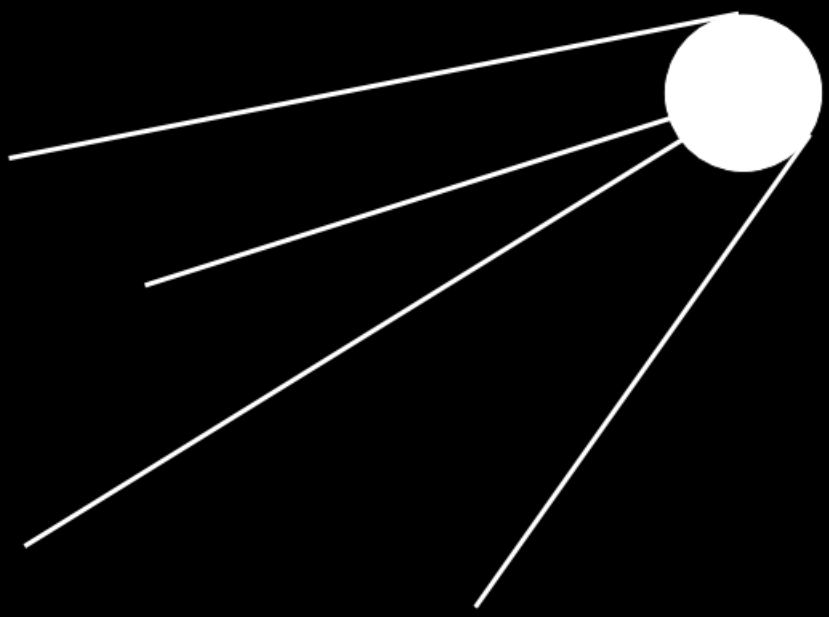

KlammerScript Manual

Manuel Odendahl

Contents

Chapter 1

KlammerScript Introduction

1.1 Introduction

KlammerScript is a simple language that looks a lot like Lisp, but actually is JavaScript in disguise. Actually, it is JavaScript embedded in a host Lisp. This way, JavaScript programs can be seamlessly integrated in a Lisp web application. The programmer doesn't have to resort to a different syntax, and JavaScript code can easily be generated without having to resort to complicated string generation or 'FORMAT' expressions.

An example is worth more than a thousand words. The following Lisp expression is a call to the KlammerScript "compiler". The KlammerScript "compiler" transforms the expression in KlammerScript into an equivalent, human-readable expression in JavaScript.

```
(js
  (defun foobar (a b)
    (return (+ a b))))
```

The resulting javascript is:

```
function foobar(a, b) {
  return a + b;
}
```

Great care has been given to the indentation and overall readability of the generated JavaScript code.

1.2 Features

KlammerScript supports all the statements and expressions defined by the EcmaScript 262 standard. Lisp symbols are converted to camelcase, javascript-compliant syntax. This idea is taken from Linj by Antonio Menezes Leitao. Here are a few examples of Lisp symbol to JavaScript name conversion:

```
(js-to-string 'foobar)      => "foobar"
(js-to-string 'foo-bar)     => "fooBar"
(js-to-string 'foo-b-@-r)   => "fooBAtR"
(js-to-string 'foo-b@r)     => "fooBatr"
(js-to-string '*array)      => "Array"
(js-to-string '*math.floor) => "Math.floor"
```

It also supports additional iteration constructs, relieving the programmer of the burden of iterating over arrays. 'for' loops can be written using the customary 'DO' syntax.

```
(js
  (do ((i 0 (incf i))
       (j (aref arr i) (aref arr i)))
      ((>= i 10))
    (alert (+ "i is " i " and j is " j))))

; compiles to

for (var i = 0, j = arr[i]; i < 10; i = ++i, j = arr[i]) {
  alert("i is " + i + " and j is " + j);
}
```

KlammerScript uses the Lisp reader, allowing for reader macros. It also comes with its own macro environment, allowing host macros and KlammerScript to coexist without interfering with each other. Furthermore, KlammerScript uses its own compiler macro system, allowing for an even further customization of the generation of JavaScript. For example, the '1+' construct is implemented using a KlammerScript macro:

```
(defjsmacro 1+ (form)
  `(+ ,form 1))
```

KlammerScript allows the creation of JavaScript objects in a Lispy way, using keyword arguments.

```
(js
  (create :foo "foo"
          :bla "bla"))

; compiles to

{ foo : "foo",
  bla : "bla" }
```

KlammerScript features a HTML generator. Using the same syntax as the 'HTMLGEN' package of Franz, Inc., it can generate JavaScript string expressions. This allows for a clean integration of HTML in KlammerScript code, instead of writing the tedious and error-prone string generation code generally found in JavaScript.

```
(js
  (defun add-div (name href link-text)
    (document.write
     (html ((:div :id name)
            "The link is: "
            ((:a :href href) link-text))))))

; compiles to

function addDiv(name, href, linkText) {
  document.write("<div id=\"" + name + "\">The link is: <a href=\""
                                + href + "\">"
                                + linkText + "</a></div>");
}
```

In order to have a complete web application framework available in Lisp, KlammerScript also provides a sexp-based syntax for CSS files. Thus, a complete web application featuring HTML, CSS and JavaScript documents can be generated using Lisp syntax, allowing the programmer to use Lisp macros to factor out the redundancies and complexities of Web syntax. For example, to generate a CSS inline node in a HTML document:

```
(html-stream *standard-output*
  (html
   (:html
    (:head
     (css (* :border "1px solid black")
          (div.bl0rg :font-family "serif")
          (("a:active" "a:hoover") :color "black" :size "200%"))))))

; which produces

<html><head><style type="text/css">
<!--
* {
   border:1px solid black;
}

div.bl0rg {
   font-family:serif;
}

a:active,a:hoover {
   color:black;
   size:200%;
}

-->
</style>
```

```
</head>
</html>
```

1.3 Getting KlammerScript

KlammerScript can be obtained from the BKNR subversion repository at

```
svn://bknr.net/trunk/bknr/src/js
```

KlammerScript does not depend on any part of BKNR though. You can download snapshots of KlammerScript at the webpage

```
http://bknr.net/klammerscript
```

or using asdf-install.

```
(asdf-install:install 'klammerscript)
```

After downloading the KlammerScript sourcecode, set up the ASDF central registry by adding a symlink to "klammerscript.asd". Then use ASDF to load KlammerScript. You may want to edit the ASDF file to remove the dependency on the Allegroserve HTMLGEN facility.

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

KlammerScript was written by Manuel Odendahl. He can be reached at

```
manuel@bknr.net
```

Chapter 2

KlammerScript Tutorial

2.1 Klammerscript Tutorial

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

2.2 Setting up the Klammerscript 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 Klammerscript. I assume you have installed and downloaded Allegroserve and Klammerscript, 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 :klammerscript)

; ... 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 :js :net.html.generator))

(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 Klammerscript.

2.3 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. Klammerscript 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 "Klammerscript tutorial: 1st example"))
    (:body (:h1 "Klammerscript 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:

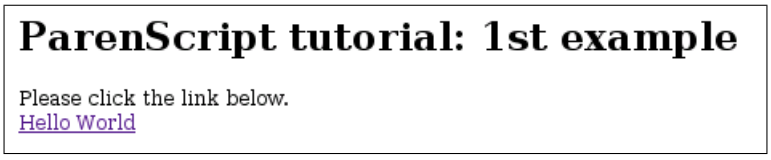

Figure 2.1: Embedded Klammerscript example

```
<html><head><title>Klammerscript tutorial: 1st example</title>
</head>
<body><h1>Klammerscript 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>
```

2.4 Adding an inline Klammerscript

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 "Klammerscript tutorial: 2nd example")
     (js-script
      (defun greeting-callback ()
        (alert "Hello World"))))
    (:body
     (:h1 "Klammerscript 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>Klammerscript tutorial: 2nd example</title>
<script type="text/javascript">
// <![CDATA[
function greetingCallback() {
```

Figure 2.2: Inline Klammerscript example

```
  alert("Hello World");
}
// ]]>
</script>
</head>
<body><h1>Klammerscript 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>
```

2.5 Generating a JavaScript file

The best way to integrate Klammerscript into a Lisp application is to generate a JavaScript file from Klammerscript 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 "Klammerscript tutorial: 3rd example")
     ((:script :language "JavaScript" :src "/tutorial1.js")))
    (:body
     (:h1 "Klammerscript 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>Klammerscript tutorial: 3rd example</title>
<script language="JavaScript" src="/tutorial1.js"></script>
</head>
<body><h1>Klammerscript 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>
```

2.6 A Klammerscript slideshow

While developing Klammerscript, I used JavaScript programs from the web and rewrote them using Klammerscript. 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 Klammerscript 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 Klammerscript 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 Klammerscript. 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 Klammerscript 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 "Klammerscript slideshow")
           ((:script :language "JavaScript"
                     :src "/slideshow.js"))
           (js-script
            (defvar *linkornot* 0)
            (defvar photos (array "photo1.png"
                                  "photo2.png"
                                  "photo3.png"))))
    (:body (:h1 "Klammerscript 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>Klammerscript 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>Klammerscript 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>
```

Figure 2.3: Klammerscript Slideshow

The actual slideshow application is generated by the function 'JS-SLIDESHOW', which generates a Klammerscript 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];
}
```

2.7 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 "Klammerscript 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 "Klammerscript 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

!
```

Chapter 3

KlammerScript Language Reference

3.1 KlammerScript Language Reference

This chapters describes the core constructs of KlammerScript, as well as its compilation model. This chapter is aimed to be a comprehensive reference for KlammerScript developers. Programmers looking for how to tweak the KlammerScript compiler itself should turn to the KlammerScript Internals chapter.

3.2 Statements and Expressions

In contrast to Lisp, where everything is an expression, JavaScript makes the difference between an expression, which evaluates to a value, and a statement, which has no value. Examples for JavaScript statements are 'for', 'with' and 'while'. Most KlammerScript forms are expression, but certain special forms are not (the forms which are transformed to a JavaScript statement). All KlammerScript expressions are statements though. Certain forms, like 'IF' and 'PROGN', generate different JavaScript constructs whether they are used in an expression context or a statement context. For example:

```
(+ i (if 1 2 3)) => i + (1 ? 2 : 3)

(if 1 2 3)
   => if (1) {
        2;
      } else {
        3;
      }
```

3.3 Symbol conversion

Lisp symbols are converted to JavaScript symbols by following a few simple rules. Special characters '!', '?', '#', '$', '@', '%', '/', '*' and '+' get replaced by their written-out equivalents "bang", "what", "hash", "dollar", "at", "percent", "slash", "start" and "plus" respectively.

