introduction

July 11, 2006

Contents

1 Introduction

ParenScript 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 ParenScript "compiler". The ParenScript "compiler" transforms the expression in ParenScript 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.

2 Features

ParenScript 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);
}
"
```

ParenScript uses the Lisp reader, allowing for reader macros. It also comes with its own macro environment, allowing host macros and ParenScript to coexist without interfering with each other. Furthermore, ParenScript 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 ParenScript macro:

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

ParenScript 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' }
"
```

ParenScript 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 ParenScript 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, ParenScript 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>
```

3 Getting ParenScript

ParenScript can be obtained from the BKNR subversion repository at

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

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

```
http://bknr.net/parenscript
```

or using asdf-install.

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

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

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

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

```
manuel@bknr.net
```

4 ParenScript Tutorial

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

5 Setting up the ParenScript environment

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

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

; ... lots of compiler output ...

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

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

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

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

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

The next command starts the webserver on the port 8000.

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

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

6 A simple embedded example

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

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

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

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

```
(defun tutorial1 (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript tutorial: 1st example"))
    (:body (:h1 "ParenScript tutorial: 1st example")
           (:p "Please click the link below." :br
```

```
                ((:a :href "#" :onclick (js-inline
                                         (alert "Hello World")))
                 "Hello World"))))))
```

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

```
<html><head><title>ParenScript tutorial: 1st example</title>
</head>
<body><h1>ParenScript tutorial: 1st example</h1>
<p>Please click the link below.<br/>
<a href="#"
   onclick="javascript:alert(&quot;Hello World&quot;);">Hello World</a>
</p>
</body>
</html>
```

7 Adding an inline ParenScript

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

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

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

```
<html><head><title>ParenScript tutorial: 2nd example</title>
<script type="text/javascript">
// <![CDATA[
```

```
function greetingCallback() {
  alert("Hello World");
}
// ]]>
</script>
</head>
<body><h1>ParenScript tutorial: 2nd example</h1>
<p>Please click the link below.<br/>
<a href="#"
   onclick="javascript:greetingCallback();">Hello World</a>
<br/>
And maybe this link too.<br/>
<a href="#"
   onclick="javascript:greetingCallback();">Knock knock</a>
<br/>

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

8 Generating a JavaScript file

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

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

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

(defun tutorial1 (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head
     (:title "ParenScript tutorial: 3rd example")
     ((:script :language "JavaScript" :src "/tutorial1.js")))
    (:body
     (:h1 "ParenScript tutorial: 3rd example")
```

```
      (:p "Please click the link below." :br
          ((:a :href "#" :onclick (js-inline (greeting-callback)))
           "Hello World")
          :br "And maybe this link too." :br
          ((:a :href "#" :onclick (js-inline (greeting-callback)))
           "Knock knock")
          :br "And finally a third link." :br
          ((:a :href "#" :onclick (js-inline (greeting-callback)))
           "Hello there"))))))
```

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

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

and the following HTML code:

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

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

9 A ParenScript slideshow

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

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

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

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

```
                          (with-http-body (req ent)
                            (slideshow req ent)))))

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

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

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

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

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

```
(defun slideshow (req ent)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript slideshow")
           ((:script :language "JavaScript"
                     :src "/slideshow.js"))
           (js-script
            (defvar *linkornot* 0)
            (defvar photos (array "photo1.png"
                                  "photo2.png"
                                  "photo3.png"))))
    (:body (:h1 "ParenScript slideshow")
         (:body (:h2 "Hello")
                ((:table :border 0
                         :cellspacing 0
                         :cellpadding 0)
                 (:tr ((:td :width "100%" :colspan 2 :height 22)
           (:center
            (js-script
             (let ((img
```

```
                    (html
                     ((:img :src (aref photos 0)
                            :name "photoslider"
                            :style ( + "filter:"
                                       (js (reveal-trans
                                            (setf duration 2)
                                            (setf transition 23))))
                               :border 0)))))
                (document.write
                 (if (= *linkornot* 1)
                     (html ((:a :href "#"
                                :onclick (js-inline (transport)))
                            img))
                     img)))))))
                  (:tr ((:td :width "50%" :height "21")
                        ((:p :align "left")
                         ((:a :href "#"
                              :onclick (js-inline (backward)
                                                  (return false)))
                          "Previous Slide")))
                       ((:td :width "50%" :height "21")
                        ((:p :align "right")
                         ((:a :href "#"
                              :onclick (js-inline (forward)
                                                  (return false)))
                          "Next Slide")))))))))))
```

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

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

```
// ]]>
</script>
</center>
</td>
</tr>
<tr><td width="50%" height="21"><p align="left">
<a href="#"
   onclick="javascript:backward(); return false;">Previous Slide</a>

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

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

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

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

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

   (defvar *which* 0)
```

