reference

March 5, 2009

Contents

1 Parenscript Language Reference

Create a useful package for the code here...

```
(in-package #:cl-user)
(defpackage #:ps-ref (:use #:ps))
(in-package #:ps-ref)
```

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 DO* DOEACH DOLIST
DOTIMES DOUBLE ELSE ENUM EQL EXPORT EXTENDS F FALSE FINAL FINALLY
FLOAT FLOOR FOR FOR-IN FUNCTION GOTO IF IMPLEMENTS IMPORT IN INCF
INSTANCEOF INT INTERFACE JS LABELED-FOR LAMBDA LET LET* LEXICAL-LET
LEXICAL-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'
```

Special characters such as newline and backspace are converted into their corresponding JavaScript escape sequences.

```
"   "   => '\\t'
```

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, F, FALSE, NIL, UNDEFINED, THIS
```

The Lisp symbols `T` and `FALSE` (or `F`) are converted to their JavaScript boolean equivalents `true` and `false`.

```
T     => true

FALSE => false

F => 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 ])

((slot-value this 'blorg) 1 2) => this.blorg(1, 2)

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

((slot-value (aref foobar 1) 'blorg) NIL T) => foobar[1].blorg(null, true)
```

Note that while most method calls can be abbreviated using the "." trick in symbol names (see "Symbol Conversion" above), this is not advised due to the fact that "object.function" is treated as a symbol distinct from both "object" and "function," which will cause problems if Parenscript package prefixes or package obfuscation is used.

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

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 parenthesized, according to the JavaScript operator precedence that can be found in table form at: http://www.codehouse.com/javascript/p

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

The pre increment and decrement operators are also available. `INCF` and `DECF` are the pre-incrementing and pre-decrementing operators. These operators can take only one argument.

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

; (SETQ {lhs rhs}*)
; (PSETQ {lhs rhs}*)
;
; lhs ::= a Parenscript symbol
; rhs ::= a Parenscript expression
```

Assignment is done using the SETF, PSETF, SETQ, and PSETQ forms, which are 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 (+ a 2 3 4 a))   => a += 2 + 3 + 4 + a;

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

The PSETF and PSETQ forms perform parallel assignment of places or variables using a number of temporary variables created by PS-GENSYM. For example:

```
(let* ((a 1) (b 2))
  (psetf a b b a))
=> var a = 1;
   var b = 2;
   var _js1 = b;
   var _js2 = a;
   a = _js1;
   b = _js2;
```

The SETQ and PSETQ forms operate identically to SETF and PSETF, but throw a compile-time error if the left-hand side form is not a symbol. For example:

```
(setq a 1)   => a = 1;

;; but...

(setq (aref a 0) 1)
;; => ERROR: The value (AREF A 0) is not of type SYMBOL.
```

New types of setf places can be defined in one of two ways: using DEFSETF or using DEFUN with a setf function name; both are analogous to their Common Lisp counterparts. DEFSETF supports both long and short forms, while DEFUN of a setf place generates a JavaScript function name with the __setf_ prefix:

```
(defun (setf color) (new-color el)
  (setf (slot-value (slot-value el 'style) 'color) new-color))
=> function __setf_color(newColor, el) {
       el.style.color = newColor;
   };

(setf (color some-div) (+ 23 "em"))
=> var _js2 = someDiv;
   var _js1 = 23 + 'em';
   __setf_color(_js1, _js2);
```

Note that temporary variables are generated to preserve evaluation order of the arguments as they would be in Lisp. The following example illustrates how setf places can be used to provide a uniform protocol for positioning elements in HTML pages:

```
(defsetf left (el) (offset)
  `(setf (slot-value (slot-value ,el 'style) 'left) ,offset))
=> null

(setf (left some-div) (+ 123 "px"))
=> var _js2 = someDiv;
   var _js1 = 123 + 'px';
   _js2.style.left = _js1;

(progn (defmacro left (el)
         `(slot-value ,el 'offset-left))
       (left some-div))
=> someDiv.offsetLeft;
```

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}?)
; (VAR var {value}?)
; (LET ({var | (var value)}*) body)
; (LET* ({var | (var value)}*) body)
; (LEXICAL-LET ({var | (var value)}*) body)
; (LEXICAL-LET* ({var | (var value)}*) body)
;
; var   ::= a Lisp symbol
; value ::= a Parenscript expression
; body  ::= a list of Parenscript statements
```

Parenscript special variables can be declared using the DEFVAR special form, which is similar to its equivalent form in Lisp. Note that the result is undefined if DEFVAR is not used as a top-level form.

