The Dys-Functional Student

Chris Clack[1] and Colin Myers[2]

[1] Dept. of Computer Science, University College London, London WC1E 6BT
Email: clack@cs.uc.ac.uk
[2] School of Computer Science, University of Westminster, London W1M 8JS
Email: colin@wmin.ac.uk

Abstract. Functional languages liberate students from complex syntax, complex semantics and complex memory allocation; allowing them to concentrate on problem-solving. However, functional programming is not a universal panacea. Students still have problems with language features, program concepts and the legacy of the imperative paradigm. This paper aims to assist the lecturer asked to teach a functional language for the first time. We present typical student mistakes, attempt to explain why these mistakes arise, and propose possible remedies.

1 Introduction

Functional languages have been taught for a number of years at both University College London (UCL) and the University of Westminster, covering undergraduate and postgraduate courses.

The proximity of UCL and Westminster has resulted in a close collaboration. In particular, we have jointly produced two functional programming text books [Mye93, Cla94], together with on-line support in the form of coursework questions (solutions emailed on request), past exam papers, etc. These can be accessed via the World Wide Web at: http://www.cs.ucl.ac.uk/staff/C.Clack/teaching.

Our combined experience has proven extremely positive. For example, UCL first year undergraduates (many of whom have never programmed before) undertake a large Miranda[3] project at the end of the first term, including substantial programming tasks such as chess end-games, the travelling salesman problem and cross-compilation (this benefit has also been reported by, for example, [Lam93]). Furthermore, many advanced programmers find the experience of using a functional language improves their ability to solve problems in imperative languages.

Nonetheless, students find programming difficult. The aim of this paper is to help answer the question "What mistakes do students make and what does this tell us about how students think?"

Section 2 outlines our approach to teaching functional languages (course details are provided in the Appendix); Section 3 presents a collection of typical student problems that arise from Miranda language features; Section 4 looks at general programming problems and Section 5 at issues arising from previous experience of imperative programming. Each section contains *recommendations* for the lecturer to help overcome student errors and improve programming style.

[3] Miranda is a trademark of Research Software Ltd.

1.1 Scope

Space does not permit us to cover all classes and kinds of student mistakes. The approach taken by [Mye93] and [Cla94] is to teach programming practice by means of both correct code (to show how to do it) and incorrect code (how not to do it — examples often taken from actual student errors). We do not replicate this work, nor the valuable observations of [App94, Joo93, Mol93].

Furthermore, our discussion is limited to "programming in the small"; consequently, there is little consideration of data abstraction, design or larger software engineering issues.

2 The Teaching Process

From 40% to 100% of our students have had some previous experience of programming (often BASIC, Pascal, C or some assembler) and they are loath to discard their current programming skills. Indeed, experienced programmers will often be hostile since they do not believe that functional languages are used "in the real world".

This problem is exemplified by [Mor82], who asks whether functional programs are like *haikus*, in that they can teach through their innate elegance, even though in practice, they are as useless as a formal poem; or more like *karate* in that they have both an innate elegance and are useful in a fight.

We tackle this problem head-on and emphasize to these students that functional programming is a higher-level process that will make them better imperative programmers.

Rather than teach "functional programming" in general, our approach is to teach programming skills in a particular language (either Standard ML or Miranda). This is used to develop problem-solving skills which can be transferred to other paradigms. We avoid the notion of "programming as a mathematical exercise" because few of our students have sufficient background in mathematics to appreciate such a concept. This does not mean that we avoid rigour; merely that we emphasize a pragmatic approach.

3 Language features

This section details student problems that can be traced back to the features of Miranda. This should *not* be interpreted as a criticism of the language features themselves; the intention is to highlight the student learning path and the points at which conceptual jumps must be taken.

3.1 Problems with types

Students are introduced to *types* at a very early stage. However, data typing is not a straightforward issue for the naïve student:

Booleans.

Whilst students are generally happy to categorize numbers separately from text, they are less sure about Booleans. Booleans are more abstract than numbers and text; students cannot see, hear, taste, smell or touch "truth". Furthermore, a message (text) may itself be true or false; this causes further confusion, which manifests itself in difficulties with treating Booleans as data values, and is the source of the following common errors.

1. Returning strings instead of Booleans:

```
>equals_six :: num -> [char]
>equals_six x = "true", if x = 6
>             = "false", otherwise
```

2. Including unnecessary tests:

```
>equals_six :: num -> bool
>equals_six x = True, if x = 6
>             = False, otherwise
```

The simple version of the `equals_six` function often comes as rather a surprise:

```
>equals_six :: num -> bool
>equals_six x = (x = 6)
```

Numbers and their representation.

