

The Magnificent Maniposynth

Bimodal Tangible Functional Programming

The image shows a screenshot of the Maniposynth IDE. On the left, a code editor displays OCaml code for calculating the length of a list. On the right, a visual execution environment shows the state of the program, including variable bindings and function calls.

```
int list
1 let int_list = [ 0; 0; 0 ] [@@pos 69, 72]
2
'a list -> int
3 let rec length list =
4   match list with
5   | hd :: tail ->
6     let length2 = length tail [@@pos 55, 12] in
7     1 + length2
8   | [] -> 0
9   [@@pos 77, 200]
10
int
11 let length_int = length int_list [@@pos 276, 76]
12
```

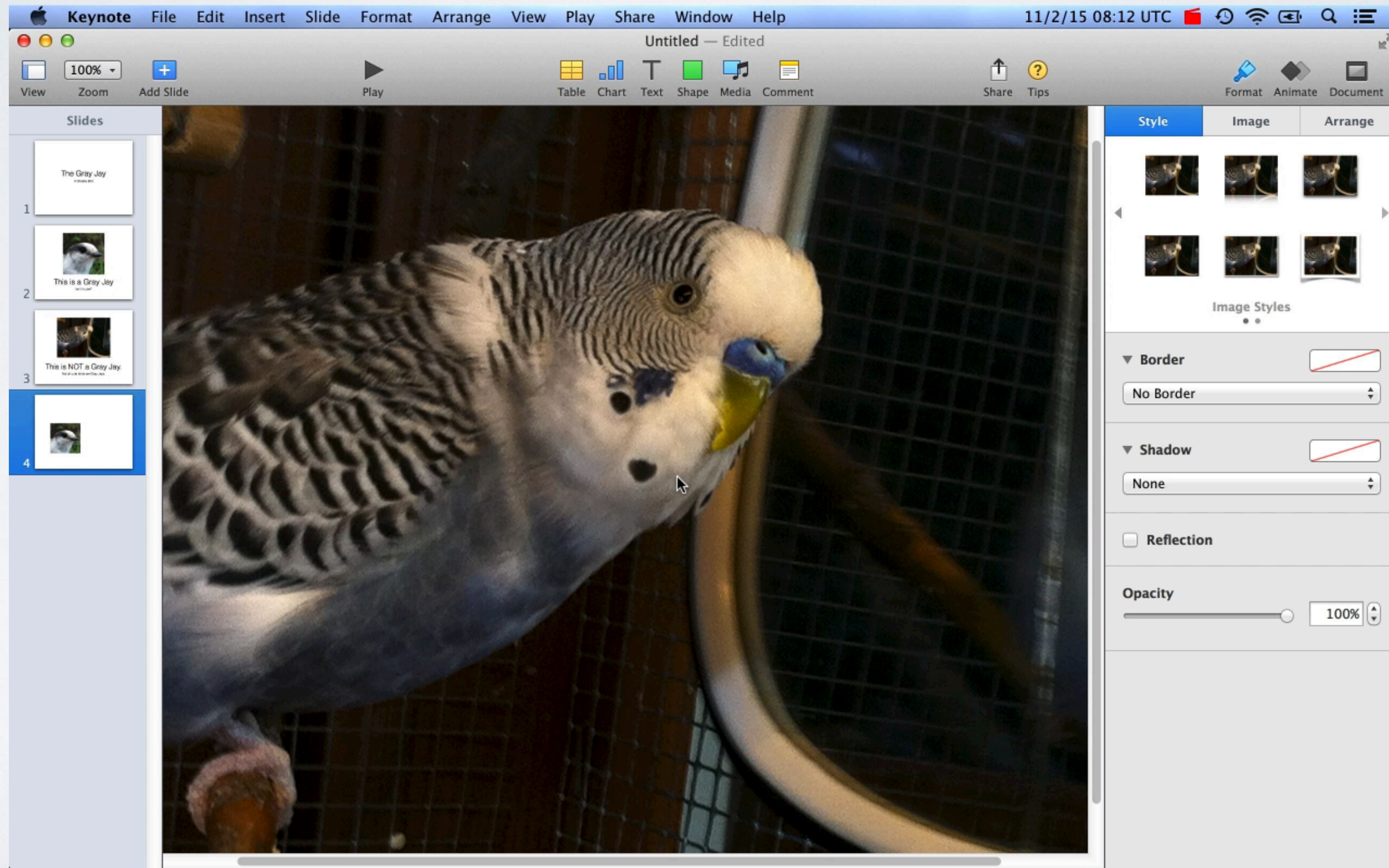
The visual execution environment on the right shows the following state:

- Top level: `int_list = [0; 0; 0]` and `length_int = length [0; 0; 0] int_list` with value `3`.
- Function `length` (recursive):
 - Return: `3`, `2`, `1`, `0` (corresponding to list elements `hd` and `tail`).
 - Bindings inside function: `hd = 0`, `tail = []`, `length2 = length [] tail` with value `0`.
 - Return expression(s) and value(s): `1 + length2` with value `1`, and `0`.

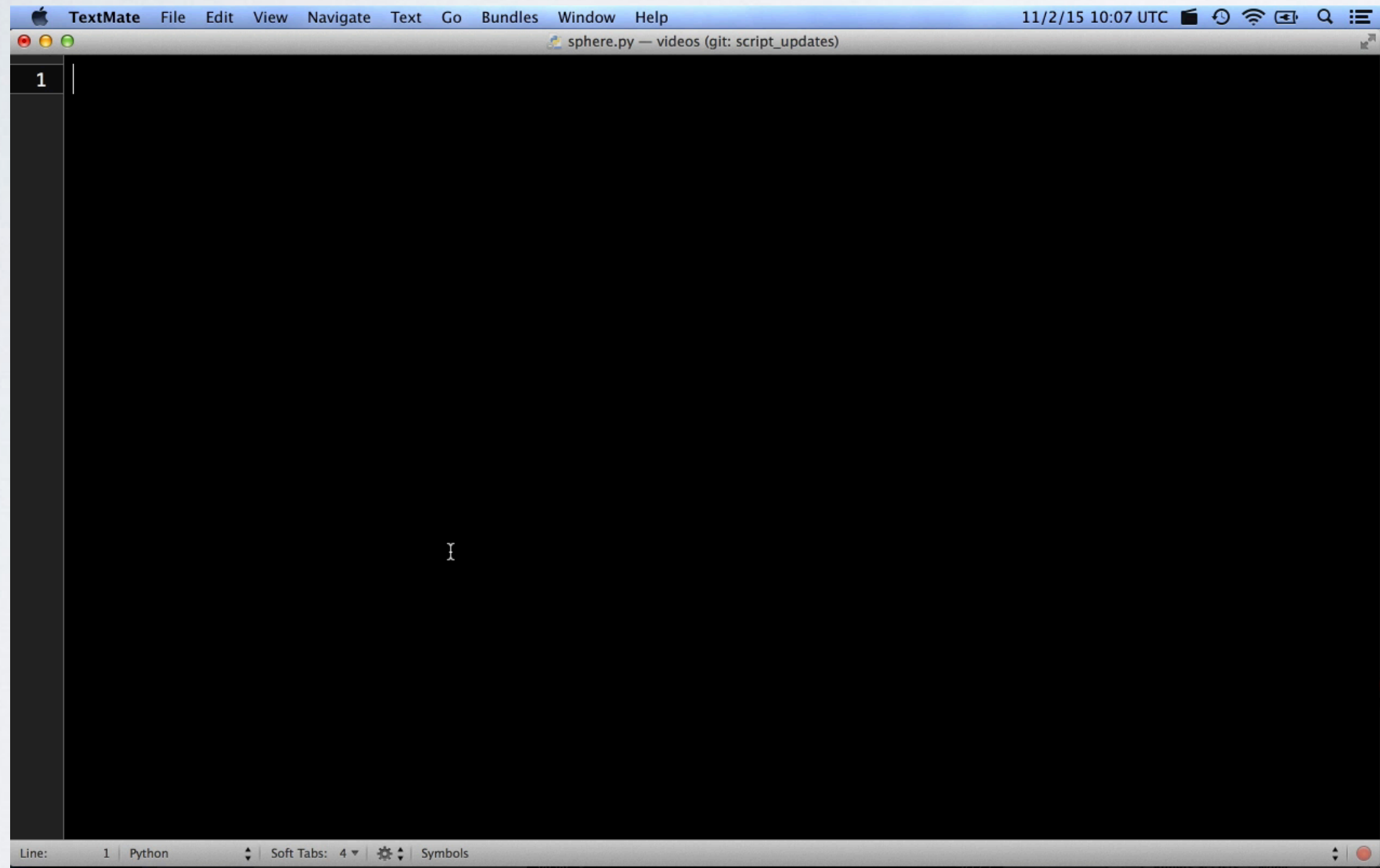
A `Synth` button is visible in the bottom right corner of the IDE.