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

One feature present in Parenscript that is not part of Common Lisp are lexically-scoped global variables, which are declared using the VAR special form. Parenscript provides two versions of the LET and LET* special forms for manipulating local variables: SIMPLE-LET / SIMPLE-LET* and LEXICAL-LET / LEXICAL-LET*. By default, LET and LET* are aliased to SIMPLE-LET and SIMPLE-LET*, respectively. SIMPLE-LET and SIMPLE-LET* bind their variable lists using simple JavaScript assignment. This means that you cannot rely on the bindings going out of scope at the end of the form. LEXICAL-LET and LEXICAL-LET* actually introduce new lexical environments for the variable bindings by creating anonymous functions. As you would expect, SIMPLE-LET and LEXICAL-LET do parallel binding of their variable lists, while SIMPLE-LET* and LEXICAL-LET* bind their variable lists sequentially. examples:

```
(simple-let* ((a 0) (b 1))
  (alert (+ a b)))
=> var a = 0;
   var b = 1;
   alert(a + b);

(simple-let* ((a "World") (b "Hello"))
  (simple-let ((a b) (b a))
    (alert (+ a b))))
=> var a = 'World';
   var b = 'Hello';
   var _js_a1 = b;
   var _js_b2 = a;
   var a = _js_a1;
   var b = _js_b2;
   delete _js_a1;
   delete _js_b2;
   alert(a + b);

(simple-let* ((a 0) (b 1))
  (lexical-let* ((a 9) (b 8))
```

```
    (alert (+ a b)))
  (alert (+ a b)))
=> var a = 0;
   var b = 1;
   (function () {
       var a = 9;
       var b = 8;
       alert(a + b);
   })();
   alert(a + b);

(simple-let* ((a "World") (b "Hello"))
  (lexical-let ((a b) (b a))
    (alert (+ a b)))
  (alert (+ a b)))
=> var a = 'World';
   var b = 'Hello';
   (function (a, b) {
       alert(a + b);
   })(b, a);
   alert(a + b);
```

Moreover, beware that scoping rules in Lisp and JavaScript are quite different. For example, don't rely on closures capturing local variables in the way that you would normally expect.

15 Iteration constructs

```
; (DO ({var | (var {init}? {step}?)}*) (end-test {result}?) body)
; (DO* ({var | (var {init}? {step}?)}*) (end-test {result}?) body)
; (DOTIMES (var numeric-form {result}?) body)
; (DOLIST (var list-form {result}?) body)
; (DOEACH ({var | (key value)} object-form {result}?) 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-form  ::= a Parenscript expression resulting in an object
; init         ::= a Parenscript expression
; step         ::= a Parenscript expression
; end-test     ::= a Parenscript expression
; result       ::= a Parenscript expression
; body         ::= a list of Parenscript statements
```

All interation special forms are transformed into JavaScript `for` statements and, if needed, lambda expressions. `DO`, `DO*`, and `DOTIMES` carry the same semantics as their Common Lisp equivalents. `DO*` (note the variety of possible init-forms:

```
(do* ((a) b (c (array "a" "b" "c" "d" "e"))
      (d 0 (1+ d))
      (e (aref c d) (aref c d)))
```

```
      ((or (= d c.length) (eql e "x")))
   (setf a d b e)
   (document.write (+ "a: " a " b: " b "<br/>")))
=> for (var a = null, b = null, c = ['a', 'b', 'c', 'd', 'e'], d = 0, e = c[d]; !(d == c.lengt
        a = d;
        b = e;
        document.write('a: ' + a + ' b: ' + b + '<br/>');
    };
```

DO (note the parallel assignment):

```
(do ((i 0 (1+ i))
     (s 0 (+ s i (1+ i))))
    ((> i 10))
  (document.write (+ "i: " i " s: " s "<br/>")))
=> var _js_i1 = 0;
   var _js_s2 = 0;
   var i = _js_i1;
   var s = _js_s2;
   delete _js_i1;
   delete _js_s2;
   for (; i <= 10; ) {
       document.write('i: ' + i + ' s: ' + s + '<br/>');
       var _js3 = i + 1;
       var _js4 = s + i + (i + 1);
       i = _js3;
       s = _js4;
   };
```

compare to DO*:

```
(do* ((i 0 (1+ i))
      (s 0 (+ s i (1- i))))
     ((> i 10))
  (document.write (+ "i: " i " s: " s "<br/>")))
=> for (var i = 0, s = 0; i <= 10; i += 1, s += i + (i - 1)) {
       document.write('i: ' + i + ' s: ' + s + '<br/>');
   };
```

