

Chapter 1

Utilities for Handling Descot RDF Stores

The module `descot-rdf-utilities` allows for a common place to define utilities for handling Descot RDF Stores, which include things like accessing fields and objects of predicates relating to libraries, like record accessors, making it much easier to get to the library data, as well as defining some basic constant URIs that are used everywhere.

1. URI Constants

`descot-rdf-utilities` defines a number of URI constants to reference already defined URIs for Descot Stores.

```

<descot uris>≡
(define dscts
  (lambda (tail)
    (string-append "http://descot.sacrideo.us/10-rdf-schema#" tail)))
(define dscts:name (dscts "name"))
(define dscts:names (dscts "names"))
(define dscts:desc (dscts "desc"))
(define dscts:homepage (dscts "homepage"))
(define dscts:authors (dscts "authors"))
(define dscts:email (dscts "email"))
(define dscts:license (dscts "license"))
(define dscts:categories (dscts "categories"))
(define dscts:contact (dscts "contact"))
(define dscts:Library (dscts "Library"))
(define dscts:copyright (dscts "copyright"))
(define dscts:copyright-type (dscts "Copyright"))
(define dscts:copyright-year (dscts "copyright-year"))
(define dscts:copyright-owner (dscts "copyright-owner"))
(define dscts:modified (dscts "modified"))
(define dscts:creation (dscts "creation"))
(define dscts:version (dscts "version"))
(define dscts:implementation (dscts "implementation"))
(define dscts:location (dscts "location"))
(define dscts:Single-file (dscts "Single-file"))
(define dscts:Archive (dscts "Archive"))
(define dscts:CVS (dscts "CVS"))
(define dscts:cvs-root (dscts "cvs-root"))
(define dscts:cvs-module (dscts "cvs-module"))

```

There are also some standard RDF URIs that are commonly used with Descot Stores. These are defined here as well.

```

<rdf uris>≡
(define rdfs
  (lambda (tail)
    (string-append "http://www.w3.org/2000/01/rdf-schema#" tail)))
(define prefix-with-rdf
  (lambda (tail)
    (string-append "http://www.w3.org/1999/02/22-rdf-syntax-ns#" tail)))
(define rdf:nil (prefix-with-rdf "nil"))
(define rdf:rest (prefix-with-rdf "rest"))
(define rdf:first (prefix-with-rdf "first"))
(define rdf:Type (prefix-with-rdf "type"))

```

2. Looping over RDF Lists

One nice ability is to automatically loop over RDF Lists defined with the standard nested RDF List nodes. This is accomplished with the help of `foof loop`, which allows for the extension of its looping macros to handle other constructs such as RDF Lists.

Looping is accomplished as a standard FOR iterator in `foof loop`, where the iterator is given a store and a node as its starting place.

```
(for elem rest (in-rdf-list store start-node))
```

(in-rdf-list iterator)≡

```
(define-syntax in-rdf-list
  (lambda (x)
    (syntax-case x ()
      (syntax-case x ()
        [(_ (element-variable pair-variable) (rdf-store list-node) next . rest)
         (and (identifier? #'element-variable) (identifier? #'pair-variable))
         #'(next
            ([rdf-pair]
             (rdf-graph/triples-by-subject rdf-store list-node))]
            [pair-variable rdf-pair
             (if (rdf-object=? rdf:nil rdf-rest)
                 #f
                 (rdf-graph/triples-by-subject rdf-store rdf-rest)))]
            ()
            [(not pair-variable)]
            [(element-variable)
             (find-rdf-object-by-predicate pair-variable rdf:first)]
            [(rdf-rest) (find-rdf-object-by-predicate pair-variable rdf:rest)]]
            . rest))]
        [(_ (element-variable) (rdf-store list-node) next . rest)
         (identifier? #'element-variable)
         #'(in-rdf-list (element-variable pair)
                       (rdf-store list-node)
                       next . rest))]))
```

3. Finding over Lists of Triples

It is convenient, given a list of RDF Triples, to search for a single RDF Triple. This can be done using `find` but is inconvenient because of the constant higher-order procedures that have to be written to handle the RDF comparisons. `descot-rdf-utilities` defines a couple convenience functions for handling these things more easily.

`find-by-predicate` searches a list of RDF Triples for a matching triple based on common predicates.

(find-by-predicate)≡

```
(define find-rdf-predicate
  (lambda (lst pred)
    (find (lambda (e) (rdf-predicate=? (rdf-triple/predicate e) pred)) lst)))
```

`find-rdf-object-by-predicate` extends `find-by-predicate` to the special case where one just requires the object of the RDF Triple.

```
(find-rdf-object-by-predicate)≡
(define find-rdf-object-by-predicate
  (lambda (lst predicate)
    (let ([res (find-rdf-predicate lst predicate)])
      (and res (rdf-triple/object res)))))
```

4. Parsing Turtle Files

While Taylor Campbell's RDF Parsers are fairly easy to use, it makes sense to simplify the process of parsing by encapsulating the needed code into a single procedure that only takes a filename of the rdf-store in turtle format to work.

```
(parse-turtle-file)≡
(define parse-turtle-file
  (lambda (fname . maybe-graph)
    (parse-file turtle-parser:document fname
      (make-turtle-parser-context
        (lambda (triple graph)
          (rdf-graph/add-triple! graph triple)
          graph)
        (if (null? maybe-graph) (make-rdf-graph) (car maybe-graph)))
      (lambda (graph context stream) graph)
      (lambda (perror context stream)
        (error 'parse-turtle-file
          "Parse Error at ~a: ~{ ~s~}~%"
          (parse-error/position perror)
          (parse-error/messages perror))))))
```

5. Handling Categories

Categories are a special case field because it is used to group libraries into a tree where there would otherwise be no organization. This is a common thing to do with packaging systems, and matches what users expect. However, RDF Stores don't always have this information so readily available, so we have to make some procedures to help.

Firstly, it is useful to know all the categories that exist in a store.

```
(store categories)≡
(define store-categories
  (lambda (store)
    (remove-duplicate-categories
      (collect-list
        (for category-list
          (in-list (rdf-graph/triples-by-predicate store dscts:categories)))
        (for category (in-rdf-list store (rdf-triple/object category-list)))
          (rdf-literal/lexical-form category))))))
```

Secondly, we would like to find out which libraries have a given category.

```
<libraries in category>≡
(define libraries-in-category
  (lambda (category)
    (collect-list
      (for lib-triple
        (in-list
          (rdf-graph/triples-by-predicate (current-store) dscts:categories)))
      (if (has-category? category (rdf-triple/object lib-triple))
          (rdf-triple/subject lib-triple))))))
```

`libraries-in-category` requires a filter to determine whether a library has a given category. This serves a similar purpose to things like `memq`.

```
<has category>≡
(define has-category?
  (lambda (category rdf-category-list)
    (collect-or (for cat (in-rdf-list (current-store) rdf-category-list)
                 (string=? category (rdf-literal/lexical-form cat))))))
```

It is also nice to get the list of categories for a library.

```
<library's categories>≡
(define library-categories
  (lambda (store db)
    (collect-list
      (for category
        (in-rdf-list store
          (rdf-predicate-map/lookup db dscts:categories rdf:nil)))
      category))))
```

6. Making RDF Predicate Maps

`rdf-maps` provides for a type of RDF Predicate Map which is a table/dictionary keyed on RDF Predicates. This is nice, but unfortunately, there is no way to get this from the normal RDF Graph interface, so we have to do this on our own. This is a bit of a hack, but until the interface is made better, there probably isn't much that I can do.

```
<make rdf predicate map>≡
(define make-filled-rdf-predicate-map
  (lambda (store name)
    (let ([db (make-rdf-predicate-map)])
      (rdf-graph/for-each-triple-by-subject store name
        (lambda (triple)
          (rdf-predicate-map/insert! db (rdf-triple/predicate triple)
            (rdf-triple/object triple))))
      db)))
```

7. RDF Node to String Conversion

Why isn't there a generalized string converter for RDF Triples? I do not know, but at least we can get something working on our own.

```
<rdf node to string>≡  
(define rdf-node->string  
  (lambda (node)  
    (cond  
      [(rdf-uri-ref? node) (rdf-uri-ref->string node)]  
      [(rdf-bnode? node) (string-append "_:" (rdf-bnode/name node))]  
      [(rdf-literal? node) (rdf-literal/lexical-form node)]  
      [else (error 'rdf-node->string "Unknown type: ~s" node)])))
```

8. RDF Library Accessors

`descot-rdf-utilities` also defines a series of library accessors for getting at the fields of an RDF Library node and transforming them into something useful.

