Shrinking JavaScript for CS1

Boyd Anderson
Martin Henz
Kok-Lim Low
Daryl Tan
National University of Singapore

Abstract

In teaching and learning programming at first-year-university level, simple languages with small feature sets are preferable over industry-strength languages with extensive feature sets, to reduce the learners' cognitive load. At the same time, there is increasing pressure to familiarise students with mainstream languages early in their learning journey, and these languages accumulate features as years go by. In response to these competing requirements, we developed Source, a collection of JavaScript sublanguages with feature sets just expressive enough to introduce first-year computer science students to the elements of computation. These languages are supported by a web-based programming environment custom-built for learning at beginner's level, which provides transpiler, interpreter, virtual machine, and algebraic-stepper-based implementations of the languages, and includes tracing, debugging, visualization, type-inference, and smart-editor features. This paper motivates the choice of JavaScript as starting point and describes the syntax and semantics of the Source languages compared to their parent language, and their implementations in the system. We report our experiences in developing and improving the languages and implementations over a period of three years, teaching a total of 1561 computer science first-year students at a university.

CCS Concepts: • **Software and its engineering → General programming languages**; *Integrated and visual development environments.*

Keywords: teaching programming, JavaScript, learning tools, learning environments

ACM Reference Format:
Boyd Anderson, Martin Henz, Kok-Lim Low, and Daryl Tan. 2021. Shrinking JavaScript for CS1. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3484272.3484970

SPLASH-E '21, October 20, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9089-7/21/10.
https://doi.org/10.1145/3484272.3484970

1 Introduction and Related Work

1.1 The Problem with Large Languages

Programming languages are like trees: they keep growing until they die (and they usually die slowly). With a rise in popularity of a language comes an increasing pressure on the language designers to undertake only backwards-compatible changes, which means that existing features cannot be removed. Thus, programming languages rarely shrink: once a feature is introduced it will stay until the language dies. A notable exception was the introduction of Python 3.0 in 2008, which was not backwards-compatible with Python 2 and which removed several language features. Language designers face continuous demands from their user communities and in response regularly add new features while making sure that they are distinguishable from existing features to retain backwards compatibility. Over the years, languages grow by accumulating new features. For example, Java's latest specification [11] runs at 844 pages, and JavaScript's [12] at 860 pages. Novice students in a Java-based course justifiably ask: "Should I learn *all of* Java to excel in the course?"

On the other hand, schools are facing pressure from students and industry to teach languages that are of industrial relevance, as a result of short-term considerations such as finding internships/interns as well as medium-term considerations involving employability and graduate skill sets. For example, "Industrial relevance" is cited as the second-most important factor in the choice of first-year programming language by instructors in Australasia, only marginally behind "pedagogical benefits" [22].

For instructors, large languages pose several challenges. Professional language implementations and tools need to remain standard-compliant and grow with the languages and therefore become harder to master. Some language features are difficult to avoid as they pervade the language design and usage. As an example, the introduction of generic types in Java made it harder to use Java in first-year courses. The most problematic aspect of large languages for teaching is classroom control. First-year courses today enroll complete novices as well as learners with ten years or more of programming experience. The latter tend to use language features that might be tangential to the learning outcome

of a particular course segment. In this situation, instructors face the dilemma either to explain the feature, thereby deviating from the learning objective, or to discourage the use of the feature without much explanation, thereby leaving the students' justified curiosity unsatisfied.

1.2 Our Starting Point: SICP

The book "Structure and Interpretation of Computer Programs (SICP) by Abelson and Sussman [1] introduces the reader to central ideas of computation by establishing a series of mental models for computation. Chapters 1–3 cover programming concepts that are common to all modern high-level programming languages. The original first two editions of SICP use the programming language Scheme in their program examples, whose minimalistic, expression-oriented syntax allows the book to focus on the underlying ideas rather than the design of the chosen language. Chapters 4 and 5 use Scheme to formulate language processors for Scheme, deepening the readers' understanding of the mental models and exploring language extensions and alternatives.

Since its publication in 1984, SICP has been adopted as textbook by universities and colleges around the world, including the National University of Singapore (NUS), which introduced the SICP-based first-year course CS1101S in 1997. In the mid-1990s, the languages Python, JavaScript, and Ruby emerged, which share central design elements with Scheme, but which employ a more complex, statement-oriented syntax that uses familiar algebraic (infix) notation. Their rise in popularity led instructors to adapt their SICP-based courses, typically by translating the example programs to their language of choice, by adding material specific to that language, and by removing the material of chapters 4 and 5.

