Making The Analogy: Alternative Delivery Techniques for First Year Programming Courses

Enda Dunican
Computing Physics and Mathematics Department
Institute of Technology Carlow, Ireland
Enda.Dunican@ITCarlow.ie

Keywords: POP-I.B. barriers to programming, POPIV.B. teaching design

Abstract

This paper discusses some of the fundamental problems encountered by students on first year computer programming courses at Irish Institutes of Technology. It's content is based on the author's practical experience in the classroom and programming laboratories which will be supported in the near future by empirical data gathering. The reasons for many difficulties experienced by students are discussed. The objective of the proposed research is to use the analogy-based approach to learning. A structure called an analogy tree will be built and implemented in a learning tool which will use animation to simulate the relevant analogies. Finally, the future work required, and, potential problems for this project will be discussed .

Problems Encountered

When embarking on the delivery of a first year programming course, lecturers are faced with a number of problems. Firstly, programming students are a product of the primary and secondary educational systems that do not have a logic/problem-solving module in any of their subjects. This puts Irish students at a disadvantage compared to many of their counterparts in other countries where programming is taught at elementary and secondary level. While this problem is primarily outside the control of the lecturers, novel techniques will have to be developed by them to fill this obvious void in order to enhance student's chances of coming to terms with issues of logic and problem solving.

The second major problem relates to the abstract nature of the programming task. Notions such as variables, data types, dynamic memory etc have no counterparts in everyday life, and, grasping these fundamental programming concepts is not straightforward.

Thirdly, the rigid demands of syntax compared to the inexact and loose nature of the English language result in many students not being able to successfully write compilable programs. Many programming lecturers witness this in the classroom where some students are somewhat adequate at developing semi-correct pseudocode but cannot translate it into a syntax error-free block of code.

From practical experience one can identify three categories of novice programming students. The first category are those who do not have the aptitude to understand the basic concepts, this is often a result of a misguided choice of course. The second category are those who may grasp the fundamental concepts if exposed to effective teaching approaches, and the third are those who are fully comfortable with the abstract nature of programming concepts. The main objective of programming instructors is to ensure that most students in the second category progress into the third. This will not happen unless adequate pedagogical techniques are utilised. A major difficulty here is that it takes very few negative experiences at the early stages to disillusion the student. Very often students show a lot of enthusiasm at the start [Kurland 1989] but this decreases as barrier after barrier emerges in the learning cycle.

Potential Solutions

If we are to address the problems identified in the previous section we must look at them in a pragmatic fashion. With a subject like programming, students digest the course material at different speeds and the problems they have are wide and varied. Because of this it must be accepted as a fundamental premise that one mode of delivery using one set of examples and concepts will not work for *all* students. The problem with most programming courses is that they don't provide for individual pacing of the material. Most on-line teaching tools also suffer from this malady. If you don't understand a concept they send you back to the start of the chapter/module again and push you through the same material in the somewhat futile hope that a little bit more of the same material might filter through on each subsequent occasion. A cynical piece of code to represent this is illustrated in fig 1

```
Repeat
  Run Section 1 {Same material as last time!}
  Run Section 2 {Same material as last time!}
  Take Test     {Indefinitely}
Until I Eventually Pass {Because I've learned off answers}
```

Fig 1. Cynical Repeat Loop

Taking all of this into consideration we must reinforce the fact that there is no panacea. It is unlikely that we will find a text book or online teaching package that will explain every concept adequately to every student. It is up to programming academics to develop pedagogic approaches that will embrace the following notions:

1 An illustrative concept that works for one student will not necessarily work for the next student. Whatever tools or techniques are developed must be capable of being personalised with respect to both delivery and content.

2 If an illustrative concept is not understood there must be others that can replace it from an alternative perspective.

3 The approach used must draw upon as many different examples and techniques currently available and should be grouped into one delivery package.

Current Research

In an earlier section we focussed on what the students *don't know* when they embark on a third-level programming course. Many textbooks and teaching approaches attempt to cover numerous programming concepts e.g. I/O, data types sorting and searching etc. In most cases, our Irish students will never have been exposed to anything resembling these programming fundamentals before. It is therefore important that the techniques utilised base their illustrative examples on concepts they have seen before i.e. focus on things they *do know* and build upon them. A significant problem in this respect is that we assume that students can understand and break down the components of their thinking process [Mayer 1989a]. This process is exacerbated if the students have been exposed to "rote learning" [Mayer 1989b].

