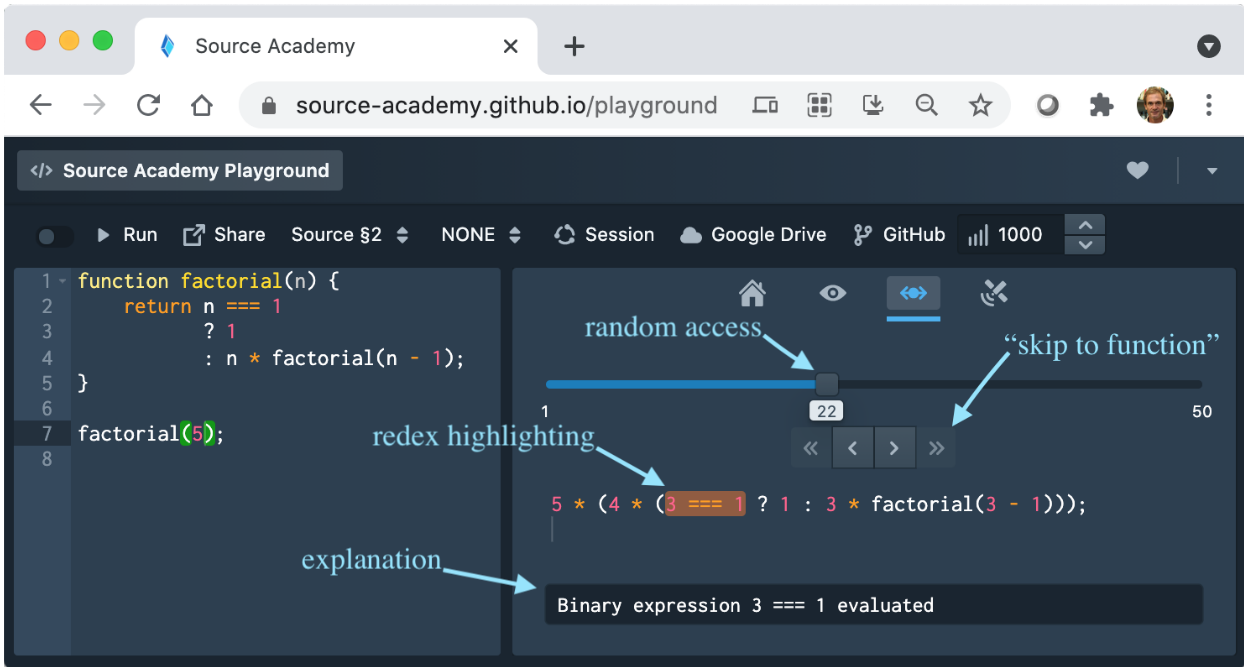
A Stepper for a Functional JavaScript Sublanguage

Martin Henz
Thomas Tan
Zachary Chua
Peter Jung
Yee-Jian Tan
Xinyi Zhang
Jingjing Zhao
henz@comp.nus.edu.sg,{e0177215,e0543984,e0552326,e0325774,e0424857,e0424694}@u.nus.edu
National University of Singapore (NUS)



```
function factorial(n) {
    return n === 1
           ? 1
           : n * factorial(n - 1);
}

factorial(5);
```

```
5 * (4 * (3 === 1 ? 1 : 3 * factorial(3 - 1)));
```

```
Binary expression 3 === 1 evaluated
```

Figure 1. The stepper, showing the step just after the third call of `factorial`, written in JavaScript.

Abstract

The first two chapters of the introductory computer science textbook *Structure and Interpretation of Computer Programs, JavaScript Edition (SICP JS)*, use a subset of JavaScript called *Source §2*. The book introduces the reduction-based "substitution model" as a first mental model for the evaluation of Source §2 programs. To support the learner in adopting this mental model, we built an *algebraic stepper*—a tool for visualizing the evaluation of Source §2 programs according to the model. As a sublanguage of JavaScript, Source §2 differs from other purely functional programming languages by using a statement-oriented syntax, with statement sequences, return statements, and block-scoped declarations. For the purpose of this tool description, we distill these distinguishing features—along with explicit recursion—into a Source §2 sublanguage that we call *Source §0*, and focus on a stepper for this language. We formalize the substitution model of Source §0 as a lambda-calculus-style reduction semantics that handles explicit recursion by term graph rewriting and faithfully implements the JavaScript specification, when restricted to that language. Our implementation of the stepper represents term graphs by persistent data structures that maximize

SPLASH-E '21, October 20, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9089-7/21/10.
https://doi.org/10.1145/3484272.3484968

sharing and enable random access to all steps. This work presents the first reduction-based semantics for a JavaScript sublanguage and the first algebraic stepper for a language with return statements and block-scoped declarations. The tool supports the learner with step-level explanations, redex highlighting, and function-level skipping and can also be used for teaching the applicative-order-reduction lambda calculus.

CCS Concepts: **• Software and its engineering → Integrated and visual development environments; • Applied computing → Computer-assisted instruction; • Theory of computation →** *Lambda calculus.*

Keywords: programming environments, education, functional languages, lambda calculus, semantics, JavaScript, stepper, term graph rewriting

ACM Reference Format:
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 SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3484272.3484968

1 Introduction

A hallmark of the introductory computer science textbook *Structure and Interpretation of Computer Programs* (SICP, [1]) is its *functional-programming-first* approach. Computation is introduced as a step-by-step process of simplification and function unfolding, reminiscent of familiar mental models in mathematical calculation and equation solving. In the book, this process is called the *substitution model*, and can in principle explain the evaluation of all programs in its first two chapters. The JavaScript adaptation of SICP—SICP JS [2]—uses JavaScript for all programs instead of the language Scheme used in the first two editions of SICP. JavaScript employs a statement-oriented syntax, which is the syntactic style of most mainstream languages as of 2021, including Java, Python, and C, rather than Scheme's expression-oriented syntax. The most significant change in the first two chapters of SICP JS, compared to SICP, is the use of return statements and block-scoped declarations.

While teaching Computer Science first-year students using SICP JS, we saw the need for a tool that explains the evaluation of programs in the first two chapters according to the substitution model. Existing JavaScript implementations, including the browsers' JavaScript debugging tools, are not suitable for this task, because the substitution model radically differs from their underlying computational models. Section 2 specifies Source §0, a sublanguage of JavaScript that includes the most important features that distinguish JavaScript from Scheme. Section 3 specifies its formal, lambda-calculus-style reduction semantics, and Section 4 describes the notion of substitution employed in the

```
 prog   ::=  stmt ...                        program
 stmt   ::=  function name
             ( names ) block                 function decl.
          |  return expr ;                   return statement
          |  expr ;                          expression st'ment
names   ::=  ε | name ( , name ) ...         parameters
block   ::=  { prog }                        function body
 expr   ::=  number | true | false
          |  undefined                       literal expression
          |  name                            name expression
          |  expr bin-op expr                binary op. comb.
          |  expr ( exprs )                  function appl.
          |  expr ? expr : expr              conditional expr.
          |  ( expr )                        parenthesized expr.
bin-op  ::=  + | - | * | / | ===
          |  > | < | >= | <=                 binary operator
exprs   ::=  ε | expr ( , expr ) ...         argument expr's
```

Figure 2. Source §0 in Backus-Naur Form [18]; **bold** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, and (x) . . . for zero or more repetitions of x.

semantics. Section 5 outlines how this semantics is implemented in our stepper and how the tool is integrated in the *Source Academy*, our online environment for teaching SICP JS [23]. Section 6 gives example sessions of the stepper to illustrate its basic features. Section 7 introduces the advanced feature of *function skipping*, demonstrates its use and discusses its implementation. Numerous stepper tools and systems for designing and implementing such tools are presented in the literature. In Section 8, we compare the most closely related systems to our approach, before concluding and discussing planned extensions in Section 9.