Recognizing the fundamental soundness of SICP's pedagogy and universality of learning objectives, we set out to modernize the material in its entirety. Along the way, we saw two opportunities to improve the material, which we will further discuss in Sections 2 and 3:

- Most modern languages are statement-oriented, whereas Scheme is expression-oriented. We set out to modernize SICP by explaining the interpretation of statement-oriented languages, including the treatment of return statements in these languages.
- Scheme's minimalistic design provides a direct representation of programs as data structures, which cannot be taken for granted in modern languages. We wanted to keep the language processors of Chapters 4 and 5 that make use of programs as data structures, but explain how those language processors can be implemented in modern languages.

The JavaScript Edition of SICP [2], to be published in 2022, resulted from our efforts and includes these improvements.

1.3 The Choice of JavaScript

The SICP approach does not require Scheme, but the informal and liberal treatment of data structures makes it easier to adapt SICP to a dynamically typed language with automatic memory management such as Python, Ruby, Clojure, Lua, and JavaScript. Among the most popular of these languages, Python and JavaScript, we chose JavaScript because its block-scoped constant and variable declarations match Scheme's scoping better than Python's function scope, because JavaScript's standard specifies proper tail calls, and because Python does not syntactically distinguish variable declaration from assignment, which makes it more difficult to limit the discourse to purely functional programming in Chapters 1 and 2. The language standard ECMAScript 2015 introduced lambda expressions, tail recursion, and block-scoped variables and constants, which enabled the course material to move closer to the original Scheme-based version.

1.4 Shrinking JavaScript

In order to cope with the challenges posed by large languages listed in Section 1.1, we decided to systematically restrict the use of language features to match the pedagogical requirements of SICP. Since the switch of our CS1101S to JavaScript in 2012, we found the need to gradually introduce language features and prevent the use of features that were not introduced yet. This would allow us to control the classroom and provide a level playing field across a wide variety of students' prior knowledge. Some restrictions such as avoiding the == operator were proposed by Douglas Crockford in [5], who also developed the linting tool JSlint to enforce such restrictions.

We decided to follow a more comprehensive approach than linting and instead to invest in a full-fledged, course-specific web-based programming environment called *Source Academy* [23]. Initially, we introduced a larger language in each week of the course, so Source Week 4 would include only those features needed until Week 4 of the semester, and the Source Academy would only provide those features when Source Week 4 was chosen. To reduce our administrative overhead and the students' cognitive load, we limited the number of languages to four in 2018. Each chapter x of SICP comes with a JavaScript sublanguage that we shall call Source §x.

1.5 Other Related Work

The first systematic shrinking of a programming language for teaching was undertaken in 1974 for the language PL/I [15]. The DrScheme implementation of Scheme systematically restricted the language features [8] to follow the textbook "How to Design Programs" by Felleisen et al. [7]. The textbook "Functional JavaScript" [9] introduces functional programming using the language JavaScript without physically restricting the user to any sublanguage. The language Grace

also gave rise to a sequence of sublanguages designed for the needs of teachers and learners [16].

1.6 Overview

Section 2 introduces the language needs of each SICP chapter and describes the respective JavaScript sublanguage. Section 3 introduces the semantic concepts pursued by the SICP chapters and how we present these concepts to the users of the Source Academy. Section 4 describes the language-specific pedagogical features of the system that we built to support the readers of SICP. Section 5 discusses programming language and system implementation aspects of the Source Academy, and Section 6 reports our experiences with introducing this system and improving it over three consecutive years.

2 Shrinking the Syntax of JavaScript

The languages Source §x are sublanguages of JavaScript in the sense that every Source §x program is a JavaScript program. The languages are designed to be just expressive enough to handle the programs of SICP, up to and including the chapter in which the programs appear. The number of syntactic forms used in the Source languages is as small as the number of special forms used in the corresponding chapter of SICP and thus, in the words of Felleisen et al [6], Source serves to "liberate the introductory course from the tyranny of syntax".

2.1 Source §1

Chapter 1 of SICP introduces a purely functional language with numbers, boolean values and strings, but without any other data structures. The syntax of Source §1 depicted in Figure 1 deviates from Scheme in the following ways:

- The language is block-structured. The scope of constant and function declarations is the immediately surrounding block.
- The language is statement-oriented, following most modern languages, including Python, C and Java. Return statements can appear anywhere in the body of a function.
- In Scheme, the branches of conditionals **`cond`** and **`if`** can contain local declarations, whereas JavaScript's conditional expressions cannot.[1] Therefore, we include conditional statements in addition to conditional expressions. The branches of conditional statements are blocks whose declarations are local to the block.
- Similarly, we allow for the block body variant of lambda expressions, to achieve the generality of lambda expressions in Scheme.
- Arithmetic expressions use infix notation, which is much more common in modern languages than the prefix notation use in Scheme.

[1] Technically, expressions can include declarations by using the "immediately invoked function expression" pattern, but this is syntactically cumbersome.