```
!?#$@% => bangwhathashdollaratpercent
```

The '-' is an indication that the following character should be converted to uppercase. Thus, '-' separated symbols are converted to camelcase. The '_' character however is left untouched.

```
bla-foo-bar => blaFooBar
```

If you want a JavaScript symbol beginning with an uppercase, you can either use a leading '-', which can be misleading in a mathematical context, or a leading '*'.

```
*array => Array
```

The '.' character is left as is in symbols. This allows the KlammerScript programmer to use a practical shortcut when accessing slots or methods of JavaScript objects. Instead of writing

```
(slot-value foobar 'slot)
```

we can write

```
foobar.slot
```

A symbol beggining and ending with '+' or '*' is converted to all uppercase, to signify that this is a constant or a global variable.

```
*global-array*        => GLOBALARRAY

*global-array*.length => GLOBALARRAY.length
```

3.3.1 Reserved Keywords

The following keywords and symbols are reserved in KlammerScript, and should not be used as variable names.

```
! ~ ++ -- * / % + - << >> >>> < > <= >= == != ==== !== & ^ | && ||
*= /= %= += -= <<= >>= >>>= &= ^= |= 1- 1+
ABSTRACT AND AREF ARRAY BOOLEAN BREAK BYTE CASE CATCH CC-IF CHAR CLASS
COMMA CONST CONTINUE CREATE DEBUGGER DECF DEFAULT DEFUN DEFVAR DELETE
DO DOEACH DOLIST DOTIMES DOUBLE ELSE ENUM EQL EXPORT EXTENDS FALSE
```

```
FINAL FINALLY FLOAT FLOOR FOR FUNCTION GOTO IF IMPLEMENTS IMPORT IN INCF
INSTANCEOF INT INTERFACE JS LAMBDA LET LISP LIST LONG MAKE-ARRAY NATIVE NEW
NIL NOT OR PACKAGE PRIVATE PROGN PROTECTED PUBLIC RANDOM REGEX RETURN
SETF SHORT SLOT-VALUE STATIC SUPER SWITCH SYMBOL-MACROLET SYNCHRONIZED T
THIS THROW THROWS TRANSIENT TRY TYPEOF UNDEFINED UNLESS VAR VOID VOLATILE
WHEN WHILE WITH WITH-SLOTS
```

3.4 Literal values

3.4.1 Number literals

```
; number ::= a Lisp number
```

KlammerScript supports the standard JavaScript literal values. Numbers are compiled into JavaScript numbers.

```
1       => 1

123.123 => 123.123
```

Note that the base is not conserved between Lisp and JavaScript.

```
#x10    => 16
```

3.4.2 String literals

```
; string ::= a Lisp string
```

Lisp strings are converted into JavaScript literals.

```
"foobar"      => "foobar"
"bratzel bub" => "bratzel bub"
```

Escapes in Lisp are not converted to JavaScript escapes. However, to avoid having to use double backslashes when constructing a string, you can use the CL-INTERPOL library by Edi Weitz.

3.4.3 Array literals

```
; (ARRAY {values}*)
; (MAKE-ARRAY {values}*)
; (AREF array index)
;
; values ::= a KlammerScript expression
; array  ::= a KlammerScript expression
; index  ::= a KlammerScript expression
```

Array literals can be created using the 'ARRAY' form.

```
(array)       => [ ]

(array 1 2 3) => [ 1, 2, 3 ]

(array (array 2 3)
       (array "foobar" "bratzel bub"))
   => [ [ 2, 3 ], [ "foobar", "bratzel bub" ] ]
```

Arrays can also be created with a call to the 'Array' function using the 'MAKE-ARRAY'. The two forms have the exact same semantic on the JavaScript side.

```
(make-array)       => new Array()

(make-array 1 2 3) => new Array(1, 2, 3)

(make-array
 (make-array 2 3)
 (make-array "foobar" "bratzel bub"))
  => new Array(new Array(2, 3), new Array("foobar", "bratzel bub"))
```

Indexing arrays in KlammerScript is done using the form 'AREF'. Note that JavaScript knows of no such thing as an array. Subscripting an array is in fact reading a property from an object. So in a semantic sense, there is no real difference between 'AREF' and 'SLOT-VALUE'.

3.4.4 Object literals

```
; (CREATE {name value}*)
; (SLOT-VALUE object slot-name)
; (WITH-SLOTS ({slot-name}*) object body)
;
; name      ::= a KlammerScript symbol or a Lisp keyword
; value     ::= a KlammerScript expression
; object    ::= a KlammerScript object expression
; slot-name ::= a quoted Lisp symbol
; body      ::= a list of KlammerScript statements
```

Object literals can be create using the 'CREATE' form. Arguments to the 'CREATE' form is a list of property names and values. To be more "lispy", the property names can be keywords.

```
(create :foo "bar" :blorg 1)
   => { foo : "bar",
        blorg : 1 }

(create :foo "hihi"
        :blorg (array 1 2 3)
```

```
          :another-object (create :schtrunz 1))
    => { foo : "hihi",
         blorg : [ 1, 2, 3 ],
         anotherObject : { schtrunz : 1 } }
```

Object properties can be accessed using the 'SLOT-VALUE' form, which takes an object and a slot-name.

```
(slot-value an-object 'foo) => anObject.foo
```

A programmer can also use the "." symbol notation explained above.

```
an-object.foo => anObject.foo
```

The form 'WITH-SLOTS' can be used to bind the given slot-name symbols to a macro that will expand into a 'SLOT-VALUE' form at expansion time.

```
(with-slots (a b c) this
  (+ a b c))
    => this.a + this.b + this.c
```

3.4.5 Regular Expression literals

```
; (REGEX regex)
;
; regex ::= a Lisp string
```

Regular expressions can be created by using the 'REGEX' form. The regex form actually does nothing at all to its argument, and prints it as is.

```
(regex "/foobar/i") => /foobar/i
```

Here CL-INTERPOL proves really useful.

```
(regex #?r"/([^\s]+)foobar/i") => /([^\s]+)foobar/i
```

3.4.6 Literal symbols

```
; T, FALSE, NIL, UNDEFINED, THIS
```

The Lisp symbols 'T' and 'FALSE' are converted to their JavaScript boolean equivalents 'true' and 'false'.

```
T     => true
FALSE => false
```

The Lisp symbol 'NIL' is converted to the JavaScript keyword 'null'.

```
NIL => null
```

The Lisp symbol 'UNDEFINED' is converted to the JavaScript keyword 'undefined'.

```
UNDEFINED => undefined
```

The Lisp symbol 'THIS' is converted to the JavaScript keyword 'this'.

```
THIS => this
```

3.5 Variables

```
; variable ::= a Lisp symbol
```

All the other literal Lisp values that are not recognized as special forms or symbol macros are converted to JavaScript variables. This extreme freedom is actually quite useful, as it allows the KlammerScript programmer to be flexible, as flexible as JavaScript itself.

```
variable    => variable

a-variable  => aVariable

*math       => Math

*math.floor => Math.floor
```

3.6 Function calls and method calls

```
; (function {argument}*)
; (method   object {argument}*)
;
; function ::= a KlammerScript expression or a Lisp symbol
; method   ::= a Lisp symbol beginning with .
; object   ::= a KlammerScript expression
; argument ::= a KlammerScript expression
```

Any list passed to the JavaScript that is not recognized as a macro or a special form (see "Macro Expansion" below) is interpreted as a function call. The function call is converted to the normal JavaScript function call representation, with the arguments given in paren after the function name.

```
(blorg 1 2) => blorg(1, 2)

(foobar (blorg 1 2) (blabla 3 4) (array 2 3 4))
   => foobar(blorg(1, 2), blabla(3, 4), [ 2, 3, 4 ])

((aref foo i) 1 2) => foo[i](1, 2)
```

A method call is a function call where the function name is a symbol and begins with a "." . In a method call, the name of the function is append to its first argument, thus reflecting the method call syntax of JavaScript. Please note that most method calls can be abbreviated using the "." trick in symbol names (see "Symbol Conversion" above).

```
(.blorg this 1 2) => this.blorg(1, 2)

(this.blorg 1 2) => this.blorg(1, 2)

(.blorg (aref foobar 1) NIL T)
   => foobar[1].blorg(null, true)
```

3.7 Operator Expressions

```
; (operator {argument}*)
; (single-operator argument)
;
; operator ::= one of *, /, %, +, -, <<, >>, >>>, < >, EQL,
;              ==, !=, =, ===, !==, &, ^, |, &&, AND, ||, OR.
; single-operator ::= one of INCF, DECF, ++, --, NOT, !
; argument ::= a KlammerScript expression
```

Operator forms are similar to function call forms, but have an operator as function name.

Please note that '=' is converted to '==' in JavaScript. The '=' KlammerScript operator is not the assignment operator. Unlike JavaScript, KlammerScript supports multiple arguments to the operators.

```
(* 1 2)   => 1 * 2

(= 1 2)   => 1 == 2

(eql 1 2) => 1 == 2
```

Note that the resulting expression is correctly parenthized, according to the JavaScript operator precedence that can be found in table form at:

```
   http://www.codehouse.com/javascript/precedence/

(* 1 (+ 2 3 4) 4 (/ 6 7))
  => 1 * (2 + 3 + 4) * 4 * (6 / 7)
```

The pre/post increment and decrement operators are also available. 'INCF' and 'DECF' are the pre-incrementing and pre-decrementing operators, and '++' and '–' are the post-decrementing version of the operators. These operators can take only one argument.

```
(++ i)   => i++

(-- i)   => i--

(incf i) => ++i

(decf i) => --i
```

The '1+' and '1-' operators are shortforms for adding and substracting 1.

```
(1- i) => i - 1

(1+ i) => i + 1
```

The 'not' operator actually optimizes the code a bit. If 'not' is used on another boolean-returning operator, the operator is reversed.

