UNIVERSITY OF CALIFORNIA

Los Angeles

Seymour: A Live Programming Environment for the Classroom

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Saketh Ram Kasibatla

2018

ABSTRACT OF THE THESIS

Seymour: A Live Programming Environment for the Classroom

by

Saketh Ram Kasibatla
Master of Science in Computer Science
University of California, Los Angeles, 2018
Professor Todd D. Millstein, Chair

A programming environment that visualizes a program's execution can help users better understand their programs. However, building such an environment for programmers at large is difficult, as it is not clear how to effectively visualize large, complex programs. We would like to make progress on this problem by first focusing on a subproblem: creating a better programming experience for learning and teaching programming. To explore this approach, we built Seymour, a prototype live programming environment that visualizes all events in a program's execution. Seymour features two complementary visualizations—one that shows a detailed view of a method call, and another that depicts the whole execution at a high level. These two visualizations come together to create a compelling user experience that we plan to test and improve with student feedback.

The thesis of Saketh Ram Kasibatla is approved.

Miryung Kim

Jens Palsberg

Alessandro Warth

Todd D. Millstein, Committee Chair

University of California, Los Angeles

2018

To Amma, Nanna, Gautham, and the “village” of close ones that have shaped my life.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to Alex Warth and Todd Millstein for their collaboration and guidance. Their patience and advice was integral to making progress on this project, and I have learned and grown a great deal by working with them for these past two years.

Alex has been an incredible mentor, teacher and collaborator. Observing his approach taught me a great deal about how research is conducted. His design for the VM described in chapter 4, and the various language runtimes we experimented with while developing Seymour also taught me a great deal about effective software engineering. I will always be grateful for the long rope he gave me to explore different side alleys and tangents around our work and will look back fondly on our discussions at the whiteboard in the HARC office.

I also owe Todd my sincere thanks. He supported and guided me, giving me the freedom to explore projects I was interested in and giving helpful feedback throughout. He taught me how to communicate my research more effectively, giving feedback on the paper, talk, and thesis that resulted from this work. I am grateful for his always open door and insistence that I center the discussion of my research around the problems it solves.

I would also like to thank Jens Palsberg and Miryung Kim for discussing and reviewing my thesis. I (quite literally) could not have finished this degree without you.

Thanks to the members of the Programming Languages and Software Engineering group at UCLA for helping me better communicate the ideas in my thesis, and for their support throughout the writing process.

Finally, thank you to Bret Victor, Toby Schachman, Daniel Windham, Chaim Gingold, Jonathan Edwards, Glen Chiacchieri, Patrick Dubroy, Aran Lunzer, Yoshiki Ohshima, Marko Röder, the rest of the gang at HARC for many invaluable conversations that helped me formulate the ideas expressed in the thesis.

Chapters 1, 2, 3, 5 and 6 are based on “Seymour: Live Programming for the Classroom” [15], which appeared in the 2017 Workshop on Live Programming Systems. This work was funded by the Human Advancement Research Community (HARC), a part of Y Combinator Research.

CHAPTER 1

Introduction

Simulating what a program will do is essential for program understanding, and programmers are bad at it. Programming's steep learning curve is due in large part to the difficulty of understanding what a program will do when run [6, 23]. Even professional software developers, experts at "playing computer", spend much of their time fixing bugs created due to misunderstandings about what their code does [19].

Both professional developers and learners currently use print statements and debuggers to address misunderstandings. These tools help them collect information about their programs and correct misconceptions. However, these tools are fundamentally flawed. They require the user to actively select which parts of the program they need information about in order to make progress. This can prove difficult for students without a good mental model of program execution, and for professionals who are fixing a bug due to an incorrect hypothesis.

Live programming environments can help programmers see patterns in their programs' behavior by providing always-on visualizations [14, 18]. These show the user patterns in a program's execution as it is written, allowing runtime data to inform their mental models in a tighter feedback loop. However, effectively presenting the details of a program's execution is difficult, as even the smallest programs produce a great deal of information when run.

While the live programming community has provided many inspiring visions of programming, we have yet to see these ideas significantly impact how we program today. Due to the sheer volume of data that large programs produce, it may be too difficult to create an experience that fulfills the needs of programmers at large in one shot. Perhaps we can solve a smaller version of the problem and adapt some of the techniques we develop to a broader solution. We have created an environment that aims to provide a better programming experience for an undergraduate introduction to programming course.

There are several advantages that this use case affords us. First, programs that students write are far smaller and less complex than those written for commercial purposes. They are also guided by a class curriculum, including homework assignments and projects. This allows us to create new experiences without having to design and optimize for very large programs. Furthermore, because the teacher controls class assignments, we can tailor the system to each assignment, instead of attempting to create a fully general experience. Finally, even if our programming experience fails to scale to be useful in the 'real world,' it has the potential to help students overcome programming's steep learning curve.

This thesis presents Seymour, an early prototype that follows this strategy. The goal of Seymour is to help beginners develop an intuition for program semantics and encourage experimentation. We do this by rendering complete information about a program's execution into a visualization that tells a cohesive story. Students and teachers spend significant time simulating the behavior of a program (mentally, or on paper), often erroneously. An automated method to create visual explanations of these programs can significantly ease the mental burden of teachers and students alike when introduced in the right context, e.g., by having the professor and students draw the visualization before transitioning to a computer-generated version [22].

This thesis contributes an overview+detail [5] visualization of program execution with two parts:

1. a *micro visualization* that shows line-by-line details about the program's execution, including readable stories for arbitrary control structures and summaries of expressions' side effects, and

2. a *macro visualization* that serves as an overview of the program's execution, and can be used as a tool to focus on different points in the execution.

Together, these components serve as a useful interface to organize and navigate through the execution trace of a program.

The rest of this thesis is structured as follows:

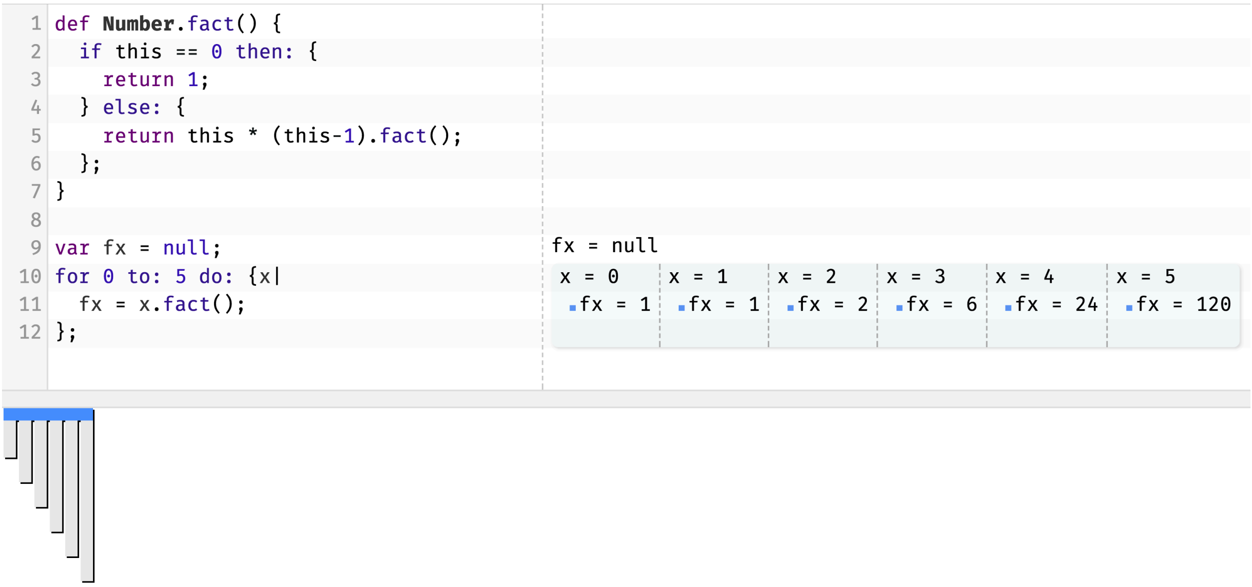
- Chapter 2 discusses the design of the macro and micro visualizations,

- Chapter 3 walks the reader through how a user might work with Seymour to write a program,
- Chapter 4 discusses the implementation of the environment and visualizations,
- Chapter 5 discusses related work,
- Chapter 6 shares future directions that we are interested in exploring, and
- Chapter 7 concludes.

CHAPTER 2

The Programming Environment

Seymour's programming environment consists of three different components, as shown in the screenshot below:



```
def Number.fact() {
  if this == 0 then: {
    return 1;
  } else: {
    return this * (this-1).fact();
  };
}

var fx = null;
for 0 to: 5 do: {x|
  fx = x.fact();
};
```

```
fx = null
 x = 0  | x = 1  | x = 2  | x = 3  | x = 4   | x = 5
 fx = 1 | fx = 1 | fx = 2 | fx = 6 | fx = 24 | fx = 120
```

Figure 2.1: The Seymour Programming Environment [1]

These are:

1. the code editor (top left);

2. the *micro visualization* (top right), which provides details about the activation of a method or function; and

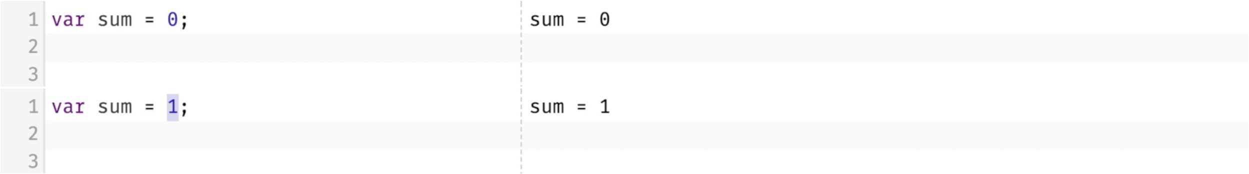
[1] The prototype shown in this paper relies on our own language an unholy combination of JavaScript and Smalltalk.

3. the *macro visualization* (bottom), which provides an overview of the entire program's execution.

These components help the programmer see and understand the dynamic behavior of their code even while they are writing it.

2.1 The Micro Visualization

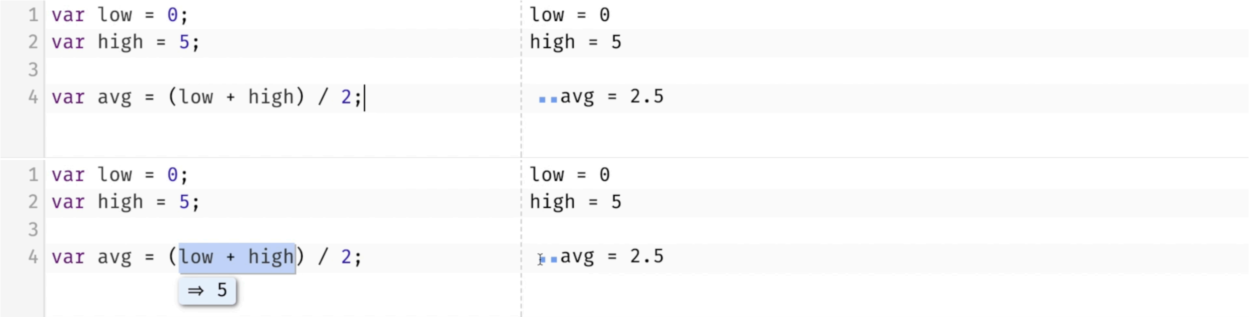
The micro visualization displays details about the execution of the code. These details are shown next to the lines of code that produced them, and they are updated as the programmer makes changes to the code.



```
var sum = 0;                              sum = 0


var sum = 1;                              sum = 1


```

Figure 2.2: Effects are shown next to lines

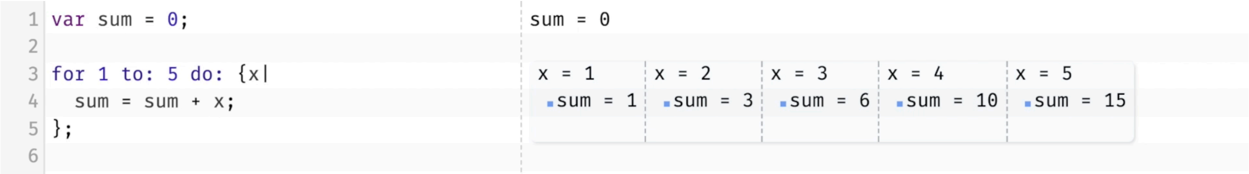
Each subexpression in the program is rendered as a blue dot. To see the value of a subexpression, the programmer simply hovers over its corresponding blue dot.



```
var low = 0;                              low = 0
var high = 5;                             high = 5

var avg = (low + high) / 2;                ▪▪avg = 2.5


var low = 0;                              low = 0
var high = 5;                             high = 5

var avg = (low + high) / 2;                ▪▪avg = 2.5
           ⇒ 5
```

Figure 2.3: Hovering over subexpressions (blue dots) shows their value

The micro visualization displays loops using a columnar format, where each column represents an iteration of the loop. The programmer can read through a row to see all of the effects from a line of code over time, or read through a column to see what happened in a particular iteration.

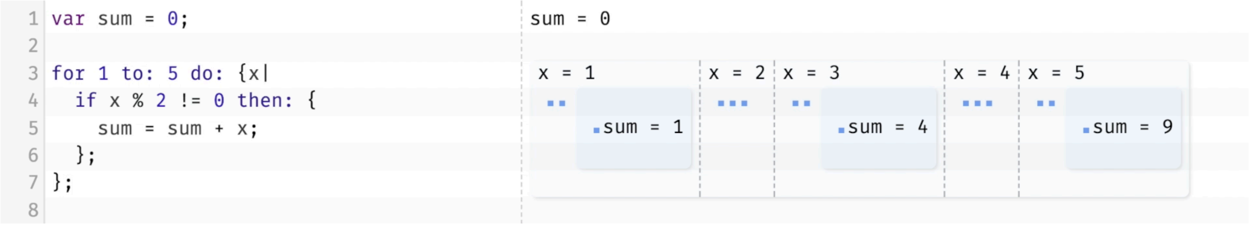


```
var sum = 0;

for 1 to: 5 do: {x|
  sum = sum + x;
};

```

```
sum = 0

x = 1     x = 2     x = 3     x = 4      x = 5
sum = 1   sum = 3   sum = 6   sum = 10   sum = 15
```

Figure 2.4: A loop, visualized using multiple columns

Because Seymour only shows effects next to executed lines of code, it is easy to 'follow the flow' [25] of the program over time. For example, in figure 2.5, the events for the 'then' branch are only shown when '`x % 2 != 0`'.



```
var sum = 0;

for 1 to: 5 do: {x|
  if x % 2 != 0 then: {
    sum = sum + x;
  };
};

```

```
sum = 0

x = 1       x = 2   x = 3       x = 4   x = 5
..          ...     ..          ...     ..
   sum = 1             sum = 4             sum = 9
```

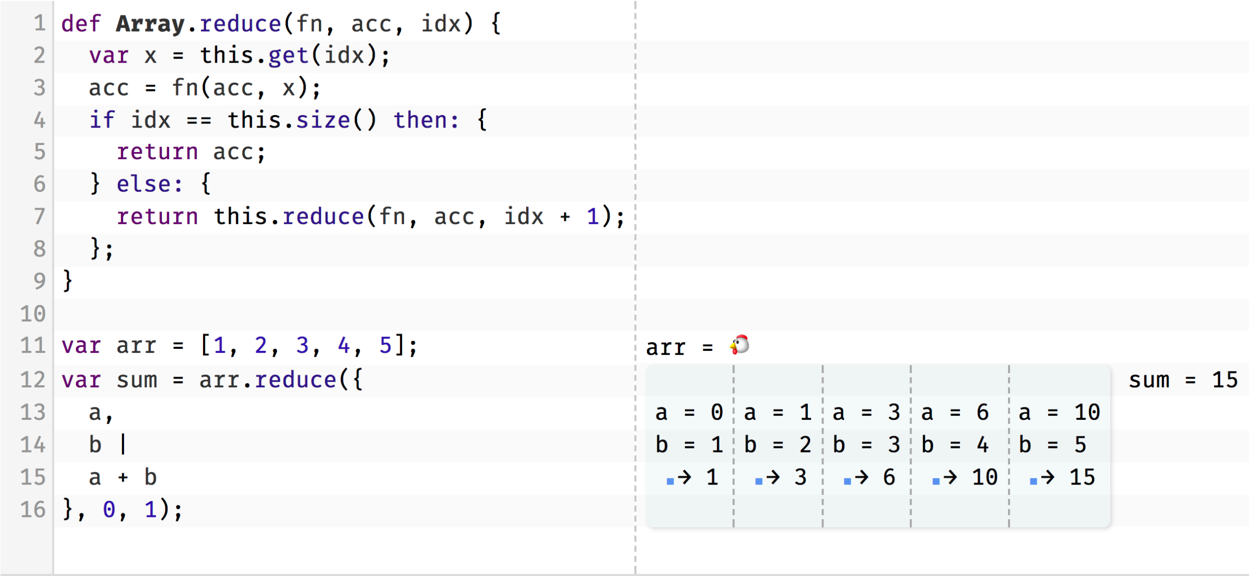
Figure 2.5: A loop with logic. Iterations where the then block is run show an assignment

In addition to built-in loop constructs, programmers often rely on user-defined control structures, e.g., `map`, `filter`, and `reduce`. While these are not loops in the strict sense of the word, they are 'loop-like' so the programmer would benefit from visualizing them as such. Seymour automatically detects loopy behavior and provides a columnar visualization for it, as shown in figure 2.5. In fact, all built-in control structures in Seymour's language are simply message sends/method calls (as in Smalltalk [9]) and are rendered in a columnar manner using the same loop mechanism—i.e., there is no special handling for built-in control structures.

Note that the call to `reduce()` is rendered as a loop even though its implementation is recursive. Seymour's micro visualization and a mechanism for detecting loopiness are described in chapter 4.

While creating the micro visualization, we sought to mitigate users' mental simulation of the program. Seymour shows every state change, without requiring interaction, so that the user does not have to simulate the program and interactively view parts of the state.