DOTIMES:

```
(let* ((arr (array "a" "b" "c" "d" "e")))
  (dotimes (i arr.length)
    (document.write (+ "i: " i " arr[i]: " (aref arr i) "<br/>"))))
=> var arr = ['a', 'b', 'c', 'd', 'e'];
   for (var i = 0; i < arr.length; i += 1) {
       document.write('i: ' + i + ' arr[i]: ' + arr[i] + '<br/>');
   };
```

DOTIMES with return value:

```
(let* ((res 0))
  (alert (+ "Summation to 10 is "
            (dotimes (i 10 res)
              (incf res (1+ i))))))
```

```
=> var res = 0;
   alert('Summation to 10 is ' + (function () {
       for (var i = 0; i < 10; i += 1) {
           res += i + 1;
       };
       return res;
   })());
```

`DOLIST` is like CL:DOLIST, but that it operates on numbered JS arrays/vectors.

```
(let* ((l (list 1 2 4 8 16 32)))
  (dolist (c l)
    (document.write (+ "c: " c "<br/>"))))
=> var l = [1, 2, 4, 8, 16, 32];
   for (var c = null, _js_arrvar2 = l, _js_idx1 = 0; _js_idx1 < _js_arrvar2.length; _js_idx1 +
       c = _js_arrvar2[_js_idx1];
       document.write('c: ' + c + '<br/>');
   };

(let* ((l (list 1 2 4 8 16 32))
       (s 0))
  (alert (+ "Sum of " l " is: "
            (dolist (c l s)
              (incf s c)))))
=> var l = [1, 2, 4, 8, 16, 32];
   var s = 0;
   alert('Sum of ' + l + ' is: ' + (function () {
       for (var c = null, _js_arrvar2 = l, _js_idx1 = 0; _js_idx1 < _js_arrvar2.length; _js_id
           c = _js_arrvar2[_js_idx1];
           s += c;
       };
       return s;
   })());
```

`DOEACH` iterates across the enumerable properties of JS objects, binding either simply the key of each slot, or alternatively, both the key and the value.

```
(let* ((obj (create :a 1 :b 2 :c 3)))
  (doeach (i obj)
    (document.write (+ i ": " (aref obj i) "<br/>"))))
=> var obj = { a : 1, b : 2, c : 3 };
   for (var i in obj) {
       document.write(i + ': ' + obj[i] + '<br/>');
   };

(let* ((obj (create :a 1 :b 2 :c 3)))
  (doeach ((k v) obj)
    (document.write (+ k ": " v "<br/>"))))
=> var obj = { a : 1, b : 2, c : 3 };
   var v;
   for (var k in obj) {
       v = obj[k];
       document.write(k + ': ' + v + '<br/>');
   };
```

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

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

The HTML generator of Parenscript is very similar to the htmlgen HTML generator library included with 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.

```
(ps-html ((:a :href "foobar") "blorg"))
=> '<A HREF=\"foobar\">blorg</A>'

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

We can recursively call the Parenscript compiler in an HTML expression.

```
(document.write
  (ps-html ((:a :href "#"
              :onclick (ps-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
         (ps-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>';
```

20 Macrology

```
; (DEFPSMACRO name lambda-list macro-body)
; (DEFPSMACRO/PS name lambda-list macro-body)
; (DEFPSMACRO+PS name lambda-list macro-body)
; (DEFINE-PS-SYMBOL-MACRO symbol expansion)
; (IMPORT-MACROS-FROM-LISP symbol*)
; (MACROLET ({name lambda-list macro-body}*) body)
; (SYMBOL-MACROLET ({name macro-body}*) body)
; (PS-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 `DEFPSMACRO`, the Parenscript language can be extended. `DEFPSMACRO` 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.

```
(defpsmacro 1- (form)
  `(- ,form 1))
```

```
(defpsmacro 1+ (form)
  `(+ ,form 1))
```

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

```
(defpsmacro dolist ((var array &optional (result nil result?)) &body body)
  (let ((idx (ps-gensym "_js_idx"))
        (arrvar (ps-gensym "_js_arrvar")))
    `(do* (,var
           (,arrvar ,array)
           (,idx 0 (1+ ,idx)))
          ((>= ,idx (slot-value ,arrvar 'length))
           ,@(when result? (list result)))
       (setq ,var (aref ,arrvar ,idx))
       ,@body)))
```