(RDF Library Accessors)≡

```
(define library-ids
  (lambda (store)
    (map rdf-triple/subject
      (filter
        (lambda (triple)
          (rdf-predicate=? rdf:type (rdf-triple/predicate triple)))
        (rdf-graph/triples-by-object store dscts:Library))))))

(define library-title
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:name #f)))

(define library-names
  (lambda (store db)
    (collect-list
      (for elem
        (in-rdf-list store (rdf-predicate-map/lookup db dscts:names rdf:nil)))
      elem)))

(define library-description
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:desc #f)))

(define library-copyright
  (lambda (store db)
    (let ([year (rdf-predicate-map/lookup db dscts:copyright-year #f)]
          [owner (rdf-predicate-map/lookup db dscts:copyright-owner #f)])
      (or (and year owner '(,year ,@(make-author-pair store owner)))
          '()))))

(define library-homepage
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:homepage #f)))

(define library-license-name
  (lambda (store db)
    (let ([res (rdf-predicate-map/lookup db dscts:license #f)])
      (if res
          (find-rdf-object-by-predicate
            (rdf-graph/triples-by-subject store res)
            dscts:name)
          (make-rdf-plain-literal "Unknown" #f)))))

(define library-authors
  (lambda (store db)
    (collect-list
      (for author
        (in-rdf-list store
```

```

        (rdf-predicate-map/lookup db dscts:authors rdf:nil)))
    (make-author-pair store author))))

(define make-author-pair
  (lambda (store author)
    (let ([triples (rdf-graph/triples-by-subject store author)])
      (cons (author-email triples) (author-name triples)))))

(define author-email
  (lambda (triples-list)
    (find-rdf-object-by-predicate triples-list dscts:email)))

(define author-name
  (lambda (triples-list)
    (find-rdf-object-by-predicate triples-list dscts:name)))

(define library-contact
  (lambda (store db)
    (let ([res (rdf-predicate-map/lookup db dscts:contact #f)])
      (if res
          (make-author-pair store res)
          (error 'library-contact "Maintainer should not be empty."))))))

(define library-created
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:creation #f)))

(define library-modified
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:modified #f)))

(define library-version
  (lambda (db)
    (rdf-predicate-map/lookup db dscts:version #f)))

(define library-implementation
  (lambda (store db)
    (let* ([impl-node (rdf-predicate-map/lookup db dscts:implementation #f)]
           [impl-list (rdf-graph/triples-by-subject store impl-node)])
      (cons
       (find-rdf-object-by-predicate impl-list dscts:name)
       (find-rdf-object-by-predicate impl-list dscts:homepage)))))

(define library-location
  (lambda (store db)
    (let ([node (rdf-predicate-map/lookup db dscts:location #f)])
      (if node
          (let* ([location-info (rdf-graph/triples-by-subject store node)]
                 [type (find-rdf-object-by-predicate location-info rdf:type)])
            (cond
             [(rdf-object=? type dscts:Single-file) (cons 'single node)]
             [(rdf-object=? type dscts:Archive) (cons 'archive node)]
             [(rdf-object=? type dscts:CVS)
              ])))))))

```

```
(cons 'cvs
      (cons
        (find-rdf-object-by-predicate location-info dscts:cvs-root)
        (find-rdf-object-by-predicate location-info
          dscts:cvs-module))))))
'(unknown))))
```

9. Module Declaration

`descot-rdf-utilities` is a module declared thus:

<rdf-util.ss>≡
<License>

```
(module descot-rdf-utilities
  (dscts dscts:name dscts:desc dscts:homepage dscts:authors dscts:email
    dscts:license dscts:categories dscts:contact dscts:Library dscts:version
    dscts:copyright dscts:copyright-type dscts:copyright-owner
    dscts:copyright-year dscts:implementation dscts:location dscts:modified
    dscts:Single-file dscts:Archive dscts:CVS
    rdfs prefix-with-rdf rdf:nil rdf:rest rdf:first rdf:Type
    store-categories
    (in-rdf-list rdf-graph/triples-by-subject rdf-object=?
      find-rdf-object-by-predicate)
    find-rdf-predicate find-rdf-object-by-predicate
    parse-turtle-file
    libraries-in-category
    rdf-node->string
    library-ids library-names
    make-filled-rdf-predicate-map
    library-title library-description library-homepage library-copyright
    library-authors library-license-name library-created library-modified
    library-contact library-version library-implementation
    library-location library-categories)
```

<imports>

<rdf uris>

<descot uris>

<parse-turtle-file>

<in-rdf-list iterator>

<libraries in category>

<store categories>

<has category>

<make rdf predicate map>

<rdf node to string>

<find-rdf-object-by-predicate>

<find-by-predicate>

<RDF Library Accessors>

<library's categories>

)

10. Imports

The following imports are expected from Aaron Hsu's Chez Scheme Libraries for Chez Scheme Version 7.4.

```
<imports>≡  
(import scheme)  
(import rdf-list-graphs)  
(import rdf)  
(import rdf-maps)  
(import text-parser-combinators)  
(import rdf-turtle-parser)  
(import parse-errors)  
(import foof-loop)  
(import nested-foof-loop)
```

11. Licensing

descot-rdf-utilities is licensed under an ISC License.

```
<License>≡  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;;; Utilities for handling Descot RDFs  
;;;  
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>  
;;;  
;;; Permission to use, copy, modify, and distribute this software for  
;;; any purpose with or without fee is hereby granted, provided that the  
;;; above copyright notice and this permission notice appear in all  
;;; copies.  
;;;  
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL  
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED  
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE  
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL  
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA  
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER  
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR  
;;; PERFORMANCE OF THIS SOFTWARE.
```

Chapter 2

RDF Library Formatting in HTML

Descot RDF programs can be converted into HTML representations in the form of SXML so that they can be incorporated directly into web pages. To do so, we rely on two procedures: `library->html` and `library->short-html`. `library->html` handles formatting a library in a full page format with all the information, and `library->short-html` will format a library with just the minimally relevant information optimized for listings.

This interface is documented by the `rdf2html.ss` file, which is a module exporting the above procedures:

```
<rdf2html.ss>≡  
<License>  
  
(module rdf2html  
  (library->html library->short-html)  
  
  <rdf2html imports>  
  
  <long library conversion>  
  
  <short library conversion>  
  
  <long library rendering>  
  
  <rendering a short library>  
  
  <summary line>  
  
  <default empty library rendering>  
  
  <render an archive download>  
  
  <rendering single file downloads>  
  
  <rendering CVS locations>  
  
  <generic location rendering>  
  
  <additional rendering procedures>  
  
)
```

1. Rendering

A full library description has fields defined for the title of the library, the primary name, a description, a list of alternative libraries, its homepage, the dependencies, alternative library names, the bindings that library exports, its license, authors, creation date, last modification date, maintainer, version, implementation, categories, copyright information, and its download and retrieval information. To render this information, we use a set of rendering procedures. At the moment, let's avoid doing the alternative libraries, exports, and imports.

There should also be a way to get to the literal RDF store description that is stored in some format. To do this, we put a left float containing a link to the actual location of the library.

```
<long library rendering>≡
(define render-library
  (lambda (store name db)
    (render-wrap
      (render-left-float
        (render-homepage
          (encode-library-id (render-rdf-object name))
          "RDF"))
      (render-title (render-homepage (library-homepage db) (library-title db)))
      (render-names (library-names store db))
      (render-categories (library-categories store db))
      (render-description (library-description db))
      (render-table
        (render-row "Version" (render-rdf-object (library-version db)))
        (render-row "Implementation"
          (render-implementation (library-implementation store db)))
        (render-row
          "Authors"
          (fold-right (lambda (e s) (if (null? s) '(,e) '(,e " " ,@s)))
            '()
            (map render-author
              (library-authors store db))))
        (render-row "Maintainer"
          (render-author (library-contact store db)))
        (render-row "Copyright"
          (render-copyright (library-copyright store db)))
        (render-row "License"
          (rdf-literal/lexical-form (library-license-name store db)))
        (render-row "Created On:" (render-time (library-created db)))
        (render-row "Last Modified On:"
          (render-time (library-modified db))))
      (render-download-info store (library-location store db))))))
```

If we are rendering only a short library, we only need to provide the author, title, list of names, and a description (which may be cut short). It is probably also good to provide some version information.

```
<rendering a short library>≡
(define render-short-library
  (lambda (prefix store name db)
    (render-wrap
      (render-left-float (render-author (car (library-authors store db))))
      (render-title
        (render-browse-id-link prefix name (library-title db)))
      (render-names (library-names store db))
      (render-description (library-description db))
      (render-summary store db))))
```

We would like the short description to have a small summary of most of the important information, such as version and implementation, so we implement this as a special line after the description.

```
<summary line>≡
(define render-summary
  (lambda (store db)
    '(p (@ (class "lib_summary"))
      "Version " ,(render-rdf-object (library-version db))
      " for " ,(render-implementation (library-implementation store db))
      " via " ,(render-download-type (car (library-location store db)))
      ".")))
```

When `library->html` is called, it should be called with the Descot RDF Store and the name of the library to be converted to HTML. If the library is not found, a default rendering for non-existent libraries should be returned, otherwise, its value should be the result of rendering the library.

```
<default empty library rendering>≡
(define render-no-library
  (lambda (name)
    (render-wrap
      (render-title "Sorry!")
      '(p "Library " ,name " does not exist.))))
```

Finally, we can define `library->html` and `library->short-html`. In `library->html` we expect to receive the RDF Store along with the library name.

```
<long library conversion>≡
(define library->html
  (lambda (rdf-store library-name)
    (let ([db (make-filled-rdf-predicate-map rdf-store library-name)])
      (if (zero? (rdf-predicate-map/size db))
          (render-no-library library-name)
          (render-library rdf-store library-name db)))))
```

In `library->short-html` we expect to get a path prefix for the browsing id, which we prepend to the library id to get a URL pointing to the actual long description of the library. This would usually be something like “`http://descot.sacrideo.us/db/browse?id=`”. Other than this, the interface is the same as `library->html`.

