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

Alan Dipert
alan@dipert.org

ABSTRACT

This paper introduces JavaScript-Assisted Common Lisp (JACL), a new Web-browser based implementation of an extended subset of Common Lisp. JACL — which is under active development and approaching utility — is an effort to facilitate the use of Common Lisp in overcoming the challenges of Single-page Web Application (SPA) development. JACL promotes interactive development in the Web browser environment with its *asynchronous reader* and Chrome DevTools-based REPL client. JACL also includes an optimizing Lisp-to-JavaScript compiler capable of generating competitively small and efficient JavaScript.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic compilers**; **Run-time environments**.

KEYWORDS

Common Lisp, JavaScript, web applications

ACM Reference Format:
Alan Dipert. 2020. JACL: A Common Lisp for Developing Single-Page Web Applications. In *ELS '20: European Lisp Symposium, April 27–28, 2020, Zürich, Switzerland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

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 [11, 27, 29]. Each new language promotes one or more paradigms, application architectures, or development workflows, and claims some advantage relative to the status quo.

This paper presents one new such language, JavaScript-Assisted Common Lisp (JACL), an implementation of an extended subset of Common Lisp. The primary goal of the JACL project is to ease SPA development by applying Common Lisp — a proven[6, 12, 14] substrate for UI innovation — to the difficult challenges now faced by developers. The JACL language is envisioned as the means to that goal.

Most popular, contemporary compile-to-JavaScript languages are oriented around the affordances of static type checking; JACL, as a Lisp, is not. Compared to similar languages that *are* Lisps, JACL differentiates itself in two particular ways: with its approach to the REPL, and with its compilation techniques.

ELS '20, April 27–28, 2020, Zürich, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

2 RELATED WORK

Many Lisps exist that either compile to JavaScript or are interpreted by a JavaScript program. Here, only Lisps that have either demonstrated industrial utility or that implement a significant subset of Common Lisp are featured.

2.1 Parenscript

Released in 2005[1], 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.

2.2 SLip

SLip[2, 3] is arguably the most ambitious Common Lisp-on-JavaScript system created to date, even though it intentionally diverges[4] 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*[22].

Lisp files may be batch-compiled to FASLs. FASLs represent code as JavaScript code instead of as Lisp data. The browser is able to load FASLs faster than SLip code because the JavaScript parser in the browser is much faster than the SLip reader. Despite the ability to produce FASLs, the interpreted nature of SLip precludes the system from producing competitively fast or small application deliverables. Consequently, SLip does not satisfy the JACL project goal of facilitating large-scale industrial SPA development.

2.3 JSCL

Of existing Common Lisps, JSCL[23, 24] is the one aligned most closely with the JACL project goal. Unlike Parenscript, JSCL is a genuine Lisp system. And unlike SLip, JSCL compiles directly to JavaScript instead of to an interpreted bytecode. It 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.

2.3.1 *Synchronous reader.*

JSCL supports reading from string-backed pseudo-streams. Input streams from which characters may be synchronously consumed are not an abstraction supported by JavaScript in Web browsers. With a few obscure exceptions[1], JavaScript programs may only receive input asynchronously. An error is signaled if the end of a string-backed input stream is encountered before the reader has finished reading a datum.

Because input strings may not contain partial data, the REPL necessitates a "pre"-reader process that accumulates characters in response to asynchronous input events, and invokes the reader only once a complete form — as a string — has been accumulated. Such a pre-reader can be found in the JSCL REPL implementation. It handles standard syntax, but has the potential to be stymied by extended syntax, such as that added by reader macros. As such, the pre-reader is a separate, degenerate reader that limits what's capable of being read by the underlying, full-featured synchronous reader.

2.3.2 *Compiler organization.*

JSCL compilation is performed in two stages:

- Conversion from Lisp to a JavaScript Abstract Syntax Tree (AST) represented as S-expressions.
- Conversion from JavaScript AST to JavaScript strings. Some code-size optimization of arithmetic expressions is also performed in this stage.

The first stage, the conversion from Lisp to JavaScript Abstract Syntax Trees (AST), is where the implementation of the Lisp special forms in terms of JavaScript language constructs and runtime support is performed. This conversion is done in a single pass in which macro expansion, lexical analysis, and JavaScript AST generation all occur. The lexical environment is maintained in a dynamically-scoped variable as the compiler descends into Lisp code and produces JavaScript AST.