The stepper is implemented in an open-source programming environment custom-built for first-year students. The beginning of Section 6 explains how to use the system and how to navigate to the stepper.

2 Syntax of Source §0

To focus on the distinguishing features of the stepper, Figure 2 specifies a Source §2 sublanguage that we call *Source §0.* We limit the discussion in this section and Sections 3 and 4 to Source §0 to avoid excessive formalism in this tool description. The full language Source §2—specified in [16]—adds blocks as statements, constant declarations, conditional statements, lambda expressions, unary operators, more binary operators, strings, and pairs as primitive data structures.

A full Source §2 stepper is implemented in the Source Academy and described in [9].

In the Source languages, the bodies of functions are blocks, which may contain return statements and expression statements in any position, as illustrated by the following (somewhat contrived) example program, which we will call *f-apply*.

```
function f(x) { 17; return x + 1; 57; }
2 + 3; f(4 * 5) - 6;
```

The first two chapters of SICP JS adhere to purely functional programming (no assignment, stateful operations or loops), and rely on (mutual) recursion as the central means of control flow. In the following example program, which we will call *even-odd*, the declarations of the mutually recursive functions even and odd are followed by the application of even to the literal 5.

```
function even(n) {
    return n === 0 ? true : odd(n - 1);
}
function odd(n) {
    return n === 0 ? false : even(n - 1);
}
even(5);
```

In JavaScript (and Source), the scope of a function declaration is the surrounding block, or the whole program if there is no surrounding block as in this case. As in JavaScript, we stipulate that function declarations are moved ("hoisted") to the beginning of the block or program, as a preprocessing step.

3 Reduction

In this section, we define a reduction relation $\rightarrow^*$ that describes the evaluation of Source §0 programs. The stepper visualizes the evaluation process. The reduction of the *even-odd* program above will include the following steps; the exact shapes of *even* and *odd* during reduction are explained in Section 3.2.

$$\begin{array}{llll} & \textit{even-odd} & & \\ \rightarrow^* & \textit{even}(\texttt{5}); & \rightarrow^* & \texttt{5 === 0 ? true : } \textit{odd}(\texttt{5 - 1}); \\ \rightarrow^* & \textit{odd}(\texttt{5 - 1}); & \rightarrow^* & \texttt{4 === 0 ? false : } \textit{even}(\texttt{4 - 1}); \\ \rightarrow^* & \textit{odd}(\texttt{1 - 1}); & \rightarrow^* & \texttt{0 === 0 ? false : } \textit{even}(\texttt{0 - 1}); \\ \rightarrow^* & \texttt{false}; & & \end{array}$$

We will define the one-step reduction function $\rightarrow$ as a partial function from programs/statements/expressions to programs/statements/expressions, and $\rightarrow^*$ as its reflexive transitive closure. The one-step reduction function is the least relation that meets all rules given in this section using Gentzen-style. A *value* is a literal expression or a recursive function definition (introduced in Section 3.2), and values are denoted with the letter v in the following.

A *reduction* is a sequence of programs $p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n$, where p_n is not reducible, i.e. there is no program q such that $p_n \rightarrow q$. If p_n is of the form v;, we call v the *result* of the program evaluation. If p_n is the empty sequence of statements, we declare the result of the program evaluation to be the value **undefined**. Reduction can get "stuck"; the result can be a program that has neither of these two forms, which corresponds to a runtime error, for example when a function is used as an operand of the arithmetic operation `*`.

3.1 Operator Combinations, Conditionals, and Sequences

The rules *Op-Comb-Intro* (OCI1, OCI2), *Op-Comb-Red* (OCR), *Cond-Intro* (CI), and *Cond-Red* (CR1, CR2) implement the binary operator combinations and conditional expressions of Source §0.

$$\frac{e_1 \rightarrow e_1'}{e_1 \textit{ bin-op } e_2 \rightarrow e_1' \textit{ bin-op } e_2}\,[\text{OCI1}]$$

$$\frac{e_2 \rightarrow e_2'}{v \textit{ bin-op } e_2 \rightarrow v \textit{ bin-op } e_2'}\,[\text{OCI2}]$$

$$\frac{v \text{ is result of } v_1 \textit{ bin-op } v_2}{v_1 \textit{ bin-op } v_2 \rightarrow v}\,[\text{OCR}]$$

$$\frac{e_1 \rightarrow e_1'}{e_1 \textbf{ ? } e_2 \textbf{ : } e_3 \rightarrow e_1' \textbf{ ? } e_2 \textbf{ : } e_3}\,[\text{CI}]$$

$$\frac{}{\texttt{true ? } e_1 \texttt{ : } e_2 \rightarrow e_1}\,[\text{CR1}] \qquad \frac{}{\texttt{false ? } e_1 \texttt{ : } e_2 \rightarrow e_2}\,[\text{CR2}]$$

Our proof trees identify in square brackets each rule being applied. The base case at the top of a tree is always a reduction rule ...-Red rule (...R...).

$$\dfrac{\dfrac{\dfrac{}{\boxed{\texttt{1 + 2}} \rightarrow \texttt{3}}\,[\text{OCR}]}{\texttt{3 === } \boxed{\texttt{1 + 2}} \rightarrow \texttt{3 === 3}}\,[\text{OCI1}]}{\texttt{3 === } \boxed{\texttt{1 + 2}} \texttt{ ? 17 : 42} \rightarrow \texttt{3 === 3 ? 17 : 42}}\,[\text{CI}]$$

The program component where a reduction rule can be applied is called a *redex*. JavaScript semantics prescribes that there is at most one redex in any program with only one applicable rule, and therefore $\rightarrow$ must be a function. The redexes of our example proof trees of Section 3 are highlighted with a gray background.

Expression-Statement-Intro (ESI): An expression statement as first statement in a statement sequence can be reduced if the expression can be reduced.

$$\frac{\textit{expr} \rightarrow \textit{expr}'}{\textit{expr}\,;\ \textit{prog} \rightarrow \textit{expr}'\,;\ \textit{prog}}\,[\text{ESI}]$$

First-Statement-Red (FSR), *First-Statement-Intro* (FSI): The JavaScript specification [15] categorizes statement sequences statically as *value-producing* (in our case: sequences containing expression or return statements) and *not value-producing* (in our case: sequences of declarations). A statement v; in front of a value-producing statement sequence is discarded, and a statement v; in front of a not value-producing statement sequence is retained.

The rule FSI explains why the final value of a program is retained, the remaining not value-producing statement sequence being the empty sequence in this case. In Source §2, the FSI rule also explains why the value of the program

```
1; const x = 0;
```

is 1 and not **`undefined`**.

$$\frac{\textit{prog}\text{ is value-producing}}{v\texttt{;}\ \textit{prog} \longrightarrow \textit{prog}}\,[\text{FSR}]$$

$$\frac{\textit{prog} \longrightarrow \textit{prog}'\text{ and }\textit{prog}\text{ is not value-producing}}{v\texttt{;}\ \textit{prog} \longrightarrow v\texttt{;}\ \textit{prog}'}\,[\text{FSI}]$$

Two steps from the reduction of *f-apply* illustrate some of the rules introduced so far; the shape of *fun* is explained in Section 3.2.

$$\frac{\dfrac{}{\texttt{2 + 3} \longrightarrow \texttt{5}}\,[\text{OCR}]}{\texttt{2 + 3;}\ \textit{fun}\texttt{(4 * 5) - 6;} \longrightarrow \texttt{5;}\ \textit{fun}\texttt{(4 * 5) - 6;}}\,[\text{ESI}]$$

$$\frac{}{\texttt{5;}\ \textit{fun}\texttt{(4 * 5) - 6;} \longrightarrow \textit{fun}\texttt{(4 * 5) - 6;}}\,[\text{FSR}]$$

3.2 Function Declaration and Application

Function-Declaration-Red (FDR): Function declarations are reduced as follows.

$$\frac{}{\begin{array}{l}\texttt{function}\ f\ \texttt{(}\textit{names}\texttt{)}\ \textit{block}\ \textit{prog}\\ \qquad\qquad \longrightarrow\ \textit{prog}[f \leftarrow \mu f.\texttt{(}\textit{names}\texttt{)}\ \texttt{=>}\ \textit{block}]\end{array}}\,[\text{FDR}]$$

Here, μ-terms [3] are used to represent recursive function definitions. They are not part of Source §0, but appear during reduction.

expr ::= μ *name*.**(** *names* **)** **=>** *block* recursive funct. def.