One technique being investigated in this research is the use of *analogy*. Analogy consists of two key concepts i.e. the *source* and the *target*. The source represents a piece of knowledge that one is familiar with. The target refers to the less familiar piece of knowledge. When an analogy is made the features of the source are mapped onto the target (Dunbar 2000). The aim of this research is to pick sources familiar to the majority of students.

Many teaching approaches attempt to introduce notions to illustrate how certain programming concepts actually work. Very often these notions themselves can be as abstract and complex as those concepts they are being used to describe. A common example of this is use of diagrams and memory addresses to describe linked lists. To many students the concept of a memory address itself is abstract. Therefore it seems to be a futile exercise to use one abstract concept to describe another. Part of the current research is to try and build upon existing concepts that students are already familiar with and relate them to programming concepts, i.e. they can form a direct association between the two. In analogy terms pick a simple tangible source and apply it to a fundamental programming concept that will be the target. This approach is similar to design-pattern work in object oriented programming and the notion of "templates" (Linn 1989). It is hoped that a library of *analogy patterns* will be developed to support this work. Furthermore each analogy will have a set of code patterns which can be used to implement it. The remainder of this section describes some analogies being designed in current research.

Analogy 1 – Children's Shape Toy – Assignment Statements

This analogy is the use of children's toys to introduce assignment statements. A very simple source to use is that of a children's shapes toy as illustrated in fig 2.

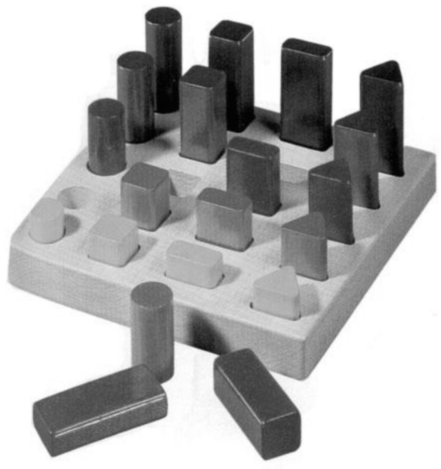
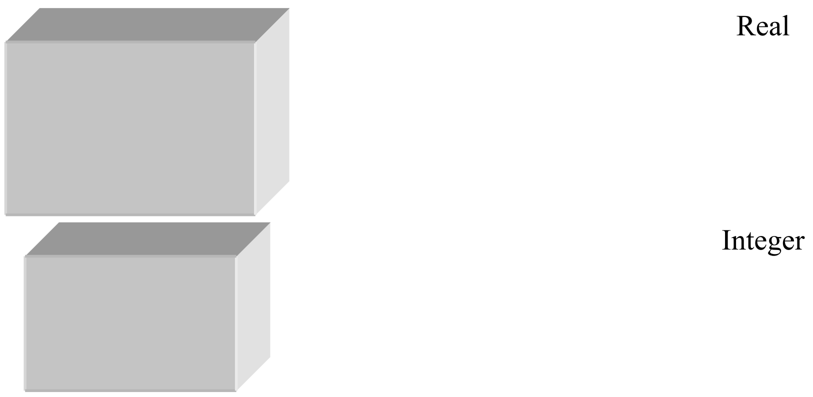


Fig 2 Children's Shape Toy

In the authors experience one of the most common problems experienced by first year students is with the fundamental programming concept of data types, and in particular difficulties in relation to assignment statements and type mismatches. This is especially evident in Pascal which is a strongly typed language. In figs 2 and 3 we can use a classic children's toy to illustrate how assignments statements work. An integer shape can be stored in an integer hole or a real hole. However a circle shape(whether it is a string or a char shape) can't fit into either a real or integer hole. On the other hand a char shape can fit into a string but not vice versa . This can be enhanced with the use of colour for each block. With analogies like these we are considering introducing students to them in a classroom devoid of computers, programming languages and compilers. Once they have fully grasped these concepts they can take them into the programming lab and directly relate them to well-chosen lab exercises.



