The Parenscript Common Lisp to JavaScript compiler

Vladimir Sedach <vsedach@gmail.com>

What Parenscript is not

- Not a full ANSI INCITS 226-1994 (the standard formerly known as X3J13) implementation in JavaScript
- Not a Lisp interpreter in JavaScript
- Not a framework, toolkit, UI or AJAX library

What Parenscript doesn't do

- No dependencies or run-time library
  - Any JS code produced runs as-is
- No new data types
- No weird calling convention
  - Any PS code can call any JS code directly and vice-versa
- No whole program compilation
  - JS code can be produced incrementally

Where is Parenscript used?

- Web frameworks: BKNR, UCW, TPD2, Weblocks
- Libraries: cl-closure-template, Suave, css-lite, clouchdb, uri-template
- Many startups and commercial projects

Parenscript history

- March 2005, Manuel Odendahl announces initial release
- Swallowed up by the bese project in 2006
- Skysheet developers spin Parenscript out as separate project again in 2007, take over development, do lots of PR

Similar projects

- Not many options for Common Lisp:
  - Jason Kantz's JSGEN (2006, s-exp over JS)
  - Peter Seibel's Lispscript (2006?, s-exp over JS)
- Many more options for Scheme:
  - Scheme2JS (2005?)
  - Feeley and Gaudron's JSS
  - Many others
- Moritz Heidkamp's survey of Lisp-in-JavaScript implementations lists 28!

Use cases

- Lisp at runtime
- No Lisp at runtime

Lisp at runtime

- Compile to inline JavaScript in HTML page
- Stream client-specific JavaScript to browser via XMLHttpRequest or page loads
- SWANK-based development environment

No Lisp at runtime

- Compile to static files sent to browser
  - If done with a runtime, can use all of CL features for a build/deploy/cache system
- Compile to library to be used by other JS code
- Compile to JS client- and server-side (node.js) code at the same time
  - Deploy JS only
  - Parenscript used as static compiler for whole web stack

Implementation Techniques

What should JS code look like?

- Write CL, but debug JS
- Generated code has to be short, readable, idiomatic
- ”Optimization” in the context of this talk means ”more readable code”
- What about performance?
  - Can only assume that JS compilers try to optimize for code written by humans (ie – readable and idiomatic code)

Statements vs Expressions

- In JavaScript, can transform any statement to an expression by wrapping it in a lambda
  - Except for BREAK, CONTINUE, and RETURN
- Many optimizations to avoid lambda-wrapping possible, depending on combination of special forms

Function argument lists

- Full Common Lisp lambda lists in JavaScript
  - In a way completely compatible with other JavaScript code
- ARGUMENTS is great
- NULL (argument provided) vs UNDEFINED (argument not provided)
- Keywords: f(”key2”, value2, ”key1”, value1);
  - CL keywords and JS strings both self-evaluating

Macros

- Share the same representation (macro-function) in CL and PS compilers
- But have different namespaces
- Share same code between client (CL) and server (JS) even if implementation details completely different (ex – IO, graphics)
  - Without modifying CL code
  - PS macros can shadow CL functions

Data structures

- No car/cdr
- Many Common Lisp forms are sequence oriented (DOLIST, LOOP, etc.)
- Most idiomatic Common Lisp code uses these forms
- Most code turns out to care about sequences, not linked lists vs arrays, and works as-is with JS arrays

Namespace (Lisp-1 and Lisp-N)

- JavaScript is a Lisp-1
- Common Lisp is a Lisp-4
  - With macros, Lisp-N
- Parenscript tries to be a Lisp-2 (separate variable and function namespaces)
- Code can be output unambiguously for lexically bound names (via α-renaming), but free variables referring to globals might cause conflicts
- No thought given to hygiene

Namespace (package system)

- All Parenscript symbols are CL symbols
- Packages can be given prefixes to namespace in JS:
  - foo::bar => foo_bar
- Package system also used for obfuscation and/or minification
- Exports list used to produce JS libraries with minified internals and prefixed API names

Namespace (case and escape)

- CL readtable case-insensitive by default, but can be set to preserve case
- Parenscript recognizes mixed-case symbols and outputs them correctly
- Simple translation for case-insensitive ones:
  - foo-bar => fooBar
- JS reserved chars escaped:
  - foo* => foostar

JavaScript dot operator

- Causes confusion
  - Some people think it's ok to use it for namespacing
- foo.bar is really (getprop foo 'bar)
- What about foo.bar.baz(foobar)?
  - (funcall (getprop (getprop foo 'bar) 'baz) foobar)
- Macros to the rescue:
  - (funcall (@ foo bar baz) foobar)

JavaScript dot operator 2

- Why not (foo.bar.baz foobar)?
- Parenscript 1 did it in compilation step
  - Turns out this doesn't work well with package system, breaks symbol macros
  - Symbol macros used for a lot of things in Parenscript 2 compiler
- Possible to do it in the reader
  - But ”.” is already reserved for dotted lists
  - Dotted lists are evil; remove them from Lisp

Lexical scoping

- Done by using α-renaming to declare variables in nearest function scope:

```
(lambda ()
  (let ((x 1))
    (let ((x 2))
      x)))
```

```
function () {
  var x = 1;
  var x1 = 2;
  return x1;
}
```

Lexical scoping: loops

- For variables in loops, JS creates a shared binding for all iterations, breaking closures
- WITH trick from Scheme2JS:

```
(dotimes (i 10)
  (lambda () (+ i 1)))
```

```
for (var i = 0; i < 10; i += 1) {
  with ({i : i}) {
    function () {
      return i + 1;
};};};
```

Dynamic scoping

- TRY/FINALLY to set a global variable

```
(let ((*foo* 1))
   (+ *foo* 2))
```

```
var FOO_TMPSTACK7;
try {
  FOO_TMPSTACK7 = FOO;
  FOO = 1;
  return FOO + 2;
}
finally {
  FOO = FOO_TMPSTACK7;
};
```

Multiple value return

- ARGUMENTS has 'callee' property (function)
- 'callee' has 'caller' property (function)
- Functions are JS objects, can attach arbitrary properties to them
- To return MV:
  - arguments.callee.caller.mv = [val2, ...]
- To receive MV:
  - others_vals = arguments.callee.mv

Multiple value return 2

- Actually requires more cleanup code than this.
- arguments.callee works even when calling to/from functions that don't know about Parenscript (a global ”return value arg” wouldn't)
- Non-lexical returns (see next slide) with multiple values easier to implement (but aren't yet)

Control transfer

- JavaScript only supports RETURNing from innermost function, BREAKing out of innermost block or loop
- Can implement CL's BLOCK/RETURN-FROM and CATCH/THROW using TRY-CATCH-FINALLY in general case
- Optimizations possible depending on special forms, whether a return value is wanted

Tools and future developments

CLOS

- Red Daly's PSOS library provides CLOS and Common Lisp's condition and resumable exception system
  - Not part of core Parenscript because it requires run-time library and introduces new types
- Possible to build up a full Common Lisp in JavaScript this way using Parenscript as a cross-compiler

Development environments

- SWANK-based
  - slime-proxy
- Non-SWANK
  - Numen (V8 only, to be released)

Future directions

- JS implementations have mostly caught on speed-wise
  - But have not, and most likely will not, catch up in efficient memory consumption
- Would like a full Common Lisp implementation in JS on the server (V8, Rhino, etc.) and web browser (Firefox, Chrome, etc.)