Introductory Programming: Let Us Cut through the Clutter!

[Extended Abstract]

Abhiram G. Ranade
Department of Computer Science and Engineering
IIT Bombay
Powai, Mumbai, India
ranade@cse.iitb.ac.in

ABSTRACT

Introductory programming courses often leave students unimpressed. We feel this is because teaching approaches (a) overemphasize the syntactic aspects of the programming language being taught instead of using programming to do interesting things, (b) do not respect the computational maturity/intellectual leanings of the students, and (c) are simply not fun enough.

We have developed an approach which we believe addresses these issues in the context of teaching introductory programming to college students majoring in science and engineering. We use the C++ programming language augmented with a graphics library and some linguistic devices we have developed. We believe that our approach enables interesting material to be handled from day one and generally garners more student interest.

Keywords

Introductory programming; C++; pedagogy; graphics.

1. INTRODUCTION

Computer programming is a unique skill which is at once deeply theoretical and strikingly hands-on/practical. It is its own science, but yet it can enable you to explore other subjects such as the sciences, engineering and even arts. Computer programming can be psychologically liberating in that a computer *obeys* the student, who for most of his/her life has deferred to elders. Computer programming thus has the potential to empower students and unleash their creative abilities.

And yet, introductory programming courses can appear boring. The most common reason given for this is that a lot of dull information must be conveyed before anything interesting can start. In the most popular languages such as C, C++ and Java your first program typically contains a lot of arcane mumbo jumbo, and even after that it barely prints "Hello world!". Even the rest of the course may convey the impression that computer programming is about getting the semicolons right and mastering obscure trivia ("What does `i += ++i+k++;` do?").

We believe that the introductory programming course should not only provide programming skills but also convey the power, the excitement and the pervasiveness of computer programming. Doing this is not easy. One of the major requirements is to use good programming examples: those which are exciting and challenging to the students, and yet not too hard. Another problem, ubiquitous in all education, is of providing the right motivation. Why should I learn inheritance? Why should I learn software engineering? We must either take the time to explain the motivation in a manner that can be appreciated by the student, or we must simply defer the topic to a later date. All this requires us to understand our student: what excites her, what she already knows on which we can build.

In this paper our goal is to design an introductory programming course for first year college students majoring in science and engineering. We begin in Section 2 by clarifying what we feel is of importance in introductory programming. In Section 3 we survey some of the major approaches to programming education. In Section 4 we present our approach. By being clear about the goals of the introductory programming course, by focussing the course on programming examples that are interesting to our target student, and by employing some linguistic devices that we have developed (based on preprocessor macros), we believe it is possible to design a course that enables interesting ideas to presented from day one, and generally make for efficient learning.

2. WHAT SHOULD WE TEACH?

For an introductory course, we believe that it is appropriate to define "programming" as "the act of expressing in a programming language the computations that you can perform manually". We mean the word "computation" in its widest sense: including not only arithmetic, but also algebra, geometry, calculus, physics, and even art. We want students to view computing as an extension of thinking – as pervasive and as powerful. By implicitly emphasizing the similarity between human computation and computer computation, we hope to inspire confidence in the student towards the subject.[1]

Programming involves three kinds of skills:

1. Expressing problems from different domains in terms of numbers and questions on those numbers.
2. Observing patterns in (manual) computation.
3. Expressing the patterns using appropriate language constructs.

Introductory programming courses and books often focus mostly on learning the language constructs. The first two skills are independent of the language, and more important and harder.

[1]Humans do not always think algorithmically, e.g. playing chess. But such exceptions really drive home the point about what humans indeed do algorithmically.

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ISBN 978-1-4503-4231-5/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2899415.2899430

The first skill is about realizing that everything from calculus to games can be represented on a computer. It is not enough to merely give this infomation to the student; if the information is to sink in, we must get the student to write programs relating to diverse areas: geometry, symbolic computation, graphics. Students find this exhilarating and empowering.