```
(not (< i 2))   => i >= 2

(not (eql i 2)) => i != 2
```

3.8 Body forms

```
; (PROGN {statement}*) in statement context
; (PROGN {expression}*) in expression context
;
; statement  ::= a KlammerScript statement
; expression ::= a KlammerScript expression
```

The 'PROGN' special form defines a sequence of statements when used in a statement context, or sequence of expression when used in an expression context. The 'PROGN' special form is added implicitly around the branches of conditional executions forms, function declarations and iteration constructs. For example, in a statement context:

```
(progn (blorg i) (blafoo i))
   => blorg(i);
      blafoo(i);
```

In an expression context:

```
(+ i (progn (blorg i) (blafoo i)))
   => i + (blorg(i), blafoo(i))
```

A 'PROGN' form doesn't lead to additional indentation or additional braces around it's body.

3.9 Function Definition

```
; (DEFUN name ({argument}*) body)
; (LAMBDA ({argument}*) body)
;
; name     ::= a Lisp Symbol
; argument ::= a Lisp symbol
; body     ::= a list of KlammerScript statements
```

As in Lisp, functions are defined using the 'DEFUN' form, which takes a name, a list of arguments, and a function body. An implicit 'PROGN' is added around the body statements.

```
(defun a-function (a b)
  (return (+ a b)))
  => function aFunction(a, b) {
       return a + b;
     }
```

Anonymous functions can be created using the 'LAMBDA' form, which is the same as 'DEFUN', but without function name. In fact, 'LAMBDA' creates a 'DEFUN' with an empty function name.

```
(lambda (a b) (return (+ a b)))
  => function (a, b) {
       return a + b;
     }
```

3.10 Assignment

```
; (SETF {lhs rhs}*)
;
; lhs ::= a KlammerScript left hand side expression
; rhs ::= a KlammerScript expression
```

Assignment is done using the 'SETF' form, which is transformed into a series of assignments using the JavaScript '=' operator.

```
(setf a 1) => a = 1

(setf a 2 b 3 c 4 x (+ a b c))
   => a = 2;
      b = 3;
      c = 4;
      x = a + b + c;
```

The 'SETF' form can transform assignments of a variable with an operator expression using this variable into a more "efficient" assignment operator form. For example:

```
(setf a (1+ a))          => a++

(setf a (* 2 3 4 a 4 a)) => a *= 2 * 3 * 4 * 4 * a

(setf a (- 1 a))         => a = 1 - a
```

3.11 Single argument statements

```
; (RETURN {value}?)
; (THROW  {value}?)
;
; value ::= a KlammerScript expression
```

The single argument statements 'return' and 'throw' are generated by the form 'RETURN' and 'THROW'. 'THROW' has to be used inside a 'TRY' form. 'RETURN' is used to return a value from a function call.

```
(return 1)       => return 1

(throw "foobar") => throw "foobar"
```

3.12 Single argument expression

```
; (DELETE     {value})
; (VOID       {value})
; (TYPEOF     {value})
; (INSTANCEOF {value})
; (NEW        {value})
;
; value ::= a KlammerScript expression
```

The single argument expressions 'delete', 'void', 'typeof', 'instanceof' and 'new' are generated by the forms 'DELETE', 'VOID', 'TYPEOF', 'INSTANCEOF' and 'NEW'. They all take a KlammerScript expression.

```
(delete (new (*foobar 2 3 4))) => delete new Foobar(2, 3, 4)

(if (= (typeof blorg) *string)
    (alert (+ "blorg is a string: " blorg))
    (alert "blorg is not a string"))
  => if (typeof blorg == String) {
       alert("blorg is a string: " + blorg);
     } else {
       alert("blorg is not a string");
     }
```

3.13 Conditional Statements

```
; (IF conditional then {else})
; (WHEN condition then)
; (UNLESS condition then)
;
; condition ::= a KlammerScript expression
; then      ::= a KlammerScript statement in statement context, a
;               KlammerScript expression in expression context
; else      ::= a KlammerScript statement in statement context, a
;               KlammerScript expression in expression context
```

The 'IF' form compiles to the 'if' javascript construct. An explicit 'PROGN' around the then branch and the else branch is needed if they consist of more than one statement. When the 'IF' form is used in an expression context, a JavaScript '?', ':' operator form is generated.

```
(if (blorg.is-correct)
    (progn (carry-on) (return i))
    (alert "blorg is not correct!"))
   => if (blorg.isCorrect()) {
        carryOn();
        return i;
      } else {
        alert("blorg is not correct!");
      }

(+ i (if (blorg.add-one) 1 2))
   => i + (blorg.addOne() ? 1 : 2)
```

The 'WHEN' and 'UNLESS' forms can be used as shortcuts for the 'IF' form.

```
(when (blorg.is-correct)
  (carry-on)
  (return i))
    => if (blorg.isCorrect()) {
         carryOn();
         return i;
       }

(unless (blorg.is-correct)
  (alert "blorg is not correct!"))
    => if (!blorg.isCorrect()) {
         alert("blorg is not correct!");
       }
```

3.14 Variable declaration

```
; (DEFVAR var {value}?)
; (LET ({var | (var value)) body)
```

```
;
; var   ::= a Lisp symbol
; value ::= a KlammerScript expression
; body  ::= a list of KlammerScript statements
```

Variables (either local or global) can be declared using the 'DEFVAR' form, which is similar to its equivalent form in Lisp. The 'DEFVAR' is converted to "var ... = ..." form in JavaScript.

```
(defvar *a* (array 1 2 3)) => var A = [ 1, 2, 3 ]

(if (= i 1)
    (progn (defvar blorg "hallo")
           (alert blorg))
    (progn (defvar blorg "blitzel")
           (alert blorg)))
   => if (i == 1) {
        var blorg = "hallo";
        alert(blorg);
      } else {
        var blorg = "blitzel";
        alert(blorg);
      }
```

A more lispy way to declare local variable is to use the 'LET' form, which is similar to its Lisp form.

```
(if (= i 1)
    (let ((blorg "hallo"))
      (alert blorg))
    (let ((blorg "blitzel"))
      (alert blorg)))
   => if (i == 1) {
        var blorg = "hallo";
        alert(blorg);
      } else {
        var blorg = "blitzel";
        alert(blorg);
      }
```

However, beware that scoping in Lisp and JavaScript are quite different. For example, don't rely on closures capturing local variables in the way you'd think they would.

3.15 Iteration constructs

```
; (DO ({var | (var {init}? {step}?)}*) (end-test) body)
; (DOTIMES (var numeric-form) body)
```

```
; (DOLIST (var list-form) body)
; (DOEACH (var object) body)
; (WHILE end-test body)
;
; var          ::= a Lisp symbol
; numeric-form ::= a KlammerScript expression resulting in a number
; list-form    ::= a KlammerScript expression resulting in an array
; object       ::= a KlammerScript expression resulting in an object
; init         ::= a KlammerScript expression
; step         ::= a KlammerScript expression
; end-test     ::= a KlammerScript expression
; body         ::= a list of KlammerScript statements
```

The 'DO' form, which is similar to its Lisp form, is transformed into a JavaScript 'for' statement. Note that the KlammerScript 'DO' form does not have a return value, that is because 'for' is a statement and not an expression in JavaScript.

```
(do ((i 0 (1+ i))
     (l (aref blorg i) (aref blorg i)))
    ((or (= i blorg.length)
         (eql l "Fumitastic")))
  (document.write (+ "L is " l)))
 => for (var i = 0, l = blorg[i];
    !(i == blorg.length || l == "Fumitastic");
         i = i + 1, l = blorg[i]) {
      document.write("L is " + l);
    }
```

The 'DOTIMES' form, which lets a variable iterate from 0 upto an end value, is a shortcut for 'DO'.

```
(dotimes (i blorg.length)
  (document.write (+ "L is " (aref blorg i))))
 => for (var i = 0; i != blorg.length; i = i++) {
      document.write("L is " + blorg[i]);
    }
```

The 'DOLIST' form is a shortcut for iterating over an array. Note that this form creates temporary variables using a function called 'JS-GENSYM', which is similar to its Lisp counterpart 'GENSYM'.

```
(dolist (l blorg)
  (document.write (+ "L is " l)))
 => var tmpArr1 = blorg;
    for (var tmpI2 = 0; tmpI2 < tmpArr1.length;
    tmpI2 = tmpI2++) {
      var l = tmpArr1[tmpI2];
      document.write("L is " + l);
    }
```

The 'DOEACH' form is converted to a 'for (var .. in ..)' form in JavaScript. It is used to iterate over the enumerable properties of an object.

```
(doeach (i object)
   (document.write (+ i " is " (aref object i))))
   => for (var i in object) {
        document.write(i + " is " + object[i]);
      }
```

The 'WHILE' form is transformed to the JavaScript form 'while', and loops until a termination test evaluates to false.

```
(while (film.is-not-finished)
  (this.eat (new *popcorn)))
   => while (film.isNotFinished()) {
        this.eat(new Popcorn);
      }
```

3.16 The 'CASE' statement

```
; (CASE case-value clause*)
;
; clause     ::= (value body)
; case-value ::= a KlammerScript expression
; value      ::= a KlammerScript expression
; body       ::= a list of KlammerScript statements
```

The Lisp 'CASE' form is transformed to a 'switch' statement in JavaScript. Note that 'CASE' is not an expression in KlammerScript. The default case is not named 'T' in KlammerScript, but 'DEFAULT' instead.

```
(case (aref blorg i)
  (1 (alert "one"))
  (2 (alert "two"))
  (default (alert "default clause")))
    => switch (blorg[i]) {
         case 1:   alert("one");
         case 2:   alert("two");
         default:   alert("default clause");
       }
```

3.17 The 'WITH' statement

```
; (WITH (object) body)
;
; object ::= a KlammerScript expression evaluating to an object
; body   ::= a list of KlammerScript statements
```

The 'WITH' form is compiled to a JavaScript 'with' statements, and adds the object 'object' as an intermediary scope objects when executing the body.

```
(with ((create :foo "foo" :i "i"))
  (alert (+ "i is now intermediary scoped: " i)))
   => with ({ foo : "foo",
              i : "i" }) {
        alert("i is now intermediary scoped: " + i);
      }
```

3.18 The 'TRY' statement

```
; (TRY body {(:CATCH (var) body)}? {(:FINALLY body)}?)
;
; body ::= a list of KlammerScript statements
; var  ::= a Lisp symbol
```

The 'TRY' form is converted to a JavaScript 'try' statement, and can be used to catch expressions thrown by the 'THROW' form. The body of the catch clause is invoked when an exception is catched, and the body of the finally is always invoked when leaving the body of the 'TRY' form.

```
(try (throw "i")
 (:catch (error)
   (alert (+ "an error happened: " error)))
 (:finally
   (alert "Leaving the try form")))
   => try {
        throw "i";
      } catch (error) {
        alert("an error happened: " + error);
      } finally {
        alert("Leaving the try form");
      }
```

3.19 The HTML Generator

```
; (HTML html-expression)
```

The HTML generator of KlammerScript is very similar to the HTML generator included in AllegroServe. It accepts the same input forms as the AllegroServer HTML generator. However, non-HTML construct are compiled to JavaScript by the KlammerScript compiler. The resulting expression is a JavaScript expression.

