Coral: An Ultra-Simple Language For Learning to Program

Dr. Alex Daniel Edgcomb, Zybooks

Alex Edgcomb is Sr. Software Engineer at zyBooks.com, a startup spun-off from UC Riverside that develops interactive, web-native learning materials for STEM courses. Alex is also a research specialist at UC Riverside, studying the efficacy of web-native content and digital education.

Prof. Frank Vahid, University of California, Riverside

Frank Vahid is a Professor of Computer Science and Engineering at the Univ. of California, Riverside. His research interests include embedded systems design, and engineering education. He is a co-founder of zyBooks.com.

Prof. Roman Lysecky, University of Arizona

Roman Lysecky is a Professor of Electrical and Computer Engineering at the University of Arizona. He received his Ph.D. in Computer Science from the University of California, Riverside in 2005. His research interests include embedded systems, runtime optimization, non-intrusive system observation methods, data-adaptable systems, and embedded system security. He has recently coauthored multiple textbooks, published by zyBooks, that utilize a web-native, interactive, and animated approach, which has shown notable increases in student learning and course grades.

Coral: An Ultra-Simple Language For Learning to Program

Abstract

University-level introductory programming courses, sometimes called CS0 (non-majors) and CS1 (majors), often teach an industry language, such as Java, C++, and Python. However, such languages were designed for professionals, not learners. Some CS0 courses teach a graphical programming language, such as Scratch, Snap, and Alice. However, many instructors want a more serious feel for college students that also leads more directly into an industry language. In late 2017, we designed Coral, an ultra-simple language for learning to program that has both textual code and a graphical flowchart representation. Coral's educational simulator was designed hand-in-hand with the language. Coral was designed specifically for learning core programming concepts: input/output, variables, assignments, expressions, branches, loops, functions, and arrays. Coral is intended as a step in learning; once Coral is learned, students might transition to an industry language, which is mostly learning syntax since the student has already learned core programming concepts via Coral. This paper describes Coral, including the design philosophy and pedagogical considerations. This paper also includes data on usage and perspectives by 159 students of Coral during Fall 2018, finding that students with minimal programming experience can independently learn Coral input/output, variables, branching, and loops in fewer than 3 hours on average by independently completing the readings and homeworks. Coral has been used by about 2600 students at 21 universities.

1. Introduction: Why a new language?

Industry coding languages like Python, Java, and C++ were designed mostly for professionals, not learners. Python is often considered the simplest to learn, but as one long-time instructor put it, “even Python has its ‘Gotchas'”, which is supported by some research where evidence was found that students struggle with Python as much as with C++ [1][2][3]. For example, some Python syntax is non-intuitive to learners, like reading integers. Another example is that the lack of static typing in Python can yield hard-to-debug type-related errors. We considered subsetting/redefining Python for learners, but knew the needed departures could make transitioning to real Python confusing. Some features of Python (and other coding languages) do seem to be quite useful for learning to code, such as Python's required indentation, something that Java and C++ instructors frequently need to remind students of, to make Java and C++ code easier to read.

Languages like Scratch [4], Snap [5], and Alice [6] help attract people to computing, especially in K-12. These languages use block-based programming, rather than textual code. Block-based programming languages tend to have simpler syntax, which is visualized via connecting patterns. However, many introductory programming instructors say they want something with a more

serious feel for college students (and some students say the same), and/or that leads more directly into industry languages.

Raptor is a flowchart language for learners [7]. A flowchart visually captures the execution model, which can be especially helpful if a standard layout is used, but that's not typically the case when learners build a flowchart. Also, a newer approach is needed that uses HTML5 to enable ubiquity of operating system and electronic device usage. Plus, a unified flowchart / code language is needed to help lead into an industry language.

Coral was designed with equivalent code and flowchart versions, unlike existing languages. The flowchart is auto-generated from the code. Coral's educational simulator was also designed hand-in-hand with the language [8]. Coral was designed specifically for learning core programming concepts: input/output, variables, assignments, expressions, branches, loops, functions, and arrays. Once learned, students are ready to transition to an industry language (if desired), wherein the student could focus more on syntax issues of the particular industry language since the core concepts were already learned. Coral is intended as a step in learning, not a language for producing real applications (though future uses of Coral for real applications is not beyond consideration).

This paper describes the Coral language, including numerous examples of textual code and corresponding flowchart(s). This paper also describes the Coral educational simulator, and shares early usage data from introductory programming students.