Brian Hempel and Ravi Chugh

Direct Manipulation UI

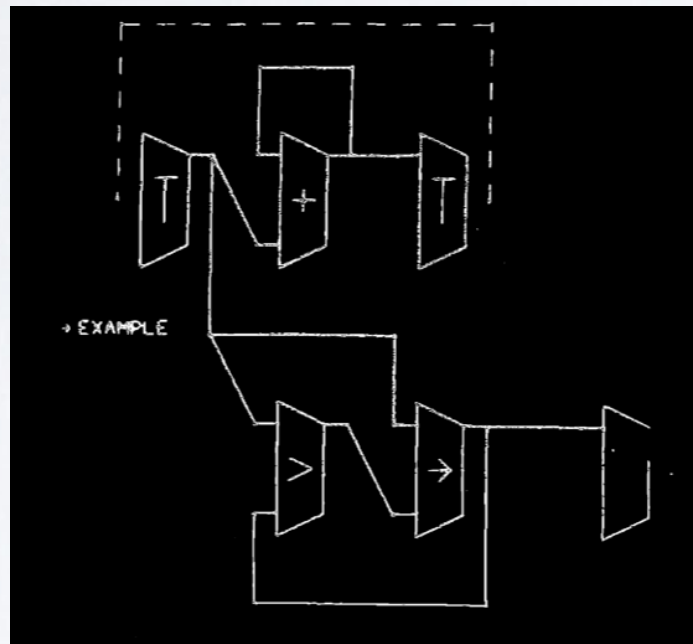


Why not D.M. for programming?



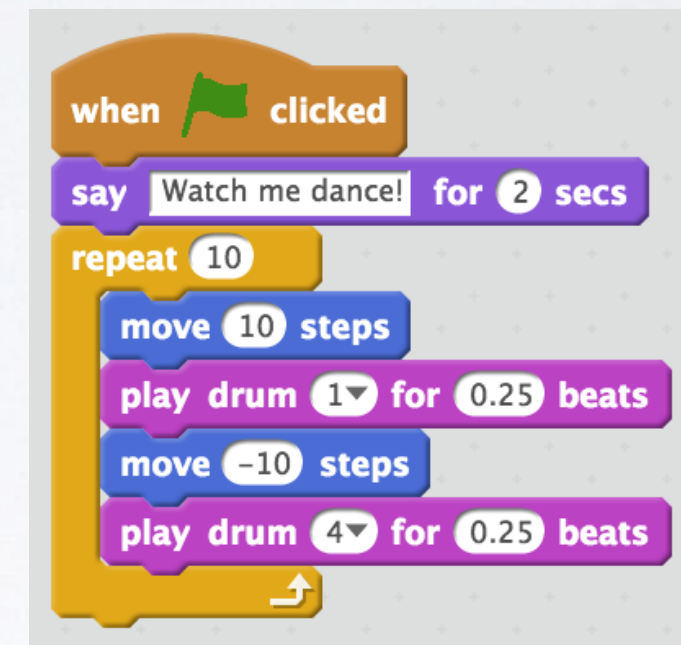
D.M. the AST...

Nodes-and-wires



The On-line Graphical Specification
of Computer Procedures
W. Sutherland (1966)

Blocks



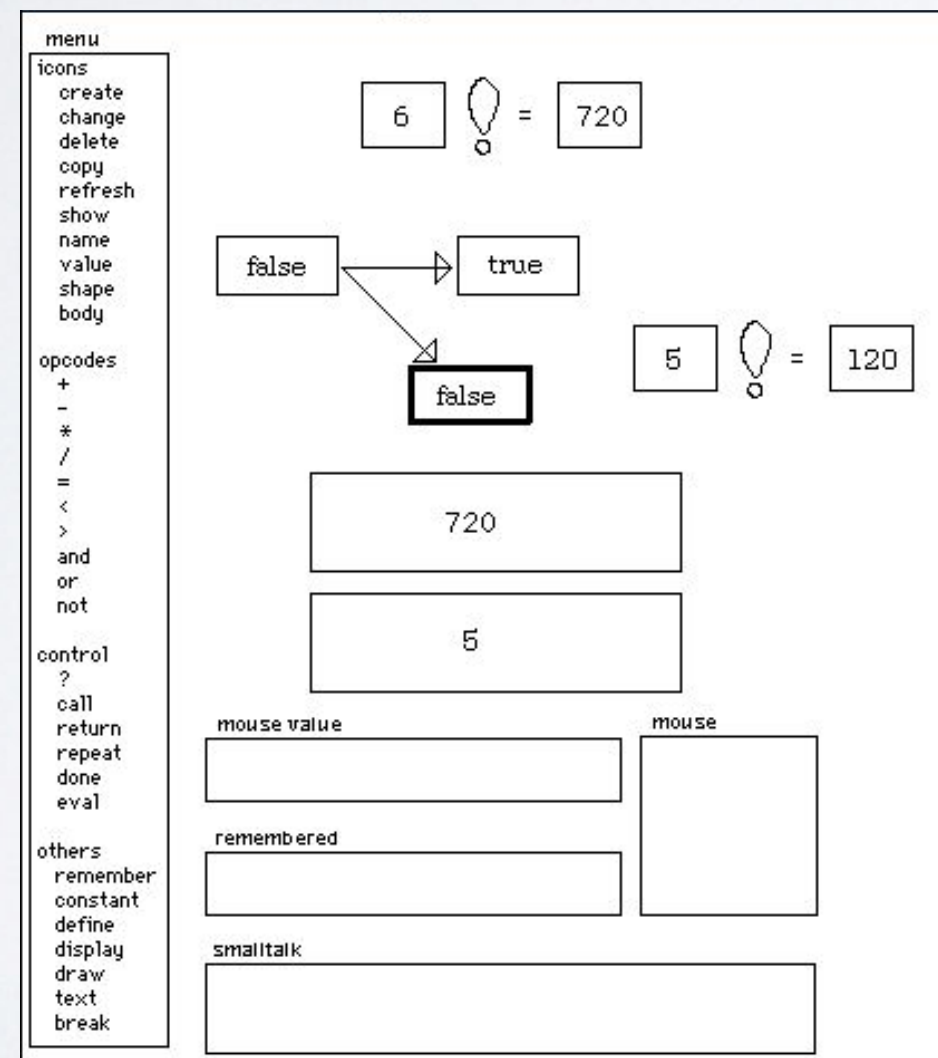
Scratch
Resnick et al. (2009)