Students often do not understand the difference between a number and its representation; in everyday use they rarely use anything other than base ten to represent numbers. Despite the fact that most students have learnt about different number bases at school, they have difficulty with the number-to-string conversion function `ntos`.

This function takes input of type `num` and returns its denary (base 10) representation as a string. Thus, `ntos 23` returns `"23"`. Many students either cannot grasp the difference between `23` and `"23"`, or cannot understand what is involved in the conversion from a number to a string. When questioned, students agree that a computer stores all numbers as binary, but point out that Miranda treats all numbers as denary (viz. `x = 23`). Until challenged, some students tacitly assume that Miranda stores numbers as strings (as given by the built-in function `show`).

This concept needs much care; it may help to start with a function that gives the octal (or any non-denary) representation of a number.

Functions and type messages.

Many students find it difficult to accept that a function's type is a *mapping* from one type to another. They prefer to think it is the target type. It is probably this more than anything that leads to errors in function definition.

We encourage students to provide function type declarations before function definitions. Sadly, with higher order functions and with complex data structures (such as lists of lists or user-defined types) students often omit the type declaration when they write the function; because it "generates errors".

Recommendations.

Types are not intuitive. In particular, it is not obvious that "truth" can be manipulated as a data value. The lecturer must motivate the manipulation of truth values and should spend some time explaining the meaning of types; further, it is important to insist on type declarations before function definitions. Indeed, it is a good specification exercise to ask students to provide function type declarations without the function code.

Type error messages.

Both [Lam93] and [Joo93] comment that Miranda's type error reporting is not particularly helpful to the student. [Joo93] observes that it is mainly used to highlight the error's textual position but not the nature of the error. This is also our experience. For example, consider the following:

```
>splitsen :: [char] -> [[char]]
>splitsen sen = sss (sen, "")
>
>sss :: ([char],[char]) -> [[char]]
>sss (x : xs, wac)
>  = wac : sss(xs, ""), if x = " "
>  = sss(xs, wac : x), otherwise
```

This program generates the following error messages (we have wrapped some lines):

```
type error in definition of sss
(line  6 of "script.m")
cannot unify ([[char]],[char])
with ([char],[char])
(line  7 of "script.m")
cannot unify ([[char]],[char])
with ([char],[char])
incorrect declaration
(line  4 of "script.m")
specified, sss::([char],[char])->[[char]]
inferred,  sss::([[char]],char)->[char]
```

This is clearly unintelligible for the average student. Perhaps something closer to SML's error reporting would be more appropriate.

3.2 Problems with recursion

Not recognizing terminating cases.

When introducing the principle of recursion it is normal to introduce the idea of a base case and a general case; for example, zero and non-zero integers or the empty and non-empty lists. Students find this template easy enough to grasp but often will not understand how to extend it to deal with many base cases or with multiple lists.

This is exemplified in an answer to an exercise to give the pairwise addition of two lists. The code is taken from an actual student.

```
>listadd :: [num] -> [num] -> [num]
>listadd [] []  = []
>listadd (x : xs) [] = x : listadd xs []
>listadd [] (y : ys) = y : listadd  [] ys
>listadd (x : xs) (y : ys)
> = (x + y) : listadd xs ys
```

In the above code the second and third patterns should be treated as terminating cases but the student has erroneously thought that both lists must be empty to satisfy a terminating case. Perhaps they have been misled by writing the second and third patterns in the format:

```
>listadd (x : xs) [] = ???
>listadd [] (y : ys) = ???
```

If they started from the assumption that an empty list *anywhere* might be a base case then they might have written these patterns as:

```
>listadd any [] = ???
>listadd [] any = ???
```

In this manner they may have been less tempted on the path of fruitless recursion.

Note that the alternative approach of considering the general case/s and letting the base cases "fall out" would *not* particularly help to remove this problem. The student can justifiably argue that the following patterns are general cases:

```
>listadd (x : xs) [] = ???
>listadd [] (y : ys) = ???
```

Too many base cases.

It is also worthwhile noting that most students who gave an operationally correct answer also gave three base cases:

```
>listadd [] [] = []
>listadd [] any = any
>listadd any [] = any
```

In fact, there is no need to identify the first (double empty list) pattern. When questioned why they have included it, most students have argued for "efficiency purposes"; so that processing could stop immediately if two empty lists were encountered! They fail to recognize that the second and third base cases are specialized enough and that the extra test can only introduce extra processing.

Recommendations.

