Xtreme Shrinking of JS for Teaching

Martin Henz

joint work with Boyd Anderson, Kok-Lim Low, and Daryl Tan

National University of Singapore (NUS)
On sabbatical at Uppsala University

Goal

- Teaching CS 101 at National University of Singapore
- 750 students, 3 professors, 101 teaching assistants
- Emphasis: Mental models for computation
- Good news: SICP
- Bad news: SICP is written using Scheme
- Goal: Adapt SICP to a modern language

Why JavaScript?

- Scheme and JS have functions as first-class values
- Scheme and JS are dynamically-typed
- Scheme and JS use automatic memory management

No coincidence: Bendan Eich wanted to embed Scheme in the browser

ECMAScript 2015 was the turning point for SICP JS

- Block-scoped **`let`** and **`const`**
- "**`=>`**" notation for lambda expressions
- Proper tail calls required by standard

The story of SICP (JS)

1. Functions
2. Data
3. State
4. Interpretation

[5. Compilation]

JavaScript is too big for teaching SICP

- Source §1: JavaScript sublanguage for SICP JS Chapter 1
  - Lambda calculus plus statements, primitive values, explicit recursion
- Source §2: for SICP JS Chapter 2
  - Source §1 plus pairs
- Source §3: for SICP JS Chapter 3
  - Source §2 plus variables and assignment (our CS1 course also adds arrays and loops)
- Source §4: for SICP JS Chapter 4
  - Source §3 plus a parse function

Source §1

Source §1

program	::=	*statement* . . .	program
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

Source §1 (continued)

