J. Functional Programming 3 (1):35–48, January 1993

Functional programming for business students

PHIL MOLYNEUX

Kingston Business School, Kingston University, Kingston upon Thames KT2 7LB, UK
(*e-mail:* molyneux@kingston.ac.uk)

Abstract

A functional language, Miranda, is being used on an introductory programming course for business students. This paper describes the rationale for such a course and choice of language. An application in the area of operations management, which is used in teaching, is given as an example of the benefits of using a functional language in this area. The reaction of the students and staff to this (for them) new paradigm of programming is described. Some conclusions are drawn for computing education for non-specialists.

1 Introduction

This paper discusses the use of a functional language, Miranda[1] as a teaching vehicle for the introduction of some programming concepts to non-computing science students, namely some of the business students at Kingston Business School. Section 2 describes the reasons for introducing a functional style of programming, and why students in a business school have a need to learn any sort of programming. Section 3 gives an example of using functional programming to animate a topic in operations management to illustrate the advantages of such an approach to non-specialist students. Section 4 discusses the reactions of students and staff to the introduction of functional programming, and outlines some of the successes and difficulties that have been encountered. The final section discusses some of the features and issues that may aid the spread of functional languages beyond computing science departments and research laboratories.

The particular course which uses Miranda is the undergraduate degree entitled *Business Information Technology*, which aims to produce students with hybrid skills in business, computing and management science. This paper is based on the experience of introducing functional programming to three cohorts of undergraduates at both first and second year level from 1990. We hope that our experience may be useful for those teachers who might be considering using a functional language on an introductory course but wish to have some evidence that such a teaching vehicle can be used on a non-specialist course.

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

2 Why should business students learn about programming?

It may seem strange that business students should be introduced to programming at all, whether functional or conventional. However, the enormous growth of the personal computer market in the 1980s has lead to the rapid spread of computing of various sorts throughout many companies. This has led in turn to the demand for students from courses such as ours to be familiar with a variety of computing concepts. The ease of access to many common business packages and their apparent ease of use at a superficial level has dangerously misled many students and companies to develop many poorly programmed applications. Whilst the activities of the 'business PC programmer' would be regarded as 'programming in the small' by professional software engineers, the business students themselves tend to acquire their perception of software development and information technology through these activities. In turn this will flavour their judgement later at a management level when involved in specification of systems involving IT. The programs and applications our students write may not be 'safety-critical' but may well end up being 'company-failure-critical'.

Some authors of software engineering texts who classify the history of computing by generations of programmers rather than languages or hardware assert that with the advent of the PC we are in danger of producing a huge number of amateur programmers (see, for example, Macro and Buxton, 1987). We believe that they are right, and hence have included courses on software development for our students. Having made the decision to include programming courses explicitly, we then face the usual syllabus design problem of resolving the tension between course *content* and course *context* – how can we devise examples that will appear relevant to business students while exposing clearly the concepts we wish them to learn ?

One of the main activities of our students is to construct models. The domains of interest may be largely connected with financial transactions but the principles of building models are the same as in any other subject. They need to learn how to design relevant abstractions of information from inputs to some real world system; to represent and transform the information to predict the system outputs, and finally to assess the quality of the model. By obliging the students to take a programming course we force them to attempt to understand how models can be represented.

2.1 Why choose a functional language?

If business students are to learn some programming concepts, why should we choose a functional language ? Subject to the constraints of the background of our students and their perceived interests, we need a language that has sufficient expressive power to enable students to implement their models quickly, yet has general abstraction mechanisms and elegant ways of combining small modules into larger modules without a large syntax to be learnt. The embedded programming languages of common business packages such as spreadsheets or databases could not be seriously considered for this role. Any usage of the macro languages of spreadsheets seems to induce a sense of *deja vu* for those who have previously encountered assembly

language programming – we would not want our students to have to start from this perspective. As Papert (1980) has argued, it is important to expose the students to 'powerful ideas' through use of a computer language – the commonly used packages and languages in business seem to have developed in a way which makes it hard to use them to expose ideas about modularity of design, abstraction and reasoning about programs.

