(Preliminary Version)

COMP 2650 Experience: Teaching Functional Programming to Non-Majors
{Alternate Title: Slip broccoli under the cheese}

Ashoke Deb
Department of Computer Science
Memorial University
CANADA

Abstract

A proposal to teach functional programming, in an introductory service course, to non-majors with minimal mathematics background, is certain to face skepticism, cynicism, and strong opposition.

It turns out that such objections are without sound justifications, and unnecessary.

By exposing and emphasizing concepts available in ordinary, and common, tools. such as Windows, DOS commands, MSWORD, Excel etc, a smooth and successful transitions can be made to functional programming.

Such methodology was used in COMP 2650, usually taken by non-majors, with minimal mathematics background, and was very successful.

1 Background

The course COMP 2650 has been offered for many years as a computer literacy course, and it is intended basically to give non-computer science students an exposure to various aspects of computers and computer science.

The Calendar description of the course is as follows:

> **COMP 2650 Problem Solving with Personal Computers** is an overview of tools and techniques that current computer technology offers in a PC based networked environment. The emphases are on conceptual understanding of the software. from exploring the capabilities of the existing software tools to learning methods of extending these capabilities. The key topics include problem solving strategies, visual programming. macro language operations etc.
>
> Prerequisite: Math 1000 or equivalent

There are more or less 32 of fifty minute lecture hours, and equal number of mandatory laboratory sessions.

It basically starts with coverage on Windows, some MSDOS commands, then to MSWORD, Excel, Access. Then, in the programming component, it used Visual Basic.

Given the number of lecture sessions, and the topics to be covered, there are at most 6 weeks that can be devoted to the programming component.

The students from varied disciplines, such as Kinesiology and Physical Education, Business, Science, and Arts, take this course either as a part of the requirement, or as an elective.

The course did not achieve great success for a number of reasons: First, coverage of the tools were basically a semester long user's guide, and most students already knew how to navigate these tools by "hunting and gathering"; Second, when the course came to the programming part using an imperative language, all the unnatural, idiosyncratic concepts appeared to the students as gibberish, illogical and unnatural.

A couple of years ago, I was assigned, for the first time, to teach COMP 2650.

I proposed that for the programming component I will use a Functional Language.

That proposal immediately met with strong oppositions, skepticism and cynicism, from the Head of the Department, Undergraduate Studies Committee,and the Dean of the Faculty of Science.

2 Reasons why functional programming should not be taught in an introductory course for the non-majors

Some of the cited reasons are:

- Functional languages are esoteric, highly mathematical, and are meant for professionals with lot of formal background;
- Undergraduate Non-majors do not have the mathematical sophistication to understand or use functional languages'
- Functional languages are research curiosity, and they are not stable;
- The languages are not available on PC platform;
- Those that are available on PCs, do not have good error messaging and error handling support;
- There is no industry designed and supported functional language;
- There is no good textbook at the level of non-major undergraduates;

- Laboratory assistants are not knowledgeable in functional programming; hence they will not be able to assist the students.
- Most universities do not teach functional programming at the introductory level and to non-majors; Best universities have best students; even they don't dare!
- Say what? You are going to teach them recursion, type deduction, scoping, polymorphism, lists, strings, records, pattern matching, list and string manipulations, good programming, art of programming, literate programming, science of programming – in six weeks, to non-majors – really? Do you know that It takes just about two weeks to teach them "variable as a box" in Basic??

Some of the reasons cited above cannot totally be dismissed. But, many of the concerns stems from misinformation, lack of knowledge, and lethargy on the part of the educators.

At the time of the proposal, before it's been class tested, I did not have hard evidence to present to the authorities who make the final decision.

I argued that:

- Functional languages are based on sound mathematics. It does not necessarily require that the programmer has to be a sophisticated mathematician.
- There are functional languages which are available on PC, and are well supported.
- High-level concepts of programming which are difficult to teach in imperative languages are quite natural, easier to teach in functional languages.

Finally, I chose Functional F# which is available on PC, and had a visual editor, and also there were couple of books on the subject

Nevertheless, I realized that getting a cold start on functional programming near the end of the semester will be abrupt and disconnected from the rest of the course.