(short library conversion)≡

```
(define library->short-html
  (lambda (browse-id-prefix rdf-store library-name)
    (let ([db (make-filled-rdf-predicate-map rdf-store library-name)])
      (if (zero? (rdf-predicate-map/size db))
          (render-no-library library-name)
          (render-short-library browse-id-prefix rdf-store library-name db))))))
```

We now need to define all of our supporting rendering procedures.

```
<additional rendering procedures>≡
(define render-wrap (lambda (elems '(div ,@elems)))
  (define render-title (lambda (title) '(h3 ,(render-rdf-object title))))
  (define render-description (lambda (text) '(p ,(render-rdf-object text))))
  (define render-table (lambda (rows '(table ,@rows)))
    (define render-row (lambda (cols '(tr ,@(map render-column cols))))
      (define render-header-row (lambda (cols '(tr ,@(map render-head cols))))
        (define render-column (lambda (column) '(td ,column))
          (define render-head (lambda (column) '(th ,column)))

          (define render-names
            (lambda (names)
              '(p (@ (class "library_names")
                ,fold-right (lambda (e s) (if s '(,e " , " ,@s) '(,e))
                  #f
                  (map rdf-literal/lexical-form names))))))

          (define render-list
            (lambda (type attrs elems)
              '(,type (@ ,@attrs) ,@(map (lambda (e) '(li ,e)) elems))))

          (define render-author
            (lambda (author)
              (let ([email (rdf-literal/lexical-form (car author))]
                    [name (rdf-literal/lexical-form (cdr author))])
                (or (and name email '(a (@ (href ,(make-email-href email))) ,name))
                    (and email '(a (@ (href ,(make-email-href email))) ,email))
                    name
                    ""))))))

          (define make-email-href
            (lambda (email)
              (string-append "mailto:" email)))

          (define render-browse-id-link
            (lambda (prefix url txt)
              (render-homepage
                (let ([encoding (encode-library-id (render-rdf-object url))])
                  (if (pair? encoding)
                      (apply string-append (cons prefix encoding))
                      (string-append prefix encoding)))
                txt)))

          (define encode-library-id
            (make-char-quotator '((#\# . "%23"))))

          (define render-homepage
            (lambda (url txt)
              (if url
                  '(a (@ (href ,(render-rdf-object url))) ,(render-rdf-object txt))
                    (render-rdf-object txt))))
```

```

(define render-copyright
  (lambda (fields)
    (let ([year (car fields)]
          [author (cdr fields)])
      '(,(rdf-literal/lexical-form year) ,(render-author author))))))

(define render-time
  (lambda (obj)
    (rdf-literal/lexical-form obj)))

(define render-left-float
  (lambda (elems)
    '(div (@ (class "left_box")) ,@elems)))

(define render-rdf-object
  (lambda (obj)
    (cond
      [(pair? obj) obj]
      [(string? obj) obj]
      [(rdf-literal? obj) (rdf-literal/lexical-form obj)]
      [else ""])))

(define render-implementation
  (lambda (impl)
    (render-homepage (cdr impl) (car impl))))

(define render-categories
  (lambda (cats)
    '(p (@ (class "categories")) "Categories: "
      ,@(fold-right
        (lambda (e s)
          (let ([ne (capitalize-string (render-rdf-object e))])
            (if s '(,ne " , " ,@s '(,ne))))
          #f
          cats))))))

(define render-download-type
  (lambda (type)
    (case type
      [(cvs) "CVS"]
      [(archive) "Archive"]
      [(single) "a single Scheme file"]
      [else "Unknown means"])))

```

2. Rendering Locations and Download Information

When a library instance is examined, it will have information on how to download it. There are three main distribution methods used by the average library distributor: source repository, archive, or single file. That is, either the library is contained in a single file, some sort of archive file, or it is available through some source control mechanism like CVS.

Rendering a single file is easy to do, since the `dscts:location` predicate will have an object pointing to a node that is itself the URL for the download. Additionally, the one sentence that will be associated with that node is the `type` field indicating that the node is a Single File download node. Easy to render.

```
<rendering single file downloads>≡
(define render-single-file-location
  (lambda (node)
    '(p
      "This library is distributed as a single, self-contained "
      ,(render-homepage node "Scheme File")
      ".")))

```

Rendering an archive really isn't that much more difficult:

```
<render an archive download>≡
(define render-archive-location
  (lambda (node)
    '(p
      "This library is distributed as an "
      ,(render-homepage node "archive")
      ".")))

```

Rendering source control is a little harder, but should be done extensibly. In the simple case, we can handle special cases like CVS, where the only important things are the `CVSROOT` and `Module` names, [TODO] though it would be nice to have the command for downloading right there:

```
<rendering CVS locations>≡
(define render-cvs-location
  (lambda (root module)
    '((p "This library is distributed using CVS."
      ,(render-table
        (render-header-row "CVSROOT" "Module")
        (render-row
          (render-rdf-object root)
          (render-rdf-object module))))))

```

Once we have these specialized procedures, we can dispatch on the type of the location node.

```
<generic location rendering>≡
(define render-download-info
  (lambda (store info)
    (case (car info)
      [(single) (render-single-file-location (cdr info))]
      [(archive) (render-archive-location (cdr info))]
      [(cvs) (render-cvs-location (cadr info) (caddr info))]
      [else '()])))

```

3. Imports

rdf2html requires the following imports for Chez Scheme Version 7.4:

```
(rdf2html imports)=  
(import scheme)  
(import rdf-list-graphs)  
(import rdf)  
(import rdf-maps)  
(import foof-loop)  
(import nested-foof-loop)  
(import descot-rdf-utilities)  
(import oleg-util)
```

4. Licensing

This file is licensed under an ISC License:

```
(License)+=  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;;; Parse RDF Collections to make a web hierarchy  
;;;  
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>  
;;;  
;;; Permission to use, copy, modify, and distribute this software for  
;;; any purpose with or without fee is hereby granted, provided that the  
;;; above copyright notice and this permission notice appear in all  
;;; copies.  
;;;  
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL  
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED  
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE  
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL  
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA  
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER  
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR  
;;; PERFORMANCE OF THIS SOFTWARE.
```

Chapter 3

Descot Web Application

The Descot Web Application provides a web-based interface for accessing the Descot System for search, browsing, and submission. It is designed to permit easy use for those who do not wish to have a special client that integrates with their system.

1. Pages

There are two main dynamic pages defined for Descot at the moment: Browsing and Searching. Browsing includes the display of normal library descriptions and information in HTML form. Submitting is also handled dynamically, so we should take care of that here as well.

```
<Pages>≡  
<Browsing>
```

```
<Searching>
```

```
<Submitting>
```

Pages that should be added are submissions and RDF Library Retrieval, where each library is exported with its actual properties. These are a major TODO.

Browsing includes displaying the library, showing a navigatable category list, and showing the libraries in a given category. We give three separate page procedures to handle each of these functions individually.

```
<Browsing>≡  
<Library>
```

```
<Category Browse>
```

```
<Category List>
```

The simplest library page to do, and the most used, is the library rendering page, which will simply display the results of querying for a specific library URL.

```
<Library>≡  
(define library-page  
  (lambda (url)  
    (descot-wrapper "Browse" descot-stylesheet  
      (library->html (current-store) url))))
```

[Notice the use of `descot-wrapper` to help everything go along.]

Next, we use our iterators and the like to get the listing of libraries for each category. We always try to make sure that category names are capitalized when they are displayed, but lower-cased when they are stored.

```
<Category Browse>≡
(define category-page
  (lambda (category)
    (descot-wrapper (format #f "Category ~a" (capitalize-string category))
      descot-stylesheet
      (collect-list
        (for lib-name (in-list (libraries-in-category category)))
          (library->short-html (string-append descot-browse-path "?id=")
            (current-store)
            lib-name))))))
```

Here we do have a little bit of a hack since we know the browsing interface, and we pass a parameter `"?id="` explicitly. This will be okay, and I want to avoid over engineering something before it even gets off the ground.

When someone first goes to the browse link, if they do not enter any kind of query parameter in the URL, they will be shown a list of the available package categories, on which they can click and navigate. We use `make-column-table-with-size` to layout the table for us, expecting that it will lay them out in downward alphabetical order according to the columns we specify. This column width is set externally to help with different layouts.

```
<Category List>≡
(define browse-page
  (lambda ()
    (descot-wrapper "Browse" descot-stylesheet
      (let ([cats (store-categories (current-store))])
        (if (null? cats)
          '(p "No Categories are available.")
          (make-column-table-with-size
            (browser-column-width)
            (categories->links cats)))))))
```

Now, we progress to searching, which isn't nearly so complicated in its overall layout, even though it does use some procedures in the back to hide some complexity. Overall, there is the basic search results page, and the Advanced Search page, which is the landing page for search.

```
<Searching>≡
<Search Results>
```

```
<Advanced Search>
```

Advanced search is not yet implemented, so we just use a place holder there.

```
<Advanced Search>≡
(define advanced-search-page
  (lambda ()
    (descot-wrapper "Advanced Search" descot-stylesheet
      '(p "Sorry, advanced search is not yet implemented.))))
```

To implement the search results, we have to take the `match?` procedure that we are given and use it on every single library that we can find. We expect the `match?` procedure to be of the kind that we can apply just to a library node/ID.