Despite many claims that recursion is easier to understand than iterative control structures, there is still a need to emphasize how to identify base cases, especially when more than one parameter of recursion is involved. There is also a need for a tool to think about general cases; for this we recommend the use of structural induction, as described in [Cla94].

At some point, program efficiency may also be mentioned. Beginners are told to put more general patterns last (because they often use overlapping patterns); once students have gained some facility with non-overlapping patterns, they may be invited to think more carefully about operational semantics and efficiency. For example, with `listadd` it may be best to place the base cases after the general case because they are less likely to occur for any given pair of lists.

3.3 Accumulating trouble

As has been observed, many students find it difficult to identify an appropriate base case for the parameter of recursion: in other words they have difficult with *last things*. Sadly, many also have difficulty with *first things* especially where numbers or accumulators are concerned.

The following exercise is typically answered incorrectly by over half the students of any given level of experience:

> *Find the two largest numbers in a list of numbers.*

This specification seems easy enough to read but it generates one or more of the following mistakes.

Inadequate error handling.

For example, not catering for lists with less than two numbers. Curiously, if students are asked to discover the largest number in a list of numbers they will normally deal with the empty list as an error or exception but *two* largest numbers somehow deadens their validation skills.

False assumptions about the data.

Many students assume that "numbers" means "natural numbers". One solution to this simple exercise is to invoke an auxiliary function with two accumulators representing the two largest numbers and alter these accumulators, as required, on each recursive application:

```
>largestaux [] max1 max2 = (max1, max2)
>largestaux (x : xs) max1 max2
> = largestaux xs max1 max2, if x < max1
> = largestaux xs max2 x, if x > max2
> = largestaux xs x max2, otherwise
```

The common student mistake is to invoke the function as follows:

```
>largest any = largestaux any 0 0
```

The possibility that numbers might include negative numbers is rarely recognized.

Overcomplication.

It seems a universal truth that students will chose the more complicated of any two options. The above `largestaux` function relies upon the simple fact that `max1` will never be larger than `max2`. This means that it must be suitably invoked:

```
>largest []
> = error "at least two items needed"
>largest (x : [])
> = error "at least two items needed"
>largest (x : y : rest)
> = largestaux rest x y, if x < y
> = largestaux rest y x, otherwise
```

A little time looking at accumulator initialization often saves subsequent extra programming effort. Many students invoke `largestaux` as follows:

```
>largest (x : y  : rest)
> = largestaux rest x y
```

If the accumulators are unordered then the function `largestaux` is obliged somehow to sort the current `x, max1` and max2 and discard the smallest.

Innocence.

Another student solution is to select the largest number from the list and then search the list for the next largest number. Those students choosing this method rarely deal with the possibility of the two largest numbers being the same size. When asked why they have chosen this method, they occasionally say that they did not realize that they could identify the first *two* numbers in a list.

This error probably arises because it is so common to introduce list processing in the form:

```
>listhandler [] = ???
>listhandler (x : xs) = ???
```

Some students blink and miss more extended examples such as `(x : y : rest)`. Similarly the more passive student will not see the legality or desirability of more than one accumulator.

Laziness.

One of the problems with teaching is that students sometimes listen. It is now reasonably accepted that programming is theft, i.e. lecturers encourage students to re-use proven, working code. In the above example, clever students have been known to sort the list, reverse the result and extract the first two elements via pattern matching. The mistake is for the lecturer to penalize a good example of code re-use: however, some comments on algorithmic efficiency might be appropriate.

Recommendations.

Firstly, students should be encouraged to consider whether their functions are partial or total. The evidence is that the more complex the algorithmic task the more neglected the validation (irrespective of its level of complexity or difficulty).

Secondly, students must identify what is meant by a "number". David Turner, in conversation with the authors, has indicated that if Miranda 2 ever gets written then it may have two numeric types: natural numbers and floating point numbers. This may well be a more natural reflection of how students actually think about numbers.

Thirdly, students must consider the initialization of accumulators very carefully: to avoid errors and save on programming effort.

Finally, lecturers must show that pattern matching can extract any given element from a list.

3.4 Trouble with lists

Weaker students encounter two basic problems with lists: list construction and empty lists.

List construction.

Students are given two syntactic constructs for list manipulation: *cons* (`:`) and *append* (`++`). Furthermore, *cons* can be used on either side of a function equation *for different purposes.*

Whilst students are attracted to the simplicity of pattern matching, they are confused by the two uses of *cons*: both within a pattern to deconstruct a list and, conversely, within a function body to construct a list. There is no simple recipe that they can follow that will always work. They must make a choice at an abstract level of program design, and this is difficult for the weaker students.

Empty lists.