3 General methodology adopted

We decided to introduce concepts as early as possible, and introduce them from within the territory familiar to the students.

Greater part of the class lectures was devoted to exploring and emphasizing those concepts which have broad implications, and useful in programming as well.

Hands-on experience and detailed knowledge of various features of the application software were given in the lab sessions.

This two-way migration between high level concepts that they did not know and aspects of the applications that they were used to created a synergy that the students found interesting and engaging.

Although the above idea may not be new, the journey from the conceptual abstraction to actual programming using functional language is much shorter. As a result, for the students it seemed immediate and intuitive. Programming concepts – such as recursion, types, polymorphism, lists and string manipulations, pattern matching, scope rules, local and global definitions etc, – which are very difficult and time consuming in imperative paradigm, were much easier to deliver in our context, even to non-majors with basic math background.

3.1 Windows and Recursion, Trees, Stacks, Queues, and Method of Successive Refinement

Students were already familiar with the use of *folders* in *Windows*. Our discussion on *Directories and Files* immediately allowed us to introduce *recursive definitions* and *tree* structures. It was followed by additional examples and then continued discussions such as convergence (*termination*) of recursive definitions, why all *self referencing* do not make a recursive definition etc.

Other structures, such as *stacks* and *queues*, followed the discussion on trees. Windows *undo* sequence and line up at the printer were convincing examples of stacks and queues.

At this point, I broadened the discussion to the *method of successive refinement of problems* – which is obtained by asking "what" is to be done at each stage, rather than "how". It was shown that such process of problem solving also gives rise to a tree structure. We had substantive discussions on the methods of *top-down design* and *bottom-up development*.

Given the diverse interests of the students, we used examples from science project design, business proposals, report writing, to wedding planning, for the purpose of demonstrating the generality and usefulness of the method of successive refinement.

As it turned out, at this point students were already applying these concepts and techniques in their other courses, and vastly benefited from it.

Then, we moved on to the topic of DOS commands.

3.2 MSDOS and tree, file redirection and pipe

A large subset of DOS commands were discussed to show the parallel between system level commands and Windows level functions. We purposefully included DOS commands *tree*, file redirection ($>$ and $<$) and pipe ($|$). The tree command displays the tree structure which is not obvious from the Windows display, and that was convincing to the students. DOS file redirection commands and pipe command paved the way to the eventual discussion of $|>$ operator (forward) and $>>$ operator (function composition) in F#.

