JACL: A Common Lisp for Developing Single-Page Web Applications

Alan Dipert
alan@dipert.org

ABSTRACT

This paper demonstrates JavaScript-Assisted Common Lisp (JACL), an experimental Web-browser based implementation of an extended subset of Common Lisp. JACL, which is in the early stages of development, is an effort to explore new techniques for large-scale Single-page Web Application (SPA) development in Lisp. JACL includes an optimizing Lisp-to-JavaScript compiler and interoperates with JavaScript. JACL promotes interactive, *residential* development in the Web browser environment with its *asynchronous reader* and Chrome DevTools-based REPL client.

CCS CONCEPTS

• **Software and its engineering → Dynamic compilers**; **Runtime environments**.

KEYWORDS

Common Lisp, JavaScript, web applications

ACM Reference Format:
Alan Dipert. 2020. JACL: A Common Lisp for Developing Single-Page Web Applications. In *Proceedings of ELS '20: European Lisp Symposium (ELS '20).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.3764494

1 INTRODUCTION

The demand for SPAs in the past decade has only grown, and users and stakeholders continually expect larger and more sophisticated applications. Unfortunately, large-scale development on the Web browser platform presents a particular set of challenges that are not easily overcome. Developers have responded to these challenges by creating a widening variety of special-purpose programming languages that compile to JavaScript [12, 23, 24]. Each new language promotes one or more paradigms, application architectures, or development workflows, and claims some advantage relative to the status quo.

This paper demonstrates one new such language, JavaScript-Assisted Common Lisp (JACL), an experimental implementation of an extended subset of Common Lisp. JACL was created to explore new techniques for applying Common Lisp — a proven[6, 13, 14] substrate for UI innovation — to SPA development.

Many projects involving compilation of Lisp to JavaScript precede JACL. Lisps that have either demonstrated industrial utility or that implement a significant subset of Common Lisp are surveyed in appendix A. Like many of these related efforts, JACL includes an online, optimizing compiler and supports interoperation with JavaScript. JACL distinguishes itself from these efforts by placing special emphasis on the value of *residential* development style, where both applications and the tools used to create them co-evolve in a shared environment. JACL provides fundamental support for residential development with its *asynchronous reader*.

ELS '20, April 27–28, 2020, Zürich, Switzerland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.5281/zenodo.3764494

2 INTEROPERATION WITH JAVASCRIPT

JACL integrates tightly with JavaScript and depends heavily on the JavaScript runtime. As a result, JACL enjoys roughly the same applicability and performance characteristics as the JavaScript platform. However, this high degree of integration is at odds with comformance to the Common Lisp specification, and so JACL will never strictly conform.

2.1 Object Types

JACL introduces several of its own object types, currently implemented in JavaScript, including `Cons`, `LispSymbol`, and `LispString`. `Cons` and `LispSymbol` are introduced because JavaScript does not include direct equivalents. `LispString` is introduced because the native JavaScript `String` is immutable, whereas Lisp strings are mutable.

JACL includes support for only one numeric type, the JavaScript `Number` object. The JavaScript `Number` is a double-precision 64-bit IEEE 754 value. The JACL reader interprets integers as `Number` objects. In the future, JACL will also interpret floating-point numbers as `Number`. This decision trades ANSI conformance for performance. If either type were boxed, arithmetic performance would suffer intolerably. JSCL[20] and Valtan[11] make the same tradeoff.

JACL functions are JavaScript functions, and may be invoked by JavaScript callbacks without a special calling convention. JavaScript functions named as Lisp values may be invoked with `FUNCALL` or `APPLY`. Neither arguments nor return values are automatically coerced to or from any particular set of object types.

2.2 Operators

The JACL compiler supports a special operator for constructing fragments of JavaScript code, verbatim, from Lisp. The semantics of this operator, `JACL:%JS`, are inspired by a similar feature of ClojureScript[9], `js*`. For example, the following JACL code displays the number 3 in an alert box:

```
(JACL:%JS "window.alert(~{})" 3)
```

The character sequence `~{}` is distinct from any plausible JavaScript syntax and so is used as placeholder syntax. There must be as many placeholders as there are arguments to `JACL:%JS`.

