Programming as a Second Language

By Justin Solomon

Subject: Computer science, foreign language

Grades: K–12 (Ages 5–18)

Technology: Programming

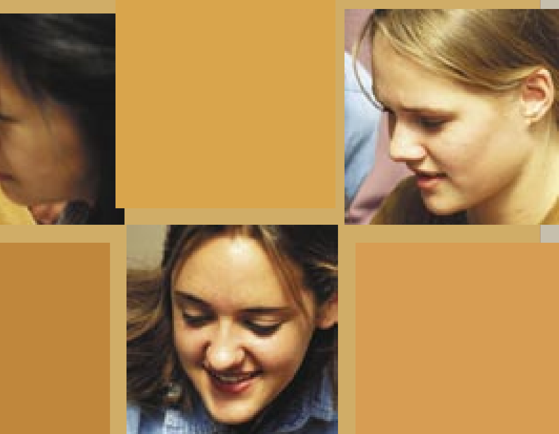
Standards: *NETS•S* 1, 3; *NETS•T* II; *NETS•A* I (http://www.iste.org/standards/)

Understanding the concepts and development of programming can benefit students by demanding precision of thought and requiring students to clarify and improve on their problem-solving process. The goal of beginning programming instruction should be to introduce the subject in terms familiar to students.

Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.

Standard methods of teaching an introductory course in computer science, designed to introduce computer programming as a tool for mathematicians and engineers at the university level, are unnecessarily complicated and difficult. They lack a common thread that unifies each unit of material and frequently make use of mathematical notation, technical keywords, and other terms or symbols unfamiliar to the average middle or high school student. Furthermore, they tend to focus on a *single* more advanced language, leading students to believe that each programming language is a distinct and separate entity with a tenuous—at best—link to other languages. Ironically, traditional high school-level computer science classes are often driven by the Advanced Placement (AP) curriculum dictated by the College Board, which establishes a narrow set of benchmarks for evaluating proficiency in computer programming. According to research done by Allen Tucker, these benchmarks promote memorization over understanding and are of limited practical use to the average student. (***Editor's note:*** Find this and other Resources on p. 39.) As a result, students view programming as more of a snapshot than a continuum, failing to see the value in pursuing a higher level of programming and computer applications proficiency.

To meet the educational requirements of a wider and increasingly younger audience, computer programming instruction needs a comprehensive overhaul, with the goal of fitting into school-based curricula as a "second language," starting as early as the elementary years. To begin the process of revising basic computer science curriculum, the goal of beginning programming instruction should be to introduce the subject in terms more familiar to the average student.



With simple adjustments to methods of programming instruction, students could be shown that linguistic terms, such as specific parts of speech or elements of grammar, can be common denominators in computer programming. Using spoken and written language as a metaphor for programming can show commonalities and differences across programming languages, the correlation among these differences, and the repetitive patterns of the correlations. In addition, this concept can be adapted to reduce the intimidation factor of introductory programming, making it accessible to students from all backgrounds, ages, and genders. With this model, introductory courses in computer programming are more likely to provide a solid basis for students in all algorithmic thought, as well as a foundation for progress to related applications and extensions, possibly including more advanced languages with specific advantages and particular uses.

According to research conducted by Letitia Naigles, students introduced to foreign language at a young age find it easier to acquire proficiency in that language. The same holds true for computer programming. Although programming has become an important subject for high school students to learn, there is no reason it cannot be introduced earlier. As a discipline, it provides students with a unique way of thinking based on logic and reasoning. A rudimentary understanding of the concepts and development of programming can benefit students by introducing them to such topics as stepwise refinement and the use of documentation. It demands precision of thought and requires students to clarify and improve on their problem-solving process. Programming has applications in almost any field, from lan-