By "observing patterns in computation" we mean something like: is some sequence of computations repeated? If so, how many times? We want students to introspect over how they compute/solve problems manually. They already know many interesting computations, starting from simple arithmetic on numbers with an arbitrary number of digits to linear algebra, calculus, and also calculations in physics. They have even seen recursion, e.g. taking the derivative of a sum simply involves adding the derivatives of the addends! Besides identifying the iteration or recursion present in their calculations, we also expect students to identify function abstraction and data abstraction – this is again a part of the idea of seeing patterns.

The last step is learning a programming language (though not all its idiosyncracies) and expressing the computation using that language. We are not suggesting that this step is easy. However, we believe that by focussing on appropiate computational examples we can motivate the syntax and semantics of language constructs.[2] This will help make the syntax/semantics appear natural and help learning. We will also show soon that we can use devices like *macros* to create more pedagogically convenient syntax for use in the early part of the course whereby we can plunge into interesting material right away.

3. APPROACHES

We will next survey some of the dominant approaches to programming education.

3.1 The C approach

The chronologically earliest, and perhaps the most dominant approach even today, is due to Kernighan and Ritchie[9]. Their little book has trained a huge number of programmers, and is lauded for its clarity. However, it nevertheless has to deal with the idiosyncracies of the C language in which the syntax often overwhelms the principles. A successor to this is perhaps the book on introductory programming through C++ by Stroustrup[16]. C++ is a clear improvement over C. But even in C++, there is serious danger of the syntax overwhelming the principles, especially for a beginner. The classic example of this is the introductory program:

```
#include <iostream> using
namespace std;

int main(){
  cout << "Hello world!" << endl;
}
```

To a beginner this must appear like some mysterious incantations – potentially leading to intimidation or to boredom. And on top of it, the program accomplishes precious little. Similar problems can appear quite often. For example, the student can get lost while understanding the details and syntactic aspects of a concept such as inheritance, or bored by the various tricky operators of C++.

This approach often draws programming examples mainly from operating systems or compiler viewpoints. These examples, concerning files and parsing, may not generate excitement among engineering and science students.

[2]Appealing to prior experience of the student in mathematics helps in motivating language constructs too: for example, the operator $+$ is routinely "overloaded" in mathematics to mean many things.

3.2 The SICP approach

A strikingly different approach is offered by Abelson and Sussman, as documented in their book *The Structure and Interpretation of Computer programs* (SICP)[1]. SICP is a heady brew that you cannot put down, and Sussman's lectures on MIT OCW from 1986 are thrilling and give goosebumps. The appeal of the approach could be attributed to the simplicity of the Scheme language syntax which makes it possible to get to interesting topics very quickly. SICP does get quickly to interesting topics, e.g. the Babylonian algorithm for square roots, and representation and manipulation of (mathematical) functions symbolically. To see the computer doing tricky algorithms and "doing calculus" can raise the stature of the subject in the eyes of the learner.

However, overall it cannot be said that SICP is introductory. The Babylonian algorithm, for example, is compact but actually quite deep. Discussions about possible semantics of constructs, e.g. of assignments in the functional framework, are greatly exciting for experts, but difficult for novices. One indication of all this are the customer reviews of SICP on amazon.com. Of the 212 reviews listed at the time of writing this, 62 % reviews give it 5 stars, and it is clear that these are reviews from experienced programmers for whom SICP sketches out the grand panorama of programming to which they can relate. On the other hand 25 % of the reviews give it 1 star, and these are reviews from beginners who are frustrated because SICP purports to be introductory, but really demands considerable computational maturity.

3.3 The program derivation approach

Another approach is due to the Dijkstra school[5, 8]. They propose strategies using which programs and their proofs of correctness are generated ("derived") simultaneously. Dijkstra essentialy suggests that beginners should be kept away from computers before they master the logical calculus needed to argue correctness. However many of the "interesting" examples of program derivation are really about discovering new algorithms, e.g. an $O(n)$ time algorithm is derived for a problem that has an obvious $O(n^2)$ time algorithm. The techniques are often elegant and worth mastering; however in our opinion this should perhaps come in a later course.