...but those are **expressions** not output **values**

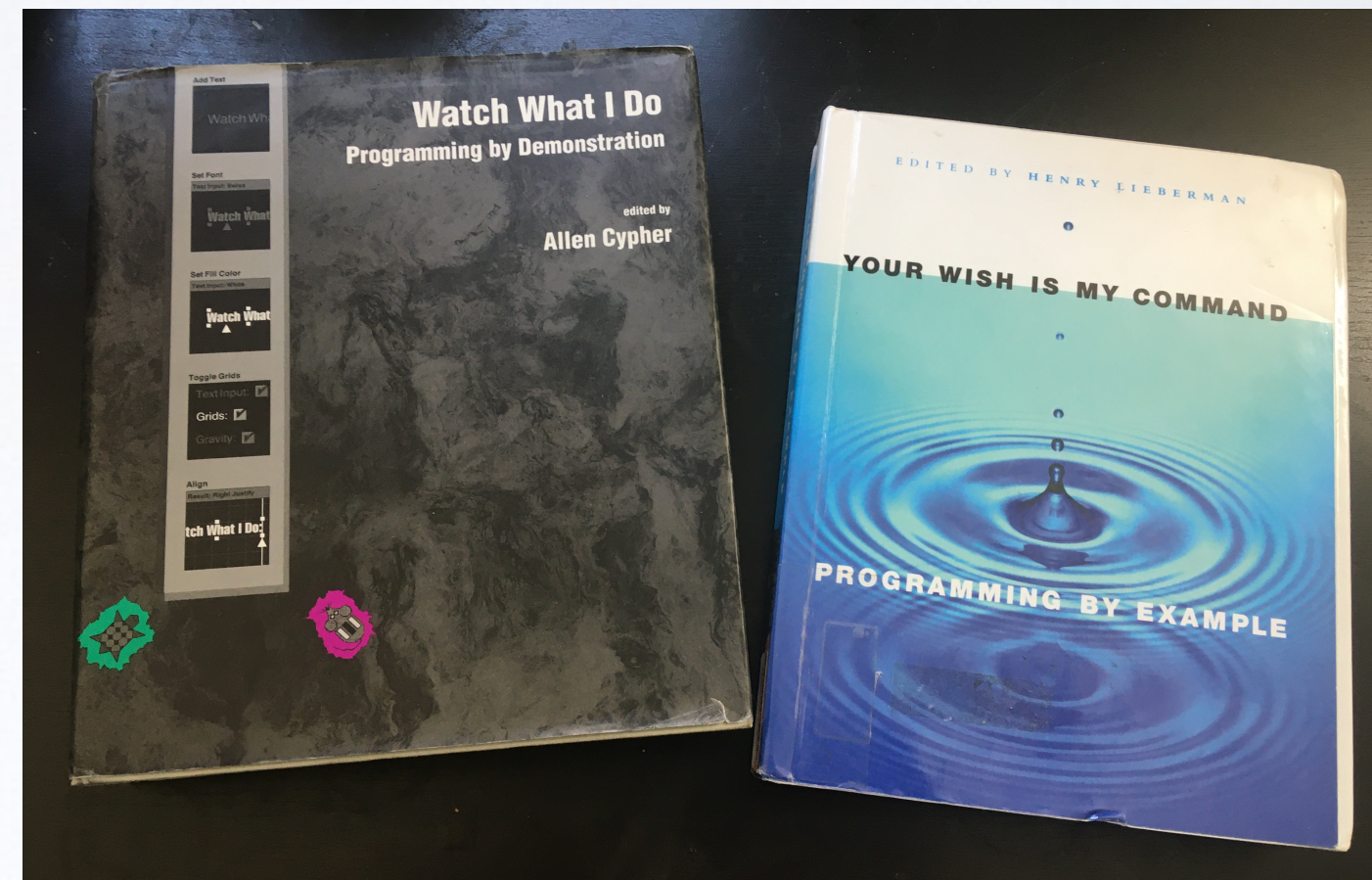
D.M. the values...

Programming by Demonstration (PBD)

You give a step-by-step demonstration of what you want the computer to do.

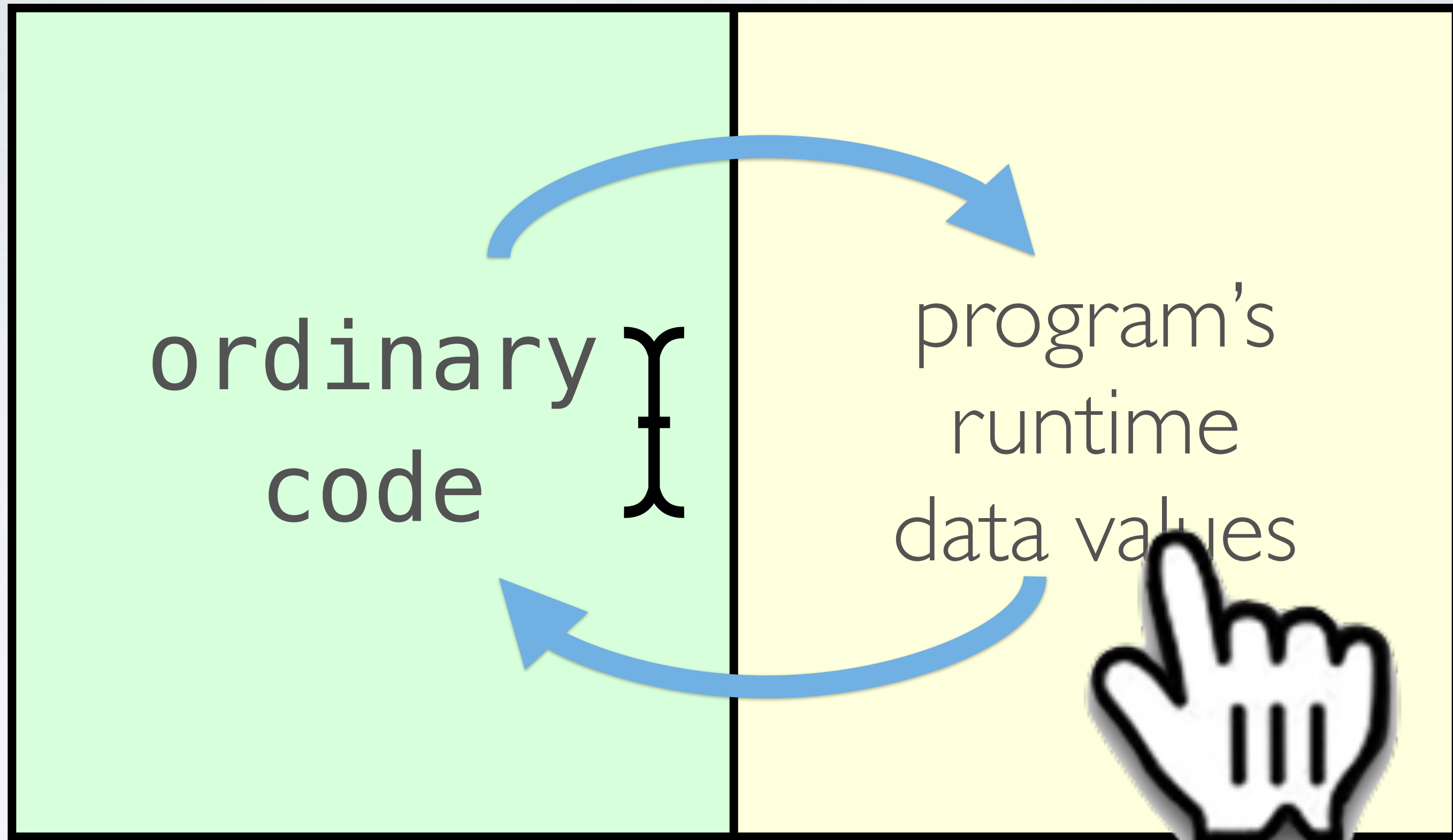


Pygmalion
Smith (1975)



Have you ever used PBD?