We use `library->short-html` to print our results in a more succinct fashion from the main library browsing page.

```

<Search Results>≡
(define search-page
  (lambda (match?)
    (define search
      (lambda ()
        (collect-list
          (for name
            (in-list (matching-libraries match? (library-ids (current-store))))))
        (library->short-html
          (string-append descot-browse-path "?id=")
          (current-store)
          name))))
      (descot-wrapper "Search Results" descot-stylesheet
        (let ([results (search)])
          (if (null? results)
              '(p "Sorry, no results found.")
              results))))))

```

2. Library Submission

When someone wishes to submit a new library, at the moment, we will allow them to do so without much help. Right now, the submission on the DWA is merely a form with fields that are not checked or anything. The submissions will then be saved in a special folder that is given by the web parameters module. We will use gensym printing for handling the unique names for each submission. However, normal `write` form gensyms come out with funny characters in them, so we will need to convert them to something nicer:

```

<Submitting>≡
(define format-gensym
  (lambda (id)
    (define cleaner (make-char-quotator '(#\space . ":")))
    (let ([id-string (with-output-to-string (lambda () (write id)))]
          (apply string-append
            (cleaner (substring id-string 2 (1- (string-length id-string)))))))

```

submit-page will return out the basic form used to accept submissions.

```
<Submitting>+≡
(define submit-page
  (lambda ()
    (descot-wrapper "Submit a New Library" descot-stylesheet
      '(form (@ (method "POST") (enctype "multipart/form-data")
                (action ,descot-submit-path))
              (p (input (@ (name "code") (type "file")))
                  (input (@ (name "file-type") (type "radio") (value "srdf")
                            (checked "checked")))
                    "SRDF"
                  (input (@ (name "file-type") (type "radio") (value "turtle")))
                    "Turtle")
                  (p (button (@ (type "submit")) "Submit Library"))))))))
```

On a successful submission, we need to let them know they succeeded.

```
<Submitting>+≡
(define successful-submit-page
  (lambda ()
    (descot-wrapper "Success!" descot-stylesheet
      '(p
        "Thank you for submitting a library to Descot and helping "
        "us to grow! At the moment, submissions are not updated "
        "automatically in the database; they must first be reviewed "
        "and if they are approved, they will show up in the database "
        "within a couple of days [hopefully]. Please bear with us "
        "as we continue to improve Descot!"))))
```

On failure, we should print out an error, and if debugging is enabled, we should print out the error message.

```
<Submitting>+≡
(define submit-failure-page
  (lambda (name fmts . vals)
    (descot-wrapper "Failure!" descot-stylesheet
      '(p
        "An error occured when submitting your library. "
        "It is possible that the Library is not formatted "
        "correctly, or that there was a server error. "
        "Please examine your library for errors and try again. "
        "If this error persists, please contact "
        (descot-maintainer)
        "."))))
```

The maintainer comes from the web parameters `descot-maintainer-email` and `descot-maintainer-name`.

```
<Submitting>+≡  
(define descot-maintainer  
  (lambda ()  
    '(a (@ (href ,descot-maintainer-email)) ,descot-maintainer-name)))
```

See the next section for information about how we actually handle the submission when it comes in.

3. Page Handlers

Page handlers in MIT Scheme `mod_lisp` handle the urls and mime-types that are requested from the server. Here we use mostly URL based dispatch with default mime types. Generally, other things should be served statically by the Apache Web Server if it can for various reasons.

There are really only two search URLs defined, the browse and the search path. Each one is expected to take some parameters to help determine what page should be returned.

Additionally, we have a general RDF handler for those URLs that should return some kind of Turtle representation of the sentences containing that URL as the root.

```
<Handlers>≡  
<Browse Handler>
```

```
<Search Handler>
```

```
<RDF Handler>
```

```
<Submission Handler>
```

For the browse URL the two mutually exclusive parameter `id` and `cat` tell us whether we should be displaying a library or a category listing. If neither of those are used, then we use the basic category listing.

```
<Browse Handler>≡  
(define-subtree-handler descot-browse-path 'application/xhtml+xml  
  (lambda (path port)  
    (let ([id (http-request-url-parameter 'id)]  
          [cat (http-request-url-parameter 'cat)])  
      (write-html-page  
        (cond  
          [id (lambda () (library-page id))]  
          [cat (lambda () (category-page cat))]  
          [else browse-page])  
        port))))
```

For search, we use the Advanced Search page as our landing page, or, if we are given a query, we try to list the results.

```
<Search Handler>≡
(define-subtree-handler descot-search-path 'application/xhtml+xml
  (lambda (path port)
    (let ([query (http-request-url-parameter 'q)])
      (write-html-page
        (cond
          [query (lambda () (search-page (make-search-proc query)))]
          [else advanced-search-page])
        port))))
```

4. Serving RDF Sentences for a Library

When a user tries to retrieve a URL which corresponds to the URL of a given RDF entity, they should be presented with the set of RDF sentences that make up that library. If they want this information in another form, they can use the browse pages to get that. For now, this is the paradigm I want to use, but this may change, since technically it is possible to serve the HTML with the RDF depending on what the client requests.

At any rate, right now, the easiest way to do this is to take the url request and reconstitute the entire URL from the pieces in `mod_lisp`, and check to see whether such a thing exists as a subject in the current store. If it does, then we just print out all the sentences that have it as a subject.

We need to get around some other “features” of Apache and `mod_lisp` in order to serve all the right URLs, since we can’t assign the lisp-handler to take everything and then leave out one specific file set; root lisp-handlers don’t seem to be working the way they should. However, since all of our RDF files are in `libs/`, `impls/`, &c., we can workaroud this problem by explicitly listing these in the `httpd.conf` file.

This won’t make a difference in our default handler here, except that it will never have to serve static content.

```
<RDF Handler>≡
(define-subtree-handler descot-rdf-path 'application/x-srdf
  (lambda (path port)
    (write-descot-request
      (format #f "http://~a~a" (http-request-header 'host) (http-request-url))
      port)))
```

It should be noted that RDF URLs coming in with fragments in them must be encoded. That is, the # character in the URL must be encoded using the %23 string, or else the URL will come through without the fragment. This appears to be an artifact of `mod_lisp`, but I have not confirmed this yet.

5. Accepting Submissions

When handling Submissions, we will grab the post parameters and check if they are null. If it isn't, then we have a submission and we can go ahead and write out the library. Otherwise, we'll just display the main submit landing page.

```
<Submission Handler>≡
(define-subtree-handler descot-submit-path 'application/xhtml+xml
  (lambda (path port)
    (let ([params (http-request-post-parameter-bindings)])
      (if (or (not params) (null? params))
          (write-html-page submit-page port)
          (begin
             (http-response-header 'content-type "text/plain")
             (format port "Would write to: ~a~%"
                       (string-append
                        descot-submit-store
                        (string (directory-separator))
                        (format-gensym (gensym "submission"))))
             (format port "Submitted by ~a on ~a.~%"
                       (http-request-header 'remote-ip-addr)
                       (date-and-time))
             (newline port)
             (pretty-print params port))))))
```

6. Optimization/Initialization/Running

DWA is configured with standard unsafe optimizations, and the entire program is run from one nested LET expression. All definitions must come before the handlers, since the handlers are actually expressions rather than definitions.

```
<web-app.ss>≡  
<Web App License>  
  
(eval-when (compile)  
  (generate-inspector-information #f)  
  (optimize-level 3))  
  
<Shared Objects>  
  
(let ()  
  
<Imports>  
  
<Pages>  
  
<Handlers>  
  
<Initialization>  
  
)
```

To initialize the program, we first make sure that we are using XHTML generators [Ed. – This is a Chez Libs specific adjustment to Oleg’s SXML code to allow the correct ending tags to work].

```
<Initialization>≡  
;; We are using XML  
(xml-tags? #t)
```

We need to make sure that we initialize the server api the way we want. The default SRDF format will be used for the store, but we specify the store location from our parameters:

```
<Initialization>+≡  
(descot-store descot-store-root)
```

At the moment, we read in the entire store into memory for simplicity. this makes our accessor functions much easier to write, so we assume this. However, this could have ramifications in the future if the size of the stores grows too large. Instead of doing this, it would be better if the stores were grabbed directly from the filesystem when they were needed, assuming we have a heavy memory constraint. On the other hand, I have an inkling thought that this won’t matter on today’s high memory systems. At least, I hope not.

```
<Initialization>+≡  
(current-store (read-descot-store (descot-store)))
```

Now we can safely start the mod_lisp server.

```
<Initialization>+≡  
(start-mod-lisp-server)
```

Right now, the server is meant to be launched using something like `nohup(1)` with backgrounding so that the process can be thrown into the background and it will run with output to some file.

7. Imports/Dependencies

DWA requires a few libraries to work, including normal Chez Scheme features, plus SRFIs 14 and 13, MIT Scheme compatible `mod-lisp` interface, Descot's RDF utilities and tools, Taylor Campbell's RDF libraries, foof loop by the same, Alex Shinn's Irregular Expressions, and Oleg's SXML transformers.

```
<Imports>≡
(include "lib/sockets.ss")
(include "lib/assert.ss")
(include "lib/syn-param.ss")
(include "lib/srfi-8.ss")
(include "lib/foof-loop.ss")
(include "lib/nested-foof-loop.ss")
(include "lib/io-util.ss")
(include "lib/srfi-45.ss")
(include "lib/stream.ss")
(include "lib/perror.ss")
(include "lib/let-opt.ss")
(include "lib/check-arg.ss")
(include "lib/srfi-1.ss")
(include "lib/matcomb.ss")
(include "lib/srfi-14.ss")
(include "lib/matttext.ss")
(include "lib/parcomb.ss")
(include "lib/partext.ss")
(include "lib/char-utils.ss")
(include "lib/srfi-13.ss")
(include "lib/srfi-23.ss")
(include "lib/uri.ss")
(include "lib/irregex.ss")
(include "lib/myenv-chez.ss")
(include "lib/oleg-util.ss")
(include "lib/oleg-sxml-tree-trans.ss")
(include "lib/oleg-sxml-to-html.ss")
(include "lib/mod-lisp.ss")
(include "lib/rdf.ss")
(include "lib/rdf-list-graph.ss")
(include "lib/rdf-turtle-parser.ss")
(include "lib/rdf-map.ss")
(include "rdf-util.ss")
(include "rdf2html.ss")
(include "web-param.ss")
(include "web-util.ss")
(include "web-gen.ss")
(include "srdf.ss")
(include "server.ss")

(import scheme)
(import mod-lisp)
(import sxml-to-html)
(import rdf2html)
```

```

(import descot-rdf-utilities)
(import rdf-list-graphs)
(import foof-loop)
(import nested-foof-loop)
(import rdf)
(import srfi-14)
(import srfi-13)
(import irregular-expressions)
(import descot-web-parameters)
(import descot-web-generators)
(import descot-web-utilities)
(import oleg-util)
(import descot-server)

```

this also means that DWA requires a few shared objects to be loaded into Chez Scheme in order to be used successfully.

```

⟨Shared Objects⟩≡
(load-shared-object "libc.so")
(load-shared-object "./lib/sockets.so.1.0")

```

8. Licensing

Descot's Web Application is licensed under an ISC License.

```

⟨Web App License⟩≡
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Web Application Component of Descot
;;;
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>
;;;
;;; Permission to use, copy, modify, and distribute this software for
;;; any purpose with or without fee is hereby granted, provided that the
;;; above copyright notice and this permission notice appear in all
;;; copies.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
;;; PERFORMANCE OF THIS SOFTWARE.

