Beautiful and Usable

"The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music." Donald Knuth (The Art of Computer Programming)

"We often feel beauty in simple code. If a program is hard to understand, it cannot be considered beautiful." Yukihiro Matsumoto (the creator of the Ruby programming language)

Robert Sheehan
University of Auckland
Auckland, New Zealand
r.sheehan@auckland.ac.nz

Stephanie Sheehan
Unitec
Auckland, New Zealand
ssheehan@unitec.ac.nz

Abstract
This paper is concerned with two of the workshop questions: what are the precursors to computer science skills and what are the relationships between programming and computational thinking? Firstly, we suggest that a precursor of programming is the appreciation of beauty and elegance. Secondly, even though there is a strong feeling among computer scientists that computer science is not programming we think that separating programming from computational thinking reduces computational thinking to a subset of problem solving and it loses its reason for being treated as an important component of education. Linking both of these questions is the problem of producing a programming environment for children which is both beautiful and usable.

Author Keywords
Children and programming; Computational thinking; appreciation of beauty

ACM Classification Keywords
D.3.2 Language Classifications - *Specialized application languages*

Beauty and Elegance
A very important part of being human is our ability to experience beauty. This produces feelings of awe, causes us to reflect on who we are and what a wondrous universe we inhabit, and to connect at a deep level with others. This is not something we commonly associate with computer science or programming. We will avoid discussing the problem of beauty [1], instead we will assume that we all have concepts of beauty. Some of those concepts can be explained by symmetry, consistency, clarity, order, simplicity, harmony etc.
As in the quote from Knuth we wish to concentrate on computing and programming as an art, one in which beauty can be discovered and produced. In the same way that mathematics can only be fully appreciated and understood when we consider creativity and beauty [2] we wish to claim the same thing for computer science and programming.
There are some heuristics to recognise beauty in computer programs. They include: utility, correctness, readability, efficiency and scalability [3]. Another approach is looking for the elegance of the representation. Many computer scientists consider programming languages elegant if they generalise well and enable efficient encodings of abstractions. This view sees elegance as removing redundancy [4]. Loops are better than inline duplications of code, abstracting