When the Business Information Technology course was launched in 1986, the syllabus proposed Pascal as a teaching vehicle. Pascal at the time had the advantage of its tradition as a teaching vehicle with the wide availability of texts, but, of course, all the disadvantages of encouraging an imperative style of programming and the need to express abstract data types via pointers and records or arrays. In retrospect, since few members of the department were familiar with functional languages there needed to be a device to enable a transition from Pascal. This was achieved by adopting a functional style in Pascal as far as possible (see Harrison, 1989) for an example of using this approach on a data structures course in Modula-2). This was one way in which some staff were persuaded that moving to a functional language that more directly supports a functional style can be a natural evolutionary step rather than a radical jump. This is a point emphasised also by Hudak (1989) in his section dispelling myths about functional languages.

Other languages were considered: COBOL and C were rejected as steps in the wrong direction in spite of requests for them from employers. Prolog has been and is being used with the students, but whilst it is relatively easy to introduce simple database examples or list manipulating programs, many teachers have found that anything much more advanced becomes extremely difficult for novice programmers.

In spite of the large number of available texts for LISP compared to those for modern functional languages, we did not consider using LISP since it would have been difficult to discuss important issues such as data types without a metalanguage (see Wadler, 1987) for a convincing discussion of the choice between LISP and a modern functional language).

In 1990 the course was due for a review, and this provided an opportunity to begin the movement to a new language. At the same time Kingston had version 2 of the Miranda system available; there were articles to help convince colleagues of the virtues of a functional style and language (see, for example, Hughes, 1989 and Hudak, 1989), and there was also an undergraduate text available (Bird and Wadler, 1988) using a Miranda-like notation. There were also other texts such as Reade (1989) and Wikstrom (1987) which could provide back-up material even though they used a different functional language, SML. Hence it was decided to introduce Miranda as an introductory teaching vehicle to students in their second year with a view to moving it to the first year if it seemed appropriate. Miranda has been used in the second year from 1990 and in the first year from 1991 as the students' first explicit exposure to programming. The students are also introduced to command languages, spreadsheets and other business packages which could be regarded as having programming aspects, but these are not developed explicitly.

Of course, the introduction of a new language requires the preparation of examples that students can either immediately understand or at least perceive as in some sense

relevant. In the next section there is an example of using Miranda to produce a solution to a simple problem in operations management. This aims to demonstrate that the description of the solution in a functional language can be written in a style similar to that which students would see on an operations management course, and thus provide a powerful link across the degree. The following section describes the reactions of students and staff.

3 An example from introductory teaching

As with all teaching there is a certain tension between concepts and context. The example given here is one of several that we use to demonstrate that after only a few weeks of study of Miranda it is possible for students to implement algorithms used elsewhere in their degree courses. Whether or not the students see the value of a functional style in itself, examples such as this one have great value in providing a concrete link between algorithms and models as expounded in textbooks and commercial packages, which tend to hide the underlying models.

In this example we develop a Miranda script to perform some critical path analysis (CPA) calculations. The technique used is encountered by the students in courses on operations management or operational research and is covered in all the introductory textbooks (see, for example, Taha, 1992 or Wilkes, 1989). Many texts on data structures in an imperative language also cover critical path analysis (see, for example, Horowitz and Sahni, 1984), and there are several commercial packages – our students would currently use Hoskyns Project Manager Workbench. There is generally a large gulf between the presentation of critical path algorithms on paper and either their implementation in Pascal or the various commands of the commercial package. Indeed, a student may concentrate on the presentational aspects of the commercial package to the detriment of gaining an understanding of what the underlying model is about. Instead of form following function, form dominates function.

The advantage of using Miranda to implement the CPA algorithms is that the implementation can indeed look very similar to the descriptions in the operational research texts. Furthermore, the example can be used to motivate the discussion of the different ways that a project can be represented, and in turn this leads to the discussion of the representation of graphs and graph algorithms.

The following is an abbreviated version of the example as it is given to the students.