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

Source §1 examples

Runes: https://share.sourceacademy.org/8n9p6

Curves: https://share.sourceacademy.org/funwithcurves

Source §2

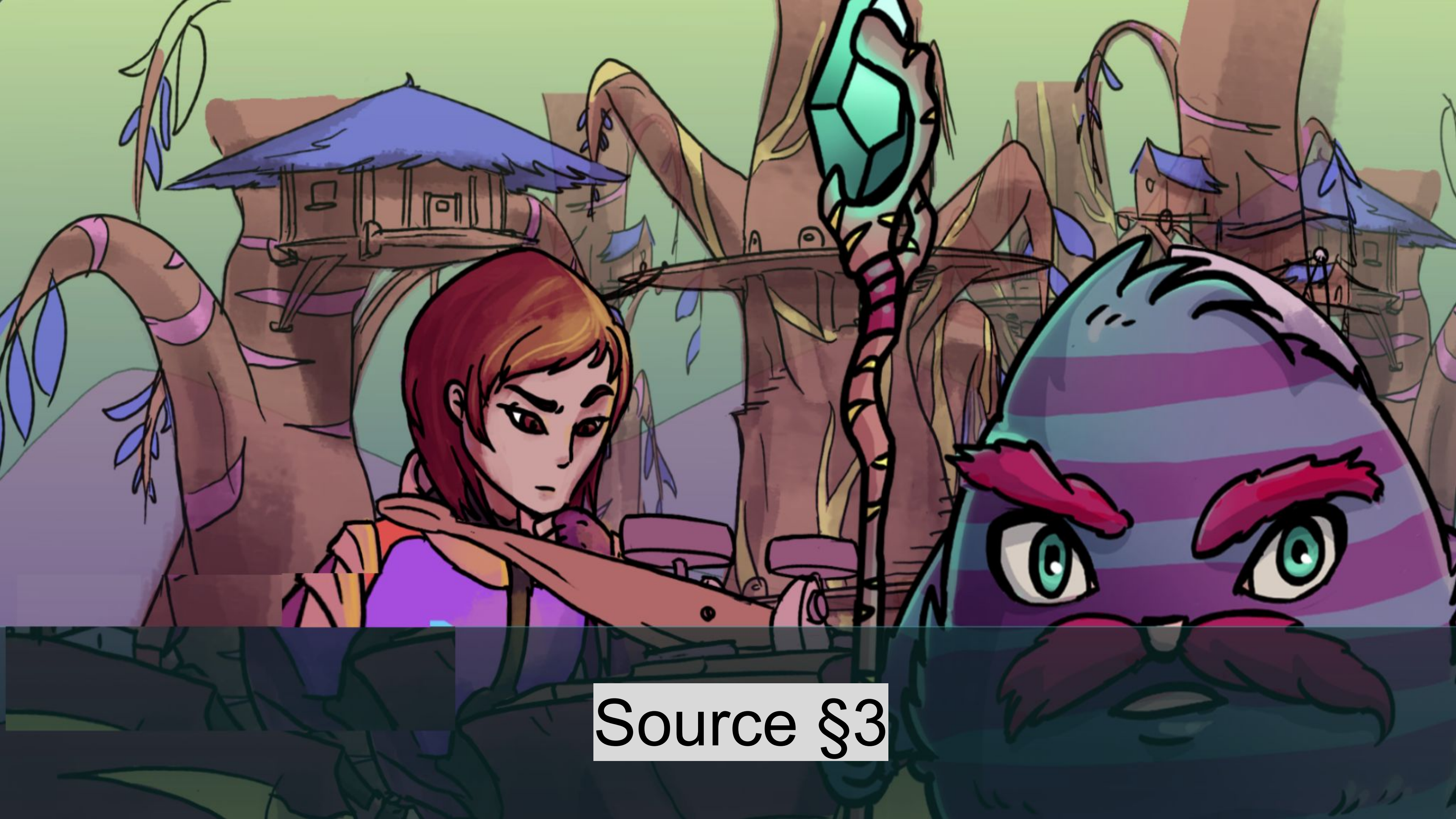
Source §2

- Add primitive expression **`null`** for empty list (Scheme’s **`nil`**)
- Add **`pair`**, **`head`**, **`tail`** (Scheme’s **`cons`**, **`car`**, **`cdr`**)
- Add library for list processing (map/reduce/filter)

Source §2 examples

Functional audio processing: https://share.sourceacademy.org/xd31b

Sound contest 2019 winner: https://share.sourceacademy.org/g9n0m


Source §3

Source §3

- Required by SICP:

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

- Required by our CS1:
  - while loops, for loops
  - Arrays:

expression	::=	. . .	
	|	*expression* **[** *expression* **]**	array access
	|	*expression* **[** *expression* **]** **=** *expression*	array assignment
	|	**[** *expressions* **]**	literal array expression

Source §3 examples

Composing video filters: https://share.sourceacademy.org/u6cg6

Motion detector: https://share.sourceacademy.org/kcubt

Wow... Can't believe that worked!

Source §4

Source §4

- Add function `parse` for meta programming

Source Academy

Free (https://github.com/source-academy), developed *for* students *by* students

- Source Academy (https://sourceacademy.org): server-less, on Github pages
- Source Academy @ X (https://about.sourceacademy.org) adds:
  - Scalable backend (written in Elixir, currently hosted on AWS)
  - Motivational game
  - Achievements
  - Assignments (uploading, submission, manual and automatic grading)
  - Contests
  - Course management support

In-browser language implementations (js-slang)

- Parser: restricts students to chosen sublanguage
- Transpiler: JavaScript-to-JavaScript translation ensures proper tail calls (PTC) even when the browser does not implement PTC, adds pedagogical error messages
- Stepper: based on small-step reduction semantics
- Compilers from Source to SMVL virtual machine language: used for robotics and SICP 3.4
- Interpreters: used for environment visualizer and SICP 4.3

Outcome

Outcome: Shrinking language

Shrinking JS for CS1 is **liberating** everyone involved:

- Students: “I can achieve what my ‘expert programmer’ peers can achieve.”
- Instructor: “I don’t need to worry about language features that I don’t cover.”
- Implementer: “I can design and implement new tools in a semester project.”