Students often confuse `[]` with `[[]]`, and `[]` with `""`. In particular, they are unaware that `[]` and `""` have different types. They are confused that `[]` is an empty list of *anything.*

There is a compulsion to use `[[]]` as the terminating value in a stack recursion producing a list of lists — the fact that `[]` can also be an empty list of type `[[*]]` is mystifying to some students.

The above problems are illustrated by the following student solutions to list-processing questions.

> *Write a function that will take a list of names and "shuffle" them by interleaving them a given number of times. To interleave a list of names, the list should be cut into two halves and then put together by taking elements from each half alternately.*

Here is one student's answer (note the comments):

```
The output from sshuffle should be a list but
Miranda will not accept me using the cons ":"
thingy so I've had to use append "++".

>shuffle :: [[char]] -> [char]
>shuffle [] = ""
>shuffle lst = sshuffle (0,(#l div 2),lst)

>sshuffle :: (num,num,[[char]]) -> [char]
>sshuffle (i,pos,lst) = [], if i = #lst
>sshuffle (i,pos,lst)
>  = (lst ! (pos+i)) ++ (lst ! i)
>    ++ (sshuffle (i+1,pos,lst))
```

Ignore the fact that the function doesn't answer the question (it will only do a single interleave) and that it needs further attention to avoid list indexing problems. The important issue is the confusion over lists and the important error is that the return type is `[char]` instead of `[[char]]`.

First, the student uses `""` as a terminating value for `shuffle`; from this point, all is lost because `""` has type `[char]`. Of course, the student should have used `[]` which has type `[*]` and can therefore happily accommodate a return type of `[[char]]`.

Second, by the student's own admission, there is a confusion over the use of *cons* and *append*. After making the apparently reasonable choice to use *append*, they miss the requirement to put the first two elements inside list brackets:

```
>sshuffle (i,s,l)
> = [(l ! (s+i))] ++ [(l ! i)]
>   ++ (sshuffle (i+1,s,l))
```

With just these two changes, the type definitions could be altered to `[[char]] -> [[char]]` and everything would be type correct.

Another example:

> *Write a function which takes a string of characters and returns the list of all words in the sentence, by dividing the string up at the spaces.*

```
>splitsen :: [char] -> [[char]]
>splitsen sen = sss (sen,"")
>
>sss :: ([char],[char]) -> [[char]]
>sss (x:xs,wac)
>  = wac : sss(xs,""), if x=" "
>  = sss(xs,wac:x), otherwise
```

There are three basic problems: Firstly, the guard confuses a string with a character; it should be `if x = ' '`. Next, the final line shows a deep confusion regarding *cons*: the types are the wrong way around. Finally, the following base case is missing:

```
>sss ([], wac) = [wac]
```

On questioning, the student explained that Miranda had produced multiple type errors that were seemingly unrelated to the program. It was therefore difficult to debug the program (see Section 3.1.5).

As a final example of list abuse:

Write a function to transpose matrices.

```
>ro_list :: [[*]] -> [[*]]
>ro_list [] = []
>ro_list l
>  = (map hd l) : (ro_list (map tl l))
```

Here, each list item is successively tailed until only `ro_list [ [], [], [], [], ..., [] ]` remains. Then `ro_list` calls `map hd ([]:rest)` causing a `hd []` error. On questioning, the student thought there was a problem with the language! In fact, the solution is to include the following pattern:

```
>ro_list ([]:rest) = ro_list rest
```

Recommendations

It is wise to constrain weaker students so that they do not lose confidence. One approach is that weak students should use *cons* only in patterns to deconstruct a list and use *append* in function bodies.

Type problems, and problems involving empty lists, can only really be countered by providing the students with plenty of simple exercises that have been carefully constructed to illustrate the differences. Again, it is helpful to ask students to think about "last things" — i.e. what happens as termination is approached — and to encourage them to *visualize* the changes being made to data. Visualization is particularly helpful when lists of lists are manipulated (for example, successively mapping `tl` over a list of lists).

3.5 Higher order dysfunctions

The joy of functional programming is really obtained with the ability to understand and use higher order functions (HOFs). Students readily accept and understand simple and explicit HOF usage yet, despite lecturer enthusiasm, students seldom understand their implicit usage and more rarely employ their full potential. Students make simple use `map` and `filter` but are less willing to explore the power of partial application and function composition. We now describe some of the more common student misgivings.

Parameter passing.

