Table of Contents

ARDP - Another RDF document parser

RDF is one of the building block of the `semantic web`. The designated format `XML/RDF` has complex and somewhat hard to read syntax. Therefore several alternative formats were created so that it's easier for humans to read them, but most of them are not compatible with each other. The goal of this thesis is to create tool to serialize Turtle (and N-Triples) syntax to simple triples.

The implementation program called ARDP (another RDF document parser) should be focused on performance and correct handling of Unicode charset. The implementation is done in `C` language with use of the newest syntax `C11`. It uses the `GNU Autotools` as it's build system and `clang` as compiler (`GCC` is not tested due to code reliability on clang's features, extensions and syntactic sugar).

The primary development and deployment platforms are `Mac OSX` and `GNU\Linux` or other `*nix` system with support of the Autotools and Clang(LLVM) compiler. The `Windows` platform is supported via `cygwin` but is not being tested. The primary platforms are tested via `Continuous Integration` service `Travis-CI`.

Run compile this project one should execute following steps:

```
git clone https://github.com/michto01/ardp.git
cd ardp
./autogen.sh
mkdir build  #optional
cd build     #optional
../configure # ./configure if optional steps were not taken
make
make check   #optional
make install #may require `sudo` access
```

Current implementation build status: build passing

Introduction

```
One man\'s constant is another man\'s variable. -- A.J. Perlis
```

For humans extracting information from reading the web content is not very challenging, but for machine it's exponentially more complex. To resolve this complexion in extraction important and relevant information from the web the extension of WWW was proposed. This extension is called `semantic web`.

It helps with extracting and processing information from web by human but also more importantly to machines.

Probably most important goal of the semantic web is to describe relationships between individual data sources. For this exact purpose as one of the pillars of the semantic web `Resource Description Framework` - `(RDF)` exists. The RDF denotes those relationships as `triples` which then form vast graphs.

Primary format for description of triples in RDF become `XML/RDF`. Due to it's markup based syntax of XML it's easy for the machines to extrapolate the data from it using already available tools and libraries. The cons of this format are huge file sizes because of the markup overhead and the XML syntax is not optimized for human readability. This predicament gave birth to other formats how to improve the description of the triples. Those formats are usually not compatible between each other and present different solution to the problems.

One of the most know formats is `RDF/Turtle` which is a subset of larger and more powerful syntax Notation-3 usually denoted as simply `N3`.

This thesis focuses exactly on this format and it's simpler subset `N-Triples`. The goal of the implementation if to create memory-efficient, high performance parser to allow fast readout of the triples from files.

1 Resource Description Framework

Resource Description Framework (from here on referred to as simply `RDF`) is `W3C` specification for description and modeling of objects, their relationships and properties. Data stored in RDF format is formed by `TRIPLET: SUBJECT - PREDICAMENT - OBJECT` . The triplet tells us:

```
"SUBJECT has property PREDICAMENT and the predicament has value of
OBJECT"
```

For storing RDF data the special storage techniques and tools were designed, which allows additional functions such as querying, inserting or deleting the data inside.

The other rather interesting property of RDF is ability to deduce additional information - in simplicity it is creation of triplets not (physically) existing in the RDF storage.

1.1 Relationship with the Semantic Web

Semantic web is all about the data. On the internet there are large quantities of the data available in different forms. The most dominant forms are still texts, images and tables. Those are easy to understand for human, as he draws his own logical conclusions about the data and links them together.

Machines do not possess such ability to do logical deductions, so how it can then recognize relationships between information? This exact problem is challenged by `Semantic Web` .

The basic principle is to complement the in human readable form (text, image ...) by metadata. And from the metadata the machine will determine relationships between objects.

The semantic web development is driven by the World Wide Web Consortium and it's technologies specifications contains specification for `RDF` and `OWL` .

1.2 RDF as a graph

At the data represented in `RDF metadata` we can look as on oriented graph. The root node is represented by subject, edge is predicament and the result node is object. The nodes can be of following types:

```
* Entities (has its own definition in dictionary, has its own URI)
* Literals
* Blank nodes (not identified by URI)
```

1.3 RDF serialization formats

For exchanging the `RDF metadata` there are numerous file formats in use. The main focus of this thesis in on the `Notation 3` and particularly it's subset: `Turtle` , `N-Triples` and `N-Quads` . Other well known formats are: `RDF/XML` , `RDF/JSON` , `TriG` . For the following discussion about each format the following example metadata will be used:

OBJECT	PERDICAMENT	SUBJECT
ex:schools#VSB	ex:studiesAt	ex:Student
"xyz123"	ex:login	ex:Student
ex:Student	ex:teaches	ex:Profesor

1.3.1 RDF/XML

Primary format for the data specified in `W3C` documents in 1999 for semantic web RDF description.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:ex="http://example.com/#">

    <rdf:Description rdf:about="http://example.com/Student">
      <ex:studiesAt>http://example.com/schools#VSB</ex:studiesAt>
      <ex:login>xyz123</ex:login>
    </rdf:Description>

    <rdf:Description rdf:about="http://example.com/Profesor">
      <ex:teaches>http://example.com/Student</ex:teaches>
    </rdf:Description>
</rdf:RDF>
```

1.3.2 RDF/JSON

Based on format which drives `Web 2.0` . `JSON` notation was specified for javaScript objects (hence the name JSON - JavaScript Object Notation). It gained popularity in recent years and therefore it should not be supprise to anyone that it's used as alternative to more verbose `RDF/XML` syntax

```
{
  "http://example.org/Student" : {
    "http://example.org/#studiesAt" :
       [
            {
                "value" : "http://example.com/schools#VSB",
                "type"  : "uri"
            }
       ],
    "http://example.org/#login" :
       [
            {
                "value" : "xyz123",
                "type"  : "literal"
            }
       ]
  },
  "http://example.org/Profesor" : {
    "http://example.org/#teaches" :
       [
            {
                "value" : "http://example.com/Student",
                "type"  : "uri"
            }
       ]
  }
}
```

1.3.3 Turtle

Subset of the `Notation 3` syntax omitting some less common construction such as implication (`=>`) and keywords such as `has` . Currently most common alternative to the `RDF/XML` syntax.

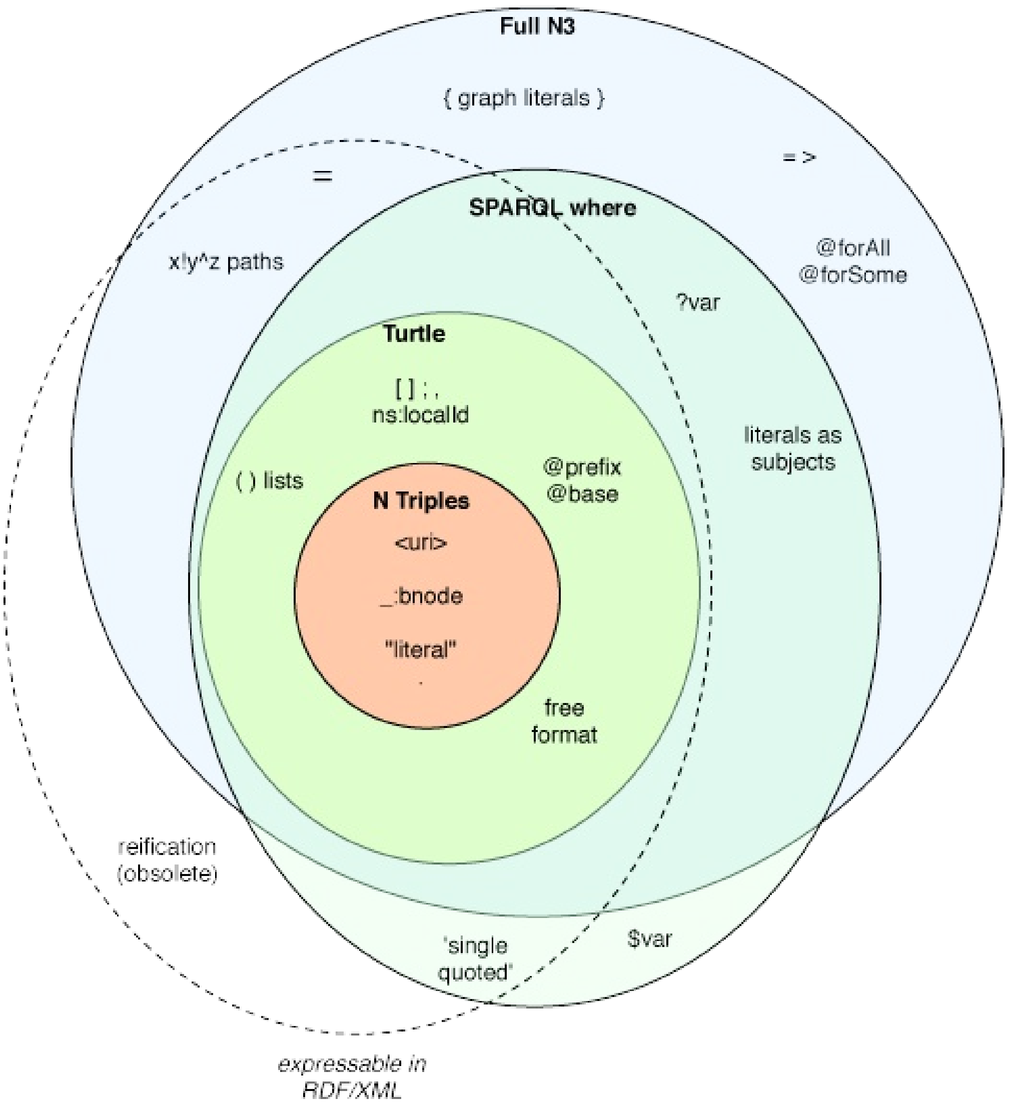
```
@prefix ns0: <http://example.com/#> .