On the other hand, discussion of program correctness, e.g. assertions and loop invariants is often neglected in introductory programming. These elementary ideas enable programmers to check that "corner cases" are correctly handled, and hence clearly have a place in introductory programming.

3.4 Logo

Logo[12, 6] was invented as a language for teaching programming to children. A major theme in Logo is *turtle graphics*: students get to program the on-screen movement of a symbolic animal, the turtle. The turtle has a pen which draws on the screen as the turtle moves; programs are to be written to draw interesting pictures on the screen. One of the important ideas in Logo pedagogy is to encourage students to "be the turtle". Student are encouraged to ask themselves "If you walk on the figure to be drawn, how much would you move and how much would you need to turn?". Such introspection enables students to make programming very personal and thus make a deep connection to it.

Logo has had many successors, one of the important ones is Scratch[15]. A popular genre of Scratch programming is *narrative* – the graphical objects in Scratch can be animated (along with audio) and a Scratch program is thus the story script. An analog of this in Logo is making the turtle write your name on the screen, or draw pictures without much symmetry. Logo and Scratch are often taught in schools, and often the syllabi focus on this narrative mode.

Less commonly, Logo and Scratch are used for drawing structured pictures, in which the programmer must understand the symmetries in the picture being drawn and implement them in the program by using suitable loops or recursion. In some sense, this *mathematical* mode of usage was the dream of the Logo inventors[6]. This resonates with our idea of "observing patterns in computation and expressing them using appropriate language constructs". We believe that adults with a greater background in mathematics are likely to be more excited about the mathematical mode.

3.5 Domain driven approaches

A class of approaches is motivated by the observation that we never write programs in a vacuum – programs are always written to solve problems. To help in this, one popular idea is to teach programming in conjuction with some application, e.g. robotics[10], or graphics. The Alice system[4] is is built around a 3 dimensional graphics package. We feel that three dimensional graphics is not entirely intuitive and produces a cognitive load which may distract students from the programming content. Several approaches use two dimensional graphics also. EZ windows is a two dimensional graphics system accompanying a well known textbook[3]. Like many such approaches, the book uses graphics as an add-on. It is not used as a pedagogic tool to help present concepts without clutter. Certainly not to help introduce programming on day 1. Programming education has also has been conceived to be given in the context of a geographical application[11]. These approaches are attractive also because they force the student to work alongside an existing system. This is useful because in the modern workplace programmers hardly develop programs for scratch, but rather work to enhance or modify existing programs. A drawback of the approach is that the domain chosen may not appeal to every student, or learning the domain (e.g. robotics principles, 3d graphics principles) may place additional learning burden on the student.

3.6 Paradigm based approaches

Introductory programming has been introduced in the context of specific paradigms, e.g. functional programming using ML[7], or approaches such as objects-first[2]. The Alice system[4] is motivated specifically by the desire to teach object oriented programming. Often the idea is, "let us teach students how to think correctly before other paradigms corrupt them".

From a pragmatic standpoint, we believe that important ideas (e.g. iteration, recursion, object orientation) from every paradigm need to be taught. Our preference is to introduce ideas in increasing order of cognitive complexity, motivated by the demands of the application problem being solved.

4. OUR APPROACH

We would like to have an approach which combines the best of all the approaches described above. Specifically, we would like to get to the heart of programming in the first lecture of the course, just as Logo does for children, and Abelson and Sussman[1] (seemingly) manage to do for adults. Like them, we would not like to be bogged down by syntax. We would like to go beyond basic arithmetic and text processing to provide programming examples and problems to our students without introducing the cognitive load of teaching a completely new topic such as robotics. We would like to teach difficult concepts such as recursion and object oriented programming, but with motivating examples that will appeal to our target audience. As to the choice of the language, we would like to use a mainstream language to improve the chance of our ideas getting accepted.

