Iconic Programming for Flowcharts, Java, Turing, etc

Stephen Chen
Information Technology Program
York University
4700 Keele Street
Toronto, Ontario
sychen@yorku.ca

Stephen Morris
Computer Science Department
Dr. Norman Bethune Collegiate Institute
200 Fundy Bay Blvd
Scarborough, Ontario
stevemorris@rogers.com

ABSTRACT
One of the largest barriers to learning programming is the precise and complex syntax required to write programs. This barrier is a key impediment to the integration of programming into the core curriculum of general high school science courses – there is not enough time to learn both syntax and programming in a three-week course module. The newly developed "Iconic Programmer" allows executable programs to be written through mouse clicks and menus, includes symbol by symbol translation into Java and Turing, and comes complete with a three-week lesson plan suitable to new programmers. To date, the new tool has been used effectively with full-semester, introductory programming courses at both the university and high school level.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms
Human Factors

Keywords
CS0, CS1, high school programming, non-majors, visualization tools

1. INTRODUCTION
The increasing pervasiveness of computers in modern society is raising the status of computer programming towards the level of being a core topic for any general education program. This rising status supports Alan Perlis' belief that any general, liberal education would benefit from a foundation that includes computer programming since programming involves the concept of processes [4]. Unfortunately, traditional introductory programming courses at the university level have notoriously high attrition rates [7][9] and discouragingly low non-major appeal [5]. These negative features are part of the factors that keep computer programming out of the general education foundation of high school curricula.

Another negative factor is the high complexity of teaching introductory programming [2]. The heavy syntax and technology overhead (e.g. text editors and compilers) can also lead to an undesirable mix of delayed gratification and technophobic anxiety. To address these complexity issues, a large number of educational tools (e.g. [3][10][11]) has been developed. However, many of these tools are still aimed at full-semester, university-level courses.

A university's introductory programming course can arguably be taught in high school (e.g. Advanced Placement), but this will not necessarily increase the number of students that will be exposed to computer programming. Further, since prior programming experience is a key indicator of future programming success [1][6] (and interest?), there is still cause to push the introduction of programming earlier and deeper into the high school curriculum. Ideally, computer programming could become a standard general science topic (like the periodic table, the scientific method, plant biology, etc) presented to all high school students.

To the same extent that the periodic table is not all of chemistry, the first introduction of programming does not have to be complete – just self-contained. An easily self-contained module (and that which speaks best to the concept of processes) is the design and development of algorithms. When teaching/learning algorithms, a useful visualization and design tool to use before coding is the flowchart. The following tool allows flowcharts to be taught as a self-contained module without the need to discuss code, compilers, or any other complexity.

The "Iconic Programmer" is an interactive tool that allows programs to be developed in the form of flowcharts through a graphical and menu-based interface. When complete (or at any point during development), the flowchart programs can be executed by stepping through the flowchart components one at a time. Each of these components represents a sequence, a branch, or a loop, so their execution is a completely accurate depiction of how a structured program operates. To solidify the concept that flowcharts are real programs, the developed flowcharts can also be converted into Java or Turing (present capability), or and other high-level language (easily extendable).

2. ICONIC PROGRAMMING
The key to making programming more accessible is to simplify it as much as possible without losing its essence. From the perspective of understanding processes [4], the essence of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'05, June 27–29, 2005, Monte de Caparica, Portugal.
Copyright 2005 ACM 1-59593-024-8/05/0006 …$5.00.