2. Coral description and examples

This section describes Coral's language constructs through examples, and explains the language design decisions. Coral has a flowchart and textual representation, and supports core programming concepts: Input/output, variables, branching, loops, arrays, and functions. The textual representation was specifically designed to look and read like pseudocode, so new programmers can quickly learn to read Coral.

Figure 1 shows an example of putting quoted text (a string literal) to output. The syntax reads like a sentence and deliberately does not introduce a function call, in contrast to Python, Java, and C, nor an operator like in C++. Note that a flowchart always starts with a Start node and ends with an End node, as shown in Figure 1(b).

A variable is a memory location that holds a value. As shown in Figure 2(a), integer x declares a variable named x, able to hold an integer value (32-bit signed). Then, x = 5 assigns x with the value 5. In Figure 2(b), the declaration is shown visually in memory, and the assignment is in a

node. Another data type is float that stores a floating-point value (32-bit). An example of a variable put to output: Put x to output

Figure 1: Example of string literal output in Coral for (a) textual code and (b) flowchart.

A variable must be declared before being assigned a value. Each line is a statement, and only one statement per line is allowed; and thus no semicolon is required to separate statements. Each statement executes one at a time, and, in the flowchart, each statement is a unique node. Like most languages, a variable's name starts with a letter, then any letters, digits, or underscores. Valid: x, x2, numDogs, total_width. Invalid: 2x, num-cats, flag!.

In textual code, all variables must be declared at the top of the code, before any other code, which avoids common novice programmer issues, such as where to put variable declarations and scope. As shown later in functions, variable declarations must be at the top of the respective function.

For now, Coral does not support user's declaring boolean or string variables, as our design philosophy was to only include what was absolutely necessary; however, such data types may later be supported as some instructors have suggested needs for boolean and string variables.

Figure 2: Example of variable declaration and assignment as (a) textual code and (b) flowchart.

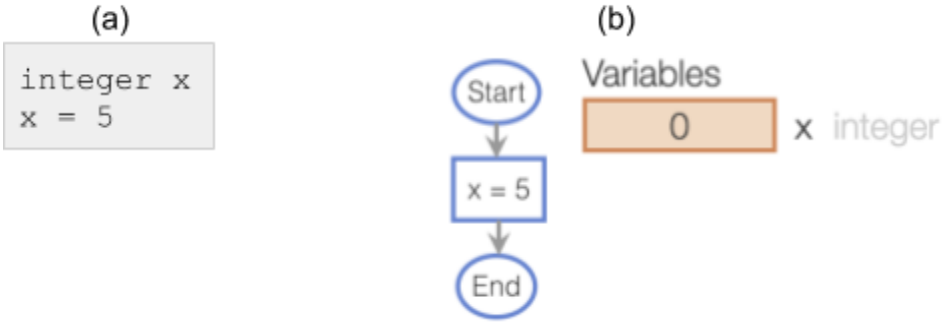


Figure 3 shows the input construct, which, like the output construct, uses a syntax that reads as a sentence, rather than using a function or operator. Coral only supports input of integer and floating-point numbers. A floating-point input assigned to an integer variable is automatically

typecast, and vice-versa. For example, an input of 4.5 assigned to an integer variable will store 4 in the integer variable.

Figure 3: Example of getting the next input as (b) textual code and (b) flowchart.

(a)

```
integer x
x = Get next input
```

(b)

An assignment statement assigns a variable with the value of an expression. As shown in Figure 4(a), the third statement assigns x with 2 times x's current value. If x was 4, the statement assigns x with 2 * 4, or 8. An assignment statement's right-hand side can be an arithmetic expression involving +, -, *, /, %, and (). Ex: x = (3 * y) + (z / 2).

Figure 4: Sample program that prints double the value of the input value in (a) textual code and (b) flowchart.

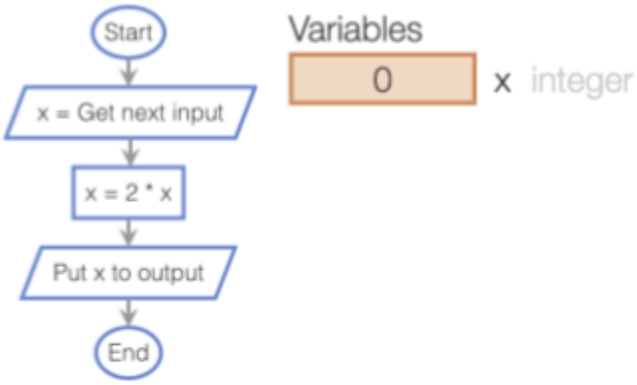
(a)

```
integer x

x = Get next input
x = 2 * x

Put x to output
```

(b)



An if-else construct implements branching. As shown in Figure 5, if $x < 0$ is true, the first branch executes, outputting "Negative". Else, the second branch executes, outputting "Non-negative". After the if-else construct, "\nDone" is output. In textual code, an indentation is 3 spaces, and indentation is required, like in Python. Differently from Python, a colon is not required at the end of the condition, and, differently from C, C++, and Java, parentheses are not needed, nor are curly braces needed for multiple sub-statements.

