KTH Computer Science
and Communication

Semantic parsing of Swedish laws

Using RDF to parse and format legal documents in accordance with applicable standards to contribute to a more semantic web

MIKAEL FALGARD

Master's Thesis at NADA
Supervisor: Sten Andersson
Examiner: Anders Lansner

TRITA xxx yyyy-nn

Abstract

This project has been carried out as a thesis project at the School of Computing at the Royal Institute of Technology in Stockholm (KTH) on behalf of Notisum AB. The project aims to investigate whether Notisum's current process to parse and format downloaded legal documents can be improved by a more modern process. Part of the process is to mark up the data in the legal documents according to Rättsinformationsprojektets proposed standard. This will contribute to a more semantic web where all data is meaningful to both humans and computers. The Resource Description Framework (RDF), is a framework that is used to describe or model information from web sources, have been used to produce a prototype to parse and format documents (mainly laws) from Swedish Constitution (SFS) in the programming language Python. This work also included research into the phenomenon of "The Semantic Web", what it prejudge and how to proceed in order to meet its requirements. Finally, it proposes recommendations on how Notisum can proceed with the use of the developed method for parsing legislations in a meaningful way and also some improvements that could be implemented.

Referat

Utnyttja RDF för att tolka och formatera juridiska dokument i enlighet med tillämpliga standarder för att bidra till en mer semantisk webb

Detta projekt har utförts som ett examensarbete på skolan för datavetenskap och kommunikation på Kungliga Tekniska Högskolan i Stockholm (KTH) på uppdrag av Notisum AB. Projektets syfte är att undersöka huruvida Notisums nuvarande process för att formatera nerladdade juridiska dokument kan förbättras genom en modernare process. En del av processen är att märka upp datat i dokumenten enligt Rättsinformationsprojektets föreslagna standard. Detta ska bidra till en mer semantisk webb där all data är meningsfull för både människor och datorer. Resource Description Framework (RDF), ett ramverk som används för att beskriva eller modellera information från webbkällor har använts för att ta fram en prototyp för att formattera dokument (lagar) från Svensk Författnigsamling i språket Python. I arbetet ingick även forskning kring fenomenet "Den Semantiska Webben", vad den innbär och hur man kan gå till väga för att uppfylla dess krav. Slutligen föreslås rekommendationer för hur Notisum kan gå vidare med användandet av den framtagna metoden för att märka upp lagar på ett meningsfullt sätt och även vissa förbättringar som skulle kunna genomföras.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Legal information online

According to *Rättsinformationsförordningen* (1999:175) basic legal information has to be provided both the public administration, and to private individuals in electronic form. For example the collection of statutory law, the *Svensk Författningssamling* (SFS) is published in the Government Offices legal databases. These databases are old and hard to navigate since they only consist of plain text documents.

1.2 Uniform standards

One of the goals with *Rättsinformationsförordningen* is that all legal information should be coherent, searchable from a single location and have a uniform presentation. Today's decentralized system where authorities are responsible for their own documents, leads to a couple of problems. One of them being that the documents don't follow the same format standards.

In 2006 a project called *Rättsinformationsprojektet* was commenced to develop the legal information system further. A first step was to assure that the document that is going to be published contains the required meta data for that type of document. The next step is to assure that the information is delivered in a correct format.[1] For example, date data must be unambiguously expressed in machine-readable form.

1.3 Rättsnätet

Rättsnätet is a free web service on `www.notisum.se`[2] that provides legal information. Rättsnätet's content consists of information gathered from authority databases and is processed in several steps from pure text information into a rich XML file.

[1] Guidelines for publisher, developers etc are found on `http://www.lagrummet.se`

[2] Notisum AB is the company that I am writing this thesis at.

Notisum also provides a premium service that includes additional information and services.

1.4 Lagen.nu

A similar website to Notisum is lagen.nu, they also provide all Swedish laws as a free web service. A difference is that lagen.nu follows standards regarding the Semantic Web[3] in a better way than Notisum's Rättsnätet does. Lagen.nu is created by an individual with voluntary interest in law, the Semantic Web and is also involved in *rättsinformationsprojektet*.

1.5 Problem

Today Rättsnätet use processing steps that downloads, analyzes and transforms authority information from plain text to structured and marked up XML information. This process is a collection of programs written in Delphi Pascal and C#. These programs, especially those that have been written in Delphi Pascal, are old and difficult to maintain. The conversion parts are constructed using a traditional technique similar to that used to write compilers.

Instead of using compiler technology lagen.nu use regular expressions to parse legal documents. The conversion programs lagen.nu use are written in object-oriented Python, which is more suitable for the task than Delphi Pascal that Notisum use today. Additionally lagen.nu is based on a document model from the Semantic Web, with the Resource Description Framework, RDF, as a base. Because of this the code base in lagen.nu has greater flexibility and is more about standards than Notisum.

The problem that this thesis focus on is how to modernize and replace Rättsnätet's current platform to a more modern process. A process where the result should be based on the Semantic Web and follow standards proposed by Rättsinformationsprojektet. Some aspects of the problem to keep in mind are:

- The amount of documents, since there is over 200.000 legal documents, the program requires a fast process.
- Legal source texts contains references that should be interpreted differently depending on the context, and to be marked up automatically with a certain margin of error.
- Cross-references between the 200,000+ documents require theoretically, an array of 40 billion nodes.