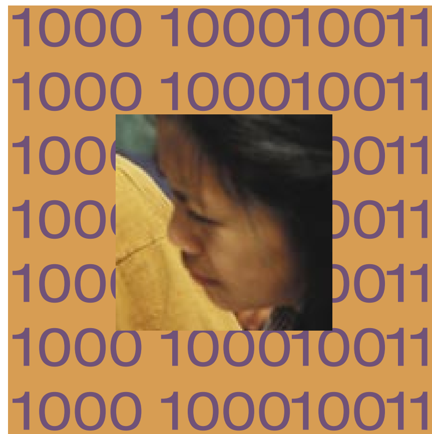
Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.

guage arts and design to architecture, and could be viewed as the basis for any type of computer use. Even the most fundamental knowledge of programming can prove helpful to students working on multimedia presentations, classroom projects, spreadsheets, word processing, or data aggregation.

Unfortunately, the somewhat archaic model for lessons in programming tends to target older, higher-level students who already understand the abstractions and jargon used by computer technicians. This frequently leads to discouragement and failure among younger students who might otherwise benefit from knowledge of computer programming. Chris Stephenson's recent survey of high school computer science teachers reported that many of the schools that do not offer computer science instruction decided that it was a field best left to colleges and universities due to "academic rigor." This notion is unfounded. Many students in both elementary and middle school are proficient in programming, even though they do not have the traditional technical or mathematical backgrounds.

According to Peter Van Roy's basic constructivist model of education, students who study computer science as an extension of what they already know tend to gain a deeper understanding of the subject material. Also, Nicola Henze and Wolfgang Nejdl found that students who learn answers by rote are less likely to understand or apply them. For these reasons, a common-sense model of computer science education would use something any student can understand: language. One must learn the basic constructs of a language and how to manipulate them to be proficient in any language, be it designed for communication among computers or humans. If a computer science curriculum encouraged students to make connections between computer languages and the languages they speak, students would have greater ease in remembering how to code as well as fewer syntax errors. Drawing connections between program syntax and punctuation, between language structure and grammar, students can recognize programming problems in their everyday lives, making abstract topics much more concrete. They will have the ability to take an algorithmic and logical approach to problem-solving both inside and outside the technical realm.

After they make these connections, students will be able to communicate in the language of the computer how problems can be solved in the most



efficient way possible. Programming will be much easier to understand and it will become less problematic to retain the large amounts of knowledge needed to write useful programs. Integrated into other classroom work, computer programming can be a useful, creative, and thoroughly entertaining second language for students at all levels.

First Words

According to the American Speech-Language-Hearing Association, after simple nouns, the first words typically used by infants are commands. Straightforward and largely uncomplicated, these commands are just enough for a young child to convey his or her needs and desires. Unsurprisingly, the vocabulary and structures used to form these demands are usually quite direct, as most toddlers do not have a deep understanding of complex grammatical structures.

Programs written by novice computer science students follow the same pattern. Beginners find it hard to accept that certain statements are necessary without understanding their purpose. For example, consider the following program written in Java, a fairly complex programming language popular for introductory classes in computer science:

```
import javax.swing.
JOptionPane;

public class SimpleProgram
{
  public static void
main(String a[]) {
    String answer;
    answer = JOptionPane.s
howInputDialog("What is the
password?");
    if (Integer.parseInt(a
nswer) == 123)
      JOptionPane.showMess
ageDialog(null, "Correct");
    else
      JOptionPane.sh
owMessageDialog(null,
"Incorrect");
  }
}
```

This code is hardly readable. Most beginning programmers do not know how or why methods are declared, yet alone what it means to call them *public, static,* and *void.* Also, a single typo in this and most other Java programs could result in any number of cryptic errors, from the simple *unclosed string literal* to the terrifying *thread death exception*. In fact, running the above program with the input "wrongPassword" will make it stop running and output the following error message:

Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.

```
Exception in thread "main"
java.lang.NumberFormatExce
ption: wrongPassword
  at java.lang.Integer.par
seInt(Integer.java:426)
  at java.lang.Integer.par
seInt(Integer.java:476)
  at SimpleProgram.main(Si
mpleProgram.java:7)
```