- Domain-specific
- Rarely textual code



Bimodal Programming

ALVIS Live!

Hundhausen & Brown (2007)

The screenshot displays the ALVIS Live! software interface. At the top, there is an "Execution Speed" slider and "Execution Controls" buttons. The "Script Editor" contains the following code:

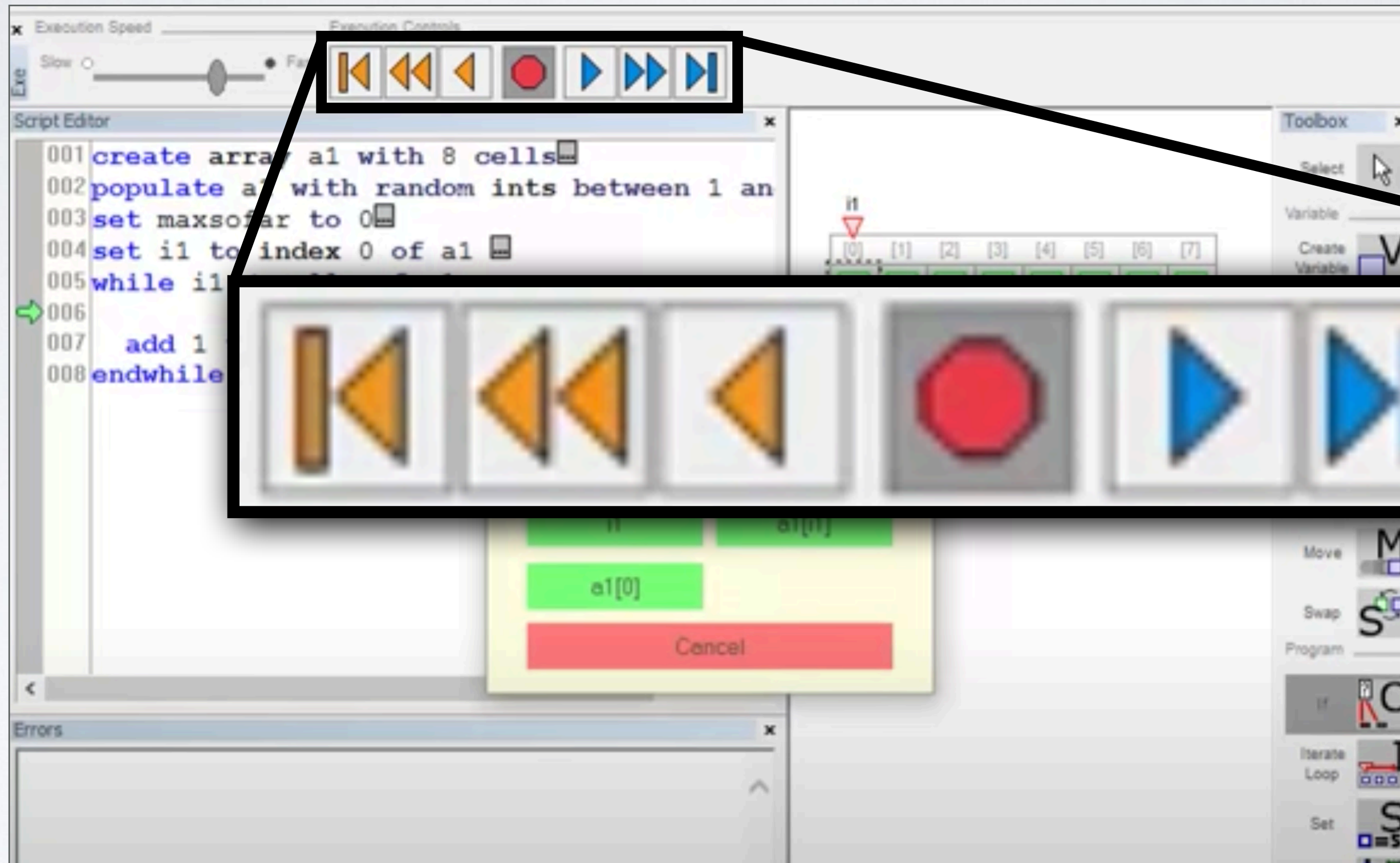
```
001 create array a1 with 8 cells
002 populate a1 with random ints between 1 an
003 set maxsofar to 0
004 set i1 to index 0 of a1
005 while i1 < cells of a1
006     add 1 to i1
007 endwhile
```

Below the script editor, a variable viewer shows an array `a1` with 8 cells containing the values [35, 74, 66, 98, 65, 82, 70, 83]. A red triangle points to the first cell, and a green box highlights the value 35. A variable `maxsofar` is shown with the value 0. A dialog box titled "Array" is open, displaying the text "if x?y Please clarify what x is" and two green buttons labeled "i1" and "a1[1]". A red "Cancel" button is at the bottom of the dialog. The "Toolbox" on the right contains various icons for creating variables, arrays, and performing operations like "Populate", "Move", "Swap", "Program", "Iterate Loop", and "Set".

Array algorithms (for education)

ALVIS Live!