2.2 Source §2

Following Scheme's minimalist syntax, Source §1 does not include pattern matching supported by functional languages such as OCaml and Haskell. As a result, the programs in SICP JS Chapter 2 look and feel like their counterparts in SICP, and Source §2 only adds the primitive literal expression `null`.

expression ::= *number* | **`true`** | **`false`** | *string* | **`null`** | ...

2.3 Source §3–5

We exploit the neat syntactic distinction of ECMAScript 2015 between two block-scoped kinds of names: constants (declared with **`const`**) and variables (declared with **`let`**). To discuss imperative programming in SICP Chapter 3, Source §3 introduces variable declaration and assignment, and extends the syntax of Figure 1 as shown in Figure 2. Our version of Source §3 adds syntactic support for arrays and **`while`** and **`for`** loops, which are not used in SICP Chapter 3, but mentioned in passing in SICP JS 4.1.2, in order to augment Chapter 3's treatment of imperative programming with idiomatic examples. Chapters 4 and 5 do not necessitate the introduction of syntactic constructs beyond Source §3.

3 Shrinking the Semantics of JavaScript

The semantics of the Source languages are faithful to the semantics of JavaScript in the sense that if a program evaluates to a result without error according to the Source semantics, it evaluates to the same result without error according to the JavaScript semantics. Nothing "works" in Source that doesn't "work" in the same way in JavaScript. This means that the programming skills that students acquire when using Source directly transfer to JavaScript. Crockford [5] criticizes the excessively liberal coercion and overloading of JavaScript's operators. We take the liberty to restrict them such that the operators <, >,..., and + either take two numbers or two strings as operands. Any other operand type combinations lead to an error. This rarely causes problems, but simplifies the language significantly and reduces the cognitive load on the students.

3.1 Source §1

SICP informally introduces the *substitution model* as a mental model for the execution of purely functional Scheme programs. We provide a semantic foundation for Source §1–2 in [4] that formalizes the substitution model using term graph rewriting.

program	::=	*import-directive* ... *statement* ...	program
import-directive	::=	**import {** *names* **} from** *string* **;**	import directive
names	::=	ϵ | *name* (**,** *name*) ...	list of names
statement	::=	**const** *name* **=** *expression* **;**	constant declaration
	|	**function** *name* **(** *names* **)** *block*	function declaration
	|	**return** *expression* **;**	return statement
	|	*if-statement*	conditional statement
	|	*block*	block statement
	|	*expression* **;**	expression statement
if-statement	::=	**if (** *expression* **)** *block*	
		else (*block* | *if-statement*)	conditional statement
block	::=	**{** *statement* ... **}**	block statement
expression	::=	*number* | **true** | **false** | *string*	primitive literal expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression* **(** *expressions* **)**	function application
	|	(*name* | **(** *names* **)**) **=>** *expression*	lambda expression (expression body)
	|	(*name* | **(** *names* **)**) **=>** *block*	lambda expression (block body)
	|	*expression* **?** *expression* **:** *expression*	conditional expression
	|	**(** *expression* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | **%** | **===** | **!==**	
	|	**>** | **<** | **>=** | **<=** | **&&** | **||**	binary operator
unary-operator	::=	**!** | **-**	unary operator
expressions	::=	ϵ | *expression* (**,** *expression*) ...	argument expressions

Figure 1. Syntax of Source §1, in Backus-Naur Form [18] with **bold** font for keywords, *italics* for nonterminal symbols, ϵ for nothing, $x \mid y$ for x or y, and $x \ldots$ for zero or more repetitions of x

statement	::=	...	
	|	**let** *name* **=** *expression* **;**	variable decl.
expression	::=	...	
	|	*name* **=** *expression*	variable assgmt

Figure 2. Syntax of Source §3, an extension of the BNF of Figure 1

3.2 Source §2

SICP Chapter 2 "Building Abstractions with Data" introduces only the most minimalist data structure, the pair. Source §2 follows suit by introducing the constructor `pair` and the selectors `head` and `tail` (traditionally called `car` and `cdr` in the Lisp/Scheme communities). We resisted the introduction of objects, literal or otherwise, and thus sidestepped the complexities of JavaScript's support for object-oriented programming, such as the "prototype chain". Any program that uses literal objects, or object property access using "." (dot) is rejected by the Source parser. The memory model of Source includes only pairs and is described in SICP JS Section 5.3, including a stop-and-copy garbage collector comprising less than 60 register machine instructions, given in SICP JS 5.3.2.

3.3 Source §3

The semantic foundation of Source §3 is introduced informally in SICP JS Section 3.2, and operationalized in Section

4.1 in form of a metacircular evaluator. Most syntactic constructs of Source §3 come with their own evaluation function, and the main evaluation function `evaluate` given in Figure 3 dispatches to them using syntax predicates, see Section 3.4. Figure 4 shows the function `apply` that implements function calls. The mutually recursive functions `evaluate` and `apply` form the core of the evaluator. This evaluator is similar to the metacircular evaluator for Scheme given in SICP Section 4.1, with the notable exception that return statements, also given in Figure 4, construct explicit return values which are used in `apply` to return the value contained in the return value, or return `undefined` instead.