[3]The Semantic Web is a collaborative movement that promotes common data formats on the World Wide Web.

- The output from the process needs to be structurally similar to Notisums current process output so the post process (generating html) can be used with as little tweaking as possible.

1.6 Limitations

The implementation part of the thesis will be limited to the parsing step of the process. It will not cover the downloading or final transformation to html that is ready to be published.
To make this project feasible it is limited to handling only SFS data. Other types of data sources that a complete process would need to handle are for example:

- Propositions
- Supreme Court summaries
- Decisions of the Courts of Appeal, and the Labour Court
- European legal regulations and directives

The goal is of course to create a generic and loosely coupled code base, so that it is easy to implement these modules in a later stage.

1.7 Abbreviations and Vocabulary

To be able to understand this report, the reader needs to understand the following abbreviations. Given the complexity of a legal document[4] I will also present a vocabulary of how words are defined in the thesis.

SFS

The Swedish Code of Statutes “Svensk författningssamling” (SFS) is the official publication of all Swedish laws enacted by the Riksdag and ordinances issued by the Government[5]. Every law and ordinance has an SFS number, it consists of a four digit year, a colon, and then an incrementing number by year. For instance the Ordinance on tattooing dyes have the SFS number 2012:503[6].

SFSR

The Statute Register (SFSR) includes registry information of Swedish Code of Statutes (SFS), amendments, references to preparatory work, etc.

[4]For example "a paragraph" in a Swedish legal document does not correspond to a regular paragraph in the English language.

[5]SFS - `http://en.wikipedia.org/wiki/Swedish_Code_of_Statutes`

[6]PDF copy of 2012:503 - `http://notisum.se/rnp/sls/sfs/20120503.pdf`

SFST

Statutes in full text (SFST) includes Swedish Code of Statutes (SFS) in full text, ie all applicable laws and regulations.

URI

Short for Unified Resource Identifier. Uniquely identifies resources in a collection[7], for example an hypertext transfer protocol url specifies a webpage on the internet.

XML

Extensible Markup Language (XML) defines a set of rules for encoding documents in a format that is readable for both humans and machines[8].

HTML

HyperText Markup Language (HTML) is the main language for displaying web pages and other information that can be displayed in a web browser[9].

XHTML

Extensible HyperText Markup Language (XHTML) is HTML written as XML. Documents written in XHTML needs to be well formed, elements must be properly nested and they always have to be closed. Because of this, XHTML can be parsed using standard XML parsers, unlike HTML, which requires a lenient HTML-specific parser. XHTML files are saved with the extension .xht.

RDF

Resource Description Framwork (RDF) is a general method to describe or model information that is implemented in web resources[10].

UTF-8

A encoding that can represent every character in the Unicode character set. UTF-8 has become the dominant character encoding for the World-Wide Web.

ISO-8859-1

A character encoding that is generally intended for "Western European" languages, it encodes what it refers to as "Latin alphabet no 1" consisting of 191 characters from the Latin script.

[7]URI - `http://www.w3.org/TR/uri-clarification/`
[8]XML - `http://en.wikipedia.org/wiki/XML`
[9]HTML - `http://en.wikipedia.org/wiki/Html`
[10]More detailed information about RDF in the 'Background' section

N3

Notation3, or N3 is a shorthand non-XML serialization of RDF models, designed with human-readability in mind which makes it more compact and readable than XML RDF notation.

Dublin Core

The Dublin Core metadata terms are a set of vocabulary terms which can be used for simple resource description, or combining metadata vocabularies of different metadata standards in Semantic web implementations.

The Semantic Web

The word semantic stands for "the meaning of". The semantic of something is the meaning of something. The Semantic Web = a Web with a meaning.

Chapter 2

Background

2.1 The Semantic Web - A Web with a meaning

The Semantic Web is a collaborative movement led by W3C[1] that promotes common data formats on the world wide web by encouraging semantic content in web pages. W3C's "Semantic Web Vision" is a future where:

- Web information has exact meaning
- Web information can be understood and processed by computers
- Computers can integrate information from the web

Natural languages have ambiguous meanings and even a human reader may in some cases have problems understanding the correct meaning of a text. If someone ask me "Do you know Zlatan?" they may refer to if I know him as a friend or if Iknow who he is. It is the same for computers when they are trying to understand the meaning of a text. By adding labels to the text we can make it easier to interpret and thus creating a semantic web, formed so that software can collect and analyze data. The aim is that system can present the answer to a query instead of a page where the answer can be found, and answer queries where the answer is spread over several documents. The semantic web is a web of data, where the data can be stored in documents in various ways. Using RDF is a way to structure data, so that it can be linked to other data with information and realations to objects.

2.1.1 RDF

Resource Description Framwork (RDF) is a general method to describe or model information that is implemented in web resources. It is based upon the idea of making statements about resources in the form of triples of subject-predicate-object. The subject denotes the resource (it can be anything that can have a URI), the predicate denotes traits or aspects of the resource and also expresses a relationship between

[1]The international standards body, the World Wide Web Consortium (W3C)

the subject and the object. For example: "The sky" (subject) "has the color" (predicate) "blue" (object) could be an RDF triple.