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))
  )
)
```

Figure 1 – a factorial function in Scheme

procedures and classes is better than not. A related view is that elegance is associated with simplicity and consistency. In this view for example a simple syntax which can be extended to cover a wide range of programming structures or paradigms is seen as very elegant. A good example of this is the Scheme programming language (Figure 1).

One traditional computer science definition of elegance is that of the efficiency of the encoding of code and data. Efficiency here is defined as the power of the representation divided by the size of the representation. This form of efficiency as an objective measure rates languages and programs with high abstraction ability as better than those without.

The appreciation of this type of beauty is beyond children as programmers; it may well be beyond many professional programmers. We would argue that an appreciation of beauty in the natural world, in painting, sculpture, music, poetry and literature and an understanding of why we consider things beautiful can be developed and fostered in children [5] and doing this will help them become more creative and artistic as problem solvers [6]. In turn this will assist children in appreciating and creating beautiful digital artefacts.

Problems with concentrating on the mechanical

If we don't see beauty as an important part of computer science we end up with a utilitarian rather than a humanistic justification for its study. Part of the problem with computer programming is that it forces humans to think like computers. As Papert said in 1980 [7] the purpose of solving problems with Logo was to help children "to think like a computer". Processes to solve problems need to be broken down into small chunks that a machine can perform using its limited instruction set.

There are ways around this. We can build libraries of higher level processes which we can compose to create solutions for restricted ranges of problems. By restricting the range we make it simpler to provide the necessary tools to work in that domain. In this way the building blocks are more human scale. The computer languages and environments which are most successful in leading children into programming have taken some aspect of a child's world and provided the tools to deal with that aspect. Two notable examples of this are the Logo turtle and Scratch's control of 2D sprites [8].

These examples have demonstrated that we can produce programming environments which do something children will engage with. They make it easy to manipulate objects that children can understand and want to play with.

Environments such as Logo and Scratch can hence be used to teach programming to children and hopefully help us to convey the aspects of Computational Thinking which we are interested in. This is wider than the ability to produce computer programs alone [9]. If these aspects cannot, at least in theory, be implemented on a computer then there is no reason to call them Computational Thinking. We model something from the real world in such a way that it can be manipulated by a computer to solve our problem.

Here we are looking at the second question: the relationship between Computational Thinking and programming. In order for Computational Thinking to be more than a particular style of problem solving, the solutions it provides must be able to be embodied in an executing program. Unfortunately doing this is difficult.

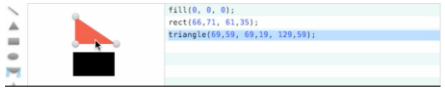

Figure 2 – selecting a figure on the left inserts both a default drawing object in the preview screen and fills in the corresponding code – from worrydream.com

Low-threshold high-ceiling

When designing programs there is a tradeoff between power and simplicity. More options and increased flexibility mean more information is required from the user in order to select the options and provide the data necessary to carry out the user's intentions. Also the structure of programs gets increasingly complex as the size of the programs grows.

Papert was probably the first to employ the concept of low-threshold high-ceiling in the design of a computer system [7]. The idea is to make the entry threshold to a system easy for a novice but to allow that system also to be used productively for advanced users, the high-ceiling. Papert and his collaborators used this as the central design principle of the Logo programming language. In this he was more successful than most users of Logo were ever aware. Logo was a form of Lisp [10] one of the first and most flexible programming languages. Because of this Logo was and is completely capable of being used as a general purpose programming language and can be used in a very sophisticated manner [11]. So the ceiling for Logo is effectively the sky.

Of course one of the difficulties with Logo is the fact that it is programmed with freeform text. This adds all of the problems associated with syntax errors. Not only does a child have to get the structure of the language correct but each individual word has to be spelt correctly and all punctuation has to fit the demands of the syntax. This is even before the program statements can be shown to be logically correct or not. Various attempts have been made to solve this problem all the way up to Scratch which makes syntax and spelling errors impossible by utilising drag and drop.

Ways forward

Computing is a young subject. We are still working out better ways of interacting with our computers. We find the suggestions of two quite different and yet related approaches congruent with the goals and difficulties above. Bret Victor [12] is well known for his original take on art, communication and computers. His web site worrydream.com includes a large collection of ways we haven't so far taken to make the things we do with computers more fitting to humans. Many of these include ways to make programming more accessible and pleasurable. A major concern of his is to engage other human capabilities when programming e.g. thinking visually as well as thinking linguistically. Some of this goes in the direction of reducing Norman's gulfs of execution and evaluation [13].

Victor suggests that getting immediate feedback on programs is important for novices. Thus he recommends quick autocomplete with useful default values and visible and continuously updatable state (Figure 2). "Getting something on the screen" as he puts it should be easy and then programming can progress by reacting to the current state.

Jonathan Edwards was motivated by some of the same concerns as Victor. He uses the idea of programming by reacting in his Subtext programming language [14]. He focusses on using the computer to do the things that computers do well and enabling the users to do the things humans do well. Currently when we read code we have to become a computer and execute the code in our heads. The Subtext programming language and environment looks very much like a traditional language and IDE but it is not. The code is not a string of text which both we and the computer have to interpret. Instead it is a live representation of an execution tree of code and data. This means that code

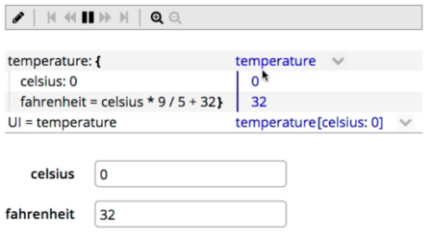

Figure 3 – continuously evaluated code in Subtext with values and UI showing

is created by copying parts of that tree, so most syntax problems are avoided. More importantly to understand how values in the program were produced the programmer moves around inside the tree and can follow the flow of information in a way that reduces the cognitive load (Figure 3).

Conclusion

We have suggested that computer science as an art is a place where beauty can be discovered and produced. The experience and knowledge of beauty can help us become more creative and artistic as programmers. We aspire to work characterized by elegance and simplicity. Combining the approaches discussed above into a drawing, game, simulation, story telling program for children would make programming easier to engage with, easier to understand, and easier to debug and modify. Thus children can more easily translate the worlds in their imaginations into dynamic and interactive digital realities. It would also be possible to turn this into a "real" programming language which would have the sky as its ceiling just as Logo did. It may also be beautiful.

References

1. Scruton, R. Beauty: A Very Short Introduction. Oxford University Press, 2011.
2. Whitcombe, A. Mathematics Creativity, Imagination, Beauty. Mathematics in School, 17, 2 (March 1988), 13-15.
3. Oram, A. and Wilson, G. Beautiful Code: Leading Programmers Explain How They Think. O'Reilly Media, 2007.
4. Wolff, J. G. Simplicity and Power – Some Unifying Ideas in Computing. The Computer Journal, 33, 6 (January 1990), 518-534.
5. Cohen, S. Children and the environment: aesthetic learning. Childhood Education, 70, 5 (October 1994), 302-304.
6. Isbell, R. and Raines, S. Creativity and the arts with young children. Cengage Learning, 2012.
7. Papert, S. Mindstorms : children, computers and powerful ideas. Harvester, Brighton, 1980.
8. Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. The Scratch Programming Language and Environment. Trans. Comput. Educ., 10, 4 (November 2010), 1-15.
9. Barr, V. and Stephenson, C. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? ACM Inroads, 2, 1 (March 2011), 48-54.
10. McCarthy, J. LISP: a programming system for symbolic manipulations. In Proceedings of the Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery (Cambridge, Massachusetts, 1959). ACM, 612243, 1-4.
11. Harvey, B. Computer Science Logo Style. The MIT Press, 1997.
12. Victor, B. Humane representation of thought: a trail map for the 21st century. In Proceedings of the Proceedings of the 27th annual ACM symposium on User interface software and technology (Honolulu, Hawaii, USA, 2014). ACM, 2642920, 699-699.
13. Norman, D. A. The design of everyday things. Basic books, 2002.
14. Edwards, J. Subtext: uncovering the simplicity of programming. SIGPLAN Not., 40, 10 (October 2005), 505-518.