3.4 Source §4-5

By using JavaScript, chapters 1–3 introduce the reader to the syntactic style of most mainstream languages today. However, that same syntactic style gave rise to significant changes in chapters 4 and 5 of SICP JS, because the direct representation of programs as data structures could no longer be taken for granted. This provided SICP JS with an opportunity to introduce the notion of program parsing in section 4.1.2, an important component of programming-language processors. Source §4 adds the function `parse` to the language, which specifies a tagged-list representation of syntax trees, following the BNF from Figures 1 and 2. The parse rules are informally described in SICP JS Section 4.1.2, along with the syntax predicates and selectors used in `evaluate` in Figure 3. The formal specification of `parse` is aided by restrictions in the syntax of Source, compared to JavaScript:

- no optional semicolons,
- no object or classes, and
- no switch, exception handling

4 Learner-Focused Development Environment

The Source Academy is equipped with several tools and features to support independent, as well as collaborative, learning of SICP. The most important among them are the algebraic stepper, the data visualizer, and the environment visualizer, as they can provide powerful visual aids and feedback for the study of the three central ideas in Chapters 1 to 3 of SICP, namely, the substitution model, data representation and abstraction, and the environment model.

These tools have similar purposes to the inspection and visualization tools of the BlueJ system [17], where users are able to visualize the state and evolution of program execution.

4.1 Algebraic Stepper

Following the semantics of Source §1–2 in Sections 3.1 and 3.2 we developed an algebraic stepper [14] that lets the user interactively discover the sequence of substitution steps that emerge from the evaluation of a program. The tool proves invaluable to connect students with no programming background to the familiar domain of mathematics. Figure 5 shows an interactive stepper session in the programming environment.

4.2 Data Visualizer

A data visualization tool implements *box-and-pointer diagrams*—the graphical data model of SICP §2—following the example of numerous similar tools in software engineering and learning environments, for example the Java visualizer for IntelliJ [20] and Racket's Sdraw library [21]. Figure 6 shows the visualizer displaying the result of symbolic differentiation according to SICP 2.3.2.

4.3 Environment Visualizer

To help with the understanding of the environment model in SICP, students can make use of an environment visualizer. They can inspect a Source program's current execution state by setting breakpoints before the relevant program lines. Execution will be paused at a breakpoint, displaying the current environment frames in the fashion described in SICP 3.2. Execution can be resumed afterwards. Figure 7 shows an interactive session with the environment visualizer. The environment visualizer achieves for the environment model what the Java visualizer for IntelliJ [20] achieves for the execution model of the Java Virtual Machine.

4.4 Robotics

We have designed a series of imperative programming exercises in which students are required to build LEGO MINDSTORMS EV3 [19] robots and write Source §3 programs to control the robots to accomplish a set of assigned tasks. The Source Academy provides the capability to wirelessly upload their solutions from the web browser to the EV3 Bricks for execution. See [3] how this feature is used in online teaching.

5 Implementation

The Source Academy is a medium-scale free-software project, whose GitHub organization [10] currently comprises 139 developers, mostly undergraduate students of our university who carry out software project courses or independent projects by contributing to the system. The programming language implementation of Source is kept independent from the programming environment for future extensibility.

5.1 Parsing

Since any Source program is also a valid JavaScript program, we parse and generate its abstract syntax tree using Acorn, an open-source JavaScript parser [13]. We then check for any disallowed JavaScript syntax. If any are present, an error is thrown and the offending constructs listed out. The final output is a validated abstract syntax tree.

```
function evaluate(component, env) {
   return is_literal(component)
          ? literal_value(component)
          : is_name(component)
          ? lookup_symbol_value(symbol_of_name(component), env)
          : is_application(component)
          ? apply(evaluate(function_expression(component), env),
                  list_of_values(arg_expressions(component), env))
          : is_operator_combination(component)
          ? evaluate(operator_combination_to_application(component), env)
          : is_conditional(component)
          ? eval_conditional(component, env)
          : is_lambda_expression(component)
          ? make_function(lambda_parameter_symbols(component),
                          lambda_body(component), env)
          : is_sequence(component)
          ? eval_sequence(sequence_statements(component), env)
          : is_block(component)
          ? eval_block(component, env)
          : is_return_statement(component)
          ? eval_return_statement(component, env)
          : is_function_declaration(component)
          ? evaluate(function_decl_to_constant_decl(component), env)
          : is_declaration(component)
          ? eval_declaration(component, env)
          : is_assignment(component)
          ? eval_assignment(component, env)
          : error(component, "unknown syntax -- evaluate");
}
```

Figure 3. Main function `evaluate` of a metacircular evaluator for Source §4

5.2 Algebraic Stepper

The stepper implements the semantics of Source §1–2, which is based on term-graph rewriting rather than term (tree) rewriting, because as evaluation proceeds, the syntax trees evolve into possibly cyclic graphs in order to capture the semantics of block-scoped names and recursive functions. The implementation stores the syntax graph for each step in an array and lets the user access the steps with a slider as shown in Figure 5. This is possible even for computations with several thousand steps because there is significant sharing between the graphs. For more details on the implementation of the stepper, see [14].