One problem lies with the partial application of curried functions. After much exposure students can understand and define functions in terms of partial applications:

```
>multiply x y = x * y
>mult3 = multiply 3
```

They are far less likely to use them as parameters to other HOFs. The following code is rare:

```
>mapmult3 = map (multiply 3)
```

Students are more likely to write:

```
>mapmult3 nlist = map mult3 nlist
>                 where
>                 mult3 = multiply 3
```

Notice the student code is more explicit. Everything that is needed is *visible* and is either a constant or has a name. Students know that a function translates from one domain to another and expect *to see* this throughout their code. What the code fails to demonstrate is a willingness to undertake abstraction in the form of partial parameterization.

Mistrusting HOFs.

It appears that students do not trust that HOF code can be so concise as to not require a base case. The following code is typical in adding a superfluous test for `[]`:

```
>cubesplusone [] = []
>cubesplusone ns = map cinc ns
>                  where
>                  cinc x = (x^3) + 1
```

Once again notice the fear of passing a partial application as a parameter. Here, there is the added complication that a function composition of sections would be required (`((+1).(^3))`).

Manipulating HOFs.

As observed by [Lam93], students do not take naturally to using general recursive functions such as `foldr`. We agree, and notice that students are only comfortable if `foldr`'s functional parameter is an operator such as `+` or `++`, which is associative and possesses an obvious identity.

Without guidance, most students do not discover HOF equivalences. For example, here is a typical answer to generate the sum of the squares of a number list:

```
>sumofsqrs xlist = sumlist (map square xlist)
>                  where sumlist = foldr (+) 0
>                        square x = x * x
```

A more concise (perhaps more efficient) answer is:

```
>sumofsqrs = foldr ((+) . (^2)) 0
```

This exploits the equivalence:

```
foldr f def (map g x) = foldr (f . g) def x
```

It is likely that students miss the equivalence because they have broken the problem into manageable chunks, i.e. firstly squaring each list item to create a new list, and then summing that list.

Overstretching the student.

Some HOF exercises are just too hard, although they can be simply expressed and have an elegant solution which may demonstrate many important points. Setting such questions actively discourages students. Examples include generating a powerset (depending on the student background), manipulating combinators and rewriting `filter` in terms of `foldr`.

Here is the best student answer we have seen to meet this last requirement:

```
>ffilter fun alist
>  = foldr ((++) . auxr fun) [] alist
>    where
>    auxr fun elem = [elem], if fun elem
>                  = [], otherwise
```

This was a very good attempt; it needs an imaginative leap to represent an excluded item as `[]`, and hence to use `++` to combine the remaining items. Notice, however, that `auxr` did not require `fun` which was already in scope, and that mentioning `alist` was superfluous.

Recommendations.

Firstly, HO abstraction should be illustrated by means of textual substitutions [Cla94]. For example (where `==>` represents a substitution step):

```
     mapmult3 [5]
==>  map (multiply 3) [5]
==>  ((multiply 3) 5) : map (multiply 3) []
==>  (multiply 3 5) : map (multiply 3) []
==>  15 : map (multiply 3) []
==>  15 : []
```

Secondly, lecturers should emphasize that HOFs such as map, filter, foldr and foldl encapsulate *both the general case and the base case* of list recursion.

Furthermore, lecturers should control their enthusiasm for HOFs and avoid setting "clever" questions just for the sake of the question.

4 Programming problems

It is seductive to believe that using a functional language will somehow cause all the normal problems associated with programming to go away. This is not the case. However, the use of a functional language does facilitate the diagnosis of programming problems (often if only because the programs are shorter than the equivalent imperative programs).

4.1 Assumptions and comments

One of the major problems that students appear to have is to make assumptions when not required and not to make them when required.

It is by now well-established that it is difficult to provide an accurate natural language specification of even the simplest of problems. This makes setting student exercises a potentially hazardous business, relying upon a tacit understanding, by both lecturer and student, of what is required from the student at any given stage in their course.

To help resolve such ambiguity, it is not uncommon to precede exercises with a request such as:

> *Remember to add appropriate comments and assumptions. If any specification is vague then ask for clarification.*

Given that most students only attempt coursework twenty-four hours (at best) before the deadline, it is a rare event for a lecturer to be asked for clarification, *even if the original question is highly ambiguous.* Instead the better student will tend to resolve any ambiguity in the same way as the lecturer *and often not bother to explain the ambiguity.* Their answer will generally be accepted as correct and given full credit, despite the absence of clarifying comments.

Weaker students often fail to see the question's ambiguity and so either make no assumptions and hence provide a good answer to a simpler problem; or leap to some wild "resolution" of the ambiguity, also not bothering to state their assumptions. For both cases students lose marks and hence lose some confidence in their own ability.