Every occurrence of *name* in *block* represents the recursive function definition itself, and thus can be seen as a cycle in the expression. Here, $t[x \leftarrow s]$ denotes the capture-avoiding substitution of s for x in t (details in Section 3.2), according to the following scoping rules. In JavaScript [15], the scope of a function declaration is the surrounding block or the whole program if there is no surrounding block, and the scope of parameters is the function body. The scope of *name* in μ *name*.**(** *names* **)** **=>** *block* is *block*.

The first two reduction steps for the *even-odd* program illustrate how function declarations give rise to cyclic recursive function definitions.
even-odd $\longrightarrow_{\text{FDR}}$

```
function odd(n) {      (μ even.(n) => {
 return n === 0          return n === 0
  ? false                 ? true
  : (μ even.(n) => {      : (μ odd.(n) => {
      return n === 0          return n === 0
       ? true                  ? false
       : odd(n - 1);           : (μ even.(n) => {
     })                             return n === 0
     (n - 1);          →FDR          ? true
}                                    : odd(n - 1);
(μ even.(n) => {                   })
  return n === 0                   (n - 1);
   ? true                        })
   : odd(n - 1);                 (n - 1);
 })                            })
(5);                           (5);
```

Function-application-Red (FR): JavaScript's specification [15] stipulates applicative-order reduction and thus, the application of a function can be reduced, only if all arguments are values. Since all function declarations substitute the function name by a μ-term, we combine β-reduction of the lambda calculus with the μ-rule, stated in [3] in the context of the $\lambda\mu$-calculus: $\mu x.Z \longrightarrow Z[x \leftarrow \mu x.Z]$

$$\frac{\textit{expr} = \mu f.\texttt{(}\ x_1 \ldots x_n\ \texttt{)}\ \texttt{=>}\ \textit{block}}{\textit{expr}\texttt{(}v_1 \ldots v_n\texttt{)} \longrightarrow \textit{block}[x_1 \leftarrow v_1]\ldots[x_n \leftarrow v_n][f \leftarrow \textit{expr}]}$$

This rule [FR] introduces a *block expression*, the second new kind of expressions that is not part of Source §0:

expr ::= *block* block expression

The result of the second step in the reduction of *even-odd* is further reduced with FR (with no free occurrences of even to be unfolded), and then with BER3 (which we will introduce in Section 3.3):

```
{
 return 5 === 0          5 === 0
  ? true                  ? true
  : (μ odd.(n) => {       : (μ odd.(n) => {
      return n === 0          return n === 0
       ? false                 ? false
       : (μ even.(n) => {      : (μ even.(n) => {
           return n === 0          return n === 0
→FR,ESI     ? true    →BER3,ESI     ? true
            : odd(n - 1);           : odd(n - 1);
         })                      })
         (n - 1);                (n - 1);
     })                      })
    (5 - 1);                (5 - 1);
};
```

Function-application-Intro (FI1, FI2): The application introduction rules stipulate strict, left-to-right reduction of the components of function applications, following JavaScript's specification [15].

$$\frac{expr \;\rightarrow\; expr'}{expr\ \textbf{(}\ exprs\ \textbf{)} \;\rightarrow\; expr'\ \textbf{(}\ exprs\ \textbf{)}}\,[\text{FI1}]$$

$$\frac{expr \;\rightarrow\; expr'}{v\ \textbf{(}\ v_1, \ldots, v_i,\ expr, \ldots\ \textbf{)} \;\rightarrow\; v\ \textbf{(}\ v_1, \ldots, v_i,\ expr', \ldots\ \textbf{)}}\,[\text{FI2}]$$

3.3 Block Expressions

The rule FR gives rise to block expressions, which are blocks that appear as expressions as a result of the unfolding of a function application expression.

Block-Expression-Intro/Reduce (BEI, BER1, BER2): A block expression is reducible if its body is reducible. A block expression whose body is the empty statement sequence or only contains a single value statement reduces to the value **undefined**.

$$\frac{prog \rightarrow prog'}{\textbf{\{}\ prog\ \textbf{\}} \rightarrow \textbf{\{}\ prog'\ \textbf{\}}}\,[\text{BEI}] \qquad \frac{}{\textbf{\{ \}} \rightarrow \textbf{undefined}}\,[\text{BER1}]$$

$$\frac{}{\textbf{\{}\ v\ \textbf{; \}} \rightarrow \textbf{undefined}}\,[\text{BER2}]$$

BER1 and BER2 reflect that in JavaScript, a function returns the value **undefined**, if the reduction of the function body does not encounter a return statement, for example:

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{function g(x) \{ x; \} g(5);}} \\
\rightarrow^{*} & \texttt{\{ 5; \};} \\
\rightarrow_{\text{BER2}} & \texttt{undefined;}
\end{array}$$

Block-Expression-Red (BER3): A block expression whose body starts with a return statement reduces to the return expression of the return statement and ignores any subsequent statements.

$$\frac{}{\textbf{\{ return}\ expr\ \textbf{;}\ stmt \ldots\ \textbf{\}} \;\rightarrow\; expr}\,[\text{BER3}]$$

Each line in the following reduction of the example *f-apply* in Section 2 is annotated by all rules involved in the reduction, and the redexes of ...-Red rules are highlighted.

$$\begin{array}{rl}
f\text{-}apply \ \rightarrow_{\text{FDR}} & \texttt{2+3; (}\mu\texttt{f.(x) => \{...\})(4 * 5) - 6;} \\
\rightarrow_{\text{OCR,ESI}} & \texttt{5; (}\mu\texttt{f.(x) => \{...\})(4 * 5) - 6;} \\
\rightarrow_{\text{FSR,ESI}} & \texttt{(}\mu\texttt{f.(x) => \{...\})(4 * 5) - 6;} \\
\rightarrow_{\text{OCR,FI2,OCI1,ESI}} & \texttt{(}\mu\texttt{f.(x) => \{...\})(20) - 6;} \\
\rightarrow_{\text{FR,OCI1,ESI}} & \texttt{\{ 17; return 20 + 1; 57; \} - 6;} \\
\rightarrow_{\text{FSR,BEI,OCI1,ESI}} & \texttt{\{ return 20 + 1; 57; \} - 6;} \\
\rightarrow_{\text{BER3,OCI1,ESI}} & \texttt{(20 + 1) - 6;} \\
\rightarrow_{\text{OCR,ESI}} & \texttt{21 - 6;} \\
\rightarrow_{\text{OCR,ESI}} & \texttt{15;}
\end{array}$$