5.3 Environment Visualizer

The environment visualizer is built on top of an interpreter for Source. Like the metacircular evaluator described in Section 3.3, our interpreter keeps track of environment frames and their variables. To ensure that our interpreter is tail recursive, whenever a function call is made in the tail position, a trampolining [24] version is returned instead of the function call.

5.4 Robotics

As we only support Source §3 programs and not all of JavaScript, we were able to build a virtual machine language for running our programs with moderate effort, known as the Source Virtual Machine Language (SVML) and its interpreter (Sinter). This interpreter is written in C++ and can be ported to many other architectures. The SVML compiler is written in TypeScript and therefore can be run in a web browser.

Our robotics system consists of four main components: SVML, Sinter, the Source-to-SVML compiler, and an MQTT-based protocol/library to transfer programs to a robot and return the output of any computation (Sling). These components (SVML, Sinter, and Sling) are included in a customized version of the ev3dev linux distro running on the LEGO MINDSTORMS EV3 robot platform. This customized image

```
function apply(fun, args) {
   if (is_primitive_function(fun)) {
      return apply_primitive_function(fun, args);
   } else if (is_compound_function(fun)) {
      const result = evaluate(function_body(fun),
                              extend_environment(
                                  function_parameters(fun),
                                  args,
                                  function_environment(fun)));
      return is_return_value(result)
             ? return_value_content(result)
             : undefined;
   } else {
      error(fun, "unknown function type -- apply");
   }
}
function eval_return_statement(component, env) {
    return make_return_value(
               evaluate(return_expression(component), env));
}
```

Figure 4. Functions apply and eval_return_statement

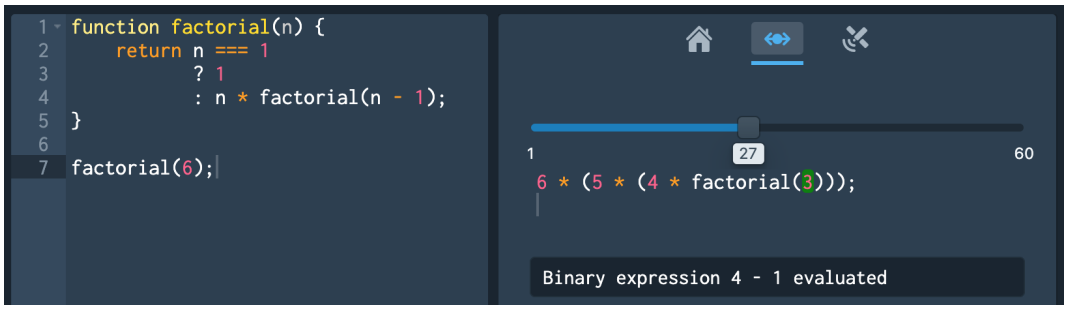

Figure 5. Stepping through the substitutions that evaluate a program, according to the Source §1–2 semantics

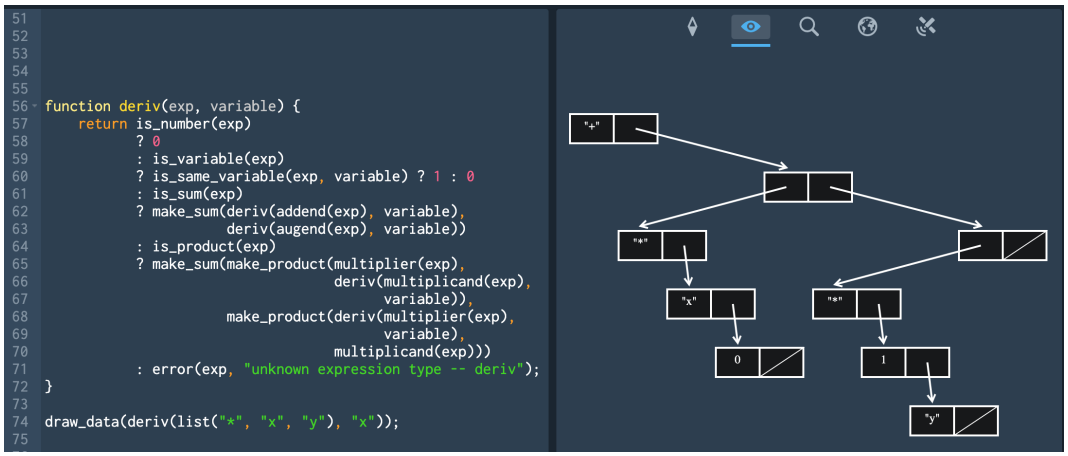

Figure 6. Displaying the result of symbolic differentiation of xy/dx (non-simplifying version)