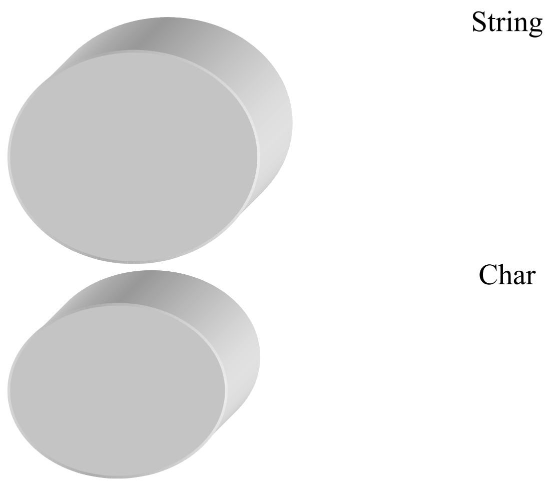


Fig 3 Shapes representing data types

Analogy 2 – Paper & Boxes – Highest and Lowest Number

This analogy is used to solve a typical problem that consists of keyboard I/O and elementary logic to take a list of numbers and determine what the largest and smallest number generated. Three boxes will be used, holding the current, largest and smallest numbers respectively. In the computer–free classroom these will be A4 boxes with labels as in fig 4. The lecturer will write a number on a sheet of A4 paper and put it into the box label "current". Comparisons will be made between the contents of the box labelled "current" and the boxes labelled "highest" and "lowest". If the boxes "highest" and "lowest" are empty then a copy of the contents of the "current" box is placed in both. The sequence continues until number entry has ceased.

In the computer lab, the use of animation will illustrate the sequence of events, indicating how the paper holding the current number will drop into the appropriate box. When the animation is finished the individual boxes will hold the relevant values and these can be mapped onto variables in a program with corresponding names. This analogy is similar to the "concrete model" developed by (Mayer 1989b) where he builds a model of a file management system using filing cabinets with named drawers and a desk with labelled sorting baskets. A potential problem that will have to be addressed here is the possibility of "missaplication of analogy" (DuBoulay 2000). This refer to the fact that a student believes that since a variable is a box it can hold more that one value at a given time.

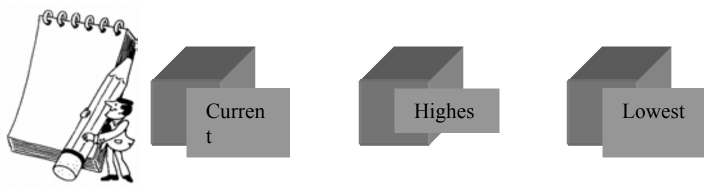


Fig 4. Paper and Boxes Analogy

Analogy 3 – Long Lost Relative Vs Leaflet Distributor

A simple analogy to explain the concept of array manipulation. Searching an array for a specific value can be compared to a long lost relative looking for their cousin. They know the person they are seeking lives in a certain housing estate so they start at the first house knocking on the door and asking if that particular person lives there. As soon as they find the person they are seeking they stop, i.e. they don't need to visit any subsequent houses (code to implement this is a while loop). The use of animation with the scenario depicted in fig 5 can illustrate this. However initialising all array elements can use the same house analogy, but on this occasion using the concept of the leaflet distributor who must visit every house (code to implement this is a for loop). When the animation for both of these two analogies is complete they can be mapped on to simple code implementations :

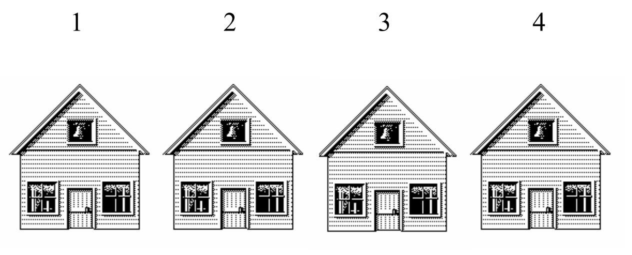


Fig 5 Long Lost Relative Vs Leaflet Distributor Analogy

Long Lost Relative – While Loop :

```
HouseNumber = 1
While (Relative NOT Found) AND ( HouseNumber <= 4 do)
        Move on to HouseNumber + 1
```

Leaflet Distributor – For Loop

```
For HouseNumber := 1 to 4 do
        Drop Leaflet at HouseNumber
```

These represent simple problem solving algorithms that can be best illustrated with the animation of simple analogies.

