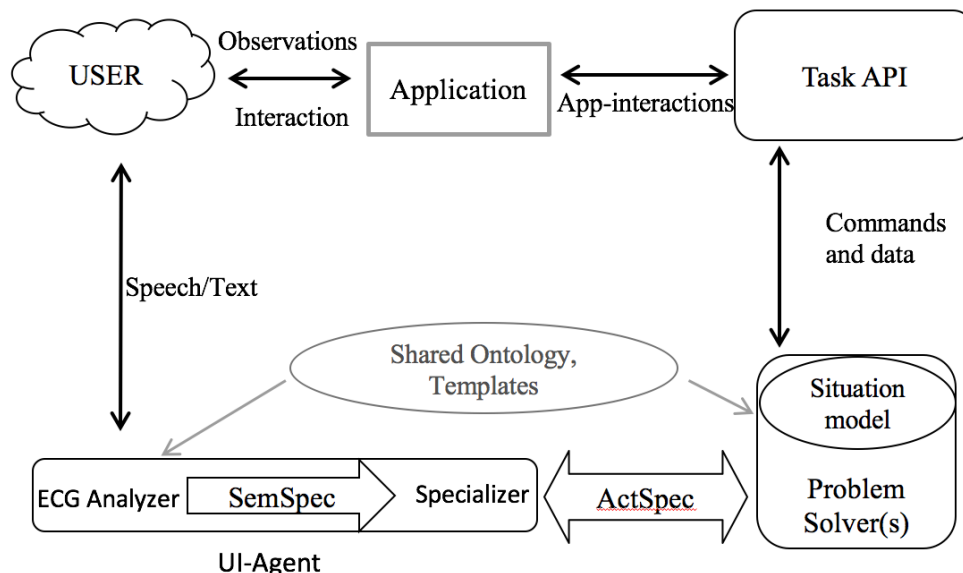


Figure 2.2: The ECG System diagram. Note that in the Hesperian Product, users do not provide text/speech directly to the UI-Agent since they interact only with the wiki.

handles input before it reaches the Analyzer, it is also the ideal place for preprocessing to occur. This is important for section 4.2.

The “Application Side” of the system includes the Problem Solver and the target application. The Problem Solver is responsible for reading the ActSpec and executing any logic to go from the ActSpec to application API calls. Since the intermediary logic varies significantly by application, the Core Problem Solver contains only minimal predefined structure and typically requires very different logic depending on the domain. Obviously the application and its API are entirely domain dependent.

2.3 MediaWiki

Apart from the ECG System, the other major component of the Hesperian Product is a wiki built using the MediaWiki² software package [30]. MediaWiki is an open source, extensible, wiki platform developed by the Wikimedia Foundation and originally used in Wikipedia. We specifically use version 1.29 for this work.

Out of the box, MediaWiki provides a full-featured wiki with the ability to create new pages, edit those pages and search for content. The software supports a number of extensions that can modify the wiki’s behavior as well as add functionality. The extensions can run both on the server-side and the client-side. In this particular case, the extension we will discuss modifies the search behavior of the wiki.

²www.mediawiki.org/wiki/MediaWiki

Chapter 3

Hesperian Wiki

The Hesperian wiki is a healthcare wiki maintained by the Hesperian Society and is the target application of this Hesperian Product. The wiki, the product of a previous collaboration with ICSI [30], contains all the information from various books published by the Hesperian Society. The information is structured according to books, chapters and subsections, just as it is in the physical books themselves. The search engine searches across all the books. The Hesperian Society maintains wikis for numerous languages but the focus of our Hesperian Product is their English wiki. Note that, unlike many wikis, only Hesperian Society staff, not general visitors, can edit the contents of the wiki.

In order to integrate the ECG System with the Hesperian wiki, we needed to upgrade the wiki software as well as various important extensions. As of this writing, the publicly available version of the wiki has been upgrading according to methods described in this section. In the following two subsections we describe the upgraded version of the wiki and the process of upgrading the wiki.

3.1 Hesperian Wiki Version Information

The current (upgraded) version of the wiki is MediaWiki 1.29.1 and it runs on an Ubuntu 16.04 LTS server using Apache2. It uses PHP 5.6 and MySQL 5.1. These versions were chosen to smooth out the upgrade procedure by avoiding upgrading parts other than MediaWiki whenever possible. The version of MediaWiki is the latest version that was available at the time.

The wiki uses a custom skin to present additional information and modify the page layout. It also uses a few extensions apart from the core extensions packaged with every MediaWiki application but most do not have a material impact this project and are omitted. One extension that is important is the GoogleSiteSearch extension¹ version 3.0, which is used to replace the default MediaWiki search with Google Search. Components of this extension are

¹www.mediawiki.org/wiki/Extension:GoogleSiteSearch



Figure 3.1: Left: English Hesperian wiki homepage. Middle: Table of contents for the book, *Where Women Have No Doctor*. Right: Page corresponding to chapter about “Common Problems during Pregnancy”.

incorporated into the custom extension we develop for the Hesperian Product. The extension we develop is discussed in greater detail in Section 4.3.

3.2 Upgrading the Wiki

The original wiki was terribly out of date which would have made developing anything to work with it extremely difficult. To fix that, we decided to upgrade the version of MediaWiki along with any other parts of the wiki that needed upgrading in the process. The wiki originally used MediaWiki 1.17 running on Ubuntu 10.04 LTS. It used PHP 5.3 and MySQL 5.1.

The main problem with upgrading the wiki was the incompatibility between the Hesperian custom skin and new MediaWiki APIs. Changing the skin to be compatible required a full refactor of the skin to match the new API while retaining the previous look and feel of the website. Additional changes include upgrading, replacing or simply removing out of date extensions, updating the database and adding additional languages. The details of the upgrade procedure are too specific to be discussed here, but a generalized version of the instructions we produced in the upgrade process are included in Appendix A.

To test the upgrade procedure as well as for later development on the Hesperian Product, we ran the wiki inside a virtual machine using VMware Fusion 8.5. The wiki was fully reproduced and ran exactly as expected within the virtual machine.

Chapter 4

The Hesperian Product

To discuss the Hesperian Product, we first analyze the user interface of the product to establish the use cases and motivations considered in the product's design. Then we consider the Language Side and Application Side of the system respectively. Finally, we discuss a few example queries and how they behave in the system. All code for the system, apart from the source code of the Hesperian wiki itself, is available on our Github¹.

To develop the system we held numerous discussions with individuals working for the Hesperian Society as well as reviewing thousands of queries that were input into its existing search engine. From these queries we selected a diverse subset (Appendix B) to focus on for the system in order to illustrate various features. While we attempted to choose queries that reflect the diversity of input received by the system, they are not representative of the density of each type of query. For instance, single-word queries make up a significant portion of all queries, but analyzing multiple such queries is neither informative nor linguistically interesting. While selecting queries, we did not consider whether they would be practical to implement.

4.1 User Interface

We began our work by considering how the Hesperian Product would be used. Here we discuss two types of users, the site visitors and content editors, as well as their respective workflows. The focus is not the visual design or layout of the user interface but rather the features.

Site Visitors

Site visitors access the wiki only to learn information. They are only able to access public pages and they will frequently retrieve information using the search engine. These users typically are looking for information regarding a specific procedure, disease or symptom and

¹www.github.com/ICSI-Berkeley/ecg_hesperian

the user may exclude relevant information from their initial query. To them, the search engine is a fairly standard black box, out of which they expect to get relevant search results.

To that end, the search interface behaves nearly identically to the default search engine in the wiki. Figures 4.1 and 4.2 shows what this looks like. When the user inputs a query, it is rewritten without their knowledge to produce a new “ECG query”. Both the original query and the ECG query are executed in the search engine, and only the top ECG query result is presented above the user query results. If the system determines it could use clarifying information, then it will present the user with a single multiple choice clarifying question which is placed just above the ECG query result. Answering the question refreshes the ECG result to incorporate the extra information. Within the same session, a user will not be asked the same clarification question twice.

If the ECG System does not contain certain words in its lexicon, it may ask the user for synonyms those words (Figure 4.3). Once the user has filled out a synonym for every unknown word, the system will rerun the query replacing the unknown words with the synonyms. This can repeat as many times as necessary but should not occur too frequently because the ECG system attempts to find synonyms automatically before asking the users.

If there are any errors in the process, an error message is clearly displayed in the ECG result box. Errors can occur either due to connection issues to the ECG System or if the System cannot understand a query.

Content Editors

These users encompass those who edit the wiki content as well as wiki administrators. Note, that they are either Hesperian Society staff or volunteers, not general users. They can edit pages and add content, which may expand the language used on the wiki. They will want the content they add to be accessible by site visitors and so will want to be able to modify the relevant parts of the Hesperian Product. However, they may not be system experts and so modifying the underlying code and grammar would be difficult.

We anticipate one activity that content editors will have to do is add support for new terminology introduced into the wiki. This can be done using the Token Editor [34] in the Workbench, which we improve to make it easier to use. If the morphology entry for that token is missing, then the Token Editor will notify the content editor and she can add an entry for it using the newly developed Morphology Editor (section 4.2). Content editors can figure out which tokens to add by comparing the word frequency on the wiki to the words contained in the ECG System’s lexicon. This is done by running a simple script that crawls the wiki to count words and then checks those against the lexicon before turning the top N results. Of course, this script could be turned into an application with a GUI.