4 Capture-Avoiding Substitution

As in the lambda calculus, we define substitution by applying substitution rules recursively, following the structure of the program. It avoids *capturing* by alpha-renaming, when the component to be substituted into a name binder contains one of the binder's names as a free name. In Source §0, name binders are function declarations, whose parameters are binding, blocks, whose function declarations are binding, and μ-terms of the form $\mu\ name.(names)$ `=>` $block$, where *name* and all names in *names* are binding. Capture-avoiding substitution of an expression e for a name y in a μ-term is defined as follows:

$$\begin{array}{r}(\mu x.(names)\ \texttt{=>}\ block) \\ [y \leftarrow e]\end{array} \;=\; \begin{cases}
(\mu z.(names)\ \texttt{=>} \\ \quad block[x \leftarrow z])[y \leftarrow e] \\ \text{if } x \text{ occurs free in } e \\[1ex]
(\mu z.(names')\ \texttt{=>} \\ \quad block[n \leftarrow z])[y \leftarrow e] \\ \text{if } n \in names \text{ occurs free in } e \\[1ex]
\mu x.(names)\ \texttt{=>} \\ \qquad block[y \leftarrow e] \\ \text{otherwise}
\end{cases}$$

where z is *fresh*, neither being x nor occurring in *names*, *block* or e, and where *names'* is the result of replacing n by z in *names*. The replacement of x or n by z prior to substitution in the first two cases is called alpha-renaming.

The rules for avoiding capturing when applying substitution to function declarations and blocks given in [9] similarly avoid the capturing of free names in the substituted components by parameters and local names of the function declarations and blocks.

5 Implementation and Integration

The Source Academy is web-based, written in TypeScript using React. The TypeScript sources are compiled to JavaScript and bundled into a web application that gets served to the browser when the learner visits the public website of the Source Academy [14]. When the learners select the stepper, their programs are not executed using the default Source language implementation but by invoking the stepper component of a collection of open-source implementations of the Source languages that also includes transpilers to JavaScript, interpreters, compilers and virtual machines.

Upon invoking the stepper, the program is parsed into an abstract syntax tree (AST) following the ESTree spec of the JavaScript syntax [13]. Instead of representing μ-terms explicitly, we decided to implement back-references to enclosing function declarations simply by cyclic references to their nodes in the AST, which as a result is more aptly called an *abstract syntax graph* (ASG). The syntax graph of the given program is repeatedly reduced until the result is reached. The ASGs of all reduction steps are stored in an array, and are converted back into program text in the frontend of the Source Academy whenever necessary. The learner can access the steps by dragging a slider or using hotkeys. We extend the ESTree spec for our purpose by allowing cycles and coreferences, and by adding block expressions. In the interface, we display $\mu x.expr$ simply as x, so the reduction steps for the *even-odd* example appear as shown in the beginning of Section 3.

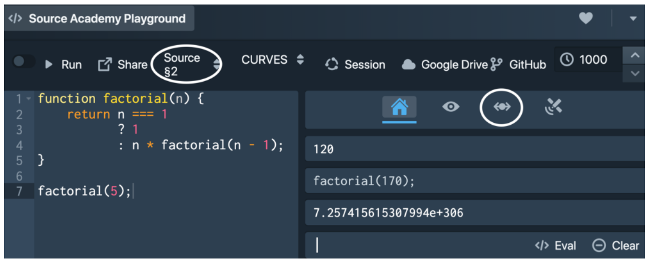
Our implementation of the reduction rules of Section 3 represents ASGs as *persistent data structures*. In the terminology of [11], the ASG is *partially persistent* during reduction: all previous versions of the ASG can be accessed but only the newest version is being modified (constructed). Careful implementation of reduction and substitution results in significant sharing of data structures across the elements of the array of ASGs. As typical for persistent data structures, a reduction step copies only the path from the root node of the program to the redex. Our implementation of substitution keeps track of the already processed nodes of the syntax graph to avoid any repeated application of recursive substitution rules to nodes of cyclic ASGs. Substituted expressions are shared across the replacements sites, introducing coreferences in the resulting ASG. Coreferences within and sharing between ASGs enable the greedy generation and retention in browser memory of all steps of the learner program.

The run time of the stepper is not perceptible for most textbook examples and learner programs. In our experience of using the stepper in an introductory computer science course in NUS, the performance of the stepper is more than sufficient. An adjustable step limit allows the learner to play with non-terminating programs. As a gauge of the stepper's performance, the evaluation of the following infinite loop takes 12 seconds with a step limit of 50,000, on a 2.2 GHz MacBook Pro using the Google Chrome browser Version 20.

```
function f() { return f(); } f();
```

6 Example Sessions

The web-based IDE of the Source Academy lets the learner enter their programs in an editor on the left. After pressing "Run", the program is evaluated and the result is displayed in a read-eval-print-loop (REPL) on the right, which allows the learner to test and explore the functions declared in the program.



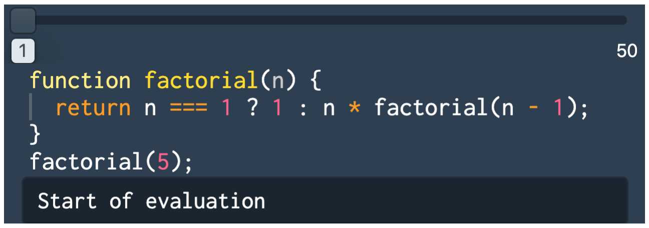
```
function factorial(n) {
    return n === 1
           ? 1
           : n * factorial(n - 1);
}

factorial(5);
```

```
120
factorial(170);
7.257415615307994e+306
```

Selection of the Source language (Source §1, §2, §3, or §4, see oval highlighted in white) restricts the evaluator to only accept programs of the selected JavaScript sublanguage. The stepper is available when one of the purely functional JavaScript sublanguages, Source §1 or Source §2, is selected. The languages Source §3 and Source §4 include assignment and destructive operations on pairs and thus, the evaluation of their programs cannot be explained with a reduction-based model.

After clicking on the stepper icon (see circle highlighted in white), the evaluation switches to reduction and the REPL is replaced by the stepper. The learner can set a limit for the number of steps (with a default value of 1000), to handle long reductions. Figure 1 above the abstract shows the stepper after the learner moves the slider to Step 22. Each reduction is shown in two steps: before (even step number) and after (odd step number) the reduction. The redex is highlighted in red in even-numbered steps and the result of reducing the redex is highlighted in green in odd-numbered steps.

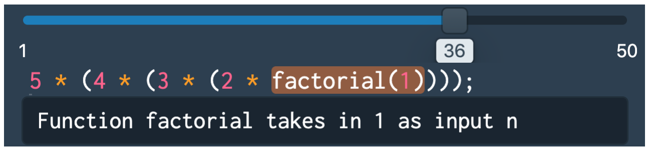
Each step displays an explanation text below the program.



```
function factorial(n) {
  return n === 1 ? 1 : n * factorial(n - 1);
}
factorial(5);
```

```
Start of evaluation
```

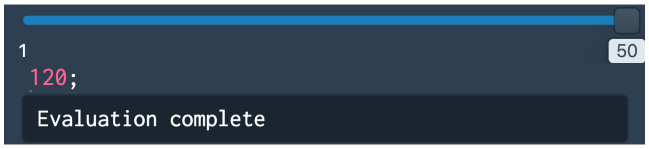
Step 22 in Figure 1 shows the program just after the third call of `factorial`. Step 36 below shows the program just before the final recursive call of `factorial`, and the deferred multiplications that have accumulated during the previous recursive calls.



```
5 * (4 * (3 * (2 * factorial(1))));
```

```
Function factorial takes in 1 as input n
```

The last step shows the result of `factorial(5)`.