Almost every type of simple error comes with its own confusing name and "exception." Forced to explain such obscure packages and "features," Java reference manuals are often dauntingly long for the average student. Many times these errors are so long they scroll off the console or output window. Now observe a similar program in BASIC:

```
PRINT "What is the
password?"
INPUT password
IF password = 123 PRINT
"Correct"
ELSE PRINT "Incorrect"
```

or its counterpart on a TI-83 calculator:

```
PROGRAM: PASSWORD
ClrHome
Disp "PASSWORD?"
Input P
If (P=123)
Disp "CORRECT"
If (P≠123)
Disp "INCORRECT"
```

These programs have one-third to one-half the number of lines of their counterpart in Java and are much easier to read and to write. They look closer to a set of directions given from one person to another so that they could be read from top to bottom without much interpretation. Also, every command has a clear purpose, making the debugging process straightforward. No "keywords" are required to precede certain statements, and the programmer does not need to surround blocks of code with any type of bracket or brace. Everything runs from the beginning to the end, one line at a time. The immediate feedback provided by writing short programs and observing their output helps students recognize the importance of every set of programming skills in the context of their own computer needs. This prevents students from memorizing code segments without understanding how they can be used.

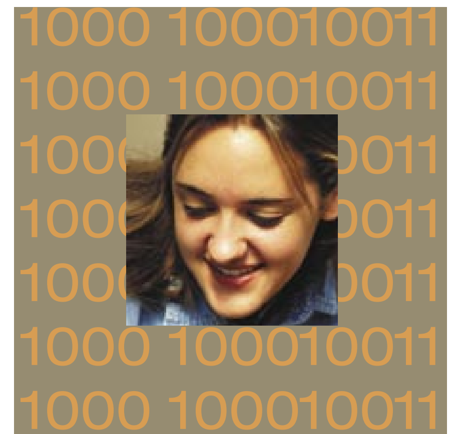
Differences in readability alone are enough to make a compelling argument for the use of simple interpreted languages by beginners. Several such interpreters are freely available or for a small price on the Internet. These languages, including any member of the BASIC family, make it easy for programmers to focus on the different structures, commands, and algorithms without concerning themselves over the form or style of their programs. They encourage the development of good programming style and are similar to the pseudocode used in more advanced textbooks. Also, Marcelo Zanconi and his colleagues found that the use of simple programming languages encourages teaching the programming process instead of the features and peculiarities of a particular language.

Once students have learned basic syntax rules, they should be able to read their programs like normal text. An effective exercise would be to predict the output of programs before running them and to "translate" programs from computer language to everyday language and back. This type of work emphasizes the relationship between programming and real-life situations. It also will encourage the student to recognize the "punctuation" of coding, and that there are several elements that make a program acceptable to the computer but still readable by humans.

Recognizing Verbs and Other Useful Parts of Speech

The necessity for more complex communication drives children to construct longer sentences by stringing more meaningful words together. Despite their utility, commands alone do not convey action, provide a name or label, or describe anything in detail. Thus, new language-based constructs include at least a subject and a verb, and often incorporate modifiers and adjectives.

These terms transfer directly to basic structured programming techniques using subroutines and functions. A subroutine is like a pronoun. Subroutines represent segments of code that can be reused to perform the same calculations or output every time they are called. They encourage new programmers to avoid cutting and pasting similar blocks of code for a more sophisticated model. Similarly, pronouns are used to shorten the names of their antecedents. Pronouns could be replaced with the noun they represent without changing the meaning of their context. In the same way,



Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.

every time a subroutine is called it has the same "meaning" to the computer and produces exactly the same action.