In addition to `JACL:%JS`, the JACL compiler currently supports three more special operators for interacting with the host platform: `JACL:%NEW`, `JACL:%DOT` and `JACL:%CALL`. These operators perform JavaScript object instantiation, field access, and function calls, respectively. Since JACL functions are JavaScript functions, `JACL:%CALL` is the basis for `FUNCALL` in JACL, and for function calls generally.

JACL also supplies a convenience macro, `JACL:\.` or "the dot macro" for performing a series of field accesses and method calls[1] concisely. The dot macro takes direct inspiration from the `..` macro of Clojure[8]. `JACL:\.` expands to zero or more nested `JACL:%DOT` or `JACL:%CALL` forms. Here is an example of a `JACL:\.` form — equivalent to the JavaScript expression `(123).toString().length` — and its corresponding expansion:

```
(\. 123 (|toString|) |length|)
(%DOT (%CALL 123 |toString|) |length|)
```

Note that JavaScript identifiers are case sensitive, and so case-preserving, pipe-delimited Lisp symbols must be used to refer to JavaScript object field and method names. The *readtable case* of the JACL reader cannot currently be modified. The dot macro also recognizes Lisp or JavaScript strings as JavaScript identifiers.

2.3 Reader Macros

JACL includes two reader macros to support interoperation with JavaScript. These macros may be added to the `*READTABLE*` by calling the function `(JACL:ENABLE-JS-SYNTAX)`. `@"` denotes JavaScript `String` objects and `@|` denotes JavaScript identifiers.

For example, the following two forms, which both evaluate to a JavaScript `String`, are equivalent:

```
@"Hello"
(\. "Hello" (|toString|))
```

`@|` may generally be used in place of the `JACL:%JS` special form to refer to JavaScript identifiers. `(JACL:%JS "alert")` and `@|alert|` are equivalent.

3 RUNNING JACL PROGRAMS

Currently, JACL programs may be evaluated in the Web browser in two ways: by adding Lisp `<script>` tags to the `<head>` of a Web page that also includes `jacl.js`, or by using the `jacl` tool included in the JACL distribution[1] to connect to a running Web browser.

3.1 Lisp Scripts

Development of JACL itself is currently driven primarily by modifying `jacl.js` and the `boot.lisp` and `jacl-tests.lisp` Lisp scripts. The Lisp scripts are included in the `jacl.html` file in the JACL distribution[1]. After each modification, the Web browser is reloaded, and test results are displayed.

This Lisp script-based workflow is similar to the traditional JavaScript development workflow and has served JACL development so far. However, Lisp scripts require runtime parsing and compilation of JACL source code, among other inefficiencies. Reloading the Web browser also destroys the entire runtime environment.

[1] Strictly speaking, JavaScript "method calls" are normal function calls but with a particular value of `this`.

The easiest way to create JACL programs in this manner is to start with the `jacl.html` Web page provided by JACL and then modify it by removing or adding new Lisp scripts.

It is imagined that ultimately, Lisp sources will be incorporated into the Lisp *image* exclusively by the REPL client tool. An arrangement such as this decouples source code loading from the Web browser lifecycle. Production executables may then be produced at any time from the Lisp image using a Lisp function in a manner similar to the `SAVE-LISP-AND-DIE`[22] function in SBCL or the `DELIVER`[16] function in LispWorks.

3.2 REPL

JACL includes a REPL client program, `jacl`, that may be used to execute JACL programs in a Web browser from a terminal on the host. This process is described in detail in the `RUN.md` document included in the JACL distribution[1], but is summarized here.

In order to use the REPL, the user must first start either the Google Chrome or Chromium browser with the remote debugging feature enabled. With remote debugging enabled, the Web browser may be controlled using a client program over a WebSocket connection. Then, the user must navigate to a Web page that includes at least `jacl.js` and `boot.lisp`.

Finally, the user must start the `jacl` REPL client in a terminal. `jacl` leverages the remote debugging feature as a REPL transport, using it to send and receive characters between the host and the remote JACL runtime. The `jacl` tool is currently written in R[21] and uses the `chromote`[7] package for interacting with the remote Chrome or Chromium browser.