Let's take a look at an example[2] of an RDF object describing two CDs.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cd="http://www.recshop.fake/cd#"/>

  <rdf:Description
  rdf:about="http://www.recshop.fake/cd/Empire Burlesque"/>
    <cd:artist>Bob Dylan</cd:artist>
    <cd:country>USA</cd:country>
    <cd:company>Columbia</cd:company>
    <cd:price>10.90</cd:price>
    <cd:year>1985</cd:year>
  </rdf:Description>

  <rdf:Description
  rdf:about="http://www.recshop.fake/cd/Hide your heart"/>
    <cd:artist>Bonnie Tyler</cd:artist>
    <cd:country>UK</cd:country>
    <cd:company>CBS Records</cd:company>
    <cd:price>9.90</cd:price>
    <cd:year>1988</cd:year>
  </rdf:Description>
</rdf:RDF>
```

The two lines at the top, **xmlns:rdf** and **xmlns:cd** define from which namespace elements with the rdf and cd prefix are from. The **rdf:Description** element contains the description of the resource identified by the **rdf:about** attribute. Finally the elements **cd:artist, cd:country, cd:company** etc. are properties of the resource.

2.2 Linked Data - Connect data across the Web

The Semantic Web isn't just about putting data on the web. It is about creating connections through links, so that when you have some of it, you can find other, related, data.

The term Linked data describes a method of publishing structured data so that

[2]Full example can be viewed at `http://www.w3schools.com/rdf/rdf_example.asp`

it can be interlinked and become more useful. This is done by following a data model which is based on resources and that describe their properties, including relationships to other resources. By using standardized names for resources and property types, data sets from many different providers integrated. By expressing the relationships between resources in different data sets, a user may discover additional relevant data.
The foundation of linked data are the technologies HTTP and URIs. Today we use them (and many other services) to present human readers with web pages, linked data extends this to share information in a way that can be read, connected and queried automatically by computers. This is as a concept is not new, and have been used in other similar situations such as database network models and headings in library catalogs.

Tim Berners-Lee,[3] the father of the World Wide Web coined the term *"Linked Data"* and outlined four principles of linked data:

1. Use URIs to identify things.

2. Use HTTP URIs so that these things can be referred to and looked up ("dereferenced") by people and user agents.

3. Provide useful information about the thing when its URI is dereferenced, using standard formats such as RDF/XML.

4. Include links to other, related URIs in the exposed data to improve discovery of other related information on the Web.

A community project called *Linking Open Data*[4], organized by W3C aims to extend the Web by publishing various open datasets as RDF on the Web by creating RDF links between data items from different data sources. In October 2007, the datasets consisted of over two billion[5] RDF triples, which were interlinked by over two million RDF links. By September 2011 this had grown to 31 billion RDF triples, interlinked by around 504 million RDF links.

The image below shows datasets that have been published in Linked Data format, by contributors to the Linking Open Data community project and other individuals and organizations. The purpose of this image is merely to depict the extent of such a project, not to present the details of the data sets.[6]

[3]TODO: add src on Tim
[4]WC3 Semantic Web Education and Outreach group's - Linking Open Data community project
[5]TODO: Add src
[6]See `http://lod-cloud.net/` for a more detailed image with links to all the contributors.

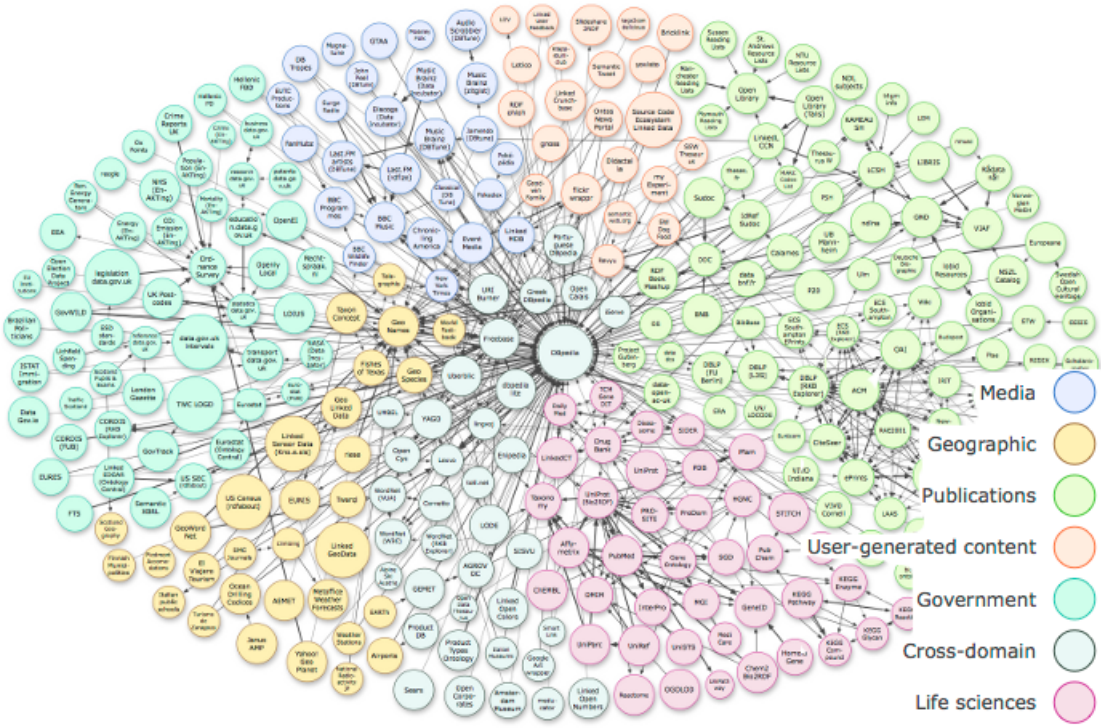

Figure 2.1: Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch.

2.3 Swedish law

The main legislative powers are the parliament (Riksdagen) and the government (Regeringen), these institutions can adopt statutes which are published in the main official collection of statutory law, the *Svensk Författningssamling* (SFS). The statutes enacted by the parliament are referred to as laws, and the statutes enacted by the government as ordinances.

When a statute is changed, this is done by adopting a new statute (the change statute) that states what sections of the old statute (the base statute) are to be changed, and how. There is not a new merged version published of the statute in SFS, only the base statute and change statute(s).

Two recurring definitions in this paper are legal sources and legal source documents. A legal source is a type of legal information such as a constitution in SFS or verdicts from one of the Swedish courts. A legal source document is a specific document for a certain legal source, such as *Personuppgiftslagen* (SFS 1998:204).

2.3.1 Law structure

Today there's 3000 constitutions but there's no well defined structure that describes how a law should look like. One thing that is common is that each law has an SFS number, including legislations amending already existing laws. The SFS number consists of a four digit year, a colon and a serial number assigned in chronological order of the date of issue.

2.3.2 Rättsinformationssystemet

Rättsinformationssystemet is the state's system to make legal information such as laws, legislative histories, court cases, etc. - available to the public via the Internet. This information is produced by a number of different authorities. Today, each agency is responsible for publishing "their" information via their own website. It is all tied together by the website `http://www.lagrummet.se` which is referring to each agency's respective legal information page.
This decentralized approach has its advantages. It gives the authorities a lot of freedom to design their information in a way that fits them. A downside is that they use this freedom to invent their own standards. *Rättsinformationsförordningen*[7] is not very detailed on how the information should be presented, only that it should be published.

An additional purpose with the system is the ability for the private sector to re-use government information to create added value services[8]. Legal information is of great value, that could be even better utilized if it was easier to re-use the information.

[7] Rättsinformationssystemet is regulated in this ordinance (1999:175)
[8] Services like Rättsnätet from Notisum and lagen.nu

Part II

Results

Chapter 3

Method

