Tutorial

August 1, 2007

Contents

1 ParenScript Tutorial

This chapter is a short introductory tutorial to ParenScript. It hopefully will give you an idea how ParenScript can be used in a Lisp web application.

2 Setting up the ParenScript environment

In this tutorial, we will use the Portable Allegroserve webserver to serve the tutorial web application. We use the ASDF system to load both Allegroserve and ParenScript. I assume you have installed and downloaded Allegroserve and Parenscript, and know how to setup the central registry for ASDF.

```
(asdf:oos 'asdf:load-op :aserve)

; ... lots of compiler output ...

(asdf:oos 'asdf:load-op :parenscript)

; ... lots of compiler output ...
```

The tutorial will be placed in its own package, which we first have to define.

```
(defpackage :js-tutorial
  (:use :common-lisp :net.aserve :net.html.generator :parenscript))

(in-package :js-tutorial)
```

The next command starts the webserver on the port 8000.

```
(start :port 8000)
```

We are now ready to generate the first JavaScript-enabled webpages using ParenScript.

3 A simple embedded example

The first document we will generate is a simple HTML document, which features a single hyperlink. When clicking the hyperlink, a JavaScript handler opens a popup alert window with the string "Hello world". To facilitate the development, we will factor out the HTML generation to a separate function, and setup a handler for the url "/tutorial1", which will generate HTTP headers and call the function `TUTORIAL1`. At first, our function does nothing.

```
(defun tutorial1 (req ent)
  (declare (ignore req ent))
  nil)

(publish :path "/tutorial1"
         :content-type "text/html; charset=ISO-8859-1"
         :function #'(lambda (req ent)
                       (with-http-response (req ent)
                         (with-http-body (req ent)
                           (tutorial1 req ent)))))
```

Browsing "http://localhost:8000/tutorial1" should return an empty HTML page. It's now time to fill this rather page with content. ParenScript features a macro that generates a string that can be used as an attribute value of HTML nodes.

```
(defun tutorial1 (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript tutorial: 1st example"))
    (:body (:h1 "ParenScript tutorial: 1st example")
           (:p "Please click the link below." :br
               ((:a :href "#" :onclick (js-inline
                                        (alert "Hello World")))
                "Hello World"))))))
```

Browsing "http://localhost:8000/tutorial1" should return the following HTML:

```
<html><head><title>ParenScript tutorial: 1st example</title>
</head>
<body><h1>ParenScript tutorial: 1st example</h1>
<p>Please click the link below.<br/>
```

```
<a href="#"
   onclick="javascript:alert(&quot;Hello World&quot;);">Hello World</a>
</p>
</body>
</html>
```

4 Adding an inline ParenScript

Suppose we now want to have a general greeting function. One way to do this is to add the javascript in a `SCRIPT` element at the top of the HTML page. This is done using the `JS-SCRIPT` macro which will generate the necessary XML and comment tricks to cleanly embed JavaScript. We will redefine our `TUTORIAL1` function and add a few links:

```
(defun tutorial1 (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head
     (:title "ParenScript tutorial: 2nd example")
     (js-script
      (defun greeting-callback ()
        (alert "Hello World"))))
    (:body
     (:h1 "ParenScript tutorial: 2nd example")
     (:p "Please click the link below." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Hello World")
         :br "And maybe this link too." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Knock knock")
         :br "And finally a third link." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Hello there"))))))
```

This will generate the following HTML page, with the embedded JavaScript nicely sitting on top. Take note how `GREETING-CALLBACK` was converted to camelcase, and how the lispy `DEFUN` was converted to a JavaScript function declaration.

```
<html><head><title>ParenScript tutorial: 2nd example</title>
<script type="text/javascript">
// <![CDATA[
function greetingCallback() {
  alert("Hello World");
}
// ]]>
</script>
</head>
<body><h1>ParenScript tutorial: 2nd example</h1>
<p>Please click the link below.<br/>
<a href="#"
   onclick="javascript:greetingCallback();">Hello World</a>
```

```
<br/>
And maybe this link too.<br/>
<a href="#"
   onclick="javascript:greetingCallback();">Knock knock</a>
<br/>

And finally a third link.<br/>
<a href="#"
   onclick="javascript:greetingCallback();">Hello there</a>
</p>
</body>
</html>
```