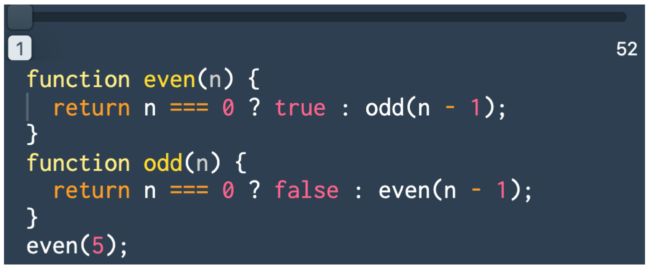
```
120;
```

```
Evaluation complete
```

To save space, we clipped off the display of the explanation in the images below.

6.1 Stepping through the Program *even-odd*

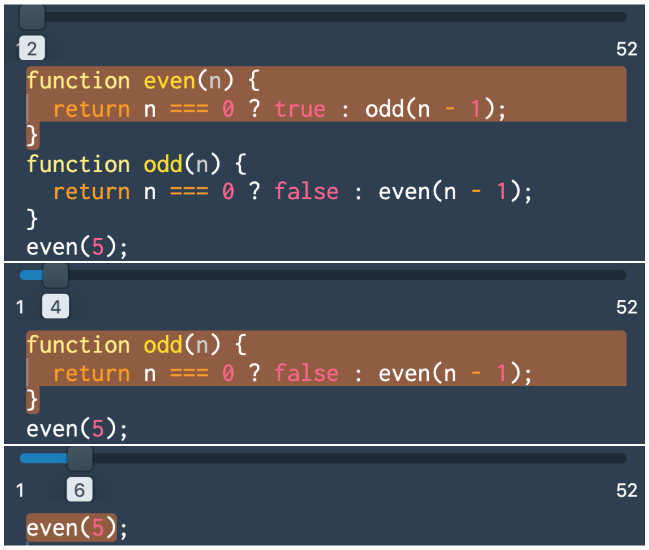
We show how the stepper visualizes the evaluation of the *even-odd* example discussed in Section 3.2.



```
function even(n) {
  return n === 0 ? true : odd(n - 1);
}
function odd(n) {
  return n === 0 ? false : even(n - 1);
}
even(5);
```

The functions even and odd are declared and substituted into the rest of the program, where even is applied to 5. The stepper highlights the next expression to be reduced.

After reducing the declaration `function even(n) {...}` with FDR in Section 3.2, the name even in `even(n - 1);` and `even(5);` refers to a μ-term μ even.*expr*, which the stepper represents by just printing even to keep the program readable.

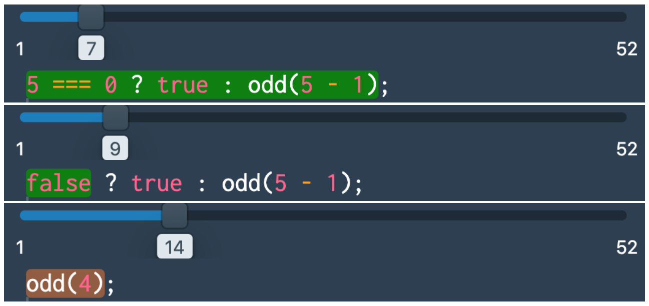


```
function even(n) {
  return n === 0 ? true : odd(n - 1);
}
function odd(n) {
  return n === 0 ? false : even(n - 1);
}
even(5);
```

```
function odd(n) {
  return n === 0 ? false : even(n - 1);
}
even(5);
```

```
even(5);
```

When a recursive function definition is applied whose body consists of a single return statement as in `even(5)`, the stepper combines FR in Section 3.2 and BER3 in Section 3.3 into a single step. The result in this case is a conditional expression in which the condition `5 === 0` is reduced to **false** with CI in Section 3.1, returning the alternative expression `odd(5 - 1)` using CR2 in Section 3.1, which is reduced to `odd(4)`.

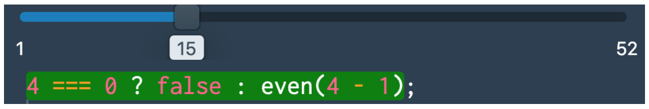


```
5 === 0 ? true : odd(5 - 1);
```

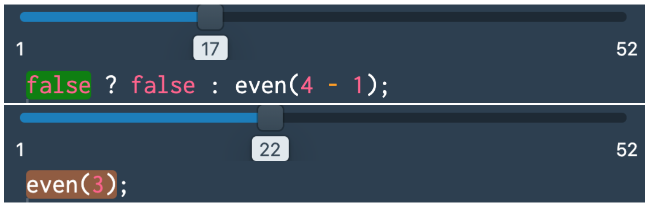
```
false ? true : odd(5 - 1);
```

```
odd(4);
```

Similarly, the application of the recursive function definition odd is replaced with the return expression of its body, in which the condition `4 === 0` is reduced to **false**, returning the alternative expression `even(4 - 1)`, which is reduced to `even(3)`.



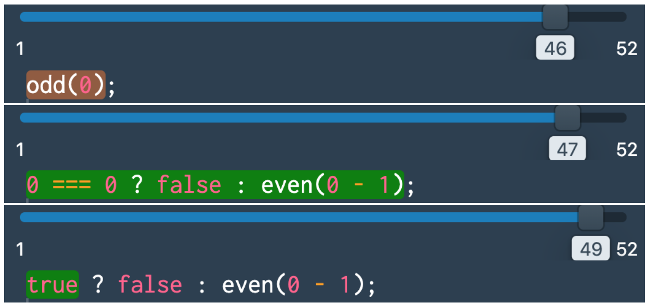
```
4 === 0 ? false : even(4 - 1);
```



```
false ? false : even(4 - 1);
```

```
even(3);
```

The functions even and odd call each other recursively until the base case `odd(0)` is reached. This application is replaced with the return expression of odd, in which the condition `0 === 0` is now reduced to **true**,...

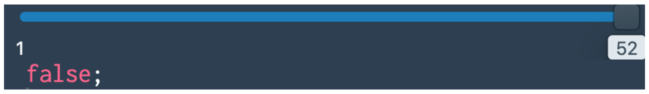


```
odd(0);
```

```
0 === 0 ? false : even(0 - 1);
```

```
true ? false : even(0 - 1);
```

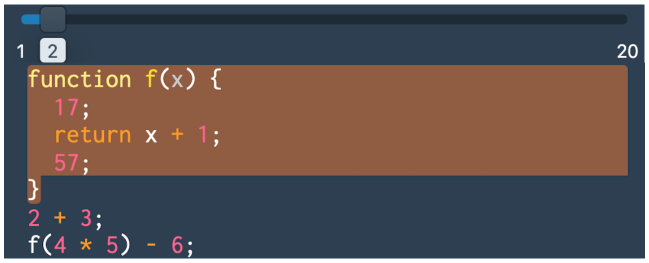
...finally returning the consequent expression **false**.



```
false;
```

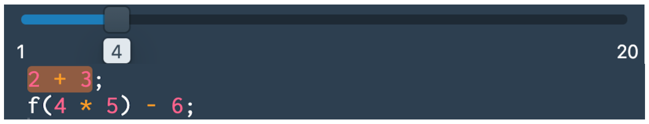
6.2 Stepping through the *f-apply* Program

We illustrate how the stepper visualizes the reduction of the *f-apply* example given at the end of Section 3.3.