```

Chapter 4

Printing RDF Graphs

`rdf-printing` is a module for printing rdf graphs and triples. It outputs them in simplistic Turtle based format, but does not use nearly the features that `turtle` has. It is much easier to just print a single triple per line, without any attempts at prefixing or other Turtle syntaxes. That is, these procedures were designed to output computer parsible RDF Graphs in Turtle format, without any consideration to human readability. Take note!

Each procedure takes the datum to print, and optionally a port to which it should print. If no port is given, then the `current-output-port` is used.

```
<rdf-print.ss>≡  
<License>
```

```
<Module Form>
```

1. Exports and Name

```
<Module Form>≡  
(module rdf-printing  
  (write-rdf-triple->turtle write-rdf-graph->turtle write-rdf-triples->turtle)
```

```
<Imports>
```

```
<Output Procedures>
```

```
)
```

2. Dependencies

`rdf-printing` requires Taylor Campbell's RDF Graphs and foof loop libraries.

```
<Imports>+≡  
(import scheme)  
(import rdf-list-graphs)  
(import rdf)  
(import foof-loop)  
(import nested-foof-loop)
```

3. Output Procedures

```
<Output Procedures>≡  
<RDFs to Strings>
```

```
<Printing Triples>
```

```
<Printing Triple Lists>
```

```
<Printing RDF Graphs>
```

The three main output procedures do essentially the same thing on three different data structures: `write-rdf-triple->turtle`, `write-rdf-graph->turtle`, and `write-rdf-triples->turtle`. They are supported by a means of converting an `rdf-node` into its appropriate textual representation in turtle format called `rdf-node->string`.

```

<RDFs to Strings>≡
(define rdf-node->string
  (lambda (node)
    (cond
      [(rdf-uri-ref? node) (rdf-uri-ref->string node)]
      [(rdf-plain-literal? node)
       (format #f "~a~@[~a~]"
               (rdf-literal/lexical-form node)
               (rdf-plain-literal/language-tag node))]
      [(rdf-bnode? node) (string-append "_:" (rdf-bnode/name node))]
      [(rdf-typed-literal? node)
       (format #f "~a^^~a"
               (rdf-literal/lexical-form node)
               (rdf-typed-literal/datatype-uri node))]
      [else (error 'rdf-node->string "Bad node ~a" node)])))

```

When printing a triple, Turtle notation just has whitespace between the Subject, Predicate, and Object, with a single period at the end.

```

<Printing Triples>≡
(define write-rdf-triple->turtle
  (case-lambda
    [(triple)
     (%write-rdf-triple triple (current-output-port))]
    [(triple port)
     (%write-rdf-triple triple port)]))

(define %write-rdf-triple
  (lambda (triple port)
    (format port "~a ~a ~a.~n"
            (rdf-node->string (rdf-triple/subject triple))
            (rdf-node->string (rdf-triple/predicate triple))
            (rdf-node->string (rdf-triple/object triple)))))

```

Printing a triple list is a trivial iteration over the triples in a list.

```

<Printing Triple Lists>≡
(define write-rdf-triples->turtle
  (case-lambda
    [(triple)
     (%write-rdf-triple-list triple (current-output-port))]
    [(triple port)
     (%write-rdf-triple-list triple port)]))

(define %write-rdf-triple-list
  (lambda (triple-list port)
    (for-each (lambda (e) (write-rdf-triple e port)) triple-list)))

```

To print a graph, we should simply use the `rdf-graph/for-each-triple` function made available to use from Taylor Campbell's RDF Graphs.

```
<Printing RDF Graphs>≡
(define write-rdf-graph->turtle
  (case-lambda
    [(graph)
     (%write-rdf-graph graph (current-output-port))]
    [(graph port)
     (%write-rdf-graph graph port)]))

(define %write-rdf-graph
  (lambda (graph port)
    (rdf-graph/for-each-triple graph
      (lambda (e) (write-rdf-triple e port))))))
```

4. Licensing

`rdf-printing` is licensed under an ISC License.

```
<License>+≡
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Output RDF Graphs to Turtle
;;;
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>
;;;
;;; Permission to use, copy, modify, and distribute this software for
;;; any purpose with or without fee is hereby granted, provided that the
;;; above copyright notice and this permission notice appear in all
;;; copies.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
;;; PERFORMANCE OF THIS SOFTWARE.
```

Chapter 5

Generating Web Sites with Descot

Descot web pages are generated with the help of two procedures: `write-html-page` and `descot-wrapper`. The first makes sure that pages can be easily printed in `mod_lisp` by standardizing the interface to using generators and a port. The second performs the wrapping of all pages in the Descot headers and footers.

```
<web-gen.ss>≡
<License>

(module descot-web-generators
  (write-html-page descot-wrapper)
  (import scheme)
  (import descot-web-parameters)
  (import descot-web-utilities)
  (import sxml-to-html)

<Writing>

<Wrapping>

(xml-tags? #t)

)
```

Note that Descot web pages are written in XHTML, so we have to use the Chez Lib extension on Oleg's SXML, `xml-tags?` parameter, in order to ensure the right output.

1. Writing Pages

Descot uses `mod_lisp` which normally calls a procedure with a path and port. Normally, we may have a page generator that generates the SXML that we wish to print, but in order to print it, we use `write-html-page` to convert the SXML to XHTML and format it according to normal W3C standards, all sent to the given port.

```
<Writing>≡
(define write-html-page
  (lambda (generator port)
    (display xhtml-decl port)
    (display
      (with-output-to-string (lambda () (sxml->html (generator))))
      port)))
```

2. Wrapping Pages

Every Descot page is wrapped in a header and a footer that contains the navigation and search. `descot-wrapper` allows this to be done easily, and also provides a little flexibility in the way of style sheets and subheadings for each page.

```
<Wrapping>≡
(define descot-wrapper
  (lambda (page-name css . body-elems)
    (html
      (head
        (title (string-append descot-title " -- " page-name))
        (link css "stylesheet" "text/css" "screen"))
      (body
        (div '(@ (id "beta")) descot-beta-msg)
        (h1 descot-title)
        (div '(@ (id "search"))
          (form descot-search-path "get"
            (p
              (input '(name "q") '(type "text"))
              (button '(@ (type "submit")) "Search"))))
          (if (zero? (string-length page-name)) "" (h2 page-name))
        body-elems
        (ul '(@ (id "menu"))
          (li (a '(@ (href ,descot-about-path)) "Info"))
          (li (a '(@ (href ,descot-blog-path)) "Blog"))
          (li (a '(@ (href ,descot-browse-path)) "Browse"))
          (li (a '(@ (href ,descot-search-path)) "Advanced Search"))
          (li (a '(@ (href ,descot-submit-path)) "Submit a Library")))
        (div '(@ (id "copyright"))
          (p
            "Copyright "
            '|&copy;|
            " 2009 Aaron Hsu. All rights reserved."))))))
```

3. Licensing

This file is licensed under the ISC License.

```
<License)+≡  
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  
;;; Generate Static Web Pages for Descot Web Application  
;;;  
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>  
;;;  
;;; Permission to use, copy, modify, and distribute this software for  
;;; any purpose with or without fee is hereby granted, provided that the  
;;; above copyright notice and this permission notice appear in all  
;;; copies.  
;;;  
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL  
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED  
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE  
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL  
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA  
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER  
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR  
;;; PERFORMANCE OF THIS SOFTWARE.
```

Chapter 6

S-expression RDF Format

1. Informative Description

SRDF is an s-expression based format for describing RDF Graphs. It is meant to be mostly equivalent in its form to Turtle. Since the language is S-expression based, it is easier for Scheme and Lisp parsers to parse it. Parsers for other languages can also be written very easily. This makes it particularly nice for use in automated systems or in areas where S-expressions are the natural representation format.

As a shortcut in this discussion, strings, booleans, numbers, and symbols are all assumed to be Scheme values of the same. Specifically, we assume that the reader will be able to read the special symbols `&`, `^`, `=`, `_`, and `*`, as symbols directly from the input.

SRDF forms are composed of a series of RDF triples and possibly, prefix definitions. Prefixes take the form `(= name "uri")`, and associate a given Scheme symbol with a URI string. Otherwise, the form is an RDF triple or a set of triples.

Normal triples are just a list of three elements, each a URI. However, it is possible to have a subject associated with more predicate and objects by replacing the list that would hold the single predicate and object with a list of such predicates and objects. Likewise, one can specify more objects to be associated with a given subject and predicate by doing the same thing with the object list, and replacing the `cdr` that would normally hold the object with a list of such objects.

If the `cadr` of a predicate pair contains a list of objects, this represents a collection of objects, and is created in the same way that a turtle collection syntax is created: by associating a series of blank nodes with the right predicates with each of the objects listed.

