Block 3, Week 2
Racket, HTML and Web Applications

Franco Raimondi

16th February 2014

1 Introduction

In this week we are going to build a simple HTTP server using Racket. The web server produces HTML pages and reacts to values being submitted using a browser. Racket generates HTML using *lists*, and therefore we are going to revise operations on lists before moving to the actual server.

1.1 Lists in Racket: a quick revision

This section is taken mainly from `http://www.ccs.neu.edu/home/matthias/HtDP2e/i2-3.html` (if you open this page remember that BSL simply means "Beginner Student Language").
Suppose you want to define the list `("apple" 42 "pear")`: this list has 3 elements and the "pure" Racket definition of it would be something like

```
(cons "apple" (cons 42 (cons "pear" empty)))
```

You have seen at least two other ways of writing it:

```
(list "apple" 42 "pear")
```

and

```
'("apple" 42 "pear")
```

In these two cases the keyword `list` and the single quote are effectively expanded to the `cons` construction above.
There is another way to write a list in a quick way: using a single backward quote (look carefully at the difference between this expression and the one above):

```
`("apple" 42 "pear")
```

We will see in a moment that the single backward quote has interesting features that can be used to solve the following problem: what happens if you evaluate this expression?

```
'(1 (+ 1 1) 3)
```

Try it in DrRacket! Do you obtain the list `(1 2 3)`? No, what you obtain is the list `'(1 (+ 1 1) 3)`: this is a list of three elements, the first element is the number 1, the second element is the *list* `(+ 1 1)` and the third element is the number 3. **THIS IS IMPORTANT**, you need to understand this before proceeding! The second element of `'(1 (+ 1 1) 3)` **IS NOT** the function + applied to two arguments (1 and 1). This second element is *a list of three elements*: the first element is the character "+", the second element is the number 1 and the third element is the number 1. If you don't believe me, try to evaluate:

Exercise 1.1 *Evaluate the following expressions in DrRacket:*

- `(second '(1 (+ 1 1) 3))`
- `(first (second '(1 (+ 1 1) 3)))`

Please ask for additional explanation if this is not clear!

The single backward quote can be used if we want Racket to *evaluate* something in a list built as above.

Exercise 1.2 *Evaluate the following expression in DrRacket:*

- `` `(1 ,(+ 1 1) 3) ``

What happens is the following: when Racket evaluates an expression starting with a single backward quote, it pushes the quote in front of all the elements in the list. In the exercise above, this means `` (`1 `,(+ 1 1) `3) ``. A single (backward) quote in front of a number (or a string) disappears; when a single backward quote meets a comma, they both disappear. This means that the previous expression becomes `(1 (+ 1 1) 3)`: **notice that there is no quote in front of the +!**: this is now a "proper" function, and it is evaluated accordingly.

Exercise 1.3 *Evaluate the following expressions and explain the results:*

```
(define x 42)
'(41 x 43)
`(41 x 43)
`(41 ,x 43)
```

OK so now you should be able to *evaluate* elements in a list. There is an additional complication we need to explore before proceeding to HTML in Racket. The expression `(rest '(1 2 3 4 5))` is evaluated to the list `'(2 3 4 5)`.

Exercise 1.4 *How many element does the list* `` `(1 ,(rest '(1 2 3 4 5))) `` *have? (HINT: try to evaluate* `` (length `(1 ,(rest '(1 2 3 4 5)))) ``

The list in the previous exercise has two elements: the first element is the number 1 and the second element is the list `(2 3 4 5)`. If you want a list with 5 elements you need to *splice* a "nested list into a surrounding list." You can use `cons` (see Block 1), or you can use something called *unquote-splicing*. The shorthand for this operation is the sequence of characters `,@`.

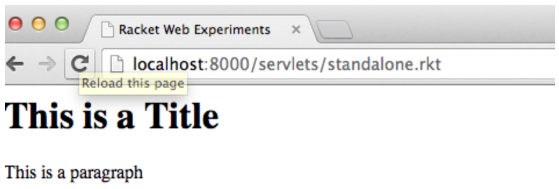
Exercise 1.5 *Evaluate and compare the following expressions:*

- `` `(1 ,(rest '(1 2 3 4 5))) ``
- `` `(1 ,@(rest '(1 2 3 4 5))) ``

Please make sure that you understand the notions of quote, quasiquote (this is the name for the single backward quote) and unquote-splicing before proceeding to the next section!