The flowchart layout, as seen in Figure 5(b), was deliberately designed to mimic the textual code indentation of branch statements. Another branching construct is elseif, which may follow an if or elseif, but not an else, statement. Note that conditional type conversions are not allowed. For example, `not 1` is not allowed.

Figure 5: Branching example as (a) textual code and (b) flowchart. Flowchart layout mimics indentated textual code.

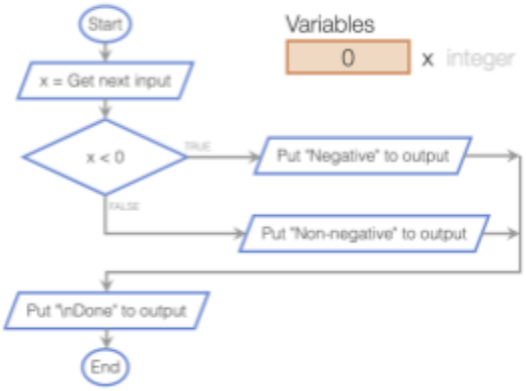
(a)

```
integer x

x = Get next input
if x < 0
   Put "Negative" to output
else
   Put "Non-negative" to output

Put "\nDone" to output
```

(b)



A while loop construct executes its sub-statements while its condition is true. As shown in Figure 6, x is gotten from input, and while x > 0, the loop outputs x's square. Again the flowchart layout mimics the indentation of the textual code for the sub-statements of the loop.

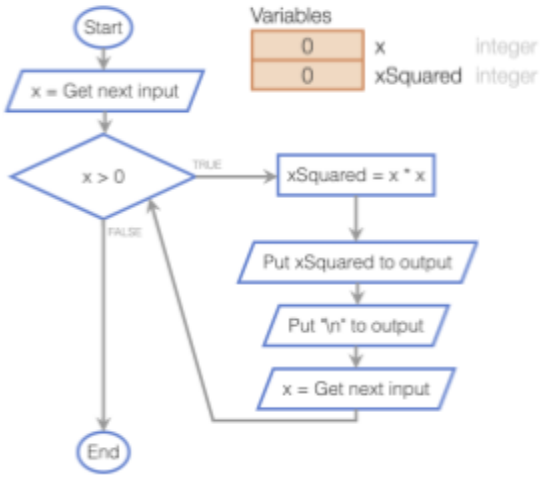
Figure 6: Example of while loop construct that continues looping while x (gotten from user input) is positive, as (a) textual code and (b) flowchart.

(a)

```
integer x
integer xSquared

x = Get next input
while x > 0
   xSquared = x * x
   Put xSquared to output
   Put "\n" to output
   x = Get next input
```

(b)



A for loop construct iterates a specified number of times. The for loop in Figure 7 iterates with i being 0, 1, 2, 3, then 4. i = 0 executes only once, when the for loop first executes. Then condition i < 5 is checked, and if true the sub-statements execute. After, the update i = i + 1 executes, and i < 5 is checked again; the update and check repeat until the condition is false. The textual code, in Figure 7(a), is a similar syntax to the for loop in C, C++, and Java, with a key difference being not needing parentheses around the three parts of the for loop. We chose this syntax, over more general for loops that iterate through a range, due to having a clear correspondence with a while

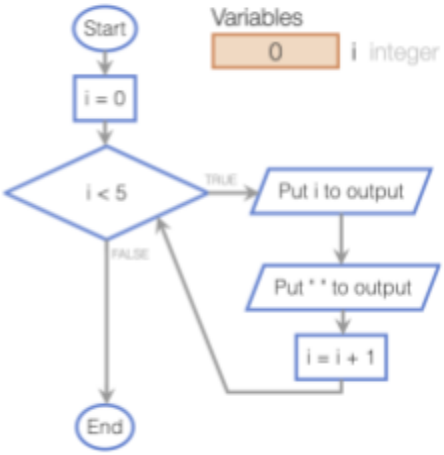
loop's initialization, loop expression, and update statements, which has the added benefit of having the same visualization in a flowchart.

Figure 7: For loop example that iterates i from 0 to 4 in (a) textual code and (b) flowchart. The for loops three parts are separated into three nodes in the flowchart.