Development of Analogy-Based Teaching

The research being currently undertaken aims to use both computer and non-computer based implementations of the analogies described in the last section. Non-computer based work will entail the use of tangible objects described in the analogy. Computer-based representations will consist of the animation of the analogies where possible and the mapping of these analogies to simple computer code. Current research is working on the development of an *analogy tree*, which will be the fundamental structure behind the computer-based teaching tool. The structure of this tree structure is quite simple. Each node in the analogy tree will represent an analogy to illustrate a given programming concept. If this analogy does not work (i.e. doesn't adequately explain the concept to the student), or part of it doesn't work, child nodes will be developed that will provide simpler or simplified analogies. A schematic of this analogy tree is represented in fig 6.

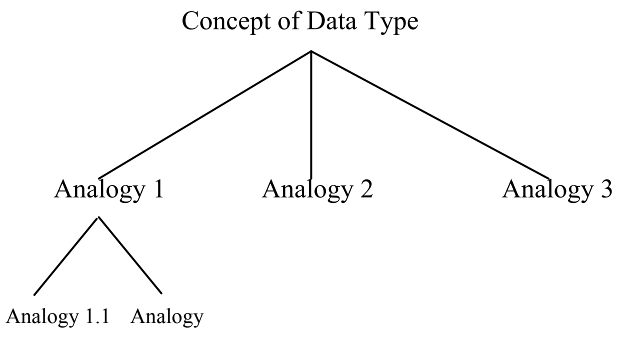


Fig 6 Portion of Analogy Tree Structure

In an ideal scenario many of the analogies at the higher levels will suffice and explain sufficiently the concept under study. For example in fig 6 analogy 1 may be sufficient without having to go deeper down the tree or visit other analogies at the same level. This structure is still only in its design stage and a number of key issues have yet to be investigated :

1 The effectiveness of each analogy, i.e. if it doesn't explain the concept to the student, what part of the required cognitive reaction did it not stimulate

2 How to map each analogy to the relevant area of human cognition. We must ensure that a given set of analogies will explore all possible angles in an attempt to explain the concept fully.

3 How relevant feedback given to the student.

Finally work is also being conducted on compiling a list of the most common mistakes and problems being encountered by students. Developing successful pedagogic approaches will not depend on understanding how many students get it right, rather it depends on understanding how and why many of them make mistakes.

Conclusions and Future Work

This paper has identified the practical problems associated with the delivery of programming courses to first year students in Irish Institutes of Technology. The research on the development of an analogy tree is only in its embryonic stage. A large amount of work has yet to be conducted carried out :

1 A formal study of the major problems that novice students encounter. Empirical data will have to be gathered in the computer programming laboratories followed by various statistical analyses. This data gathering will be conducted on three different languages being taught at the institute Pascal, C and Java.

2 The structure of the analogy tree has to be developed and each analogy will have to classified into one the following categories :

Concept – Explains a fundamental programming notion e.g. a variable

Syntax – Explains syntactical issues for example, e.g. how each open bracket must have a closed bracket

Problem Solving – Explains problem solving techniques e.g. how to sort an array

3. How to overcome many of the purported problems with analogy-based approaches to learning as identified in literature (DuBoulay 1989).

References

DuBoulay, B. (1989). Some Difficulties of Learning to Program. Soloway and Spohrer eds. Lawrence Erlbaum Associates.

Gentner, D. (2000) Perspectives from Cognitive Science. MIT press Cambridge MA.

Kurland, M et al. (1989) The Study of the Development of Programming Ability and Thinking Skills in High School Students. Soloway and Spohrer eds. Lawrence Erlbaum Associates.

Linn, M. Dalbey, (1989) J. Cognitive Consequences of Programming Instruction. Studying the Novice Programmer. Soloway and Spohrer eds. Lawrence Erlbaum Associates.

Mayer, R. (1989b). The Psychology of How Novices Learn Computer Programming. Soloway and Spohrer eds. Lawrence Erlbaum Associates.

Mayer, R. et al. (1989a) Learning to Program and Learning to Think : What's the Connection? Soloway and Spohrer eds. Lawrence Erlbaum Associates.