Code for `TAGBODY` is generated in the first stage, and the generated code is much slower than comparable JavaScript code. Every control transfer initiated by `GO` results in a JavaScript exception being thrown, which is an expensive operation. Since many Common Lisp operators have *implicit tagbodies*, and since most other iteration operators are expressed in terms of `TAGBODY`, this performance problem pervades the JSCL system.

More efficient ways of implementing `TAGBODY` are not hard to imagine, but the JSCL compiler does not amene itself to the implementation of this or other high-level optimizations, as JSCL lacks a sufficiently expressive intermediate representation (IR). High-level optimizations and transformations are important for a Lisp to JavaScript compiler to perform, because they are exactly the kind that a JavaScript optimizer could not later perform, provided only JavaScript code.

[1]`window.prompt()` and `window.confirm()` are two JavaScript functions that may be used to synchronously query the user for input.

2.4 ClojureScript

A discussion of industrial Lisp technology in the SPA setting would be incomplete without mention of ClojureScript [9]. ClojureScript is probably the most successful Lisp dialect for building SPAs by number of commercial users [10].

ClojureScript targets JavaScript, and 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.

Since its introduction[15], ClojureScript has heavily promoted and prioritized the ability to produce high-performance deliverables. It has always been capable of generating JavaScript deliverables amenable to aggressive optimization by the Google Closure Compiler. In this respect, ClojureScript aligns closely with the JACL project goal of competitive application performance. In fact, experience with, and admiration for, ClojureScript is the reason the ability to produce high-performance deliverables is considered a crucial capability of JACL.

Other than the fact that JACL is a Common Lisp and ClojureScript is not, the biggest difference between the two is that JACL promotes a browser-based development environment with minimal host-side tooling. ClojureScript, in contrast, promotes[21] a development experience oriented around compilation performed on the host.

3 DESIGN AND IMPLEMENTATION

From a design perspective, JACL is an effort to balance the requirements of an interactive and practical Lisp development environment with the constraints imposed by the Web browser platform. JACL proposes several innovations with respect to previous work in pursuit of this balance.

3.1 Asynchronous reader

The basis for interactive development in Lisp is undeniably the REPL, but as the JSCL "pre-reader" demonstrates, even the direct approach to this simple mechanism is hampered by the asynchronous model of input imposed by JavaScript.[19]. Traditionally, Lisp readers are implemented in environments with a blocking function for obtaining input, like `getc(1)` on Unix. The blocking nature of input consumption allows the reader to consume nested input recursively, using the call stack to accumulate structures. In JavaScript, input arrives asynchronously, and only when the call stack is empty. To mitigate this difficulty, the JACL reader facility is completely asynchronous. Conceptually, it is the JSCL REPL "pre-reader" taken to its inevitable conclusion.

The JACL reader is implemented as a JavaScript class, `Reader`. `Reader` instances are parameterized by an input source. One such input source is the `BufferedStream` class. The input source asynchronously notifies the reader instance when characters are available. The reader incrementally consumes these characters. Once the reader has accumulated a Lisp datum, it notifies its subscribers of the availability of the datum.

The JACL reader implementation makes extensive use of modern JavaScript features to support asynchronous programming including promises, iterators, async functions, async iterators, and the `await` keyword. These features simplify the JACL implementation and aid its performance [18]. It is hoped that JACL will eventually be written in itself, and that these features will be accessible from Lisp, perhaps as a set of implementation-dependent *declaration specifiers* available in `DECLARE` expressions.

The following example demonstrates, in JavaScript, the process by which the JACL reader consumes characters and produces Lisp objects. A `BufferedStream` and `Reader` are instantiated, sent characters asynchronously, and then the resulting Lisp object is printed to the JavaScript console.

```
(async () => {
  let bs = new BufferedStream(),
      rdr = new Reader(bs);

  window.setTimeout(() => bs.write("1"), 1000);
  window.setTimeout(() => bs.write("2"), 2000);
  window.setTimeout(() => bs.write("3"), 3000);
  window.setTimeout(() => bs.write(" "), 4000);

  console.log(await rdr.read());
})();
```

In the preceding example, `window.setTimeout()` is used to enqueue several JavaScript functions for execution after 1000, 2000, 3000, and 4000 milliseconds. Each enqueued function writes a character of input to the `BufferedStream` `bs` when invoked.

Before any enqueued function is invoked, execution proceeds to the `console.log` call, but is suspended by the `await` keyword.

The `await` keyword expects a JavaScript `Promise` object on its right side, and JavaScript execution remains suspended until the `Promise` has "resolved", or notified its subscribers that the pending computation it represents has completed. `rdr.read()` is an `async` function that returns such a `Promise`.