The ECG System also keeps track of synonyms users input when certain words are not understood. If that synonym leads to a valid parse, then the original word is added to a log along with the probable type and properties taken from the synonym’s token. This log of token candidates can then be used to help guide the addition of tokens for terms people are using. Again, this could be turned into an application with a GUI.

Search results

Q severe acute abdominal pain ✕ Search

Improve your results:
What is the gender of the patient?
 Male Female
Submit

[Pain in the Belly or Gut - Hesperian Health Guides](#)
 en.hesperian.org/.../New_Where_There_Is_No_Doctor:Pain_in_the_Belly_or_Gut

 Sudden onset of **severe pain** in the gut, that keeps getting worse, with no diarrhea, is likely **acute abdomen**. **Acute abdomen** can be caused by obstruction, appendicitis, ectopic pregnancy, or other dangerous problems. If you see these **signs**, ...

About 3 results (0.53 seconds) Sort by: Relevance ▾

[Pain in the Belly or Gut - Hesperian Health Guides](#)
 en.hesperian.org/.../New_Where_There_Is_No_Doctor:Pain_in_the_Belly_or_Gut

 woman bent over with **pain**, holding her belly. Sudden onset of **severe pain** in the gut, that keeps getting worse, with no diarrhea, is likely **acute abdomen**. **Acute abdomen** can be caused by obstruction, appendicitis, ectopic pregnancy, or other ...

[Belly Pain, Diarrhea, and Worms - Hesperian Health Guides](#)
 en.hesperian.org/.../New_Where_There_Is_No_Doctor:Chapter_15:_Belly_Pain,_Diarrhea,_and_Worms

 Mar 15, 2018 ... Are there **bowel** movements? **Severe pain** with few or no stools is also a sign of **acute abdomen**. Is there **pain** with nausea and vomiting? This may be from eating spoiled food. Drink plenty of fluids, like rehydration drink. Does the **pain** come after eating? Is there a burning feeling in the chest or belly?

[Kinds of Pain in the Lower Abdomen - Hesperian Health Guides](#)
 en.hesperian.org/.../Where_Women_Have_No_Doctor:Kinds_of_Pain_in_the_Lower_Abdomen


 **Pain** with diarrhea, **intestinal** infection from bacteria or parasites, See 'diarrhea'. **Severe pain** in the first 3 months of pregnancy, often with bleeding that comes and goes. WWWND10 Ch21 Page 355-2.png. pregnancy in the tube, URGENT! Go to a hospital right away. **Severe pain** in the last 3 months of pregnancy, with or ...

Figure 4.1: The result of searching for “severe acute abdominal pain” without providing any additional information.

Search results


Q severe acute abdominal pain ✕ Search

[Sudden, Severe Pain in the Abdomen - Hesperian Health Guides](#)
 en.hesperian.org/.../Where_Women_Have_No_Doctor:Sudden,_Severe_Pain_in_the_Abdomen


Some lower **abdominal pain** is an emergency. If you have any of the following danger **signs**, go to the nearest hospital. A trained health worker will need to do an examination of your **abdomen**, a pelvic exam, and perhaps special tests. See information about how to do an **abdominal** exam and a pelvic exam.

About 3 results (0.53 seconds) Sort by: Relevance ▾


[Pain in the Belly or Gut - Hesperian Health Guides](#)
 en.hesperian.org/.../New_Where_There_Is_No_Doctor:Pain_in_the_Belly_or_Gut

 woman bent over with **pain**, holding her belly. Sudden onset of **severe pain** in the gut, that keeps getting worse, with no diarrhea, is likely **acute abdomen**. **Acute abdomen** can be caused by obstruction, appendicitis, ectopic pregnancy, or other ...

[Belly Pain, Diarrhea, and Worms - Hesperian Health Guides](#)
 en.hesperian.org/.../New_Where_There_Is_No_Doctor:Chapter_15:_Belly_Pain,_Diarrhea,_and_Worms

 Mar 15, 2018 ... Are there **bowel** movements? **Severe pain** with few or no stools is also a sign of **acute abdomen**. Is there **pain** with nausea and vomiting? This may be from eating spoiled food. Drink plenty of fluids, like rehydration drink. Does the **pain** come after eating? Is there a burning feeling in the chest or belly?

[Kinds of Pain in the Lower Abdomen - Hesperian Health Guides](#)
 en.hesperian.org/.../Where_Women_Have_No_Doctor:Kinds_of_Pain_in_the_Lower_Abdomen

 **Pain** with diarrhea, **intestinal** infection from bacteria or parasites, See 'diarrhea'. **Severe pain** in the first 3 months of pregnancy, often with bleeding that comes and goes. WWWND10 Ch21 Page 355-2.png. pregnancy in the tube, URGENT! Go to a hospital right away. **Severe pain** in the last 3 months of pregnancy, with or ...

1

powered by Google Custom Search

Figure 4.2: The result of searching for “severe acute abdominal pain” after clarifying that the patient’s gender is female.

The image shows a search interface. At the top, there is a search bar containing the text "procedure for surgical abortion" and a "Search" button. Below the search bar, a message reads: "We had trouble understanding a few words. Could you please provide synonyms?:". Underneath this message, the word "procedure" is followed by an arrow and the text "synonym...". At the bottom of this section, there is a "Submit" button.

Figure 4.3: If the system cannot recognize a word, like “procedure”, then it asks for synonyms.

4.2 Language Side

Schemas

As mentioned before, in ECG, schemas are used to represent semantics. Therefore in order to design the schemas used in the application, we must look at the meanings being conveyed. In this case, the central concerns of nearly all queries are conditions and treatments. Also, queries often contain useful information relating to the patient in the form of age or gender. To that end we design a few specialized, highly interconnected schemas to capture as much information as possible about these topics (Figure 4.4).

Most of these Hesperian schemas are subcases of a core grammar schema called the RD schema, short for referent descriptor. It is intended to carry the meaning of noun phrases, which is appropriate since most search queries are noun phrases [2]. The three top level schemas are the Condition, Treatment and Patient schemas. The Disease and Symptom schemas inherit from the Condition schema and the Procedure and Drug schemas inherit from the Treatment schema.

These schemas are interconnected via their roles and constraints. For example, a Treatment has a role for the related Condition and the patient of both is constrained to be the same instance of Patient. The other constraints on these schemas are ontological constraints in order to ensure their values satisfy the desired types. For example, the location of a Condition must be a bodyPart and a Patient must be a person.

Apart from the RD-based schemas, we also have the HesperianBagSchema and a number of schemas representing processes, both of which will be discussed in greater detail later.

Types and Tokens

We added several lexical types for the Hesperian Product, largely corresponding to the various RD-based schemas (e.g. ConditionType, ProcedureType, etc.). All the types have multiple roles that can be filled while defining tokens. For example, the ProcedureType

```

schema Condition
  subcase of RD
  roles
    loc: RD
    patient: Patient
    severity: ScalarModifier
    duration: RD
    trigger: RD
  constraints
    patient.ontological-category <-- @person
    severity.value <-- @scalarValue
    loc.ontological-category <-- @bodyPart

schema Treatment
  subcase of RD
  roles
    patient: Patient
    loc: RD
    cond: Condition
  constraints
    patient.ontological-category <-- @person
    loc.ontological-category <-- @bodyPart
    cond.ontological-category <-- @conditionType
    cond.patient <--> patient

```

Figure 4.4: The Condition and Treatment schemas used in the Hesperian grammar.

includes a role for patient gender, so an “abortion” token can specify that the patient must be female.

Because of the flexibility of language, the use of words in the queries does not always align with nice grammar rules. In light of this, some tokens of these types are not usually considered nouns. For example, the word “pregnant” is frequently used like a noun in queries even though it is an adjective. Therefore, there is a token of ConditionType for “pregnant”. ECG has the flexibility to handle these changes but it does require custom morphology entries.

Classification of Queries

We examined a number of different kinds of queries and, while most do not fit into neat categories, we will attempt to describe different types of problems shared between the queries. Members of a class will use many of the same constructions and schemas in their SemSpec. Queries may belong to multiple classifications based on the problems they involve.

Bag-of-Words Queries: This is the largest class of queries. These queries are simply lists of terms thrown together in a fairly meaningless order. Occasionally these queries are separated by “or” or “and”, however, there is still no structure.

Example *“blisters fever baby tummy”*

While we can infer that the baby is a patient and the blisters and fever are symptoms, the language does not provide enough structure to extract more meaning. For example, we cannot conclude that the location of the blisters is on the baby’s tummy. In fact, several related queries show the user was actually referring to “runny tummy” or diarrhea (Appendix B).

Noun-Noun Compounds: Although these appear similar to bag-of-words queries, taken together, they give rise to greater meaning.

Example: *“abortion pills”, “stomach pain”*

In the first example, if the words are taken individually, then it seems like a list of Treatments. However, taken together we realize that the pills are used to perform the abortion. Similarly, in the second example, we can conclude that the location of the pain is on the patient’s stomach.

“Prepositional” Phrases: Prepositions are frequently used to explain a relationship between multiple referents. While the examples below are technically noun phrases when taken in their entirety, they all involve prepositional phrases.

Example: *“injection in vein”, “medicine for cough”*

In the first example, we can learn that the location of the injection is the patient’s vein. In the second example, we can learn that the condition being treated by the medicine is the patient’s cough.