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

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

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

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

JS-SLIDESHOW generates the following JavaScript code:

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

```
    applyEffect();
    document.images.photoslider.src = photos[WHICH];
    playEffect();
    keepTrack();
  }
}
function forward() {
  if (WHICH < photos.length - 1) {
    ++WHICH;
    applyEffect();
    document.images.photoslider.src = photos[WHICH];
    playEffect();
    keepTrack();
  }
}
function transport() {
  window.location = photoslink[WHICH];
}
```

10 Customizing the slideshow

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

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

```
(defun slideshow2 (req ent image-urls)
  (declare (ignore req ent))
  (html
   (:html
    (:head (:title "ParenScript slideshow")
           ((:script :language "JavaScript"
                     :src "/slideshow.js"))
           ((:script :type "text/javascript")
            (:princ (format nil "~%// <![CDATA[~%"))
            (:princ (js (defvar *linkornot* 0)))
            (:princ (js-to-string `(defvar photos
                                     (array ,@image-urls))))
            (:princ (format nil "~%// ]]>~%"))))
    (:body (:h1 "ParenScript slideshow")
         (:body (:h2 "Hello")
                ((:table :border 0
                         :cellspacing 0
                         :cellpadding 0)
                 (:tr ((:td :width "100%" :colspan 2 :height 22)
         (:center
          (js-script
           (let ((img
                  (html
                   ((:img :src (aref photos 0)
                          :name "photoslider"
                          :style ( + "filter:"
                                     (js (reveal-trans
                                          (setf duration 2)
                                          (setf transition 23))))
                          :border 0)))))
                (document.write
                 (if (= *linkornot* 1)
                     (html ((:a :href "#"
                                :onclick (js-inline (transport)))
                            img))
                     img)))))))
                 (:tr ((:td :width "50%" :height "21")
                       ((:p :align "left")
                        ((:a :href "#"
                             :onclick (js-inline (backward)
                                                 (return false)))
                         "Previous Slide")))
                      ((:td :width "50%" :height "21")
                       ((:p :align "right")
                        ((:a :href "#"
                             :onclick (js-inline (forward)
                                                 (return false)))
                         "Next Slide")))))))))))
```

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

```
(publish-slideshow "/bknr/"
  `("/home/manuel/bknr-sputnik.png"
```

```
    "/home/manuel/bknrlogo_red648.png"
    "/home/manuel/screenshots/screenshot-14.03.2005-11.54.33.png"))
```

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

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

11 ParenScript Language Reference

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

12 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 ParenScript forms are expression, but certain special forms are not (the forms which are transformed to a JavaScript statement). All ParenScript 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;
      }
```

13 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", "at", "percent", "slash", "start" and "plus" respectively. The `$` character is untouched.

```
!?#@% => bangwhathashatpercent
```

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 ParenScript 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
```

13.1 Reserved Keywords

The following keywords and symbols are reserved in ParenScript, 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
```

14 Literal values

14.1 Number literals

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

ParenScript 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
```

14.2 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.

14.3 Array literals

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

14.4 Object literals

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

14.5 Regular Expression literals

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

Regular expressions can be created by using the `REGEX` form. If the argument does not start with a slash, it is surrounded by slashes to make it a proper JavaScript regex. If the argument starts with a slash it is left as it is. This makes it possible to use modifiers such as slash-i (case-insensitive) or slash-g (match-globally (all)).

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

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

Here CL-INTERPOL proves really useful.

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

14.6 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
```

15 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 ParenScript programmer to be flexible, as flexible as JavaScript itself.

```
variable    => variable

a-variable  => aVariable

*math       => Math

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

16 Function calls and method calls

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

17 Operator Expressions

```
; (operator {argument}*)
; (single-operator argument)
;
; operator ::= one of *, /, %, +, -, <<, >>, >>>, < >, EQL,
;              ==, !=, =, ===, !==, &, ^, |, &&, AND, ||, OR.
; single-operator ::= one of INCF, DECF, ++, --, NOT, !
; argument ::= a ParenScript 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 = ParenScript operator is not the assignment operator. Unlike JavaScript, ParenScript 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
```

18 Body forms

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

19 Function Definition

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

20 Assignment

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

21 Single argument statements

```
; (RETURN {value}?)
; (THROW  {value}?)
;
; value ::= a ParenScript 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'
```

22 Single argument expression

```
; (DELETE     {value})
; (VOID       {value})
; (TYPEOF     {value})
; (INSTANCEOF {value})
; (NEW        {value})
;
; value ::= a ParenScript 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 ParenScript 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');
     }
```