1.2 HTML: a very short introduction

There is no need for you to spend a lot of time on this section if you are already familiar with HTML and HTML forms, but please make sure that you understand all the concepts below before proceeding to the next section where we generate HTML pages using Racket.



This is a Title

This is a paragraph

Figure 1: A very simple HTML page in a browser

HTML is a *markup language*: you need to provide annotations to specify how to render the document. You have seen something similar with LaTeXand maybe Scribble, so you should not have a problem with HTML. Let's start from a simple example:

```
<html>
  <head>
    <title>Racket Web Experiments</title>
  </head>
  <body>
    <h1>This is a Title</h1>
    <p>This is a paragraph</p>
  </body>
</html>
```

Save this code to a file, for instance test.html, double click on the file and your broswer should open and present a page similar to the one in Figure 1. The code above and the result should be self-explanatory; there are plenty of tutorials available if you are interested in additional details about HTML. For our purposes, we are going to see *HTML forms*. A form is a mechanism to interact with a user through a web page. The user could enter a text in a text field or select some options, and then submit the values to a server. Let's start from a simple example in which a user can enter his/her name:

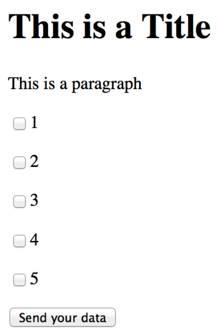
```
<html>
  <head>
    <title>Racket Web Experiments</title>
  </head>
  <body>
    <h1>This is a Title</h1>
    <p>This is a paragraph</p>
   <form>
     <p>Please enter your name:
           <input name="username" />
     </p>
    <p>
        <input type="submit" value="Send your data" />
    </p>
  </form>
  </body>
</html>
```

Again, the code should be very easy to understand: the new tag `<form>` identifies the beginning and the end of a form. Inside a form, we use a tag called `<input>`. This tag needs a `name` and a `type` (if no type is specified, the type is assumed to be a text field). The second `<input>` tag above is of type `submit`, which means that it is a button to "send" the form. The value of the submit input tag specifies what to write on the actual button. Each `input` tag can have a name and in the first case the name of the tag is `username`.

Exercise 1.6 *Copy the code above in a file with extension* `.html` *and open it in a browser. Check that the result is what you expect and that nothing happens when you click the submit button (we will see why in a few paragraphs).*

Checkboxes are among the various kinds of tags that could be used. In the next section we will use checkboxes to specify the PINs of an Arduino board that we want to set to high (to switch on and off LEDs). Again, an example will make things clearer.

```
<html>
  <head>
    <title>Racket Web Experiments</title>
  </head>
  <body>
    <h1>This is a Title</h1>
    <p>This is a paragraph</p>
   <form action="some/url">
      <p><input name="pin" type="checkbox" value="1" />1</p>
      <p><input name="pin" type="checkbox" value="2" />2</p>
      <p><input name="pin" type="checkbox" value="3" />3</p>
      <p><input name="pin" type="checkbox" value="4" />4</p>
      <p><input name="pin" type="checkbox" value="5" />5</p>
    <p>
        <input type="submit" value="Send your data" />
    </p>
  </form>
  </body>
</html>
```


This is a Title

This is a paragraph

Figure 2: A simple HTML form with checkboxes

Notice the new type `checkbox` and notice that all the five checkboxes have the same name: we have created a group of checkboxes. The checkboxes are inside a paragraph, identified with the `<p>` tag. If you check carefully the code above, you'll see that I have slightly modified line 8: I have included an `action` to specify a URL. This is the URL to which the values of the form will be sent when you press the submit button.

Exercise 1.7 *Save the code above in a* `.html` *file and make sure that the result looks similar to the one in Figure 2 when you open the file in a browser.*

If you are confident with HTML forms and Racket lists (especially with the notion of quasiquote, unquote and splicing), you can now move to the next section where we will write Racket code to generate HTML pages and to process forms.

2 Web Applications in Racket

This section is taken mainly from `http://docs.racket-lang.org/continue/` with some minor changes.

2.1 Static HTML pages

Let's start from the definition of HTML in Racket: here we use **X-expressions** (xexpr). As usual, an example can be helpful: suppose you want to define a paragraph of the form `<p>Hello World</p>`. This is represented by the following xexpr: `'(p "Hello World")`. As explained above, this is just a convenient way of defining a list. In this case, the list is `(list 'p "Hello World")`. In general, an xexpr is just a sequence of nested lists. For instance, the HTML corresponding to the page depicted in Figure 1 above is encoded by the following xexpr:

```
'(html (head (title "Racket Web Experiments"))
  (body
    (h1 "This is a Title")
    (p "This is a paragraph") ) )
```

This is just a list of lists. There are other ways to specify an xexpr (see link above) but we are going to use just this one in the rest of the workshop.