A method call, in addition to producing a result, can modify program state. If its side effects are not shown, the user must fall back to mentally simulating the program. Consider the program in

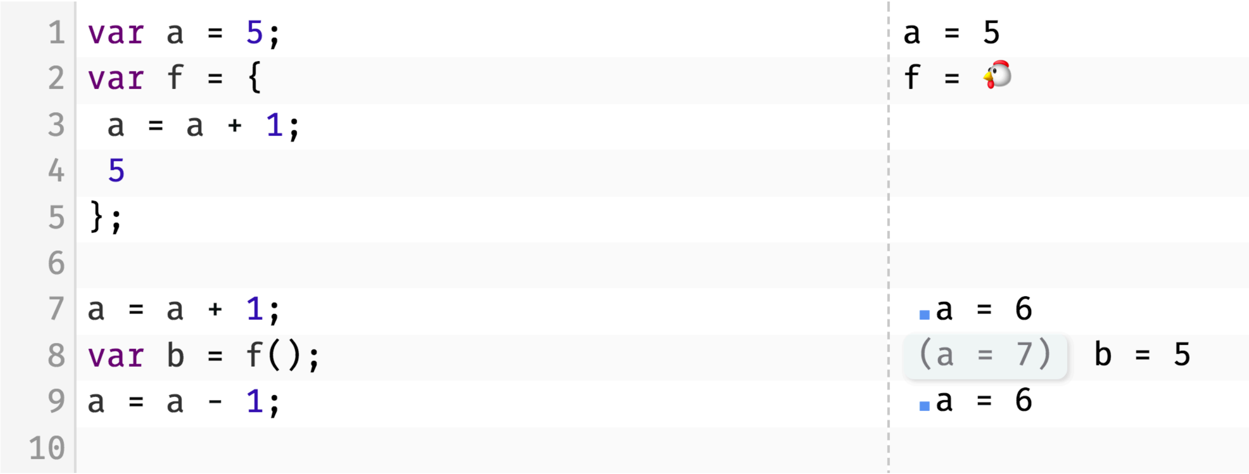


```
def Array.reduce(fn, acc, idx) {
  var x = this.get(idx);
  acc = fn(acc, x);
  if idx == this.size() then: {
    return acc;
  } else: {
    return this.reduce(fn, acc, idx + 1);
  };
}

var arr = [1, 2, 3, 4, 5];
var sum = arr.reduce({
  a,
  b |
  a + b
}, 0, 1);
```

```
arr = 🐔
                                          sum = 15
a = 0 ¦ a = 1 ¦ a = 3 ¦ a = 6 ¦ a = 10
b = 1 ¦ b = 2 ¦ b = 3 ¦ b = 4 ¦ b = 5
 ▪→ 1 ¦  ▪→ 3 ¦  ▪→ 6 ¦  ▪→ 10¦  ▪→ 15
```

Figure 2.6: Reduce implemented in the Seymour language. Our algorithms can recognize user-defined control structures [2]

figure 2.7:



```
var a = 5;
var f = {
 a = a + 1;
 5
};

a = a + 1;
var b = f();
a = a - 1;
```

```
a = 5
f = 🐔




▪a = 6
(a = 7)  b = 5
▪a = 6
```

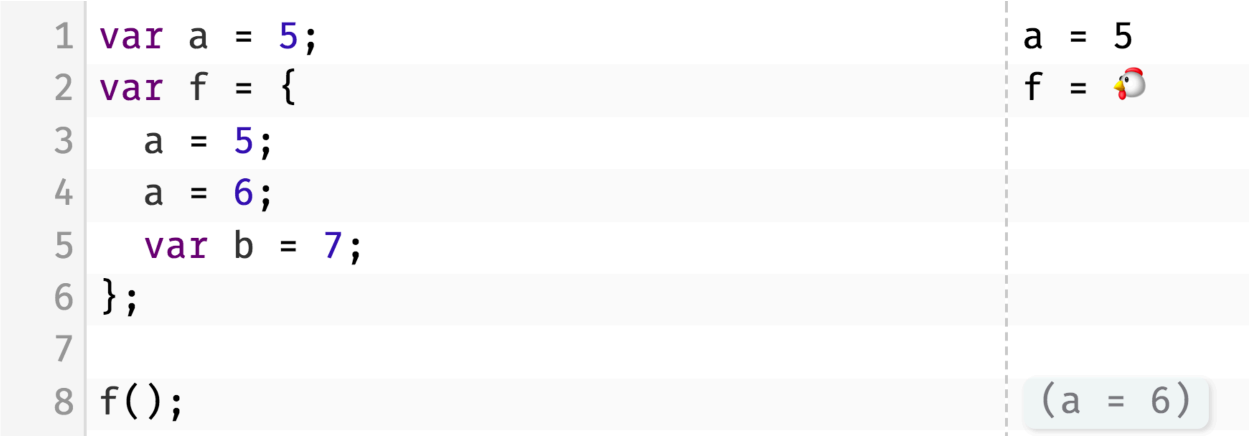
Figure 2.7: The call to f() has its side effects summarized

Because the call to f() is preceded by a statement that increments a, and followed by a statement that decrements it, the user might expect a to be 5 at the end of the program. If Seymour did not show that f() set a to 7, the user would need to read f()s source to figure out why a is not 5.

[2] We use emoji to identify different objects, instead of memory addresses or object IDs.

Because `f()`s side effects are shown, the user can confirm at a glance that `a = 6` because `f()` set `a` to `7` before it was decremented.

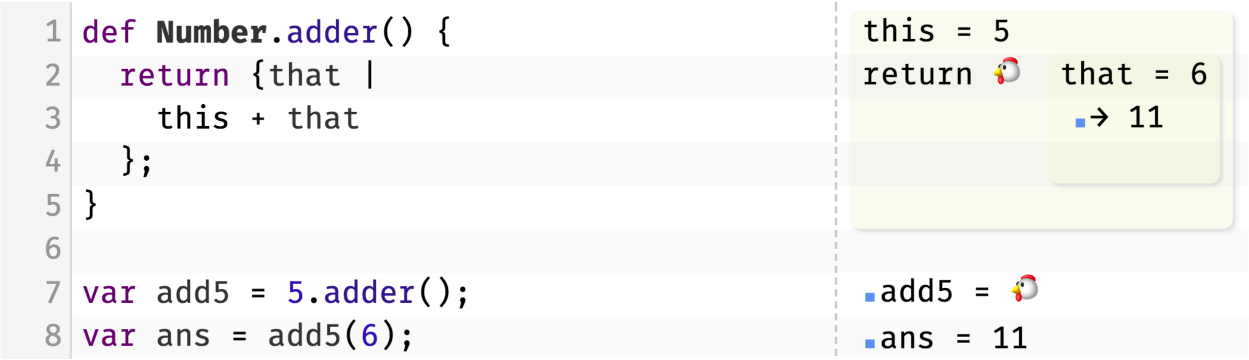
For each call, all side effects that are relevant to the execution are summarized. Specifically, the micro visualization shows the last value of variables that are written multiple times, and hides the values of local variables that do not affect computation past the duration of the call. In figure 2.8, we only show `a`'s final value, and `b = 7` does not appear in `f()`'s summary.



```
var a = 5;                          a = 5
var f = {                           f = 🐔
  a = 5;
  a = 6;
  var b = 7;
};

f();                                (a = 6)
```

Figure 2.8: Changes to `b` are not shown in `f()`'s summary

A calls summary elides most of its implementation details in order to give the user 'just enough' information to continue reading the story of the execution. If the user is interested in these details, Seymour can show the story of any call. In fact, the whole program is treated as a call and its implementation details are displayed like those of any other call, such as the call to `add5()` in figure 2.9.



```
def Number.adder() {                this = 5
  return {that |                    return 🐔    that = 6
    this + that                                   → 11
  };
}

var add5 = 5.adder();               add5 = 🐔
var ans = add5(6);                  ans = 11
```

Figure 2.9: The details of `add5()` are shown in yellow

When visualizing a call, we also show all lexical scopes that the callee (i.e., a method or a block) has access to. All the information needed to calculate the calls result is on screen, and the method only has access to the values it can "see". In figure 2.9, which visualizes the call to `add5()` on line 8, Seymour shows the value of `this` that `add5()` has access to, explaining why it returns `11`.

2.2 The Macro Visualization

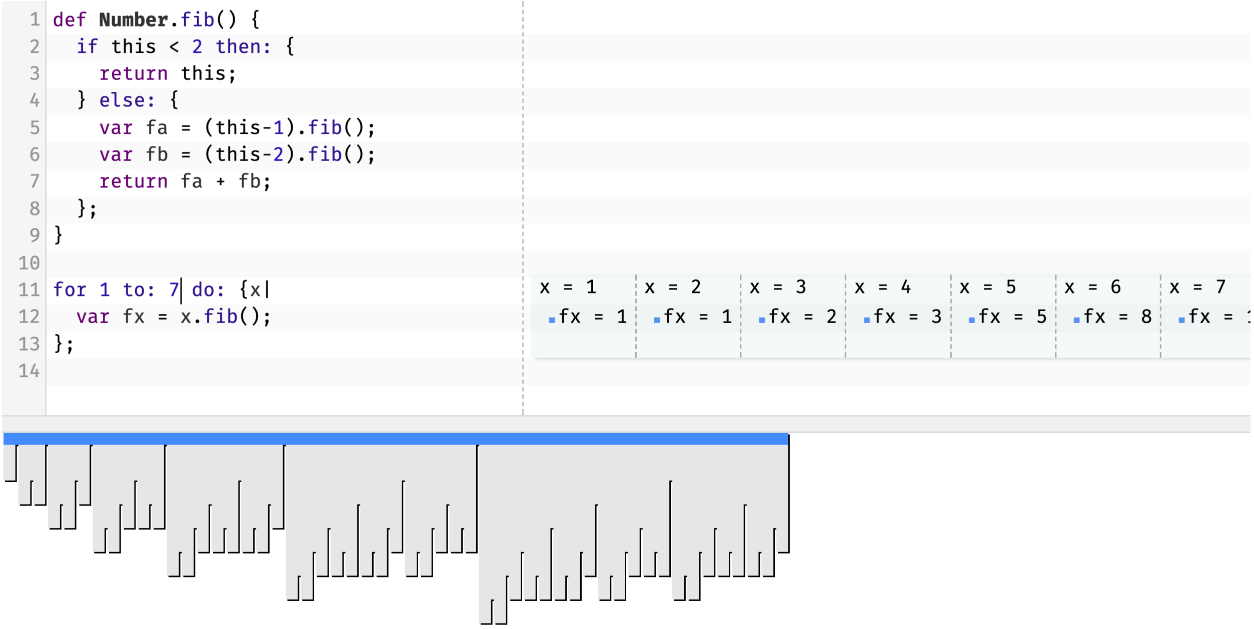
Seymour's micro visualization can show any single call in the program. However, views of disconnected calls are not enough to completely understand the program. Programmers need a low-level understanding of individual calls and a high-level understanding of how they are related.

The macro visualization helps the user build this high-level understanding by

1. showing overall patterns in the program;
2. serving as a stable, global context to which the micro visualization can be related; and
3. providing a user interface for *focusing* the micro visualization on different calls.

The visualization is a variation on the icicle plot [17]—a method for visualizing hierarchical data that makes it easy to see the 'shape' of a computation. Each method call is drawn as a rectangular node on screen. Time progresses from left to right with clusters of calls acting as landmarks on the program's timeline. All calls that a method makes are drawn below its node in the visualization. Calls that do not make any further calls (leaf calls) have fixed width, and all other calls are wide enough to contain all of their children; calls with more subcalls are wider than those with fewer subcalls. Google Chrome's DevTools [28] uses a similar visualization to provide an overview of calls that occur in a profile, but instead of giving leaf calls a fixed width, all calls are scaled based on run time.

The macro visualization does not change as the user selects different calls, instead always showing every call in the program. However, without further help from the system, it is difficult to connect the time-oriented macro visualization with the code-oriented micro visualization, so Seymour helps the user make this connection with parallel highlighting. When the user hovers over a call in



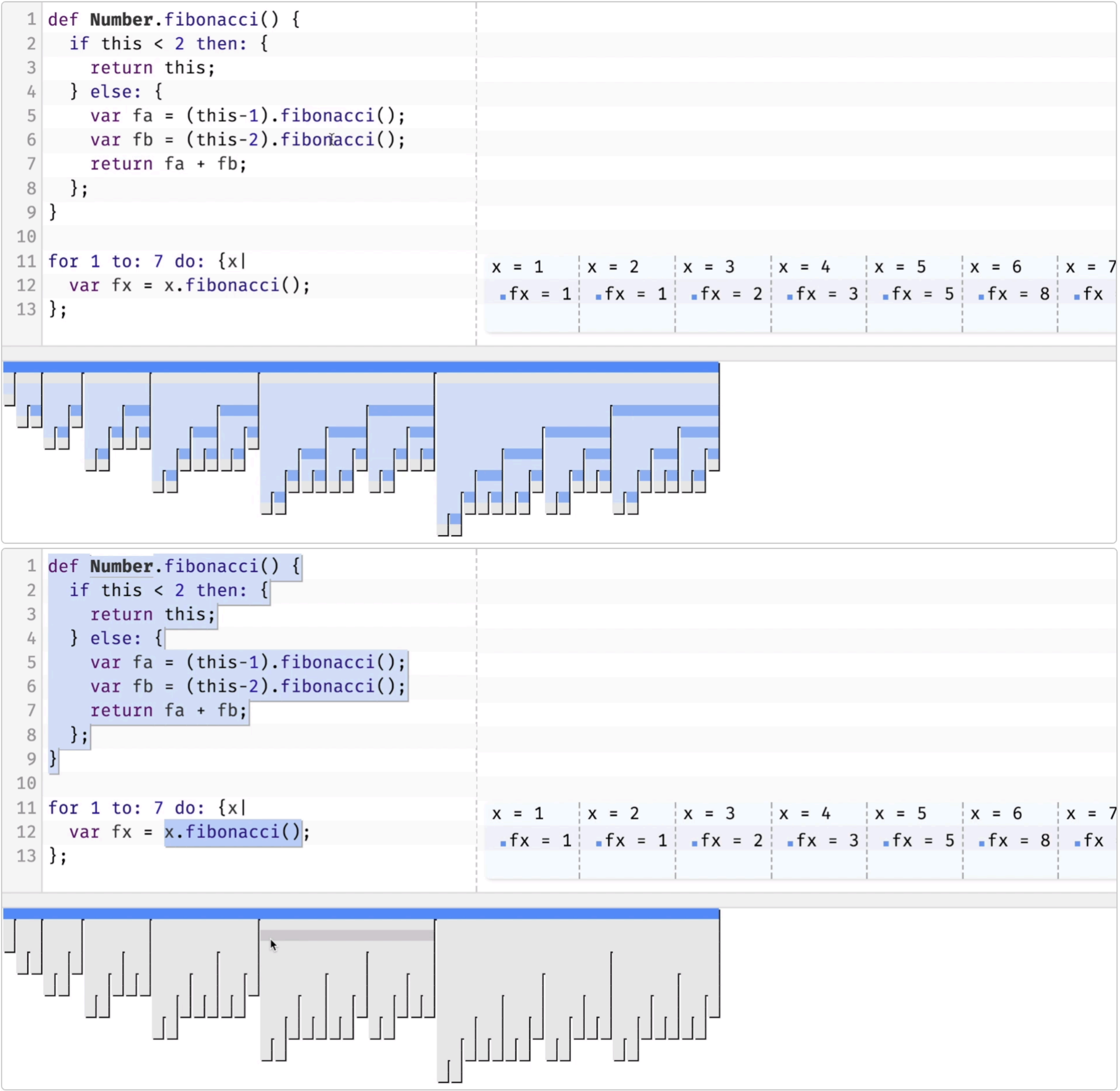
```
def Number.fib() {
  if this < 2 then: {
    return this;
  } else: {
    var fa = (this-1).fib();
    var fb = (this-2).fib();
    return fa + fb;
  };
}

for 1 to: 7 do: {x|
  var fx = x.fib();
};
```

```
x = 1   x = 2   x = 3   x = 4   x = 5   x = 6   x = 7
 fx = 1  fx = 1  fx = 2  fx = 3  fx = 5  fx = 8  fx =
```

Figure 2.10: The fibonacci sequence, visualized

Note the repeating structure in the macro visualization

the code, all nodes corresponding to that call in the macro visualization are highlighted. Similarly, when the user hovers over a node in the macro visualization, Seymour highlights the definition and site of the call in question.



```
def Number.fibonacci() {
  if this < 2 then: {
    return this;
  } else: {
    var fa = (this-1).fibonacci();
    var fb = (this-2).fibonacci();
    return fa + fb;
  };
}

for 1 to: 7 do: {x|
  var fx = x.fibonacci();
};
```

```
x = 1    x = 2    x = 3    x = 4    x = 5    x = 6    x = 7
 fx = 1   fx = 1   fx = 2   fx = 3   fx = 5   fx = 8   fx
```

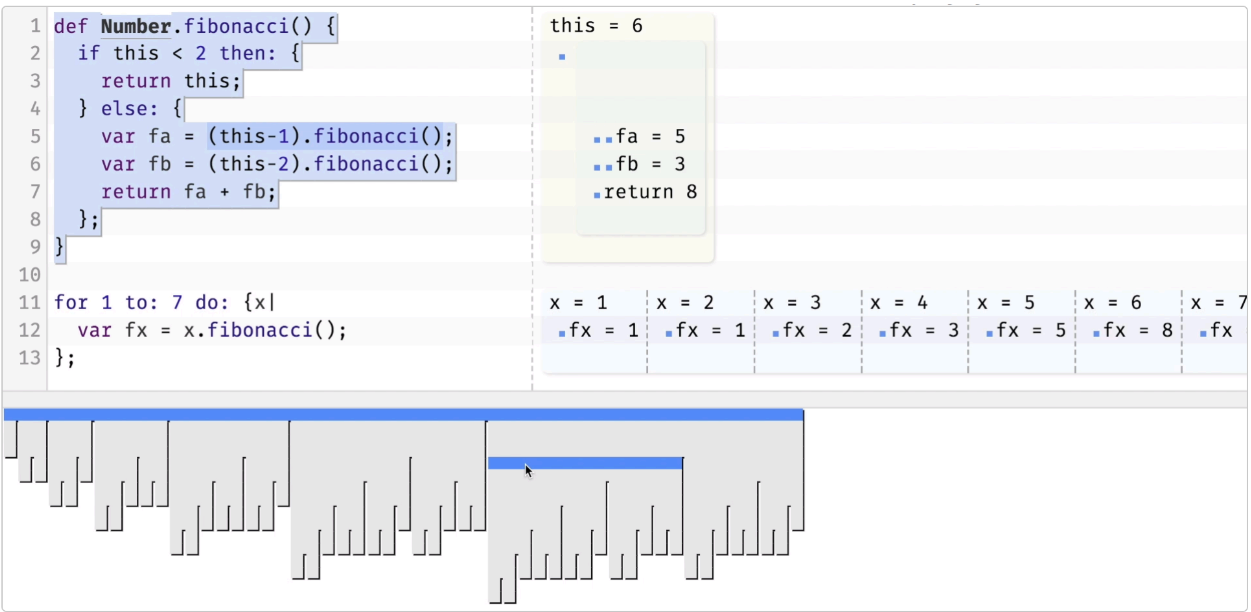
```
def Number.fibonacci() {
  if this < 2 then: {
    return this;
  } else: {
    var fa = (this-1).fibonacci();
    var fb = (this-2).fibonacci();
    return fa + fb;
  };
}

for 1 to: 7 do: {x|
  var fx = x.fibonacci();
};
```

```
x = 1    x = 2    x = 3    x = 4    x = 5    x = 6    x = 7
 fx = 1   fx = 1   fx = 2   fx = 3   fx = 5   fx = 8   fx
```

Figure 2.11: Parallel Highlighting

Hovering over code highlights the corresponding parts of the macro visualization and vice versa.