(a)

```
integer i
for i = 0; i < 5; i = i + 1
   Put i to output
   Put " " to output
```

(b)



Comments help a human understand code. Each comment starts with //, and must be on its own line. As shown in Figure 8, a comment is above the for loop, and another comment is above a put to output. A comment usually describes the code below the comment, so the flowchart, as shown in Figure 8(b), puts the comment in the node of the code just below. Sequential comments are put into the same node.

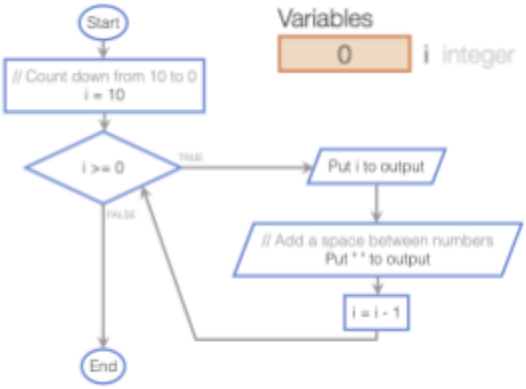
Figure 8: Sample program with comments in (a) textual code and (b) flowchart. Comments in the flowchart are shown above the associated statement.

(a)

```
integer i

// Count down from 10 to 0
for i = 10; i >= 0; i = i - 1
   Put i to output
   // Add a space between numbers
   Put " " to output
```

(b)



An array variable holds many values rather than just one value. The number of values (size) can be set in the declaration, as shown in Figure 9(a). The size can be read as myArray.size, useful in loops iterating through the array, as shown in Figure 9. The size also makes clear how many elements are in the array. As shown in Figure 9(b), memory shows the name of the array, along

with the specific syntax to access each value associated with the array, such as userNums.size and userNums[0].

An array may be initialized with a size of ?, but the size must be set with a number before an array element can be accessed. Also, the array size can only be set with a number once, and then the size cannot change, not even to ?. This way a user-entered value can be used to set the size of an array, but the novice programmer cannot make a common resizing mistake.

Figure 9: Example of an array declaration, assignment, and read in (a) textual code and (b) flowchart. Each element in the array is shown in a contiguous block of memory.

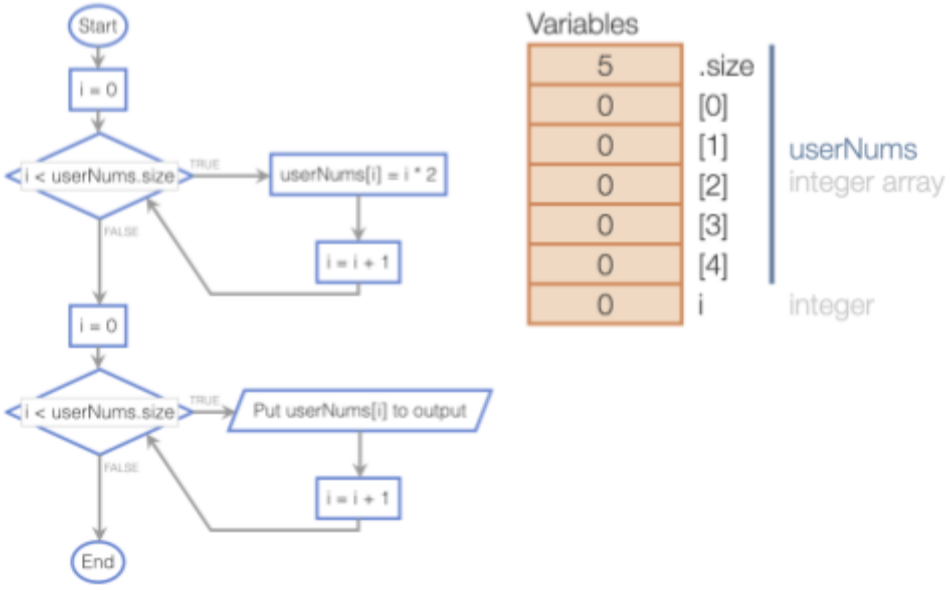
(a)

```
integer array(5) userNums
integer i

for i = 0; i < userNums.size; i = i + 1
   userNums[i] = i * 2

for i = 0; i < userNums.size; i = i + 1
   Put userNums[i] to output
```

(b)



A function is a statement group callable from code. A function definition may have parameter variables and a return variable. The functions returns the return variable's last value. If a user defines a function, the main code must then be in a main function (which is otherwise implicit). The syntax of the function definition follows the same syntax as many other languages, like Java

and Python, but adds a keyword "returns", which is deliberately included to help the definition read more like a sentence.