The following amplifies the first of these points:

Write a function which takes a character and returns the next letter in the alphabet (make it return 'a' if given 'z').

This exercise is horrid in that it allows the confusion of ASCII characters with lowercase alphabetic letters. What should be returned by `'Z'`? What about digits and other non-alphabetic characters?

Typically, only 10% of students will actually state their full assumptions. Of the rest, about half will assume that the input was alphabetic and catered for both `'z'` (as explicitly requested) and `'Z'` (as was perhaps was implied). Thus they answer a simpler question than intended.

The other half will decide (quite reasonably) that all non-alphabetic characters should raise an error (though will not comment that this was their assumption). Here, they should be given full credit for making a reasonable assumption; it would be churlish to act otherwise. However, apart from the code itself, there will be no visible indication that they have really thought through the problem.

Misunderstanding the question.

It is surprisingly easy for students to misunderstand a seemingly innocuous question:

Write a function to test if two lists have the same elements, regardless of their order.

The simple answer uses the list difference operator:

```
>testlist :: [*] -> [*] -> bool
>testlist xs ys = ((xs -- ys) = [])
>                & ((ys -- xs) = [])
```

However, consider this student answer:

```
>testlist :: [*] -> [*] -> bool
>testlist a b = True, if #(a--(a--b)) > 0
>             = False, otherwise
```

At first sight this could be either a misunderstanding of how the list difference operator works, or just poor reasoning. On further analysis, it becomes clear that the student is testing whether *two lists have any elements the same* rather than whether they have the same elements!

Here is another common misunderstanding of the above question:

```
>testlist :: [*]->[*]->bool
>testlist xs ys = ~ notsame xs ys

>notsame :: [*]->[*]->bool
>notsame xs ys
>     = (member (isinxs xs ys) False)
>       \/ (member (isinys ys xs) False)

>isinys = isinxs
>isinxs xs ys = map (member xs) ys
```

Here, the student has extended the question so that the function tests for the same elements *irrespective of repetition*. Some students did this deliberately; they stated their assumptions in their comments, thereby acknowledging that they were answering a different question. Others seemed not to realize that they were ignoring repetitions:

```
>testlist l1 l2 = and (same_test l1 l2)

>same_test (x : xs) list2
>     = others ++ test
>       where
>       test = ((member list2 x) : [])
>       others = (same_test xs list2)
```

The problem of misunderstanding the question often arises because there is a cultural gap between the lecturer and the students. The lecturer believes that it is "obvious" that the question cannot possibly have more than one interpretation, whereas the students think that a different interpretation is perfectly reasonable.

Students are often bewildered by questions because of naïvety, preconceptions, different schooling and different nationalities. They have to learn the culture of the university and the lecturer's expectations.

Recommendations.

Firstly, the ambiguity of natural language makes it hard to set even the simplest of unambiguous exercises. However, this is not necessarily a bad thing as it will often yield a good indication of which students are willing to think around a problem; which students passively take the path of least resistance; and which students are extra-terrestrials.

Secondly, notwithstanding the fact that ambiguity can be useful, we should endeavour to provide more precise questions *illustrated with examples*.

Thirdly, students should be obliged to use sensible comments even for apparently simple exercises. Of course, this begs the question 'what is a sensible comment?'.

4.2 Analysis and testing

Many students find analysis very hard. They find it so difficult to find an answer that works at all for just one simple test case, that they cannot face analyzing the problem further to provide a full solution. Testing is often viewed as pure drudgery; students often feel that they have found an answer to the question and that is good enough. Analysis and testing are, of course, key skills of programming and it is very important that students are trained to be thorough in both.

Here are two examples based on the previous `testlist` exercise. `testlist1` uses the test `(x=y)` instead of `(#x = #y)` (showing a lack of testing), whereas `testlist2` neglects to check for `((y -- x) = [])` (showing a lack of analysis).

```
>
>testlist1 :: [*] -> [*] -> bool
>testlist1 x y = (same x y = #x) & (x=y)
>
>same :: [*] -> [*] -> num
>same [] any = 0
>same (x:xs) ys
>  = 1 + same xs ys, if member ys x
>  = same xs ys, otherwise
>
>testlist2 :: [*] -> [*] -> bool
>testlist2 x y = ((x -- y) = [])
>
```

Recommendations.

Oblige students to provide a test plan, test data and test results.