Outcome: Source Academy

91% of students in 2021 said they agree/strongly agree that the Source Academy helped them “understand the structure and interpretation of computer programs”

Some anonymous CS1101S student feedback:

- “Source Academy was a brilliant and fun platform to use. The format of paths, missions, and quests kept my interest up throughout the course.”
- “The Source Academy was nothing short of a marvel; I cannot imagine the amount of effort and resources that were needed to make it a success…”

In conclusion: JavaScript for CS1

- EcmaScript 2015 enabled seamless use of JavaScript in SICP-based courses

- JavaScript works for CS1 **if you shrink it**

Can we build an inclusive global community of learners of entry-level computer science?

Some fun with Source §1

Runes: https://share.sourceacademy.org/rightsplit

Curves: https://share.sourceacademy.org/funwithcurves

Some fun with Source §2

Functional audio processing: https://share.sourceacademy.org/echo

Sound contest 2019 winner: https://share.sourceacademy.org/0iz2g

Some fun with Source §3

Composing video filters: https://share.sourceacademy.org/funwithfilters

Motion detector: https://share.sourceacademy.org/motiondetector

The Solution (in Scheme and C)

```
(define (range bst low high)
List *range(BST *bst, int low, int high) {
  (cond ((< (datum bst) low)
  if (bst->datum < low)
         (range (right-branch bst) low high))
          return range(bst->right, low, high);
        ((> (datum bst) high)
         else if (bst->datum > high)
         (range (left-branch bst) low high))
          return range(bst->left, low, high);
        (else
         else return
         (append (range (left-branch bst) low high)
          append(range(bst->left, low, high),
                 (cons (datum bst)
                  cons(bst->datum,
                       (range (right-branch bst) low high))))))
                        range(bst->right, low, high))); }
```

From:
Brian Harvey's
"Last Lecture"
at Berkeley,
May 3 2013

λSnap!

Parser

The Source Academy uses Acorn[1], an open-source JavaScript parser, to build the Abstract Syntax Tree (AST).

We also check for any disallowed JavaScript syntax and return an error if any is found. What we get at the end is a valid Source AST.

[1]https://github.com/acornjs/acorn

Is SICP JS more complex than the original? If so: why?

Apart from the superficial syntax issues, SICP JS differs from SICP in two major ways:

(1) It adds return statements to the language: you can return from a function anywhere in the body
(2) It adds the notion of parsing: the text of a program can be transformed into a data structure

But the question is: What are the concepts that need to be covered today, when the ambition is “Structure and Interpretation of Computer Programs”?

- Return statements?
- Language processing of non-Lisp-like languages?

If the answer in these two cases is “Yes” then adding Return statements and Parsing is not a bug but a feature:

A reader who is interested in the “structure and interpretation of computer programs” should learn about return statements and what they mean, because they occur in most languages that are in popular use today.

Similarly, a reader should be exposed to parsing because it is the key to implementing any language that is not Lisp-like.

Learning experiences

Learning experiences

Background

- 1970s-90s: Hal Abelson and Gerald Jay Sussman spearhead education with Structure and Interpretation of Computer Programs
- 1997: NUS adopts SICP in a CS1 course called CS1101S
- 2008: JavaScript adaptation of SICP starts
- 2012: CS1101S converts from Scheme to JavaScript
- 2015: EcmaScript 2015 enables “serious” work on SICP JS
- **2018: CS1101S becomes compulsory for all CS first-year students**

The challenge: scaling from 120 student in 2017 to 667 students in 2021

Why use JavaScript rather than Python?

- Proper tail calls (PTC) is in JavaScript standard (ES2021).
- Python does not specify PTC.

- Functional programming is at least as elegant in JavaScript as in Scheme.
- Python imposes syntactic restrictions on lambda expressions.

- JavaScript clearly distinguishes assignment from declaration (since ES2015).
- Python does not syntactically distinguish between assignment and declaration.

Plus: All the fun in the World Wide Web!