```
(html ((:a :href "foobar") "blorg"))
  => "<a href=\"foobar\">blorg</a>"

(html ((:a :href (generate-a-link)) "blorg"))
  => "<a href=\"" + generateALink() + "\">blorg</a>"
```

We can recursively call the JS compiler in a HTML expression.

```
(document.write
 (html ((:a :href "#"
            :onclick (js-inline (transport))) "link")))
  => document.write("<a href=\"#\" onclick=\""
                    + "javascript:transport();"
                    + "\">link</a>")
```

3.20 Macrology

```
; (DEFJSMACRO name lambda-list macro-body)
; (MACROLET ({name lambda-list macro-body}*) body)
; (SYMBOL-MACROLET ({name macro-body}*) body)
; (JS-GENSYM {string}?)
;
; name        ::= a Lisp symbol
; lambda-list ::= a lambda list
; macro-body  ::= a Lisp body evaluating to KlammerScript code
; body        ::= a list of KlammerScript statements
; string      ::= a string
```

KlammerScript can be extended using macros, just like Lisp can be extended using Lisp macros. Using the special Lisp form 'DEFJSMACRO', the KlammerScript language can be extended. 'DEFJSMACRO' adds the new macro to the toplevel macro environment, which is always accessible during KlammerScript compilation. For example, the '1+' and '1-' operators are implemented using macros.

```
(defjsmacro 1- (form)
  `(- ,form 1))

(defjsmacro 1+ (form)
  `(+ ,form 1))
```

A more complicated KlammerScript macro example is the implementation of the 'DOLIST' form (note how 'JS-GENSYM', the KlammerScript of 'GENSYM', is used to generate new KlammerScript variable names):

```
(defjsmacro dolist (i-array &rest body)
  (let ((var (first i-array))
```

```
        (array (second i-array))
        (arrvar (js-gensym "arr"))
        (idx (js-gensym "i")))
    `(let ((,arrvar ,array))
       (do ((,idx 0 (++ ,idx)))
           ((>= ,idx (slot-value ,arrvar 'length)))
         (let ((,var (aref ,arrvar ,idx)))
           ,@body)))))
```

Macros can be added dynamically to the macro environment by using the KlammerScript 'MACROLET' form (note that while 'DEFJSMACRO' is a Lisp form, 'MACROLET' and 'SYMBOL-MACROLET' are KlammerScript forms). KlammerScript also supports symbol macros, which can be introduced using the KlammerScript form 'SYMBOL-MACROLET'. A new macro environment is created and added to the current macro environment list while compiling the body of the 'SYMBOL-MACROLET' form. For example, the KlammerScript 'WITH-SLOTS' is implemented using symbol macros.

```
(defjsmacro with-slots (slots object &rest body)
  `(symbol-macrolet ,(mapcar #'(lambda (slot)
                                 `(,slot '(slot-value ,object ',slot)))
                             slots)
     ,@body))
```

3.21 The KlammerScript Compiler

```
; (JS-COMPILE expr)
; (JS-TO-STRINGS compiled-expr position)
; (JS-TO-STATEMENT-STRINGS compiled-expr position)
;
; compiled-expr ::= a compiled KlammerScript expression
; position      ::= a column number
;
; (JS-TO-STRING expression)
; (JS-TO-LINE expression)
;
; expression ::= a Lisp list of KlammerScript code
;
; (JS body)
; (JS-INLINE body)
; (JS-FILE body)
; (JS-SCRIPT body)
;
; body ::= a list of KlammerScript statements
```

The KlammerScript compiler can be invoked from withing Lisp and from within KlammerScript itself. The primary API function is 'JS-COMPILE',

which takes a list of KlammerScript, and returns an internal object representing the compiled KlammerScript.

```
(js-compile '(foobar 1 2))
  => #<JS::FUNCTION-CALL {584AA5DD}>
```

This internal object can be transformed to a string using the methods 'JS-TO-STRINGS' and 'JS-TO-STATEMENT-STRINGS', which interpret the KlammerScript in expression and in statement context respectively. They take an additional parameter indicating the start-position on a line (please note that the indentation code is not perfect, and this string interface will likely be changed). They return a list of strings, where each string represents a new line of JavaScript code. They can be joined together to form a single string.

```
(js-to-strings (js-compile '(foobar 1 2)) 0)
  => ("foobar(1, 2)")
```

As a shortcut, KlammerScript provides the functions 'JS-TO-STRING' and 'JS-TO-LINE', which return the JavaScript string of the compiled expression passed as an argument.

```
(js-to-string '(foobar 1 2))
  => "foobar(1, 2)"
```

For static KlammerScript code, the macros 'JS', 'JS-INLINE', 'JS-FILE' and 'JS-SCRIPT' avoid the need to quote the KlammerScript expression. All these forms add an implicit 'PROGN' form around the body. 'JS' returns a string of the compiled body, where the other expression return an expression that can be embedded in a HTML generation construct using the AllegroServe HTML generator. 'JS-SCRIPT' generates a "SCRIPT" node, 'JS-INLINE' generates a string to be used in node attributs, and 'JS-FILE' prints the compiled KlammerScript code to the HTML stream. These macros are also available inside KlammerScript itself, and generate strings that can be used inside KlammerScript code. Note that 'JS-INLINE' in KlammerScript is not the same 'JS-INLINE' form as in Lisp, for example. The same goes for the other compilation macros.

Chapter 4

KlammerScript Internals

4.1 KlammerScript internals

4.1.1 JS Package

The KlammerScript compiler is contained within the 'JS' package.

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

4.2 Introduction

The KlammerScript language is a lispy representation of Javascript. It tries to follow the EcmaScript standard (Ecma 262). The standard can be downloaded at the following URL:

```
;; http://www.ecma-international.org/publications/standards/Ecma-262.htm
```

This chapter describes the inner workings of the KlammerScript "compiler". Compiler is actually pretty far-fetched. Most of the work of KlammerScript goes into the generation of a "human-readable" JavaScript representation and indentation of the KlammerScript language.

However, the inner workings of form expansion and the macro system can be very useful to programmers who want to extend the KlammerScript to contain their own operators and macros.

4.2.1 Overview of the KlammerScript compiler

The KlammerScript compiler takes a Lisp list as input. This Lisp list actually is KlammerScript code. The compiler then treewalks this KlammerScript code and converts it into an internal representation which consists of CLOS objects. These CLOS objects represent the Abstract Syntax Tree of the program to be compiled. Finally, this abstract syntax tree is optimised

a bit, and converted to a list of strings representing "indented" JavaScript code. Actually, most of the code of the KlammerScript compiler handles the "correct" indentation of JavaScript code.

In order to recognize and handle KlammerScript language constructs, the compiler uses a sort of compiler macros which are called "JS-COMPILER-MACRO'. These compiler macros are stored in a hash table and looked up by the compiler while walking the source code. In addition to compiler macro, KlammerScript also supports "normal" macros. In order to avoid the mixing of LISP macros and KlammerScript macros, KlammerScript macros are stored in their own environment, which can be extended using forms like "MACROLET".

The KlammerScript compiler makes only a few peephole optimizations in order to reduce the "generated" code that would clutter up the generated JavaScript code. One optimization for example is to reduce assignment operations to assignment operators when possible (when the left hand of the assignment appears in an arithmetic operation on the right hand side).

4.3 JavaScript name conversion

Normally, the Lisp reader cannot differentiate between lower-case and uppercase symbols, and reads in symbols either as completely uppercase or lowercase. However, in the JavaScript world, camel-case symbol ("symbolsLikeThis") are quite common. In order to map from "natural" Lisp symbols to JavaScript symbols, a conversion function named 'SYMBOL-TO-JS' is used. This function is modelled after the similar conversion of Lisp symbols to Java symbols in the Lisp-to-Java compiler linj. Special characters allowed in Lisp symbols but now in JavaScript are replaced with their "full name". These special symbols are stored in the association list '*SPECIAL-CHARS*'. \index{*special-chars*} \index{special characters}

```
(defparameter *special-chars*
  '((#\! . "Bang")
    (#\? . "What")
    (#\# . "Hash")
    (#\$ . "Dollar")
    (#\@ . "At")
    (#\% . "Percent")
    (#\+ . "Plus")
    (#\* . "Star")
    (#\/ . "Slash")))
```

'STRING-CHARS' is a helper function. \index{STRING-CHARS}

```
(defun string-chars (string)
  (coerce string 'list))
```

In order to convert Lisp "constants" and "global variables", which usually feature the characters '*' or '+' at their beginning and end, we recognize such symbols and convert them to full uppercase JavaScript symbols. The function 'CONSTANT-STRING-P' checks if a symbol begins and end with either '*' or '+'. \index{CONSTANT-STRING-P} \index{constants} \index{global variables}

```
(defun constant-string-p (string)
  (let ((len (length string))
        (constant-chars '(#\+ #\*)))
    (and (> len 2)
         (member (char string 0) constant-chars)
         (member (char string (1- len)) constant-chars))))

(defun first-uppercase-p (string)
  (and (> (length string) 1)
       (member (char string 0) '(#\+ #\*))))
```

The function 'SYMBOL-TO-JS' does the actual work. It first splits the symbol to be converted at '.'-boundaries, in order to allow for slot-access shortcuts to conserve their camelcase. Each symbol part is then processed by a recursive call to 'SYMBOL-TO-JS'. Finally, all the converted symbols are joined using the character '.'. For example,

```
;; (symbol-to-js 'foo-bar.broesel-foo) => "fooBar.broeselFoo"
```

The function itself holds the current "case state", which can be either lowercase or uppercase. In the lowercase state, characters are converted to lowercase characters, in the uppercase state, characters are obviously converted to uppercase. When a character is a special character, it is replaced by its full name which is looked up in the association list '*SPECIAL-CHARS*'. When the character is '-', the case state is switched for the next character, which usually puts the case state into uppercase for one character. Thus, symbols are converted to camelcase characters at '-' boundaries. The function starts in the lowercase mode. For example,

```
;; (symbol-to-js 'foo-bar) => "fooBar"
```

Here is the code for "SYMBOL-TO-JS".