<http://example.com/Student>
  ns0:studiesAt "http://example.com/schools#VSB" ;
  ns0:login "xyz123" .

<http://example.com/Profesor> ns0:teaches "http://example.com/Student" .
```

1.3.3 N-Triples

The subset of the `Turtle` notation simplifying it's syntax and omitting more complex construction in favor of very simple triple syntax.

```
<http://example.com/Student> <http://example.com/#studiesAt> "http://example.com/schools#
<http://example.com/Student> <http://example.com/#login> "xyz123" .
<http://example.com/Profesor> <http://example.com/#teaches> "http://example.com/Student"
```

As is seen from those examples, the `N3` based examples are much easier to read to human.



Character encoding

Historically the programs were using the most basic set of characters standardized as ASCII. Since then many encoding been created to support languages which uses different symbols than latin alphabet.

The N-Triples file specification defined the N-Triples files to be ASCII encoded with escapes to specify the nonstandard characters. The Turtle and XML specifications on the other hand specifies the UTF-8 as the file character encoding format. Since then the new N-Triples files are also allowed to be UTF-8 encoded.

Unicode

Unicode provides unique number for every character, no matter the platform, program or language. That way it is able to provide consistent encoding across all systems and platforms supporting it. It contains more then *120 000* characters in *129* modern and historic scripts and multiple symbol sets. As of June 2015, the most recent standard is *Unicode 8.0*.

The unicode character can be expressed using 4B types, for space efficiency purposes 2B encodings are more commonly used. The 2B encoding allows to save first 2^16 characters, also called the **Basic Multilingual Plane** (BMP), which contains most of modern scripts, without any change. Rest of the characters are expressed using two 2B characters. Unicode has special characters for this purpose, creating *surrogate pairs*.

Unicode also has **Private Use Area** (PUA). It is range of codepoint which the Unicode consortium vowed not to assign to allow custom characters and symbols to be used by the consumers.

ASCII

ASCII stands for *American Standard Code for Information Interchange*. It was developed from telegraphic codes. It defines 127 ($0x00 ... 0x7F$) characters, from which the first 32 characters are are non-printing characters, also called control-characters.

N-Triples define ASCII escape sequences for Unicode codepoints. Table below shows the groups of Unicode characters and their respective escapes in N-Triples. The 'H' character stands for hexadecimal character and the case of the 'u' letter marks short or long codepoint.

***u* character code**	**N-Triples escape**
#x0 - #x8	\uHHHH
#x9, #xA	\t, \n
#xB - #xC	\uHHHH
#xD	\r
#xE - #x1F	\uHHHH
#x20 - #x21	*u* character
#x22	\"
#x23 - #x5B	*u* character
#x5C	\\
#x5D - #x7E	*u* character
#x7F - #xFFFF	\uHHHH
#x10000 - #x10FFFF	\UHHHHHHHH

UTF-8

The UTF-8 is character encoding which is capable of expressing most of the characters and code-points defined by Unicode. The design idea around this encoding was clever hack to allow its backward compatibility with the ASCII character encoding.

To express the ASCII character using the UTF-8 encoding, no change to existing code is required. For rest of the Unicode characters, 2-4B are used. The original proposal contained up to 6B characters, but RFC 3639 restricted the UTF-8 to end at U+10FFFF to match constraint of UTF-16 character encoding. This lead to removal of proposed 5 and 6 byte sequences as well as almost 1000000 of 4 byte ones. Table below demonstrates this encoding.

***u* character code**	**Bytes in seqence**	**Encoded bits**
U+00000000 - U+0000007F	1	0xxxxxxx
U+00000080 - U+000007FF	2	110xxxxx 10xxxxxx
U+00000800 - U+0000FFFF	3	1110xxxx 10xxxxxx 10xxxxxx
U+00010000 - U+001FFFFF	4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

GNU Make toolchain

Most of the open-source project use the build tools to help people to access the program easier and to unify the building process.

One of such tools is the GNU Autotools toolchain. It consists of several programs such as `autoconf` , `aclocal` and `make` .

This toolchain helps to automate creation of Makefiles which then compile programs and it`s dependencies.

The normal working process for building process is then simplified for the user. It usually consist of four steps:

```
./configure [-optional-paramaters]
make
make check
make install
```

In first step we call the script which then populates all of the Makefile templates in our project and check for all dependencies and requirements.

If all of the requirements are met then we can simply follow the next command `make` which calls the tools such as compilers and preprocessors to compile the project. This step takes place in the build directory if the prefix was not specified.

The next (optional) step is to run self test on the compiled binaries and libraries.

If all is well we then proceed to the final step of installing the necessery binaries and libraries into the host system. In this step the `sudo` (or equivalent) is required because the application may be copied to locations requiring administator priviledges to modify.

Installation guide

Following chapters outlines the necessary dependencies and environment for building the **ARDP**. The tested environment were OSX and Ubuntu, but generic approach should apply to all *nix platforms.

The recommened approach on GNU/Linux is to translate the Debian apt-get command into the package manager command of chosen distribution, as package managers usually resolve additional decencies which may be required for the packages and were not discovered during their presence on test distribution.

Linux dependencies

The generic linux installation requires several tools to be installed. This guide suppose that the distribution has the basic development support and that *llvm-clang* is already installed and is relatively up-to-date (3.5.0+). If the target platform is Debian based distribution, single command should build and install all required dependencies:

```
sudo apt-get -y install libdispatch-dev
zlib1g-dev bzip2 libbz2-dev autotools-dev
autoconf ragel lemon
```

Zlib installation:

```
    wget http://www.zlib.net/zlib-1.2.8.tar.gz
    tar -xvzf zlib-1.2.8.tar.gz
    cd zlib-1.2.8
    ./configure && make && sudo make install
```

BZip2 installation:

```
    wget http://www.bzip.org/1.0.6/bzip2-1.0.6.tar.gz
    tar -xvzf bzip2-1.0.6.tar.gz
    cd bzip2-1.0.6
    ./configure && make && sudo make install
```

LibDispatch installation:

```
          #Needs to have CLANG installed for the BlockRuntime
          git clone git://git.macosforge.org/libdispatch.git
          cd libdispatch
          sh autogen.sh
          ./configure CC=clang --with-blocks-runtime=/usr/local/lib
          make check
          make && sudo make install
```

OSX dependencies

In order to install the dependencies used to build the ARDP on OSX, one should install XCode using the the AppStore. After installation of the Xcode, command for installing the CLT should be entered in terminal:

```
xcode-select --install
```

If any of the dependencies are missing, it is recommended to install Homebrew package manager from **https://brew.sh** and installing the depencecies from it, or installing **wget* command and follow the steps as specified in linux generic installation from source.

Installing the ARDP

After all dependencies are installed the ARDP should be ready to be build and installed.

ARDP can be obtained using two methods: one is the ARDP version included on CD which is part of this thesis. In such case one should copy the ***ardp-*.tar.gz***, unpack it and skip the following steps related to git.

The other method is to download the sources from github git repository. The github has the newest version of the work regardless of the status of CD and is recommended as source, for improvements and changes are likely to be made to codebase. The github code which is identical to the content of the CD is tagged with label 'thesis'.

```
        git clone https://www.github.com/michto01/ardp
        cd ardp
        sh autogen.sh # bootstrap the package.
```

As is visible in the previous listing, the git repository requires aditional step, namely running the script autogen.sh, which creates files, which are target specific and therefore are in .gitignore file.\bigbreak

Optionally, one can generate the new code from the tools ragel and lemon. This allows to change and fine-tune the behavior of the generators tools; eg. change the style of generated output for ragel:

```
# from ardp root directory
ragel src/lib/lexers/turtle.rl -G2 src/lib/lexers/turtle.c
cd src/lib/parsers
lemon -s ./turtle_parser.y
```

The last step is to finally build the ARDP. It is done using the standard Unix autotools combo; to maintain the clarity of the package, build is done in separate directory as to avoid processed files mixup with source code:

```
mkdir build && cd build # optional
../configure               # ./configure if building from root
make
make check # optional
sudo make install
```