Macros can be defined in Parenscript code itself (as opposed to from Lisp) by using the Parenscript MACROLET and DEFMACRO forms. Note that macros defined this way are defined in a null lexical environment (ex - (let ((x 1)) (defmacro baz (y) '(+ ,y ,x))) will not work), since the surrounding Parenscript code is just translated to JavaScript and not actually evaluated. Parenscript also supports the use of macros defined in the underlying Lisp environment. 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/PS and DEFMACRO+PS. DEFMACRO/PS defines a Lisp macro and then imports it into the Parenscript macro environment, while DEFMACRO+PS defines two macros with the same name and expansion, one in Parenscript and one in Lisp. DEFMACRO+PS 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 or defined in Lisp with DEFINE-PS-SYMBOL-MACRO. For example, the Parenscript WITH-SLOTS is implemented using symbol macros.

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

21 The Parenscript namespace system

```
; (setf (PS-PACKAGE-PREFIX package-designator) string)
```

Although JavaScript does not offer namespacing or a package system, Parenscript does provide a namespace mechanism for generated JavaScript by integrating with the Common Lisp package system. Since Parenscript code is normally read in by the Lisp reader, all symbols (except for uninterned ones, ie - those specified with the #: reader macro) have a Lisp package. By default, no packages are prefixed. You can specify that symbols in a particular package receive a prefix when translated to JavaScript with the `PS-PACKAGE-PREFIX` place.

```
(defpackage "PS-REF.MY-LIBRARY"
  (:use "PARENSCRIPT"))
(setf (ps-package-prefix "PS-REF.MY-LIBRARY") "my_library_")

(defun ps-ref.my-library::library-function (x y)
  (return (+ x y)))
  -> function my_library_libraryFunction(x, y) {
        return x + y;
     }
```

22 Identifier obfuscation

```
; (OBFUSCATE-PACKAGE package-designator &optional symbol-map)
; (UNOBFUSCATE-PACKAGE package-designator)
```

Similar to the namespace mechanism, Parenscript provides a facility to generate obfuscated identifiers in specified CL packages. The function `OBFUSCATE-PACKAGE` may optionally be passed a hash-table or a closure that maps symbols to their obfuscated counterparts. By default, the mapping is done using `PS-GENSYM`.

```
(defpackage "PS-REF.OBFUSCATE-ME")
(obfuscate-package "PS-REF.OBFUSCATE-ME"
  (let ((code-pt-counter #x8CF0)
        (symbol-map (make-hash-table)))
    (lambda (symbol)
      (or (gethash symbol symbol-map)
          (setf (gethash symbol symbol-map)
                (make-symbol (string (code-char (incf code-pt-counter)))))))))

(defun ps-ref.obfuscate-me::a-function (a b ps-ref.obfuscate-me::foo)
  (+ a (ps-ref.my-library::library-function b ps-ref.obfuscate-me::foo)))
  -> function (a, b, ) {
       a + my_library_libraryFunction(b, );
     }
```

The obfuscation and namespace facilities can be used on packages at the same time.

23 The Parenscript Compiler

```
; (PS &body body)
; (PS* &body body)
```

```
; (PS1* parenscript-form)
; (PS-INLINE form &optional *js-string-delimiter*)
; (PS-INLINE* form &optional *js-string-delimiter*)

; (LISP lisp-forms)
;
; body ::= Parenscript statements comprising an implicit 'PROGN'
```

For static Parenscript code, the macro `PS` compiles the provided forms at Common Lisp macro-expansion time. `PS*` and `PS1*` evaluate their arguments and then compile them. All these forms except for `PS1*` treat the given forms as an implicit PROGN. `PS-INLINE` and `PS-INLINE*` take a single Parenscript form and output a string starting with "javascript:" that can be used in HTML node attributes. As well, they provide an argument to bind the value of *js-string-delimiter* to control the value of the JavaScript string escape character to be compatible with whatever the HTML generation mechanism is used (for example, if HTML strings are delimited using #\', using #\" will avoid conflicts without requiring the output JavaScript code to be escaped). By default the value is taken from *js-inline-string-delimiter*. Parenscript can also call out to arbitrary Common Lisp code at code output time using the special form `LISP`. The form provided to `LISP` is evaluated, and its result is compiled as though it were Parenscript code. For `PS` and `PS-INLINE`, the Parenscript output code is generated at macro-expansion time, and the `LISP` statements are inserted inline and have access to the enclosing Common Lisp lexical environment. `PS*` and `PS1*` evaluate the `LISP` forms with eval, providing them access to the current dynamic environment only.