Questions: Although users are almost always searching for the answer to a question, fully formed questions represent a relatively small number of queries. That said, most questions that do appear are one of the wh-types. Alternatively, some are also “can” questions. Questions can include a question mark but usually do not.

Example: *“what is family planning”, “can hookworms affect the scalp?”*

Knowing what is being asked for can help determine the type of response to give. The first example is asking for a description of family planning and the second example wants a yes/no answer to whether hookworms can affect the scalp. For the most part, this information is not all that useful in the information retrieval system demonstrated in this project. However, a more robust or structured information retrieval system could make use of this information.

Actions: This class, generally speaking, can be quite broad. However, the kinds of actions involved in health related queries are more constrained.

Example: *“how to treat malaria”, “how to terminate pregnancy”*

In the first example, the “treat” action involves applying a Treatment on the malaria Condition. In the second example, the “terminate” action involves ending the pregnancy Con-

dition, which means the patient is receiving an abortion Treatment.

Analysis of Queries

In this section we describe how each of the classes of query problems is addressed through the grammar. Bear in mind that there are other issues with queries such as misspellings and improper grammar. The former is handled by a spell check component described in Section 4.2. The latter can be handled in part by loosening constraints and allowing more constructions to unify. However, this cannot be done too much so as to avoid losing all advantages of using ECG.

Bag of Words

All bag of words queries are handled by the HesperianBag construction. The construction has a meaning of HesperianBagSchema and is structured like a linked list with the “this” role set to a noun phrase and the “next” role set to another HesperianBag. Normally, in the core grammar, in order to be considered a noun phrase, a Kernel needs a determiner or to be an identifiable type (e.g. proper nouns). Therefore, we need another construction, Undetermined_Kernel, to cause any Kernel to act like a noun phrase. Because the HesperianBag construction’s constraints are so minimal, it matches extremely easily. Figure 4.5 shows the structure of the HesperianBag, HesperianBagSchema, and Undetermined_Kernel.

Consider the query “blisters fever baby tummy”. Each word will individually have meanings Symptom, Symptom, Patient and RD respectively. These meanings will be contained in a linked list of HesperianBagSchemas. More generally, the inclusion of the HesperianBag and Undetermined_Kernel also make the analysis process much less brittle than previous ECG applications. This is important since the language usage patterns in queries are significantly less constrained and the requirements of the application are flexible. Of course, in an application like robotic control, where precision is critical, this degree of flexibility would not be built into the grammar [10].

Noun-Noun Compounds

Previous work on the core grammar developed the NounNounCompound construction and NounNounModifier schema. To handle noun-noun compounds in the Hesperian setting, we subcase NounNounCompound into a number of new constructions such as LocatedCondition (Figure 4.6), to handle “stomach pain”, and ProcedureDrug, to handle “abortion pills”. The appropriate roles within each schema are connected to create the combined meaning. In LocatedCondition, the location of the pain Symptom is set to the stomach RD. In ProcedureDrug, the drug used to perform the abortion procedure is set to the pills Drug.

```

schema HesperianBagSchema
  roles
    this: RD
    next: HesperianBagSchema

construction HesperianBag
  subcase of RootType
  constructional
  constituents
    np: NP
    optional cc: CoordinatingConjunction
    optional hb: HesperianBag
  form
  constraints
    np.f before cc.f
    cc.f before hb.f
  meaning: HesperianBagSchema
  constraints
    self.m.this <--> np.m
    self.m.next <--> hb.m
    self.m.this.ontological-category <-- @hesperian

construction Undetermined_Kernel
  subcase of NP
  constructional
  constituents
    k: Kernel
  constraints
    self.features <--> k.features
  meaning
  constraints
    self.m <--> k.m

```

Figure 4.5: The HesperianBag construction and schema and the Undetermined_Kernel construction.


```

construction Treatment-For-Condition
  subcase of NP
  constructional
  constituents
    trmt: NP
    for: For-PP
  form
  constraints
    trmt.f before for.f
  meaning: Treatment
  constraints
    self.m <--> trmt.m
    trmt.m.ontological-category <-- @treatmentType
    self.m.condition <--> for.np.m
    for.np.m.ontological-category <-- @conditionType

construction LocatedCondition
  subcase of NounNounCompound, NP
  meaning: Condition
  constraints
    self.m <--> head.m
    self.m.location <--> mod.m
    head.m.ontological-category <-- @conditionType
    mod.m.ontological-category <-- @bodyPart

```

Figure 4.6: The Treatment-For-Condition construction and the LocatedCondition construction.

“Prepositional” Phrases

The core grammar already contains various constructions to handle prepositional phrases. Combining these prepositional phrases with another Hesperian noun phrase works very similarly to the noun-noun compounds. We have different constructions for handling different valid noun phrase and prepositional phrase pairs.

In the case of “injection in vein”, the Prep-Locative-Treatment construction connects the location of the injection Treatment with the meaning of a locative prepositional phrase. For “medicine for cough”, the Treatment-For-Condition (Figure 4.6) construction connects the condition of the medicine Treatment to the meaning of a “for” prepositional phrase.

Questions

The core grammar contains nearly all the necessary constructions for questions already. For wh-questions, the wh-word is treated as a noun phrase. Therefore the SemSpec for the query,

“what is family planning”, looks very similar to that of a declarative statement regarding a referent called what. Of course, there still are markers to indicate that it is a question.

For “can” questions, there are existing constructions for sentences with auxiliary inversion. In this case, the auxiliary is “can”. The SemSpec for “can hookworms affect the scalp?” will have hookworms as the participant in the verb phrase, “affect the scalp”. Again, the SemSpec is marked to indicate that this is a yes/no question.

Actions

Once more, a lot of the structure needed to interpret actions comes from the core grammar. Numerous general constructions already exist for things like transitive actions or copulas (e.g. “is”). Many more common actions also did not require any specific implementations because they are based on relatively primitive schemas. In the query “what is family planning”, the copula constructions and schemas used come entirely from the core grammar.

However, there are still a number of Hesperian specific actions. For example, the act of treating involved in “how to treat malaria”, requires a specific TransitiveTreatment construction with the meaning of TreatmentProcess schema. The specific implementation allows the appropriate roles to be bound together within TreatmentProcess.

Similarly, the query “how to terminate pregnancy” involves the TransitiveAbortion construction with a meaning of AbortionProcess. The TransitiveAbortion construction is a subcase of the TransitiveCauseToEnd construction that is specifically for when the thing being ended is a pregnancy. While it may not always be useful to build constructions that are this specific, having more robust coverage of abortion-related language would be useful for this application. Since a large number of queries on the Hesperian Wiki relate to abortion, without such a specific construction, it can be tricky to figure out that such a query is talking about abortion.

Specializing Queries

The Specializer is responsible for pulling out task relevant information from the SemSpec. The Hesperian Specializer requires only two changes: handling the RD-based schemas in a special way and logic to handle the HesperianBagSchema. All other changes are made directly to the ActSpec templates and are discussed in section 4.3.

Normally, RDs and their subtypes match a template called an objectDescriptor template. However, since the RDs we use in our grammar are much more specific, we choose to use more specific descriptor templates. These templates correspond to the RD-based schemas themselves (e.g. conditionDescriptor, patientDescriptor, etc.) and contain the corresponding roles. To accommodate this we simply overwrite one function in our Hesperian Specializer to use the appropriate descriptors.

We use a HesperianBagDescriptor template to mimic the linked-list structure of a HesperianBag. We add a special control path in the Hesperian Specializer to copy over the linked list.

```

schema TreatmentProcess
  subcase of Process
  roles
    treatment: Treatment
    patient: Patient
    condition: Condition
    actionary: @treatmentVerb
  constraints
    patient <--> treatment.patient
    condition <--> treatment.condition
    condition.patient <--> patient

schema CauseToEnd
  subcase of Process
  roles
    endedThing

schema AbortionProcess
  subcase of CauseToEnd
  roles
    treatment: Procedure
    patient: Patient
    condition: Condition
    actionary: @end
  constraints
    patient <--> treatment.patient
    condition <--> treatment.condition
    condition.patient <--> patient
    condition.ontological-category <-- @pregnancy
    treatment.ontological-category <-- @abortion
    patient.gender <-- @female
    endedThing <--> condition
    protagonist <--> patient

```

Figure 4.7: The schemas for TreatmentProcess, CauseToEnd and AbortionProcess.

Core System Improvements

To accommodate challenges presented by this new domain, we made a number of changes to language side aspects of the system that will ultimately make their way back into the core version of the system (Figure 2.2). These changes include modifications to the Analyzer, adding preprocessing steps to the UI-Agent and improving the tools in the Workbench.

Multiword Token Support

One feature that is particularly useful in medical applications is support for multiword tokens. These are tokens where both words are needed together to give it meaning. For example, “pink eye” (also written “pinkeye”), would previously have been parsed as being an eye that is pink in color. However, this should instead refer to the disease. In accommodating this requirement, we focused simply on two word tokens.

We modified the token matching step in the analyzer. In this step the analyzer splits the input utterance by spaces and then matches each word to one in its lexicon. Instead of simply considering the current word, the analyzer now also considers the combination of the current word and the subsequent word. We allow tokens to include an underscore to indicate when it is a multiword token. For example, putting the token “pink_eye” in the tokens file would cause the analyzer to match the pair of words, “pink eye”, during its token matching step.