The `jacl` program has no knowledge of JACL syntax or semantics; it merely sends and receives characters. The intentional simplicity of `jacl` is part of the larger project goal of promoting residential-style tool and program development in the target environment. The simplicity of `jacl` is possible because of the asynchronous nature of the JACL reader. Incoming characters delivered over the WebSocket debugging connection are received by callback functions in the Web browser. The received characters are asynchronously and incrementally parsed into Lisp data. When a complete datum is formed, the compiler is called, and the resulting JavaScript is evaluated. Finally, any output is sent back over the debugger connection and received and printed by the `jacl` program.

4 CONCLUSION

We introduced JACL, a new and experimental Common Lisp created to explore techniques for building sophisticated SPAs. JACL integrates tightly with the Web browser platform and interoperates directly with JavaScript. Compared to other browser-based Lisps, JACL promotes residential development, and introduces a new technique for integrating the REPL into the development workflow.

5 FUTURE WORK

JACL currently lacks many basic Common Lisp data types, functions, and operators. Ultimately, JACL should support as much of Common Lisp as is possible, accounting for the severe limitations imposed by JavaScript and the Web platform. Fortunately, the many other existing Common Lisps that compile to JavaScript demonstrate that a compelling implementation is achievable.

An in-browser REPL and other tools for interacting with the JACL runtime in the Web browser would be desirable. Such tools could optionally remain as parts of deployed applications and provide a degree of introspection and extension capability even after the application has been deployed.

Other than work related to missing features such as multiple values, CLOS, and the conditions system, much design work remains with regard to the specific affordances of the `jacl` tool. For example, it's unclear how a large JACL project involving library dependencies and multiple source files should be managed and loaded.

6 ACKNOWLEDGMENTS

The author wishes to thank Micha Niskin, Bart Botta, Kevin Lynagh, Lionel Henry, and Andy Keep for invaluable feedback on early versions of this paper. The author wishes to express particular thanks to Robert Strandh not only for his feedback, but also for his guidance on the writing process.

The author also wishes to express special gratitude to his beautiful wife, Sandra Dipert, for her encouragement and support.

Finally, the author wishes to express his deepest thanks to his father, the late Randall R. Dipert, for first telling him about Lisp and many other things besides.