The visualization is also a user interface for navigating between calls in the micro visualization. When the user clicks on a node in the macro visualization, Seymour focuses the micro visualization on the call associated with that node. This makes it easy for the user to switch between contexts, drill down, and gather detailed information about the program. The currently focused call is colored dark blue in the macro visualization, along with all activations whose local variables the callee can access [3].



```
def Number.fibonacci() {
  if this < 2 then: {
    return this;
  } else: {
    var fa = (this-1).fibonacci();
    var fb = (this-2).fibonacci();
    return fa + fb;
  };
}

for 1 to: 7 do: {x|
  var fx = x.fibonacci();
};
```

```
this = 6

    fa = 5
    fb = 3
    return 8

x = 1  | x = 2  | x = 3  | x = 4  | x = 5  | x = 6  | x = 7
 fx = 1|  fx = 1|  fx = 2|  fx = 3|  fx = 5|  fx = 8|  fx
```

Figure 2.12: Focusing on a message send

Clicking on a message send in the macro visualization *focuses* it in the micro visualization

[3] Since every call can access the program's global state, the root node of the macro visualization is always highlighted

CHAPTER 3

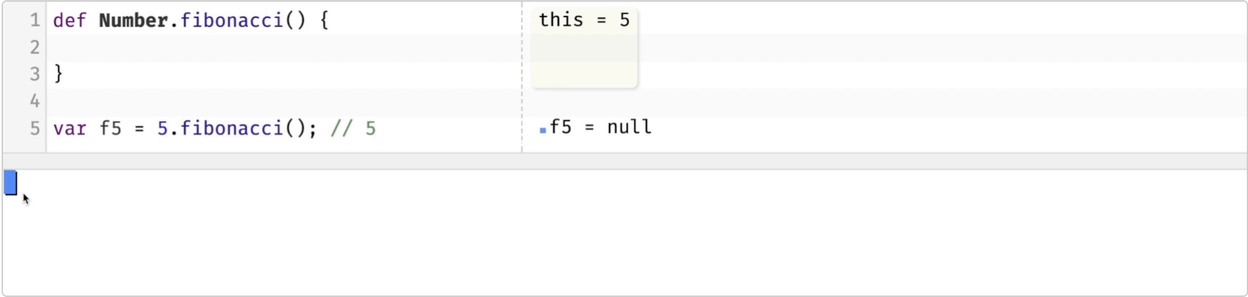
Live Programming with Seymour

In this section, we will give the reader a feel for Seymour's live programming experience. Several examples show how the macro visualization, the micro visualization, and 'focusing' come together to let the user see the execution of a program as they write it. In our original paper, each example consisted of a video followed by commentary. In this thesis, videos are replaced by key frames. The original video can be seen in the online paper [15].

3.1 Number.fibonacci()

In the previous section, we saw how a user might interact with a fully written implementation of `Number.fibonacci()`. This example shows how one might implement `fibonacci()` by getting several concrete examples working.

The user starts by writing a call to `fibonacci()` and a blank method definition. Initially, the plan is to get `5.fibonacci()` working.

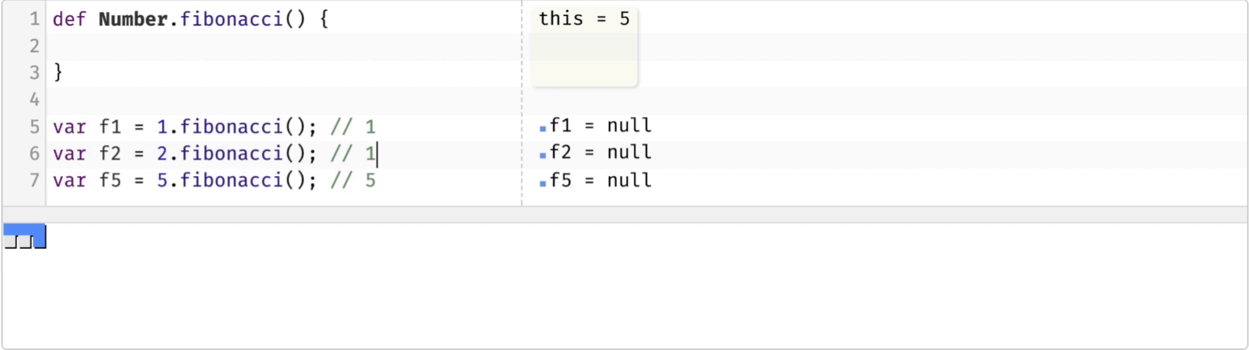


```
def Number.fibonacci() {

}

var f5 = 5.fibonacci(); // 5
```

```
this = 5



f5 = null
```

Figure 3.1: A blank method definition and test

Realizing that 5.fibonacci()s result depends on 4 and 3s results, the user decides to concentrate on 1.fibonacci() and 2.fibonacci() (the base cases) instead.

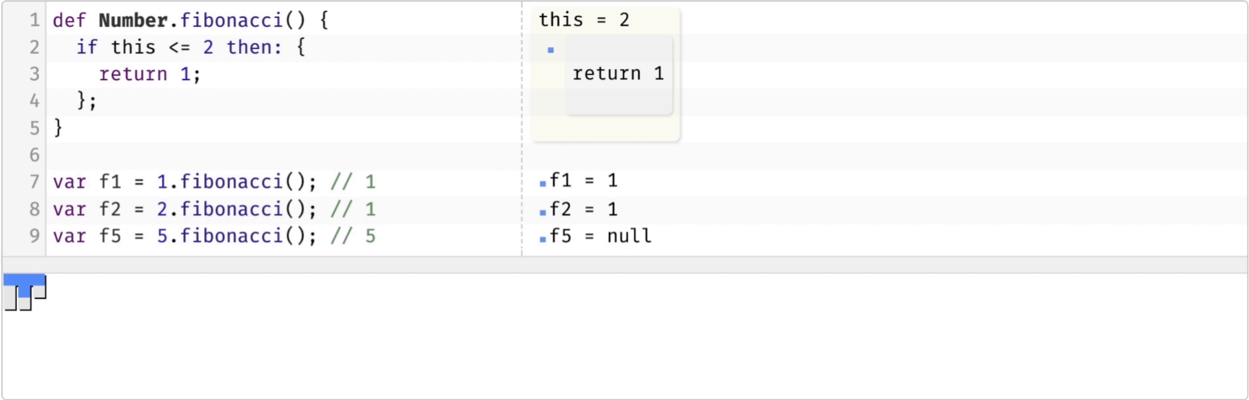


```
def Number.fibonacci() {

}

var f1 = 1.fibonacci(); // 1
var f2 = 2.fibonacci(); // 1
var f5 = 5.fibonacci(); // 5
```

```
this = 5



f1 = null
f2 = null
f5 = null
```

Figure 3.2: Focusing on the method definition

Focusing on 2.fibonacci(), the user writes code to handle the base cases. The calls to 1.fibonacci() and 2.fibonacci() update to show the correct answer. While the user edits the code, the micro visualization remains focused on the same call, allowing them to see how the edits affect the methods execution. This technique is described in McDirmid and Edwards' "Programming with Managed Time," [21] section 3.

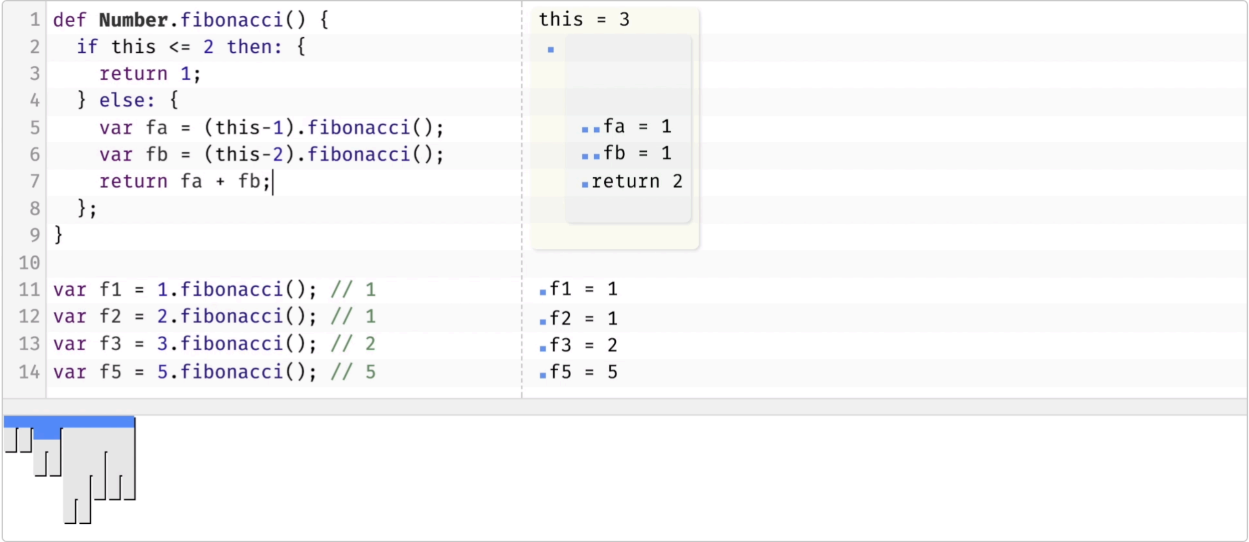


```
def Number.fibonacci() {
  if this <= 2 then: {
    return 1;
  };
}

var f1 = 1.fibonacci(); // 1
var f2 = 2.fibonacci(); // 1
var f5 = 5.fibonacci(); // 5
```

```
this = 2

   return 1



f1 = 1
f2 = 1
f5 = null
```

Figure 3.3: The Base Case

Now, they have all the parts necessary to write the rest of the implementation. They focus on 3.fibonacci() and implement the recursive case. They can see the results of each recursive call and use that information to complete the implementation. Similar methods for using runtime information are described in Allen's Anatomy of Lisp, section 6.21 [1] and Edwards' "Example Centric Programming" [8].



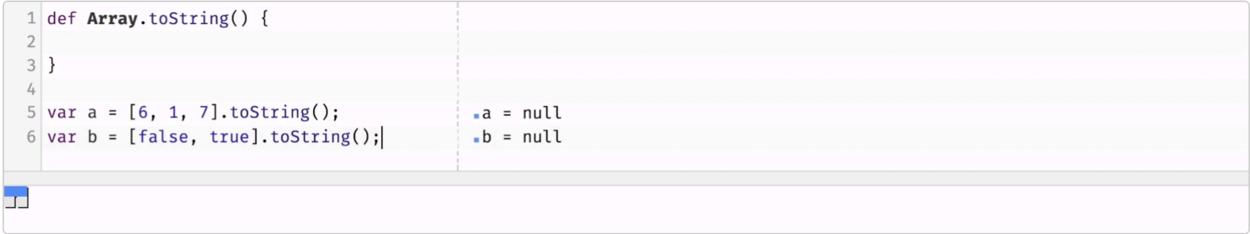
```
def Number.fibonacci() {                    this = 3
  if this <= 2 then: {
    return 1;
  } else: {
    var fa = (this-1).fibonacci();               fa = 1
    var fb = (this-2).fibonacci();               fb = 1
    return fa + fb;                             return 2
  };
}

var f1 = 1.fibonacci(); // 1                f1 = 1
var f2 = 2.fibonacci(); // 1                f2 = 1
var f3 = 3.fibonacci(); // 2                f3 = 2
var f5 = 5.fibonacci(); // 5                f5 = 5
```

Figure 3.4: A complete implementation of 'Number.fibonacci()'

3.2 Array.toString()

This example shows how the user can work with loops in Seymour. They will implement Array.toString(), which must include the string representation of element of an array.

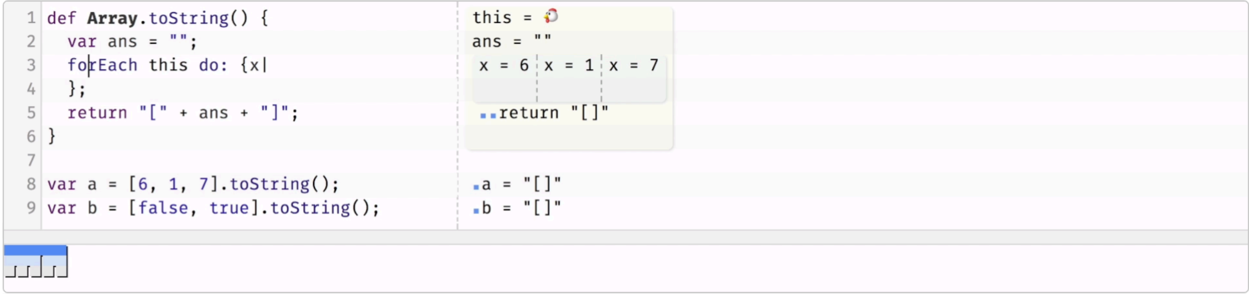
Once again, the user calls the method with a couple examples and tries to get them working.



```
def Array.toString() {

}

var a = [6, 1, 7].toString();              a = null
var b = [false, true].toString();          b = null
```

Figure 3.5: An empty method definition with tests

Writing the implementation continues in much the same manner as the last example. The user realizes that the result must show each element of the array, so they loop. The loop shows the value of each element. These values will be used as building blocks to compute the answer.

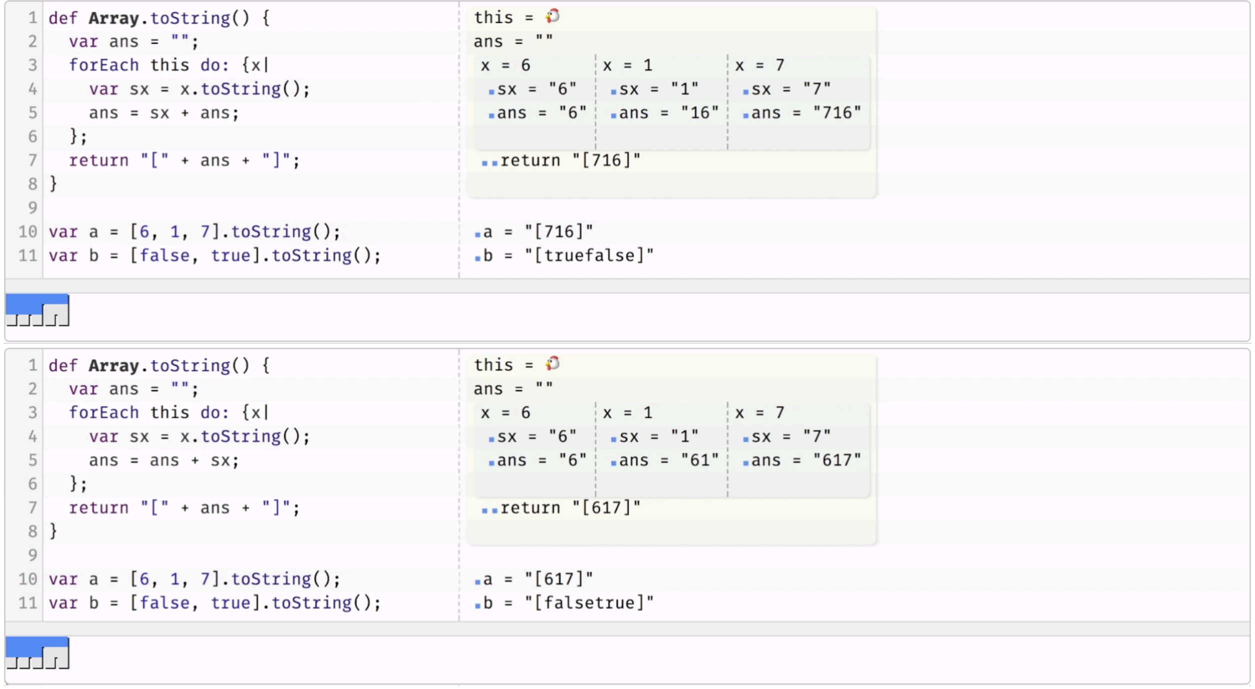


```
def Array.toString() {
  var ans = "";
  forEach this do: {x|
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

```
this =
ans = ""
 x = 6 | x = 1 | x = 7

 return "[]"

a = "[]"
b = "[]"
```

Figure 3.6: An incomplete implementation

The user takes advantage of the liveness of the environment and experiments with different ways to combining the elements of the array. Initially, they incorrectly prepend sx to ans. After seeing the items appended to ans in reverse order, the user corrects their mistake.