Spell Checking

When dealing with user input, words can frequently be spelled incorrectly and previously this caused the System to fail on the input. We fix this by adding spelling correction that attempts to replace misspelled words with words the system understands. To accomplish this, we use a Python package, `pyenchant`², that provides access to the `Enchant`³ spelling library.

In the UI-Agent, before an utterance reaches the Analyzer, every word in the utterance is checked to see if it is in the English language and is in the Analyzer’s lexicon. If both conditions are true or only the second condition is true, the word is left as is. If only the first condition is true, then it is handled by the synonym matching component described in the next section. If neither condition is true, then the word undergoes spelling correction.

The `pyenchant` package provides an API to create your own word lists and comes with a word list for English. If an input word is not in the word list, `pyenchant` suggests similar words from the word list. We use `pyenchant` to create a word list from the Analyzer’s lexicon and then use the top suggestion from that list to replace the misspelled word. If there are no suggestions from the Analyzer’s lexicon, then we take the top suggestion from the English language and pass that suggestion to the synonym matching component. If there

²www.github.com/rfk/pyenchant

³www.abiword.github.io/enchant/

are no suggestions, then we use the synonym matching fail state logic, described in the next section.

Synonym Matching

User input may not match any particular word stored in the Analyzer's lexicon, but there may be another word with the same meaning in the lexicon. In such cases we try to map from the unknown word to a word in the lexicon. We do this by first part-of-speech tagging the sentence using NLTK⁴. Then the unknown word along with its part-of-speech tag is passed into WordNet [14] in order to get synonyms. Once a synonym is recovered, the part-of-speech tag and the grammar's morphology are used to recover the correct wordform. Finally, the resulting word is checked to see if it appears in the lexicon. The first synonym to appear in the Analyzer's lexicon is used. This is not unlike what is done in [39].

If a word cannot be fixed by spelling correction and synonym matching, then it is added to a list of failed words. The list of failed words is then returned to the wiki and displayed to the user in conjunction with a request to either rephrase their query or provide synonyms for those particular words. If they provide synonyms, then the failed words will be replaced in the query before running it through the system again. If all the words in the modified query can be matched to the lexicon, then the failed words are added to a log of candidate tokens. Along with the failed words, we log the type and role values of the successful replacement word since they will likely be the same for the failed word. A system administrator can then use the log to guide the addition of new tokens.

Improved Token Editor and Morphology Editor

To make adding tokens easier for non-experts, we improve the existing Token Editor [34] in the ECG workbench. The new iteration of the Token Editor now allows users to view and delete added constraints to a token. Additionally, it checks to see if there is a corresponding morphology entry. If none is present, then the user has the option now of creating a morphology entry using another tool.

The Morphology Editor allows the user to enter a lemma along with its parent type. Upon selecting the parent type, the tool then presents the user with a list of valid inflections for which users can enter word forms. When the user submits, the tool adds a corresponding entry to the desired morphology file.

A simple web crawler script, implemented using the Scrapy⁵ framework, allows us to retrieve frequencies of all words in the wiki. This list is then cross-checked against the Analyzer's lexicon to determine which tokens need to be added. A similar procedure can be implemented for words from a collection of user queries. This would allow a developer or system administrator to quickly identify and prioritize which tokens to add.

⁴www.nltk.org

⁵www.scrapy.org

Token Editor

Enter Lemma:

Select Parent Type:

Select Ontology File:

Select Token File:

Select Role to Modify:

Set Role Item:

Role	Item
self.m.ontological-c...	@abortion
self.m.patient.gender	@female

Additional ontology parents (optional):

Application mapping (optional):

Morphology Editor

Enter Lemma:

Select Parent Type:

Select Morphology File:

Select Inflection to Modify:

Set Wordform:

Inflection	Wordform
Singular	pink_eye
Plural	pink_eyes

Figure 4.8: Top: Token Editor while adding token for “abortion”. Bottom: Morphology Editor while adding entry for “pink_eye”.

```
{'condition': [('pregnancy', 3)],
 'gender': [('female', 3)],
 'symptom': [('bleeding', 1)],
 'symptom_duration': [('1..4.0 weeks', 1)],
 'treatment': [('abortion', 2)]}
```

Figure 4.9: The information extracted from the query “bleeding for four weeks after an abortion”

4.3 Application Side

Action Specification Template Design

The Action Specification, or ActSpec, defines the interface between the Specializer and the Problem Solver. It is entirely defined through templates which declare what information the Specializer should extract and send to the Problem Solver. The core system already contains a number of ActSpec templates to facilitate the more general and common elements of language. While the core templates remain largely unchanged, we add a few templates to address the unique needs of this application.

As mentioned in Section 4.2, we add several new descriptor templates, inheriting from the general objectDescriptor, to address the different RD-based schemas. These descriptors have roles corresponding to their respective schemas. Additionally, we add a hesperian-BagDescriptor template to copy the linked-list structure of the schemas.

We also add parameter templates for each of the new actions. Again, there is nothing too surprising here as their roles correspond directly to the relevant roles in the corresponding schemas.

Query Information Extraction

The job of the Hesperian Problem Solver is primarily to convert an ActSpec into a query. We separate this process into two steps, information extraction and query rewriting. Our approach to information extraction mimics the typical Problem Solver design used in previous ECG products. Every template is handled by a specific function with the function calls following the structure of the ActSpec.

When a function encounters a fact worth keeping, it is saved to a global dictionary. For example, if a symptomDescriptor states that the type of a symptom is “pain”, then the value “pain” will be stored under the key “symptom” in the dictionary. Since a particular field may be encountered multiple times within an ActSpec, with differing values in different places (e.g. “blisters fever baby tummy” has two symptoms) the dictionary stores values for each key as a set, with new values being added to the set. Once all the information is extracted, it can be saved to the user state as well as used for building a query.

Figure 4.9 shows the information extracted for the query “bleeding for four weeks after an abortion”. The corresponding ActSpec is in Appendix C. In its current format, the extracted information loses a lot of information from the ActSpec’s structure. For example, it is not clear from the extracted information alone that the “abortion” was the trigger for the “bleeding”. This fact is available in the ActSpec, but we do not save information at that granularity simply because it is not useful with our current information retrieval system, Google Search. However, clearly there is a considerable amount of structured information available that could be used by a sufficiently robust information retrieval system.

Query Rewriting

The limitations imposed by our current information retrieval system play a role in how queries are rewritten. Specifically, queries are first written purely as bags of words consisting of every combination of information taken from the ActSpec. These bags are then ranked based on a score assigned to their parts. Queries are then executed in decreasing order of score until one returns a result.

The terms used in the constructed queries are combinations of the extracted information (e.g. Figure 4.9) as well as additional terms known to be useful in the context of the Hesperian wiki. For instance, although a user may be asking about “symptoms”, the wiki frequently uses the term “signs” as well, so we add the term “(symptoms—signs)” to the query. In other cases, we replace the fact with a more relevant term. If the user is female, then results corresponding to the book “Where Women Have No Doctor” should be prioritized so we use the “where_women_have_no_doctor” instead.

The scores of a query are given by the sum of the scores of their component terms. The base value of each term is determined based on its key in the extracted information. For example, “symptom” receives a higher base score than “symptom_location”. The base values used are determined experimentally but are based on the idea that queries revolve around the conditions and treatments and other information is extra. None of the base values are negative, so the highest ranked query is always the one with all the terms. Subsequent queries progressively remove the least valuable terms.

Notice that, in addition to the values in Figure 4.9, the tuples also contain numbers. These numbers indicate the depth of that fact within the ActSpec. This is a crude metric for the importance of that piece of information to the query but works decently well in practice. In the case of Figure 4.9, we know that the patient was previously pregnant, hence the abortion, but the pregnancy is not particularly relevant to this query. It also is stored quite deep in the ActSpec (see Appendix C), because it is not immediately related to the “bleeding” symptom. The depth of the fact weights the value of the corresponding term, with deeper facts being exponentially worse than shallower ones.

In practice, the first query, which contains all the terms, usually finds a search result. However, when a result is not found, the scores of the queries are very useful in creating follow up queries until a result is found. In the case of the query “can hookworms affect the scalp?”, the system attempts two queries:

hookworm (disease|symptom|sign) scalp affect
hookworm (disease|symptom|sign) affect

For this query the system has extracted the facts that the disease is “hookworm”, the action is “affect” and the affected entity is the “scalp”. The first query includes all the information of the original plus an additional term for diseases. However, it does not produce any results. The second query removes the location “scalp” because it is the least relevant portion of the query. The result is a page about worms and hookworms. In this case, the best result does not fully answer the question but it does address as much of the question as possible.

Hesperian Wiki Extension

The Problem Solver communicates with a custom Mediawiki extension installed into the Hesperian Wiki. The extension implements the ECG System’s communication protocol, provides information to the Problem Solver, receives responses from the Problem Solver, executes queries and displays the information to the user. While this may seem like a large number of functions, the development and integration process is fairly painless thanks to Mediawiki’s robust extension support and the ease of implementing the ECG System’s communication.