The work with this thesis can be divided into two parts, the first being reading litterature regarding the Semantic Web, Linked Data and related subjects. The litterature also included reading many articels, blog post and watching podcasts,[1] that were great to get a better understaning of the concept and importance of a more semantic web.

The second part was to create a prototyp. Before I started to work on that I looked at the existing system, and tried to get some understanding of what steps the process went through. At this point a I got a lot of guidence from Magnus at Notisum, the author of the existing code. He explained the necessary steps and why they are important for the end result. However I didn't spend to much time on the old code, since I needed to create something different (and take factors such as semantics into mind).
With no background or knowledge in law it can be quite hard to understand the structure of laws and references between them. That is something I picked up as work went along with the prototype, many times I had to stop and try to figure out what I tried to do and why.

3.1 Methodology

I devloped the prototype in several iterations, a simplfied version of agile.[2] The first step was to get the program to read downloaded documents and save them as a new file. After that was in place, I always had a running prototype that would go through all files in a specified folder and apply transformations to them before saving them with a new exstension. This way I could add new functionality and try it out right away. I tried to put general functions and helpers seperate from SFS-specific logic so that it is possible to reuse a lot of the code when other legal sources needs to be implemented.

[1] TODO, links to references, podcasts etc.
[2] Agile ... TODO

3.2 Existing system

Below follows a short description of the steps the process goes through today. The steps in blue bubbles (first and last) are written in C# and could with minor modifications be used together with the prototype. The two steps in pink bubbles (second step which is split in two different parts and the third step) are programs written in Pascal and will be replaced by the prototype. Then there's a green bubble (fourth step) which also with some modifications, could be reused with the output from the prototype.

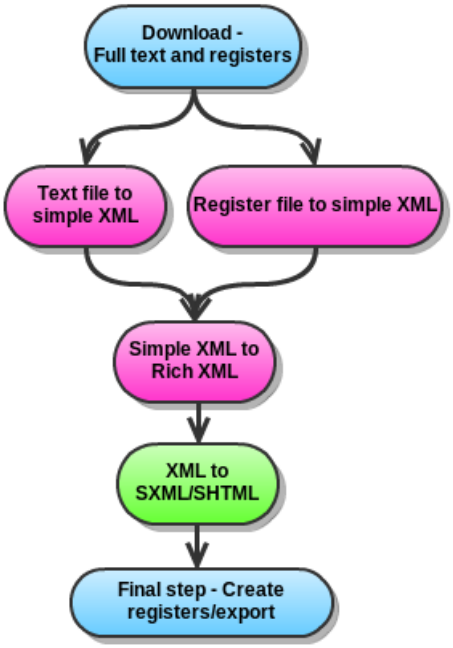

Figure 3.1: Chart describing the existing process

For the sake of simplicity lets say we are only handling SFS laws[3]. First we have a module that handles the actual downloading of the legal source files, (full text and register files[4] for each law) this part is written in C# and could be reused[5] for the prototype. Next we have two programs that for each law converts the downloaded files to very simple XML files, one for the full text and one for the registerfile. The next step is also an XML transformation, it takes the simple XML files and turns them into more comprehensive XML files. Now it's time to create files that can be viewed in a browser, this is done in several steps where data structures containing information regarding the laws are created with information like paragraph references and links. In the end a SHTML[6] file that is ready to be

[3]In reality the program handles all kind of legal source documents.
[4]TODO, explain Registerfile
[5]This part just have to download the files, then the prototype can do it's magic on those files.
[6]TODO, explain SHTML

published with this information is created. The final step is to export these files (and other legal source documents) to databases and update the website to display the newly created or updated files.

3.3 Prototype

The main differences between the old system and the prototype are:

- Language, the prototype is written in Python[7] which is a object oriented[8] languge that is well suited for tasks like this one. It is easy to divide the code into separate modules that handle different tasks. Furthermore it is convinient to work with (apply transformations etc.) objects such as whole documents, paragraphs.
- Follows standards proposed by Rättsinformationsprojektet for legal documents.
- The generated files will be marked up with RDF to contribute to a more semantic web.

3.3.1 Delimitations

The prototype does not have to handle the downloading of the legal source documents, these can be presumed exist on the computer running the prototype.
The aim for the prototype is to deliver XML files, the final step where these files are transformed into HTML files is simple and needs to be controlled by Notisum. However this step is done by the prototype but might not be the way Notisum will do it, if they use the prototype.

[7] TODO, something about python
[8] something about OO

Chapter 4

Implementation

In this chapter I will discuss how the implementation was done and describe some implementation ideas and some parts of the prototype.

4.1 Interface

In order to interact with the program, read status and error information some sort of user interface is needed. Most common is a graphic user interface[1] that for example uses icons and menus. However, this falls outside the scope of this project, which focuses on the core of the program rather than the user experience. Instead, a rudimentary command line interface[2] was implemented, where the user can specify arguments at the start of the program (which is done via a terminal). During the parse and transformation process status updates will be presented in the terminal, for example if something fails the user will be notified. Also when the program is done the user will receive information about what was processed and the outcome.

The main file is called Controller.py and to run the program from a terminal an argument specifying what the program should do is needed. The intresting arguments for this thesis are 'ParseAll' and 'GenerateAll' other possible arguments are 'DownloadAll' and possible candidates to implement would be only 'Parse' or 'Generate' followed by a specific source, for example 'Parse 1997:240'. Also one can specify what types of legal sources that should be run, (will be relevant when more sources than SFS are implemented.) default behavior is to run for all types. There's also two available flags to use, '-d' for debug mode and '-h' or '–help' for help instructions.

[1] A graphic user interface is a human-computer interface i.e., a way for humans to interact with computers.

[2] Command line interface (CLI), an interface which use only text and are accessed solely by a keyboard. For example via an terminal in Linux.