4.3 Basic skills

Finally, some students still require help with their basic programming skills. The first example is from a student who is over-eager to give error messages (the function should actually return True for two empty lists) and over-eager to use `where` blocks.

```
>testlist :: [*]->[*]->bool
>testlist [] [] = error "Both lists empty"
>testlist [] any = error "One list empty"
>testlist any [] = error "One list empty"
>testlist l1 l2 = and (same_test l1 l2)
>
>same_test :: [*]->[*]->[bool]
>same_test (x : []) l2
>          = test
>            where
>            test = ((member l2 x) : [])
>same_test (x : xs) l2
>          = others ++ test
>            where
>            test = ((member l2 x) : [])
>            others = (same_test xs l2)
```

The second example is from a student who has forgotten that the base cases might be called immediately rather than at the end of the recursion (the function could be called as `testlist [] [1,2,3]` or as `testlist [] []`, both of which would provide the wrong answer). Furthermore, they have clearly failed to find a solution, as evidenced by the recursive call which seems to display a certain desperation!

```
>testlist :: ([*],[*])->bool
>testlist ([],anylist) = True
>testlist (anylist,[]) = False
>testlist (a:b,c:d)
>      = ((a=c) & testlist (b,d))
>        \/ testlist ((a:b),d)
```

5 The Grasp of the Imperative

Once a student has learnt to program using an imperative language, they find it difficult to escape its grasp. The converse is not true — students who first learn with a functional language go on to become good imperative programmers.

5.1 Pattern matching and guards

For the purposes of this section the term *pattern matching* will be used to describe the matching of a pattern to one of several alternative function patterns, e.g. the following function `cstylenot` has two patterns:

```
>cstylenot True = 0
>cstylenot False = 1
```

Using "guards" there are two possible versions:

```
>cstylenot1 x = 0, if x = True
>             = 1, otherwise
>
>cstylenot2 x = 0, if x = True
>             = 1, if x = False
```

The received wisdom is that pattern matching is probably better where discrete cases can be identified (as above). This corresponds more naturally to mapping the possible cases of input to their desired output: an observation that has proven useful from Jackson Structured Programming onwards. Yet at least half of final year BSc students and non-conversion MSc students chose the guard approach to solve similar simple problems or subproblems.

This is probably because students who can program fluently in a procedural language such as C, C++ or Modula-2 find it difficult to think in terms of functions. They find the transition to mathematical functions rather than imperative language "functions" hard. They eventually cope with definitions such as `f x = x + x` and that `f` can be treated as a value rather than some computation: however, they find it alien to have more than one definition of `f`, such as with pattern matching.

By contrast, students who learn a functional language as their first serious programming language can readily accept the idea of alternative patterns for a function. This leads more naturally to an appreciation of looking at the nature of input rather than fretting about operational expediency.

Given that most professional programmers do not use a functional language, it is not unreasonable to ask what advantages students gain from a pattern matching approach. The first advantage has been stated above: input awareness. The second is that imperative language `case`, `switch` or `evaluate` commands are really forms of pattern matching. The functional programmer should have no difficulty converting to a good imperative language style. The converse is not necessarily true.

5.2 Variables

Control.

The controlling of recursion can come as a nasty shock to experienced programmers. The more stubborn students (especially part-timers who earn their living from programming) will attempt to use external variables to achieve iteration rather than accept parameters of recursion. This is typical:

```
>x = 1
>loop y = y, if x = 3
>       = loop x + y, otherwise
```

Of course, this kind of error is a consequence of the students' imperative programming legacy and can easily be overcome within a function definition. However, many students cling to the procedural model which encourages them to use variable assignment and global state to solve problems; they are particularly puzzled by the functional language approach to file input-output and interaction.

Name redefinition.

Miranda enforces referential transparency by refusing to allow the re-use of any name already bound by the programmer in the script file or by the standard environment. Given the large number of names within the latter, this can lead to programmer frustration and, more importantly, to hidden errors if the programmer fails to alter all references to an offending redefinition.

5.3 Formal methods

One of the strongest arguments for the use of functional programming languages is that they have a sound mathematical basis. Consequently students who have a good grasp of discrete mathematics should become good functional programmers.

Its difficult to say whether this observation really holds for students who have already been swallowed by the imperative programming paradigm. The following student-generated code for writing the set membership function is typical:

```
>member []  x = False
>member (y : rest) x
>       = True, if x = y
>       = member rest x, otherwise
```

Students initially fail to see that the second pattern is better written as a disjunction:

```
>member (front :: rest) x
>       = (x = front) \/ (member rest x)
```

Instead they believe that a conditional must be used, and used in the manner of a control statement rather than a selector. This is probably a result of too much imperative programming.

Notice that Miranda permits an elegant pattern matching solution for this particular problem:

```
>member []  x = False
>member (x : rest) x = True
>member (y : rest) x = member rest x
```

Recommendations.