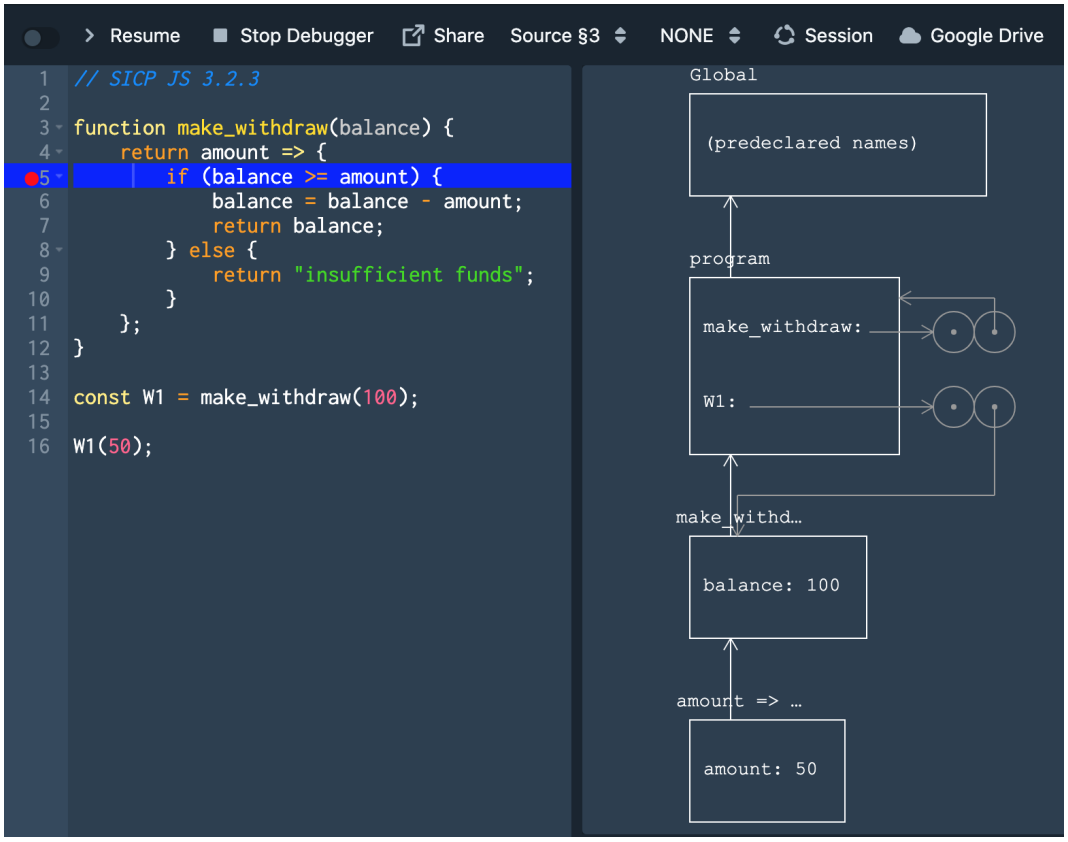

Figure 7. The environment visualizer showing the environment of a program at a breakpoint

is flashed on to the EV3 computer and then the device generates a UUID which can be registered on the Source Academy. On the EV3 robot, Sling connects to the backend and awaits instructions. Programs are then compiled to SVML in the web browser and sent to the EV3 robot. These programs are interpreted by Sinter, and any output is sent back to the web browser. See [3] for more details on the implementation of our robotics system.

5.5 Transpiler

Even though we have an interpreter for Source used by the environment visualizer, it was too slow for computationally intensive tasks. Therefore, we decided to execute Source programs using the browsers' JavaScript engine instead, leading to a decrease in execution time by a factor of around 50.

A tail-recursive implementation—a feature known as *proper tail calls*—is necessary for the concept of iterative processes in SICP, but, although specified by the ECMAScript standard since 2015, Safari is the only modern browser as of 2021 whose JavaScript engine implements this feature. Therefore, we implement proper tail calls using a trampoline function [24] after syntactically detecting tail calls and transforming them into trampolining versions. To restrict JavaScript's operators as mentioned in Section 3, we transpile operator combinations into function calls that perform the necessary tests.

The transpiled JavaScript code is executed via JavaScript's `eval`. The execution of third-party Source programs is safe from cross-site scripting attacks, because we ensure that no JavaScript global variables can be referenced in Source and prevent the use of JavaScript features such as cookies and HTTP requests.