Since each component of the ECG system runs as a separate process, it relies on a general IPC solution called Transport [34] which works by broadcasting messages within a subnet. We use a component called a Bridge server to connect multiple subnets by copying messages from one subnet and rebroadcasting them on another subnet. In order for the Mediawiki extension to communicate with the Problem solver, it simply opens a socket directly to the Bridge server to send and receive messages.

The extension inserts Javascript into the page to capture the user search query and execute it in Google as well as send it to the Problem Solver. The Problem Solver returns the improved queries and also any clarification questions. The Javascript then executes the improved query as well as renders any clarification questions. If a user responds to a clarification question, that information is sent to the problem solver, along with the query and it is run through the system once more before returning improved results.

Clarification Questions

If the system is missing a piece of user information that might be useful, it can ask a clarification question. This can be used to resolve ambiguous input or to refine results. In its current form, the clarification questions are only asked when a piece of information is missing (e.g. gender), without regard to the information’s utility for that particular query. However, the system could easily be modified to ask the questions under more specific conditions.

Clarification questions are defined by a template specifying the text of the question, the field being determined and the option values. This produces a multiple choice question for the user.

For the query “severe acute abdominal pain”, the top result points to a general page about pain in the belly or gut (Figure 4.1). However, when it is clarified that the patient’s gender is female, then the improved result deals with “Sudden, Severe Pain in the Abdomen” for women (Figure 4.2). Clearly these kinds of clarification questions can be useful, and with an information retrieval system where it is more clear what information would be useful to narrow the search, it would be possible to better tailor these questions to the situation.

When the system fails to recognize some words in the query, it will ask the user to provide synonyms (Figure 4.3). When users input alternatives, the query is rerun through the ECG System using the alternatives in place of the failed words. If the alternatives succeed, the failed words are recorded as candidate tokens (Section 4.2).

User State

In order to retain information over the course of the interaction, the ECG system maintains state about the user. This state is associated with the browsing session and the association is destroyed when the session is over or after a period of inactivity due to the sensitive nature of medical information. Maintaining user state allows the ECG system to use previously learned information in future queries and it allows the system administrators to group related search activity and any associated feedback together.

User information consists of information gleaned from their queries, their feedback and the pairing between original and improved queries. When a user first enters a query, a session ID is associated with their browser. This session ID is then associated with a randomly generated user ID within the ECG System. As they interact with the search engine, the session ID is used to retrieve the related user ID and store information in a key-value store. If in the process of improving a query, a piece of information would be useful, then the Problem Solver will check the user information before generating a clarification question. This prevents asking for previously provided information during subsequent queries.

The current implementation assumes the information will not change within a session. If the user ever provides conflicting information, the current implementation overwrites the previous information. When the user leaves the wiki, their session ID is lost. A new session ID and user ID will be created the next time they visit the wiki and use the search engine. While we still retain the information recorded from previous users, we are no longer able to relate it to a particular session or individual.

4.4 Example Queries

We describe the Hesperian Product by working through several examples taken from query logs of the previous Hesperian wiki. The list of examples we focus on is available in Appendix B. Here we discuss a few of these examples in greater detail. In this section we mention several

schemas and constructions that are not included as Figures. While their names are mostly self-explanatory, they can be viewed in their entirety on our Github repository⁶.

Consider the query “can hookworms affect the scalp?”. In the Analyzer, this query is associated with a CauseEffect schema because of its form. The cause is the hookworms Disease and the affected thing is the scalp RD. In this case, the particular constructions used do not connect the scalp RD to the hookworms’ Disease location. The SemSpec information is preserved through the Specializer and ActSpec, and is available to the Problem Solver. During the query information extraction phase, these facts are extracted: the question is a “can” question, the disease is hookworms, the scalp entity is involved, and the action is to affect. The queries presented in Section 4.3 are generated based on these facts. Notice, that extra terms are added to help the disease match. Since the “scalp” is only a generic object, it has the lowest priority in the query and is the first term removed, leading to the second query attempted being successful in returning a result corresponding to a page on hookworms. This improved query is better because the original query returns no results.

Now consider the query “bleeding for four weeks after an abortion”. This query is represented by a Symptom schema for “bleeding”. Using the Condition-For-Time construction, we associate the four weeks with the duration of the Symptom. Using the ConditionAfterTreatment construction, we associate the abortion Treatment with the trigger of the Symptom. This information is encoded in a SymptomDescriptor ActSpec and given to the Problem Solver. Appendix C shows the ActSpec for this query and Figure 4.9 shows the information that is extracted from the query. Despite this extra information, the top result of the rewritten query and the original query is the same page on what to expect after an abortion. That said, the wiki does not contain pages that would better address the query.

For the query “alcoholism”, the meaning is a Disease schema. Because there is not much extra information available, the only fact extracted from the final ActSpec is that the disease is alcoholism. Like the hookworms example before, we add extra terms to the query to match the disease, making the improved query “alcoholism (disease|symptom|sign)”. Unfortunately these hurt the result rather than help. The top result for the original query is a page on alcohol and drug problems, whereas the top result for the improved query is a page on common cancers. This is because the page on cancers discusses how alcoholism and disease can lead to cancer. Usually the extra terms help, but in this case, the words “alcoholism” and “disease” are not used together frequently in the right context.

For the query “severe acute abdominal pain”, Figures 4.1 and 4.2 show how the results can vary considerably if we know the user’s gender. Both are still represented in the Language Side the same way, as a Symptom. However, in the Problem Solver, the gender information is included by adding a term for the book Where Women Have No Doctor. This leads to the more specific result. The gender information can come from either a clarification question or the user state. If the user responds to the clarification question, the query is re-run through the system, and the clarification information is saved to the user state for future use.

Now consider the query “abortion pills causes severe pain in left abdomen”. This query

⁶www.github.com/icsi-berkeley/ecg_grammars

uses a CausalAction schema where the abortion Procedure is the cause and the pain Symptom is the effect. After passing through the Language Side, this query produces an ActSpec which contains a number of facts but one particularly interesting one is that the patient gender is female. Although this information is not explicitly present in the query, we encode it in the “abortion” token in the grammar; the patient of an abortion must be female. This fact is used in improving this query but is also recorded to the user state for use later (e.g. “severe acute abdominal pain”). The final result produced by the improved query is a page on abortion complications which is better than the page on abdominal pain for the original query.

Finally, consider the query “procedure for surgical abortion”. Here we intentionally omit a token for the word “procedure”. Therefore, the system has trouble answering the query, and in this case does not find any synonyms. Therefore it must ask the user to provide a synonym (Figure 4.3). If we provide the synonym “operation”, the query “operation for surgical abortion” will be run through the system. Again we intentionally omit a token for “operation”. However, this time a synonym, “surgery”, is found. Thus the query that ultimately goes to the Analyzer is “surgery for surgical abortion”. The final result of the improved query and the original query is the same, a page about abortion complications. Alternatively, if we provide a synonym for “procedure” that does actually have a token, such as “treatment”, then a candidate token entry for “procedure” will be added to a log file using the values taken from the “treatment” token. The query that passes through the system in this case is “treatment for surgical abortion”. Again, the result is the same page on abortion complications.

Altogether, these queries illustrate how the system works and performs. By looking at these cases, we see what information is available at any given stage and verify the system works as expected.

Chapter 5

Related Work

Much of the broader context for our work comes from the semantic search and query understanding literature. Query understanding, as defined by [7], concerns the representation and intent discovery of a query. Related research directions include query parsing, query rewriting, exploiting user context and query classification. These typically occur as preprocessing steps prior to executing a query [26].

A lot of work has been done in the area of semantic annotation of queries in order to relate query elements to concepts and categories [17, 24, 23]. In [36] they use external knowledge base signals related to modifiers to better identify the concepts related to entities and produce better semantic annotations. This is not unlike using ontological constraints in ECG [32]. [22] specifically discusses a method for semantically tagging noun phrases which represent a large number of queries. [27] uses a domain-dependent grammar along with an SVM to rank the resulting parse trees in order to semantically tag queries. Several papers apply semantic annotation towards the creation of SPARQL queries to get information from knowledge bases [18, 35, 37]. Additionally, the results of semantic annotation can help identify user intent [19, 38].

Regarding query modification, [20] introduces the idea of query substitution, where a new query is generated to replace the user's original query. They show this can be an effective way to improve search results. In [39] they show that modifying queries to use synonyms from WordNet [14] can also lead to better matches to target documents.

Other work establishes the utility of query context. In both [5] and [15] they develop a model for query classification based in large part on previous search history.

Within this context, our work utilizes the ECG System to tackle many of the problems related to query understanding withing a specific domain. It is most closely related to work involving semantic annotation, but due to the nature of ECG, we operate on richer semantic representations. Also, our approach taken to information retrieval differs from those taken by most other work due to the facts that we are operating against a purely unstructured data source and we do not have direct control over Google Search.

Previous applications of the ECG System have focused more on the command and control of autonomous systems. A considerable amount of research has been done on applying the

ECG System in the context of human robot interaction [10, 34, 33, 21, 9, 8], using both the MORSE and ROS packages for controlling the robot. The ECG System has also been used to control the real-time strategy video game, Starcraft [10]. The application and corresponding language described in this paper differ considerably from those of previous efforts. Most notably, language in queries is frequently short and consists mostly of noun phrases [2] or unstructured collections of words, rather than being fully formed sentences. In addition to the flexibility in the language, the parameters of the application are also considerably less precise than in command and control-type applications. Further, the language used here primarily concerns healthcare topics.