As shown in Figure 10(a), a function without parameters has empty parentheses, like Main, and a function that does not return a value uses “returns nothing”, again like Main.

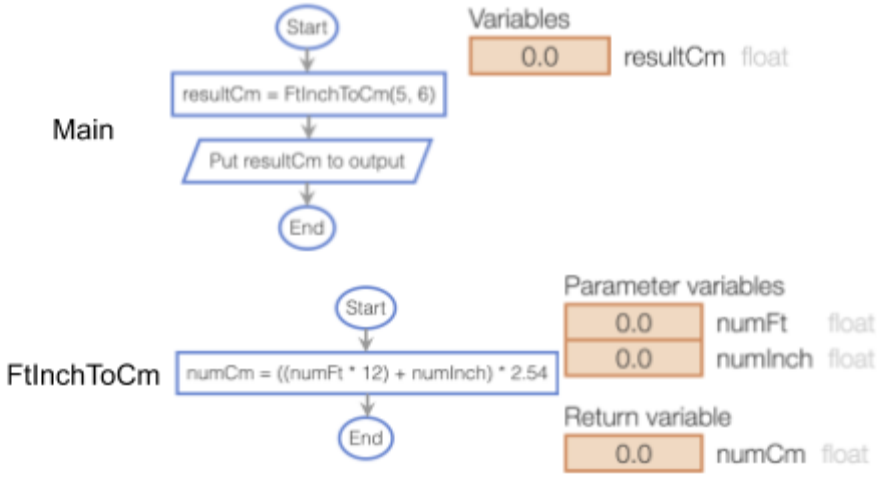
Figure 10: Example of two functions, FtInchToCm has 2 parameter variables and 1 return and Main has no parameter or return variable, in (a) textual code and (b) flowchart.

(a)

```
Function FtInchToCm(float numFt, float numInch) returns float numCm
   numCm = ((numFt * 12) + numInch) * 2.54

Function Main() returns nothing
   float resultCm
   resultCm = FtInchToCm(5, 6)
   Put resultCm to output
```

(b)



Some more examples and sample uses of functions:

- A function call’s arguments can be expressions, such as:
  z = FtInchToCm(x, y + 1)
- A function call may appear in an expression, such as:
  z = 1.0 + FtInchToCm(5, 6)
- A Put statement’s item may be an expression, so a call may appear there, such as:
  Put FtInchToCm(5, 6) to output

Coral supports several built-in functions. There are built-in math functions, such as: SquareRoot(x), RaiseToPower(x, y), and AbsoluteValue(x). Coral also supports random number generation, such as RandomNumber(0, 5), which returns a random integer between 0 and 5

(inclusive). The random number generator's seed can be set via SeedRandomNumbers(x), and the default seed is based on the current time.

3. Coral construct summary

This section summarizes the constructs supported in Coral. Table 1 lists these constructs. Examples for each construct are shown in the previous section.

Table 1: List of Coral's 10 constructs.

Construct	**Form**
Variable declaration	`[type] [identifier]`
Input statement	`[var] = Get next input`
Output statement	`Put [item] to output`
Assignment statement	`[var] = [arithexpr]`

Branch statements	`if [condexpr]`
	`  [substatements]`
	`elseif [condexpr]`
	`  [substatements]`
	`else`
	`  [substatements]`