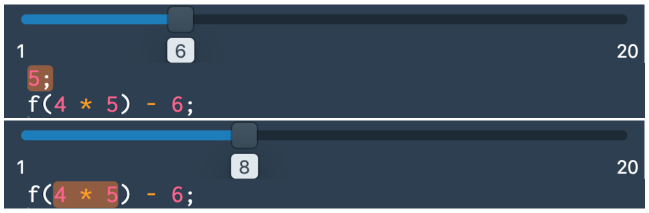
```
function f(x) {
  17;
  return x + 1;
  57;
}
2 + 3;
f(4 * 5) - 6;
```

The function f is declared and substituted into the rest of the program. After reducing the declaration `function f(x) {...}` with rule FDR in Section 3.2, the name f refers to a μ-term in `f(4 * 5) - 6;`, which the stepper displays as just f.



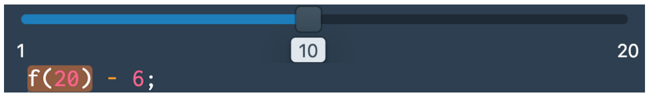
```
2 + 3;
f(4 * 5) - 6;
```

The expression statement `2 + 3;` is reduced to `5;` using ESI, but `5;` is discarded using FSR, both in Section 3.1.



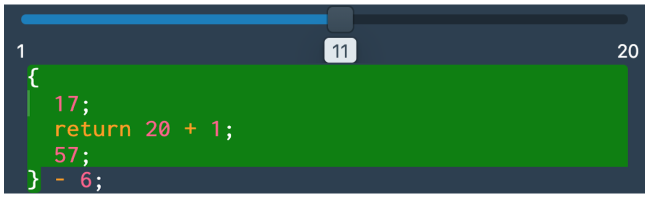
```
5;
f(4 * 5) - 6;
```

```
f(4 * 5) - 6;
```



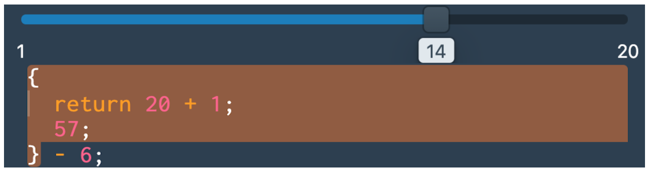
```
f(20) - 6;
```

The application of f is replaced with the body of f using FR in Section 3.2, in which all occurrences of x are substituted by 20. Note the highlighted result showing a block expression within an operator combination, which is not part of the JavaScript syntax.



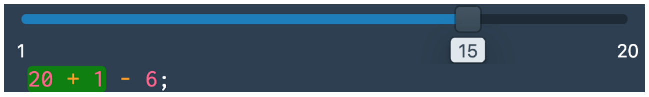
```
{
  17;
  return 20 + 1;
  57;
} - 6;
```

As with 5; in Step 6, the statement 17; is discarded due to FSR.



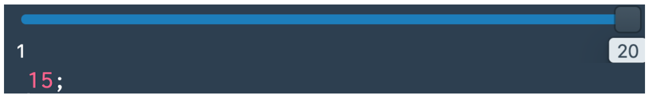
```
{
  return 20 + 1;
  57;
} - 6;
```

The block expression's first statement is now a return statement with the return expression 20 + 1. The block is reduced to the return expression due to BER3 in Section 3.3.



```
20 + 1 - 6;
```

The remaining reduction steps are completed and the program is reduced to 15;.



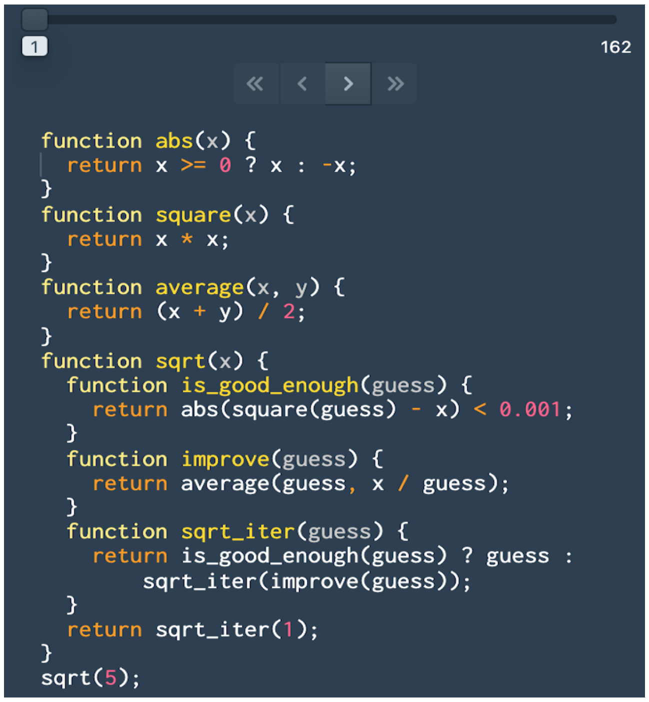
```
15;
```

7 Skipping to Function Applications

The number of steps even for relatively simple textbook programs can quickly reach several thousand. In addition to the slider and hotkeys for moving forward and backward by one step, we decided to support *function-level skipping*. When the evaluation of a program reaches a function application of a function f, the learner has in general four options:

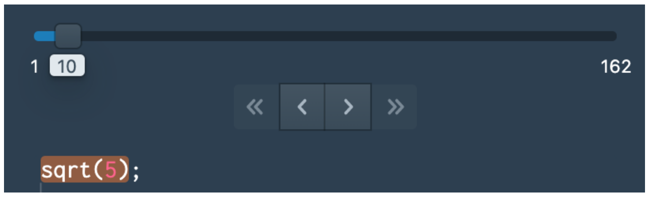
- move to the previous step (if there is one), using a < button,
- move to the next step (if there is one), using a > button,
- take a "big step" backward by skipping to the previous application of f (if there is one), using a « button, or
- take a "big step" forward by skipping to the next application of f (if there is one), using a » button.

We illustrate this feature using the example of Newton's method for computing square roots from Section 1.1.8 of SICP JS. After running the program with the stepper, it shows the entire program as "Start of evaluation".



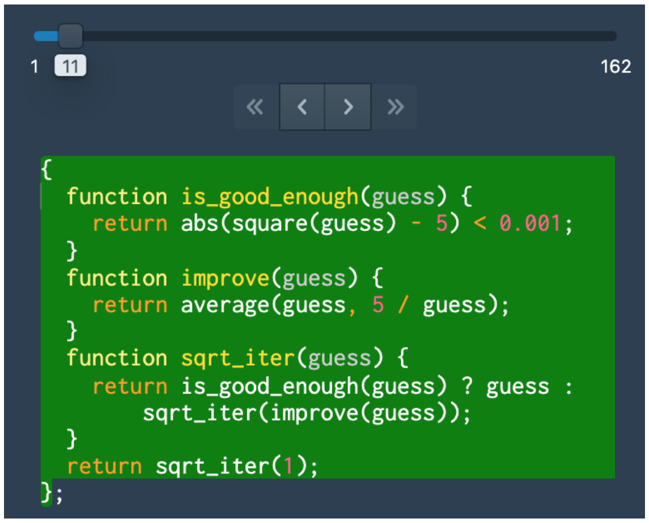
```
function abs(x) {
  return x >= 0 ? x : -x;
}
function square(x) {
  return x * x;
}
function average(x, y) {
  return (x + y) / 2;
}
function sqrt(x) {
  function is_good_enough(guess) {
    return abs(square(guess) - x) < 0.001;
  }
  function improve(guess) {
    return average(guess, x / guess);
  }
  function sqrt_iter(guess) {
    return is_good_enough(guess) ? guess :
           sqrt_iter(improve(guess));
  }
  return sqrt_iter(1);
}
sqrt(5);
```

At Step 10, the functions declarations are all substituted into the function expression of the final application sqrt(5)...