Our next step is understanding how to respond to a request by a browser. As you have seen in the lecture, the browser is the *client*, and we need to implement a *server* in Racket to respond to clients' requests. This is done by implementing a function that manipulates *requests* to a so-called *servlet*. Let's start from a minimal example that returns the HTML code above to *all* requests:

simple–static.rkt

```
1   #lang racket
2
3   (require web-server/servlet
4            web-server/servlet-env)
5
6   (define (myresponse request)
7     (response/xexpr
8      '(html (head (title "Racket Web Experiments"))
9             (body
10             (h1 "This is a Title")
11             (p "This is a paragraph") ) )
12     ))
13
14
15  (serve/servlet myresponse)
```

Let's go through the code above: it includes two items form the web-server module (line 3 and 4) and then it defines a function `myresponse` (we will see the body of this function in a moment). The function `myresponse` is called by the function `serve/servlet` (line 15): *this is the line that starts everything* and it is provided by the web-server library. The function `myresponse` between lines 6 and 12 takes a `request` and for the moment it returns a static xexpr as a response (line 7), discarding the `request`.

Exercise 2.1 *Try the following before proceeding:*

1. *Save the code above in a file and run it from DrRacket. A web browser window should appear when you click the Run button.*
2. *Modify the code above to add at least a new paragraph with a word in bold (the HTML code for this is* `<b>Test</b>`*).*

3. *Check the address in your browser: it should be something like* `http://localhost:8000/servlets/standalone.rkt`. *Try to connect from another machine (you need to know your IP address. If you don't know how to do this, ask a member of staff): it will not work, and we will see in a moment how to fix this.*

Before moving to parsing requests, let's address the third point of the previous exercise. As seen above, the Racket server only replies to requests from *localhost*. To change this default behaviour, modify line 15 as follows:

```
(serve/servlet myresponse
               #:listen-ip #f)
```

Exercise 2.2 *Try to connect to the server from another machine or device using the IP address (ask a member of staff if you don't know how to get the IP address of your machine).*

There are a number of other options that are available for the `serve/servlet` function. A full list is available at `http://docs.racket-lang.org/web-server/run.html#(part._.Full_.A.P.I)`. This is an example of some of the options:

```
(serve/servlet myresponse
               #:listen-ip #f
               #:port 8080
               #:servlet-path "/helloworld.rkt"
               #:launch-browser? #f)
```

In this case: the server listens to all requests on port 8080, the servlet is mapped to an address of the form `http://192.168.1.2:8080/helloworld.rkt` and the browser is *not* launched when the code is executed (this is useful, for instance, if you want to run the Racket server on a Rasperry Pi).

Exercise 2.3 *Try all the various options above and make sure you understand how the server behaves before moving to the next part*

2.2 Dynamic HTML pages

We start from a very simple "dynamic" page: it takes a name (for instance "Franco") and it prints "Hi Franco":

simple–dynamic.rkt

```
#lang racket

(require web-server/servlet
         web-server/servlet-env)

(define (myresponse request)
  ;; We extract the key/value pairs (if present):
  (define bindings (request-bindings request))

  ;; If there is a "name" key, we print "Hi (name)"
  (cond ((exists-binding? 'name bindings)
         (define myname (extract-binding/single 'name bindings))
         (response/xexpr
          `(html (head (title "Simple Page"))
                 (body
                  (h1 "A Simple Dynamic page")
                  (div ((class "name"))
                       (p "Hi " ,myname))
                  ))))
        (else
         ;; If there is no "name", we generate a form:
         (response/xexpr
          `(html (head (title "Simple Page"))
                 (body
                  (h1 "A Simple Dynamic page")
                  (form
                        (input ((name "name")))
                        (input ((type "submit"))))))))
         )
        )



(serve/servlet myresponse)
```

Exercise 2.4 *Copy the code above in a file and run it. Check the address bar when you submit a name.*

The key difference with the code for a static page is the presence of *parameters*. Line 8 defines a a set of *bindings*, by extracting them from the `request`. When the page is executed for the first time these bindings are empty. In particular, there is no value for the key `name` (i.e., the condition at line 11 is false), and as a result the code between lines 20 and 29 is executed: this code generates a simple HTML page with a form containing an input field of type text and a submit button.
When the user submits this form, the page is called again and at this point, if you have entered a value in the text field, the condition at line 11 is true. At this point, the HTML code at lines 14–19 is generated *dynamically*. **Notice the comma in front of** `myname` in line 18: this is a *quasiquote*, see above.
The key functions used here are:

- `(request-bindings request)` (line 8) to extract all the bindings (key/-value pairs) from a request.
- `(exists-binding? 'name bindings)` checks that there exists a parameter called "name".
- `(extract-binding/single 'name bindings)` returns the string (the value) of the parameter called "name" in bindings.