```
def Array.toString() {
  var ans = "";
  forEach this do: {x|
    var sx = x.toString();
    ans = sx + ans;
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

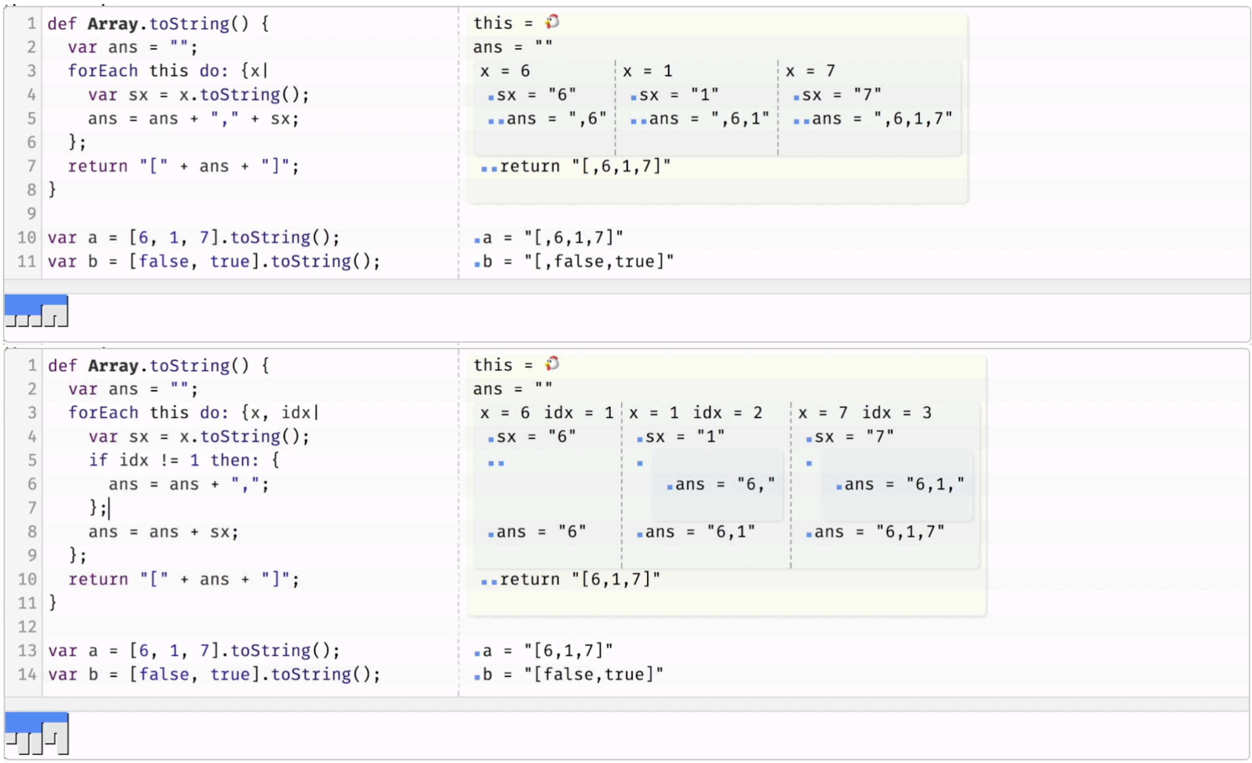
```
this =
ans = ""
 x = 6        | x = 1         | x = 7
 sx = "6"     | sx = "1"      | sx = "7"
 ans = "6"    | ans = "16"    | ans = "716"

 return "[716]"

a = "[716]"
b = "[truefalse]"
```

```
def Array.toString() {
  var ans = "";
  forEach this do: {x|
    var sx = x.toString();
    ans = ans + sx;
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

```
this =
ans = ""
 x = 6        | x = 1         | x = 7
 sx = "6"     | sx = "1"      | sx = "7"
 ans = "6"    | ans = "61"    | ans = "617"

 return "[617]"

a = "[617]"
b = "[falsetrue]"
```

Figure 3.7: Fixing an error in the implementation

The user tries out different ways to insert commas between each element. Putting a comma before every element does not work so they do not append a comma before the first element in the array.



```
def Array.toString() {
  var ans = "";
  forEach this do: {x|
    var sx = x.toString();
    ans = ans + "," + sx;
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

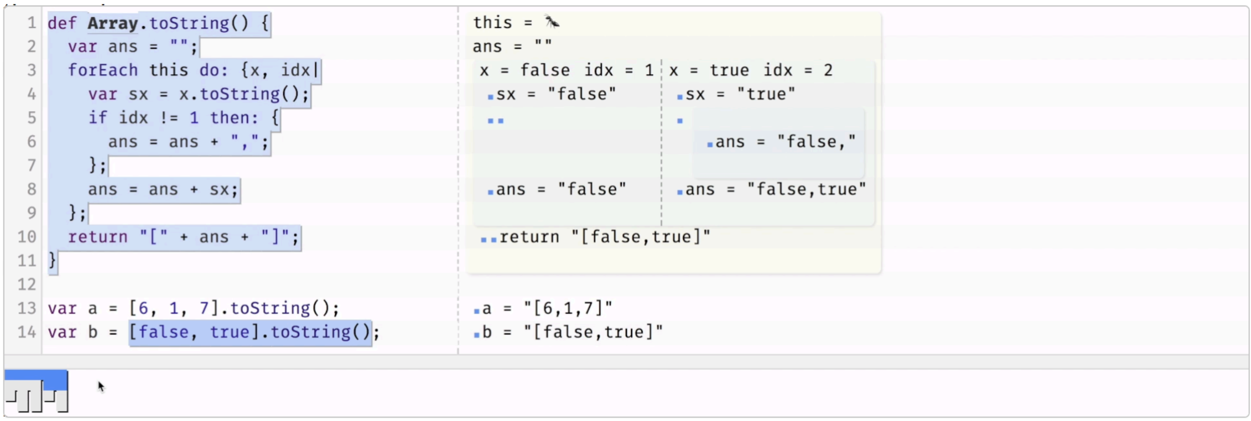
```
this =
ans = ""
 x = 6        | x = 1          | x = 7
  sx = "6"    |  sx = "1"      |  sx = "7"
  ans = ",6"  |  ans = ",6,1"  |  ans = ",6,1,7"
 return "[,6,1,7]"

a = "[,6,1,7]"
b = "[,false,true]"
```

```
def Array.toString() {
  var ans = "";
  forEach this do: {x, idx|
    var sx = x.toString();
    if idx != 1 then: {
      ans = ans + ",";
    };
    ans = ans + sx;
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

```
this =
ans = ""
 x = 6  idx = 1 | x = 1  idx = 2  | x = 7  idx = 3
  sx = "6"      |  sx = "1"       |  sx = "7"
                |                 |
                |    ans = "6,"   |    ans = "6,1,"

  ans = "6"     |  ans = "6,1"    |  ans = "6,1,7"
 return "[6,1,7]"

a = "[6,1,7]"
b = "[false,true]"
```

Figure 3.8: Adding comma separators

Finally, the user inspects the other call to make sure that the method executes as expected.



```
def Array.toString() {
  var ans = "";
  forEach this do: {x, idx|
    var sx = x.toString();
    if idx != 1 then: {
      ans = ans + ",";
    };
    ans = ans + sx;
  };
  return "[" + ans + "]";
}

var a = [6, 1, 7].toString();
var b = [false, true].toString();
```

```
this =
ans = ""
 x = false  idx = 1 | x = true  idx = 2
  sx = "false"      |  sx = "true"
                    |
                    |    ans = "false,"

  ans = "false"     |  ans = "false,true"
 return "[false,true]"

a = "[6,1,7]"
b = "[false,true]"
```

Figure 3.9: A complete implementation of ‘`Array.toString()`’

CHAPTER 4

Implementation

Seymour's implementation must support the features discussed in chapter 2. It must enable the programmer to

- see every message send from the program's execution in the macro visualization,
- focus on any send in the macro-visualization to show its details in the micro visualization,
- see side effect summaries at each call site (in the micro visualization), and
- visualize loop-like control structures (even those that are programmer-defined) in the micro visualization.

Furthermore, in order to be usable, the implementation must let the user interact with the UI before a program is finished running. This lets the user inspect the execution of a long running program (even if it is stuck in an infinite loop), and make changes based on what they see.

We implement such an interface using a pipeline that concurrently processes code, runs it, and produces visualizations. It consists of three major components—a code preprocessor, a language runtime, and a language-agnostic visualization framework. When the user types into the editor as shown in figure 4.1, information is processed by each component in the pipeline, resulting in the rendered visualizations at the end.

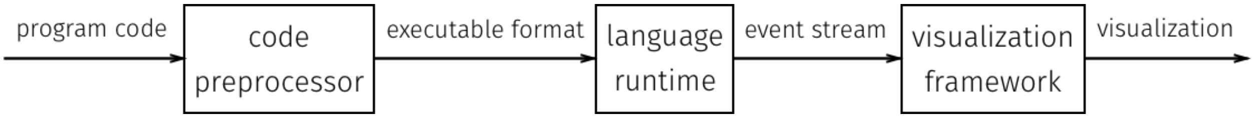


Figure 4.1: The pipeline for processing and visualizing programs as they change

First, the program text is parsed and converted into an executable format (the preprocessing phase). The workings of this component are determined by the language runtime used. Next, the language runtime generates a stream of events which are sent to the visualization framework. Implementation strategies for these two stages are discussed in section 4.1. Finally, the visualization framework uses the stream of events from the runtime to render the micro and macro visualizations. The algorithms used to render these visualizations are discussed in section 4.2.

4.1 Language Runtime Implementation Strategies

As with most widely used language implementations, our code preprocessing and language runtime stages take in code and execute it (while producing an event stream that describes the code's execution). However, our implementation has different goals than those of a REPL or compiler. Because a program might run for a very long time (or forever), it is crucial that the user receive and be able to act on feedback about the running program. They must be able to

- interact with visualizations while the program is running,
- change the program while it is running, and
- make these changes without losing track of the message send they are focused on.

As rendering visualizations takes longer than producing the event stream from which they are rendered, how quickly a program is run is not our primary concern. It is more important that our language runtime execute programs without hanging the UI, allowing for timely feedback. In this section, we present two language implementations—one for our own language and another for Python—that allow for this feedback.

4.1.1 Code Preprocessing

Before it is run, code is parsed and preprocessed into a format that the language runtime can work with. To parse the code, we used the Ohm parser generator library, which generates incremental parsers for parsing expression grammars [7]. Using an incremental parser improves parse times in interactive settings such as ours, where most edits are small. In addition, we can use parse tree nodes (many of which are preserved between edits) to identify the same message send in two different runs of a program, as described in McDirmid and Edwards' "Programming with Managed Time," section 3 [20]. This allows us to maintain focus on the same message send as a program is edited.

The parsed code is then converted into a format that its language runtime can work with. For Python, this is instrumented code, while for our own language, this is a set of instructions for a virtual machine. Figure 4.2 shows examples of these executable formats.

A

```
var sum = 0;
for 1 to: 3 do: {x|
  sum = sum + x;
};
```

B

```
Push 0
DeclVar sum // sum = 0

Push 1
Push 3
Block // pushes a block with the fol-
      // lowing body onto the stack
  // block body
  PushFromVar sum
  PushFromVar x
  Send + 1
  PopIntoVar sum // sum = sum + x
  Push null
  LocalReturn null // -> null

Send for_to:do: 2 // for 1 to: 3 do: 🐔
Drop
Done
```

C

```
sum = 0
for x in range(3):
  sum = sum + x
```

D

```
__currentEnv__ = R.program()
__env_0__ = __currentEnv__
__declEnv__ = __currentEnv__

R.memoize( "4_value", 0 )
sum = R.assignVar(1, __currentEnv__, __declEnv__, \
  "sum", R.retrieve("4_value"))

R.memoize( "14_iter", ( \
  R.memoize( "10_func", range ) , \
  R.memoize( "10_args", [5] ) , \
  R.send(2, __currentEnv__, \
    None, R.retrieve( "10_func" ) , \
    "range", R.retrieve( "10_args" ) , 10 \
  ), \
  R.receive( __currentEnv__,
    R.retrieve( "10_func" ), *R.retrieve("10_args")) \
 )[3])

__currentEnv__ = R.enterScope(3, __currentEnv__, 14)
__env_14_loop__ = __currentEnv__
__iteration_14__ = 0

for x in R.retrieve( "14_iter" ) :
  __currentEnv__ = R.enterScope(4, __currentEnv__, __iteration_14__)
  __env_14__ = __currentEnv__

  R.memoize( "-111_value", x )
  x = R.assignVar( 5, __currentEnv__, __declEnv__, \
    “x”, R.retrieve( "-111_value" )  )
  sum = R.assignVar( 6, __currentEnv__, __declEnv__, \
    "sum", (sum + x) )

  R.leaveScope(__currentEnv__)
  __currentEnv__ = __env_14_loop__
  __iteration_14__ = __iteration_14__ + 1

R.leaveScope( __currentEnv__ )
__currentEnv__ = __env_0__
```

Figure 4.2: A program and its executable formats

a) The code **b)** The code, converted to virtual machine instructions **c)** Uninstrumented Python code **d)** Instrumented Python code

4.1.2 Language Runtimes

Each of the language implementations is centered around a different strategy to let the user interact with visualizations and change the code while a program is running.

For our own language, we implemented a virtual machine in JavaScript. JavaScript is event driven, and has a single thread of execution[1]. As a result, a naïve implementation can cause the UI to hang when executing a long-running program. We avoid this by making our virtual machine execute a few instructions at a time before yielding to the UI. While executing each instruction, the VM also generates the relevant events for the event stream.

Because the VM and UI are in the same thread, the UI has easy access to the language runtime's memory. This has several advantages. Events are readily accessible to the UI as soon as they are generated, and they need not be serialized to be rendered. Also, objects in the language are readily accessible, making it easy to create more complex visualizations such as the one in figure 6.2.

However, keeping the language runtime in the same thread as the UI also has its disadvantages. The UI may maintain references to objects that would be garbage collected otherwise, leading to high memory usage. A virtual machine is also quite complex and may take a great deal of effort to implement for a language with more features than our own.

The runtime for Python uses a separate server to run the code. The code is instrumented to generate events and sent to the server using a web socket. The server uses the web socket to send events back to the UI as the code is executed. The server is written in Python, and takes advantage of an existing implementation of the language to run the code and generate events. As events are received in the UI, they are added to a queue to be processed. Only a few events are rendered at a time in order to keep the UI responsive.

Because this approach keeps the language runtime separated from the UI, events must be serialized and sent to the UI to be rendered. Any objects in memory that the UI wants to access must also be serialized, making accessing the program's memory difficult. However, this approach is signif-

[1] While JavaScript has Web Workers, which run code in a separate thread, their memory cannot be accessed from the main thread.

icantly simpler to implement, as one can rely on an existing language implementation, rather than creating their own. One need only write a parser and instrumenter for the target language.

4.2 Language Agnostic Visualizations

The events generated by the language runtime (e.g. the variable assignment '`sum = 1`', the message send '`#42.call()`', and the instance variable assignment '`#43.width = 5`') are used to update the macro and micro visualizations. Both of these visualizations are incremental and update with each event so that the user can see events shortly after they are generated. While this framework currently only includes the macro and micro visualizations, it can be used to render any visualization that can be generated from a stream of events.

4.2.1 Events

Events hold information about the operations that occur over the course of a program's execution. All events contain some basic information:

- the location of the statement or expression producing it in the source code (sourceLoc);
- an order number (orderNum), which encodes control flow information;
- and its environment (env), which represents the message send in which it was created

Each event also contains additional information based on its type. There are 8 types of events:

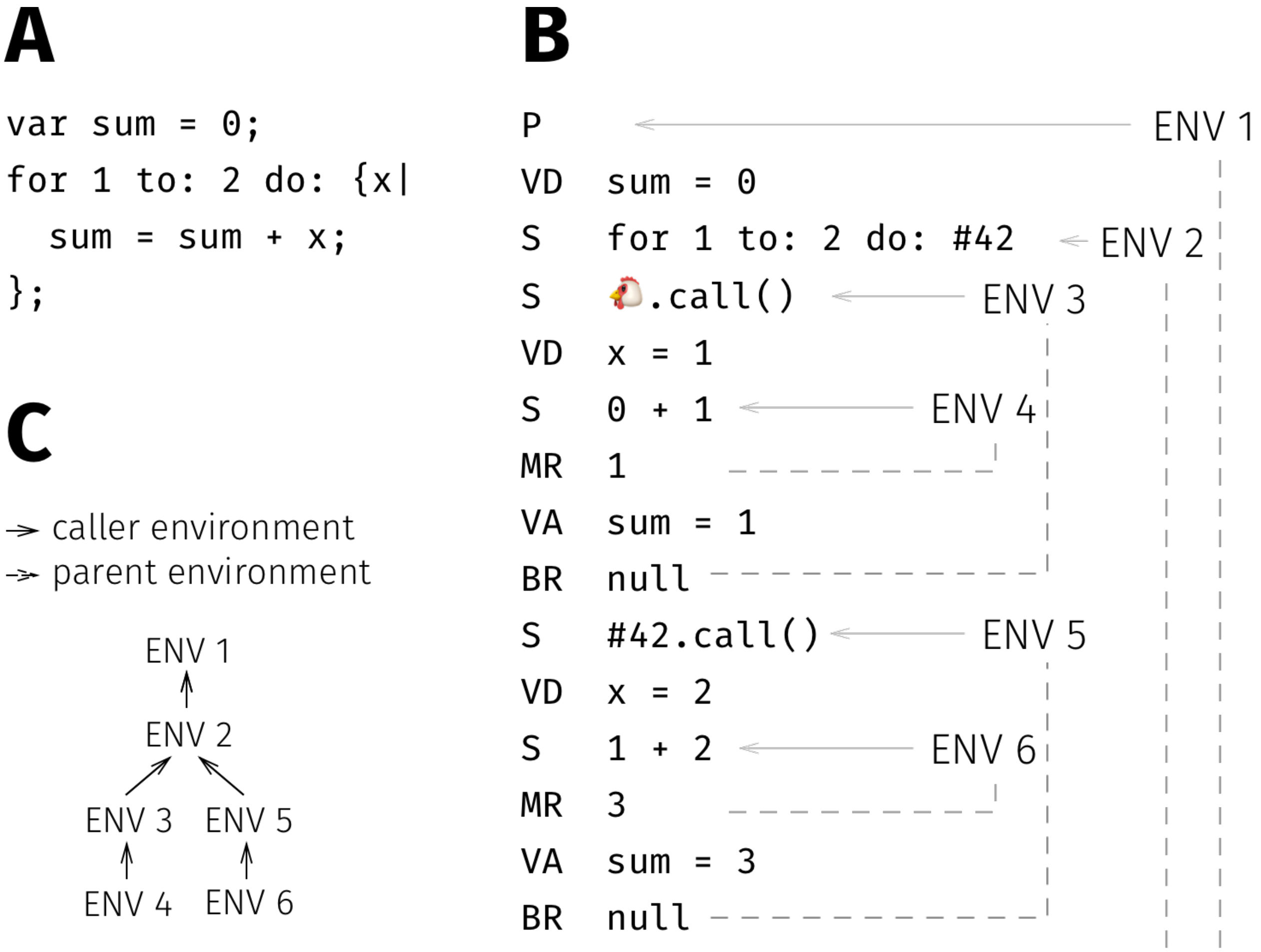
- A ProgramEvent (P) represents the execution of the entire program;
- a SendEvent (S) marks a message send, i.e. the call site of a method;
- a MethodReturnEvent (MR) marks a normal return from a method, or a nonlocal return from a block. In our language, both use the 'return' keyword and return from the closest method;
- a BlockReturnEvent (BR) marks a local return from a block. Its value is the last expression in the block. In Python, this event is used to return from a 'lambda' expression;

- a VarDeclEvent (VD) marks the declaration of a new variable;

- a VarAssignEvent (VA) marks the assignment of a variable;

- an InstantiationEvent (I) marks the creation of a new object;

- and an InstVarAssignEvent (IVA) marks an assignment to an object's instance variable.