Stepper

Processes for factorial: https://share.sourceacademy.org/factorialinstepper

Data Viz

Data visualization: SICP JS 2.2.2

Debugging append: https://share.sourceacademy.org/66ymt

Environment Visualizer

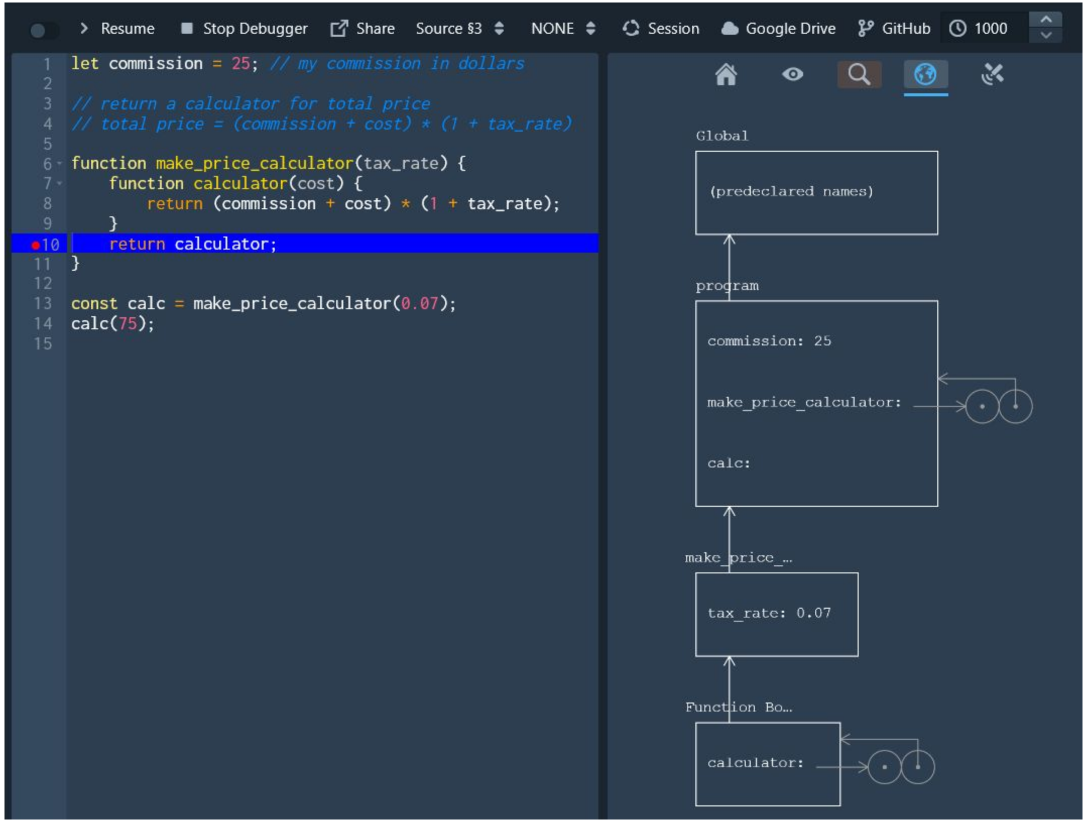
Debugging a bank account: https://share.sourceacademy.org/bankaccount

Debugging cps: https://share.sourceacademy.org/appendcps

Learning Tools: Environment Visualiser

Allows students to inspect a Source program's current execution state by setting breakpoints before the relevant program lines.

It uses a CPS-style interpreter (rather than Source transpiler)



```
let commission = 25; // my commission in dollars

// return a calculator for total price
// total price = (commission + cost) * (1 + tax_rate)

function make_price_calculator(tax_rate) {
    function calculator(cost) {
        return (commission + cost) * (1 + tax_rate);
    }
    return calculator;
}

const calc = make_price_calculator(0.07);
calc(75);
```

Why did instructors stop using Scheme for CS1?

- Programming has become a practically useful skill for students: internships, summer jobs, startups,...
- Student motivation increases when they *perceive* the language as “useful” to them
- Syntax not very important…except:
  Scheme syntax is ***so*** different from the rest