5 Generating a JavaScript file

The best way to integrate ParenScript into a Lisp application is to generate a JavaScript file from ParenScript code. This file can be cached by intermediate proxies, and webbrowsers won't have to reload the javascript code on each pageview. A standalone JavaScript can be generated using the macro `JS-FILE`. We will publish the tutorial JavaScript under "/tutorial.js".

```
(defun tutorial1-file (req ent)
  (declare (ignore req ent))
  (js-file
   (defun greeting-callback ()
     (alert "Hello World"))))

(publish :path "/tutorial1.js"
         :content-type "text/javascript; charset=ISO-8859-1"
         :function #'(lambda (req ent)
                       (with-http-response (req ent)
                         (with-http-body (req ent)
                           (tutorial1-file req ent)))))

(defun tutorial1 (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head
     (:title "ParenScript tutorial: 3rd example")
     ((:script :language "JavaScript" :src "/tutorial1.js")))
    (:body
     (:h1 "ParenScript tutorial: 3rd example")
     (:p "Please click the link below." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Hello World")
         :br "And maybe this link too." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Knock knock")
         :br "And finally a third link." :br
         ((:a :href "#" :onclick (js-inline (greeting-callback)))
          "Hello there"))))))
```

This will generate the following JavaScript code under "/tutorial1.js":

```
function greetingCallback() {
  alert("Hello World");
}
```

and the following HTML code:

```
<html><head><title>ParenScript tutorial: 3rd example</title>
<script language="JavaScript" src="/tutorial1.js"></script>
</head>
<body><h1>ParenScript tutorial: 3rd example</h1>
<p>Please click the link below.<br/>
<a href="#" onclick="javascript:greetingCallback();">Hello World</a>
<br/>
And maybe this link too.<br/>
<a href="#" onclick="javascript:greetingCallback();">Knock knock</a>
<br/>

And finally a third link.<br/>
<a href="#" onclick="javascript:greetingCallback();">Hello there</a>
</p>
</body>
</html>
```

6 A ParenScript slideshow

While developing ParenScript, I used JavaScript programs from the web and rewrote them using ParenScript. This is a nice slideshow example from

```
http://www.dynamicdrive.com/dynamicindex14/dhtmlslide.htm
```

The slideshow will be accessible under "/slideshow", and will slide through the images "photo1.png", "photo2.png" and "photo3.png". The first ParenScript version will be very similar to the original JavaScript code. The second version will then show how to integrate data from the Lisp environment into the ParenScript code, allowing us to customize the slideshow application by supplying a list of image names. We first setup the slideshow path.

```
(publish :path "/slideshow"
         :content-type "text/html"
         :function #'(lambda (req ent)
                       (with-http-response (req ent)
                         (with-http-body (req ent)
                           (slideshow req ent)))))

(publish :path "/slideshow.js"
         :content-type "text/html"
         :function #'(lambda (req ent)
                       (with-http-response (req ent)
                         (with-http-body (req ent)
                           (js-slideshow req ent)))))
```

The images are just random images I found on my harddrive. We will publish them by hand for now.

```
(publish-file :path "/photo1.png"
              :file "/home/manuel/bknr-sputnik.png")
(publish-file :path "/photo2.png"
              :file "/home/manuel/bknrlogo_red648.png")
(publish-file :path "/photo3.png"
              :file "/home/manuel/bknr-sputnik.png")
```

The function `SLIDESHOW` generates the HTML code for the main slideshow page. It also features little bits of ParenScript. These are the callbacks on the links for the slideshow application. In this special case, the javascript generates the links itself by using `document.write` in a "SCRIPT" element. Users that don't have JavaScript enabled won't see anything at all.