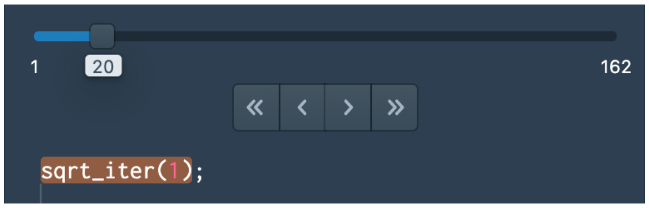
```
sqrt(5);
```

...which unfolds the body of the recursive function definition sqrt, resulting in another example of a non-JavaScript block expression.



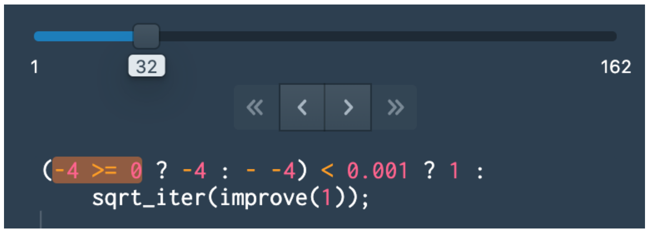
```
{
  function is_good_enough(guess) {
    return abs(square(guess) - 5) < 0.001;
  }
  function improve(guess) {
    return average(guess, 5 / guess);
  }
  function sqrt_iter(guess) {
    return is_good_enough(guess) ? guess :
           sqrt_iter(improve(guess));
  }
  return sqrt_iter(1);
};
```

Reduction continues to the application of sqrt_iter at Step 20.



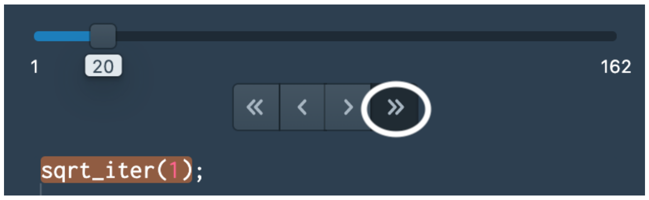
```
sqrt_iter(1);
```

Using only the arrow button > or hotkeys, the learner would need to click through a large number of steps...



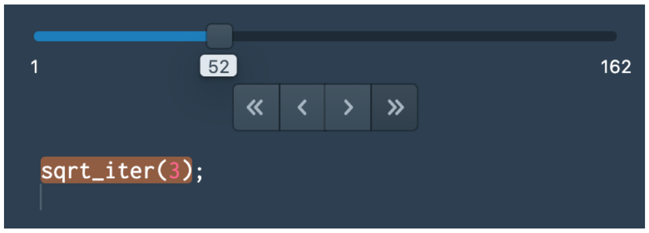
```
(-4 >= 0 ? -4 : - -4) < 0.001 ? 1 :
    sqrt_iter(improve(1));
```

...in search for the next call of `sqrt_iter`. Instead of carrying out small reduction steps, the learner can skip from the application of `sqrt_iter` in Step 20...



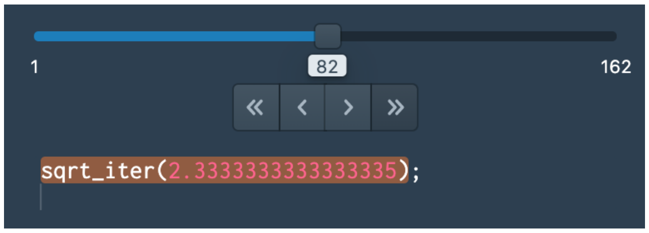
```
sqrt_iter(1);
```

...to the next application of this function, by pressing the » button, highlighted above with a white circle, skipping to Step 52.

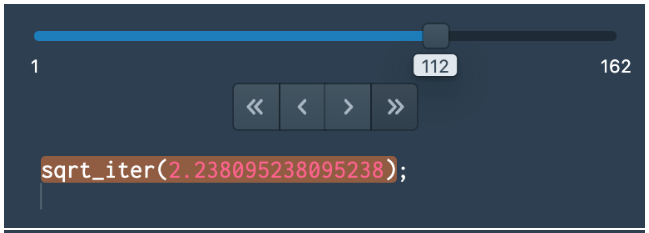


```
sqrt_iter(3);
```

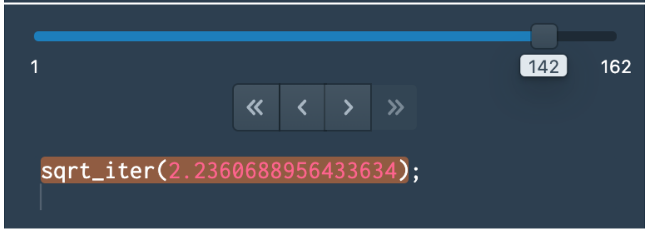
Repeated skipping allows the learner to quickly see the reduction steps that involve `sqrt_iter`.



```
sqrt_iter(2.3333333333333335);
```

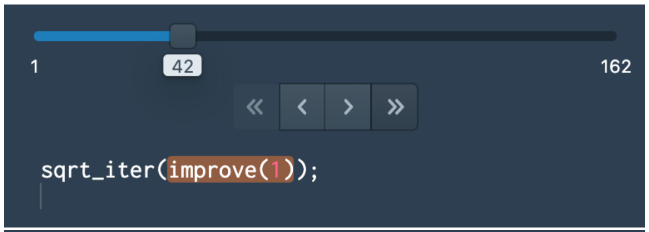


```
sqrt_iter(2.238095238095238);
```

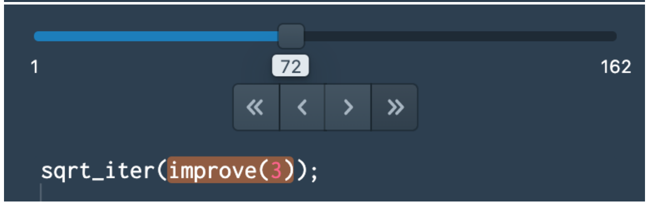


```
sqrt_iter(2.2360688956433634);
```

Similarly, the learner may decide to trace the calls of the `improve` function by repeatedly pressing the » button, starting in Step 42.



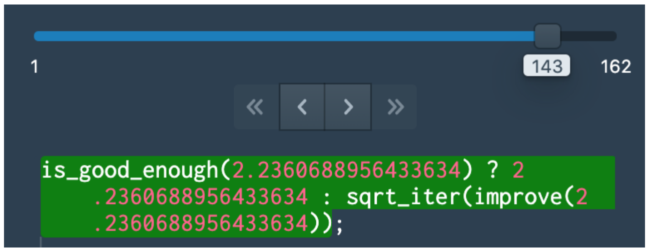
```
sqrt_iter(improve(1));
```



```
sqrt_iter(improve(3));
```

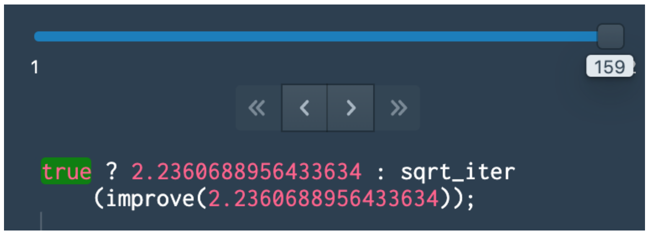
...followed by `sqrt_iter(improve(2.333));` in Step 102 and `sqrt_iter(improve(2.238));` in Step 132 (not shown).

The last call of the function `is_good_enough`...