Once `rdr` has completed a form — in this case, the number 123, after about 4000 milliseconds have elapsed — execution continues, and `123` is printed to the JavaScript console.

The "read" portion of the JACL REPL is implemented by first instantiating `BufferedStream` and `Reader` objects. Then, in an asynchronous loop, objects are consumed from the `Reader`, analyzed, compiled, and evaluated.

Concurrently, characters may be sent to the `BufferedStream` instantiated by the REPL by calling the `write()` or `writeEach()` methods of the `BufferedaStream` object. Neither character input nor read object consumption impede other JavaScript operations, so the JACL REPL is suitable for embedding in applications.

Because of the platform and implementation-dependent nature of the JACL reader, JACL does not support Common Lisp input streams, nor its standard `READ` and `READ-FROM-STRING` functions. Standard interfaces for extending the reader, such as the `SET-MACRO-CHARACTER` function, are not directly supported. However, the JACL reader does provide an implementation-specific way to define reader macros.

3.2 Chrome DevTools REPL

A browser-based REPL facilitates experimentation with the language by interested people, from the comfort of their Web browsers. It's also a useful debugging feature of a deployed application.

However, most developers already have a preferred text editor and a REPL interaction workflow, and so it's not within the JACL project scope to build a resident IDE in the style of SLip. Even if a resident IDE was a goal, the file system access restrictions imposed by the browser would present significant challenges.

Instead, JACL offers an alternative development REPL approach that requires minimal host tooling: the DevTools-based REPL. Google Chrome is capable of hosting a WebSocket-based debug server that implements the DevTools Protocol [13]. DevTools Protocol clients may then connect to the server and interact with open tabs, such as by evaluating arbitrary JavaScript within the context of the tab. JACL leverages the DevTools Protocol to deliver a command-line REPL client that may be run on development machines. The workflow is the following:

(1) Run Google Chrome from the shell with the `-remote-debugging-port` parameter.
(2) Navigate to the Web site hosting the JACL system you wish to interact with.
(3) Run `jacl-repl` from the shell.
(4) Be presented with a Lisp prompt.

As a simple command-line application with a textual interface, `jacl-repl` can be run in various contexts. For example, it could be run within an Emacs "inferior-lisp" buffer, and then Lisp forms could be sent from other Emacs buffers for evaluation in the REPL. It could also be run as part of a build process that pipes Lisp sources over the WebSocket for batch compilation. It is anticipated that additional host-side tools that depend on `jacl-client` will be necessary in the future to support loading source files in dependency order.

Unlike the pre-readers of SLip and JSCL, `jacl-client` is completely ignorant of Lisp syntax. `jacl-client` merely transports characters between the host machine and the Lisp system and so is *not* a pre-reader.

There are a few obvious ways the JACL REPL experience could be improved. For example, `jacl-repl` is currently an R[25] script requiring an R installation and the `chromote`[7] package. A standalone binary executable is imagined in the future in order to make it easier for developers to start working on JACL projects. Additionally, JACL has yet to define a printer for its native types, or an extensible print protocol. Object string representations are obtained by calling the generic JavaScript `toString()` method, which doesn't always produce a representation that can be read back in.

3.3 Analyzing compiler

Unlike JSCL, the JACL compiler is organized to facilitate high-level optimizations such as those that could support efficient compilation of `TAGBODY` and other fundamental Common Lisp operators.

The first compiler pass expands macros and produces an AST. The second compiler pass performs optimizations and produces a new AST. The final pass produces JavaScript code. AST nodes are represented by generic JavaScript objects with at least the following keys:

- `op`: The name of the node, as a JavaScript string.
- `env`: An object of class `Env` that represents the lexical environment of the node.
- `parent`: The parent of the node; this is `null` for the root.
- `form`: The original source data of the node, a Lisp datum.

Nodes and Env objects are immutable by convention. Functions are provided for modifying and merging these objects so as only to produce new objects. This convention reduces the possibility of optimization passes interfering with one another. It also eases understanding the AST, since every AST node contains a copy of all relevant context. As JavaScript objects, AST nodes are easily introspected using the object inspector of the Web browser.

Currently, the Env object tracks evaluation context — one of *statement*, *expression*, or *return* — lexical variables, and TAGBODY tags. In the future, it will track the remaining aspects of the lexical environment, such as lexical functions and macros.

3.3.1 Embedding JavaScript with JACL:%JS*.* Unlike JSCL or SLip, 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, 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.