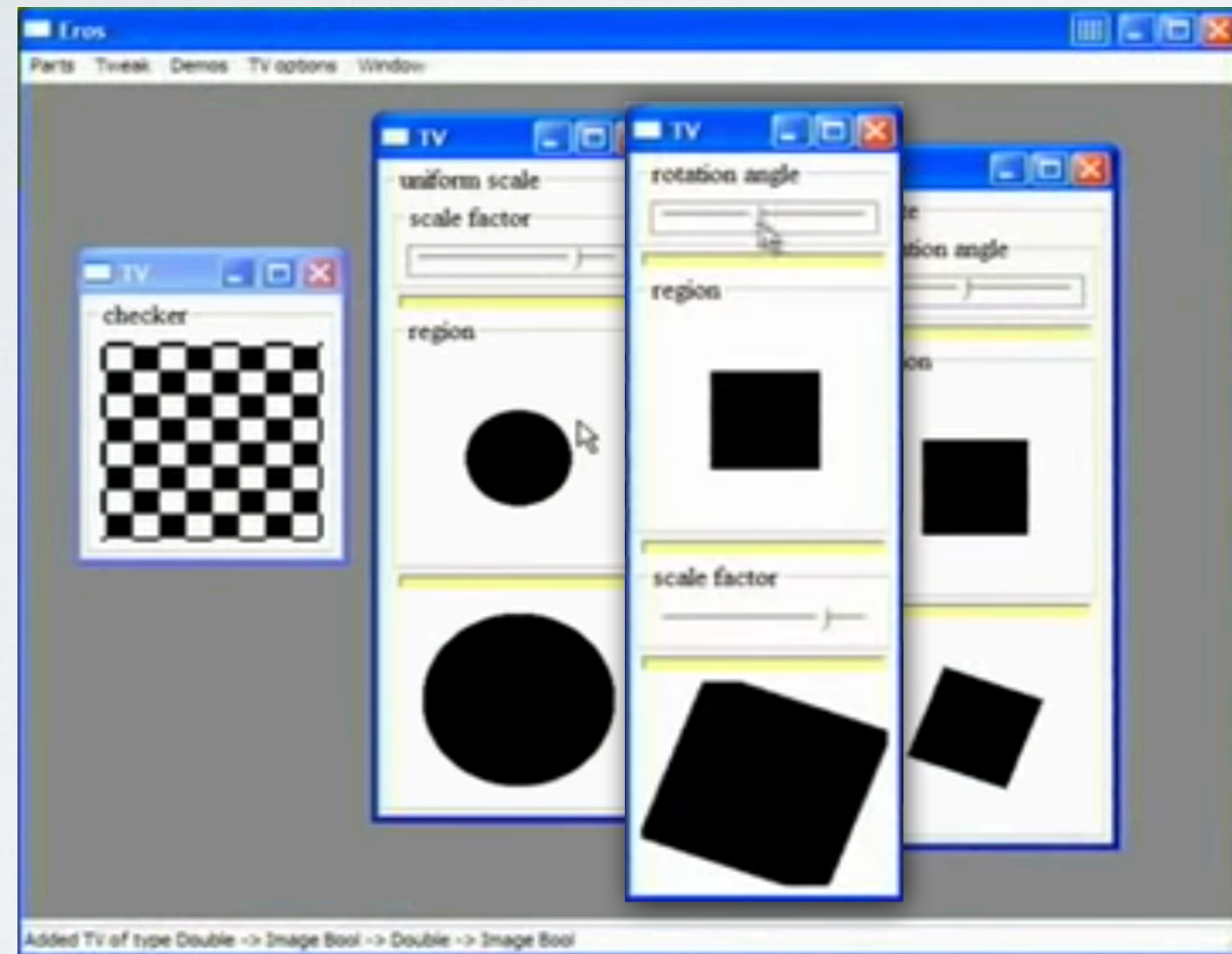
Hundhausen & Brown (2007)



Array algorithms (for education)
Very **linear**: have to manage time.

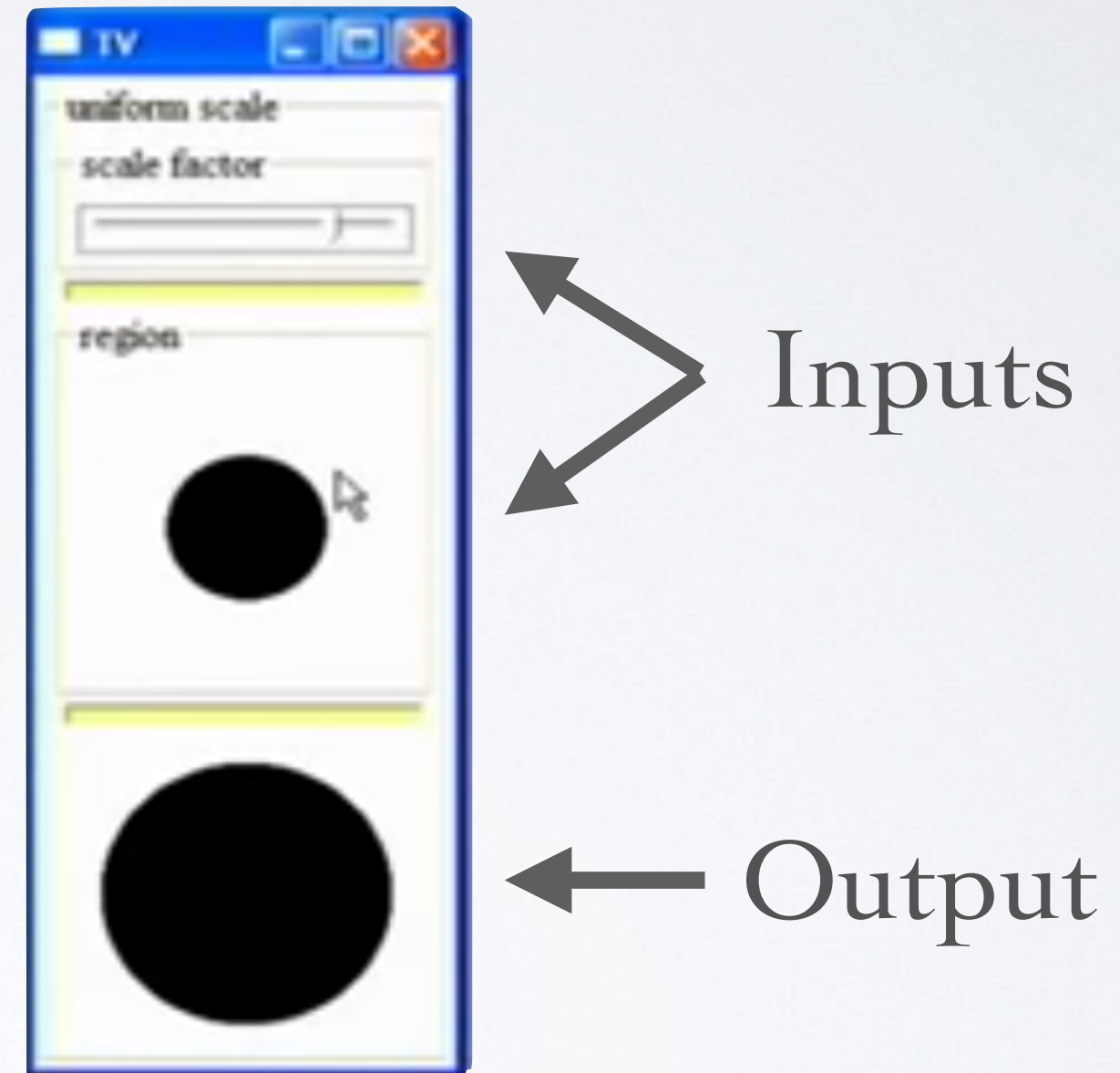
Tangible Functional Programming

Conal Elliott (2007)



Canvas of tangible values

Tangible Value (TV)



Non-linear

Pure functional programming (no state) complements **non-linear** editing because, without state, one need not manage **time**.

Goals: Non-linear + Bimodal + Synthesis

The image shows a side-by-side view of the Maniposynth IDE. On the left is a code editor with the following OCaml code:

```
1 let int_list = [ 0; 0; 0 ] [@@pos 69, 72]
2
3 'a list -> int
4 let rec length list =
5   match list with
6   | hd :: tail ->
7     let length2 = length tail [@@pos 55, 12] in
8     1 + length2
9   | [] -> 0
10  [@@pos 77, 200]
11
12 int
13 let length_int = length int_list [@@pos 276, 76]
```

On the right is the visual manipulation interface. It features a top menu with 'Undo (⌘Z)', 'Redo (⇧⌘Z)', and a dropdown menu for 'length(?)'. Below the menu, there are two main sections:

- Top level:** Contains two draggable blocks. The first is `int_list = [0; 0; 0]` with a value of `[0; 0; 0]`. The second is `length_int = length [0; 0; 0] int_list` with a value of `3`.
- Function level:** A table for the `length` function. The table has columns for `list` and `Return`. The `list` column shows patterns: `[hd tail; 0; 0]`, `[hd tail; 0]`, `[hd tail]`, and `[]`. The `Return` column shows values: `3`, `2`, `1`, and `0`. Below the table, there are blocks for `hd` (value `0`), `tail` (value `[]`), and `length2 = length [] tail` (value `0`).