```
(defun symbol-to-js (symbol)
  (when (symbolp symbol)
    (setf symbol (symbol-name symbol)))
  (let ((symbols (string-split symbol '(#\.))))
    (cond ((null symbols) "")
     ((= (length symbols) 1)
      (let (res
```

```
        (lowercase t)
        (all-uppercase nil))
         (cond ((constant-string-p symbol)
           (setf all-uppercase t
            symbol (subseq symbol 1 (1- (length symbol)))))
          ((first-uppercase-p symbol)
           (setf lowercase nil
            symbol (subseq symbol 1))))
         (flet ((reschar (c)
             (push (if (and lowercase (not all-uppercase))
             (char-downcase c)
             (char-upcase c)) res)
             (setf lowercase t)))
           (dotimes (i (length symbol))
        (let ((c (char symbol i)))
          (cond
            ((eql c #\-)
             (setf lowercase (not lowercase)))
            ((assoc c *special-chars*)
             (dolist (i (coerce (cdr (assoc c *special-chars*)) 'list))
          (reschar i)))
            (t (reschar c))))))
         (coerce (nreverse res) 'string)))
      (t (string-join (mapcar #'symbol-to-js symbols) ".")))))
```

4.4 KlammerScript types

The most generic KlammerScript type is a "statement". Almost everything is a statement in JavaScript. Unlike LISP, most statements can't be used as expressions, however. The second most generic KlammerScript type is an "expression".

Sometimes we need to compare if two KlammerScript types are "the same". For example to see if two literal values are the same. In order to check this, we do not use "EQUAL", but "JS-EQUAL", which can be overloaded as needed. "JS-EQUAL" returns "T" when the semantic of the compared types is the same. Note that we only implement "JS-EQUAL" in order to reduce arithmetic expressions for now.

\index{JS-EQUAL} First we define "JS-EQUAL" on lists, and a default method which calls "EQUAL".

```
(defgeneric js-equal (obj1 obj2))
(defmethod js-equal ((obj1 list) (obj2 list))
  (and (= (length obj1) (length obj2))
       (every #'js-equal obj1 obj2)))
(defmethod js-equal ((obj1 t) (obj2 t))
  (equal obj1 obj2))
```

In order to reduce the amount of code to be written, we use a macro "DEFJSCLASS", which creates a KlammerScripttype. It also creates a "JS-EQUAL" method which calls "JS-EQUAL" on each slot of the newly created class. \index{DEFJSCLASS}

```
(defmacro defjsclass (name superclasses slots &rest class-options)
  (let ((slot-names (mapcar #'(lambda (slot) (if (atom slot) slot (first slot))) slots)))
    `(progn
      (defclass ,name ,superclasses
   ,slots ,@class-options)
      (defmethod js-equal ((obj1 ,name) (obj2 ,name))
   (every #'(lambda (slot)
         (js-equal (slot-value obj1 slot)
              (slot-value obj2 slot)))
          ',slot-names)))))
```

Finally we defined the standard types "statement" and "expression". \index{STATEMENT} \index{EXPRESSION}

```
(defjsclass statement ()
  ((value :initarg :value :accessor value :initform nil)))

(defjsclass expression (statement)
  ((value)))
```

4.5 Indenting JavaScript code

```
(defun special-append-to-last (form elt)
  (flet ((special-append (form elt)
      (let ((len (length form)))
        (if (and (> len 0)
            (member (char form (1- len))
               '(#\; #\, #\})))
       form
       (concatenate 'string form elt)))))
    (cond ((stringp form)
      (special-append form elt))
     ((consp form)
      (let ((last (last form)))
        (if (stringp (car last))
       (rplaca last (special-append (car last) elt))
       (append-to-last (car last) elt))
       form))
     (t (error "unsupported form ~S" form)))))

(defun dwim-join (value-string-lists max-length
        &key start end
             join-before join-after
```

```
    white-space (separator " ")
    (append-to-last #'append-to-last)
    (collect t))
#+nil
(format t "value-string-lists: ~S~%" value-string-lists)

(unless start
  (setf start ""))

(unless join-before
  (setf join-before ""))

;;; collect single value-string-lists until line full

(do* ((string-lists value-string-lists (cdr string-lists))
 (string-list (car string-lists) (car string-lists))
 (cur-elt start)
 (cur-empty t)
 (white-space (or white-space (make-string (length start) :initial-element #\
 (res nil))
((null string-lists)
 (unless cur-empty
   (push cur-elt res))
 (if (null res)
     (list (concatenate 'string start end))
     (progn
  (when end
    (setf (first res)
     (funcall append-to-last (first res) end)))
  (nreverse res))))

  #+nil
  (format t "string-list: ~S~%" string-list)

  (when join-after
(unless (null (cdr string-lists))
  (funcall append-to-last string-list join-after)))

  (if (and collect (= (length string-list) 1))
  (progn
    #+nil
    (format t "cur-elt: ~S line-length ~D, max-length ~D, string: ~S~%"
       cur-elt
       (+ (length (first string-list))
          (length cur-elt))
       max-length
       (first string-list))
    (if (or cur-empty
       (< (+ (length (first string-list))
```

```
            (length cur-elt)) max-length))
       (setf cur-elt
             (concatenate 'string cur-elt
                (if cur-empty "" (concatenate 'string separator join-before))
                (first string-list))
             cur-empty nil)
       (progn
         (push cur-elt res)
         (setf cur-elt (concatenate 'string white-space
                     join-before (first string-list))
          cur-empty nil))))

     (progn
       (unless cur-empty
         (push cur-elt res)
         (setf cur-elt white-space
          cur-empty t))
       (setf res (nconc (nreverse
               (cons (concatenate 'string
                   cur-elt (if (null res)
                          "" join-before)
                   (first string-list))
                (mapcar #'(lambda (x) (concatenate 'string white-space x))
                   (cdr string-list)))) res))
       (setf cur-elt white-space cur-empty t)))))

(defmethod js-to-strings ((expression expression) start-pos)
  (list (princ-to-string (value expression))))

(defmethod js-to-statement-strings ((expression expression) start-pos)
  (js-to-strings expression start-pos))

(defmethod js-to-statement-strings ((statement statement) start-pos)
  (list (princ-to-string (value statement))))
```

4.6 KlammerScript literals

```
(defmacro defjsliteral (name string)
  "Define a Javascript literal that will expand to STRING."
  `(define-js-compiler-macro ,name () (make-instance 'expression :value ,string)))

(defjsliteral this      "this")
(defjsliteral t         "true")
(defjsliteral nil       "null")
(defjsliteral false     "false")
(defjsliteral undefined "undefined")

(defmacro defjskeyword (name string)
```

```
  "Define a Javascript keyword that will expand to STRING."
  `(define-js-compiler-macro ,name () (make-instance 'statement :value ,string)))

(defjskeyword break    "break")
(defjskeyword continue "continue")
```

4.6.1 Array literals

```
(defjsclass array-literal (expression)
  ((values :initarg :values :accessor array-values)))

(define-js-compiler-macro array (&rest values)
  (make-instance 'array-literal
       :values (mapcar #'js-compile-to-expression values)))

(defjsmacro list (&rest values)
  `(array ,@values))

(defmethod js-to-strings ((array array-literal) start-pos)
  (let ((value-string-lists
    (mapcar #'(lambda (x) (js-to-strings x (+ start-pos 2)))
       (array-values array)))
   (max-length (- 80 start-pos 2)))
    (dwim-join value-string-lists max-length
          :start "[ " :end " ]"
          :join-after ",")))

(defjsclass js-aref (expression)
  ((array :initarg :array
     :accessor aref-array)
   (index :initarg :index
     :accessor aref-index)))

(define-js-compiler-macro aref (array &rest coords)
  (make-instance 'js-aref
       :array (js-compile-to-expression array)
       :index (mapcar #'js-compile-to-expression coords)))

(defmethod js-to-strings ((aref js-aref) start-pos)
  (dwim-join (cons (js-to-strings (aref-array aref) start-pos)
         (mapcar #'(lambda (x) (dwim-join (list (js-to-strings x (+ start-pos 2))
                     (- 80 start-pos 2)
                     :start "[" :end "]"))
             (aref-index aref)))
        (- 80 start-pos 2) :separator ""
        :white-space "  "))

(defjsmacro make-array (&rest inits)
  `(new (*array ,@inits)))
```

4.6.2 String literals

```
(defjsclass string-literal (expression)
  (value))

(defmethod js-to-strings ((string string-literal) start-pos)
  (declare (ignore start-pos))
  (list (prin1-to-string (value string))))
```

4.6.3 Number literals

```
(defjsclass number-literal (expression)
  (value))
```

4.6.4 KlammerScript variables

```
(defjsclass js-variable (expression)
  (value))

(defmethod js-to-strings ((v js-variable) start-form)
  (list (symbol-to-js (value v))))
```

4.7 Arithmetic operators

```
(eval-when (:compile-toplevel :load-toplevel :execute)

  (defparameter *op-precedence-hash* (make-hash-table))

  (defparameter *op-precedences*
    '((aref)
      (slot-value)
      (! not ~)
      (* / %)
      (+ -)
      (<< >>)
      (>>>)
      (< > <= >=)
      (in if)
      (eql == != =)
      (=== !==)
      (&)
      (^)
      (\|)
      (\&\& and)
      (\|\| or)
      (setf *= /= %= += -= <<= >>= >>>= \&= ^= \|=)
      (comma)))
```

```
  ;;; generate the operator precedences from *OP-PRECEDENCES*
  (let ((precedence 1))
    (dolist (ops *op-precedences*)
      (dolist (op ops)
        (setf (gethash (klammer-intern op) *op-precedence-hash*) precedence))
      (incf precedence))))

(defun js-convert-op-name (op)
  (case op
    (and '\&\&)
    (or '\|\|)
    (not '!)
    (eql '\=\=)
    (=   '\=\=)
    (t op)))

(defun lookup-op-precedence (op)
  (gethash (klammer-intern op) *op-precedence-hash*))

(defjsclass op-form (expression)
  ((operator :initarg :operator :accessor operator)
   (args :initarg :args :accessor op-args)))

(defun op-form-p (form)
  (and (listp form)
       (not (js-compiler-macro-form-p form))
       (not (null (lookup-op-precedence (first form))))))

(defun klammer (string-list)
  (prepend-to-first string-list "(")
  (append-to-last string-list ")")
  string-list)

(defmethod expression-precedence ((expression expression))
  0)

(defmethod expression-precedence ((form op-form))
  (lookup-op-precedence (operator form)))