```
var sum = 0;
for 1 to: 2 do: {x|
  sum = sum + x;
};
```

```
P
VD  sum = 0
S   for 1 to: 2 do: #42
S   🐔.call()
VD  x = 1
S   0 + 1
MR  1
VA  sum = 1
BR  null
S   #42.call()
VD  x = 2
S   1 + 2
MR  3
VA  sum = 3
BR  null
```

Figure 4.3: Events and environments for a program

a) the code for the program **b)** the events generated when the program is run and the environments in which they occur (🐔 is the identity of the block passed to the for method) **c)** the call tree for the generated environments

Figure 4.3a shows all the events generated from the execution of a simple program. Each event in the figure is paired with some of its identifying information. Also, note that each message send begins with a SendEvent (at the call site), and ends with a (method or block) return event.

4.2.2 Environments

Environments are used to organize events. There is a unique environment for every activation of every method. Just as a SendEvent represents a call site, its corresponding environment is where all the events that are created over the course of that call are stored. Each environment contains

- the sourceLoc of its method's definition,
- a reference to its corresponding SendEvent, and
- the environment of its caller (callerEnv).

Figure 4.3b shows each SendEvent's corresponding environment, and the events occurring in each environment. Together, the events occurring in an environment make up a trace of the execution of its message send, and are displayed when the user focuses on that send. Figure 4.3c shows the relationships between the environments, indicating each environment's callerEnv.

4.2.3 The Macro Visualization

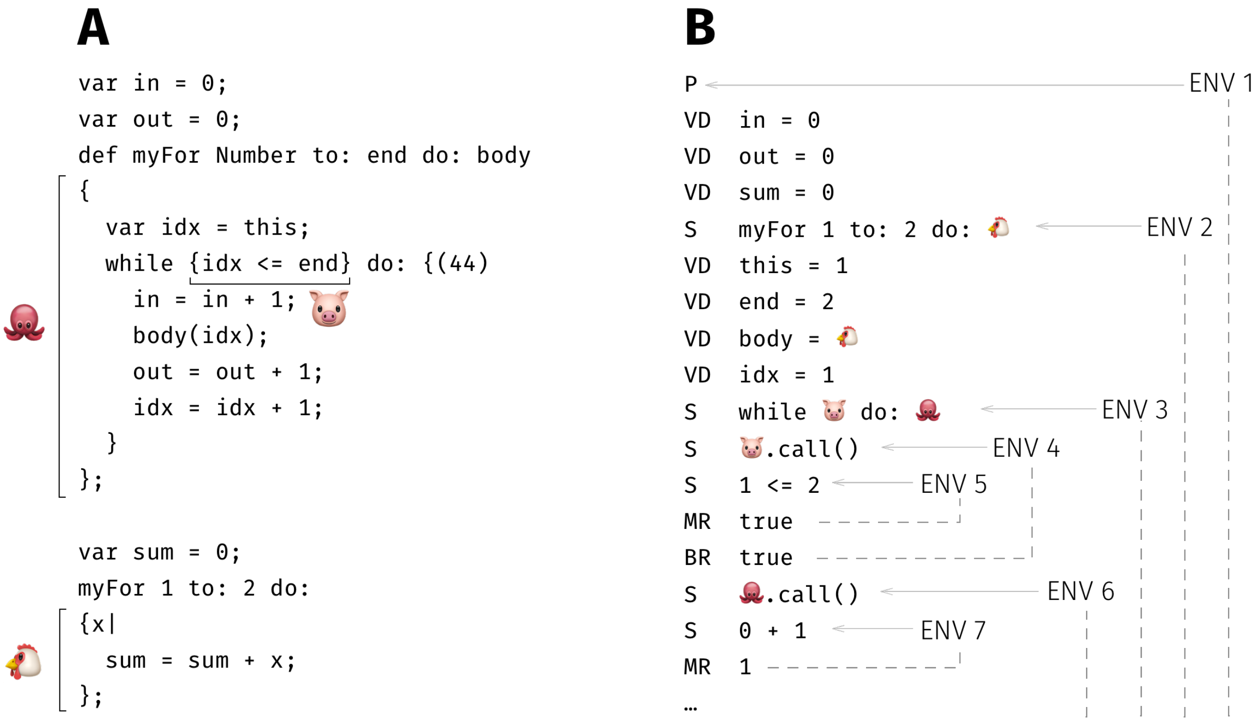
The macro visualization can be generated directly from the event stream as it has a simple tree structure. It can be built up as follows. Whenever an event is recorded, if it is a SendEvent or ProgramEvent, add a DOM node for the message send underneath its parent's DOM node.

4.2.4 The Micro Visualization

Unlike the macro visualization, where each send is placed in a single global structure, a single event may appear in multiple places in the micro visualization. For example, in figure 4.3(3) and (2), ‘`sum = 1`’ appears in the views for ‘`myFor 1 to: 2 do: 🐔`’ and ‘`Program`’. Furthermore, the events for a message send must be organized into loop iterations and side effect summaries to be visualized in a columnar format, as in figure 4.3d.

To address these challenges, we introduce the notion of models (rectangles with solid borders) and event groups (rectangles with dashed borders). Models are collections of events that correspond to visual elements in the micro visualization. We have models to represent each call site and method implementation. Event groups further organize models into iterations and side effect summaries.

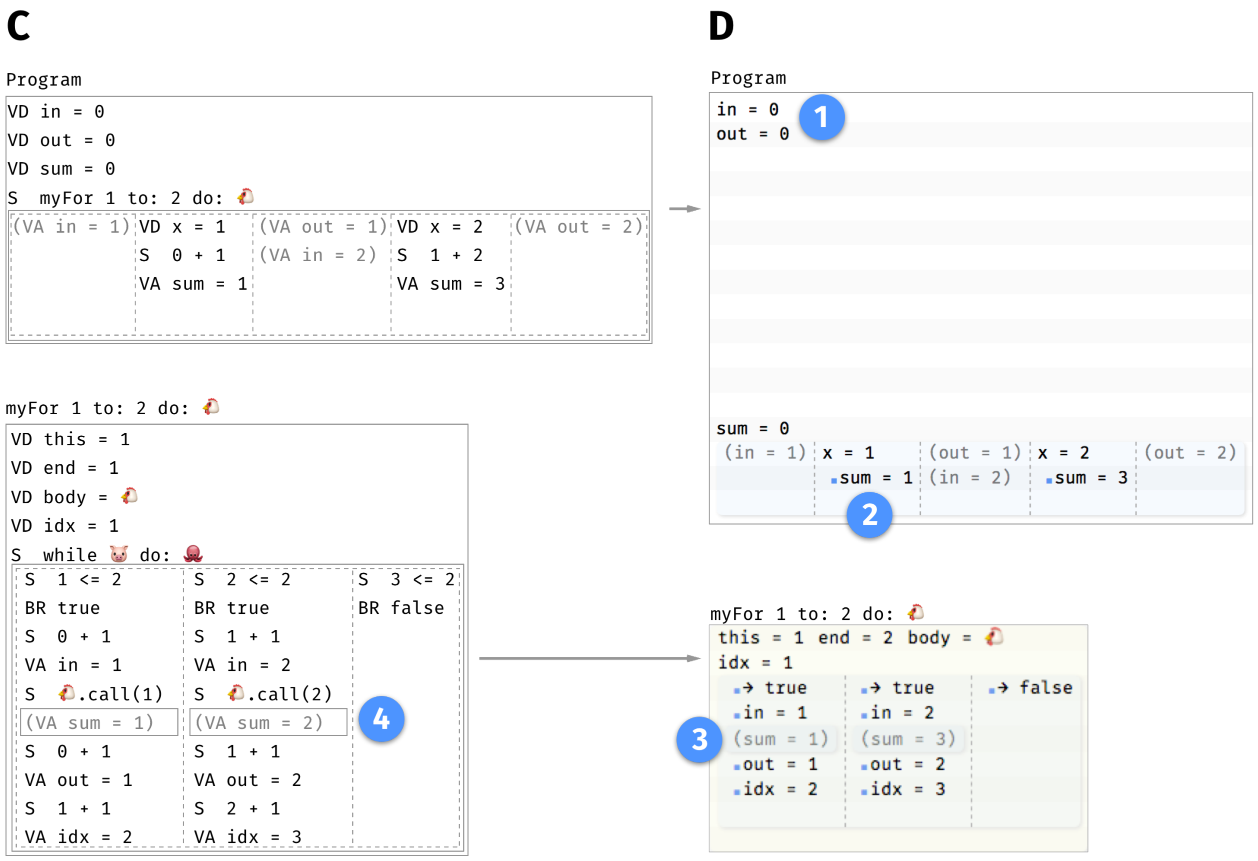
When an event is processed, we find all the models to which it belongs and add it there (resulting in the structure shown in figure 4.3c). Separately, as models are populated, a final stage renders each model into DOM nodes to create the visualization (resulting in the views in figure 4.3d).



```
var in = 0;
var out = 0;
def myFor Number to: end do: body
{
  var idx = this;
  while {idx <= end} do: {(44)
    in = in + 1; 🐷
    body(idx);
    out = out + 1;
    idx = idx + 1;
  }
};

var sum = 0;
myFor 1 to: 2 do:
{x|
  sum = sum + x;
};
```

```
P                                   ENV 1
VD  in = 0
VD  out = 0
VD  sum = 0
S   myFor 1 to: 2 do: 🐔             ENV 2
VD  this = 1
VD  end = 2
VD  body = 🐔
VD  idx = 1
S   while 🐷 do: 🐙                  ENV 3
S   🐷.call()             ENV 4
S   1 <= 2         ENV 5
MR  true
BR  true
S   🐙.call()              ENV 6
S   0 + 1          ENV 7
MR  1
…
```

(figure continued on next page)



```
Program
VD in = 0
VD out = 0
VD sum = 0
S  myFor 1 to: 2 do: 🐔
(VA in = 1) VD x = 1   (VA out = 1) VD x = 2   (VA out = 2)
            S  0 + 1   (VA in = 2)  S  1 + 2
            VA sum = 1              VA sum = 3
```

```
myFor 1 to: 2 do: 🐔
VD this = 1
VD end = 1
VD body = 🐔
VD idx = 1
S  while 🐷 do: 🐙
S  1 <= 2      S  2 <= 2      S  3 <= 2
BR true        BR true        BR false
S  0 + 1       S  1 + 1
VA in = 1      VA in = 2
S  🐔.call(1)  S  🐔.call(2)
(VA sum = 1)   (VA sum = 2)
S  0 + 1       S  1 + 1
VA out = 1     VA out = 2
S  1 + 1       S  2 + 1
VA idx = 2     VA idx = 3
```

```
Program
in = 0
out = 0
sum = 0
(in = 1) x = 1     (out = 1) x = 2     (out = 2)
         ▪sum = 1  (in = 2)  ▪sum = 3
```

```
myFor 1 to: 2 do: 🐔
this = 1  end = 2  body = 🐔
idx = 1
 ▪→ true     ▪→ true     ▪→ false
 ▪in = 1     ▪in = 2
 (sum = 1)   (sum = 3)
 ▪out = 1    ▪out = 2
 ▪idx = 2    ▪idx = 3
```

Figure 4.3: A program that adds up numbers from 1 to 2, keeping track of each time the 🐔 block is called

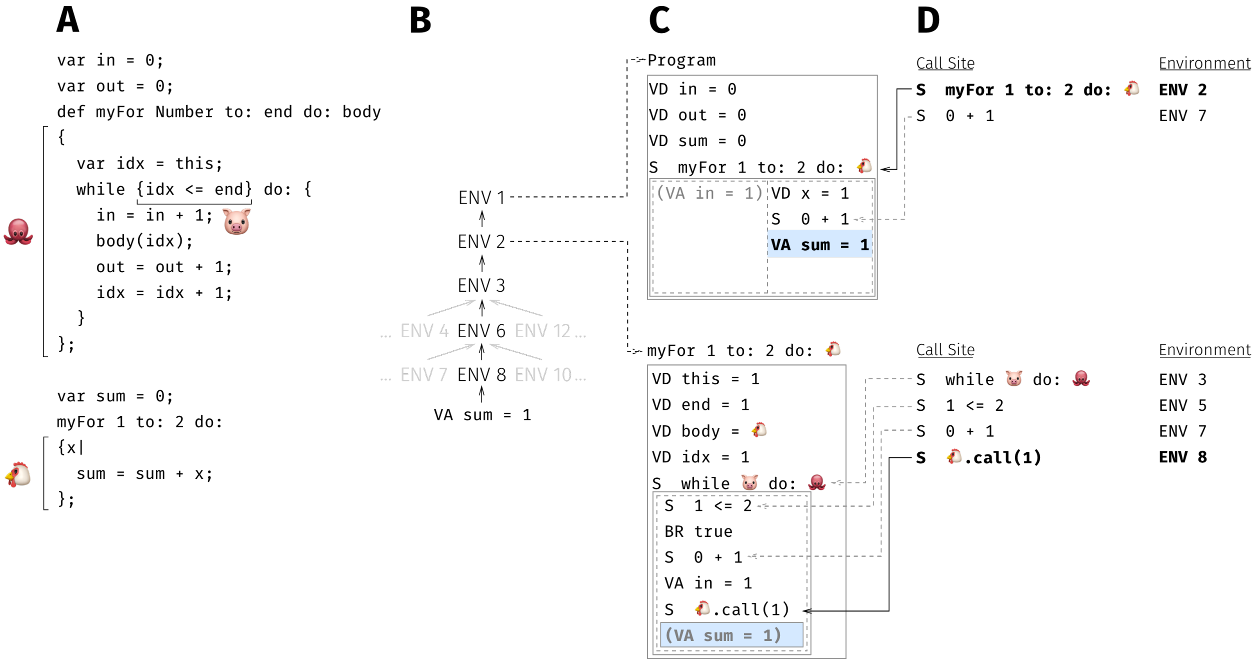
a) the code **b)** the program's event stream **c)** the models and groups for each event stream (groups are shown in dashed lines while models are shown with solid lines) **d)** the rendered views. **(1)** shows local events, **(2)** shows events in a loop iteration, and **(3)** shows events appearing as side effects. (4) shows a call site nested within another call site.

Selecting Models

An event may appear in views for many different message sends. Thus, it may be added to many different models. The process for selecting models for the event is illustrated in figure 4.4. Because an event can only show up in an implementation, an iteration, or a side effect summary, the models to which an event could be added are restricted to those associated with environments in the event's call chain. We can take advantage of this restriction to identify the relevant models.

Environments in the call chain are associated with several models to which an event may be added. Each environment has one model for its implementation (indicated with a dashed arrow in figure 4.4) and several models for each of the call sites that occur while it is active. It also maintains a list of all the call sites that occur within iterations of those call sites, and so on. figure 4.4 shows the list of call sites and environments for two implementation models.

For each environment in the call chain, an event may only appear in the most specific model for that environment. If the environment in question is the event's own environment, the most specific model is the environment's implementation model. Otherwise, it is the call site model whose send event is closest to event.env in the call chain. In figure 4.4, the most specific model for '`sum = 1`' is highlighted in each call site list. If there is no model from the event's call chain, then the event is not added to any model for the environment in question.