For scheduling purposes a project is regarded as a set of activities which have names, times and a list of activities which must precede them. We choose to represent the project as a graph with nodes as activities and edges as precedence relations. In the course the students are also given the formulation of the algorithms with edges representing activities and the nodes maintaining precedences. One of the confusing aspects of this topic for many students is that some texts represent the project one way and some the other, but very few discuss the choice. If we choose to represent activities as edges we may have to introduce dummy activities to maintain precedence relations. If we choose to represent activities as nodes it may mean that

there is more than one activity with no successors, which requires a little care in the formulation of the algorithms.

We choose to represent a graph as a list of tuples in Miranda, with each tuple having an activity name, its time and a list of predecessor activities. The Miranda definitions are:

$$activity\_name ::= A \mid B \mid C \mid D \mid E \mid F \mid G \mid H$$

(In practice we use actual names from a sample project but here we are using names as keys.)

$$\begin{aligned} time &== num \\ predecessors &== [activity\_name] \\ activity\_info &== (activity\_name, time, predecessors) \\ graph &== [activity\_info] \end{aligned}$$

The critical path analysis method has to schedule activities so that the earliest time an activity can start is late enough to allow all predecessors to finish, and the latest time an activity is allowed to start is early enough so that all successor activities can start in time not to delay the whole project beyond its minimum completion time. Extensions to the problem can take the form of other constraints such as resources available for each activity. A critical path is a path of longest duration through the graph, and the shortest time to complete the project is the maximum earliest finish time of those activities with no successors.

In the typical operational research text the above is translated into a series of calculations for earliest and latest activity start and finish times similar to the following:

activities $a(1) \ldots a(m)$
est $(a(p))$ represents Earliest Start Time for activity $a(p)$
lst $(a(p))$ represents Latest Start Time for activity $a(p)$
eft and *lft* refer similarly to finish times

The essential computations are expressed as follows:

est $(a(q))$

$=$ 0
if $a(q)$ has no predecessors

$=$ max (*est* $(a(p))$ + duration of $(a(p))$)
over all $a(p)$ which are predecessors of $a(q)$

lst $(a(p))$

$=$ min (*lst* $(a(q))$) − duration of $(a(p))$
over all $a(q)$ which are successors of $a(p)$

$=$ max (*eft* $(a(k))$) − duration of $(a(p))$
over all $a(k)$ which have no successors
if $a(p)$ has no successors

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

$$\begin{aligned} eft\ (a(p)) &= est\ (a(p)) + \text{duration of } (a(p)) \\ lft\ (a(p)) &= lst\ (a(p)) + \text{duration of } (a(p)) \end{aligned}$$

The precise notation will vary from text to text, but the above generally reflects what will be found. The presentation would, of course, need to be accompanied by worked examples and diagrams.

The above can be translated in Miranda fairly directly after an elementary introduction, including list comprehension notation. In the Miranda notation below, the definition of *duration_of*, *preds_of*, and *succs_of* (which calculate for an activity its duration, list of predecessors and successors) is deferred until the main functions have been defined. The graph is chosen to be included as a parameter to generalise the functions

```
est  ::  graph → activity_name → time
est proj act
     =  0 ,        if  # preds = 0
     =  max [ est proj act' + duration_of proj act' | act' ← preds ],
                   otherwise
        where
        preds = preds_of proj act

lst  ::  graph → activity_name → time
lst proj act
     =  min [ lst proj act' | act' ← succs ] − duration_of proj act,
                   if  # succs > 0
     =  max [ eft proj act' | act' ← nosuccs proj ] − duration_of proj act,
                   otherwise
        where
        succs = succs_of proj act
```

nosuccs proj returns a list of names of all activities in *proj* which have no successor

```
          eft  ::  graph → activity_name → time
eft proj act   =   est proj act + duration_of proj act

          lft  ::  graph → activity_name → time
lft proj act   =   lst proj act + duration_of proj act
```

The remaining functions can be defined as follows

```
duration_of  ::  graph → activity_name → time
duration_of proj act
     =  hd [ tm | (act', tm, ps) ← proj ; act' = act ]

preds_of     ::  graph → activity_name → [activity_name]
preds_of proj act
     =  hd [ ps | (act', tm, ps) ← proj ; act' = act ]
```