`SLIDESHOW` also generates a static array called `PHOTOS` which holds the links to the photos of the slideshow. This array is handled by the ParenScript code in "slideshow.js". Note how the HTML code issued by the JavaScript is generated using the `HTML` construct. In fact, we have two different HTML generators in the example below, one is the standard Lisp HTML generator, and the other is the JavaScript HTML generator, which generates a JavaScript expression.

```
(defun slideshow (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript slideshow")
           ((:script :language "JavaScript"
                     :src "/slideshow.js"))
           (js-script
            (defvar *linkornot* 0)
            (defvar photos (array "photo1.png"
                                  "photo2.png"
                                  "photo3.png"))))
    (:body (:h1 "ParenScript slideshow")
         (:body (:h2 "Hello")
                ((:table :border 0
                         :cellspacing 0
                         :cellpadding 0)
                 (:tr ((:td :width "100%" :colspan 2 :height 22)
           (:center
            (js-script
             (let ((img
                    (html
                     ((:img :src (aref photos 0)
                            :name "photoslider"
                            :style ( + "filter:"
                                       (js (reveal-trans
                                            (setf duration 2)
                                            (setf transition 23))))
                               :border 0)))))
               (document.write
                (if (= *linkornot* 1)
```

```
                    (html ((:a :href "#"
                               :onclick (js-inline (transport)))
                           img))
                    img)))))))
                 (:tr ((:td :width "50%" :height "21")
                       ((:p :align "left")
                        ((:a :href "#"
                             :onclick (js-inline (backward)
                                                 (return false)))
                         "Previous Slide")))
                      ((:td :width "50%" :height "21")
                       ((:p :align "right")
                        ((:a :href "#"
                             :onclick (js-inline (forward)
                                                 (return false)))
                         "Next Slide")))))))))))
```

SLIDESHOW generates the following HTML code (long lines have been broken down):

```
<html><head><title>ParenScript slideshow</title>
<script language="JavaScript" src="/slideshow.js"></script>
<script type="text/javascript">
// <![CDATA[
var LINKORNOT = 0;
var photos = [ "photo1.png", "photo2.png", "photo3.png" ];
// ]]>
</script>
</head>
<body><h1>ParenScript slideshow</h1>
<body><h2>Hello</h2>
<table border="0" cellspacing="0" cellpadding="0">
<tr><td width="100%" colspan="2" height="22">
<center><script type="text/javascript">
// <![CDATA[
var img =
    "<img src=\"" + photos[0]
    + "\" name=\"photoslider\"
      style=\"filter:revealTrans(duration=2,transition=23)\"
      border=\"0\"></img>";
document.write(LINKORNOT == 1 ?
                  "<a href=\"#\"
                      onclick=\"javascript:transport()\">"
                  + img + "</a>"
                  : img);
// ]]>
</script>
</center>
</td>
</tr>
<tr><td width="50%" height="21"><p align="left">
<a href="#"
   onclick="javascript:backward(); return false;">Previous Slide</a>

</p>
```

```
</td>
<td width="50%" height="21"><p align="right">
<a href="#"
   onclick="javascript:forward(); return false;">Next Slide</a>
</p>
</td>
</tr>
</table>
</body>
</body>
</html>
```

The actual slideshow application is generated by the function `JS-SLIDESHOW`, which generates a ParenScript file. The code is pretty straightforward for a lisp savy person. Symbols are converted to JavaScript variables, but the dot "." is left as is. This enables us to access object "slots" without using the `SLOT-VALUE` function all the time. However, when the object we are referring to is not a variable, but for example an element of an array, we have to revert to `SLOT-VALUE`.

```
(defun js-slideshow (req ent)
  (declare (ignore req ent))
  (js-file
   (defvar *preloaded-images* (make-array))
   (defun preload-images (photos)
     (dotimes (i photos.length)
       (setf (aref *preloaded-images* i) (new *Image)
             (slot-value (aref *preloaded-images* i) 'src)
             (aref photos i))))

   (defun apply-effect ()
     (when (and document.all photoslider.filters)
       (let ((trans photoslider.filters.reveal-trans))
         (setf (slot-value trans '*Transition)
               (floor (* (random) 23)))
         (trans.stop)
         (trans.apply))))

   (defun play-effect ()
     (when (and document.all photoslider.filters)
       (photoslider.filters.reveal-trans.play)))

   (defvar *which* 0)

   (defun keep-track ()
     (setf window.status
           (+ "Image " (1+ *which*) " of " photos.length)))

   (defun backward ()
     (when (> *which* 0)
       (decf *which*)
       (apply-effect)
       (setf document.images.photoslider.src
             (aref photos *which*))
```

```
        (play-effect)
        (keep-track)))

   (defun forward ()
     (when (< *which* (1- photos.length))
       (incf *which*)
       (apply-effect)
       (setf document.images.photoslider.src
             (aref photos *which*))
       (play-effect)
       (keep-track)))

   (defun transport ()
     (setf window.location (aref photoslink *which*)))))
```