23 Conditional Statements

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

24 Variable declaration

```
; (DEFVAR var {value}?)
; (LET ({var | (var value)) body)
;
; var   ::= a Lisp symbol
; value ::= a ParenScript expression
; body  ::= a list of ParenScript 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.

25 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 ParenScript expression resulting in a number
; list-form    ::= a ParenScript expression resulting in an array
; object       ::= a ParenScript expression resulting in an object
; init         ::= a ParenScript expression
; step         ::= a ParenScript expression
; end-test     ::= a ParenScript expression
; body         ::= a list of ParenScript statements
```

The `DO` form, which is similar to its Lisp form, is transformed into a JavaScript `for` statement. Note that the ParenScript `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 + 1) {
       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 + 1) {
         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);
      }
```

26 The 'CASE' statement

```
; (CASE case-value clause*)
;
; clause     ::= (value body) | ((value*) body) | t-clause
; case-value ::= a ParenScript expression
; value      ::= a ParenScript expression
; t-clause   ::= {t | otherwise | default} body
; body       ::= a list of ParenScript statements
```

The Lisp `CASE` form is transformed to a `switch` statement in JavaScript. Note that `CASE` is not an expression in ParenScript.

```
(case (aref blorg i)
  ((1 "one") (alert "one"))
  (2 (alert "two"))
  (t (alert "default clause")))
    => switch (blorg[i]) {
         case 1:   ;
         case 'one':
                   alert('one');
                   break;
         case 2:
                   alert('two');
                   break;
         default:   alert('default clause');
       }

; (SWITCH case-value clause*)
; clause     ::= (value body) | (default body)
```

The `SWITCH` form is the equivalent to a javascript switch statement. No break statements are inserted, and the default case is named `DEFAULT`. The `CASE` form should be prefered in most cases.

```
(switch (aref blorg i)
  (1 (alert "If I get here"))
  (2 (alert "I also get here"))
```

```
(default (alert "I always get here")))
  => switch (blorg[i]) {
       case 1:   alert('If I get here');
       case 2:   alert('I also get here');
       default:   alert('I always get here');
     }
```

27 The 'WITH' statement

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

28 The 'TRY' statement

```
; (TRY body {(:CATCH (var) body)}? {(:FINALLY body)}?)
;
; body ::= a list of ParenScript 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');
      }
```

29 The HTML Generator

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

The HTML generator of ParenScript 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 ParenScript 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>')

; (CSS-INLINE css-expression)
```

Stylesheets can also be created in ParenScript.

```
(css-inline :color "red"
            :font-size "x-small")
  => 'color:red;font-size:x-small'

(defun make-color-div(color-name)
    (return (html ((:div :style (css-inline :color color-name))
                   color-name " looks like this."))))
  => function makeColorDiv(colorName) {
       return '<div style=\"' + ('color:' + colorName) + '\">' + colorName
         + ' looks like this.</div>';
     }
```

30 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 ParenScript code
; body        ::= a list of ParenScript statements
; string      ::= a string
```

ParenScript can be extended using macros, just like Lisp can be extended using Lisp macros. Using the special Lisp form DEFJSMACRO, the ParenScript language can be extended. DEFJSMACRO adds the new macro to the toplevel macro environment, which is always accessible during ParenScript 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 ParenScript macro example is the implementation of the DOLIST form (note how JS-GENSYM, the ParenScript of GENSYM, is used to generate new ParenScript 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 ParenScript MACROLET form (note that while DEFJSMACRO is a Lisp form, MACROLET and SYMBOL-MACROLET are ParenScript forms). ParenScript also supports symbol macros, which can be introduced using the ParenScript 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 ParenScript 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))
```

31 The ParenScript Compiler

```
; (JS-COMPILE expr)
; (JS-TO-STRINGS compiled-expr position)
; (JS-TO-STATEMENT-STRINGS compiled-expr position)
;
; compiled-expr ::= a compiled ParenScript expression
; position      ::= a column number
;
; (JS-TO-STRING expression)
```

```
; (JS-TO-LINE expression)
;
; expression ::= a Lisp list of ParenScript code
;
; (JS body)
; (JS-INLINE body)
; (JS-FILE body)
; (JS-SCRIPT body)
;
; body ::= a list of ParenScript statements
```

The ParenScript compiler can be invoked from withing Lisp and from within ParenScript itself. The primary API function is JS-COMPILE, which takes a list of ParenScript, and returns an internal object representing the compiled ParenScript.

```
(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 ParenScript 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, ParenScript 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 ParenScript code, the macros JS, JS-INLINE, JS-FILE and JS-SCRIPT avoid the need to quote the ParenScript 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 ParenScript code to the HTML stream. These macros are also available inside ParenScript itself, and generate strings that can be used inside ParenScript code. Note that JS-INLINE in ParenScript is not the same JS-INLINE form as in Lisp, for example. The same goes for the other compilation macros.