```
kungen@dell-desktop:~$ python Controller.py --help
Usage: pyhton Controller.py [-d | -h] [arg]
Available flags are: -d (debug), -h--help (help)
kungen@dell-desktop:~$ python Controller.py
No arguments given
Valid arguments are: DownloadAll, GenerateAll, ParseAll
kungen@dell-desktop:~$ python Controller.py ParseAll
```

4.2 Main modules

The main modules consists of the 'Controller' and 'Source' that includes the logic to run the program. Then there's the different legal type modules that contains their specific logic.

4.2.1 Controller.py

The controller handles validation of user input (arguments and flags) if everything is correct it fetches all available legal source types (in this case just SFS) and then run the command given as an argument, ex. 'ParseAll' for each type.

4.2.2 Source.py

This module is a blueprint of how modules inheriting from it needs to look like. It contains the base classes for Controller and Parser, these classes contains functions that child classes need to implement and functions that can be overridden. Example of general functions are: checking if a file is up-to-date, trimming filenames or returning the XML or XHTML name for a certain file.
However most of the code will be specific for each legal source type and be implemented in the child modules. To support a new type of legal source for example 'EG Court cases'[3], create an 'EG' module that inherits Source.Controller and Source.Parser, that is all that is needed for it to function with the rest of the program.

4.2.3 SFS.py

This is the module specific for SFS documents, basically it takes a list of all the downloaded files[4] and parse the files one by one.
The largest and most complex class is 'SFSParser' that parses a document and creates objects for each part of the document. All objects are defined as classes[5] here, for example 'Rubrik', 'Stycke', 'Paragraf' etc. One of the parser's duties is

[3] European.. TODO

[4] The path to the base directory is specified in a config file, then it appends a path like '/sfs/dl/sfsr/1997' for example for all SFS full text documents from 1997

[5] These classes inherit from base classes in DataObjects.py.

to figure out what type of object a certain part of the text should be represented as. This is achieved by first guessing what it should be based on for example the previous section, then running functions to verify or reject that type.

4.3 Helper modules

The helper modules are used to simplify (mostly) for the Parser by defining native python types, providing different types of text readers etc.

4.3.1 DataObjects.py

Makes it possible to build an object model for each legal source document, by creating simple data objects. These objects are base data types that inherit from native python types such as unicode, list, dictionary. The data types have added support for other properties that can be set when instantiated.

4.3.2 Reference.py

Parses plaintext and finds references to other legal source documents and returns a list with "Link-objects"[6]. Depending on with which properties it is initialized, it can find different types of references (Other laws, 'Rättsfall', 'EG-lagstiftning' etc..).

4.3.3 TextReader.py

A helper class to read text files in different ways, by line, paragraph etc.

4.3.4 Util.py

A few small help functions mostly related to checking and or renaming directories and files.

4.4 Unicode

Unicode is a standard for consistent encoding, representation and handling of text. It can be implemented by different character encodings[7].
Python's Unicode string type stores characters from the Unicode character set. Unicode characters don't have an encoding; each character is represented by a number which is called its code point. However we work with text files, which contains encoded text, not characters. Each character in the text is encoded as one or more bytes in the file. If you read a line of text from a file, you get bytes, not characters.

[6] A Link-object is a base data type that inherits from 'Unicode Structure' in DataObjects.py, basically a unicode string with a 'URI' property.

[7] Example of common character encodings are, UTF-8 and ISO-8859-1, see the vocabulary in chapter 1 for more info.

To solve this, we need to decode the text strings that we read from files, to do this, we can use Python's **decode()** method on the string, and pass it the name of the encoding like so:

```
encoding = "iso-8859-1"

raw = file.readline()
txt = raw.decode(encoding)
```

When we need to save a Unicode string to a file, we have to do the opposite conversion and encode it. The **encode()** method converts from Unicode to an encoded string.

```
out = txt.encode("utf-8")
```

There is a lot of files and text to handle when parsing laws, to avoid issue with different encodings and representations of characters we try to only work with Unicode in the code. Also some of the third party libraries only work with Unicode which is another reason to stick with that. The content is converted[8] to Unicode when it is loaded into the program. A Unicode string in Python looks like this *u'string'* instead of a just *'string'* and that is the format that is used to represent strings when needed.
The source files we work with are saved as ISO-8859-1 and the generated files we create will be saved in UTF-8.

4.5 3rd party libraries

There's a few third party libraries that are used in the program to perform certain tasks.

4.5.1 BeautifulSoup

BeautifulSoup is a really helpful library when you need to parse HTML documents (even with malformed markup, i.e. non closed tags). It creates a DOM[9] for parsed pages that can be used to extract data, for example you can ask it to extract "The first row in the second table on the page". This comes very handy and have been to a great assistance during my work.

[8] The conversion is done by the Util module.

[9] Document Model Object (DOM) is a convention for representing and interacting with objects in HTML, XHTML and XML documents.

4.5.2 SimpleParse

SimpleParse allows you to generate parsers directly from EBNF[10] grammar. This library has also been very helpful during my work since it allowed me to specify rules (for different types of documents, references, headlines etc.) and then create a parser with those capabilities.

4.5.3 RDFLib

RDFLib is a library for working with rdf that contains an RDF/XML parser and serializer. It also contains graph data structures to represent data. In my case it was for example used to read and represent data from an external file with URIs to all the SFS laws.

4.5.4 Genshi

Genshi is a library that provides components for parsing, generating, and processing HTML, XML or other textual content for output generation on the web. The main feature that is used in this project is a template language. It is used to convert the parsed and marked up files to XHTML files. A nice feature is that template instances are cached to avoid having to parse the same template file more than once.

4.6 Version control

GitHub[11] is used for web hosting and revision control, the code used for this report is open source and available to checkout from the link below. On the github page there's also an issue tracker with the current open issues, bugs and some enhancements.

Project: `https://github.com/Sup3rgnu/lawParse`
Checkout[12]: `https://github.com/Sup3rgnu/lawParse.git`

[10] Extended Backus-Naur Form is a formal way to describe the grammar of languages. It consists of rules which are restrictions governing how symbols can be comined.

[11] GitHub is a web-based hosting service for software development projects that use the Git revision control system. `http://github.com`

[12] For more info regarding how to checkout, see `http://www.kernel.org/pub/software/scm/git/docs/git-checkout.html`

Chapter 5

Analysis

This section will look at the different steps in the process flow, what kind of input we expect, how the parsing is done and finally what kind of output we get.

5.1 Process flow

Let's run the program and follow one SFS statute through the process flow to see what steps it goes through and how it changes. The statute chosen[1] for this excerise is the *Home guard regulation*[2] (1997:146). A transcript of the log created (when the 'debug' flag is turned on) during the parsing of this file can be found in the appendix, if the reader wants to follow the process steps.

5.1.1 Input files

The SFS statutes published in the Government Offices legal databases consists of two files for each statute, one *Statute register file* (SFSR) and one *Statute in full text* (SFST). The program expects the downloaded files to be saved in a specific tree structure to be able to read, create and save files.

[1] The statute was chosen randomly, it does not have any specific properties that makes it more suitable as an example.

[2] Hemvärnsförordningen `http://notisum.se/rnp/sls/sfs/20050819.pdf`

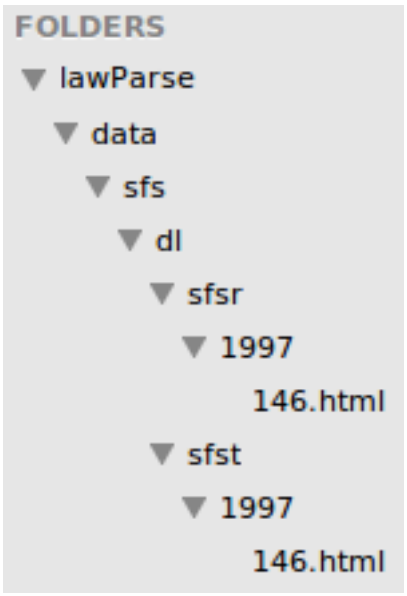

Figure 5.1: Tree structure for downloaded files

Under the root 'lawParse' there is a directory called 'data' which contains all different types of legal documents, here we have a 'sfs' directory where we store the downloaded, ('dl') parsed and eventually the generated files respectively. In all these directories we have another level for 'sfsr' and 'sfst' files, here the statutes are sorted according to the year they were created.

5.1.2 Parsing

The first step is to scan the directory containing the files, this will give us a list of files, we will loop through the list one statute at the time. For our exampel (1997:146) statute we we will have the following set of files:

```
All files connected to this file:
        {'sfst': [u'../lawParse/data/sfs/dl/sfst/1997/146.html'],
         'sfsr': [u'../lawParse/data/sfs/dl/sfsr/1997/146.html']}
```

We then transform and trim the file names so that they will be on the format '1997/146' instead of '../1997/146.html'. The actual parsing step then begins if the file met these three requirements:

1. If we have the file parsed already and it is newer than the incoming raw file, don't parse it again.

2. Filter out documents that are not proper SFS documents. They will have a name like 'N1992:31'[3]

[3]TODO: Explain why.

3. Skip parsing the documents that have been or will be revoked, they are marked as *"The constitution is repealed / shall be repealed"*[4]

The parsing begins with creating a SFS specific Parser, that has a set of properties and rules defined such as regular experssions created for parsing SFS statutes. Like this regular expression for matching a SFS number:

```
reSimpleSfsId = re.compile(r'(\d{4}:\d+)\s*$')
```

The raw string ('r') notation keeps the regular expression clean, without it every backslash would have to be prefixed with another on to escape it. Then we are looking for four (4) digits followed by a colon(:) and one (1) or more digits. The 's' at the end is there to match white spaces (behaves differently with or without the unicode flag turned on). Finally the dollar sign ($) states that the match have to be the end of the string. The other regular expressions are created in a similar fashion, in total there are about 30 expression matching everything from chapter ids and numbered lists to the definition of a revoked date.

When the SFSParser class is initialized and have all it's properties created, we can divide the parsing into two steps. One where we handle the register file (SFSR) and one for the full text (SFST). We begin with creating a registry with *registerposts* containing meta information and other information regarding changes to the statute.

```
Registerpost:
{u'Ansvarig myndighet': Link('u'F\xf6rsvarsdepartementet'',
        uri=u'http://lagen.nu/org/2008/forsvarsdepartementet'),
 u'Ikraft': DateSubject(1997, 7, 1),
 u'SFS-nummer': u'1997:146'}

Registerpost:
{u'Omfattning': [
        u'\xe4ndr. ',
        Link('u'1 \xa7'',
        uri=u'http://rinfo.lagrummet.se/publ/sfs/1997:146#P1')],
 u'Ikraft': DateSubject(2012, 7, 16),
 u'Rubrik': u'F\xf6rordning (2012:334) om \xe4ndring i
        hemv\xe4rnsf\xf6rordningen (1997:146)',
 u'SFS-nummer': u'2012:334'}
```

The first registerpost created for 1997:146 describes basic properties such as responsible authority, entry into force and SFS number. The second[5] registerpost states that there has been some changes to the statute. We can find out if it is a

[4] "Författningen är upphävd/skall upphävas"

[5] There's actually three register posts for 1997:146, I left a 'change post' out to save space

change or if some part has been revoked[6], also we can see the force into entry and the change statute's SFS number. All these values are represented with different data types discussed in the 'Implementation' section, for ex. 'Link', 'DateSubject' or a unicode 'u' string.

The second part is to deal with the full text part of the statute, we do this by creating an intermediate text file with just the raw[7] text. This file is saved in a directory called 'intermediate' on the same level as the downloaded, 'dl' directory in the file tree structure.

The file we have downloaded consist of a header[8] with information similar to the register file. We could probably reuse some information[9] but we have everything set up for parsing and it is a fairly simple procedure so we go ahead and save it again. This information will be part of the header in the HTML file when that is created later on. For the information in the header we create a corresponding object, in the example below we create UnicodeSubjects, with the values *SFS number* and *Headline*, their predicates will be an URI to that object's definition.