We use the C++ language, augmented with a library we developed, Simplecpp. Simplecpp supports two kinds of graphics: Logo style *turtle graphics* and more standard *coordinate based graphics* using which geometric shapes can be created and manipulated.

Another component of Simplecpp is a `repeat` statement, also inspired by Logo. The `repeat` statement has the form:

```
repeat (count) { statements to be repeated }
```

This causes the block of `statements to be repeated` to be executed as many times as the value of `count`. For this the statement is translated to a `for` loop using preprocessor macros which get loaded automatically; this is of course revealed to the student only towards the end of the course. The main reason for defining this statement is that it can be introduced in the very first lecture! You will see that with `repeat`, students can start writing interesting programs from day 1.

Our graphics library provides us with an additional domain for illustrating programming concepts and give interesting but challenging assignments. The turtle graphics as well as the two dimensional graphics are very intuitive, and learning them takes hardly any time. Indeed, it is possible to easily create reasonably exciting drawings and animations, e.g. Hilbert space filling curves, the *snake* game, bouncing balls, planets rotating around the sun and so on.

4.1 The first lecture in the course

The first lecture, if delivered well, can cause students to fall in love with the subject. Students tend to view it as setting the tone for the course: whether the course is going to be exciting, whether it will have interesting ideas, or whether it will just be lot of boring information. Our first lecture draws upon Logo/turtle geometry. Here is the first program that we show to students in the first lecture.

```
#include <simplecpp>
main_program{
  turtleSim();

  forward(100);  left(90);
  forward(100);  left(90);
  forward(100);  left(90);
  forward(100);

  wait(5);
}
```

As you can see, we only include the Simplecpp library, which in turn includes `iostream` and issues commands to use namespaces etc. Thus we only need to explain to the students that we need to include Simplecpp, other explanations can come later in the course. Next, we have a macro `main_program` which expands to `int main()`, so we dont need to explain what `int` means and why `main` has parentheses `()` following it. This will get explained after we discuss functions, when the students can understand everything.

The first statement `turtleSim()` in the body of the program opens the turtle simulator window, which already has a turtle at the center of the screen (a red triangle, as is customary). The command `forward(100)` causes the turtle to move forward 100 pixels. The command `left(90)` causes the turtle to turn left by 90 degrees. Thus the complete code causes the turtle to draw a square (because of the pen that drags on the screen as it moves). After that the program waits for 5 seconds, and then terminates.

Note that the first program is already providing a non-trivial ability to the students, and creates expectations in their minds, e.g. "Can I draw other kinds of polygons?". Some students might also ask if they need to write fifty `forward` statements if they wish to

draw a fifty sided polygon. The `repeat` statement can be introduced immediately. Indeed the second program of the first lecture could be

```
main_program{         // will draw a decagon.
  turtleSim();
  repeat(10){
    forward(100); right(36);
  }
}
```

The turning angle, 36 degrees, is easily calculated from the high school geometry theorem "The exterior angles of a polygon add up to 360 degrees."

We have found that the students spontaneously understand nested repeat statements, as in the program below.

```
main_program{
  turtleSim();
  repeat(4){
    repeat(10){
      forward(5); penUp();
      forward(5); penDown();
    }
    left(90);
  }
}
```

Many students spontaneously infer that this will cause a square to be drawn, using dashed lines.

Notice that on the very first day we can accomplish many things. We can force students to think algorithmically. They need to figure out the turning angles; they also need to use `repeat` statements properly to draw more complex figures. This requires matching the pattern in the drawing with the pattern of `repeat` statements in the program. This is of course a very fundamental programming activity! And we have got to in on day 1.

