Reference

August 1, 2007

Contents

1 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.

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 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;
      }
```

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", "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
```

3.1 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
```

4 Literal values

4.1 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
```

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.

4.3 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`.

4.4 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;
```

4.5 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
```

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
```

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

```
variable    => variable

a-variable  => aVariable

*math       => Math

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

6 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)
```

7 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
```

8 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.

9 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;
     }
```

10 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 (+ a 2 3 4 a))   => a += 2 + 3 + 4 + a

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

11 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'
```

12 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');
     }
```

13 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!');
       }
```

14 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.

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 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);
      }
```

16 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');
     }
```

17 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);
     }
```

18 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');
      }
```

19 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>')
```

Forms may be used in attribute lists to conditionally generate the next attribute. In this example the textarea is sometimes disabled.

```
(let ((disabled nil)
      (authorized t))
   (setf element.inner-h-t-m-l
         (html ((:textarea (or disabled (not authorized)) :disabled "disabled")
                "Edit me"))))
  =>  {
        var disabled = null;
        var authorized = true;
        element.innerHTML =
        '<textarea'
        + (disabled || !authorized ? ' disabled=\"' + 'disabled' + '\"' : '')
        + '>Edit me</textarea>';
      }

; (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>';
     }
```

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 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 defined in ParenScript itself (as opposed to Lisp) by using the ParenScript `MACROLET` and 'DEFMACRO' forms. ParenScript also supports the use of macros defined in the underlying Lisp. Existing Lisp macros can be imported into the ParenScript macro environment by 'IMPORT-MACROS-FROM-LISP'. This functionality enables code sharing between ParenScript and Lisp, and is useful in debugging since the full power of Lisp macroexpanders, editors and other supporting facilities can be used. However, it is important to note that the macroexpansion of Lisp macros and ParenScript macros takes place in their own respective environments, and many Lisp macros (especially those provided by the Lisp implementation) expand into code that is not usable by ParenScript. To make it easy for users to take advantage of these features, two additional macro definition facilities are provided by

ParenScript: 'DEFMACRO/JS' and 'DEFMACRO+JS'. 'DEFMACRO/JS' defines a Lisp macro and then imports it into the ParenScript macro environment, while 'DEFMACRO+JS' defines two macros with the same name and expansion, one in ParenScript and one in Lisp. 'DEFMACRO+JS' is used when the full 'macroexpand' of the Lisp macro yields code that cannot be used by ParenScript. 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))
```

21 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.