It is important to note the difference between an object that is a collection, and a list of objects that are each associated with the subject and predicate. The following is an instance of the former:

```
("subject-uri" "predicat-uri"  
 ("object1" "object2" ...))
```

Whereas the following is an instance of the latter:

```
("subject-uri" "predicate-uri"  
 "object1"  
 "object2"  
 "object3")
```

Normal RDF triples take the form:

```
("subject-uri" "predicate-uri" "object-uri")
```

A blank node may be inlined into the graph by using a `'*` as the begining symbol in an object context like so:

```
("subject" "pred" (* "pred" "object"))
```

Of course, blank nodes may have anything that is a valid predicate `cdr` as its `cdr`, so the following is also valid:

```
("subject" "pred" (* ("pred1" "object1") ("pred2" "object2")))
```

URIs may be described by their full path names as strings, as prefix combined paths, or as blank node paths. The following are all valid URIs:

```
"http://some.domain/path/to#blah"  
"blah"  
(: prefix "blah")  
(_ "uniqueid")
```

We use ‘.’ for prefixes and ‘_’ for blank nodes.

In addition to URIs, we also allow literals as valid `cars` for objects. Literals can be strings, numbers, booleans, or may be strings with either languages or types associated with them. The following are examples of languages and types, respectively:

```
($ "Language unspecified.")
(& "English Sentence lies here." en)
(^ "2008/01/03 14:00" (: xsd "date"))
```

2. Formal BNF Grammar

The following is a fairly formal BNF grammar with the exception of tokens such as strings, numbers, and booleans being undefined and presumed to be defined lexical values. Additionally, we define S-expression in terms of atoms and pairs, so the BNF grammar is also defined in the “longhand” notation for pairs and lists. This means that while the BNF Grammar states something like (“`subj`” . (“`pred`” . (“`obj`” . ()))) as the valid simplistic RDF triple, it is also legal in practice to use the shorthand version of this: (“`subj`” “`pred`” “`obj`”).

```
⟨rdf sexp⟩→ ⟨rdf triple⟩ | ⟨rdf triple⟩ ⟨rdf sexp⟩
⟨rdf triple⟩→ ‘(’ ⟨uri⟩ ‘.’ ⟨rdf subject tail⟩ ‘)’ | ‘(’ ‘=’ name string ‘)’
⟨rdf subject tail⟩→ ⟨rdf predicate⟩ | ⟨rdf predicate list⟩
⟨rdf predicate list⟩→ ‘(’ | ‘(’ ⟨rdf predicate⟩ ‘.’ ⟨rdf predicate list⟩ ‘)’
⟨rdf predicate⟩→ ‘(’ ⟨uri⟩ ‘.’ ⟨rdf object list⟩ ‘)’
⟨rdf object list⟩→ ‘(’ | ‘(’ ⟨rdf object⟩ ‘.’ ⟨rdf object list⟩ ‘)’
⟨rdf object⟩→ ⟨uri⟩ | ⟨literal⟩ | ⟨rdf object list⟩ | ⟨blank node list⟩
⟨blank node list⟩→ ‘(’ ‘*’ ‘.’ ⟨rdf subject tail⟩ ‘)’
⟨uri⟩→ uri | ‘(’ ‘:’ ‘.’ ⟨uri list⟩ ‘)’ | ‘(’ ‘_’ name ‘)’
⟨uri list⟩→ ‘(’ | ‘(’ ⟨uri name⟩ ‘.’ ⟨uri list⟩ ‘)’
⟨uri name⟩→ uri | name
⟨literal⟩→ ‘(’ ‘$’ string ‘)’ | number | boolean | ‘(’ ‘&’ string name ‘)’ | ‘(’ ‘^’ string ⟨uri⟩ ‘)’
```

3. Parsing

```
⟨Exports⟩≡
parse-srdf-file
parse-srdf-form
```

Parsing an SRDF form requires that an RDF Graph be in place, together with some set of prefixes about which the parser knows. Since the RDF Graph libraries available at the time¹ all use a table/dictionary basis, and so are mutation based, we will also make our prefix form mutation based. We expect very few prefixes, and thus, a simple association list makes more sense than using hashtables.

```

<Parsing Types>≡
(define-record prefix-table (entries) ()
  ((constructor %make-prefix-table)))

(define prefix-table-set!
  (lambda (tbl key value)
    (assert tbl prefix-table?)
    (let ([res (assq key (prefix-table-entries tbl))])
      (if res
          (set-cdr! res value)
          (set-prefix-table-entries! tbl
            (cons (cons key value)
                  (prefix-table-entries tbl)))))))

(define prefix-table-ref
  (lambda (tbl key default)
    (assert tbl prefix-table?)
    (let ([res (assq key (prefix-table-entries tbl))])
      (if res (cdr res) default))))

(define make-prefix-table (lambda () (%make-prefix-table '())))

```

These forms require the use of `assert` so we import this:

```

<Imports>+≡
(import scheme)
(import assertions)

```

Given a Graph to work on and a set of Prefixes to use, then we can parse every SRDF set form by form, from the start to the end. This makes it easy to parse from a file directly, because we don't need to do any backtracking ourselves, and instead, we can rely on Scheme's built-in `read` to do most of the work for us.

```

<File Parsing>≡
(define parse-srdf-file
  (lambda (fname . maybe-graph)
    (let ([graph (if (null? maybe-graph) (make-rdf-graph) (car maybe-graph))]
          [prefixes (make-prefix-table)])
      (iterate! (for srdf-form (in-file fname read))
                (parse-srdf-form graph prefixes srdf-form)
                graph)))

```

¹ RDF List Graphs and Maps by Taylor Campbell

Of course, there is a problem with this. If there is an error in the parsing, we will not know where it is. There really is not anything we are going to do about this right now.

The above form uses nested foof looping, so we need to make sure that is an import; we also use RDF List Graphs to represent the RDF information for the moment, though we may have to move to something more efficient in the future:

```

<Imports>+≡
(import foof-loop)
(import nested-foof-loop)
(import rdf-list-graphs)

```

For each SRDF form, we will do a basic hard coding handle of each design, since this is easier than trying to abstract this on top of another parsing framework. We define `parse-rdf-form` as a generic procedure for handling all SRDF forms.

$$\text{parse-srdf-form} : \langle \text{graph} \rangle \langle \text{prefixes} \rangle \langle \text{form} \rangle \rightarrow \# \langle \text{void} \rangle$$

Where *graph* and *prefixes* will be updated with the information stored in *form*.

```

<Generic Form Parsing>≡
(define parse-srdf-form
  (lambda (graph prefixes form)
    (case (car form)
      [(=) (prefix-table-set! prefixes (cadr form) (caddr form))]
      [else (parse-srdf-triple graph prefixes form)])))

```

An SRDF Triple is composed of a URI component and an RDF Subject Tail. These are parsed by `parse-srdf-uri` and `parse-srdf-subj-tail`, respectively. Parsing a triple of this nature is sort of like parsing a Scheme `let` form in that the URI will then be the “scoping” for all of the tail. The signature of `parse-srdf-triple` is similar to `parse-srdf-form`.

$$\text{parse-srdf-triple} : \langle \text{graph} \rangle \langle \text{prefixes} \rangle \langle \text{form} \rangle \rightarrow \# \langle \text{void} \rangle$$

```

<Triple Parsing>≡
(define parse-srdf-triple
  (lambda (graph prefixes form)
    (parse-srdf-subj-tail graph prefixes
      (parse-srdf-uri prefixes (car form))
      (cdr form))))

```

Parsing the RDF Subject Tail requires checking ahead a little bit to see whether the first element of the subject tail is a URI or not. If it is, then we can safely assume that we are dealing with a single subject tail with only one predicate, but otherwise, we have to assume that we are dealing with a predicate list. We are given the right URI to use as the subject.

$$\text{parse-srdf-subj-tail} : \langle \text{graph} \rangle \langle \text{prefixes} \rangle \langle \text{subject uri} \rangle \langle \text{subj tail form} \rangle \rightarrow \# \langle \text{void} \rangle$$

```

<Subject Tail Parsing>≡
(define parse-srdf-subj-tail
  (lambda (graph prefixes subject form)
    (if (valid-srdf-uri? (car form))
        (parse-srdf-predicate graph prefixes subject form)
        (for-each
          (lambda (t)
            (parse-srdf-predicate graph prefixes subject t))
          form))))

```

Parsing an SRDF Predicate is just like parsing a normal triple in form, except that we receive the subject. We grab the predicate from the `car` and parse the tail, which we will handle the same way we handle the subject tail parsing.

```
parse-srdf-predicate: <graph> <prefixes> <subject-uri> <predicate-form> → #<void>
```

<Predicate Parsing>≡

```
(define parse-srdf-predicate
  (lambda (graph prefixes subject form)
    (let ([predicate (parse-srdf-uri prefixes (car form))])
      (iterate! (for object (in-list (cdr form)))
                (parse-srdf-object graph prefixes subject predicate object)))))

(define parse-srdf-object
  (lambda (graph prefixes subject predicate object)
    (cond
      [(valid-srdf-uri? object)
       (rdf-graph/add-triple! graph
                              (make-rdf-triple subject predicate
                                                (parse-srdf-uri prefixes object)))]
      [(valid-srdf-literal? object)
       (rdf-graph/add-triple! graph
                              (make-rdf-triple subject predicate
                                                (parse-srdf-literal prefixes object)))]
      [(valid-blank-node-list? object)
       (parse-srdf-subj-tail graph prefixes (generate-bnode) (cdr object))]
      [(valid-srdf-object-list? object)
       (parse-srdf-obj-collection graph prefixes subject predicate object)]
      [else (error 'parse-srdf-object "Invalid Object: ~s" object)])))
```

3.1. Parsing Collections

RDF collects are essentially linked lists where you have a `nil` object, a predicate for the `cdr`, called `rest`, and a predicate for the `car`, called `first`. SRDF collections are objects which are object lists.