4.2 Utility of repeat and graphics

The standard looping statements in C++ (and most languages) are fairly complex: they require you to understand variables, conditions, and of course issues about loop termination. Thus standard loop statements can be introduced only after a few weeks. We cannot go this long without interesting programming examples!

This vacuum is filled nicely by the `repeat` statement and graphics. Right after the first lecture students can be asked to draw intricate patterns involving lines and arcs (a circle after all is a limit of a polygon as the number of sides increase). These make for relevant, fun, and challenging programming exercises. After discussing data types and assignment statements (but still well before discussing `while/for`), we can write code such as the following

```
int i=1;
repeat(40){
  forward(i*10);
  left(90);
  i = i + 1;
}
```

As you might guess, this draws a spiral. Note further that the code contains *reassignment* of `i` to itself. This is a concept that several students find difficult. We believe that seeing `i` used graphically as above likely helps in understanding reassignment.

Graphics is useful in explaining difficult concepts such as recursion. A (botanical) tree has recursive structure – it consists of smaller trees on top of a trunk. Thus it can be easily drawn using a recursive function.

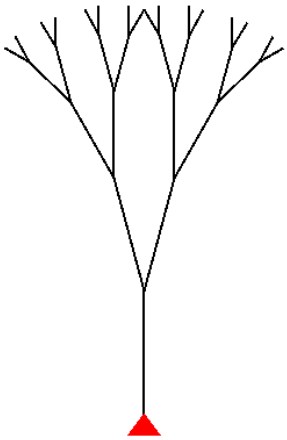

Figure 1: Tree drawn using recursion

```
void tree(int levels){
  if(levels > 0){
    forward(levels*10);
    left(15);
    tree(levels-1);
    right(30);
    tree(levels-1);
    left(15);
    forward(-levels*10);
  }
}
```

Figure 1 shows the picture produced when the above function is invoked by calling `tree(5)`. It should be noted that the tree does not appear instantaneously but is drawn by the turtle, one line at a time. Thus the recursion unfolds in real time in front of the students' eyes. We believe this helps in understanding recursion.

Coordinate based graphics can also be taught very early on. For example, here is the code for creating a rectangle and moving it.

```
Rectangle r(xc,yc,L,H);
  // center coordinates, Length, Height
r.move(deltax, deltay);
```

The astute reader will see that the first statement is really a constructor call, and the second a member function invocation. However, we explain this to students as: "the first statement creates a rectangle named `r`, the second statement moves it." Thus the students start using constructors and the dot notation well before object oriented programming is formally introduced.

Graphics can also provide a compelling motivation to learn inheritance. For drawing interesting pictures, it is necessary to compose the basic shapes together. For example, a car shape (Figure 2) might consist of a polygon shape representing the body, and two wheel shapes. A wheel might consist of a circle denoting the rim, and perhaps straight lines denoting the spokes. We would like to be able to define a car shape once, using wheel shapes defined earlier. Furthermore, it would be nice to be able to invoke member functions such as `move` on cars – this should cause all the contained shapes to move as a group. This precise functionality is provided by our `Composite` shape class.

Figure 2: A car constructed using the `Composite` class

A `Composite` class is a container class which automatically delegates operations such as `move` or `forward` or `rotate` to contained objects (`rotate` requires a bit more work than just delegation). A new composite shape such as a car or a wheel can be defined by inheriting from `Composite`.

Here is how a `Car` class can be defined.

```
class Car : public Composite{
  Polygon* body;
  Wheel *w1, *w2;
public:
  Car(double x, double y, Color c,
      Composite* owner=NULL)
    : Composite(x,y,owner){
    double bodyV[9][2]={{-150,0},
      {-150,-100}, {-100,-100}, {-75,-200},
      {50,-200}, {100,-100}, {150,-100},
      {150,0}, {-150,0}};
    body = new Polygon(0,0, bodyV, 9, this);
    body->setColor(c);
    body->setFill();
    w1 = new Wheel(-90,0,this);
    w2 = new Wheel(90,0,this);
  }
  void forward(double dx){
    Composite::forward(dx);
    // superclass forward function
    w1->rotate(dx/(RADIUS*getScale()));
    w2->rotate(dx/(RADIUS*getScale()));
  }
};
```

A car contains a polygonal `body`, and wheels `w1` and `w2` which are instances of a `Wheel` class which is another composite class. Thus composite classes can be nested. The definition of the `Wheel` class is omitted. This code also shows another interesting feature. We have overridden the `forward` function so that the wheels not only move forward, but also turn.