Exercise 2.5 *Add a second input field to enter the surname and display it on screen when the submit button is pressed.*

2.3 A dynamic web page to control Arduino LEDs

Let's now move to the following problem: use a web interface to switch on and off some LEDs connected to an Arduino board using Racket and Firmata. At this point you should be familiar with controlling an Arduino board with Racket and Firmata (see the traffic light project in Block 1). Consider now a web page similar to the one reported in Figure 2: we are going to use the checkboxes to select which LEDs to turn on and off.
The following code can be used to detect the checkboxes that are selected:

webLights2.rkt

```
#lang racket

(require web-server/servlet
         web-server/servlet-env)

;; The number of PINs)
(define NPINS 3)

;; The list of PINs to be activated. Initially, this is empty
```

```
10 (define CONTENT
11   (list)
12   )
13
14 (define (myresponse request)
15   (define bindings (request-bindings request))
16   (cond ( (exists-binding? 'pin bindings)
17           (set! CONTENT (extract-bindings 'pin bindings))
18           )
19         )
20   (response/xexpr
21    `(html (head (title "Racket Web Experiments"))
22           (body
23            (h1 "This is a Title")
24            (p "This is a paragraph")
25            (form
26             ,@(render-checkboxes CONTENT)
27             (p (input ((type "submit") (value "Send your data"))))
28             ) ;; end form
29            )))) ;; end define response-generator
30
31 (define (render-checkboxes a-list)
32   (for/list ([pin (range 1 (+ 1 NPINS))])
33     (cond ( (not (member (number->string pin) a-list))
34            `(p (input ((name "pin") (type "checkbox")
35          (value ,(number->string pin)))) ,(number->string pin))
36            )
37           (else
38            `(p (input ((name "pin") (type "checkbox") (checked "checked")
39          (value ,(number->string pin)))) ,(number->string pin))
40            )
41           )
42     )
43 )
44
45 (serve/servlet myresponse)
```

- Line 7 declares the number of PINs to be used. In this case, we use 3 PINs and we assume that LEDs are connected to digital PINs 1, 2 and 3. Line 10 declares a *list*: this is the list of PINs to be activated. At the beginning the list is empty.

- The function `myresponse` takes a request and extracts the bindings (see above). The condition starting at line 16 checks if the request contains values of a parameter called `pin`. If this is the case, the list `CONTENT` is updated accordingly: notice that here we use the function `extract-bindings` that returns a *list*.

- The main task of the function `myresponse` is to return an xexpr. As in the previous example, we build an xexpr between lines 21 and 29. However, *look carefully at* **line 26**: the line starts with the sequence of characters `,@`, which represent the operation of unquote-splicing. If you forgot what this operation does, please refer to the first section of this document!

- More details about line 26: here we call the function `render-checkboxes` that takes an argument (the list of PINs to be activated) and returns, guess what, a list, as described in the next item.

- The function `render-checkboxes` between lines 31 and 41 builds a list of xexpr using the `for/list` construct (see Block 2 and `http://docs.racket-lang.org/guide/for.html#%28part._for_list_and_for__list%29`). More in detail, we iterate over all the possible PINs from 1 to NPINS+1 (in our case, between 1 and 4 *excluded*). We then check whether the current pin in the iteration is a member of the list that is passed as an argument (line 33). If this is the case, the checkbox should be checked (line 34); otherwise, we generate an un-checked checkbox (line 37).

Exercise 2.6 *(***OPTIONAL***) This code has a bug: can you find it? If you are able to find it, can you fix it?*

Exercise 2.7 *Add the code to control the Arduino PINs, connect an Arduino board, attach the PINs and make sure that everything is working fine.*
HINT: You obviously need the Firmata library. Then, make sure you initialize the PINs appropriately. Finally, modify the function `render-checkboxes` *to switch the PINs on and off (you probably need an additional for loop).*