Firstly, we must define the three URIs used with Collections:

<RDF URIs>≡

```
(define prefix-with-rdf
  (lambda (tail)
    (string-append "http://www.w3.org/1999/02/22-rdf-syntax-ns#" tail)))
(define rdf:nil (prefix-with-rdf "nil"))
(define rdf:rest (prefix-with-rdf "rest"))
(define rdf:first (prefix-with-rdf "first"))
(define rdf:List (prefix-with-rdf "List"))
(define rdf:type (prefix-with-rdf "type"))
```

After this, we will loop over every object in the collection, creating a series of blank nodes to match with the collection.

```

⟨Collection Parsing Procedures⟩≡
(define parse-srdf-obj-collection
  (lambda (graph prefixes subject predicate obj-list)
    (let ([fbn (generate-bnode)])
      (rdf-graph/add-triple! graph
        (make-rdf-triple subject predicate fbn))
      (loop ([for object rest (in-list obj-list)]
            [with obn fbn bn]
            [let bn (if (null? (cdr rest)) rdf:nil (generate-bnode))])
        (rdf-graph/add-triple! graph (make-rdf-triple obn rdf:type rdf:List))
        (parse-srdf-object graph prefixes obn rdf:first object)
        (rdf-graph/add-triple! graph (make-rdf-triple obn rdf:rest bn))))))

```

```

⟨Collection Parsing⟩≡
⟨RDF URIs⟩

```

```

⟨Collection Parsing Procedures⟩

```

3.2. Parsing URIs and Literals

Parsing SRDF URIs does not require anything more than the prefix, and `parse-srdf-uri` should return a standard URI².

$$\text{parse-srdf-uri} : \langle \text{prefixes} \rangle \langle \text{uri form} \rangle \rightarrow \langle \text{uri} \rangle$$

```

⟨URI Parsing⟩≡
(define parse-srdf-uri
  (lambda (prefixes uri)
    (if (pair? uri)
        (case (car uri)
          [(:) (parse-srdf-uri-list prefixes (cdr uri))]
          [(_) (make-rdf-bnode (symbol->string (cadr uri)))]
          [else (error 'parse-srdf-uri "Invalid URI: ~a" uri)])
        uri)))

(define parse-srdf-uri-list
  (lambda (prefixes uri-list)
    (apply string-append
      (collect-list (for e (in-list uri-list))
        (if (string? e) e
            (or (prefix-table-ref prefixes e #f)
                (error 'parse-srdf-uri "Invalid URI Tail: ~s" uri-list)))))))

```

The above requires the use of nested foof loop, as well as Taylor Campbell's RDF Library. We should add the RDF module since this is not already added:

```

⟨Imports⟩+≡
(import rdf)

```

² We use Taylor Campbell's RDF URI implementation

Parsing a literal is a matter of checking its tag if it has one, or letting it through if it's a value such as a number or boolean.

```
<Literal Parsing>≡
(define parse-srdf-literal
  (lambda (prefixes val)
    (cond
      [(number? val) (make-rdf-plain-literal (number->string val) #f)]
      [(boolean? val) (make-rdf-plain-literal (if val "true" "false") #f)]
      [(pair? val)
       (case (car val)
         [($) (make-rdf-plain-literal (cadr val) #f)]
         [(@) (make-rdf-plain-literal (cadr val) (symbol->string (caddr val)))]
         [(^)
          (make-rdf-typed-literal (cadr val)
            (parse-srdf-uri prefixes (caddr val)))]
         [else (error 'parse-srdf-literal "Bad Tagged Literal: ~s" val)]]]
      [else (error 'parse-srdf-literal "Unknown Literal: ~s" val)])))
```

3.3. Blank Nodes

We have to have a way of generating blank nodes that are going to be unique according to some metric of uniqueness. In order to do this, we set up a parameter to hold a reasonable value from which to start, and then use `current-time` in order to get something pretty unique.

```
<Generating Blank Nodes>≡
(define bnode-seed (make-parameter 1242083868))

(define generate-bnode
  (lambda ()
    (let ([ct (current-time)])
      (make-rdf-bnode
        (string-append (symbol->string (gensym))
          (number->string (- (time-second ct) (bnode-seed)))
          "."
          (number->string (time-nanosecond ct)))))))
```

3.4. Validating elements

We may want to validate a few different elements of an SRDF, which we deal with here.

```
<Validating>≡
<Validating URIs>
```

```
<Validating Literals>
```

```
<Validating Blank Node Lists>
```

```
<Validating Object Lists>
```

If we want to validate URIs, we can use the following, which checks to make sure that we either have the right prefix or we have a string.

```
<Validating URIs>≡
(define valid-srdf-uri?
  (lambda (uri)
    (or (string? uri)
        (and (pair? uri) (memq (car uri) '(: _) #t))))
```

We can handle Literals by checking for their head tags or their other types.

```
<Validating Literals>≡
(define valid-srdf-literal?
  (lambda (obj)
    (or (number? obj) (boolean? obj)
        (and (pair? obj)
              (memq (car obj) '($ @ ^)
                    #t))))
```

A blank node list is easy, since it is only a tagged subject tail:

```
<Validating Blank Node Lists>≡
(define valid-blank-node-list?
  (lambda (obj)
    (and (pair? obj) (eq? '* (car obj)))))
```

Validating an Object list is a little more involved, since we need some sense of what an object is, as well as the potential recursion which results.

```
<Validating Object Lists>≡
(define valid-srdf-object?
  (lambda (obj)
    (or
     (valid-srdf-uri? obj)
     (valid-srdf-literal? obj)
     (valid-blank-node-list? obj)
     (valid-srdf-object-list? obj))))

(define valid-srdf-object-list?
  (lambda (obj)
    (and (pair? obj) (andmap valid-srdf-object? obj))))
```

4. Writing

```
<Exports>+≡
write-rdf-triple->srdf
write-rdf-graph->srdf
write-rdf-triples->srdf
```

A `write-srdf-triples` procedure is provided to make it easy to print out graphs to SRDF format. The format is the simplest form of SRDF triples, so it is not easy to read, but it does the job.

Firstly, one must know how to construct a string representation of a given node:

```

<Writing>+≡
(define render-rdf-node
  (lambda (node)
    (cond
      [(rdf-uri-ref? node) (rdf-uri-ref->string node)]
      [(rdf-plain-literal? node)
       (let ([lang-tag (rdf-plain-literal/language-tag node)])
         (if lang-tag
             '(@ ,(rdf-literal/lexical-form node) ,(string->symbol lang-tag))
             '($ ,(rdf-literal/lexical-form node))))])
      [(rdf-bnode? node) '(_ ,(string->symbol (rdf-bnode/name node)))]
      [(rdf-typed-literal? node)
       '(^ ,(rdf-literal/lexical-form node)
           ,(rdf-typed-literal/datatype-uri node))]
      [else (error 'render-rdf-node "Bad node ~a" node)])))

```

In order to then write a triple, we simply apply this to all of our pieces in our triple.

```

<Writing>+≡
(define write-rdf-triple->srdf
  (case-lambda
    [(triple)
     (%write-rdf-triple triple (current-output-port))]
    [(triple port)
     (%write-rdf-triple triple port)]))

(define %write-rdf-triple
  (lambda (triple port)
    (format port "~s~n"
      '(,(render-rdf-node (rdf-triple/subject triple))
        ,(render-rdf-node (rdf-triple/predicate triple))
        ,(render-rdf-node (rdf-triple/object triple))))))

```

We can then apply this over lists of triples:

```

<Writing>+≡
(define write-rdf-triples->srdf
  (case-lambda
    [(triple)
     (%write-rdf-triple-list triple (current-output-port))]
    [(triple port)
     (%write-rdf-triple-list triple port)]))

(define %write-rdf-triple-list
  (lambda (triple-list port)
    (for-each (lambda (e) (write-rdf-triple->srdf e port)) triple-list)))

```

We use `rdf-graph/for-each-triple` to go through all of the nodes in a graph:

```
<Writing>+≡
(define write-rdf-graph->srdf
  (case-lambda
    [(graph)
     (%write-rdf-graph graph (current-output-port))]
    [(graph port)
     (%write-rdf-graph graph port)]))

(define %write-rdf-graph
  (lambda (graph port)
    (rdf-graph/for-each-triple graph
      (lambda (e) (write-rdf-triple->srdf e port)))))
```

5. Module Form

The Chez Scheme module form exports `parse-srdf-form` and `parse-srdf-file` under the module name `srdf-parsing`.

```
<srdf.ss>≡
<License>

(module srdf
  (<Exports>)
  (<Imports>

<Parsing Types>

<URI Parsing>

<Literal Parsing>

<File Parsing>

<Generic Form Parsing>

<Triple Parsing>

<Subject Tail Parsing>

<Predicate Parsing>

<Collection Parsing>

<Validating>

<Generating Blank Nodes>

<Writing>
)
```

6. License

This SRDF implementation is licensed under the ISC License to Aaron Hsu.

```
<License>+≡
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; S-expression RDF Format
;;;
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>
;;;
;;; Permission to use, copy, modify, and distribute this software for
;;; any purpose with or without fee is hereby granted, provided that the
;;; above copyright notice and this permission notice appear in all
;;; copies.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
;;; PERFORMANCE OF THIS SOFTWARE.
```

Chapter 7

Descot Server API

1. Overview

The Descot Server API provides a ready-made API for storing Descot RDF graphs and accessing them in the basic manners required for the Descot Server Protocol. That is, it is a library upon which Descot Servers can be built, such as a Web Application or a GUI. It has the following module declaration:

```
<server.ss>≡  
<License>  
  
(module descot-server  
  (<Exports>)  
  (<Imports>)  
  
  <URI Resolution>  
  
  <Parameters>  
  
  <Requests>  
  
  <Updates>  
  
  <Submitting>  
  
  <Whole Store>  
  
)
```

2. The Descot Server Protocol

```
<Imports>+≡  
(import scheme)  
(import rdf-list-graphs)  
(import rdf)
```

The following describes the Descot Server Protocol and its implementation requirements. It does not implement the actual server itself, but provides the necessary background information to do so. This is because the protocol may be communicated over many different underlying protocols. For example, this system could be implemented using HTTP or Gopher, and either one would work fine.