(defmethod js-to-strings ((form op-form) start-pos)
  (let* ((precedence (expression-precedence form))
    (value-string-lists
     (mapcar #'(lambda (x)
            (let ((string-list (js-to-strings x (+ start-pos 2))))
          (if (>= (expression-precedence x) precedence)
              (klammer string-list)
              string-list)))
        (op-args form)))
    (max-length (- 80 start-pos 2))
```

```
    (op-string (format nil "~A " (operator form))))
    (dwim-join value-string-lists max-length :join-before op-string)))

(defjsmacro 1- (form)
  `(- ,form 1))

(defjsmacro 1+ (form)
  `(+ ,form 1))

(defjsclass one-op (expression)
  ((pre-p :initarg :pre-p
     :initform nil
     :accessor one-op-pre-p)
   (op :initarg :op
       :accessor one-op)))

(defmethod js-to-strings ((one-op one-op) start-pos)
  (let* ((value (value one-op))
    (value-strings (js-to-strings value start-pos)))
    (when (typep value 'op-form)
      (setf value-strings (klammer value-strings)))
    (if (one-op-pre-p one-op)
      (prepend-to-first value-strings
         (one-op one-op))
      (append-to-last value-strings
            (one-op one-op)))))

(define-js-compiler-macro incf (x)
  (make-instance 'one-op :pre-p t :op "++"
       :value (js-compile-to-expression x)))
(define-js-compiler-macro ++ (x)
  (make-instance 'one-op :pre-p nil :op "++"
       :value (js-compile-to-expression x)))
(define-js-compiler-macro decf (x)
  (make-instance 'one-op :pre-p t :op "--"
       :value (js-compile-to-expression x)))
(define-js-compiler-macro -- (x)
  (make-instance 'one-op :pre-p nil :op "--"
       :value (js-compile-to-expression x)))


(define-js-compiler-macro not (x)
  (let ((value (js-compile-to-expression x)))
    (if (and (typep value 'op-form)
        (= (length (op-args value)) 2))
   (let ((new-op (case (operator value)
         (== '!=)
         (< '>=)
         (> '<=)
```

```
          (<= '>)
          (>= '<)
          (!= '==)
          (=== '!==)
          (!== '===)
          (t nil))))
      (if new-op
          (make-instance 'op-form :operator new-op
               :args (op-args value))
          (make-instance 'one-op :pre-p t :op "!"
              :value value)))
   (make-instance 'one-op :pre-p t :op "!"
              :value value))))
```

4.8 Function calls

```
(defjsclass function-call (expression)
  ((function :initarg :function :accessor f-function)
   (args :initarg :args :accessor f-args)))

(defun funcall-form-p (form)
  (and (listp form)
       (not (op-form-p form))
       (not (js-compiler-macro-form-p form))))

(defmethod js-to-strings ((form function-call) start-pos)
  (let* ((value-string-lists
     (mapcar #'(lambda (x) (js-to-strings x (+ start-pos 2)))
        (f-args form)))
    (max-length (- 80 start-pos 2))
    (args (dwim-join value-string-lists max-length
           :start "(" :end ")" :join-after ",")))
    (dwim-join (list (js-to-strings (f-function form) (+ start-pos 2))
           args)
          max-length
          :separator "")))

(defjsclass method-call (expression)
  ((method :initarg :method :accessor m-method)
   (object :initarg :object :accessor m-object)
   (args :initarg :args :accessor m-args)))

(defmethod js-to-strings ((form method-call) start-pos)
  (let ((fname (dwim-join (list (js-to-strings (m-object form) (+ start-pos 2))
            (list (symbol-to-js (m-method form))))
           (- 80 start-pos 2)
           :end "("
           :separator "")))
```

```
    (let ((butlast (butlast fname))
     (last (car (last fname))))
      (nconc butlast
        (dwim-join (mapcar #'(lambda (x) (js-to-strings x (+ start-pos 2)))
            (m-args form))
         (- 80 start-pos 2)
         :start last
         :end ")"
         :join-after ","))))))

(defun method-call-p (form)
  (and (funcall-form-p form)
       (symbolp (first form))
       (eql (char (symbol-name (first form)) 0) #\.)))
```

4.9 Body forms

```
(defjsclass js-body (expression)
  ((stmts :initarg :stmts :accessor b-stmts)
   (indent :initarg :indent :initform "" :accessor b-indent)))

(define-js-compiler-macro progn (&rest body)
  (make-instance 'js-body
       :stmts (mapcar #'js-compile-to-statement body)))

(defmethod initialize-instance :after ((body js-body) &rest initargs)
  (declare (ignore initargs))
  (let* ((stmts (b-stmts body))
    (last (last stmts))
    (last-stmt (car last)))
    (when (typep last-stmt 'js-body)
      (setf (b-stmts body)
       (nconc (butlast stmts)
         (b-stmts last-stmt))))))


(defmethod js-to-statement-strings ((body js-body) start-pos)
  (dwim-join (mapcar #'(lambda (x) (js-to-statement-strings x (+ start-pos 2)))
           (b-stmts body))
        (- 80 start-pos 2)
        :join-after ";"
        :append-to-last #'special-append-to-last
        :start (b-indent body) :collect nil
        :end ";"))

(defmethod js-to-strings ((body js-body) start-pos)
  (dwim-join (mapcar #'(lambda (x) (js-to-strings x (+ start-pos 2)))
           (b-stmts body))
```

```
        (- 80 start-pos 2)
        :append-to-last #'special-append-to-last
        :join-after ","
        :start (b-indent body)))

(defjsclass js-sub-body (js-body)
  (stmts indent))

(defmethod js-to-statement-strings ((body js-sub-body) start-pos)
  (nconc (list "{") (call-next-method) (list "}")))

(defmethod expression-precedence ((body js-body))
  (if (= (length (b-stmts body)) 1)
      (expression-precedence (first (b-stmts body)))
      (lookup-op-precedence 'comma)))
```

4.10 Function definition

```
(defjsclass js-defun (expression)
  ((name :initarg :name :accessor d-name)
   (args :initarg :args :accessor d-args)
   (body :initarg :body :accessor d-body)))

(define-js-compiler-macro defun (name args &rest body)
  (make-instance 'js-defun
       :name (js-compile-to-symbol name)
       :args (mapcar #'js-compile-to-symbol args)
       :body (make-instance 'js-body
                  :indent "  "
                  :stmts (mapcar #'js-compile-to-statement body))))

(defmethod js-to-strings ((defun js-defun) start-pos)
  (let ((fun-header (dwim-join (mapcar #'(lambda (x) (list (symbol-to-js x)))
                   (d-args defun))
                (- 80 start-pos 2)
                :start (format nil "function ~A("
                     (symbol-to-js (d-name defun)))
                :end ") {" :join-after ","))
   (fun-body (js-to-statement-strings (d-body defun) (+ start-pos 2))))
    (nconc fun-header fun-body (list "}"))))

(defmethod js-to-statement-strings ((defun js-defun) start-pos)
  (js-to-strings defun start-pos))

(defjsmacro flet (fdefs &rest body)
  `(progn ,@(mapcar #'(lambda (fdef)
                        (cons 'defun fdef))
                    fdefs)
```

```
          ,@body))

(defjsmacro lambda (args &rest body)
  `(defun :|| ,args ,@body))
```

4.11 Object creation

```
(defjsclass js-object (expression)
  ((slots :initarg :slots
     :accessor o-slots)))

(define-js-compiler-macro create (&rest args)
  (make-instance 'js-object
       :slots (loop for (name val) on args by #'cddr
                collect (list (js-compile-to-symbol name)
                    (js-compile-to-expression val)))))

(defmethod js-to-strings ((object js-object) start-pos)
  (let ((value-string-lists
    (mapcar #'(lambda (slot)
           (dwim-join (list (js-to-strings (second slot) (+ start-pos 4)))
            (- 80 start-pos 2)
            :start (concatenate 'string (symbol-to-js (first slot)) " : ")
            :white-space "    ")) (o-slots object)))
   (max-length (- 80 start-pos 2)))
    (dwim-join value-string-lists max-length
          :start "{ "
          :end " }"
          :join-after ", "
          :white-space "  "
          :collect nil)))

(defjsclass js-slot-value (expression)
  ((object :initarg :object
      :accessor sv-object)
   (slot :initarg :slot
    :accessor sv-slot)))

(define-js-compiler-macro slot-value (obj slot)
  (make-instance 'js-slot-value :object (js-compile-to-expression obj)
       :slot (js-compile-to-symbol slot)))

(defmethod js-to-strings ((sv js-slot-value) start-pos)
  (append-to-last (js-to-strings (sv-object sv) start-pos)
        (format nil ".~A" (symbol-to-js (sv-slot sv)))))

(defjsmacro with-slots (slots object &rest body)
  `(symbol-macrolet ,(mapcar #'(lambda (slot)
```

```
              `(,slot '(slot-value ,object ',slot)))
               slots)
     ,@body))
```

4.12 Macros

```
(define-js-compiler-macro macrolet (macros &rest body)
  (let* ((macro-env (make-hash-table))
    (*js-macro-env* (cons macro-env *js-macro-env*)))
    (dolist (macro macros)
      (destructuring-bind (name arglist &rest body) macro
   (setf (gethash name macro-env)
         (compile nil `(lambda ,arglist ,@body)))))
    (js-compile `(progn ,@body))))

(defjsmacro symbol-macrolet (macros &rest body)
  `(macrolet ,(mapcar #'(lambda (macro)
          `(,(first macro) () ,@(rest macro))) macros)
    ,@body))
```

4.13 LISP evaluation

```
(defjsmacro lisp (&rest forms)
  (eval (cons 'progn forms)))
```

4.14 Return

Return takes an optional value, so it has to be handled differently than the misc operators below.