The `Composite` class thus provides considerable power for building interesting animations. But for this, you must understand and use inheritance. Isn't this compelling motivation?

We give two more examples of how graphics can be used to generate programming exercises that may interest our target student audience, i.e. students majoring in science and engineering.

The first example is shown in Figure 3. The goal in this is to trace the path of a ball as it bounces around in a box. In the simplest case (Figure 3), we consider a stationary, rigid box, with the collisions

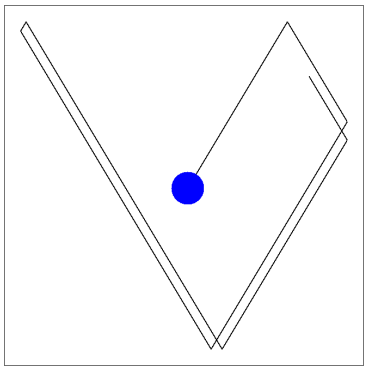

Figure 3: Path of ball bouncing in a box

being perfectly elastic. We can of course make the motion more complex, for example by involving gravity, or friction. Or we can have the box itself move due to the collisions. This can make up for progressively difficult (and interesting) assignments. Note however that even in the simple case above students might find something unexpected: usually the ball will overtrace its path after some time. This example is paradigmatic: it is an example of a *simulation* of a physical system.

Our second example concerns the processing of mathematical expressions. Can you create a library which enables you to represent mathematical expressions, and perform arithmetic or other operations on them? The expressions may or may not contain unknowns. Graphics comes in if you wish to typeset the expression in the standard manner, i.e. generate something like the following:

$$\cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}$$

The key problem in this is to determine the geometric positions of the various subexpressions and the sizes of the bars denoting division. Everyone can generate this layout "by hand", and yet it is an interesting challenge to introspect about how you do it and flesh out an algorithm. The recursion required is quite interesting. In general, since the representation of mathematical expressions is invariably heirarchical (e.g. an expression is a sum of other expressions), it provides fertile ground for using recursion. The problem also provides good motivation for using inheritance.

4.3 Overall pedagogical approach

Our general pedagogical approach is an extension of the ideas implied above. To motivate a new topic, we first find a problem which cannot be solved using what the students know so far. As far possible, we choose problems which are of interest to our students, and which also have a visual aspect, because we believe that pictures help learning. We then present the new topic, and develop complete programs required to solve the problem.

In addition, we believe it is useful to develop some larger paradigmatic case studies e.g. gravitational simulation, representation and solution of resistive circuits, and representation and manipulation of algebraic expressions. The basic theory of the case studies should

be known to the student, although it is useful to review it. We believe that all this synergistically benefits the learning of computer programming and the other sciences, and such examples are better appreciated by students of science and engineering. Overall, we want to convey the impression that you are learning problem solving than just a language.

It may be interesting to note that many books on introductory programming discuss in some detail the historical development of computers. While this history is interesting, it can create the impression that computing began with computers. This is very far from the truth! Manual computation gave rise to many algorithms that are invaluable for computing even today, e.g. Euclid's GCD algorithm, the Babylonian square root algorithm, and even the basic notions of the positional number system and the related algorithms. Our approach attempts to connect to that tradition.