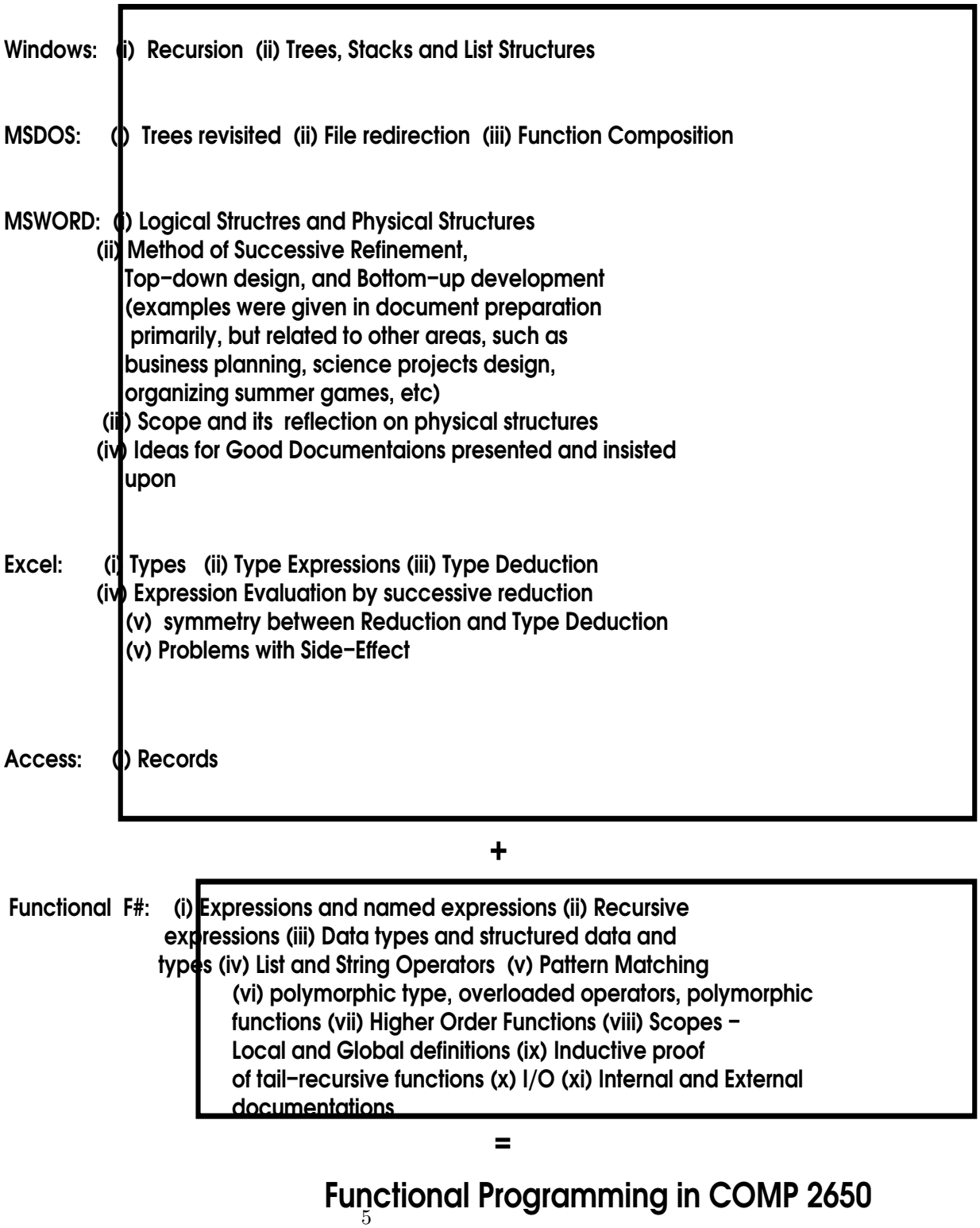

Figure 1: Concepts exposed, explored, and emphasized from within the Tools, and later used, along with F#, to teach good programming.

3.3 MSWORD and correspondence between the logical and the physical structure of a document, literate programming, and scope rules

The ability to prepare a good document is an essential tool for the students of all disciplines.

As Lamport points out, WYSIWYG formatters, like MSWORD, tempts one to the visuals first.

Designing and developing the shape of a document, and the logical relationship of the parts were emphasized via the method of successive refinement that we discussed earlier in the course.

Then we presented how the hierarchy of ideas and the importance of the ideas dictate the horizontal and vertical spacings, font sizes and style etc.

The fact that a good document structure gives rise to a tree, conveying those concepts above were fairly easy and direct.

We also had other reasons for this approach.

Knuth, and others, argued that a good program must have a good documentation - both internal and external, which he called *Literate Programming*. In literate programming, both internal and external comments precede the code development, and not as an after thought.

Also, in functional programming the physical layout of the code – in horizontal spacing - determines the scope of definitions, and also improves readability.

We felt that these ideas should be presented to the students as early as possible.

3.4 Excel and expression, evaluation, reduction by pattern matching and substitution, types and type deduction

Expression evaluation by pattern matching and substitution, the concept of irreducible expression as the manifest value, strong notion of types, type expression and type deductions are major part of functional programming.

We introduced these ideas within the context of Excel.

Although Excel does not support recursion, we extended the discussion of reduction of expressions to recursive expressions, such as *fac n = if n=0 then 1 else n*fac(n-1)*

Students did not have any difficulty in understanding and appreciating the issues. In fact, students pointed out the fact that the expression will not eventually terminate if n is negative!

We contrasted recursive definitions with side-effect $A = A + 1$ (which Excel does not allow), where A is a cell name. Seeing that Excel does not allow such thing, avoidance of side-effect was an easy sale.

In Excel, type descriptions of functions and operators are given in verbose.

For example,

+ takes two numbers and adds them to produce a result number

...