Prior to the development of the ECG System, a significant amount of work was done in the development of ECG [3, 4, 6, 29, 28]. The contributions of these works underlie the Language Side of the ECG System.

Chapter 6

Conclusion

6.1 Contribution and Limitations

In this work we demonstrate how the ECG System can be retargeted for query understanding on a healthcare wiki. We show how different modifications and features, useful in this domain, can be integrated into the system. In doing so we develop several useful improvements to the core ECG System and to the Hesperian Society’s live wikis. We also show how query information extracted by the ECG System can be used to get a clearer picture of the user’s needs and can be used to rewrite the user’s queries. Crucially, this is all done within the limitations of the existing Hesperian wiki application, demonstrating the ECG System’s flexibility as an integration-friendly natural language understanding system.

Many of the limitations of this system are the same limitations of other ECG products. It is inherently domain constrained, so queries that fall outside that domain will likely fail. Queries within the domain that are not covered by the grammar, will also fail. Also, some queries that require more world knowledge than can be encoded in the language alone can be difficult to handle. For instance, a query like “i am not worth anything” would require the Application Side to infer that the query relates to mental health. Additionally, the current implementation of the Analyzer frequently takes several seconds to return a result, making it too slow for a production search engine.

Also, there are significant limitations to our approach to information retrieval. Since, we ultimately use Google Search, a lot of how the query is actually processed is black-boxed. Further, by rewriting queries into bags of words we forfeit a lot of the information present in the ActSpec. In future work we discuss various ways these approaches can be improved.

6.2 Future Work

There are numerous directions for further exploration with this system. Foremost among them is using a different information retrieval system. Using an alternative search engine would allow us to avoid black-boxing the retrieval process. On the most basic level, this would

allow us to directly apply our term weights to the retrieval process. We could also explore new ways of using the extracted information in the search process. For example, we could use the information as inputs in a learning to rank algorithm [25] and compare it to a model without those inputs. Another approach would be to use knowledge extraction techniques to associate structured information with pages on the wiki, and use those structures to help rank pages.

We could also apply this approach to query suggestion and substitution. Queries that are asking for the same thing may get completely different results because they look different to the search engine. However, once passed through the ECG System, they would look much more similar. Combining this with user activity logs would allow us to develop a system that recommends the most effective queries. This approach could also be used to replace a query with a more effective version.

With regard to the ECG System, further work can be done in a multitude of directions. Expanding the grammar within this application would test and help refine the packages in the core grammar. Applying the ECG System to other unrelated domains and applications would similarly lead to improvements in the core System. Also, the Analyzer, could be reworked to be made faster by using resources more efficiently or adding parallelism.

6.3 Concluding Thoughts

In this work we demonstrate the utility of natural language understanding to query understanding in a specialized domain and also test the flexibility of the ECG System. As natural language becomes an increasingly important part of human computer interaction, natural language understanding will play a bigger role in computer applications. Integrating such capability into query understanding systems, among other areas, is an important step in providing an accessible user experience. As a result, robust and flexible natural language understanding systems will be necessary and the design adopted by the ECG System provides an effective template for such systems.

Appendix A

Wiki Upgrade Instructions

These are the instructions we created for upgrading the Hesperian Wiki. As a result some of the steps are relevant only for the Hesperian Wiki. Wherever possible, we have tried to make the steps more generally applicable. These instructions assume the existence of a running version of the wiki that needs to be upgraded and that the skin has already been refactored for the new wiki version. If a step is optional, it is specifically marked as being OPTIONAL.

1. Download mediawiki 1.29.1 from [here](#) and extract it to the desired location. Copy over `LocalSettings.php`, the `images` folder and the `skins/common` folder from previous wiki files.
2. Copy over any extensions from the previous wiki into the new wiki only if they are the latest version and are still compatible.
3. Download the updated versions of any extensions and extract them to `/extensions` in the new wiki.
4. Remove dependencies for any extensions that were not transferred over.
5. Copy the refactored skin into the `skins` folder of the new instance of the wiki. We assume the skin is named “hhgskin”.
6. Add the following line to `LocalSettings.php` in order to load the wiki skin.
`wfLoadSkin('hhgskin');`
7. Run the following command in order to update the database for the latest version of MediaWiki.
`php5.6 <path_to_wiki>/maintenance/update.php`
8. (OPTIONAL) Rebuild the default search index by running the following command.
`php5.6 <path_to_wiki>/maintenance/rebuildtextindex.php`

9. At this point, visiting the wiki site in a browser should result in a page that resembles the final wiki. If there are any minor visual issues, make sure to delete browser caches, purge the MediaWiki cache and possibly restart the machine running the wiki.
10. Copy the GoogleSiteSearch extension into the extensions folder. You can download the extension from [here](#).
11. In LocalSettings.php place the following lines, replacing 'YOUR_CSE_ID' with the ID corresponding to Google Site Search for this wiki. This adds Google Search and disables the built in search.


```
wfLoadExtension( 'GoogleSiteSearch' );
$wgGoogleSiteSearchCSEID = 'YOUR_CSE_ID';
$wgGoogleSiteSearchOnly = true;
```
12. Now got to the configuration for site search on Google and change the Layout to “Full Width” under “Look and Feel” settings. Customize the colors used for the search results here too.
13. Customize the title that appears before the search results by changing the values in `ECGSearchInterface/i18n`.
14. Now the search results should appear normally but be from Google. When we did this, we encountered a bug where the search input field was missing. If you encounter this problem, you can fix it in one of two ways:

- a) The HTML for the search box is already present but has a `display:none`; inline style. This can be fixed by adding the following Javascript to any script that will load onto the search page.

```
$(document).ready(function() {
  if ($('#mw-googlesitesearch-container').length != 0) {
    if ($('#mw-googlesitesearch-container form.gsc-search-box').length != 0) {
      $('#mw-googlesitesearch-container form.gsc-search-box').show();
    } else {
      $('#mw-googlesitesearch-container').on('DOMNodeInserted', function() {
        $('#mw-googlesitesearch-container form.gsc-search-box').show();
      });
    }
  }
});
```

- b) The other option is to create your own search input form. Unlike option (a), this will cause a full page refresh on every search conducted but it will also be more customizable. One way of doing this is by adding an extension that is loaded before GoogleSiteSearch that runs on the `SpecialSearchResultsPrepend` hook. It should behave like the following code.

```

use MediaWiki\Widget\SearchInputWidget;

public static function searchPrepend( $specialSearch, $out, $term ) {
    $out->enableOOUI();

    $searchWidget = new SearchInputWidget( [
        'id' => 'searchText',
        'name' => 'search',
        'autofocus' => trim( $term ) === '',
        'value' => $term,
        'dataLocation' => 'content',
        'infusable' => true,
    ] );

    $layout = new \OOUI\ActionFieldLayout( $searchWidget, new \OOUI\ButtonInputWidget( [
        'type' => 'submit',
        'label' => $specialSearch->msg( 'searchbutton' )->text(),
        'flags' => [ 'progressive', 'primary' ],
    ] ), [
        'align' => 'top',
    ] );

    $out->addHTML(
        Xml::openElement(
            'form',
            [
                'id' => 'search',
                'method' => 'get',
                'action' => wfScript(),
            ]
        ) .
        '<div id="mw-search-top-table">' .
        $layout .
        '</div>' .
        "<div class='mw-search-visualclear'></div>" .
        '</form>'
    );

    return true;
}

```

Alternatively, another option for search is ElasticSearch. Instructions for that installation are presented here and continue from step 9 above.

1. Install ElasticSearch according to the instructions presented [here](#) or according to the steps below. This guide installs ElasticSearch on the same machine that runs the wiki but its possible to install it on a different machine.

- a) Install apt-transport-https using the following command:

```
sudo apt-get install apt-transport-https
```
 - b) Add elasticsearch to the sources as follows:

```
echo "deb https://artifacts.elastic.co/packages/5.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elasticsearch-5.x.list
```
 - c) Install elasticsearch with the following command:

```
sudo apt-get update && sudo apt-get install elasticsearch
```
 - d) Install the java runtime environment

```
sudo apt-get install default-jre
```
 - e) Start the service with the following command:

```
sudo systemctl restart elasticsearch.service
```
 - f) Wait a few minutes and verify the service is running with the following command:

```
curl -XGET 'localhost:9200/?pretty'
```
2. Download the installer for Composer from [here](#) and run it to generate a composer.phar file. It will be needed for installing the Elastica extension.

```
php5.6 installer
```
 3. Download the Elastica and CirrusSearch extensions from [here](#) and place them in the extensions folder of the wiki. From within the Elastica directory run the following command:

```
php5.6 <path_to_phar>/composer.phar install --no-dev
```
 4. Follow the instructions [here](#) or below to complete the CirrusSearch setup.
 - a) Place the following lines in the LocalSettings.php file:

```
wfLoadExtension( 'Elastica' );  
require_once( "$IP/extensions/CirrusSearch/CirrusSearch.php" );  
$wgDisableSearchUpdate = true;
```
 - b) Run the following script:

```
php5.6 extensions/CirrusSearch/maintenance/updateSearchIndexConfig.php
```
 - c) Remove this line from LocalSettings.php:

```
$wgDisableSearchUpdate = true;
```
 - d) Run the following commands. They could take a while to run.

```
php5.6 extensions/CirrusSearch/maintenance/forceSearchIndex.php  
--skipLinks --indexOnSkip  
php5.6 extensions/CirrusSearch/maintenance/forceSearchIndex.php  
--skipParse
```
 - e) Finally, add the following line to LocalSettings.php:

```
$wgSearchType = 'CirrusSearch';
```

Appendix B

Selected Queries

Below is a table of all the queries we focused on while building the system. These queries were selected from a much larger set of search logs from the previous Hesperian wiki. The second column says whether query works in the system. The third column says whether the result is better, the same or worse given that the query parses. We determine the quality of the result by manually comparing the top result of both searches and seeing which is more relevant to the query. The improvement can be judged to be “same” if either both the top results are the same page or if both the top results are similarly relevant.