`JS-SLIDESHOW` generates the following JavaScript code:

```
var PRELOADEDIMAGES = new Array();
function preloadImages(photos) {
  for (var i = 0; i != photos.length; i = i++) {
    PRELOADEDIMAGES[i] = new Image;
    PRELOADEDIMAGES[i].src = photos[i];
  }
}
function applyEffect() {
  if (document.all && photoslider.filters) {
    var trans = photoslider.filters.revealTrans;
    trans.Transition = Math.floor(Math.random() * 23);
    trans.stop();
    trans.apply();
  }
}
function playEffect() {
  if (document.all && photoslider.filters) {
    photoslider.filters.revealTrans.play();
  }
}
var WHICH = 0;
function keepTrack() {
  window.status = "Image " + (WHICH + 1) + " of " +
                  photos.length;
}
function backward() {
  if (WHICH > 0) {
    --WHICH;
    applyEffect();
    document.images.photoslider.src = photos[WHICH];
    playEffect();
    keepTrack();
  }
}
function forward() {
  if (WHICH < photos.length - 1) {
    ++WHICH;
    applyEffect();
```

```
    document.images.photoslider.src = photos[WHICH];
    playEffect();
    keepTrack();
  }
}
function transport() {
  window.location = photoslink[WHICH];
}
```

7 Customizing the slideshow

For now, the slideshow has the path to all the slideshow images hardcoded in the HTML code, as well as in the publish statements. We now want to customize this by publishing a slideshow under a certain path, and giving it a list of image urls and pathnames where those images can be found. For this, we will create a function `PUBLISH-SLIDESHOW` which takes a prefix as argument, as well as a list of image pathnames to be published.

```
(defun publish-slideshow (prefix images)
  (let* ((js-url (format nil "~Aslideshow.js" prefix))
         (html-url (format nil "~Aslideshow" prefix))
         (image-urls
          (mapcar #'(lambda (image)
                      (format nil "~A~A.~A" prefix
                              (pathname-name image)
                              (pathname-type image)))
                  images)))
    (publish :path html-url
             :content-type "text/html"
             :function #'(lambda (req ent)
                           (with-http-response (req ent)
                             (with-http-body (req ent)
                               (slideshow2 req ent image-urls)))))
    (publish :path js-url
             :content-type "text/html"
             :function #'(lambda (req ent)
                           (with-http-response (req ent)
                             (with-http-body (req ent)
                               (js-slideshow req ent)))))
    (map nil #'(lambda (image url)
                 (publish-file :path url
                               :file image))
         images image-urls)))

(defun slideshow2 (req ent image-urls)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript slideshow")
           ((:script :language "JavaScript"
                     :src "/slideshow.js"))
           ((:script :type "text/javascript")
            (:princ (format nil "~%// <![CDATA[~%"))
```

```
            (:princ (js (defvar *linkornot* 0)))
            (:princ (js-to-string `(defvar photos
                                     (array ,@image-urls))))
            (:princ (format nil "~%// ]]>~%"))))
     (:body (:h1 "ParenScript slideshow")
          (:body (:h2 "Hello")
                 ((:table :border 0
                          :cellspacing 0
                          :cellpadding 0)
                  (:tr ((:td :width "100%" :colspan 2 :height 22)
          (:center
           (js-script
            (let ((img
                   (html
                    ((:img :src (aref photos 0)
                           :name "photoslider"
                           :style ( + "filter:"
                                      (js (reveal-trans
                                           (setf duration 2)
                                           (setf transition 23))))
                           :border 0)))))
                 (document.write
                  (if (= *linkornot* 1)
                      (html ((:a :href "#"
                                 :onclick (js-inline (transport)))
                             img))
                      img)))))))
                  (:tr ((:td :width "50%" :height "21")
                        ((:p :align "left")
                         ((:a :href "#"
                              :onclick (js-inline (backward)
                                                  (return false)))
                          "Previous Slide")))
                       ((:td :width "50%" :height "21")
                        ((:p :align "right")
                         ((:a :href "#"
                              :onclick (js-inline (forward)
                                                  (return false)))
                          "Next Slide")))))))))))
```

We can now publish the same slideshow as before, under the "/bknr/" prefix:

```
(publish-slideshow "/bknr/"
  `("/home/manuel/bknr-sputnik.png"
    "/home/manuel/bknrlogo_red648.png"
    "/home/manuel/screenshots/screenshot-14.03.2005-11.54.33.png"))
```

That's it, we can now access our customized slideshow under

```
http://localhost:8000/bknr/slideshow
```