```
var in = 0;
var out = 0;
def myFor Number to: end do: body
{
  var idx = this;
  while {idx <= end} do: {
    in = in + 1; 🐷
    body(idx);
    out = out + 1;
    idx = idx + 1;
  }
};

var sum = 0;
myFor 1 to: 2 do:
{x|
  sum = sum + x;
};
```

Call Site	Environment
S myFor 1 to: 2 do: 🐔	**ENV 2**
S 0 + 1	ENV 7

Call Site	Environment
S while 🐷 do: 🐙	ENV 3
S 1 <= 2	ENV 5
S 0 + 1	ENV 7
S 🐔.call(1)	**ENV 8**

Figure 4.4: Selecting models and groups for '`sum = 1`'

a) the code. Each block is annotated with an emoji id. **b)** the call chain **c)** implementation models for ENV1 and ENV2 **d)** the list of call sites for ENV1 and ENV2. The most specific call site for '`sum = 1`' is bolded, and the location where '`sum = 1`' is added is highlighted in blue.

Detecting Event Context

Once models for an event are identified, we can proceed to add the event to these models. In order to do so, we must detect the context in which it is shown.

An event can appear in three contexts:

1. *locally* (figure 4.3 (1)) if it occurred directly in the view's environment;

2. *in a loop iteration* (figure 4.3 (2)) as part of a code block passed to a control structure; or

3. *in a side effect summary* (figure 4.3 (3)) as one of a message send's side effects.

An event only appears locally in its environment's implementation container. However, it can appear in an iteration at a call site where the call site's sourceLoc contains the event's block's sourceLoc. In other words, if the event occurred inside a block which is written inline as a message send's argument, then the event is inside one of the call site's iterations. In figure 4.4, '`sum = 1`' appears in the model for '`Program`' in one of `myFor`'s iterations, as it occurs in an inline block.

If the event does not appear in an iteration, it still may appear as a side effect. This can only occur if it is an InstantiationEvent, it is an InstVarAssignEvent, or it is a VarAssignmentEvent and the environment to which it is being added is a callee of the event's declaring environment. In other words, a VarAssignmentEvent cannot appear as a side effect in the callers of its declaring message send. In figure 4.4, '`sum = 1`' appears and as a side effect for '🐔`.call()`' in the model for '`myFor`', as it is an assignment to a global variable.

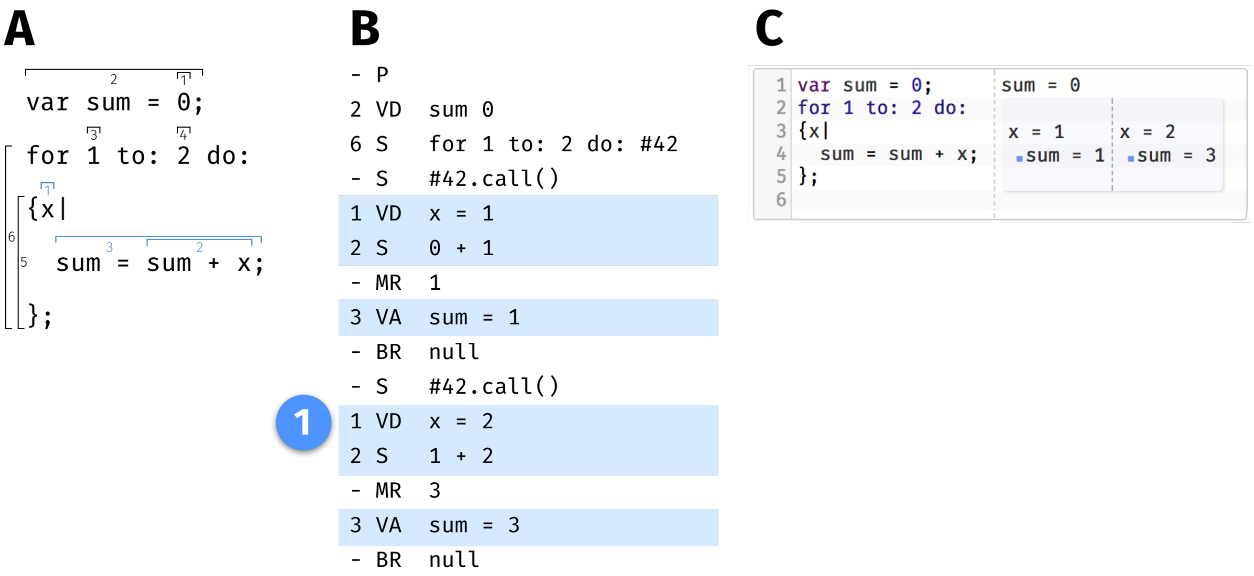
If an event does not appear locally, in an iteration, or as a side effect, then it is not added to the chosen model.

Detecting Loops

When adding events to models, they must be further organized into event groups—iterations and side effect summaries. An implementation model has only one iteration, while call site models may have many iterations and summaries. Iterations and summaries may alternate as in figure 4.3d, or iterations may appear back to back, when there are no side effects between them.

Since an iteration may follow another iteration, new iterations must be detected. orderNums are used to recognize a new iteration. They indicate the order in which expressions and statements would execute if the program had no loops, and can be generated statically, with one set of numbers for each method or block definition in the program. Figure 4.5 shows a program and event stream annotated with order numbers.

If the current event has a lower order number than the last event that was added, it is clear that there must be a new iteration, as an event that would be executed earlier than the previous event in straight line code was executed later on [2]. We see this demonstrated in figure 4.5, where the highlighted events are all being included in the same call site model. Since event (2) has a lower order number than event (1), it must be part of a new iteration.



```
var sum = 0;
for 1 to: 2 do:
{x|
  sum = sum + x;
};
```

```
- P
2 VD   sum 0
6 S    for 1 to: 2 do: #42
- S    #42.call()
1 VD   x = 1
2 S    0 + 1
- MR   1
3 VA   sum = 1
- BR   null
- S    #42.call()
1 VD   x = 2
2 S    1 + 2
- MR   3
3 VA   sum = 3
- BR   null
```

```
var sum = 0;       sum = 0
for 1 to: 2 do:
{x|                 x = 1      x = 2
  sum = sum + x;    sum = 1    sum = 3
};
```

Figure 4.5: Detecting loops using orderNums

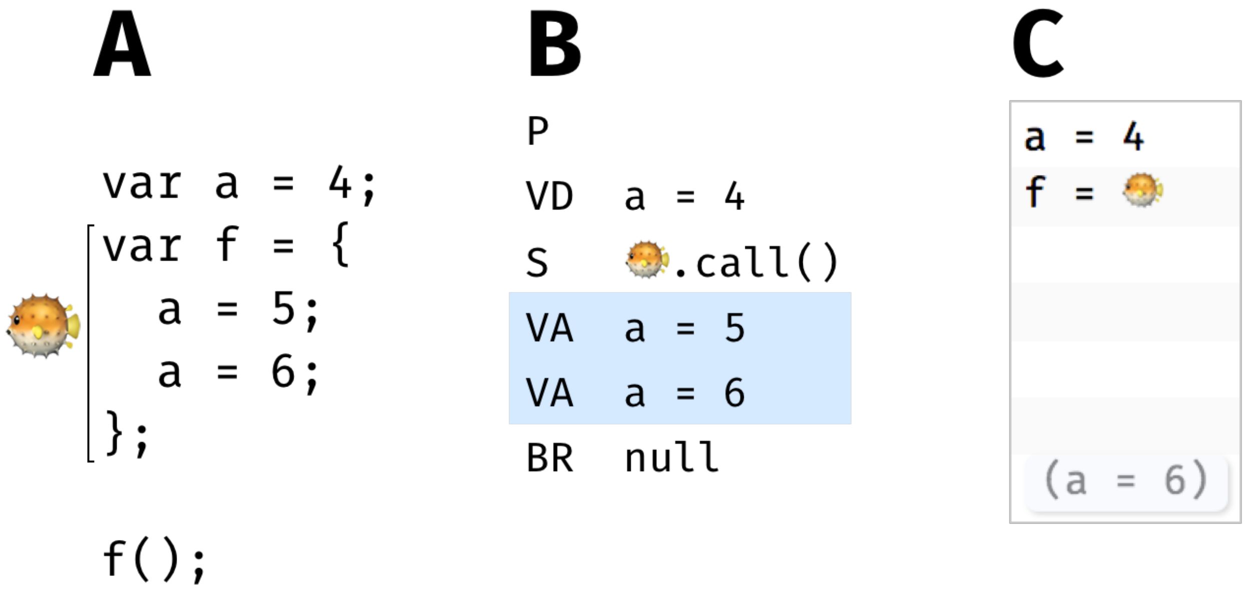
a) the code. Each expression and statement are annotated with order numbers. Annotations in black are for the program, while annotations in blue are for the block. **b)** the event stream. Each event's orderNum is shown to its left. (1) shows the point in the stream where a new iteration is detected. **c)** The final visualization.

[2] This technique was developed with input from Toby Schachman, who shared an early version of the idea that used sourceLocs instead of orderNums.

Ensuring Minimal Side Effect Summaries

Unlike iterations, which simply keep track of all events that occur within them, a side effect summary must show ‘just enough’ information to convey the changes that a method’s implementation has made to other state. For example, in figure 4.6, the variable ‘`a`’ is written twice in the call to ‘`f`’, but only the second VarAssignmentEvent is displayed, as it captures the final state of ‘`a`’.

To create minimal summaries, we use subsumption. A later event subsumes an earlier one if they both write to the same variable (same name, declaring environment) or instance variable. For example, ‘`a = 6`’ subsumes ‘`a = 5`’ in the call to ‘`f`’. Summaries only keep the last write to a variable, removing older subsumed events as new ones are added. This ensures that the final summary will contain the minimal set of events that describe the summarized changes to nonlocal state.



```
var a = 4;
var f = {
  a = 5;
  a = 6;
};

f();
```

```
P
VD  a = 4
S   🐡.call()
VA  a = 5
VA  a = 6
BR  null
```

```
a = 4
f = 🐡

(a = 6)
```

Figure 4.6: Subsumption

a) the program **b)** the event stream. The latter of the two highlighted events subsumes the former. **c)** The final visualization

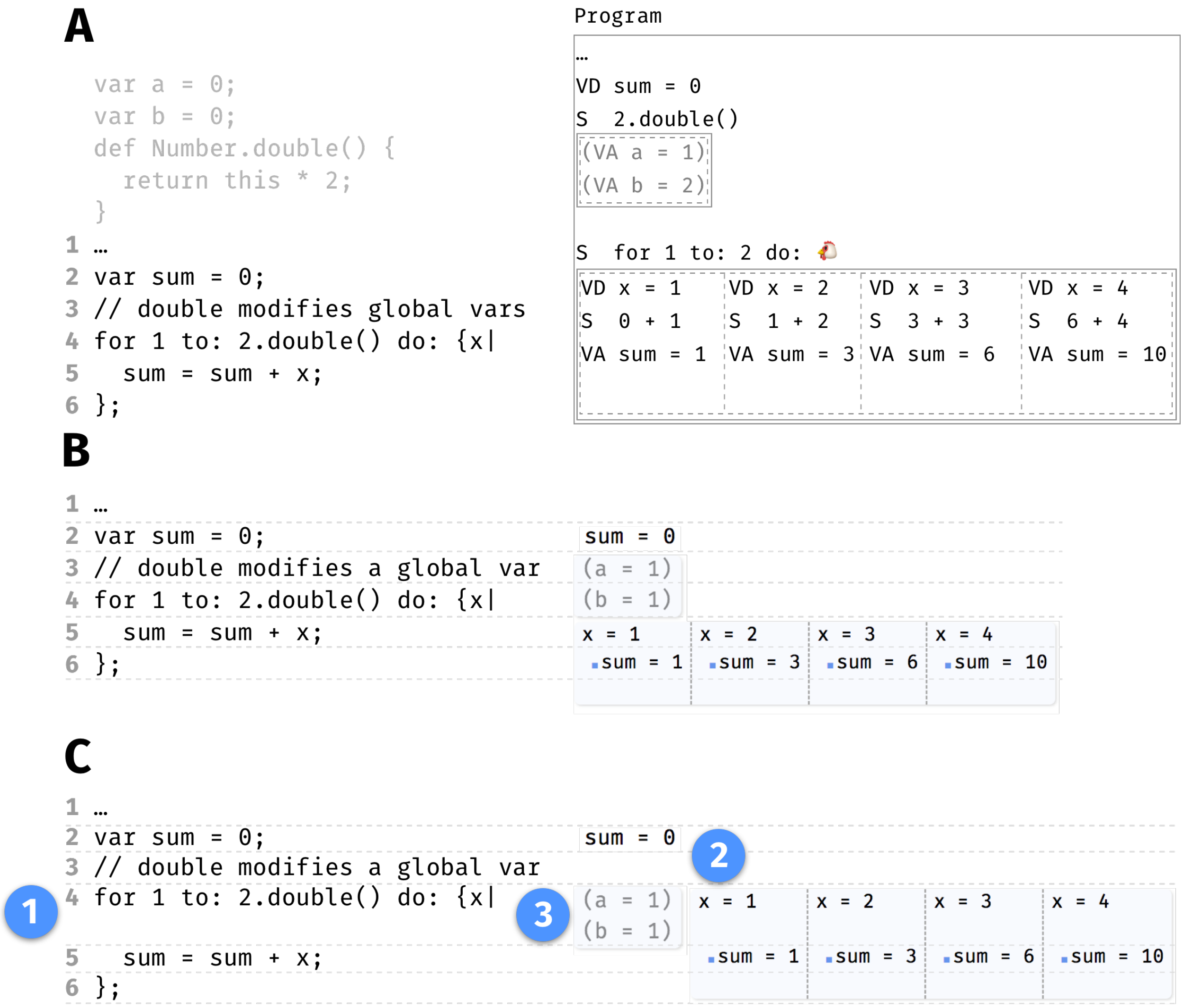
Rendering Models

Because models and groups correspond to visual elements in the micro visualization, one might assume that models can be rendered directly into DOM nodes to produce the final view. To do so, we would create a 'div' element for each model and group. Rendered events and call site models would then be added to the 'div' for their group.

However, doing so would lead to issues when rendering differently formatted programs. Figure 4.7a shows a model for one such program. Naïvely rendering this model produces the view shown in 4.7b. This view is difficult to understand as events do not line up with the lines of code that produce them. The side effect summary for '`2.double()`' is placed too high, while the loop is placed too low. The layout in figure 4.7c makes it easier to map events to the code that produced them.

Also, when rendering the view for a message send, we must render the closest activation of each of the message send's containing methods (or blocks) in the stack. We refer to these activations as parents of the current activation. This ensures that the user is aware of all state changes that might affect a message send's execution. Figure 4.8 shows an example of this nesting. Because we can see the execution of the program alongside the execution of the call to 'inc()', we know that `numCalls` is 0 before inc changes it to 1.

Both these problems can be solved using *spacers* and *wrappers*. These are extra DOM nodes that modify the naïvely rendered DOM structure, changing the placement of events in the final view. Spacers separate events, pushing them apart to match the positioning of statements in code. Wrappers keep events together, grouping them into a single DOM node. Adding spacers and wrappers along with rendered events and call site models ensures that the final visualization is readable, as shown in 4.7c and 4.8.



```
  var a = 0;
  var b = 0;
  def Number.double() {
    return this * 2;
  }
1 …
2 var sum = 0;
3 // double modifies global vars
4 for 1 to: 2.double() do: {x|
5   sum = sum + x;
6 };
```

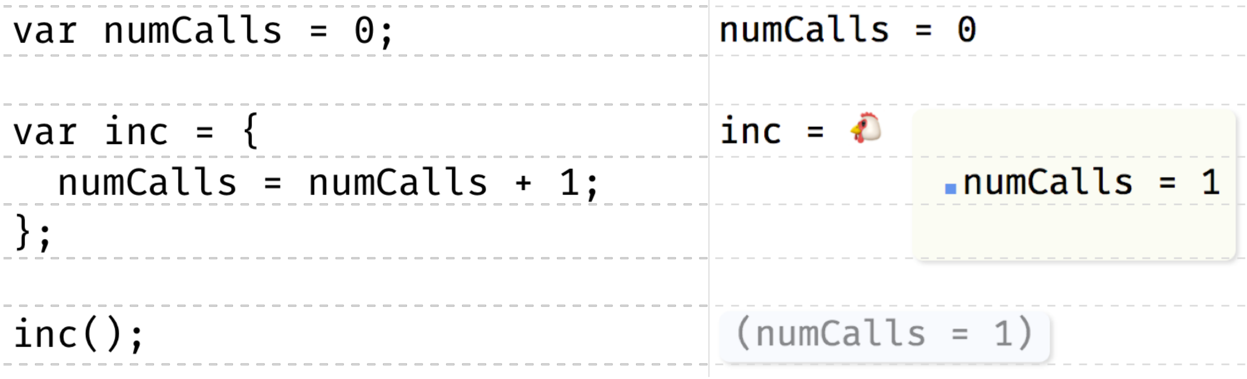
```
Program
…
VD sum = 0
S  2.double()
(VA a = 1)
(VA b = 2)

S  for 1 to: 2 do: 🐔
VD x = 1     VD x = 2     VD x = 3     VD x = 4
S  0 + 1     S  1 + 2     S  3 + 3     S  6 + 4
VA sum = 1   VA sum = 3   VA sum = 6   VA sum = 10
```

```
1 …
2 var sum = 0;                       sum = 0
3 // double modifies a global var    (a = 1)
4 for 1 to: 2.double() do: {x|       (b = 1)
5   sum = sum + x;                   x = 1     x = 2     x = 3     x = 4
6 };                                  ▪sum = 1  ▪sum = 3  ▪sum = 6  ▪sum = 10
```

```
1 …
2 var sum = 0;                       sum = 0
3 // double modifies a global var
4 for 1 to: 2.double() do: {x|       (a = 1)  x = 1     x = 2     x = 3     x = 4
                                     (b = 1)
5   sum = sum + x;                             ▪sum = 1  ▪sum = 3  ▪sum = 6  ▪sum = 10
6 };
```

Figure 4.7: Naïve Rendering vs. Correct Rendering

a) code and implementation model for the program **b)** a naïve rendering of the model **c)** a correct rendering of the model **(1)** Line 4 is padded to make space for the side effects of the micro visualization. **(2)** The loop and side effect summary are pushed down to keep line 3 empty. **(3)** The side effect summary for '`2.double()`' and the for loop are placed on the same line, as in the code.