```
(defjsclass js-return (statement) (value))
(define-js-compiler-macro return (&optional value)
  (make-instance 'js-return :value (when value
                                     (js-compile-to-expression value))))
(defmethod js-to-statement-strings ((return js-return) start-pos)
  (dwim-join
   (list (if (value return)
             (js-to-strings (value return) (+ start-pos 2))
             (list "")))
   (- 80 start-pos 2)
   :start (if (value return)
              "return "
              "return")
   :white-space "  "))
```

4.15 Miscellaneous expressions and statements

```
(defmacro define-js-single-op (name &optional (superclass 'expression))
  (let ((js-name (klammer-intern
                  (concatenate 'string "JS-" (symbol-name name)))))
  `(progn
    (defjsclass ,js-name (,superclass)
      (value))
    (define-js-compiler-macro ,name (value)
      (make-instance ',js-name :value (js-compile-to-expression value)))
    (defmethod ,(if (eql superclass 'expression)
          'js-to-strings
          'js-to-statement-strings) ((,name ,js-name) start-pos)
      (dwim-join (list (js-to-strings (value ,name) (+ start-pos 2)))
       (- 80 start-pos 2)
       :start ,(concatenate 'string (string-downcase (symbol-name name)) " ")
       :white-space "  ")))))


(define-js-single-op throw statement)
(define-js-single-op delete)
(define-js-single-op void)
(define-js-single-op typeof)
(define-js-single-op instanceof)
(define-js-single-op new)
```

4.16 Assignment

```
(defjsclass js-setf (expression)
  ((lhs :initarg :lhs :accessor setf-lhs)
   (rhsides :initarg :rhsides :accessor setf-rhsides)))

(defun assignment-op (op)
  (case op
    (+ '+=)
    (~ '~=)
    (\& '\&=)
    (\| '\|=)
    (- '-=)
    (* '*=)
    (% '%=)
    (>> '>>=)
    (^  '^=)
    (<< '<<=)
    (>>> '>>>=)
    (/   '/=)
    (t   nil)))
```

```
(defun make-js-test (lhs rhs)
  (if (and (typep rhs 'op-form)
      (member lhs (op-args rhs) :test #'js-equal))
      (let ((args-without (remove lhs (op-args rhs)
              :count 1 :test #'js-equal))
       (args-without-first (remove lhs (op-args rhs)
              :count 1 :end 1
              :test #'js-equal))
       (one (list (make-instance 'number-literal :value 1))))
   #+nil
   (format t "OPERATOR: ~S, ARGS-WITHOUT: ~S, ARGS-WITHOUT-FIRST ~S~%"
      (operator rhs)
      args-without
      args-without-first)
   (cond ((and (js-equal args-without one)
          (eql (operator rhs) '+))
          (make-instance 'one-op :pre-p nil :op "++"
              :value lhs))
         ((and (js-equal args-without-first one)
          (eql (operator rhs) '-))
          (make-instance 'one-op :pre-p nil :op "--"
              :value lhs))
         ((and (assignment-op (operator rhs))
          (member (operator rhs)
             '(+ *)))
          (make-instance 'op-form
              :operator (assignment-op (operator rhs))
              :args (list lhs (make-instance 'op-form
                         :operator (operator rhs)
                         :args args-without))))
         ((and (assignment-op (operator rhs))
          (js-equal (first (op-args rhs)) lhs))
          (make-instance 'op-form
              :operator (assignment-op (operator rhs))
              :args (list lhs (make-instance 'op-form
                         :operator (operator rhs)
                         :args (cdr (op-args rhs))))))
         (t (make-instance 'js-setf :lhs lhs :rhsides (list rhs)))))
      (make-instance 'js-setf :lhs lhs :rhsides (list rhs))))

(define-js-compiler-macro setf (&rest args)
  (let ((assignments (loop for (lhs rhs) on args by #'cddr
            for rexpr = (js-compile-to-expression rhs)
            for lexpr = (js-compile-to-expression lhs)
            collect (make-js-test lexpr rexpr))))
    (if (= (length assignments) 1)
   (first assignments)
   (make-instance 'js-body :indent "" :stmts assignments))))
```

```
(defmethod js-to-strings ((setf js-setf) start-pos)
  (dwim-join (cons (js-to-strings (setf-lhs setf) start-pos)
         (mapcar #'(lambda (x) (js-to-strings x start-pos)) (setf-rhsides setf)))
        (- 80 start-pos 2)
        :join-after " ="))

(defmethod expression-precedence ((setf js-setf))
  (lookup-op-precedence '=))
```

4.17 Variable definition

```
(defjsclass js-defvar (statement)
  ((names :initarg :names :accessor var-names)
   (value :initarg :value :accessor var-value)))

(define-js-compiler-macro defvar (name &optional value)
  (make-instance 'js-defvar :names (list (js-compile-to-symbol name))
       :value (when value (js-compile-to-expression value))))

(defmethod js-to-statement-strings ((defvar js-defvar) start-pos)
  (dwim-join (nconc (mapcar #'(lambda (x) (list (symbol-to-js x))) (var-names defvar))
          (when (var-value defvar)
            (list (js-to-strings (var-value defvar) start-pos))))
        (- 80 start-pos 2)
        :join-after " ="
        :start "var " :end ";"))
```

4.18 Variable binding

```
(define-js-compiler-macro let (decls &rest body)
  (let ((single-defvar (make-instance 'js-defvar
                 :names (mapcar #'js-compile-to-symbol
                      (remove-if-not #'atom decls))
                 :value nil))
   (defvars (mapcar #'(lambda (decl)
              (let ((name (first decl))
               (value (second decl)))
              (make-instance 'js-defvar
                   :names (list (js-compile-to-symbol name))
                   :value (js-compile-to-expression value))))
          (remove-if #'atom decls))))
    (make-instance 'js-sub-body
         :indent "  "
         :stmts (nconc (when (var-names single-defvar) (list single-defvar))
             defvars
             (mapcar #'js-compile-to-statement body)))))
```

4.19 Control structures

4.19.1 IF

```
(defjsclass js-if (expression)
  ((test :initarg :test
    :accessor if-test)
   (then :initarg :then
    :accessor if-then)
   (else :initarg :else
    :accessor if-else)))

(define-js-compiler-macro if (test then &optional else)
  (make-instance 'js-if :test (js-compile-to-expression test)
       :then (js-compile-to-body then :indent "  ")
       :else (when else
          (js-compile-to-body else :indent "  "))))

(defmethod initialize-instance :after ((if js-if) &rest initargs)
  (declare (ignore initargs))
  (when (and (if-then if)
        (typep (if-then if) 'js-sub-body))
    (change-class (if-then if) 'js-body))
  (when (and (if-else if)
        (typep (if-else if) 'js-sub-body))
    (change-class (if-else if) 'js-body)))

(defmethod js-to-statement-strings ((if js-if) start-pos)
  (let ((if-strings (dwim-join (list (js-to-strings (if-test if) 0))
                (- 80 start-pos 2)
                :start "if ("
                :end ") {"))
   (then-strings (js-to-statement-strings (if-then if) (+ start-pos 2)))
   (else-strings (when (if-else if)
         (js-to-statement-strings (if-else if)
                  (+ start-pos 2)))))
    (nconc if-strings then-strings (if else-strings
                (nconc (list "} else {") else-strings (list "}"))
                (list "}")))))

(defmethod expression-precedence ((if js-if))
  (lookup-op-precedence 'if))

(defmethod js-to-strings ((if js-if) start-pos)
  (assert (typep (if-then if) 'expression))
  (when (if-else if)
    (assert (typep (if-else if) 'expression)))
  (dwim-join (list (append-to-last (js-to-strings (if-test if) start-pos) " ?")
        (let* ((new-then (make-instance 'js-body
```

```
                    :stmts (b-stmts (if-then if))
                    :indent ""))
          (res (js-to-strings new-then start-pos)))
          (if (>= (expression-precedence (if-then if))
             (expression-precedence if))
             (klammer res)
             res))
        (list ":")
        (if (if-else if)
            (let* ((new-else (make-instance 'js-body
                        :stmts (b-stmts (if-else if))
                        :indent ""))
                   (res (js-to-strings new-else start-pos)))
             (if (>= (expression-precedence (if-else if))
                (expression-precedence if))
                (klammer res)
                res))
            (list "undefined")))
       (- 80 start-pos 2)
       :white-space "  "))

(defjsmacro when (test &rest body)
  `(if ,test (progn ,@body)))

(defjsmacro unless (test &rest body)
  `(if (not ,test) (progn ,@body)))

(defjsmacro cond (&rest tests)
  (multiple-value-bind (otherwise tests)
      (split-list tests #'(lambda (x) (eql x t)) :key #'car)
    (labels ((make-test (tests)
               (if tests
                   (let ((first (first tests)))
                     `(if ,(car first)
                          (progn ,@(cdr first))
                          ,(make-test (cdr tests))))
                   `(progn ,@(cdar otherwise)))))
      (make-test tests))))
```

4.19.2 Iteration constructs

```
(defjsclass js-for (statement)
  ((vars :initarg :vars :accessor for-vars)
   (steps :initarg :steps :accessor for-steps)
   (check :initarg :check :accessor for-check)
   (body :initarg :body :accessor for-body)))

(defun make-for-vars (decls)
  (loop for decl in decls
```

```
   for var = (if (atom decl) decl (first decl))
   for init = (if (atom decl) nil (second decl))
   collect (make-instance 'js-defvar :names (list (js-compile-to-symbol var))
                :value (js-compile-to-expression init))))

(defun make-for-steps (decls)
  (loop for decl in decls
   when (= (length decl) 3)
   collect (js-compile-to-expression `(setf ,(first decl) ,(third decl)))))

(define-js-compiler-macro do (decls termination &rest body)
  (let ((vars (make-for-vars decls))
   (steps (make-for-steps decls))
   (check (js-compile-to-expression (list 'not (first termination))))
   (body (js-compile-to-body (cons 'progn body) :indent "  ")))
    (make-instance 'js-for
         :vars vars
         :steps steps
         :check check
         :body body)))

(defjsmacro dotimes (iter &rest body)
  (let ((var (first iter))
        (times (second iter)))
  `(do ((,var 0 (++ ,var)))
       ((= ,var ,times))
     ,@body)))

(defjsmacro dolist (i-array &rest body)
  (let ((var (first i-array))
   (array (second i-array))
   (arrvar (js-gensym "arr"))
   (idx (js-gensym "i")))
    `(let ((,arrvar ,array))
      (do ((,idx 0 (++ ,idx)))
     ((>= ,idx (slot-value ,arrvar 'length)))
   (let ((,var (aref ,arrvar ,idx)))
     ,@body)))))

(defmethod js-to-statement-strings ((for js-for) start-pos)
  (let* ((init (dwim-join (mapcar #'(lambda (x)
                 (dwim-join (list (list (symbol-to-js (first (var-names x))))
                        (js-to-strings (var-value x)
                             (+ start-pos 2)))
                  (- 80 start-pos 2)
                  :join-after " ="))
              (for-vars for))
           (- 80 start-pos 2)
           :start "var " :join-after ",")
```

```
    (check (js-to-strings (for-check for) (+ start-pos 2)))
    (steps (dwim-join (mapcar #'(lambda (x)
                                  (js-to-strings x (- start-pos 2)))
                              (for-steps for))
                      (- 80 start-pos 2)
                      :join-after ","))
           (header (dwim-join (list init check steps)
                              (- 80 start-pos 2)
                              :start "for (" :end ") {"
                              :join-after ";"))
    (body (js-to-statement-strings (for-body for) (+ start-pos 2))))
    (nconc header body (list "}"))))