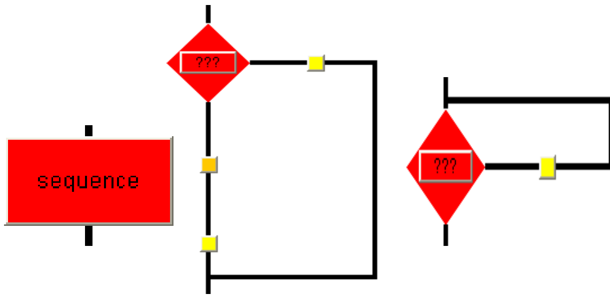

Figure 1. Icons for sequence, branching, and looping.

programming is algorithm development which can be simplified to the three structures of sequence, branching, and looping. Each of these structures can be represented by a flowchart icon (see figure 1). Iconic programming will then be performed by assembling these flowchart components into a program.

In assembling these components, it is clear that branching will allow decision making and that looping will allow repetition. However, it is less clear what sequence can represent. In order to simplify programming as much as possible (and to allow the emphasis to be placed on the process control concepts of branching and looping), sequence icons are restricted to three actions: declare a variable, make a variable assignment, and produce output (see figure 2). (In the current implementation, these actions refer primarily to integer variables, but LEGO robot commands may represent a worthwhile extension in the future.)

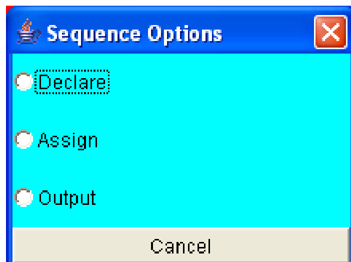

Figure 2. Sequence options.

Declaration allows new (int) variables to be created and output allows them (or text) to be displayed to the user. In between, assignment allows variables to receive and update their values. Again, there are three options for assignment: a random value, the result of a mathematical expression, and user input (see figure 3). The mathematical expression is built using pull-down menus, so no coding or syntax is required (see figure 4).

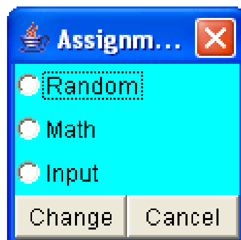

Figure 3. Assignment options.

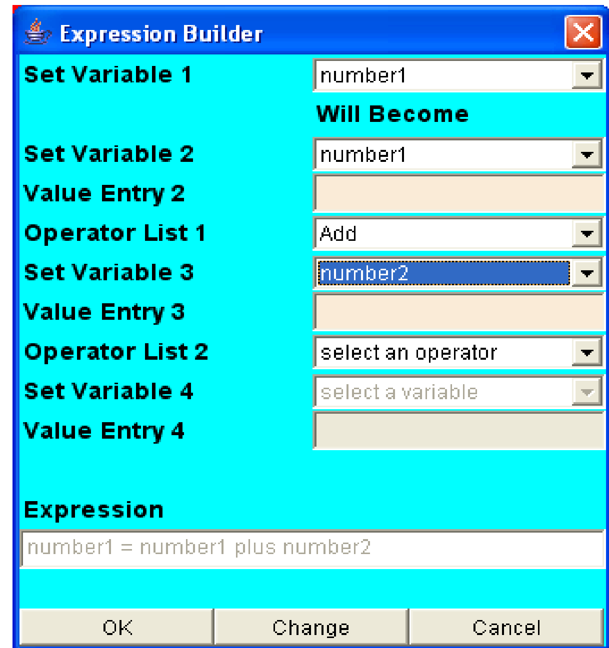

Figure 4. Building mathematical expressions.

With the above actions and options for sequence, there is enough functionality to develop complete programs for interesting, real-world situations. For example, a simple activity may involve trying to guess another person's number that is between 1 and 10. Although simple to do in live action, the ability to translate these instinctive actions into a formal algorithm is not trivial for novice programmers. The Iconic Programmer isolates this essential aspect of programming and allows it to be taught through an intuitive and visually appealing graphic interface.

3. LESSON 1: THE COMPONENTS

The three components of structured programming are sequence, branching, and looping. Basically, a computer can perform an action, make a decision about which action to perform, or repeat an action. Each of these component pieces can be discussed in the context of the number guessing activity.