2.1. Handling Incoming Requests

```
<Exports>+≡  
write-descot-request
```

A Descot Server has two required functions. It must firstly respond to requests for subjects and provide the necessary pieces. This API provides the means to output rdf subject requests according to the requirements of the Descot Protocol, which is that all the objects who have the given subject as their own are printed out, and every object of the given subject that is a blank node is also printed out, and so forth, recursively, until all the blank nodes having the given subject as an ancestor are written out as well.

write-descot-request : $\langle \text{subject-uri} \rangle \langle \text{port} \rangle \rightarrow \# \langle \text{void} \rangle$

```

<Requests>≡
(define write-rdf-subject+blanks
  (lambda (graph subject port)
    (let ([triples (rdf-graph/triples-by-subject graph subject)])
      ((descot-api-triples-writer) triples port)
      (for-each
        (lambda (triple)
          (when (rdf-bnode? (rdf-triple/object triple))
            (write-rdf-subject+blanks graph (rdf-triple/object triple) port))))
        triples))))

(define write-descot-request
  (lambda (subject . maybe-port)
    (write-rdf-subject+blanks
      ((descot-api-reader) (descot-uri->store-path subject))
      subject
      (if (null? maybe-port) (current-output-port) (car maybe-port)))))

```

We separate `write-rdf-subject+blanks` because this is an useful procedure that is used later on when we need to write this information to the filesystem.

Important Note: The astute reader may question the currently unnecessary overhead of parsing the file into an RDF Graph, and then reading it back out into the same format. We could have just read the file directory or something similar. Indeed, if we assume that the output to the user is the same as the format of the RDF Store, this is the best thing to do, and it removes a lot of complex handling here. However, this is not the case. A request may come in for the RDF to be given in any number of formats, and the server must be able to provide the Graph in the format that the client wishes to the best of its abilities. This means that we need to have this extra layer of processing.

Right now, the format detection is not implemented, but it should be. Adjusting the above to handle this shouldn't be difficult.

2.2. Server Synchronization

```

<Exports>+≡
write-descot-updates

```

The server also provides the means of printing out the necessary update document required by a Descot Server. This allows servers to easily update each other about their new changes without using much bandwidth. When sending an update, only the libraries themselves that have modified times will be sent, and only the triples with a modified node as a predicate will be written.

```
write-descot-updates : <port> → #<void>
```

```
<Updates>≡
(define write-descot-updates
  (lambda (port)
    ((descot-api-triples-writer)
     (rdf-graph/triples-by-predicate (current-store) dscts:modified)
     port)))
```

This requires the Descot RDF Utilities module and the `current-store` parameter:

```
<Imports>+≡
(import descot-rdf-utilities)
(import descot-web-parameters)
```

2.3. Submissions

```
<Exports>+≡
write-descot-store
```

While Descot Servers are not strictly required to provide submission interfaces, a standard one is provided here to allow safe interaction between the reading and writing part of the server. The functions defined in this API will play safely with one another so that the Descot Store will not be corrupted by the simultaneous use of the read and write functions (or any combination of either) presuming that this library is the only library used to access the store. Other libraries may access it provided that they behave according to the same rules.

When submitting a library, to incorporate it into the system, the provided graph will be considered, and all the files in the store will be updated according to the new information. The standard layout for a Descot Store is provided below.

Important Note: We are assuming that the graph we receive is the complete package, and so for any single node, we will erase it and completely replace it with the new one!

```
write-descot-store : <graph> → #<void>
```

```
<Submitting>+≡
(define write-descot-store
  (lambda (graph)
    (iterate! (for subject (in-list (rdf-graph/all-subjects graph)))
              (if (rdf-uri-ref? subject)
                  (let ([path (descot-uri->store-path subject)])
                    (guarantee-directory-exists (path-parent path))
                    (call-with-output-file (descot-uri->store-path subject)
                      (lambda (port) (write-rdf-subject+blanks graph subject port))
                      'replace))))))
```

The above code uses nested foof loop to loop over all the subjects in the graph, so make sure that we import this:

```
<Imports>+≡  
(import foof-loop)  
(import nested-foof-loop)
```

We use `guarantee-directory-exists` to make sure that the path works.

```
<Submitting>+≡  
(define guarantee-directory-exists  
  (lambda (path)  
    (unless (file-exists? path)  
      (guarantee-directory-exists (path-parent path))  
      (mkdir path))))
```

3. Descot Layout

```
<Exports>+≡  
descot-uri->store-path
```

This is a relatively simple layout provided for storing Descot RDF Graphs in a persistent manner while providing some level of safety. The store is laid out in a filesystem hierarchy, consisting of a series of subdirectories corresponding to different levels of an RDF URI. A given path in the file system starting at the root of the store will form a representation of a given subject URI, and the file located at said path will contain the necessary information for the printing features of this library to work. That is, it will contain all the triples that have the given URI as a subject or that have a blank node subject which is the object of a triple which has said URI as a subject.

For example, if we consider a sample URL

```
http://descot.sacrideo.us/libs/system/malloc#chez
```

We can find the necessary triples for handling this URI by looking in the graph stored in the file with the path

```
<root>/http/us/sacrideo/descot/libs/system/malloc#chez
```

When accessing one of these files, an exclusive lock is issued if the file will be changed to make sure that the integrity of the file is maintained.

We use a single URI Resolution procedure to convert from a URI we want to reference into its corresponding file system path.

$$\text{descot-uri->store-path} : \langle \text{uri-string} \rangle \rightarrow \langle \text{path-string} \rangle$$

```
<URI Resolution>≡
(define descot-uri->store-path
  (lambda (uri-string)
    (let ([uri (string->uri uri-string)])
      (format #f "~a/~a/~{~a~/~}{~a~/~}~@[#~a~]"
              (descot-store)
              (uri-scheme uri)
              (reverse
               (string-tokenize
                (uri-authority-host (uri-authority uri))
                (char-set-complement (char-set #\.)))))
              (uri-path uri)
              (uri-fragment uri))))))
```

This requires the use of SRFI-13, SRFI-14, and Taylor Campbell's URI Library:

```
<Imports>+≡
(import uri)
(import srfi-14)
(import srfi-13)
```

4. Choosing Different Formats

```
<Exports>+≡
descot-api-triples-writer
descot-api-reader
descot-store
```

Descot Stores may be stored in any arbitrary format, which means that the server must know what format the store uses. To accomplish this, this API provides two parameters to allow the application to specify how to read and write to the store.

Additionally, one may control the format for writing Descot Graphs to the file system by giving it a different handler for printing the lists of triples:

```

<Parameters>≡
(define descot-api-triples-writer
  (make-parameter write-rdf-triples->srdf
    (lambda (e)
      (unless (procedure? e)
        (error 'descot-api-triples-writer
              "Expected a procedure but found: ~s"
              e))
      e)))

```

By default we use the SRDF Triple writer that is provided by the SRDF library:

```

<Imports>+≡
(import srdf)

```

When a request comes in, the server must be able to read the files in the filesystem and parse them. To do this, a parameter containing the reading procedure is provided. This reader will return an RDF Graph that can be used later. It should have the following signature:

$$\text{reader} : \langle \text{fname} \rangle [\langle \text{graph} \rangle] \rightarrow \langle \text{rdf graph} \rangle$$

The above procedure should return an RDF graph that is the union of the graph represented by the file specified by `fname` and the graph provided in the second optional argument. If the optional argument is not specified, it should default to an empty RDF Graph.

```

<Parameters>+≡
(define descot-api-reader
  (make-parameter parse-srdf-file
    (lambda (e)
      (unless (procedure? e)
        (error 'descot-api-reader "Invalid reader value ~s" e))
      e)))

```

5. Store Location

The server also should operate on only one store at a time. For this we define a parameter to contain the root path of the store, without the tailing directory separator. This defaults to `/var/db/descot`:

```

<Parameters>+≡
(define descot-store
  (make-parameter "/var/db/descot"
    (lambda (e)
      (unless (string? e)
        (error 'descot-store "Invalid Path ~s" e))
      e)))

```

6. Getting the Entire Store

$\langle Exports \rangle + \equiv$
read-descot-store

It may be that an application wishes to operate on the entire store rather than parts of it. To do this, we provide a convenience procedure for parsing the entire Descot Store into a single RDF Graph. It will actually work on subsections of the entire store as well.

It works by recursively descending into each directory and parsing every file that it sees into the graph.

$$\text{read-descot-store} : \langle Root \rangle \rightarrow \langle RDF \text{ Graph} \rangle$$

$\langle Whole \text{ Store} \rangle \equiv$

```
(define read-descot-store
  (lambda (root . maybe-graph)
    (let ([graph (if (null? maybe-graph) (make-rdf-graph) (car maybe-graph))])
      (cond
        [(file-regular? root)
         ((descot-api-reader) root graph)]
        [(file-directory? root)
         (loop ([for file (in-list (directory-list root))]
                [with ng graph
                 (read-descot-store
                  (string-append root (string (directory-separator)) file)
                  ng)])
              => ng)]
        [else (error 'read-descot-store "Invalid Descot Store")])]))))
```

7. License

$\langle License \rangle + \equiv$

```
;;;;
;;; Desscot Server API
;;;
;;; Copyright (c) 2009 Aaron Hsu <arcfide@sacrideo.us>
;;;
;;; Permission to use, copy, modify, and distribute this software for
;;; any purpose with or without fee is hereby granted, provided that the
;;; above copyright notice and this permission notice appear in all
;;; copies.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
;;; WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
;;; WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
;;; AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
;;; DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA
;;; OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
;;; TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
;;; PERFORMANCE OF THIS SOFTWARE.
```