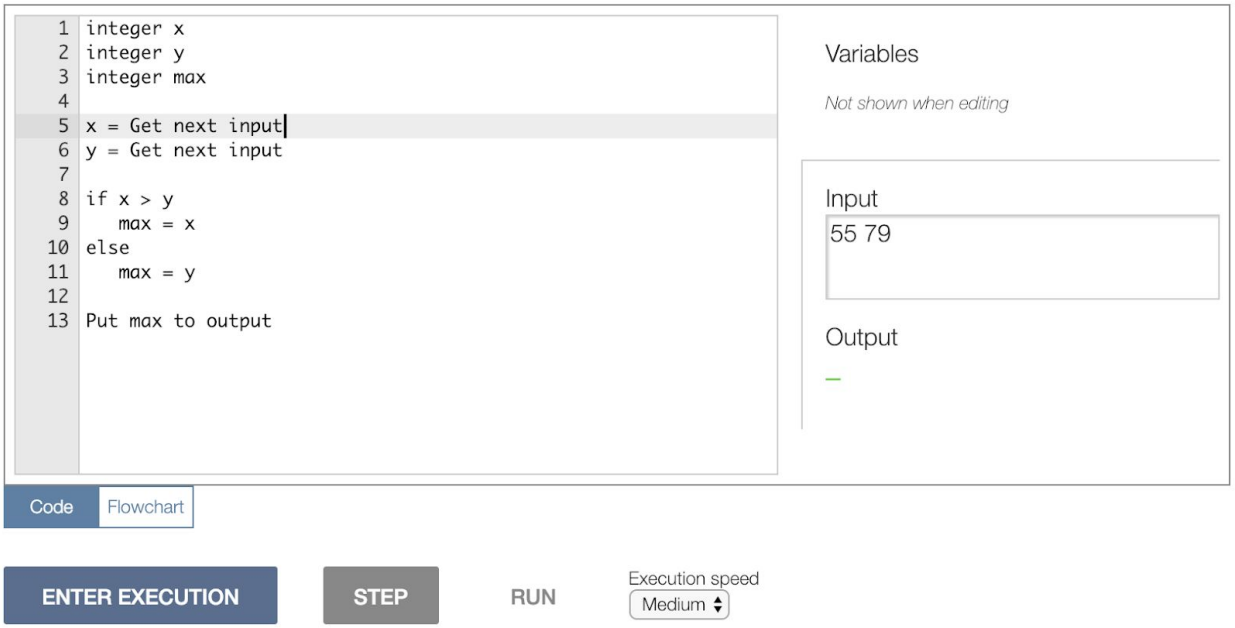
While loop statement	`while [condexpr]` `  [substatements]`
For loop statement	`for [init]; [condexpr]; [update]` `  [substatements]`
Comments	`// [any text]`
Arrays	`[type] array([size]) [identifier]`
Functions	`Function [identifier]([parameters]) returns [type] [identifier]` `  [substatements]`

4. Coral educational simulator

The Coral educational simulator was designed hand-in-hand with the Coral language. The simulator is freely available online [8]. The simulator helps learners visualize line-by-line execution. In fact, many simulator features would likely also benefit learners of professional languages, like Python and Java, if such a simulator were to support such languages. One of this paper's authors uses the simulator when teaching C++, to help students visualize variables, sequential execution, branching, loops, function calls, array accesses, and more.

Figure 11 shows the simulator when a user is editing the textual code. A user can also provide input values, and modify the execution speed, which has options Slow (2 seconds pause between line executions), Medium (1 second pause), Fast (1/2 second pause), and Instant. The simulator supports a Code and Flowchart view. Figure 11 shows the Code view. The user can click Flowchart to automatically see the corresponding flowchart to the textual code. If the textual code has a syntax error when Flowchart is clicked, then that error is shown and the editor stays in the Code view. Deliberately, only the first syntax error is shown to help new learners focus and because syntax errors become less reliable after each such error due to compiler confusion, which is a common point of confusion for new learners. Similarly, when Enter execution is clicked and there is a syntax error, the student is shown the first error and remain in edit mode. Otherwise, the simulator transitions to execution mode, shown in Figure 12.

Figure 11: User can edit the textual code and input values. User can flip to flowchart view to see the textual code as a flowchart.



In edit mode, as shown in Figure 12, the simulator automatically highlights the next line of code to execute. Line 5 just executed, wherein the next input was assigned to the variable x. In Variables, we can see x stores 55 and that 55 is highlighted, indicating the value was just assigned. In Input, 55 is grayed out, indicating the value has already been read, whereas 79 is still blue, so has not yet been read. Output has not yet received a value, so just has a waiting cursor. A user can click Step to execute the next line of code. A user can also click Run, which simply auto-clicks Step after each execution finishes. A user can click Flowchart to switch to flowchart view, shown in Figure 13, wherein the next to execute node is highlighted. A user can click Exit execution to return to edit mode.

Figure 12: Execution is shown line by line with line 6 next to execute. In Variables, x was just assigned 55 (via line 5), so 55 is highlighted. In Input, 55 was already read, so is grayed out.