The actions for the Iconic Programmer are the input, output, and assignment of integer variables. Therefore, the initial number can be input or assigned randomly, and guesses can be input as well. However, with only sequence actions, there can be no feedback or interactivity. Since, it's no fun to guess someone's number if they don't respond to your guess, some decision making ability is clearly required by a computer.

A decision is a choice between two states – the number was guessed, or the number was not guessed. The ability to make such a decision depends on a Boolean expression that is composed of relational comparisons (see figure 5). After the decision, the branching structure allows two paths with different actions to be taken. The program can now interact with a user and provide feedback on their guesses. However, if you have three guesses, what happens when you are right on your first guess?

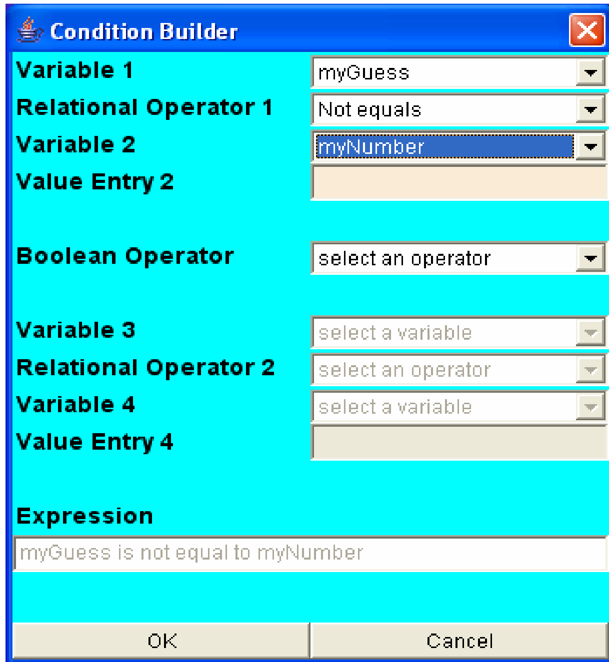

Figure 5. Building Boolean expressions.

The final component allows repetition. Based on a decision, a path that loops "backwards" to return to the decision can be followed. For example, a simple strategy of guessing from 1 to 10 consecutively until the number is found can be implemented. In a CS101I style course [2] that uses the Iconic Programmer as a supplemental tool, control structures and Boolean decisions can be presented in about 1 week (3 lecture hours). With the

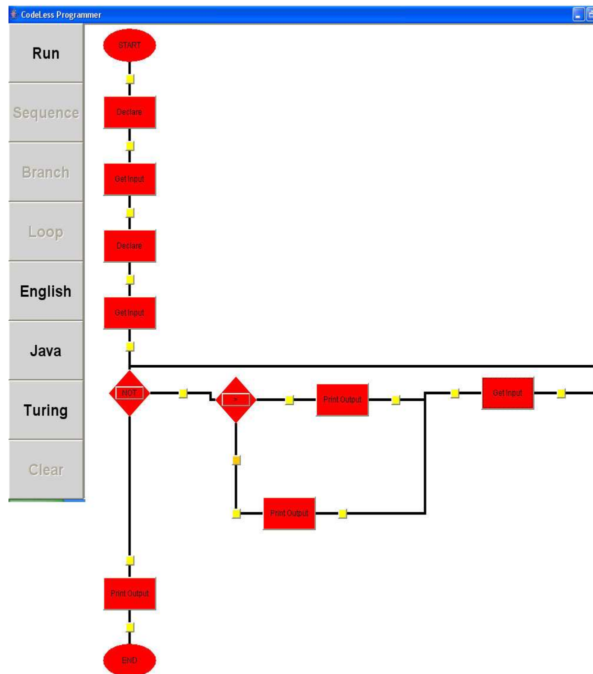

Figure 6. A complete program for number guessing.

elimination of the time required for syntax and other details, it is conceivable that the above concepts could also be presented in about 1 week to a general population of high school students.