(defjsclass for-each (statement)
  ((name :initarg :name :accessor fe-name)
   (value :initarg :value :accessor fe-value)
   (body :initarg :body :accessor fe-body)))

(define-js-compiler-macro doeach (decl &rest body)
  (make-instance 'for-each :name (js-compile-to-symbol (first decl))
       :value (js-compile-to-expression (second decl))
       :body (js-compile-to-body (cons 'progn body) :indent "  ")))

(defmethod js-to-statement-strings ((fe for-each) start-pos)
  (let ((header (dwim-join (list (list (symbol-to-js (fe-name fe)))
                                 (list "in")
                                 (js-to-strings (fe-value fe) (+ start-pos 2)))
                           (- 80 start-pos 2)
                           :start "for (var "
                           :end ") {"))
   (body (js-to-statement-strings (fe-body fe) (+ start-pos 2))))
    (nconc header body (list "}"))))

(defjsclass js-while (statement)
  ((check :initarg :check :accessor while-check)
   (body :initarg :body :accessor while-body)))

(define-js-compiler-macro while (check &rest body)
  (make-instance 'js-while
       :check (js-compile-to-expression check)
       :body (js-compile-to-body (cons 'progn body) :indent "  ")))

(defmethod js-to-statement-strings ((while js-while) start-pos)
  (let ((header (dwim-join (list (js-to-strings (while-check while) (+ start-pos 2)))
                           (- 80 start-pos 2)
                           :start "while ("
                           :end ") {"))
   (body (js-to-statement-strings (while-body while) (+ start-pos 2))))
    (nconc header body (list "}"))))
```

4.19.3 The CASE construct

```
(defjsclass js-case (statement)
  ((value :initarg :value :accessor case-value)
   (clauses :initarg :clauses :accessor case-clauses)))

(defun compile-case-val (val)
  (flet ((compile-single (val)
           (cond ((and (symbolp val)
                       (string= (symbol-name val) "DEFAULT"))
                  'default)
                 ((atom val)
                  (js-compile-to-expression val))
                 (t (error "not single elt ~A" val)))))
    (if (listp val)
        (mapcar #'compile-single val)
        (list (compile-single val)))))

(define-js-compiler-macro case (value &rest clauses)
  (let ((clauses (mapcar #'(lambda (clause)
             (let ((val (first clause))
              (body (cdr clause)))
               (list (compile-case-val val)
                (js-compile-to-body `(progn ,@body break) :indent "  "))))
          clauses))
   (check (js-compile-to-expression value)))
    (make-instance 'js-case :value check
         :clauses clauses)))

(defmethod js-to-statement-strings ((case js-case) start-pos)
  (let ((body (mapcan
               #'(lambda (clause)
                   (let ((vals (car clause))
                         (body (second clause)))
                     (append
                      (mapcan
                       #'(lambda (val)
                           (dwim-join (list (if (equal val 'default)
                                                (list "")
                                                (js-to-strings val (+ start-pos 2
                                      (- 80 start-pos 2)
                                      :start (if (eql val 'default) "  default" "
                                      :white-space "   "
                                      :end ":")) vals)
                      (js-to-statement-strings body (+ start-pos 2)))))
               (case-clauses case))))

    #+nil
    (format t "body: ~S~%" body)
    (nconc (dwim-join (list (js-to-strings (case-value case) (+ start-pos 2)))
```

```
          (- 80 start-pos 2)
          :start "switch (" :end ") {")
      body
      (list "}"))))
```

4.20 The WITH construct

```
(defjsclass js-with (statement)
  ((obj :initarg :obj :accessor with-obj)
   (body :initarg :body :accessor with-body)))

(define-js-compiler-macro with (statement &rest body)
  (make-instance 'js-with
       :obj (js-compile-to-expression (first statement))
       :body (js-compile-to-body (cons 'progn body) :indent "  ")))

(defmethod js-to-statement-strings ((with js-with) start-pos)
  (nconc (dwim-join (list (js-to-strings (with-obj with) (+ start-pos 2)))
          (- 80 start-pos 2)
          :start "with (" :end ") {")
    (js-to-statement-strings (with-body with) (+ start-pos 2))
    (list "}")))
```

4.21 Exceptions

```
(defjsclass js-try (statement)
  ((body :initarg :body :accessor try-body)
   (catch :initarg :catch :accessor try-catch)
   (finally :initarg :finally :accessor try-finally)))

(define-js-compiler-macro try (body &rest clauses)
  (let ((body (js-compile-to-body body :indent "  "))
   (catch (cdr (assoc :catch clauses)))
   (finally (cdr (assoc :finally clauses))))
    (make-instance 'js-try
         :body body
         :catch (when catch (list (js-compile-to-symbol (caar catch))
                   (js-compile-to-body (cons 'progn (cdr catch))
                        :indent "  ")))
         :finally (when finally (js-compile-to-body (cons 'progn finally)
                          :indent "  "))))))

(defmethod js-to-statement-strings ((try js-try) start-pos)
  (let* ((catch (try-catch try))
    (finally (try-finally try))
    (catch-list (when catch
             (nconc
```

```
          (dwim-join (list (list (symbol-to-js (first catch))))
                (- 80 start-pos 2)
                :start "} catch ("
                :end ") {")
          (js-to-statement-strings (second catch) (+ start-pos 2)))))
    (finally-list (when finally
          (nconc (list "} finally {")
            (js-to-statement-strings finally (+ start-pos 2))))))
    (nconc (list "try {")
      (js-to-statement-strings (try-body try) (+ start-pos 2))
      catch-list
      finally-list
      (list "}"))))
```

4.22 Regular Expressions

```
(defjsclass regex (expression)
  (value))

(define-js-compiler-macro regex (regex)
  (make-instance 'regex :value (string regex)))
```

4.23 Conditional compilation

```
(defjsclass cc-if ()
  ((test :initarg :test :accessor cc-if-test)
   (body :initarg :body :accessor cc-if-body)))

(defmethod js-to-statement-strings ((cc cc-if) start-pos)
  (nconc (list (format nil "/*@if ~A" (cc-if-test cc)))
    (mapcan #'(lambda (x) (js-to-strings x start-pos)) (cc-if-body cc))
    (list "@end @*/")))

(define-js-compiler-macro cc-if (test &rest body)
  (make-instance 'cc-if :test test
       :body (mapcar #'js-compile body)))
```

4.24 The Math library

```
(defjsmacro floor (expr)
  `(*Math.floor ,expr))

(defjsmacro random ()
  `(*Math.random))
```

4.25 XMLHttpRequest

```
(defjsmacro xml-http-request ()
  '*X-M-L-Http-Request)

(defjsmacro make-xml-http-request ()
  `(if (slot-value window xml-http-request)
       (new (xml-http-request))
       (new (*Active-X-Object "Microsoft.XMLHTTP"))))
```

4.26 The compiler API

4.26.1 Direct compiler interface

```
(defun js-compile (form)
  (setf form (js-expand-form form))
  (cond ((stringp form)
    (make-instance 'string-literal :value form))
   ((numberp form)
    (make-instance 'number-literal :value form))
   ((symbolp form)
    (let ((c-macro (js-get-compiler-macro form)))
      (if c-macro
          (funcall c-macro)
          (make-instance 'js-variable :value form))))
   ((and (consp form)
         (eql (first form) 'quote))
    (second form))
   ((consp form)
    (js-compile-list form))
   (t (error "Unknown atomar expression ~S" form))))

(defun js-compile-list (form)
  (let* ((name (car form))
    (args (cdr form))
    (js-form (js-get-compiler-macro name)))
    (cond (js-form
      (apply js-form args))

     ((op-form-p form)
      (make-instance 'op-form
           :operator (js-convert-op-name (js-compile-to-symbol (first form)))
           :args (mapcar #'js-compile-to-expression (rest form))))

     ((method-call-p form)
      (make-instance 'method-call
           :method (js-compile-to-symbol (first form))
           :object (js-compile-to-expression (second form))
```

```
            :args (mapcar #'js-compile-to-expression (cddr form))))

     ((funcall-form-p form)
      (make-instance 'function-call
           :function (js-compile-to-expression (first form))
           :args (mapcar #'js-compile-to-expression (rest form))))

     (t (error "Unknown form ~S" form)))))

(defun js-compile-to-expression (form)
  (let ((res (js-compile form)))
    (assert (typep res 'expression))
    res))

(defun js-compile-to-symbol (form)
  (let ((res (js-compile form)))
    (when (typep res 'js-variable )
      (setf res (value res)))
    (assert (symbolp res))
    res))

(defun js-compile-to-statement (form)
  (let ((res (js-compile form)))
    (assert (typep res 'statement))
    res))

(defun js-compile-to-body (form &key (indent ""))
  (let ((res (js-compile-to-statement form)))
    (if (typep res 'js-body)
   (progn (setf (b-indent res) indent)
          res)
   (make-instance 'js-body
              :indent indent
              :stmts (list res)))))
```

4.26.2 Compiler helper macros

```
(define-js-compiler-macro js (&rest body)
  (make-instance 'string-literal
       :value (string-join (js-to-statement-strings
                  (js-compile (cons 'progn body)) 0) " ")))

(define-js-compiler-macro js-inline (&rest body)
  (make-instance 'string-literal
       :value (concatenate
          'string
          "javascript:"
          (string-join (js-to-statement-strings
                   (js-compile (cons 'progn body)) 0) " "))))
```

```
(defmacro js (&rest body)
  `(string-join
    (js-to-statement-strings (js-compile '(progn ,@body)) 0)
    (string #\Newline)))

(defun js-to-string (expr)
  (string-join
   (js-to-statement-strings (js-compile expr) 0)
   (string #\Newline)))

(defun js-to-line (expr)
  (string-join
   (js-to-statement-strings (js-compile expr) 0) " "))

(defmacro js-file (&rest body)
  `(html
    (:princ
     (js ,@body))))

(defmacro js-script (&rest body)
  `((:script :type "text/javascript")
    (:princ (format nil "~%// <![CDATA[~%"))
    (:princ (js ,@body))
    (:princ (format nil "~%// ]]>~%"))))

(defmacro js-inline (&rest body)
  `(concatenate 'string "javascript:"
    (string-join (js-to-statement-strings (js-compile '(progn
    ,@body)) 0) " ")))
```

Index