The imperative paradigm is hard to break once established. It is recommended to teach a functional language before an imperative language, *in conjunction with* teaching "formal methods". The above example only needs an understanding of simple propositional calculus. Similar formalisms can help to understand the ordering of expressions, the nature of functions and so on.

The semantics and functionality of file input-output and interaction have been the subject of much recent research; however, little work has been done on the provision of friendly syntax.

6 Summary

The issues discussed in this paper are widely neglected in the literature, yet there is a need to support lecturers who teach functional programming.

Based on our experience of using functional languages to teach programming skills, we have analyzed a range of student problems and provided recommendations for solving them. Of course, space has not permitted a complete list of student mistakes and their analysis, nor have our recommendations been exhaustive. We believe that there is still much useful work to be done in this area, to support the development of new teaching strategies and to guide the development of teaching-related language features.

Acknowledgments

We thank Mark d'Inverno for his constructive comments on earlier drafts of this paper.

References

[App94] A. Appel. *A critique of Standard ML. JFP*, 3:391–429, 1993.

[Cla94] C. Clack, C. Myers & E. Poon. *Programming with Miranda.* Prentice Hall, 1994.

[Joo93] S. Joosten, K. van der Berg & G. ven der Hoeven. Teaching functional programming to first-year students. *JFP*, 3:49–65, 1993.

[Lam93] T. Lambert, P.Lindsay & K. Robinson. Using Miranda as a first programming language. *JFP*, 3:5–34, 1993.

[Mol93] P. Molyneux. Functional programming for business students. *JFP*, 3:35–48, 1993.

[Mor82] J. Morris. Real Programming in Functional Languages, in J. Darlington P. Henderson & D. Turner *Functional Programming and its Applications*, CUP, 1982.

[Mye93] C. Myers, C. Clack & E. Poon. *Programming with Standard ML.* Prentice Hall, 1993.

Appendix

Courses at the University of Westminster

1. **Title**
   (a) Functional Programming.
   (b) Functional and Application Techniques.
2. **Aims**
   (a) A specialist course for final year undergraduates. It introduces the functional programming paradigm, and contrasts this with the imperative programming paradigm. The course has a heavy practical emphasis.
   A similar but lower level course has also been taught to Information Systems students, who generally only have first year programming skills.
   (b) An advanced MSc course. In addition to the objectives in (a), it compares the major functional styles and offers some theoretical underpinning.
3. **Audience**
   Both courses are aimed at Computer Scientists and Software Engineers.
4. **Level**
   (a) Final year undergraduates.
   (b) Advanced MSc students.
5. **Prerequisites**
   (a) A good foundation in imperative programming, data structures and data abstraction together with an introductory course in formal methods.
   (b) An honours degree or equivalent in Computing or a related discipline.
6. **Recommended Textbook**
   C. Clack, C. Myers & E. Poon [Cla94].
7. **Duration**
   Both courses are 12 weeks, with 3 lectures and one tutorial class per week.
8. **Assessment:**
   Both courses: 30% coursework, 70% examination.

Courses at University College London

1. **Title**
   (a) Introductory Programming.
   (b) Functional Programming.
2. **Aims**
   (a) Foundation first-term first-year undergraduate course. It aims to provide a solid introduction to problem-solving and the expression of solutions in algorithmic terms. The course has a heavy practical emphasis.
   (b) Second-term option for conversion MSc students. It provides an alternative programming paradigm, improving problem-solving and programming skills. It also introduces implementation methods.
3. **Audience**
   (a) Taken by students on the following undergraduate degrees: Computer Science ("CS"), Maths & CS, CS & Electronic Engineering. In the past, also by CS and Cognitive Science, and some affiliate students.
   (b) Taken by students with a non-CS first degree. These students have a variety of backgrounds, ranging from Fine Arts to Physics.
4. **Level**
   (a) First year undergraduates.
   (b) Conversion MSc students.
5. **Prerequisites**
   (a) None. ('A' Level Maths and some trigonometry preferred).
   (b) A non-CS honours degree. Reasonable facility in Maths and capacity for abstract thought.
6. **Recommended Textbook**
   (a) C. Clack, C.Myers & E. Poon [Cla94].
   (b) C.Myers, C. Clack & E. Poon [Mye93].
7. **Duration**
   Both courses are 11 weeks (in a 12-week term), with 3 lectures per week. The BSc course also provides one tutorial and one laboratory class per week.
8. **Assessment**
   (a) 20% coursework, 30% project, 50% examination.
   (b) 25% coursework, 75% examination.

This article was processed using the LaTeX macro package with LLNCS style