```
var numCalls = 0;

var inc = {
  numCalls = numCalls + 1;
};

inc();
```

```
numCalls = 0

inc = 🐔
                ▪numCalls = 1


(numCalls = 1)
```

Figure 4.8: Composing Views

The implementation view of the call to '`inc()`' is composed with the implementation view for the whole program showing that numCalls was set to 0 before `inc()` was called.

Aligning the Micro Visualization and Code

In order to guarantee that rendered events added to a group's div line up with the code that produced them, spacers and wrappers may also be added. Doing so fixes alignment issues such as those in figure 4.7b. Spacers are used to keep lines without any effects empty, as in 4.7(2), and a combination of spacers and wrappers help line events up with the code that produced them as in 4.7(3).

As rendered events are added to their group's div, they are tagged with the lines on which they start and end. They are laid out from left to right and top to bottom, as with text. Each rendered event is styled with 'display: inline-block', and lines are broken up using 'br' tags. Figure 4.11a shows the program in figure 4.7 with spacers, wrappers and line breaks added.

If several lines are skipped before adding an event (i.e. the last event's end line is several lines before the new event's start line, as in 4.7(2)), a spacer is added for each skipped line. Further spacers and wrappers are added if the last event overlaps with the new one. The logic for adding rendered events is illustrated in figure 4.9.

If the event being added is the first in its line, as in 4.9a, then it is added to the group directly without spacers or wrappers. If the previous event starts on the same line as the event being added (4.9b), then it is also added directly.

If the new event overlaps with the last event but starts on a later line (4.9c), it is added to a wrapper. The wrapper has the same starting line as the previous event and contains spacers that push the new event down to its starting line.

Finally, if the new event's extent contains those of several other events (e.g. if it is a function call with other function calls producing its arguments, as in 4.9d), then the contained events are placed in a wrapper whose starting line matches that of the new event. Spacers may be added before the first contained event to push it down to the correct line. The new event is then added in the same manner as case 4.9b. Nested views are also added to the views of their parent calls in this manner, as shown in figure 4.11b.

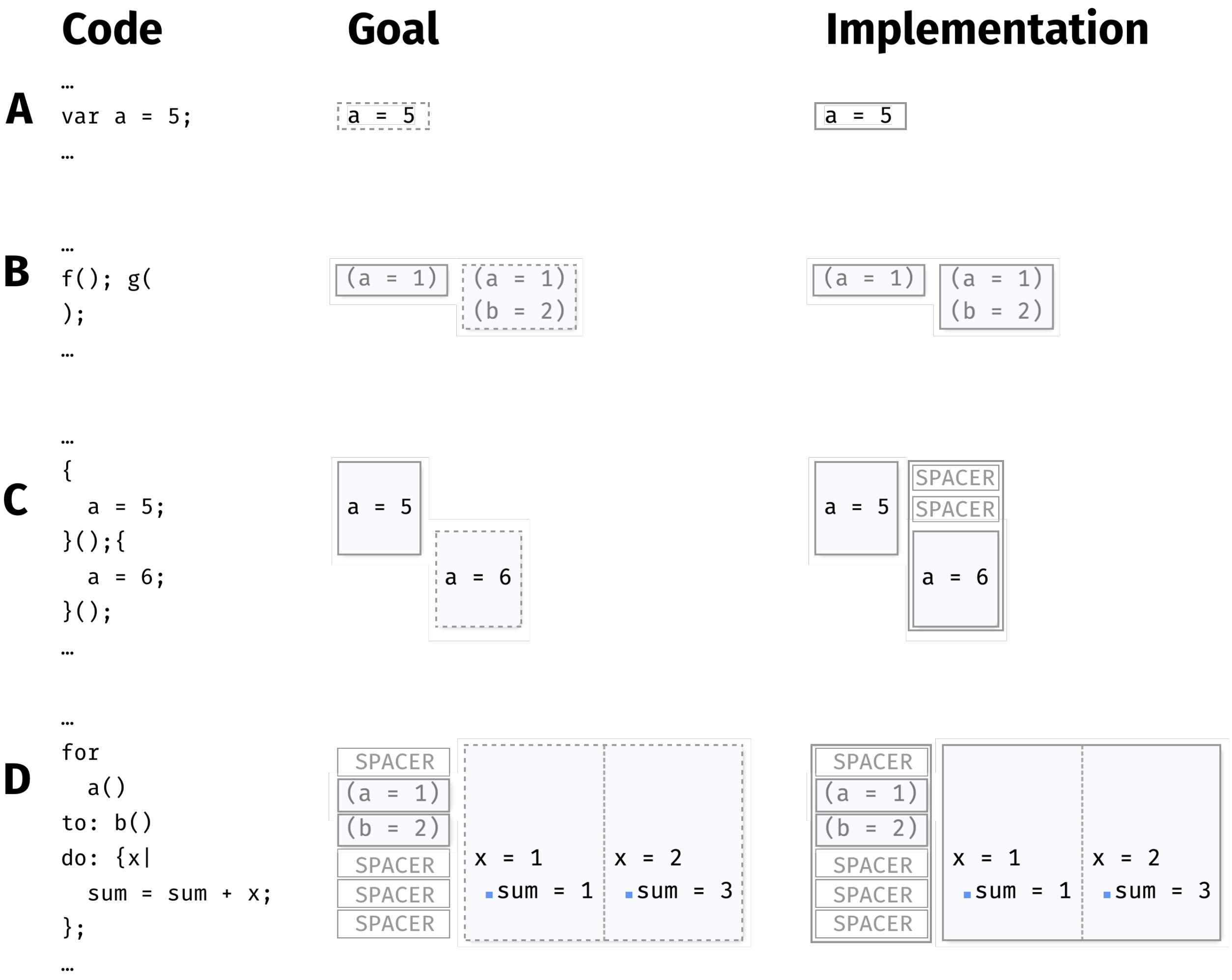


Figure 4.9: Adding spacers and wrappers to mirror code layout

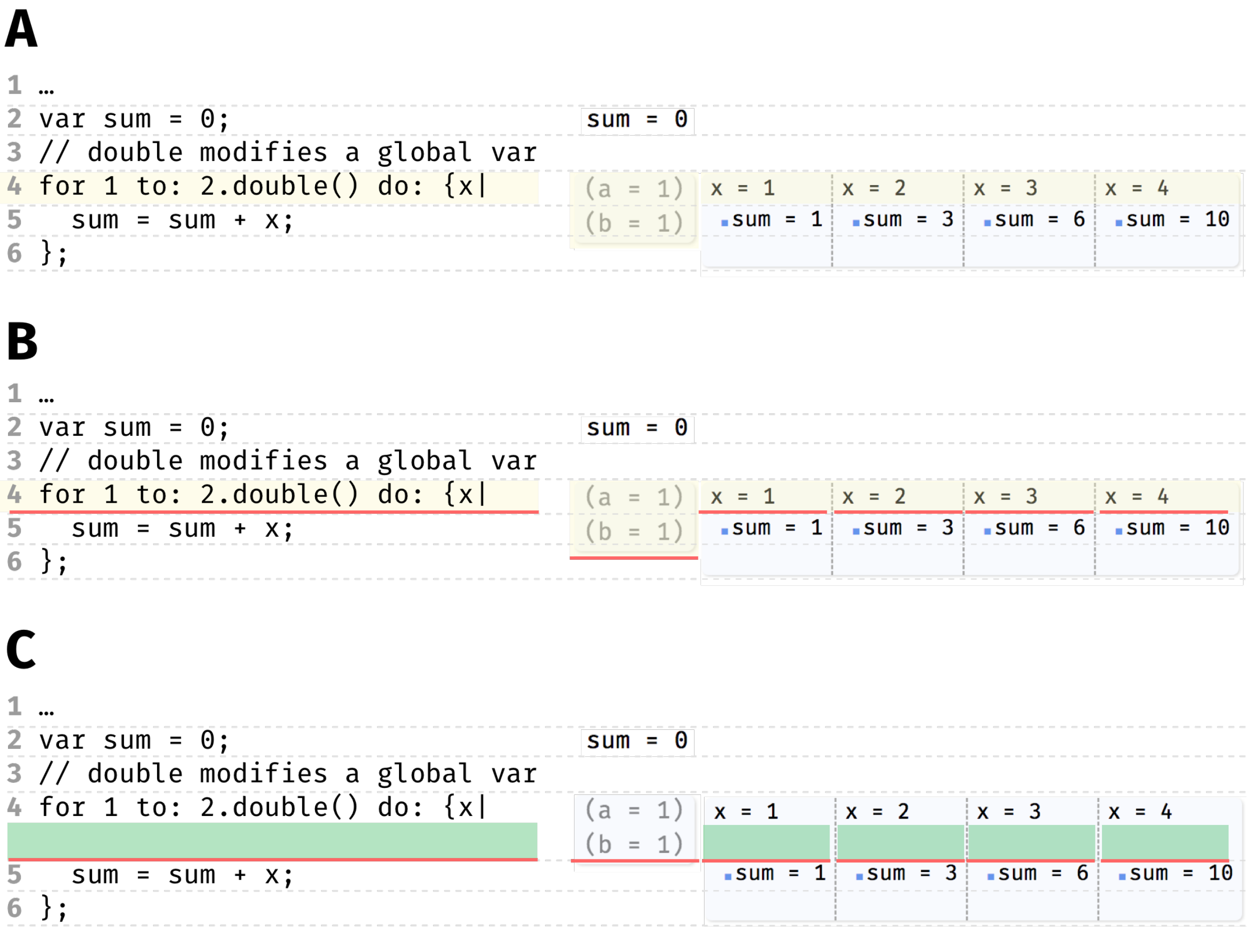
The ‘code’ column shows examples of code formatted in a manner that causes issues with a naïve layout. The ‘goal’ column shows the desired layout. Elements with solid borders have already been placed, while those with dashed borders are to be placed. The ’implementation’ column shows how the ideal layout is implemented. Wrappers are shown as an additional box around elements, and spacers are labeled as such.

Padding Elements on Each Line

Even after inserting spacers and wrappers, events may not be placed at the correct line. There may be multiple events for one column on one line. For example, in figure 4.10, the summary for ‘`2.double()`’ has 2 events on line 4. If no additional spacing is added, as in 4.10a, then the event ‘`(b = 1)`’ would be on the wrong line. 4.10c shows the micro visualization with padding added to make each column on line 4 the same height.

Adding these spacers is a global operation, as different columns may be in very different parts of the DOM structure. For example, ‘`(a = 1)`’ is part of a side effect summary and ‘`x = 1`’ is in an iteration for a different model. Also, in order to ensure that lines are padded correctly, even as the program is running, whenever an event is added to the micro visualization, we recompute the spacers for each line on which that event appears.

To pad a line *l*, as a precondition, the tops of all elements on *l* should line up, as in figure 4.10a. First, we collect all items on the line, including spacers, event summaries and lines of code. The elements for line 4 are shown in figure 4.10a. Next, the position of the bottom of each element is measured, in order to find the bottom of the lowest element on the line (figure 4.10b). Finally, each element on the line is padded so that its new bottom lines up with the lowest element bottom, as in figure 4.10c. After padding a line, all its elements’ bottoms will line up, guaranteeing the precondition for the next line.



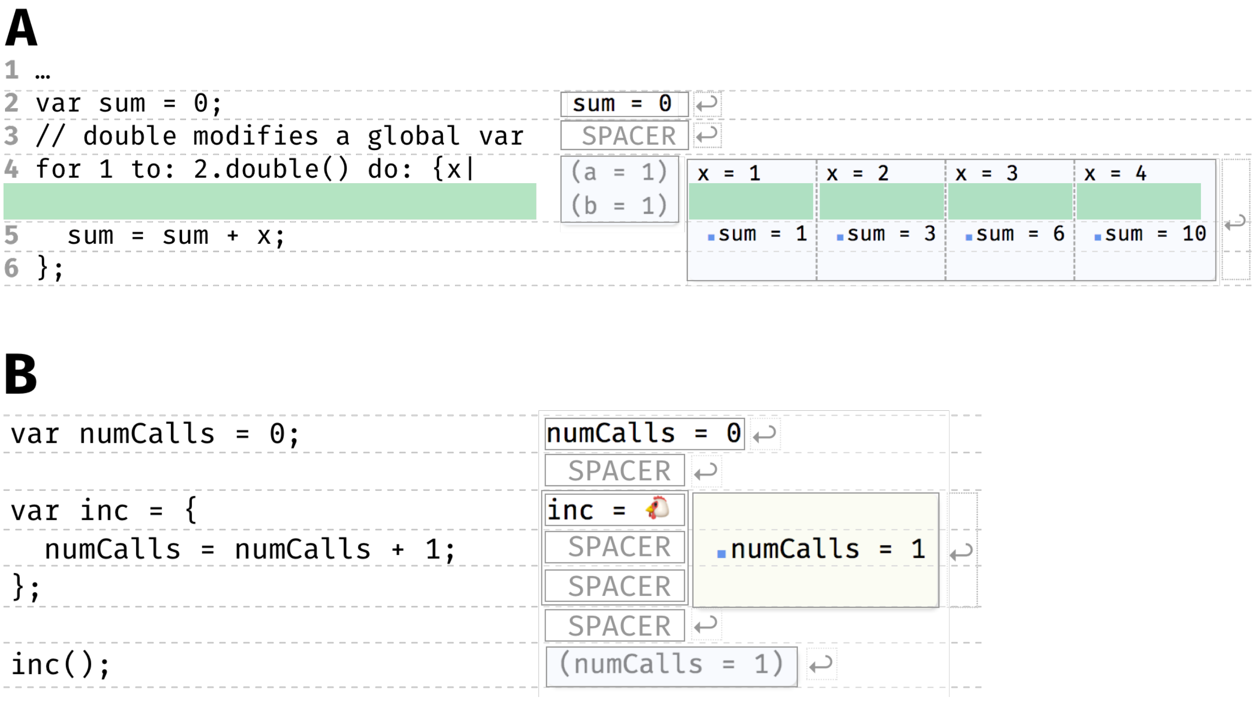
```
1 …
2 var sum = 0;                          sum = 0
3 // double modifies a global var
4 for 1 to: 2.double() do: {x|         (a = 1)  x = 1     x = 2     x = 3     x = 4
5   sum = sum + x;                      (b = 1)  sum = 1   sum = 3   sum = 6   sum = 10
6 };
```

```
1 …
2 var sum = 0;                          sum = 0
3 // double modifies a global var
4 for 1 to: 2.double() do: {x|         (a = 1)  x = 1     x = 2     x = 3     x = 4
5   sum = sum + x;                      (b = 1)  sum = 1   sum = 3   sum = 6   sum = 10
6 };
```

```
1 …
2 var sum = 0;                          sum = 0
3 // double modifies a global var
4 for 1 to: 2.double() do: {x|         (a = 1)  x = 1     x = 2     x = 3     x = 4
                                        (b = 1)
5   sum = sum + x;                               sum = 1   sum = 3   sum = 6   sum = 10
6 };
```

Figure 4.10: Padding elements on line 4

a) The elements on line 4 (along with the line in code) are highlighted. **b)** The bottom of each element is highlighted in red. **c)**Each element is padded (padding shown in green) so that the bottoms of all elements line up.



```
…
var sum = 0;
// double modifies a global var
for 1 to: 2.double() do: {x|

   sum = sum + x;
};
```

```
sum = 0
SPACER
(a = 1)
(b = 1)
x = 1    x = 2    x = 3    x = 4
sum = 1  sum = 3  sum = 6  sum = 10
```