5. EXPERIENCE

A book based on the approach presented here was published by McGraw Hill Education[14]. The Simplecpp library is available from the author's webpage and the publisher's webpage. The book has now been used several times in the introductory programming course in IIT Bombay, and is also being used in Vishwakarma Institute of Technology, Pune. The book was also used in various offerings of a Massively Open Online Course (MOOC)[13].

The author is happy to report that the feedback has been very positive.

6. CONCLUDING REMARKS

We believe that computing must be presented to students as an extension to thought and intuition, as a *universal* skill that applies to all aspects of life. This must be conveyed as *active learning*, i.e. the students should be able to write programs that touch most subjects they have studied till then in their careers. In order to implement this plan, it is necessary to use classroom time efficiently, and ensure that the focus remains on important ideas. We believe that most courses in introductory programming in C++ (as well as several other languages) spend 2-3 initial weeks describing introductory material and without discussing any interesting programs. This we believe dissipates the excitement with which the students start a school term. By using the strategies suggested here (graphics and the `repeat` statement) we believe we can get to interesting material quickly and maintain focus on it.

It may seem that the agenda of getting to many applications in science and technology might conflict with the (primary?) goal of conveying the basic computer programming concepts. However, by choosing the examples carefully, we can in fact motivate even the programming concepts better. We believe our strategy makes for better integration of programming with the prior knowledge of the student.

While the `repeat` statement is present in Logo, we believe that using it in mainstream languages has not been tried before. Besides speeding up learning at the beginning of the course, we believe that students benefit from learning it before learning the standard looping constructs. We believe that many students have difficulty in managing the control variable and the termination condition in standard looping constructs. On the other hand, `repeats` are much easier to understand (even when nested) and thus provide a graded step towards learning the standard looping constructs.

Finally, several approaches use graphics. However we believe no approach (in adult education) has used graphics to start off an introduction to programming: by drawing mathematically interesting pictures. Even adults enjoy drawing; only we need to ask them to draw more intricate pictures than what children would be asked in Logo. We believe we have integrated graphics into basic programming education at a deeper level in general. In addition, note that graphics is very important for our target student group: graphics/visualization and geometric reasoning play a central role in science and technology. And last, but not the least, graphics is fun!

7. ACKNOWLEDGMENTS

The author is grateful to Om Damani for comments.

8. REFERENCES

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] D. J. Barnes and M. Kolling. *Objects First With Java: A Practical Introduction Using BlueJ (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[3] J. Cohoon and J. Davidson. *C++ Program Design: An introduction to programming*. McGraw Hill, 2002.

[4] S. Cooper, W. Dann, and R. Pausch. Alice: A 3-d tool for introductory programming concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, Apr. 2000.

[5] E. W. Dijkstra. On the cruelty of really teaching computing science, 1988. EWD-1036.

[6] A. diSessa and H. Abelson. *Turtle Geometry: the computer as a medium for exploring mathematics*. MIT Press, Cambridge, MA, USA, 1981.

[7] M. R. Hansen and H. Rischel. *Introduction to Programming Using SML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[8] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[10] P. Lawhead, M. Duncan, C. Bland, M. Goldweber, M. Schep, D. Barnes, and R. Hollingsworth. A road map for teaching introductory programming using LEGO. In *ITiCSE-WGR '02 Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 191–201. ACM, 2002.

[11] B. Meyer. The Outside-In Method of Teaching Introductory Programming. In *Ershov Memorial Conference, volume 2890 of Lecture Notes in Computer Science*, pages 66–78, 2003.

[12] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[13] D. B. Phatak and S. Chakraborty. CS 101x Introduction to Computer Programming, 2016. https://iitbombayx.in/courses/IITBombayX/CS101.1xS16/2016_T1. Also 2015. Previous version at https://www.edx.org/course/programming-basics-iitbombayx-cs101-1x.

[14] A. Ranade. *An Introduction to Programming through C++*. McGraw Hill Education, 2014.

[15] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, Nov. 2009.

[16] B. Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley Professional, 1st edition, 2008.