4. LESSON 2: A COMPLETE PROGRAM

The number guessing activity can be expanded into a situation suitable for a complete program. Specifically, after each guess, the computer could respond "higher" if the guess is too low or "lower" if the guess is too high. The above problem now encompasses all three structures of sequence, branching, and looping.

The students can be led through the algorithm design tasks by asking leading questions (e.g. [8]). What structure do you need to take multiple guesses? What structure do you need to choose a response? What structure do you need to output a response? The answers to these questions can then be used to build and run an actual program (see figure 6 and figure 7).

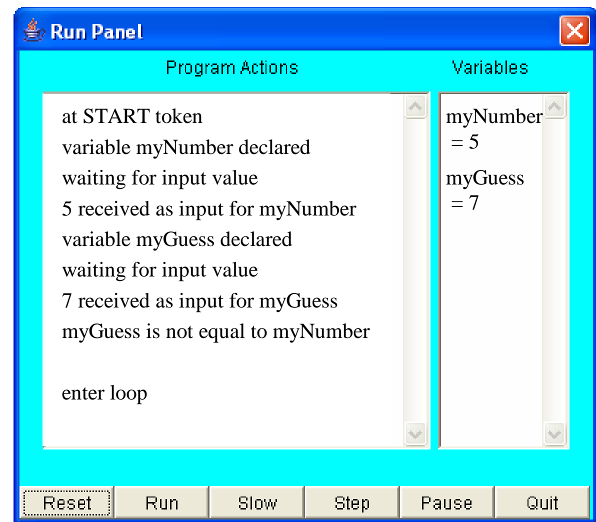

Figure 7. Running flowcharts.

Rather than syntax and coding, the Iconic Programmer allows students to focus on the design and development of algorithms. However, unlike pseudocode [8] and other paper designs, these algorithms provide real feedback and instant gratification through their ability to be executed. The underlying processes to number sequences (e.g. Fibonacci numbers), dice games (e.g. craps), user interfaces (e.g. telephone banking), etc can now be effectively explored during the second week of lectures.

5. LESSON 3: EXTENSIONS OR DETAILS

To ensure that students appreciate the legitimacy of the programs that they can write with the Iconic Programmer, it is important to demonstrate them in a second environment. At present, this demonstration can be done by clicking the "Java" button, cutting and pasting the generated code (see figure 8), and compiling this code to execute with the Java Virtual Machine. In the future, a LEGO robot option may be possible to demonstrate that sequence, branching, and looping are the basic structures behind all computing processes.

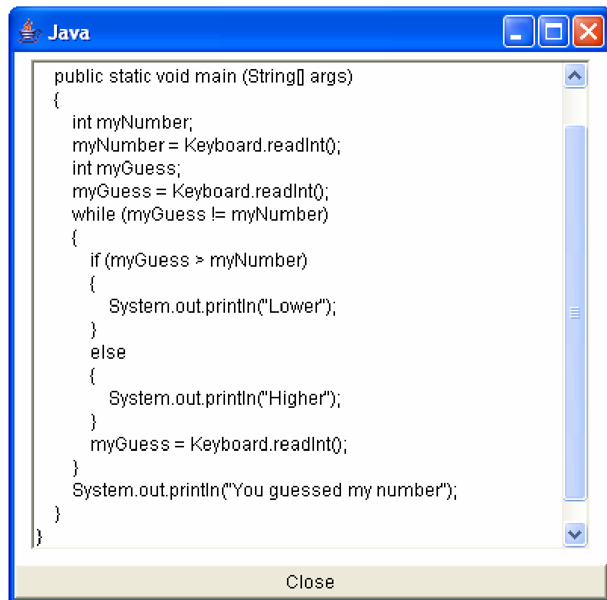
```
public static void main (String[] args)
{
   int myNumber;
   myNumber = Keyboard.readInt();
   int myGuess;
   myGuess = Keyboard.readInt();
   while (myGuess != myNumber)
   {
      if (myGuess > myNumber)
      {
         System.out.println("Lower");
      }
      else
      {
         System.out.println("Higher");
      }
      myGuess = Keyboard.readInt();
   }
   System.out.println("You guessed my number");
 }
}
```


Figure 8. The number guessing flowchart in Java.