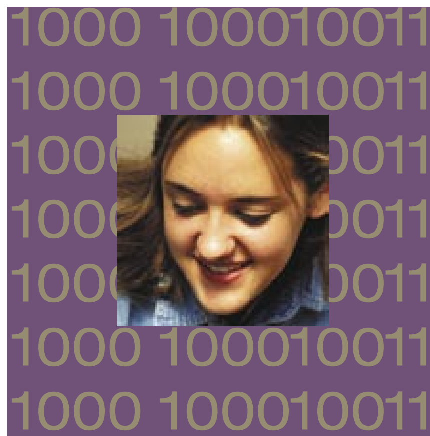
Functions, on the other hand, are more like verbs. Functions, called *methods* in Java, take one or more parameters and somehow manipulate them to eventually return a single value. Similarly, the subject of a sentence affects or changes the object through the verb. Functions change their use and meaning depending on their parameters and can be used repeatedly in the context of a single program. Verbs also change meaning depending on the context of the sentence.

After students have a complete understanding of the programming "sentence structure" and control over simple data structures, it may be time for them to move to a more powerful language that supports such paradigms as object-oriented programming (OOP) and templates. Because they have already studied constructs used in almost any language such as branching and looping, it should not be hard to transfer their skills. Now it should be more obvious, for instance, why the *main()* function exists in C++ or why most Lisp programs start with several function definitions.

Perhaps the best way to introduce students to new languages is to have them write the same program in the language with which they are already familiar and in the new language. As a result, they will notice that different programming languages are used to express the same idea in different ways. Also, they should compare the features of the two languages and hypothesize when it might be advantageous to use one rather than the other. As students will soon discover, the linguistics of most commonly used programming languages are related.

Recognizing Nouns

The next structure covered in most computer science classes is the object. Objects are abstract representations of all the functions, subroutines, and data related to a specific item or concept. This model requires programmers to group their functions and data members by purpose. OOP has become widely popular in recent years and has led to many difficult lesson plans. In fact, it has even been cited as a top reason why students drop their programming courses. Other than the traditional "black



box" lesson about objects and classes, there are hardly any resources available to help teach this particularly abstract topic.

OOP, however, could be viewed as a programming methodology that focuses on nouns rather than on verbs. To relate objects to nouns, students could list all of the actions, information, and manipulations associated with a certain noun. Then, they could combine their prior knowledge of programming techniques and simple algorithms with a short lesson on the syntax of objects and classes to actually implement a program that, on the most basic level, acts like the noun they chose. From there, students can work on progressively more difficult problems, each focusing on the fact that an object should be made for every prominent noun in the program description. Soon, they will be able to apply object-oriented design to applicable programming situations and to understand the concepts of and reasons for encapsulation.

At this point, more advanced concepts of OOP such as inheritance, polymorphism, and abstraction can be introduced individually by relating them to linguistics. Inheritance deals with the classification of nouns and their related verbs. As specific nouns are grouped together, they will form categories, which can be represented by abstract objects. For instance, "triangle," "square," and "pentagon" could be classified under the more abstract noun "shapes." Such simple relationships show naturally the need for such complex models as polymorphism, which deals with treating all child classes the same by using the member functions common to all of them. For shapes, functions that calculate area and perimeter could be called without knowing which specific shape the computer was referencing.

Communicating Ideas

No language is useful unless it can be used to communicate ideas. After all, it is not the individual words that give a sentence or paragraph its meaning, but rather the specific sequence of words used. Likewise, learning the syntax of a programming language is useless unless it can be applied to the creation of useful, or at least entertaining, programs.

There are several ways to approach the problem of creating sample situations that could be solved using a computer program. Regardless of the

Reach for the Stars of education! Advertise in L&L.
Contact Danielle Steele-Larson or visit http://www.iste.org/ll/about/
to find details on advertising in ISTE's flagship publication.

Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.