Query	Parses	Improvement
what is a tapeworm?	Yes	Same
can hookworms affect the scalp?	Yes	Better
blisters fever baby	Yes	Better
blisters fever baby or tummy	Yes	Same
blisters fever baby tummy	Yes	Same
blisters on to tongue on lips and around mouth feverish runny tummy among 2 years old	No	
blisters or tongue or lips or mouth or feverish or tummy	Yes	Same
blisters tongue lips mouth feverish runny tummy	Yes	Better
blisters tongue lips mouth feverish runny tummy 2 years	No	
blisters tongue lips mouth feverish runny tummy 2 years old	No	
blisters tongue lips mouth feverish tummy	Yes	Same
bleeding 4 weeks after abortion	Yes	Same
bleeding 5 weeks after abortion	Yes	Same
bleeding after abortion	Yes	Same
bleeding at a abortion	Yes	Same

bleeding for four weeks after an abortion	Yes	Same
bleeding four weeks after abortion	Yes	Same
bleeding weeks after abortion	No	
pregnancy missed person spotting lower backache constipation	No	
pregnant and symptoms	Yes	Better
pregnant missed period spotting 1 day lower backache constipation	No	
pregnant missed period spotting lower backache constipation	No	
pregnant not bleeding symptoms	No	
pregnant or symptoms	Yes	Better
pregnant period spotting backache constipation	Yes	Same
pregnant period spotting lower backache constipation	No	
pregnant period spotting or backache constipation	Yes	Same
pregnant period spotting or backache or constipation	Yes	Same
after giving birth how many years should i start giving birth again	No	
after mining birth how many yeast solid i start mining birth again	No	
for how long can we wait to get pregnant after c section operation	No	
for how long can we wait to have another baby after c section operation	No	
pain in lower part of stomach	No	
pain in my abdomen after 8 months of abortion	Yes	Same
pain in right side extending to leg	No	
pain in the right lower abdomen after an abortion	Yes	Better
what can i give a 2 year old for diarrhea	No	
abortion pills causes any pain in lower abdomen after year	No	
abortion pills causes pain in left abdomen	Yes	Better
abortion pills causes seeds pain in left abdomen	Yes	Better
abortion pills causes severe pain in left abdomen	Yes	Better
after take the medicine to destroy the one month pregnancy	No	
can a breast feeding mother use implant method of family planning	No	
can tape worm cause malnutrition for my child and how do i know its tapeworm	No	

which crops are green most successfully in organic farming	No	
which crops are grown most successfully in organic farming	No	
after abortion nausea and sore breast	Yes	Same
severe acute abdominal pain	Yes	Same
signs that labor has started	No	
stomach ulcers that causes vomiting	Yes	Same
what are bad signs after giving birth with operation	No	
what confirms that abortion was successful	No	
what happens when ovulation stops	No	
which tablet help get abortion	No	
i am not worth anything	No	
i don't feel confident	No	
light blood clots after an abortion	Yes	Same
open soars on the out side of child's vagina	No	
open sores on the out side of child vagina	No	
procedure for surgical abortion	Yes	Same
symptoms of abortion after on month	No	
what are the name of abortion pills	Yes	Same
what is family planning	Yes	Same
when is it safe for a woman who had previously had a cesarean section	No	
when is it safe for a women who had previously had a cesarean section	No	
when is it safe for a women who had previously had a cesarean section to have a baby	No	
when is the right time for someone who already give birth by operation to get fall pregnant again	No	
when is the right time to fall pregnant again if you are a cesarean person	No	
when is the right time to fall pregnant again if you are a cesarean poison	No	
when is the right time to fall pregnant again if you are cesarean person	No	
check if pregnant	No	
check yourself to see if you are pregnant	No	
hard breast and pain after abortion	Yes	Same
how should the bleeding be after an abortion	No	
how to give injection in vein	Yes	Same

how to give intravenous injection	Yes	Same
how to terminate pregnancy	Yes	Same
how to treat malaria	Yes	Same
bleeding in urine	Yes	Same
can i conceive again after an abortion	Yes	Same
my breast hurts and i have difficulty in urinating	No	
symptoms of aids	Yes	Better
what can cause a woman to menstruate for two months during her pregnancy	Yes	Same
cancer in stomach	Yes	Worse
can you die from an infection after abortion	Yes	Same
what is endometriosis	Yes	Better
what are hemorrhoids	Yes	Worse
can I take valium with penicillin	No	
How to lower a fever	No	
medicine for malaria fever	Yes	Worse
abdominal pain in women	Yes	Worse
alcoholism	Yes	Worse
Down's syndrome	No	
causes of obesity	Yes	Same
ampicillin for men	Yes	Same
what are medicines for gonorrhea	Yes	Same
medicine to take after abortion	No	
How to take temperature	No	
flu symptoms	Yes	Same
best medicine for cough	Yes	Same
bleeding after birth	Yes	Same

Appendix C

Sample ActSpec

```

{
  'descriptorType': 'symptomDescriptor',
  'disease': {
    'objectDescriptor': {
      'descriptorType': 'diseaseDescriptor',
      'patient': {
        'objectDescriptor': {
          'age': 'ageGroup',
          'descriptorType': 'patientDescriptor',
          'gender': 'genderValues',
          'type': 'person'
        }
      },
      'type': 'diseaseType'
    }
  },
  'duration': {
    'objectDescriptor': {
      'descriptorType': 'objectDescriptor',
      'gender': 'genderValues',
      'givenness': 'givennessValues',
      'number': 'plural',
      'quantity': {
        'amount': {
          'number': 'singular',
          'schema': 'QuantitySchema',
          'template': 'QuantitySchema',
          'value': 4.0
        },
        'property': 'time',
        'schema': 'Quantity',
        'template': 'Quantity',
        'units': 'week'
      },
      'type': 'week'
    }
  },
  'gender': 'genderValues',
  'givenness': 'givennessValues',
  'location': {
    'objectDescriptor': {
      'descriptorType': 'objectDescriptor',

```

```

    'type': 'bodyPart2'
  }
},
'number': 'singular',
'original_query': 'bleeding for four weeks after an abortion',
'patient': {
  'objectDescriptor': {
    'age': 'ageGroup',
    'descriptorType': 'patientDescriptor',
    'gender': 'genderValues',
    'type': 'person'
  }
},
'sid': '#6b8adb3e-9ee3-4405-b5f9-e27b33c4a2d3',
'trigger': {
  'objectDescriptor': {
    'condition': {
      'objectDescriptor': {
        'descriptorType': 'conditionDescriptor',
        'patient': {
          'objectDescriptor': {
            'descriptorType': 'patientDescriptor',
            'gender': 'female',
            'type': 'person'
          }
        }
      },
      'type': 'pregnancy'
    }
  },
  'descriptorType': 'treatmentDescriptor',
  'drug': {
    'objectDescriptor': {
      'condition': {
        'objectDescriptor': {
          'descriptorType': 'conditionDescriptor',
          'patient': {
            'objectDescriptor': {
              'descriptorType': 'patientDescriptor',
              'gender': 'female',
              'type': 'person'
            }
          }
        }
      },
      'type': 'pregnancy'
    }
  },
  'descriptorType': 'treatmentDescriptor',
  'patient': {
    'objectDescriptor': {
      'descriptorType': 'patientDescriptor',
      'gender': 'female',
      'type': 'person'
    }
  },
  'type': 'drugType'
}
},
'gender': 'genderValues',
'givenness': 'typeIdentifiable',
'location': {
  'objectDescriptor': {

```

```
        'descriptorType': 'objectDescriptor',
        'type': 'bodyPart2'
    }
},
'number': 'singular',
'patient': {
    'objectDescriptor': {
        'descriptorType': 'patientDescriptor',
        'gender': 'female',
        'type': 'person'
    }
},
'type': 'abortion'
}
},
'type': 'bleeding'
}
```