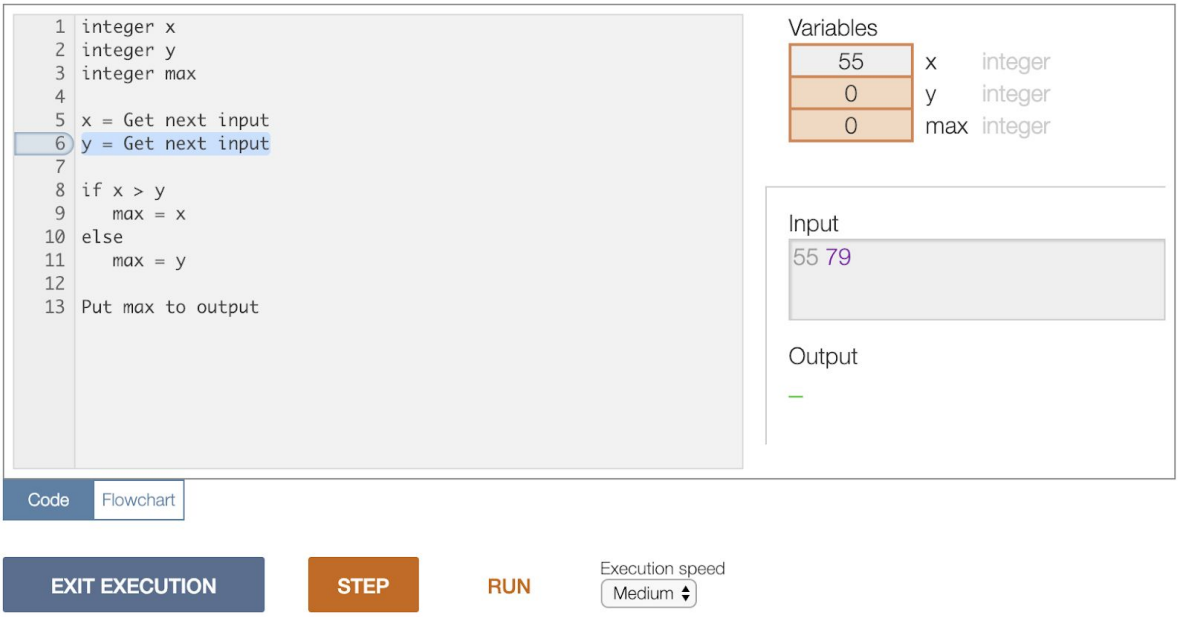
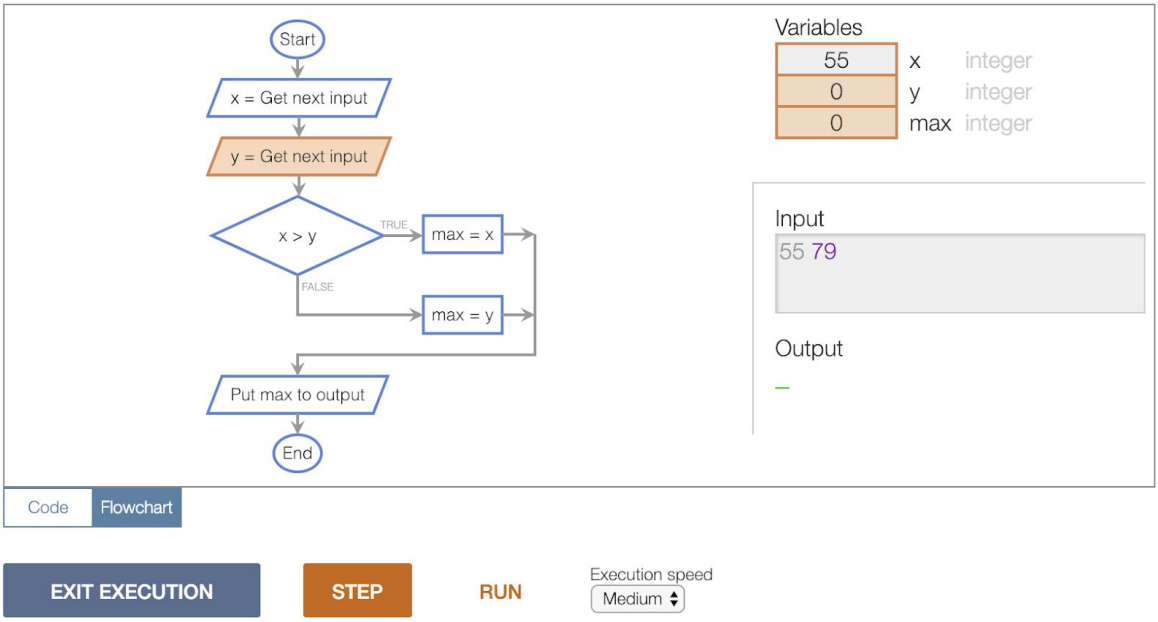


Figure 13: Simulator shown in the same execution state as Figure 12. User flipped to Flowchart, wherein the next to execute node is highlighted.