REFERENCES

[1] Alan Dipert. 2020. *JACL*. Retrieved April 16, 2020 from https://tailrecursion.com/JACL/
[2] Marco Baringer. 2005. *Parenscript*. Retrieved February 12, 2020 from https://web.archive.org/web/20051122141019/http://blogs.bl0rg.net/netzstaub/archives/000525.html
[3] Mihai Bazon. 2012-2018. *Implementation notes*. Retrieved February 12, 2020 from http://lisperator.net/slip/impl
[4] Mihai Bazon. 2012-2018. *SLip âĂŤ a Lisp system in JavaScript*. Retrieved February 12, 2020 from http://lisperator.net/slip/
[5] Mihai Bazon. 2012-2018. *Versus Common Lisp*. Retrieved February 12, 2020 from http://lisperator.net/slip/vscl
[6] Howard I. Cannon. 2007. *Flavors: A non-hierarchical approach to object-oriented programming*. Retrieved February 12, 2020 from http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf
[7] Winston Chang. [n.d.]. *chromote: Headless Chrome Web Browser Interface*. https://github.com/rstudio/chromote
[8] Cognitect, Inc. 2020. *Clojure*. Retrieved February 12, 2020 from https://clojure.org/
[9] Cognitect, Inc. 2020. *ClojureScript*. Retrieved February 12, 2020 from https://clojurescript.org/
[10] Cognitect, Inc. 2020. *Companies*. Retrieved https://clojure.org/community/companies from https://clojurescript.org/community/companies
[11] cxxxr. [n.d.]. *cxxxr/valtan*. Retrieved April 4, 2020 from https://github.com/cxxxr/valtan
[12] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs*. Retrieved February 12, 2020 from https://elm-lang.org/assets/papers/concurrent-frp.pdf
[13] B. A. Myers et al. 1990. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer* 23, 11 (Nov. 1990), 71–85. https://doi.org/10.1109/2.60882
[14] Paul Hammant. 2013. *Interface Builder's Alternative Lisp timeline*. Retrieved February 20, 2020 from https://paulhammant.com/2013/03/28/interface-builders-alternative-lisp-timeline/
[15] Rich Hickey. 2012. ClojureScript Release. Retrieved February 12, 2020 from https://www.youtube.com/watch?v=tVooR-dF_Ag
[16] LispWorks Ltd. 2017. *deliver*. Retrieved February 21, 2020 from http://www.lispworks.com/documentation/lw71/DV/html/delivery-220.htm
[17] Vladimir Sedach Marco Baringer, Henrik Hjelte. 2005-2019. *Parenscript Reference Manual*. Retrieved February 12, 2020 from https://common-lisp.net/project/parenscript/reference.html
[18] Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case studies in Common Lisp*. Morgan Kaufmann Publishers, San Francisco, CA, USA.
[19] David Vázquez Púa. 2018. Growing a Lisp compiler. Retrieved February 12, 2020 from https://www.youtube.com/watch?v=XT7JYPtWMd8
[20] David Vázquez Púa and contributors. [n.d.]. *jscl-project/jscl*. Retrieved February 12, 2020 from https://github.com/jscl-project/jscl/
[21] R Core Team. [n.d.]. *R: A Language and Environment for Statistical Computing*. http://www.R-project.org/
[22] SBCL Project Contributors. 2020. *SBCL 2.0.1 User Manual*. Retrieved February 21, 2020 from http://www.sbcl.org/manual/
[23] Soma Somasegar. 2012. *TypeScript: JavaScript Development at Application Scale*. Retrieved February 4, 2020 from https://web.archive.org/web/20121003001910/http://blogs.msdn.com/b/somasegar/archive/2012/10/01/typescript-javascript-development-at-application-scale.aspx
[24] Wikipedia contributors. 2020. Reason (syntax extension for OCaml) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Reason_(syntax_extension_for_OCaml)&oldid=940051580 [Online; accessed February 12, 2020].

A SURVEY OF RELATED LISPS

A.1 Parenscript

Released in 2005[2], Parenscript[17] was the first Common Lisp compiler to target JavaScript. Parenscript is not bootstrapped and its compiler is not written in JavaScript, and so it relies on a hosting Common Lisp system for compilation. Only JavaScript types are available to Parenscript programs at runtime, and so Parenscript is more of a syntax frontend for JavaScript than it is an interactive Lisp system. While Parenscript is not positioned to facilitate large-scale SPA development, it remains a popular way to add dynamic JavaScript-based behaviors to static Web sites.

A.2 SLip

SLip[3, 4] is arguably the most ambitious Common Lisp-on-JavaScript system created to date, even though it intentionally diverges[5] from Common Lisp in certain ways. It offers a stunning array of powerful features including a self-hosting compiler, a full set of control operators, JavaScript Foreign-Function Interface (FFI), tail-call optimization, green threads, and perhaps most impressively, a resident Emacs clone, *Ymacs*. SLip is based originally on the compiler and bytecode interpreter presented in Chapter 23 of *Paradigms of Artificial Intelligence Programming: Case studies in Common Lisp*[18].

A.3 JSCL

JSCL[19, 20] compiles directly to JavaScript and is self-hosting, includes the major control operators, and integrates tightly with JavaScript. JSCL includes a reader, compiler, and printer, and evaluation is performed by the JavaScript `eval()` function. Between these, a Read Eval Print Loop (REPL) is possible, and the JSCL distribution includes an implementation of one.

A.4 ClojureScript

ClojureScript [9, 15] is probably the most successful Lisp dialect for building SPAs by number of commercial users [10]. ClojureScript is a dialect of an earlier language, Clojure[8], which targets Java Virtual Machine (JVM) bytecode. The ClojureScript reader and macro systems were both originally hosted in Clojure, in a manner similar to Parenscript. ClojureScript prioritizes the ability to produce high-performance deliverables.

A.5 Valtan

Valtan[11] compiles to JavaScript and includes a suite of FFI operators for interoperating with JavaScript. It is self-hosting and features a sophisticated, CLOS-based compiler architecture. It also includes a REPL and several example applications.