```
if key == u'Rubrik':
        meta[key] = UnicodeSubject(val, predicate=self.labels[key])
elif key == u'SFS nr':
        meta[key] = UnicodeSubject(val, predicate=self.labels[key])
```

Here is the values for the variables above when the header infromation is created:

```
key:SFS nr
val:1997:146
self.labels[key]:
        http://rinfo.lagrummet.se/taxo/2007/09/rinfo/pub#fsNummer
key:Rubrik
val:Hemvransforordning (1997:146)
self.labels[key]:
        http://purl.org/dc/terms/title
```

To mark up the full text we call a number of methods on the raw text file, to first of, find out what the part we are looking at should be represented as and then to create an object with that type's properties. The main method is **makeForfattning** when it is called it will in turn call several other methods like **makeKapitel,**

[6]Omfattning can be for example "Change" or "Revoked" ("Ändr." or "Upph.")

[7]It is not the complete downloaded file, we get rid of the "HTML" mark up of the document so that we only have text.

[8]See bilaga X for example input file.

[9]SFS number and responsible authority could be reused for example.

makeParagraf and **makeTabell** and so on. The logic to figure out what each part should be marked up as uses a 'statehandler' to keep track of what part we just parsed and to guess what should come next. If we are inside the **makeStycke** method possible states/methods we can invoke are different types of lists and tables. Then when we reach the end of the current state we start over again and look what state the next part should be, this part can probably not be a list or a table, those states can only be reached from within certain states. Following that logic for example a paragraph can not be called within a chapter, each type has its place in the hierarchy.

All objects have an **"isObject"** and **"makeObject"** method, and they are for example used like the example below shows.

```
def guesState(self):
        try:
                if self.reader.peekLine() == '':
                        handler = self.blankLine
                elif self.isAvdelning():
                        handler = self.makeAvdelning
                elif self.isKapitel():
                        handler = self.makeKapitel
                elif self.isParagraf():
                        handler = self.makeParagrafnd{minted}
                ...
```

When we are done with the parsing the text body we add some additional information to the law's meta information such as time created and preliminary work. To find out if there is any preliminary work we check the *registerposts* in the register we have created, and save the URI to that document if existing.

5.1.3 Generating XHTML

To create an XHTML representation of the parsed statute we use a third party library called *Genshi*[10]. We use Genshi's 'TemplateLoader' to load a template we have created that specifies how we want the our XHTML to look like, ex. which values goes where etc. A nice feature with Genshi is that it uses a Stream-based filtering[11] that allows us to apply various transformations as a template is being processed, without having to parse and serialize the output again.

Below is an example of how the template file renders a headline. As the code snippet shows there is two <h> tag templates to chose from, one if it is a "nor-

[10] See the 'Implementation' section for more information regarding third party libs.
[11] `http://genshi.edgewall.org/wiki/Documentation/filters.html`

mal" headline and one if it is an sub headline, then an extra class is added to the tag.

```
<div py:def="render_rubrik(rubrik)" py:strip="" py:choose="">
    <h py:when="rubrik.type == 'underrubrik'" py:content="rubrik"
        id="${rubrik.id}" class="underrubrik">Underrubrik</h>
    <h py:otherwise="" py:content="rubrik"
        id="${rubrik.id}">Huvudrubrik</h>
</div>
```

5.1.4 RDF markup

```
class UnicodeSubject(PredicateType, UnicodeStructure):
        pass
```

lorem

```
class PredicateType(object):
    """Inheriting from this class gives the child class a
    predicate attribute that describes the RDF predicate to
    which the class is the RDF subject"""
    def __init__(self, *args, **kwargs):
        if 'predicate' in kwargs:
            self.predicate = kwargs['predicate']
            shorten = False
            for (prefix, ns) in Util.ns.items():
                if kwargs['predicate'].startswith(ns):
                    predicateUri = kwargs['predicate']
                    kwargs['predicate'] = kwargs['predicate']
                        .replace(ns, prefix + ':')
                    shorten = True
                else:
                    from rdflib import RDFS
                    self.predicate = RDFS.Resource
            super(PredicateType, self).__init__(*args, **kwargs)
```

Ex: of kwargs and args from PredicateType init:

```
{'predicate': rdflib.URIRef(
    'http://rinfo.lagrummet.se/taxo/2007/09/rinfo/pub#fsNummer')}
(u'1997:146',)

{'predicate': 'dct:references',
 'uri': u'http://rinfo.lagrummet.se/publ/sfs/1994:524'}
(u'f\xf6rordningen (1994:524)',)
```

lorem

5.2 Output

Example of the parsed html file.

5.2.1 RDF

RDF examples from the law

5.2.2 Rättsinformationsprojektet standard

Examples of places where the law is marked up with rip's standard.
Text from their specifications.

5.3 Performance

Timing for 1, 100, 1000, all laws? Maybe a nice chart. What step is the most time/power consuming?

Chapter 6

Conclusions and Discussions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus.

6.1 Results

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus.

6.2 Future development

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus.

6.2.1 RDF Database

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus.

Bibliography

[1] Ted Talk video about Linked Data with Tim Berners-Lee http://www.ted.com/talks/tim_berners_lee_on_the_next_web.html

Appendix A

RDF