Conversely, the more subtle details of the previous lessons can be handled in greater depth. For example, Boolean algebra, binary numbers, and assembly language programming are all used in the current CS101I course to put the concepts into context. Regardless of the choice, these three lessons/weeks form a complete, self-contained module that should be able to provide an adequate introduction to computer programming for a high school general science audience.

6. THE ICONIC PROGRAMMER IN PRACTICE

The Iconic Programmer has been used as a supplemental tool in a CS101I style course at York University (ITEC1620 Object-Based Programming). This course makes heavy use of flowcharts as a visualization tool to help students learn the concepts of control structures and algorithm development. The Iconic Programmer is used to demonstrate the equivalence between programming and drawing flowcharts – the flowcharts are executed and/or converted into code.

The Iconic Programmer is also being used in high school computer programming courses at Dr. Norman Bethune Collegiate Institute. The student response has been very enthusiastic. Whereas drawing flowcharts on paper was a tedious and "irrelevant" activity, the students can now see their purpose, and they enjoy interacting with them. With this positive first introduction, we hope to be able to move the tool beyond elective programming courses and into required general science courses.

7. CONCLUSIONS

The Iconic Programmer is an introductory programming tool designed primarily for use in a general (e.g. grade 9) high school science course. It has been designed to help isolate the key concepts (e.g. algorithm design) of programming away from the less important details (e.g. syntax and coding). With this isolation, it should become possible to make computer programming a self-contained, three-week course module. Classroom success in university and high school introductory programming courses suggests that this objective is achievable.

8. REFERENCES

[1] Cantwell Wilson, B., and Shrock, S. Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education* (2001), ACM Press.

[2] *Computing Curricula 2001: Computer Science*, December 2001. Online [September 1, 2002]. Available at http://www.acm.org/education/curricula.html.

[3] Cooper, S., Dann, W., and Pausch, R. Teaching Objects-first in Introductory Computer Science. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education* (2003), ACM Press.

[4] Greenberger, M. *Computers and the World of the Future*. Transcribed recordings of lectures held at the Sloan School of Business Administration, April, 1961. MIT Press.

[5] Guzdial, M., and Forte, A. Design Process for a Non-majors Computing Course. To appear in *Proceedings of the Thirty-sixth SIGCSE Technical Symposium on Computer Science Education* (2005), ACM Press.

[6] Hagan, D., and Markham, S. Does it Help to Have some Programming Experience before Beginning a Computing Degree Program? In *Proceedings of the Fifth annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, ACM Press.

[7] Hermann, N. Popyack, J., Char, B., Zoski, P., Cera, C., and Lass, R.N. Redesigning Computer Programming using Multi-level Online Modules for Mixed Audience. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education* (2003), ACM Press.

[8] Lane, H.C., and VanLehn, K. Coached Program Planning: Dialogue-Based Support for Novice Program Design. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education* (2003), ACM Press.

[9] Nagappan, N., William, L., Ferzil, M., Wiebe, E., Yang, K., Miller, C., and Balik, S. Improving the CS1 Experience with Pair Programming. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education* (2003), ACM Press.

[10] Pattis, R., Roberts, J., and Stehlik, M. *Karel the robot: a gentle introduction to the art of programming, 2nd Edition* (1994), John Wiley & Sons.

[11] Sanders, D., and Dorn, B. Jeroo: A Tool for Introducing Object-Oriented Programming. In *Proceedings of the Thirty-fourth SIGCSE Technical Symposium on Computer Science Education* (2003), ACM Press.