3.3.2 Other interoperation support. 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 compile into 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[2] concisely. The dot macro takes direct inspiration from the .. macro of Clojure. 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.

3.3.3 TAGBODY *compilation strategy.* Consider the following Common Lisp program that decrements the local variable X 10 times:

```
(let ((x 10))
  (tagbody
    start
    (when (zerop x) (go end))
    (setq x (1- x))
    (go start)
    end))
```

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

JSCL, the existing Lisp closest to JACL, would compile the preceding code into approximately[3] the following JavaScript:

```
function Jump(id, label) {
  this.id = id;
  this.label = label;
}

var X = 10;
var id = [];
var label = 0;
LOOP: while (true) {
  try {
    switch(label) {
      case 0:
        if (X === 0) throw new Jump(id, 1);
        X = X-1;
        throw new Jump(id, 0);
      case 1:
      default:
        break LOOP;
    }
  } catch (e) {
    if (e instanceof Jump && e.id === id) {
      label = e.label;
    } else {
      throw e;
    }
  }
}
```

The mechanism is ingenious. GO tags became switch labels, and jumps became throw statements. The thrown objects are instances of Jump. Each instance of Jump contains a destination label.

Unfortunately, in this scheme, every jump requires a JavaScript exception to be thrown, severely penalizing TAGBODY as previously discussed. Fortunately, a straightforward *local jump optimization* can be applied that yields a tremendous performance benefit. Local jump optimization is a known technique[28], but JACL is the first Lisp targeting JavaScript to apply it.

In order to perform this optimization, the JACL compiler first identifies local GOs in its analysis pass. These are GO nodes with no intervening LAMBDA nodes[4] between them and their respective, lexically-enclosing TAGBODYs. Then, TAGBODYs are identified that consist of only local GOs.

JavaScript generated for local GOs does not throw an exception, but instead leverages the labeled form of the JavaScript continue[20] statement to transfer control appropriately. JavaScript generated for TAGBODYs that have been determined to consist only of local jumps omits the try/catch block, saving on generated code size.

The following code is similar[5] to that generated by the JACL compiler. Cursory benchmarks A show JACL code runs several

[3]Actual JSCL output is not used because it includes type checks, generated variable names, and other code that would obscure the relevant machinery.

[4]Note that LAMBDA doesn't necessarily preclude local jump optimization if the LAMBDA is inlined, but JACL currently does not inline functions.

[5]Once more, actual compiler output has been significantly modified and reformatted for brevity.

orders of magnitude faster than JSCL, and that JACL code is almost as fast as the JavaScript statement `while(X-)`:

```
var X = 10;
var label = 0;
LOOP: while (true) {
  switch(label) {
    case 0:
      if (X === 0) {
        label = 1;
        continue LOOP;
      }
      X = X - 1;
      label = 0;
      continue LOOP;
    case 1:
    default:
      break LOOP;
  }
}
```

4 CONCLUSION

We introduced JACL, a new Common Lisp created to ease SPA development. JACL is designed as an efficient, practical tool, with the needs of industrial SPA developers in mind. JACL integrates tightly with the Web browser platform and interoperates easily with JavaScript. Compared to other browser-based Lisps, JACL places a higher emphasis on the value of the REPL, and introduces new techniques for integrating the REPL into the development workflow.

5 FUTURE WORK

In order to be practical for application development, JACL must support the creation of standalone executables. In the case of JACL, these would be single JavaScript files that may be included in an HTML page and are executed on page load. Fortunately, since JACL development is image-based, JACL should support the traditional approach of specifying a Lisp function entrypoint and dumping the Lisp image to native (JavaScript) code. The `SAVE-LISP-AND-DIE`[26] function in SBCL and the `DELIVER`[16] function in LispWorks are two examples of this functionality in other implementations.

JACL should be able to perform rudimentary optimizations such as global function and variable tree shaking[30] in order to reduce the size of generated executables. In addition, JACL should make dynamic function and variable references in executables static, so that third party tools like Google Closure Compiler[5] may optionally be used to perform additional optimization.

Other than the ability to produce optimized standalone executables, many other design and implementation tasks remain, such as support for special variables in lambda lists, `EVAL-WHEN`, macro lambda lists, `DECLARE` et al, CLOS, various other data types, compiler macros, etc. The list of tasks is enormous, but it is anticipated that these features can be implemented over time, in the order demanded by application development.

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.