References

- [1] R H. Baayen, R Piepenbrock, and L Gulikers. *CELEX2 LDC96L14*. Philadelphia, 1995.
- [2] Cory Barr, Rosie Jones, and Moira Regelson. “The Linguistic Structure of English Web-Search Queries”. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing - EMNLP '08* October (2008), p. 1021. DOI: 10.3115/1613715.1613848. URL: <http://dl.acm.org/citation.cfm?id=1613848%7B%5C%7D5Cnhttp://portal.acm.org/citation.cfm?doid=1613715.1613848>.
- [3] Benjamin K Bergen and Nancy Chang. “Embodied Construction Grammar in Simulation-Based Language Understanding”. In: *Construction grammars: Cognitive grounding and theoretical extensions* (2005), pp. 147–190. ISSN: 15481484. DOI: 10.1075/cal.3.08ber.
- [4] John Edward Bryant. “Best-Fit Constructional Analysis”. PhD thesis. University of California at Berkeley, 2008.
- [5] Huanhuan Cao et al. “Context-aware query classification”. In: *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '09* (2009), p. 3. ISSN: 00100277. DOI: 10.1145/1571941.1571945. URL: <http://portal.acm.org/citation.cfm?doid=1571941.1571945>.
- [6] Nancy Chang. “Constructing grammar: A computational model of the emergence of early constructions”. PhD thesis. University of California, Berkeley, 2008.
- [7] W Bruce Croft et al. “Query representation and understanding workshop”. In: *ACM SIGIR Forum* 44.2 (Jan. 2011), p. 48. ISSN: 01635840. DOI: 10.1145/1924475.1924485. URL: <http://portal.acm.org/citation.cfm?doid=1924475.1924485>.
- [8] Steve Doubleday, Sean Trott, and Jerome Feldman. “Processing Natural Language About Ongoing Actions”. In: (2016), pp. 171–177. arXiv: 1607.06875. URL: <http://arxiv.org/abs/1607.06875>.
- [9] Manfred Eppe, Sean Trott, and Jerome Feldman. “Exploiting deep semantics and compositionality of natural language for human-robot-interaction”. In: *IEEE International Conference on Intelligent Robots and Systems 2016-November* (2016), pp. 731–738. ISSN: 21530866. DOI: 10.1109/IRoS.2016.7759133. arXiv: 1604.06721.

- [10] Manfred Eppe et al. “Application-Independent and Integration-Friendly Natural Language Understanding”. In: *EPiC Series in Computing* 41. GCAI 2016. 2nd Global Conference on Artificial Intelligence (2016), pp. 340–352.
- [11] Jerome Feldman. “Embodied language, best-fit analysis, and formal compositionality”. In: *Physics of Life Reviews* 7 (2010). URL: <http://dx.doi.org/10.1016/j.plrev.2010.06.006>.
- [12] Jerome Feldman. *From molecule to metaphor: a neural theory of language*. MIT Press, 2006.
- [13] Jerome Feldman, John Edward Bryant, and E Dodge. “Embodied Construction Grammar”. In: *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2009, pp. 38–111. DOI: 10.1093/oxfordhb/9780199544004.013.0006.
- [14] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
- [15] Alejandro Figueroa and Guenter Neumann. “Exploiting User Search Sessions for the Semantic Categorization of Question-like Informational Search Queries”. In: *Proceedings of the Sixth International Joint Conference on Natural Language Processing* (2013), pp. 902–906. URL: <http://aclweb.org/anthology/I13-1115>.
- [16] Luca Gilardi and Jerome Feldman. “A Brief Introduction to the ECG Workbench and a First English Grammar”. URL: <ftp://ftp.icsi.berkeley.edu/pub/ntl/wb/ECG-HOWTO.pdf>.
- [17] Rafael Glater, Rodrygo L.T. Santos, and Nivio Ziviani. “Intent-Aware Semantic Query Annotation”. In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '17*. New York, New York, USA: ACM Press, 2017, pp. 485–494. ISBN: 9781450350228. DOI: 10.1145/3077136.3080825. URL: <http://dl.acm.org/citation.cfm?doid=3077136.3080825>.
- [18] Ivan Habernal and Miloslav Konopík. “SWSNL: Semantic web search using natural language”. In: *Expert Systems with Applications* 40.9 (2013), pp. 3649–3664. ISSN: 09574174. DOI: 10.1016/j.eswa.2012.12.070. URL: <http://dx.doi.org/10.1016/j.eswa.2012.12.070>.
- [19] Jian Hu et al. “Understanding user’s query intent with wikipedia”. In: *Proceedings of the 18th international conference on World wide web - WWW '09* (2009), p. 471. DOI: 10.1145/1526709.1526773. URL: <http://portal.acm.org/citation.cfm?doid=1526709.1526773>.
- [20] Rosie Jones et al. “Generating query substitutions”. In: *Proceedings of the 15th international conference on World Wide Web - WWW '06*. New York, New York, USA: ACM Press, 2006, p. 387. ISBN: 1595933239. DOI: 10.1145/1135777.1135835. URL: <http://portal.acm.org/citation.cfm?doid=1135777.1135835>.
- [21] Huda Khayrallah, Sean Trott, and Jerome Feldman. “Natural Language For Human Robot Interaction”. In: *International Conference on Human-Robot Interaction (HRI)* (2015).

- [22] Xiao Li. “Understanding the Semantic Structure of Noun Phrase Queries”. In: *the 48th Annual Meeting of the Association for Computational Linguistics* July (2010), pp. 1337–1345.
- [23] Xiao Li, Ye-Yi Wang, and Alex Acero. “Extracting structured information from user queries with semi-supervised conditional random fields”. In: *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '09* (2009), p. 572. DOI: 10.1145/1571941.1572039. URL: <http://portal.acm.org/citation.cfm?doid=1571941.1572039>.
- [24] Jingjing (MIT) Liu et al. “QUERY UNDERSTANDING ENHANCED BY HIERARCHICAL PARSING STRUCTURES”. In: *ASRU* (2013), pp. 72–77.
- [25] Tie-Yan Liu. “Learning to Rank for Information Retrieval”. In: *Foundations and Trends® in Information Retrieval* 3.3 (2007), pp. 225–331. ISSN: 1554-0669. DOI: 10.1561/1500000016. arXiv: arXiv:1208.5535v1. URL: <http://www.nowpublishers.com/article/Details/INR-016>.
- [26] Craig Macdonald, Nicola Tonellotto, and Iadh Ounis. “Efficient & Effective Selective Query Rewriting with Efficiency Predictions”. In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '17* (2017), pp. 495–504. DOI: 10.1145/3077136.3080827. URL: <http://dl.acm.org/citation.cfm?doid=3077136.3080827>.
- [27] Mehdi Manshadi and Xiao Li. “Semantic tagging of web search queries”. In: *ACL '09 Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2 - Volume 2* August (2009), pp. 861–869. DOI: 10.3115/1690219.1690267. URL: <http://dl.acm.org/citation.cfm?id=1690267>.
- [28] Eva H. Mok. “Contextual Bootstrapping for Grammar Learning”. PhD thesis. EECS Department, University of California, Berkeley, Jan. 2009. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-12.html>.
- [29] Eva H Mok and John Bryant. “A Best-Fit Approach to Productive Omission of Arguments”. In: *In proceedings of the Berkeley Linguistics Society* (2006).
- [30] Srini Narayanan and Matt Gedigian. “The Hesperian Digital Commons: A Multilingual Primary Health Resource”. In: *Proceedings of the CCC Workshop on Computer Science and Global Development*. 2009, pp. 63–64. URL: http://archive2.cra.org/ccc/files/docs/CCC%7B%5C_%7DGD%7B%5C_%7DProceedings.pdf.
- [31] Vivek Raghuram and Sean Trott. *ECG Homepage*. URL: https://github.com/icsi-berkeley/ecg%7B%5C_%7Dhomepage/wiki (visited on 02/05/2018).
- [32] V. Raghuram et al. “Semantically-driven coreference resolution with embodied construction grammar”. In: *AAAI Spring Symposium - Technical Report SS-17-01 - Feldman 2010* (2017).

- [33] Sean Trott, Manfred Eppe, and Jerome Feldman. “Recognizing Intention from Natural Language : Clarification Dialog and Construction Grammar”. In: *Workshop on Communicating Intentions in Human-Robot Interaction* (2016).
- [34] Sean Trott et al. “Natural Language Understanding and Communication for Multi-Agent Systems”. In: *AAAI Fall Symposium* (2015), pp. 137–141.
- [35] Christina Unger et al. “Template-based question answering over RDF data”. In: *Proceedings of the 21st international conference on World Wide Web - WWW '12* (2012), p. 639. ISSN: 10450823. DOI: 10.1145/2187836.2187923. arXiv: 1603.07044. URL: <http://dl.acm.org/citation.cfm?doid=2187836.2187923>.
- [36] Zhongyuan Wang et al. “Query understanding through knowledge-based conceptualization”. In: *IJCAI International Joint Conference on Artificial Intelligence 2015-January*. Ijcai (2015), pp. 3264–3270. ISSN: 10450823.
- [37] Mohamed Yahya et al. “Natural language questions for the web of data”. In: *EMNLP -CoNLL '12 July* (2012), pp. 379–390. URL: <http://dl.acm.org/citation.cfm?id=2390948.2390995>.
- [38] Shi Zhao and Yan Zhang. “Tailor knowledge graph for query understanding: linking intent topics by propagation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2014, pp. 1070–1080. DOI: 10.3115/v1/D14-1114. URL: <http://aclweb.org/anthology/D14-1114>.
- [39] Ingrid Zukerman and Bhavani Raskutti. “Lexical Query Paraphrasing for Document Retrieval”. In: *Proceedings of the 19th International Conference on Computational Linguistics - Volume 1. COLING '02*. Taipei, Taiwan: Association for Computational Linguistics, 2002, pp. 1–7. DOI: 10.3115/1072228.1072389. URL: <https://doi.org/10.3115/1072228.1072389>.