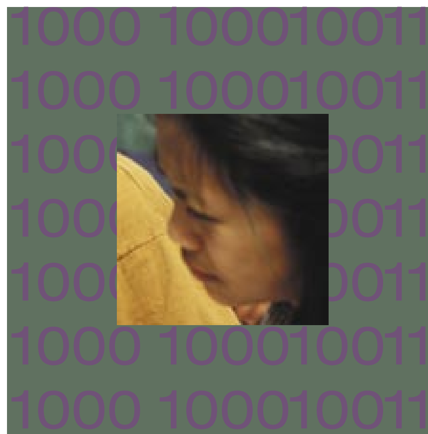
size or type of problem, however, the following principles will hold true:

- Although a problem can be designed to require the implementation of a certain algorithm or structure, the teacher should allow his or her students to discover this on their own and to explore their own ideas. No requirements can be made that a program should be written a certain way, although a runtime or memory-size limit could be enforced. By encouraging students to analyze the benefits and costs of implementing specific algorithms or solutions, they will find out why longer programs or more complex solutions can lead to exponentially faster programs.
- Allow students to work together, but make sure that they all contribute to the programming process. This way, they will all gain programming experience. Student collaboration is crucial in encouraging understanding when teachers have exhausted the traditional means of instruction. Group projects are especially important in OOP, where each member can be responsible for a single component of a larger program.
- Try to assign programs in order of their level of abstraction. Begin with programs that have an obvious implementation and then move on to applications of the specific structure or algorithm and combine with previous lessons. Traditional lesson plans that could benefit from this process would be ones that involve simple algorithms such as searching, sorting, and file manipulation.

These concepts are important not only in connecting the programming world with the real world but also in honing students' generalized problem-solving skills. There are several different ways to word a sentence, and it is important that one can understand any type of sentence structure. Similarly, teamwork introduces students to the different ways in which people approach programming problems and the various styles of writing code. The student should begin to recognize the most efficient structures and make use of them when needed. Teamwork will introduce students to careers related to computer science, where a whole team can work together to write one software product.

Achieving Fluency

With some practice, most students can gain the skills needed to be proficient in a useful programming language. Learning any language is difficult. It takes practice, study, and the ability to apply several layers of abstraction to a concrete problem. A good lesson plan that relates code to everyday life situations can make the experience much easier. Learning to code is a very rewarding process that will provide the student with skills he or she can use for life. If started early enough in a student's academic career and taken out of the context of achieving a higher score on a standardized exam such as the AP test, programming can become as natural as bilingual fluency earned through the study of a second language.



Resources

American Speech-Language-Hearing Association. (1997–2004). *How does your child hear and talk?* [Online document]. Available: http://www.asha.org/public/speech/development/child_hear_talk.htm.

Henze, N., & Nejdl, W. (1998). Constructivism in computer science education: Evaluating a teleteaching environment for project oriented learning. *Interactive Computer Aided Learning Concepts and Applications. Proceedings of the ICL 98 Workshop, Carinthian Institute of Technology, Villach, Austria, October 1998.* Available: http://citeseer.ist.psu.edu/henze98constructivism.html

Naigles, L. R. (2002). Form is easy, meaning is hard: Resolving a paradox in early child language. *Cognition, 86,* 157–199.

Stephenson, C. (2002). High school computer science education: A five-state study. *JCSE Online.* Available: http://www.iste.org/sigcs/community/jcseonline/2002/02/stephenson.cfm.

Tucker, A. (1996). Strategic directions in computer science education. *ACM Computing Surveys, 28*(4), 836–845.

Van Roy, P., (2003). The role of language paradigms in teaching programming. *Technical Symposium on Computer Science Education, 34,* 269–270.

Zanconi, M., Moroni, N., & Señas, P. (1995). An educational project in computer science for primary and high school. *SigCSE Bulletin, 27,* 27–33.

Justin Solomon is a 17-year-old junior at Thomas Jefferson High School for Science and Technology in Alexandria, Virginia. He began programming after the third grade and is currently studying artificial intelligence and computer graphics. Justin recently won first place in the Fairfax County Regional and Virginia State Science and Engineering Fairs for computer science. *He would like to thank Joshua Strong, Karen Budd, Elizabeth Lodal, Anita McAnear, and Philip East for their assistance in the development of this project.*

Copyright © 2004, ISTE (International Society for Technology in Education), 1.800.336.5191 (U.S. & Canada) or 1.541.302.3777 (Int'l), iste@iste.org, www.iste.org. All rights reserved.