6 Experiences and Future Work

Motivated by the need to adapt the SICP textbook to a modern, mainstream programming language, we have demonstrated how to shrink the real-world language JavaScript to a set of features that matches the minimalistic design of Scheme, with and for which SICP was originally written. Constraining programming novices to small sublanguages has a liberating effect on the teaching and learning process:

- Students with little or no programming background are liberated from the demotivating presence of students with years of programming experience. As an example, the tiny language Source §1 in Section 2 is sufficient for the first four weeks of a rigorous CS1 course.

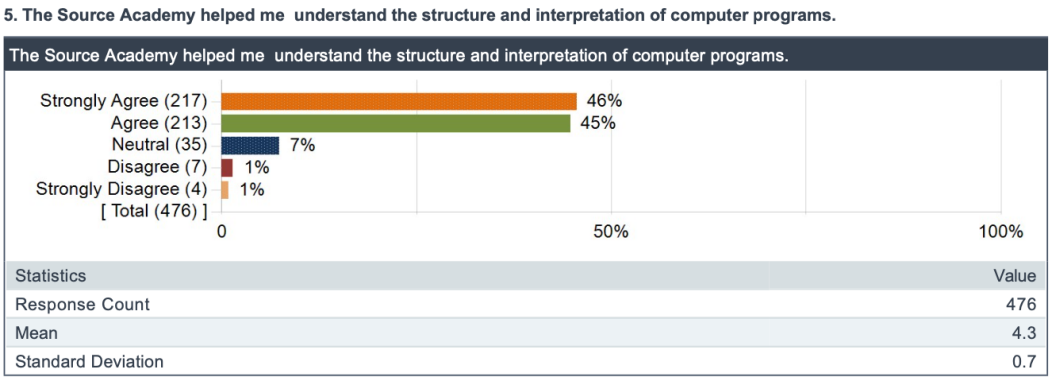

Figure 8. Student feedback in 2020 on how the course's vehicle for shrinking JavaScript, the Source Academy, contributed to the learning

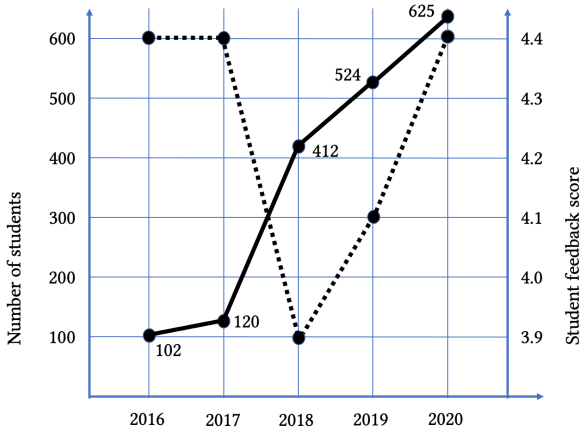

Figure 9. Student numbers (solid line; CS1101S became mandatory for CS majors from 2018); student feedback scores for "What is your overall opinion of CS1101S?" (dashed line; 1–5 Likert scale, from 1: "very poor" to 5: "very good")

- Instructors are liberated from the need to cater to student questions on language features that are not contributing to the learning objectives at hand. For example, none of the Source languages need to touch on the complexities of object-oriented programming.
- Developers of tools are liberated from the suffocating need to implement the huge feature sets of industry-strength programming languages. Development of useful learning tools becomes a realistic objective for semester-sized undergraduate student projects. All tools described in Section 4 were implemented by undergraduate students in project-based software engineering or programming language implementation courses.

The design of the syntaxes and semantics of the sublanguages, their implementations and language tools are battle-tested in three successive years (2018–2020) of using continuously improved versions in CS1101S. Prior to 2018, the course was an opt-in alternative to a CS1 course that did not follow SICP, and CS1101S had only a fraction of the cohort of computer science first-year students. That fraction was self-selected and unrepresentative of the cohort, so student feedback scores for 2016 and 2017 are not comparable to the scores of 2018–2019. Figure 9 shows the number of students and overall student feedback scores. Student feedback on CS1101S is favorable, especially in 2020, but we do not have direct evidence how the shrinking of JavaScript contributed to the success of the course. However, the main vehicle for shrinking JavaScript from the point of view of the students is the Source Academy, and Figure 8 shows how students perceived the system having contributed to their mastery of SICP concepts in 2020. In 2020, 42 students mentioned "Source Academy" in their course feedback under "What I liked about CS1101S". There were four students who mentioned "Source Academy" under "What I did not like about CS1101S", and two of these were attributable to instructor mistakes.