At the bottom right of the interface is a button labeled `Synth (⌘Y)`. The status bar at the bottom of the IDE shows 'master*', 'opam(4.07.1)', 'Spaces: 2', 'UTF-8', 'LF', and 'OCaml'.

Direct Manipulation + Synthesis = *The Magnificent Maniposynth*

The Magnificent *Maniposynth*

- **Goals**
- Demo
- Implementation
- Evaluation
- Future Work & Conclusion

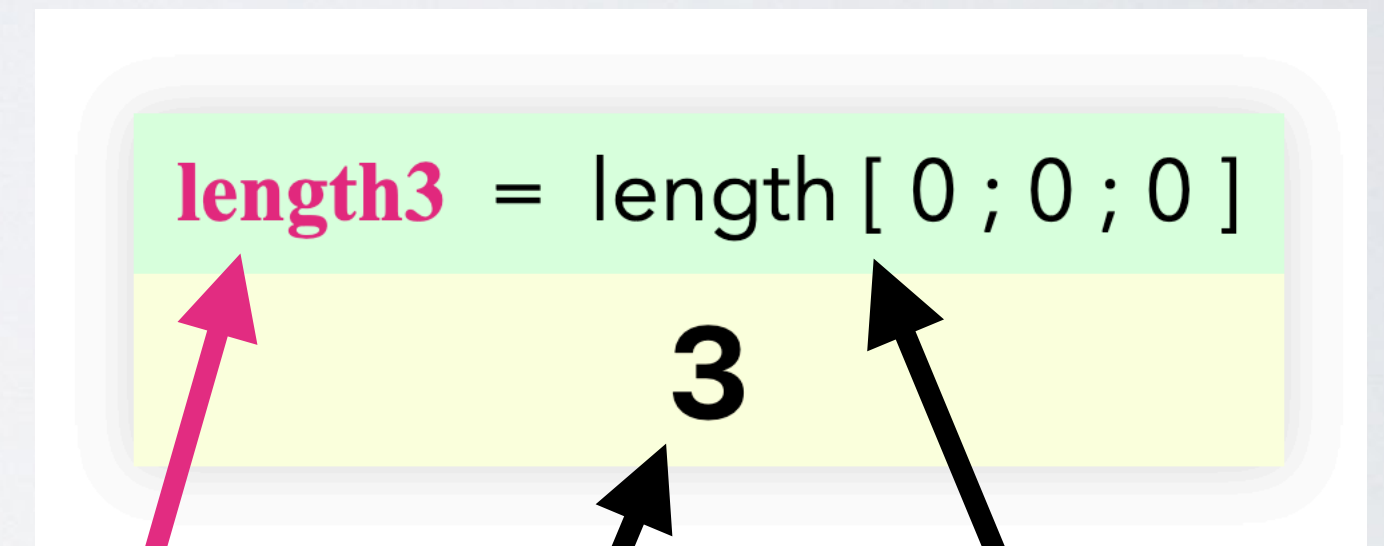
The Magnificent **Maniposynth**

- Goals
- **Demo**
- Implementation
- Evaluation
- Future Work & Conclusion

Tangible Values in Maniposynth

1 let-binding = 1 TV (Roughly)

```
4 |  
  | int  
5 | let length3 = length [ 0; 0; 0 ] [@@pos 405, 175]  
6 |
```



Pattern

Expression

Value

Demo: List Length

```

length.ml — maniposynth
length.ml 1, U x
int list
1 let int_list = [ 0; 0; 0 ] [@@pos 69, 72]
2
'a list -> int
3 let rec length list =
4   match list with
5   | hd :: tail ->
6     let length2 = length tail [@@pos 55, 12] in
7     1 + length2
8   | [] -> 0
9   [@@pos 77, 200]
int
11 let length_int = length int_list [@@pos 276, 76]
12

```

Maniposynth

localhost:1111/length.ml

The Magnificent Maniposynth

Undo (⌘Z) Redo (⇧⌘Z) if (??) then (??) else (??) length (??)

Top level - drag items from the menus above, or double-click below to write code

int_list = [0; 0; 0]

[0; 0; 0]

length_int = length ^[0; 0; 0] int_list

3

length rec

| list | [^{hd} 0 ^{tail} ; 0; 0] | [^{hd} 0 ^{tail} ; 0] | [^{hd} 0 ^{tail}] | [] |
|--------|--|---|-------------------------------------|-----|
| Return | 3 | 2 | 1 | 0 |

Bindings inside function - drag what you want below, or double-click to write code

hd

0

tail

[]

length2 = length ^[] tail

0

Return expression(s) and value(s)

1 + ⁰ length2

1

0

← list →

Synth (⌘V)

Demo: List Length

Drag to extract

length match list with | hd::tail -> tail

Destruct [0; 0; 0] +

Return ?

Bindings inside function. Drag what you want below, or do

; 0; 0]

Return expression(s) and value(s)

Autocomplete to value

(??) int

1 +

1 + []

1 + 0

1 + 0

1 + [0]

Assertions

✓ length [] = 0

✗ length [0; 0; 0] = 4

4 3

Autocomplete to extract

(??) 'a

1 + length

1 + length [0; 0; 0]

1 + length 0

1 + length ; 0; 0]

1 + length list

Synthesis

²length [0; 0] tail + 1

Accept

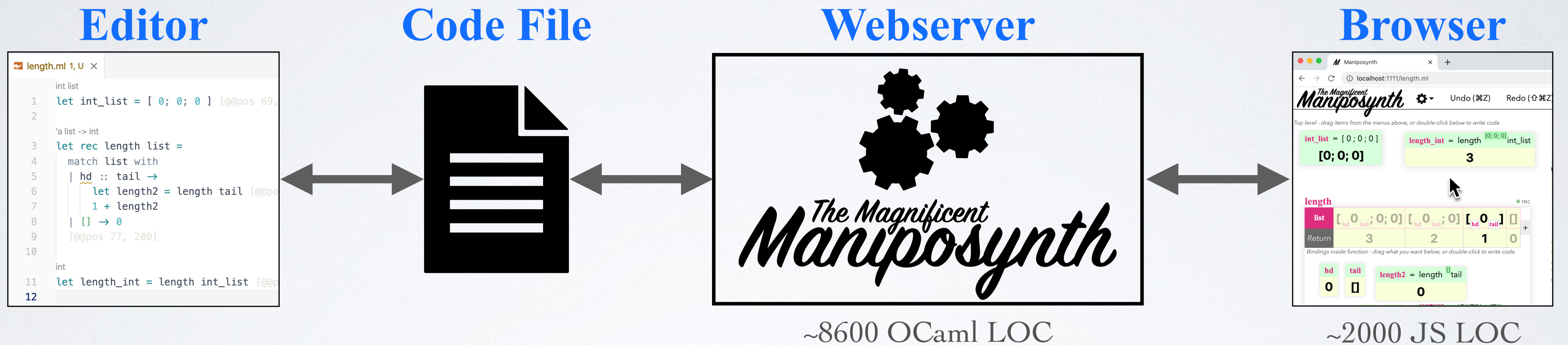
Reject

3

The Magnificent **Maniposynth**

- Goals
- Demo
- **Implementation**
 - **Interpreter**
 - **Binding reordering**
 - **Synthesizer**
- Evaluation
- Future Work & Conclusion

Architecture



Code is
"ground truth"

Server runs code
and renders HTML

Browser polls
for changes
or tells server
to do an action

Interpreter

- Adapted the interpreter from Camlboot [Courant et al. 2020]
(Couldn't just modify the standard OCaml tools because the OCaml compiler performs type erasure — can't log the value when expression is at polymorphic type!)
- On each execution step, log:

```
(exp/pat, call frame num, val, env)
```
- For live display, show value at that exp/pat with the current call frame num

Binding reordering

2D canvas is unordered, let-bindings in code are automatically reordered to bring items into scope.