5. Early data on student usage

This paper focuses on describing the Coral language; however, we have collected some early data on student usage of Coral, which may be of interest to the reader. During Fall 2018, 159 students in ENGR1, a required 1-unit Freshmen course on professional development, were

assigned chapters from an interactive textbook that teaches Coral, the Programming Concepts zyBook [9]. The majority of students were Computer Science majors, who were likely also enrolled in an introduction to programming course. Chapters 1 and 2 were assigned and due toward the middle of the term, and chapters 3 and 4 were due 5 days later. Both assignments were worth a few course points and full credit was awarded if the student completed at least 80% of both the reading and homework. Chapter 1 covered basic input and output, chapter 2 integer and floating-point variables, chapter 3 if-else and operators in branching, and chapter 4 while and for loops. The four chapters included 28 sections total with both reading (called Participation Activities, like animations and learning questions, such as multiple choice) and homework (called, Challenge Activities, which consist of about 5 levels that get incrementally harder and provide immediate feedback via auto-grading).

We also instrumented 3 surveys to collect students' subjective feedback: A survey at the beginning of chapter 1 (asking about prior programming experience), another at the end of chapter 2, and another at the end of chapter 4. Chapters 1 and 2 took students 80 minutes on average to complete, and chapters 3 and 4 took 88 minutes, for a total of 168 minutes to complete all four chapters. The surveys included agreeability questions that ranged from strongly agree (+3) to strongly disagree (-3), shown in Table 1. Students slightly agreed that the flowcharts and code were easy to understand. Students were more neutral about whether the code was easy to write. The students generally liked programming, not surprising as that is most of their chosen major. Students reported a minor amount of frustration with issues with the programming language. In fact, we had accidentally included a homework on a language concept that were not included in the chapters.

Table 1: Agreeability questions and average responses from surveys at end of chapters 2 and 4. Range: -3 (Strongly disagree) to +3 (Strongly agree).

	End of ch 2	End of ch 4
The flowcharts were easy to understand.	0.9	0.9
The code was easy to read and understand.	0.8	0.9
The code was easy to write.	0.3	0.2
I often struggled with writing the code for the challenge activities.	-0.3	-0.1
I think I like programming.	1.2	1.6
I often was frustrated by issues with the programming language.	0.4	0.6

6. Future work

This paper describes the Coral language. Future work includes assessing the efficacy of Coral to other languages. For example, comparing how well and efficiently students learn programming

concepts with Coral vs other languages (both industry and learning languages). Another measure of efficacy is how well students who have already learned Coral can then learn an industry language in a CS1 course. Such measures of efficacy would be conducted in both a controlled setting, such as a single lesson with a pre-quiz and post-quiz, as well as cross-semester comparisons, such as a course switching to Coral. Such cross-semester comparisons would include multiple courses.

7. Conclusion

Coral is an ultra-simple language that focuses on teaching core programming concepts: input/output, variables, branching, loops, arrays, and functions. Coral has both a textual code and flowchart representation, and a free educational simulator has been developed for Coral. Industry languages, like Java and Python, were primarily made for industry professionals, whereas Coral was designed for learners. Coral may be useful in CS0 to give students a flavor for programming, or CS1 to teach core programming concepts before transitioning to an industry language. In 168 minutes, students with a little programming experience independently learned Coral by working through readings and homework from 28 sections (four chapters) of an interactive textbook teaching Coral. Students also tended to report that the flowcharts and code were easy to read. Coral has been used by about 2600 students at 21 universities.

References

[1] N. Alzahrani, F. Vahid, A. Edgcomb, K. Nguyen, and R. Lysecky. Python versus C++: An analysis of student struggle on small coding exercises in introductory programming courses, ACM SIGCSE Technical Symposium on Computer Science Education, 2018.
[2] Enbody, R.J., W.F. Punch, and M. McCullen. 2009. Python CS1 as preparation for C++ CS2. ACM SIGCSE Bulletin 41, no. 1: 116-120, 2009.
[3] Enbody, R.J. and W.F. Punch. 2010. Performance of python CS1 students in mid-level non-python CS courses. In Proceedings of the 41st ACM technical symposium on Computer science education, pp. 520-523. ACM, 2010.
[4] Scratch. https://scratch.mit.edu/. Accessed February 2019.
[5] Snap. https://snap.berkeley.edu/. Accessed February 2019.
[6] Alice. https://www.alice.org/. Accessed February 2019.
[7] Raptor. https://raptor.martincarlisle.com/. Accessed February 2019.
[8] Coral Educational Simulator. https://corallanguage.org/simulator/. Accessed February 2019.
[9] zyBooks. https://www.zybooks.com/. Accessed Feb. 2019.