The design of the Source Academy separates the programming environment from the language implementation, which allows us to reuse the language-independent components for the shrinking of other mainstream languages. Pedagogically-motivated sublanguages of Python, TypeScript, and R are currently under development and will plug into the frontend in the same way as the JavaScript implementations described in Section 5. An effort to shrink TypeScript is also under way.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press, Cambridge, MA.

[2] Harold Abelson and Gerald Jay Sussman. 2022. *Structure and Interpretation of Computer Programs, JavaScript edition.* MIT Press, Cambridge, MA. Adapted to JavaScript by Martin Henz and Tobias Wrigstad with Julie Sussman.

[3] Boyd Anderson, Martin Henz, and Hao-Wei Tee. 2021. Ruggedizing CS1 Robotics: Tools and Approaches for Online Teaching. In *Proceedings of the 2021 ACM SIG-PLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA. https://doi.org/10.1145/3484272.3484969

[4] Zachary Chua, Martin Henz, Peter Jung, Thomas Tan, Yee-Jian Tan, Xinyi Zhang, and Jingjing Zhao. 2021. *Specification of Source §2 Stepper.* Source Academy Specifications. National University of Singapore. https://docs.sourceacademy.org/source_2_stepper.pdf.

[5] Douglas Crockford. 2008. *Functional JavaScript.* JavaScript: The Good Parts, Sebastopol, CA.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The Structure and Interpretation of the Computer Science Curriculum. *Journal of Functional Programming* 14, 4 (2004), 365–378. https://doi.org/10.1017/S0956796804005076

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (2nd ed.). MIT Press, Cambridge, MA.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *Journal of Functional Programming* 12, 2 (January 2002), 129–132. https://doi.org/10.1017/S0956796801004208

[9] Michael Fogus. 2013. *Functional JavaScript.* O'Reilly Media, Inc., Sebastopol, CA.

[10] Github `source-academy` organization. 2021. Source Academy. https://github.com/source-academy.

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. *The Java® Language Specification, Java SE 16 Edition.* Oracle America, Inc., Redwood Shores, CA.

[12] Jordan Harband and Kevin Smith (Eds.). 2020. *ECMAScript 2020 Language Specification* (11th ed.). Ecma International, Geneva.

[13] Marijn Haverbeke. 2021. Acorn: A small, fast, JavaScript-based JavaScript parser. `acornjs` GitHub organization, https://github.com/acornjs/acorn.

[14] Martin Henz, Thomas Tan, Zachary Chua, Peter Jung, Yee-Jian Tan, Xinyi Zhang, and Jingjing Zhao. 2021. A Stepper for a Functional JavaScript Sublanguage. In *Proceedings of the 2021 ACM SIG-PLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA. https://doi.org/10.1145/3484272.3484968

[15] Richard C. Holt and David B. Wortman. 1974. A sequence of structured subsets of PL/I. *ACM SIGCSE Bulletin, Proceedings of the fourth SIGCSE technical symposium on Computer science education* 6, 1 (January 1974), 129–132. https://doi.org/10.1145/800183.810456

[16] M. Homer, T. Jones, J. Noble, K.B. Bruce, and A.P. Black. 2014. Graceful Dialects. In *ECOOP 2014—Object-Oriented Programming (Lecture Notes in Computer Science)*, Jones R. (Ed.), Vol. 8586. Springer, Berlin, Heidelberg, 131–156. https://doi.org/10.1007/978-3-662-44202-9_6

[17] Michael Kölling. 2008. Using BlueJ to introduce programming. In *Reflections on the Teaching of Programming*, Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.). Lecture Notes in Computer Science, Vol. 4821. Springer, Berlin, Heidelberg, 98–115. https://doi.org/10.1007/978-3-540-77934-6_9

[18] Henry Ledgard. 1980. A human engineered variant of BNF. *ACM SIGPLAN Notices* 15, 10 (oct 1980), 57–62. https://doi.org/10.1145/947727.947732

[19] LEGO System A/S. 2021. LEGO MINDSTORMS. (March 2021). https://www.lego.com/en-us/themes/mindstorms.

[20] Eli Lipsitz. 2019. Java Visualizer—plugin for IntelliJ IDEA. https://github.com/elipsitz/java-visualizer-intellij-plugin.

[21] Jack Rosenthal. 2019. Sdraw: Cons-Cell Diagrams with Pict. https://docs.racket-lang.org/sdraw/index.html.

[22] Simon, Raina Mason, Tom Crick, James H. Davenport, and Ellen Murphy. 2018. Language Choice in Introductory Programming Courses at Australasian and UK Universities. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18).* ACM, Baltimore, Maryland, 852–857. https://doi.org/10.1145/3159450.3159547

[23] Source Academy Organization. 2021. Source Academy. https://about.sourceacademy.org.

[24] David Tarditi, Peter Lee, and Anurag Acharya. 1992. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems* 1, 2 (jun 1992), 161–177. https://doi.org/10.1145/151333.151343