Requirement: All names at the same “indentation level” must be unique.

```
let a = 1
let c () =
  let x = (a, b, c, d) in
  let a = 0 in
  x
let b = 2
```

```
let a = 1
let b = 2
let rec c () =
  let a = 0 in
  let d = (??) in
  let x = (a, b, c, d) in
  x
```

Synthesizer

```
'a -> 'b
```

```
let length list = (??)
```

```
let () = assert (length [ 0; 0; 0 ] = 3)
```

- No big ideas, just want it to work with
 - (a) few examples,
 - (b) no type annotations, and
 - (c) produce quality resultseven with the Pervasives functions in scope (e.g., addition, subtraction, etc).

- Type-directed, inspired by Myth (Osera and Zdancewic 2015)

- With a probabilistic context-free grammar (PCFG)

| | | | | | |
|------------------------|-------|---|--------------------------------|-------|--|
| Expressions e | $::=$ | 52% x | Names x | $::=$ | 73% $localName$ 27% $pervasivesName$ |
| | | 20% $e_1 \bar{e}_i$ | Local Names $localName$ | $::=$ | 31% $MostRecentlyIntroduced$ |
| | | 10% fun $x \rightarrow e$ | | | 20% $2ndMostRecentlyIntroduced$ |
| | | 8.1% $ctor$ | | | 11% $3rdMostRecentlyIntroduced$ |
| | | 6.6% c | | | ...etc... |
| | | 1.9% match e_1 with $\overline{C... \rightarrow e_i}$ | | | |
| | | 1.3% if e_1 then e_2 else e_3 | | | |

- More in paper and preprint appendix

The Magnificent **Maniposynth**

- Goals
- Demos
- **Implementation**
- Evaluation
- Future Work &
Conclusion

The Magnificent **Maniposynth**

- Goals
- Demos
- Implementation
- **Evaluation**
- Future Work & Conclusion

Two Evaluations

1. An expert (me) implemented 38 examples from the first lessons of a functional data structures course (IN2347 Functional Data Structures, Technische Universität München)
2. Exploratory user study with two professional OCaml programmers

Goal: **qualitative** insights. What is or is not working?

Example Implementation Results

| Function | LOC | Asserts | Time | Mouse | Keybd | Un/Re/Del | TypeErr | Crash |
|------------------|------------|-----------|--------------|------------|------------|-----------|-----------|----------|
| nat_plus | 5 | | 0.8 | 6 | 5 | | | |
| nat_minus | 8 | | 1.9 | 6 | 11 | | | |
| nat_mult | 9 | | 1.4 | 8 | 6 | | | |
| nat_exp | 13 | | 2.1 | 9 | 6 | | | |
| nat_factorial | 13 | | 1.6 | 8 | 4 | | | |
| nat_map_sumi | 10 | | 2.6 | 11 | 5 | | 1 | |
| count | 9 | | 1.9 | 9 | 11 | | | |
| length | 4 | | 0.3 | 1 | 7 | | | |
| snoc | 8 | 1 | 2.4 | 8 | 12 | 2 | | |
| reverse | 8 | | 1.5 | 4 | 9 | | | |
| nat_list_max | 17 | | 4.6 | 23 | 21 | | | |
| nat_list_sum | 13 | | 1.1 | 9 | 4 | | | |
| fold | 9 | | 3.2 | 14 | 6 | | | |
| shuffles | 14 | | 14.5 | 25 | 28 | 2 | | |
| contains | 9 | | 2.2 | 10 | 13 | 1 | | |
| distinct | 16 | | 2.4 | 9 | 11 | 2 | | |
| foldl | 10 | 1 | 1.5 | 10 | 6 | | 1 | |
| foldr | 8 | 1 | 1.8 | 10 | 5 | | | |
| slice | 12 | 3 | 9.8 | 19 | 22 | 4 | | |
| append | 8 | 1 | 1.4 | 7 | 9 | | | |
| sort_by | 21 | 3 | 6.2 | 17 | 29 | | | |
| quickselect | 13 | 1 | 13.1 | 19 | 38 | 1 | 1 | |
| sort | 16 | 3 | 5.6 | 11 | 32 | 2 | | |
| ltree_inorder | 12 | 1 | 2.9 | 7 | 20 | 1 | 1 | |
| ltree_fold | 13 | 1 | 3.1 | 13 | 13 | | | |
| ltree_mirror | 11 | 1 | 4.4 | 12 | 6 | | 1 | 1 |
| bst_contains | 14 | 3 | 6.6 | 11 | 32 | 1 | | |
| bst_contains2 | 17 | 5 | 10.4 | 20 | 41 | 2 | | |
| btree_join | 34 | 2 | 61.7 | 82 | 64 | 51 | | 2 |
| bst_delete | 36 | 2 | 14.4 | 31 | 24 | 4 | | |
| bstd_valid | 29 | 3 | 32.2 | 63 | 100 | 4 | 1 | |
| bstd_insert | 18 | 2 | 8.0 | 38 | 23 | 3 | | |
| bstd_count | 21 | 1 | 7.6 | 15 | 32 | 1 | | |
| bst_in_range | 31 | 3 | 9.3 | 23 | 39 | 3 | | |
| btree_enum | 29 | 3 | 19.2 | 31 | 51 | 6 | 3 | |
| btree_height | 15 | 1 | 1.9 | 11 | 14 | | | |
| btree_pretty | 14 | 1 | 3.7 | 4 | 21 | | 4 | |
| btree_same_shape | 19 | 1 | 8.1 | 14 | 34 | 7 | | |
| Total | 566 | 44 | 277.6 | 628 | 814 | 97 | 13 | 3 |

Fastest, 0.3min

Slowest, 62min

4.5 hours, 3 tool crashes, but success!

*type 'a btree = / Node of 'a btree * 'a * 'a btree / Empty*

Top level - drag items from the menus above, or double-click below to write code

tree1 = Node (Empty , 1 , Node (Empty , 2 , Node (Node (Empty , 3 , Empty) , 4 , Empty)))

tree2 = Node (Node (Empty , 5 , Node (Empty , 6 , Empty)) , 7 , Empty)

✓ btree_join tree1 tree2 = Node (Node (Empty , 1 , Node (Empty , 2 , Node (Empty , 3 , Empty))) , 4 , Node (Node (Empty , 5 , Node (Empty , 6 , Empty)) , 7 , Empty))

✓ btree_join tree1 Empty = Node (Node (Empty , 1 , Node (Empty , 2 , Node (Empty , 3 , Empty))) , 4 , Empty)

| btree_join | | | | | | | rec |
|------------|--|---|--|--------------|---|--|-------|
| tree1 | | | | ...2 more... | | | Empty |
| tree2 | | | | ...2 more... | | | Empty |
| Return | Node (Node (Empty , 1 , Node (Empty , 2 , Node (Empty , 3 , Empty))) , 4 , Node (Node (Empty , 5 , Node (Empty , 6 , Empty)) , 7 , Empty)) | Node (Node (Empty , 2 , Node (Empty , 3 , Empty)) , 4 , Node (Node (Empty , 5 , Node (Empty , 6 , Empty)) , 7 , Empty)) | Node (Node (Empty , 3 , Empty) , 4 , Node (Node (Empty , 5 , Node (Empty , 6 , Empty)) , 7 , Empty)) | ...2 more... | Node (Node (Empty , 2 , Node (Empty , 3 , Empty)) , 4 , Empty) | Node (Node (Empty , 3 , Empty) , 4 , Empty) | Empty |

+ Bindings inside function - drag what you want below, or double-click to write code

Return expression(s) and value(s)

Observations

Could hide the code

No trouble with binding order
(some trouble with nested match order)

Value-oriented vs. **expression**-oriented thinking

length3 = length [0 ; 0 ; 0]

3

Despite trying to place attention on **values**...