Starting from that, we presented the type definition as:

$$+ : : number, number -> number$$

We then presented the definitions on a more simplified form that follows the syntax of of the operator:

$$number + number -> number$$
$$concat\ (string, string) -> string$$
$$number\ \% -> number$$

Later, in the discussion of F#, we write this type expressions as
$+ : : number -> number -> number$
Of course, this transition through different syntax for type signature made it palatable to the audience, and then we could talk about concepts like *currying* in F# quite easily.

Then, we introduced types of structures, and distfix expressions.

Type of a list (a cell range, eg. A1:A8) was $[t]$, where t is the type of a cell, determined by its expression.

In Excel, *IF-THEN-ELSE* expression does not insist that the then-part and the else-part be of the same type.

We used the following:

$$IFTHENELSE\ (bool, t1, t2) -> t1|t2$$
or,
$$IF\ (bool)\ THEN\ t1\ ELSE\ t2 -> t1|t2$$

We then discussed type deduction of expressions from ground types. And we showed the parallel between reducing an expression and its type deduction at each stage. I believe that students had a clear understanding on this subject.

3.5 Access and records

We discussed records in this section.

But, limited time did not allow deeper discussion on the subject.

3.6 F# and Functional Programming

At this stage most of the ground work was in place. The students were motivated, partly by the fact that they uncovered new ideas from old and well traveled tools, and partly by the fact that they were successfully employing many ideas from this course to the other courses.

In functional programming, everything is an expression, and everything has a type.

The simplicity, uniformity and universality of the notion above appeal to the students.

From the perspective of an educator, it becomes less time consuming and much easier to teach powerful concepts in one framework than in other programming paradigms.

We covered the following topics:

- Expressions
- Named expressions
- Parameterized expressions
- Basic data types
- Structured data types
- Operators, and built-in functions, including lists and string operations
- Polymorphic type
- Local and global definitions
- Recursive functions
- Polymorphic functions
- Higher Order functions
- Currying
- Pattern matching
- Encapsulation
- Error handling
- Basic I/O

We deliberately avoided any reference to side-effected features available in F#.

Also, although we discussed and demonstrated the inductive proofs of tail-recursive functions, I did not make that a focal point of the subject.

Finally, I emphasized and insisted on good documentation. In that regard, their project submissions were judged, not only on the correctness of their program, but also on the development, and proper documentation.

Requirements for the external document was to include the following:

- Tree structure showing the decomposition of functions and the logical dependency among them;
- Purpose and Input Output specification of the functions;
- Table of Content;
- Explanations of built-in functions;
- Dictionary of variables
- Error handling, and
- Possible improvements.

The main requirements in the internal presentation were:

- physical layout of the code, and
- meaningful comments with code segments,

4 Performance

The course was offered in this manner for the last two years.

Judging by the performance of the students and the quality of their submission, I have to say that it was a success. The feedback from the students, after the semester, of course, were also very good.

In our CS program, we use Java as the primary teaching language; and all cs majors have to take a series of courses on programming using Java. Most of the students coming from Java regime find topics that were covered in COMP 2650 very difficult. Instructors who teach those courses also admit difficulty in teaching those concepts.

In conclusion, using my experience in teaching COMP 2650, I strongly believe that all computer science departments should re=think their philosophy and strategy in teaching introductory courses.

References

[1] Hughes, J., Why Functional Programming matters, In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[2] Syme, Granicz and Cisternino, Expert F# 2.0, Apress, 2010.

[3] R. Pickering, Beginning F#, Apress, 2009.

[4] R. Bird, Introduction to Functional Programming using Haskell, Prentice Hall, 1998.

[5] P. Hudak, The Haskell School of Expression - Learning Functional Programming Through Multimedia, Cambridge University Press, 2000.

[6] S. Thompson, Type Theory and Functional Programming, Addison Wesley, 1991.

[7] S. Thompson, Miranda: The Craft of Functional Programming, Addison Wesley, 1995.

[8] P. Henderson, Functional Programming: Application and Implementation, Prentice Hall, 1980.

[9] J. Darlington, P. Henderson and D. A. Turner, Editors, Functional Programming and Its Applications, Cambridge University Press, 1982.

[10] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[11] D. Knuth, Literate Programming, University of Chicago Press, 1992.

[12] L. Lamport, Latex= A Document Preparation System, Addison Wesley, 1986.