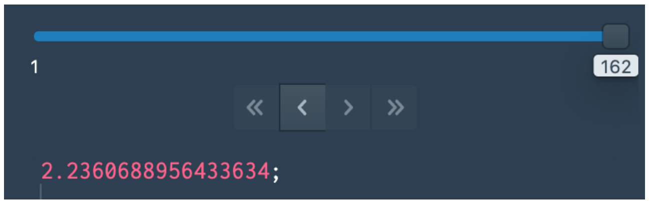
```
is_good_enough(2.2360688956433634) ? 2
    .2360688956433634 : sqrt_iter(improve(2
    .2360688956433634));
```

...returns **`true`**,...



```
true ? 2.2360688956433634 : sqrt_iter
    (improve(2.2360688956433634));
```

...and the reduction terminates with an approximation of $\sqrt{5}$ to the required accuracy as the result.



```
2.2360688956433634;
```

Our implementation of this feature makes use of the sharing of syntax graphs in the array of graphs. The buttons « and » are rendered as clickable at a function application if a previous/following application step is found, whose function expression is the same syntax graph (using pointer equality) as the function expression of the function application where the search started.

8 Discussion of Related Work

Algebraic steppers have been developed for many functional programming languages, including Scheme [10], Racket [20], Lazy Racket [7], and Haskell [8]. The Scheme, Racket, and Lazy Racket steppers generate a stack of continuations at run time by instrumenting the learner's program with "continuation marks" in a preprocessing step. In contrast to these steppers, our stepper directly reduces the given program, following the presented reduction semantics and resulting in a random-access trace data structure. In this approach, we follow Haskell's stepper, Hat, which stores the full reduction trace in a data structure called "augmented redex trail". Our stepper also shares with Hat the strategy of maximizing sharing for space efficiency, which leads in both systems to data sharing between steps, coreferences, and cycles and thus requires term graph rewriting. The direct syntactic representation of intermediate states simplifies the specification and implementation of the tool and facilitates learner interaction but foregoes opportunities for optimized compilation of the learner program.

Broadly speaking, our approach is similar to the way rewriting-based operational semantics are implemented in PLT Redex [12], a general framework for designing programming languages along with their operational semantics. As in PLT Redex, we represent the given programs syntactically during reduction, using the data structures that represent their components. To maximize data structure sharing, our data structures represent recursive functions by cycles and the instances of parameters by coreferences in their syntax graph.

As a minimal formal system for computation, the lambda calculus [4] is used in teaching the theory of computation and programming language semantics (e.g. [12]), often backed up by programming exercises or even complete experimentation environments such as LambdaLab [21] and Lambdulus [22]. These implementations are based on term (tree) rewriting and not on graph rewriting and do not handle explicit recursion, which is necessary for a direct implementation of our reduction semantics. They share with our stepper and with Hat that the intermediate results of a reduction are explicitly generated and stored. As a web-based interactive system written in TypeScript using React, Lambdulus has a similar general system architecture as the Source Academy.

Our stepper performs before/after redex highlighting as do LambdaLab and Lambdulus. A survey of functional program visualization systems in [24] lists several earlier implementations that share this feature, including the Scheme Stepper [10].

Chang introduces the JavaScript variant JAVASCRIPTY [6] in his programming languages course, the only JavaScript-related small-step reduction-based semantics that we are aware of. JAVASCRIPTY is not designed as a JavaScript sublanguage. It deviates from JavaScript in the scoping of constant declarations and supports only a restricted form of return statements. As opposed to JAVASCRIPTY [6], our approach handles JavaScript's statement-oriented syntax, allows return statements to occur anywhere in a statement sequence, and covers JavaScript's block-scoped declarations.

9 Conclusion and Future Work

This system description presents the first reduction-based semantics for a sublanguage of JavaScript and the first algebraic stepper for a language with return statements and block-scoped declarations. As partially persistent data structures, the abstract syntax graphs that represent each step enjoy significant sharing among each other, and internally through coreferences and cycles, which enables a memory-efficient implementation. Redex highlighting and function skipping contribute to the learner's experience. Since the lambda calculus is a subset of the JavaScript sublanguage covered by the stepper, it can also be used for for teaching applicative-order-reduction lambda calculus and capture-avoiding substitution.

The Source Academy includes a picture language, following Section 2.2.4 of SICP JS, which serves as a prominent example for the substitution model in our course. Future versions of the stepper should include an integration of pictures in a similar way in which the Racket IDE integrates the pictures generated by its Pict library [19]. In addition to function-level skipping, we intend to adopt the fine-grained control provided by Lambdulus [22] using break points that are specified by the learner in the program editor.

A specification of the reduction system in Coq [5] or lightweight mechanization tools (for an overview, see [17]) will allow us to mechanize proofs of correctness of the stepper or improve testing.

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] Zena M. Ariola and Jan Willem Klop. 1997. Lambda Calculus with Explicit Recursion. *Information and Computation* 139, 2 (December 1997), 154–233.

[4] Hendrik Pieter Barendregt. 2013. *The Lambda Calculus: Its Syntax and Semantics* (2nd ed.). Elsevier Science, Saint Louis, MO.

[5] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, Berlin/Heidelberg.

[6] Bor-Yuh Evan Chang. 2018. Principles and Practice in Programming Languages: A Project-Based Course, University of Colorado, Boulder. (September 2018). https://csci3155.cs.colorado.edu/csci3155-notes.pdf.

[7] Stephen Chang, John Clements, Eli Barzilay, and Matthias Felleisen. 2011. Stepping Lazy Programs. http://arxiv.org/abs/1108.4706.

[8] Olaf Chitil and Yong Luo. 2007. Structure and Properties of Traces for Functional Programs. *Electronic Notes in Theoretical Computer Science* 176, 1 (2007), 39–63. https://doi.org/10.1016/j.entcs.2006.10.032

[9] 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.

[10] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems, ESOP'01 (LNCS)*, David Sands (Ed.), Vol. 2028. Springer, Berlin, Heidelberg, New York, 320–334. https://doi.org/10.1007/3-540-45309-1_21

[11] James Driscoll, Neil Sarnak, Daniel Sleator, and Robert Tarjan. 1986. Making data structures persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, Juris Hartmanis (Ed.). STOC 1986, Berkeley, CA, 109–121. https://doi.org/10.1145/12130.12142

[12] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press, Cambridge, MA.

[13] Github estree repository. 2021. estree. https://github.com/estree/estree.

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

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

[16] Martin Henz, Ning-Yuan Lee, and Daryl Tan. 2021. *Specification of Source §2*. Technical Report. National University of Singapore. https://source-academy.github.io/source/source_2.pdf

[17] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, and Sam Tobin-Hochstadt. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Michael Hicks (Ed.). POPL 2012, Philadelphia, PA, 285–296. https://doi.org/10.1145/2103656.2103691

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

[19] Racket team. 2021. Pict: Functional Pictures. https://docs.racket-lang.org/pict.

[20] Racket team. 2021. The Stepper. https://docs.racket-lang.org/stepper.

[21] Daniel Sainati and Adrian Sampson. 2018. LambdaLab: An Interactive λ-Calculus Reducer for Learning. In *Proceedings of the 2018 ACM SIGPLAN Symposium on SPLASH-E*, Benjamin Lerner (Ed.). ACM SIGPLAN, Boston, MA, 10–19. https://doi.org/10.1145/3310089.3313180

[22] Jan Sliacky and Petr Maj. 2019. Lambdulus: Teaching Lambda Calculus Practically. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, Elisa Baniassad (Ed.). ACM SIGPLAN, Athens, Greece, 57–65. https://doi.org/10.1145/3358711.3361629

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

[24] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide. 2004. *A Survey of Program Visualizations for the Functional Paradigm*. Proceedings of the 3rd Program Visualization Workshop, Research Report CS-RR-407. Department of Computer Science, University of Warwick, 2–9.