...often thought only about **expressions**.

User Study

- 2 participants x 3 sessions x 2 hours each
- 5 and 11 years of professional OCaml experience
- Ran Maniposynth on their own computers alongside Vim
- Participants attempted exercises with varying amounts of guidance from facilitator
- Goal: qualitative insights

Observations from User Study

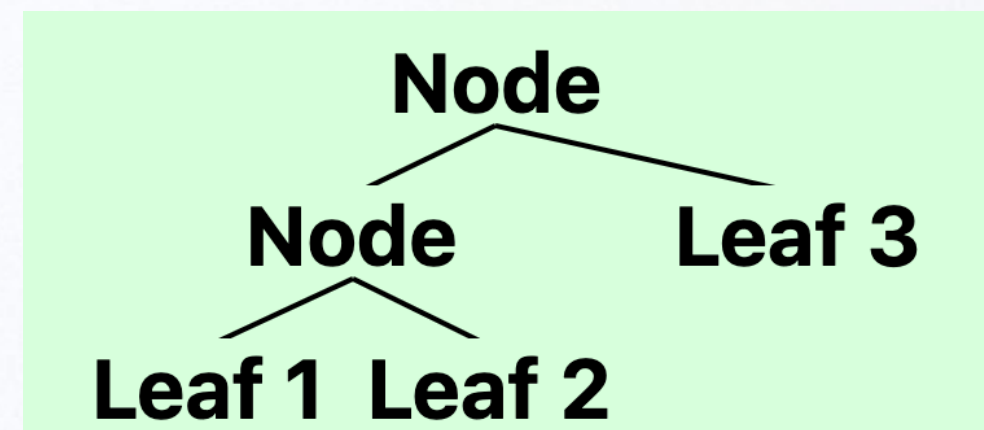
- Positive about non-linearity: “fits a lot more with how I like to write code” (P1)
- Too many colors, too few labels
- Even in session three, both participants occasionally still needed guidance from the facilitator
- Writing assertions was not a problem: both wanted to do so, unprompted
- Synthesis only produced useful results 16% of the time, but participants were not bothered when it did not
- (More in paper)

Expression-oriented vs.
Value-oriented thinking

P2 didn't fully realize they were working with live values until *after* the first exercise.

P1 & facilitator stuck on a bug that was clear from looking at the live values

P2 was so used to reading
`Node (Node (Leaf 1, Leaf 2), Leaf 3)`
they were subtly repelled by beautified trees



The Magnificent **Maniposynth**

- Goals
- Demos
- Implementation
- **Evaluation**
- Future Work & Conclusion

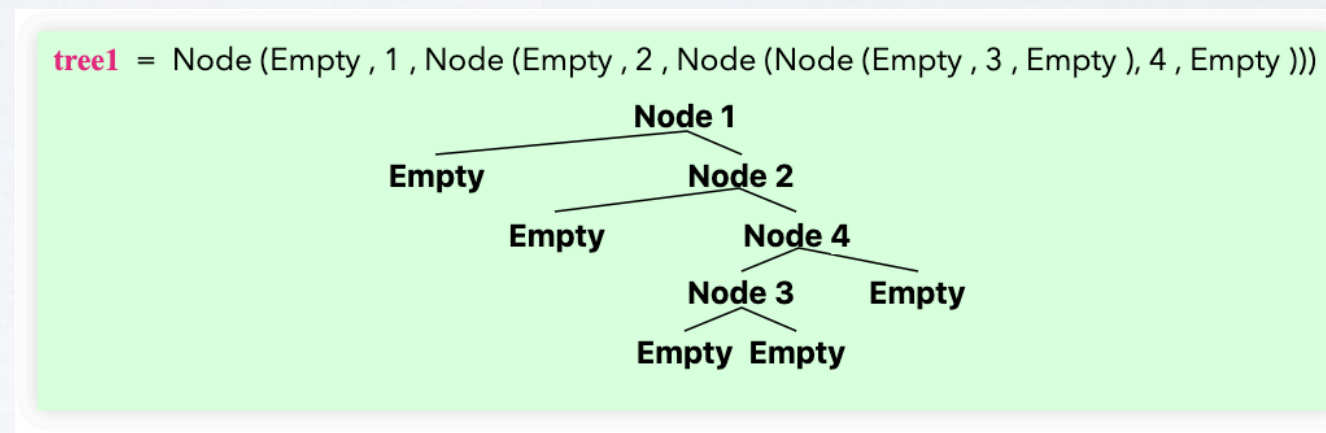
The Magnificent **Maniposynth**

- Goals
- Demos
- Implementation
- Evaluation
- **Future Work & Conclusion**

Future Work

More self-description in UI (Tooltips?)

Shrink large values



Encourage **value-oriented** thinking

- Display values instead of variable names?

```
List.mem 2 target [1; 2; 3] list
```

```
List.mem target 2 list [1; 2; 3]
```

- More actions on values?

Conclusion

Yes, you can have a **graphical, non-linear** interface even when the program is **ordinary code**.

The Magnificent Maniposynth

Bimodal Tangible Functional Programming

The image shows a screenshot of the Maniposynth IDE. On the left, a code editor displays the following OCaml code:

```
1 let int_list = [ 0; 0; 0 ] [@@pos 69, 72]
2
3 'a list -> int
4 let rec length list =
5   match list with
6   | hd :: tail ->
7     let length2 = length tail [@@pos 55, 12] in
8     1 + length2
9   | [] -> 0
10  [@@pos 77, 200]
11
12 int
13 let length_int = length int_list [@@pos 276, 76]
```

On the right, the visual execution environment shows the state of the program. At the top level, the following bindings are visible:

- `int_list = [0; 0; 0]`
- `length_int = length [0; 0; 0] int_list` with the value `3`.

The `length` function is shown with its arguments and return values:

| list | hd | tail | length2 | Return |
|-----------|----|--------|---------------|--------|
| [0; 0; 0] | 0 | [0; 0] | length [0; 0] | 3 |
| [0; 0] | 0 | [0] | length [0] | 2 |
| [0] | 0 | [] | length [] | 1 |
| [] | | | | 0 |

Below this, the function's internal state is shown:

- `hd = 0`
- `tail = []`
- `length2 = length []` with the value `0`.

At the bottom, the return expression and value are shown:

- Return expression: `1 + length2` with the value `1`.
- Return value: `0`.

A blue starburst graphic on the right contains the text "Thank you!". At the bottom right, there is a button labeled "Synth (⌘Y)".

Visit maniposynth.org for artifact and video