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

REFERENCES

[1] Marco Baringer. 2005. *Parenscript.* Retrieved February 12, 2020 from https://web.archive.org/web/20051122141019/http://blogs.bl0rg.net/netzstaub/archives/000525.html
[2] Mihai Bazon. 2012-2018. *Implementation notes.* Retrieved February 12, 2020 from http://lisperator.net/slip/impl
[3] Mihai Bazon. 2012-2018. *SLip — a Lisp system in JavaScript.* Retrieved February 12, 2020 from http://lisperator.net/slip/
[4] Mihai Bazon. 2012-2018. *Versus Common Lisp.* Retrieved February 12, 2020 from http://lisperator.net/slip/vscl
[5] Michael Bolin. 2010. *Closure: The Definitive Guide: Google Tools to Add Power to Your JavaScript.* O'Reilly Media, Sebastopol, CA, USA.
[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] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs.* Retrieved February 12, 2020 from https://elm-lang.org/assets/papers/concurrent-frp.pdf
[12] 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
[13] Google, Inc. 2020. *Chrome DevTools.* Retrieved February 12, 2020 from https://developers.google.com/web/tools/chrome-devtools
[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] Benedikt Meurer Maya Lekova. 2018. *Faster async functions and promises.* Retrieved February 12, 2020 from https://v8.dev/blog/fast-async
[19] Mozilla, Inc. 2020. *Concurrency model and the event loop.* Retrieved February 22, 2020 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
[20] Mozilla, Inc. 2020. *label - JavaScript | MDN.* Retrieved February 19, 2020 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label
[21] David Nolen. 2020. *ClojureScript Quick Start.* Retrieved February 12, 2020 from https://clojurescript.org/guides/quick-start
[22] Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case studies in Common Lisp.* Morgan Kaufmann Publishers, San Francisco, CA, USA.
[23] David Vázquez Púa. 2018. Growing a Lisp compiler. Retrieved February 12, 2020 from https://www.youtube.com/watch?v=XT7JYPtWMd8
[24] David Vázquez Púa and contributors. [n.d.]. *jscl-project/jscl.* Retrieved February 12, 2020 from https://github.com/jscl-project/jscl/
[25] R Core Team. [n.d.]. *R: A Language and Environment for Statistical Computing.* http://www.R-project.org/
[26] SBCL Project Contributors. 2020. *SBCL 2.0.1 User Manual.* Retrieved February 21, 2020 from http://www.sbcl.org/manual/
[27] 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

[28] Robert Strandh. 2020. *compile-general-purpose-asts.lisp.* Retrieved February 21, 2020 from https://github.com/robert-strandh/SICL/blob/2d322d3c7794eb4e89720b9a2fce42395a787376/Code/Cleavir2/AST-to-HIR/compile-general-purpose-asts.lisp#L200-L303
[29] 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].
[30] Wikipedia contributors. 2020. Tree shaking — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Tree_shaking&oldid=908332079 [Online; accessed February 22, 2020].

A TAGBODY PERFORMANCE BENCHMARKS

The following benchmark code was run on Google Chrome 80.0.3987.116, on Linux, using a computer with an Intel i7-3520M CPU. Times are in milliseconds.

```
function Jump(id, label) {
  this.id = id;
  this.label = label;
}

function tagbody_unoptimized(X) {
  var id = [];
  var label = 0;
  LOOP: while (true) {
    try {
      switch(label) {
        case 0:
          if (X === 0) throw new Jump(id, 1);
          X = X - 1;
          throw new Jump(id, 0);
        case 1:
        default:
          break LOOP;
      }
    } catch (e) {
      if (e instanceof Jump && e.id === id) {
        label = e.label;
      } else {
        throw e;
      }
    }
  }
}

function tagbody_optimized(X) {
  var X = 10;
  var label = 0;
  LOOP: while (true) {
    switch(label) {
      case 0:
        if (X === 0) {
          label = 1;
          continue LOOP;
        }
        X = X - 1;
        label = 0;
        continue LOOP;
      case 1:
      default:
        break LOOP;
    }
  }
}

function baseline_js(X) {
  while(X--);
}

var start = performance.now();
for (var i = 0; i < 1e6; i++) tagbody_unoptimized(10);
console.log("tagbody_unoptimized", performance.now() - start);
// tagbody_unoptimized 58994.43500000052

var start = performance.now();
for (var i = 0; i < 1e6; i++) tagbody_optimized(10);
console.log("tagbody_optimized", performance.now() - start);
// tagbody_optimized 23.61499999812804

var start = performance.now();
for (var i = 0; i < 1e6; i++) baseline_js(10);
console.log("baseline_js", performance.now() - start);
// baseline_js 15.055000003427267
```