```
var numCalls = 0;

var inc = {
   numCalls = numCalls + 1;
};

inc();
```

```
numCalls = 0
SPACER
inc = 🐔
SPACER       numCalls = 1
SPACER
SPACER
(numCalls = 1)
```

Figure 4.11: Figures 4.7 and 4.8 annotated with spacers, wrappers and padding

CHAPTER 5

Related Work

In "Inventing on Principle," [24] Victor introduced a novel visualization of program execution that has heavily influenced the design of Seymour's micro visualization. His visualization displays changes to program state over time with each change lined up with the line of code that produced it. Our micro visualization builds on Victor's visualization and extends it with *call summaries* that enable the programmer to see the side-effects of a call. In Victor's visualization, this information is only available via local expansion (or inlining) of the call, [26] and because this requires interaction, these side effects (e.g., assignments to global variables) are easy to miss.

Light Table [10] is an IDE that enables programmers to see dynamic (runtime) state inline with source code. It supports *watches* that can be attached to expressions in the code to reveal their latest value as the program is executed. However, if a statement or expression is evaluated many times (e.g., as part of the body of a function that is called more than once) Light Table does not let the programmer select the context or activation that is associated with a watch. YinYang [20] solves this problem with its *probes* that also enable the programmer to view the values of expressions. Probes are similar to Light Table's watches, but they are associated with an implicit execution context that must be activated through navigation or through a bookmark (created when a print statement is executed). By activating an execution context, the programmer can see the corresponding probe values. With Seymour, the programmer can see the details of any context by selecting it via the macro visualization. But unlike Light Table and YinYang, Seymour shows *all* of the side effects in the selected context automatically, minimizing interaction required to see program state.

DejaVu is a domain-specific IDE extension for interactive camera-based programs [16] that features a low-level canvas view that shows values of variables in a frame of interest, and a high-level timeline view that shows variable values for many frames. These views complement each other in a similar manner to the micro and macro visualizations in Seymour. But whereas DejaVu is designed specifically for the domain of camera-based programs and features visualizations tailored for displaying image data, Seymour's visualizations are designed to explain the semantics of generic programs. We discuss ways to integrate domain specific visualizations with Seymour in chapter 6.

Swift playgrounds [2] are an extension to Apple's Xcode IDE that add live programming functionality. They feature a 'timeline' visualization that plots the value of numerical variables over time. While Seymour does not currently show such plots, assignments to a variable can be used as raw data to render them, as discussed in chapter 6. Also, while a variable's timeline shows how it changes over time, it cannot be used to relate two different variables. Unlike timelines, which use a different time axis for each plot, Seymour shows changes to all variables that are relevant in a method call, allowing the user to see how different values are changed in relation to each other.

Theseus [18] is a live-programming extension to the Brackets editor. It features an always-on visualization which lets users see the relations between calls. An indicator next to each function definition shows the number of times it has been called. Clicking on an indicator shows all calls to that function in a visualization of a program trace. This visualization is similar to our macro visualization, as it nests callees below their callers. However, it only shows calls to functions that the user has chosen, filtering out irrelevant information. We discuss techniques to filter our macro visualization in a similar manner in chapter 6.

Online Python Tutor [12, 13], a visualization tool for teaching beginner programmers, features a visualization that shows the entire programs state at a single point in time. This includes the program counter, the stack, the heap, and relations between objects. Objects are drawn as boxes and are connected by arrows, as a professor might depict them in a classroom. This type of visualization is complementary to those provided by Seymour, and we believe programmers would benefit from seeing them together.

Omnicode [14] is a live IDE that also features an always-on visualization. This visualization shows the user a scatterplot matrix comparing the history of every variable to that of every other variable. The user can brush to select a region in a scatterplot and see the corresponding portions of all other scatterplots and code. Omnicode can also display a Python Tutor style visualization depicting program state at any line in the code. As with Python Tutor, these visualizations are complementary to those provided in Seymour. We are interested in letting the user add visualizations like these along with our default visualization where they are useful. We discuss how we can use our event stream as raw data to produce custom visualizations in chapter 6.

CHAPTER 6

Future Work

6.1 Classroom Trials

We envision Seymour being used as a tool to assist students and professors in an undergraduate intro to programming class. However, Seymour has not been user tested as of the time of this writing, and we have many improvements to make before it is ready to be used to teach a class. We will be showing our programming environment in classrooms in the fall in order to test it on students and adjust our designs based on their feedback. We have access to an intro to programming class and a programming languages class at UCLA, where we will be using Seymour to help students learn Python.

To support these trials, we have adapted Seymour to use Python as its underlying language. We have done this using a language-agnostic library for creating the micro and macro visualizations. With this library, adapting Seymour to a new language becomes a simple matter of connecting the language runtime to the library.

6.2 Better Support for Larger Programs

While Seymour's visualizations work well for small programs, they become harder to use as programs grow in size. With larger programs, such as those students write for a class project, we have found that the micro visualization shows too many low level details, while the macro visualization shows too few. We are interested in improving Seymour's visualizations to better explain such programs.

The macro visualization can provide a global view of small programs. However, as program size grows, it becomes too large to read and understand. We are interested in using techniques such as trace pruning [4], fisheye, and minimaps to help the user process medium and large programs.

We would also like to show more low level details in the macro visualization in order to help users better tie the macro and micro visualizations together, and to understand where to focus next. In a previous project (shown below), we visualized the execution of JavaScript methods and let the user annotate the visualization based on low level details (e.g. the value of a variable for a particular call).

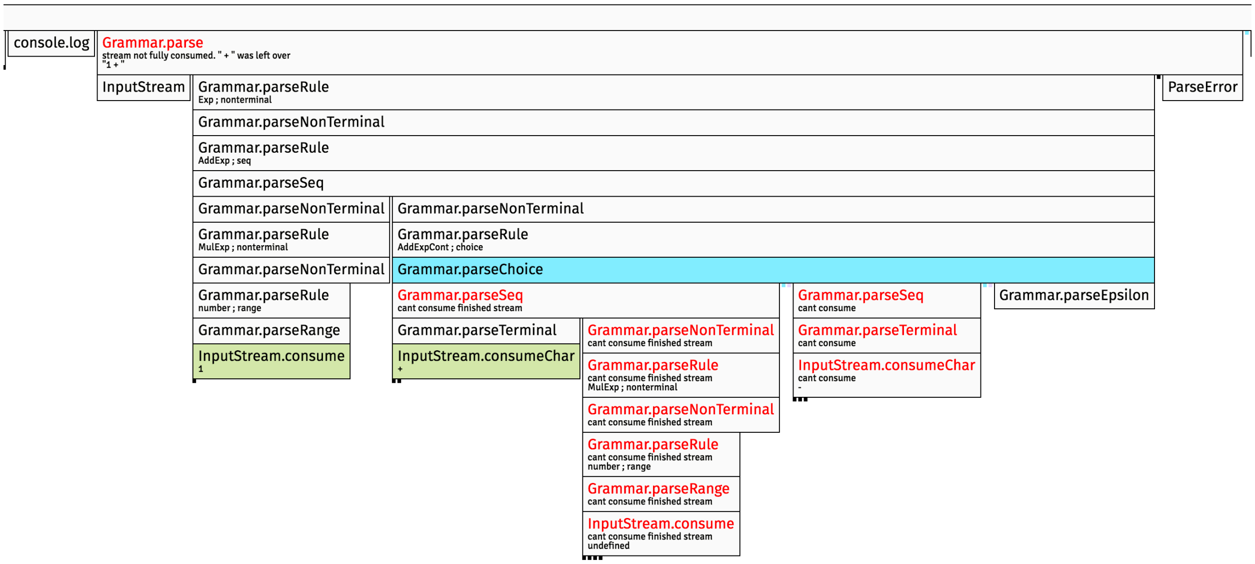
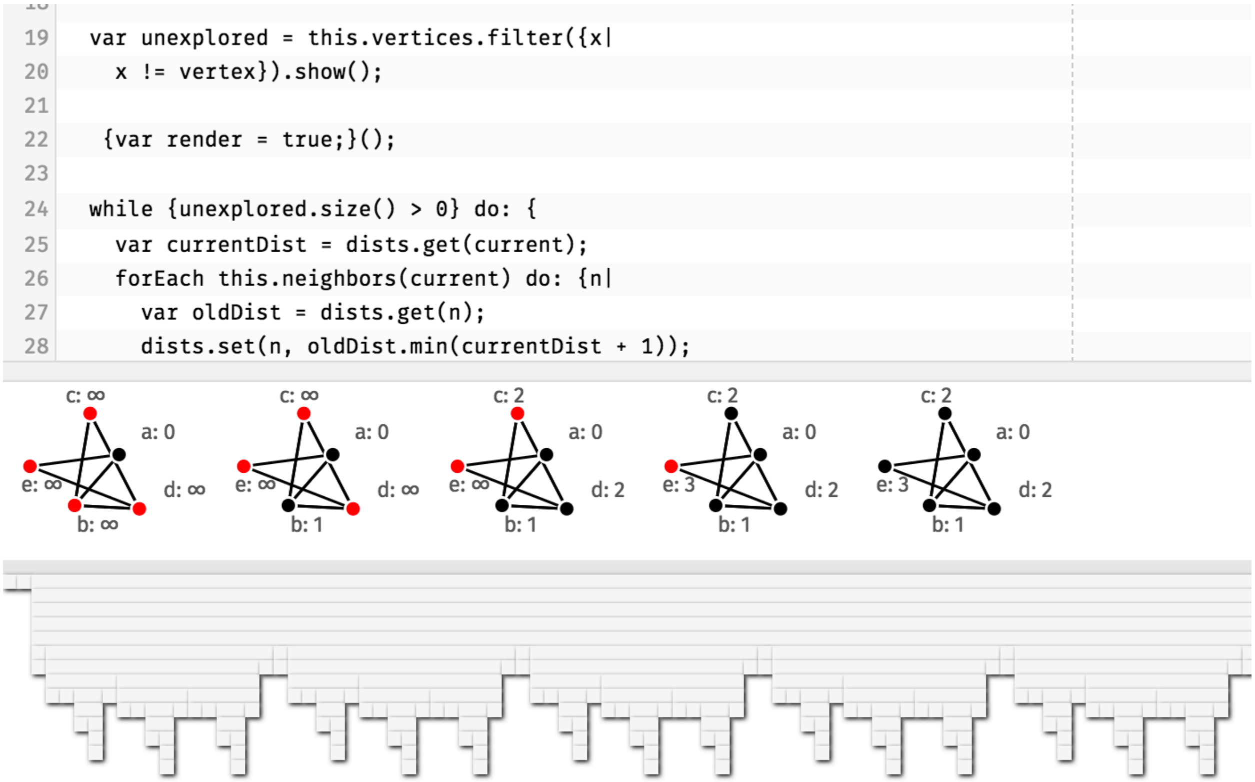


Figure 6.1: A prototype visualization of a parser generator's execution

In this example, which shows a parser generator, characters being consumed are shown in green, parse errors are shown with red text, and calls where the parser backtracked are shown in blue. To create an annotation, the user writes a query that selects the calls they want to annotate (e.g. calls that backtrack, calls that consume a character), and modifies their nodes appearance in the visualization. These modifications let the user read the macro visualization for high-level, program-specific patterns, instead of having to interact to find the information they are looking for.

Another solution we are exploring is to make it easy to create domain specific visualizations from Seymour's run-time data. These visualizations can display program information at the right level of abstraction to help the user understand the program. Instructors can use such a system to great effect as visualizations can be reused as part of course materials and can be shared with students as explanatory tools. Having computer-generated visualizations tied into the system also eases the tedious and error-prone process of illustrating algorithms on the blackboard.

Below is a screenshot of an initial attempt at visualizing Dijkstras algorithm using information from a running program. We use a timeline to show how a graph is traversed as the algorithm progresses. Red nodes are nodes that have not yet been traversed, and each nodes distance from node a is shown next to the nodes label. This visualization is live and updates as the user changes the code.



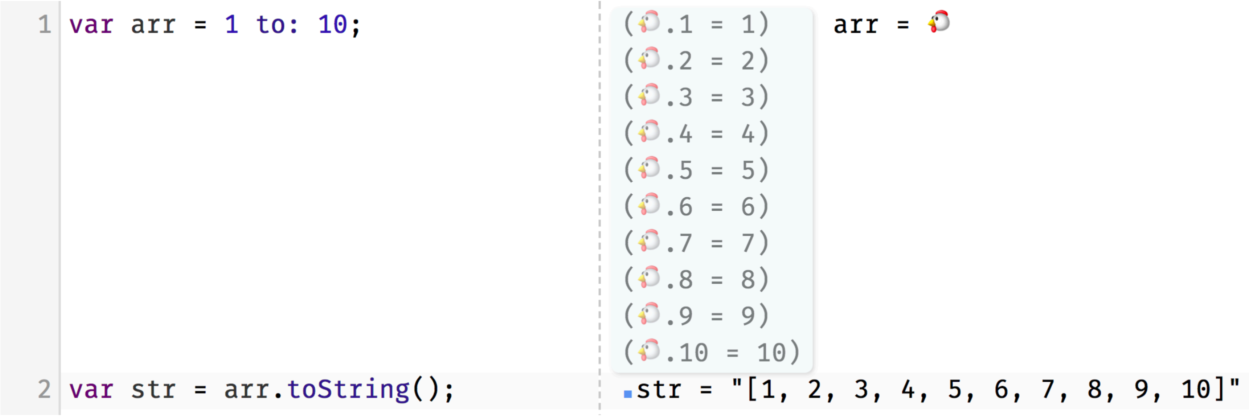
```
var unexplored = this.vertices.filter({x|
  x != vertex}).show();

 {var render = true;}();

while {unexplored.size() > 0} do: {
  var currentDist = dists.get(current);
  forEach this.neighbors(current) do: {n|
    var oldDist = dists.get(n);
    dists.set(n, oldDist.min(currentDist + 1));
```

Figure 6.2: A visualization of the graph and frontier for Dijkstra's algorithm

6.3 Language Features For Improved User Understanding

Seymour's user experience can be further improved by adding features to the programming language underlying the system. We are exploring several language extensions that could help the user better understand their programs.

Micro visualization summaries can become large if the method call being summarized has many side effects. The example below creates an array with all elements from 1 to 10, showing 10 side effects on line 1.



```
1 var arr = 1 to: 10;          (🐔.1 = 1)     arr = 🐔
                                (🐔.2 = 2)
                                (🐔.3 = 3)
                                (🐔.4 = 4)
                                (🐔.5 = 5)
                                (🐔.6 = 6)
                                (🐔.7 = 7)
                                (🐔.8 = 8)
                                (🐔.9 = 9)
                                (🐔.10 = 10)
2 var str = arr.toString();     str = "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]"
```

Figure 6.3: A function call with 10 side effects can be difficult to understand

We have begun experimenting with *modular side effects*, a language feature that allows an object to specify how its side effects should be presented to the user. As illustrated above, the micro visualization currently shows all side effects when working with an object. For example, when inserting an element into a red-black tree, the user sees all the rotations the tree conducts to maintain its invariant. With modular side effects, an object can choose how to present its internal state. A red-black tree might present itself as a set, only showing that an element has been inserted, sparing the user an excess of irrelevant details.

Together with our colleague Jonathan Edwards, we have also started exploring *interventions* a mechanism that lets programmers temporarily modify program execution to aid development. In addition to displaying the effects of a call, call summaries can be used to change the program's execution. If a method call produces an incorrect answer, the user can *intervene* and assert that the method call return the correct value. This assertion can then become a unit test, and its value can be substituted for the calls return value, allowing the user to make progress on other parts of the program and address the broken method at a later time.

CHAPTER 7

Conclusion

Seymour is a live programming environment that visualizes program execution as the user types. It features a micro visualization that shows details of the programs execution, and a macro visualization that puts the micro visualization in context, letting the user focus on different parts of the program execution. These visualizations come together to give the user a helpful live programming experience.

We are excited about the prospect of using this environment in the classroom and learning from student feedback. In doing so, we hope to make Seymour a valuable learning aid for students, and ultimately, create a better user experience for all programmers.

REFERENCES

[1] John Allen. 1978. *Anatomy of Lisp*. McGraw Hill, Inc., New York, NY, USA.

[2] Rick Ballard and Connor Wakamo. 2014. Swift Playgrounds. Retrieved from `https://developer.apple.com/videos/play/wwdc2014/408/`

[3] Kayce Basques. Performance Analysis Reference. Retrieved from `https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference`

[4] Johannes Bohnet. 2010. *Visualization of Execution Traces and its Application to Software Maintenance*. PhD Dissertation. Mathematisch-Naturwissenschaftliche Fakultät / Institut für Informatik, Potsdam. Retrieved from `https://publishup.uni-potsdam.de/opus4-ubp/frontdoor/index/index/docId/32254`

[5] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. ACM Comput. Surv. 41, 1, Article 2 (January 2009), 31 pages. DOI: `http://dx.doi.org/10.1145/1456650.1456652`

[6] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*. Vol. 2, Issue 1, pp. 57–73. DOI: `https://dx.doi.org/10.2190/3LFX-9RRF-67T8-UVK9`

[7] Patrick Dubroy and Alessandro Warth. 2017. Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (SLE 2017). 14–25. DOI: `https://doi.org/10.1145/3136014.3136022`

[8] Jonathan Edwards. 2004. Example Centric Programming. SIGPLAN Not. 39, 12 (December 2004), 84-91. DOI:`https://dx.doi.org/10.1145/1052883.1052894`

[9] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[10] Chris Granger. Light Table. Retrieved from `http://lighttable.com/`

[11] Brendan Gregg. Flame Graphs. Retrieved from `http://www.brendangregg.com/flamegraphs.html`

[12] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (SIGCSE 13), 579–584. DOI: `https://dx.doi.org/10.1145/2445196.2445368`

[13] Philip J. Guo. Live Programming Mode - Python Tutor. Retrieved from `http://pythontutor.com/live.html#mode=edit`

[14] Hyeounsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (UIST 17), 737–745. DOI: `https://dx.doi.org/10.1145/3126594.3126632`

[15] Saketh Kasibatla and Alex Warth. 2017. Seymour: Live Programming for the Classroom. In *Workshop on Live Programming Systems* (LIVE 2017). `https://harc.github.io/seymour-live2017/`

[16] Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: Integrated Support for Developing Interactive Camera-based Programs. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (UIST 12), 189–186. DOI: `https://dx.doi.org/10.1145/2380116.2380142`

[17] J. B. Kruskal and J. M. Landwehr. 1983. Icicle Plots: Better Displays for Hierarchical Clustering. *The American Statistician* 37, 2 (1983), 162–168. DOI: `https://dx.doi.org/10.2307/2685881`

[18] Tom Lieber, Joel Brandt, and Robert C. Miller. 2014. Addressing Misconceptions About Code with Always-On Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '14), 2481–2490. DOI: `https://dx.doi.org/10.1145/2556288.2557409`

[19] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. 1978. Characteristics of application software maintenance. Commun. ACM 21, 6 (June 1978), 466–471. DOI: http://dx.doi.org/10.1145/359511.359522

[20] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2013), 5362. DOI: `https://dx.doi.org/10.1145/2509578.2509585`

[21] Sean McDirmid and Jonathan Edwards. 2014. Programming with Managed Time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Onward! 2014), 110. DOI: `https://dx.doi.org/10.1145/10.1145/2661136.2661145`

[22] Juha Sorva. 2012. *Visual program simulation in introductory programming education*. PhD Dissertation. Aalto University. Retrieved from DOI: `https://aaltodoc.aalto.fi:443/handle/123456789/3534`

[23] Juha Sorva. 2013. Notional machines and introductory programming education. Trans. Comput. Educ. 13, 2, Article 8 (July 2013), 31 pages. DOI: `http://dx.doi.org/10.1145/2483710.2483713`

[24] Bret Victor. 2012. Inventing on Principle. Retrieved from `https://vimeo.com/36579366`

[25] Bret Victor. 2012. Learnable Programming. Retrieved from `http://worrydream.com/LearnableProgramming/`

[26] Bret Victor. 2013. Showreel 2011–2012. Retrieved from `https://vimeo.com/62049081#t=1m41s`

[27] Michelle Q. Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. 2000. Guidelines for Using Multiple Views in Information Visualization. In *Proceedings of the Working Conference on Advanced Visual Interfaces* (AVI 00), 110–119. DOI: `https://dx.doi.org/10.1145/345513.345271`

[28] Timeline Profiling with Chrome DevTools. *Librato Blog*. Retrieved from http://blog.librato.com/posts/chrome-devtools