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

```
succs_of      :: graph → activity_name → [activity_name]
succs_of proj act
              = [ act' | (act', tm, ps) ← proj ; member ps act ]

nosuccs       :: graph → [activity_name]
nosuccs proj
    = [ act | (act, tm, ps) ← proj ; # succs_of proj act = 0 ]
```

The critical activities are those that cannot be delayed without delaying the whole project, and hence

```
crit_activity      :: graph → activity_name → bool
crit_activity proj act = (est proj act = lst proj act)
```

The subgraph which is the critical path (or paths) can then easily be calculated

```
project_activities :: graph → [activity_name]
project_activities proj
    = [ act | (act, tm, ps) ← proj ]

critical_activities :: graph → [activity_name]
critical_activities proj
    = [ act | act ← project_activities proj ; crit_activity proj act ]

critical_subgraph :: graph → graph
critical_subgraph proj
    = [ (act, tm, crit ps) | (act, tm, ps) ← proj ; member crit_acts act ]
      where
      crit ps   = [ act' | act' ← ps ; member crit_acts act' ]
      crit_acts = critical_activities proj
```

The above example requires only a basic introduction to Miranda and some familiarity with list manipulation and list comprehensions. This means that the students can be shown examples which implement techniques that they perceive as relevant and useful elsewhere in their studies. Furthermore, the questions of efficiency and representation of algorithms and data structures can be explored by building on examples such as this. Whilst there may be issues about expressing graph algorithms in general in a functional language in ways which are both elegant and efficient, I would claim that at least we have a starting point for students discussing such issues. (For a lively discussion of this see the items on the Internet newsgroup comp.lang.functional in May 1991.)

With a small amount of extra effort, students can be asked to extend the example in a variety of ways. Possibilities include writing functions to check that the data does not represent an invalid project with cycles; writing functions to investigate the addition of resource constraints on activities; finding the second longest or near critical paths (important if uncertainty is introduced into the activity durations).

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

3.1 Other Examples Used

Other examples of a similarly introductory nature and size have been used for other areas. Ironically, in some of our teaching of business topics to engineering students we have been obliged to use some of these Miranda scripts since these students had little access to relevant PC business packages. The more interested students, who investigated beyond merely using the Miranda system as a calculator for the application area, were surprised that as novices they found the code fairly easy to read. This was in spite of the fact that in the language they had been taught, Modula-2, they had regarded the manipulation of lists, trees, and so on, as a harder part of their course.

The examples used can be classified according to their purpose:

- Applications areas, like Critical Path Analysis, where the notation used in the business texts translates very naturally into a functional language and style. This has included small examples from stock control, queueing and marketing models.
- Examples, such as the calendar case study from Bird and Wadler (1988) and the character pictures script from Reade (1989), which illustrate a functional style.
- Use in teaching more conventional business packages such as spreadsheets. The advantage of implementing the solution to some business problem in both a spreadsheet and a functional language is that the functional language can highlight the general programming points which the deceptive ease of use of a spreadsheet hides.

The last category of examples is probably of particular importance for the non-specialist student. This is because combined use of spreadsheet and a functional language demonstrates the relevance of general programming concepts to the day to day usage of packages. It also helps the student think about the desirable directions for developing business software. Indeed, the growth of the amateur programmer using spreadsheets has become a problem that should not be ignored. In a recent Financial Times survey (Software at Work, 19 March 1992) it is reported that:

> Uncontrolled use of spreadsheet packages risks the danger of amateur programmers producing incorrect business models on which important decisions might be based.

The exercise combining a spreadsheet and Miranda is a typical first year financial cash flow exercise involving around 20 equations defining costs, forecast inflation, sales and so on. The students are introduced to the exercise at a time when they have met the relevant financial concepts and have been introduced to a spreadsheet. The exercise is outlined in class and the students are asked to calculate the net present value of the financial project using both a spreadsheet and Miranda. The students are required to lay out their results in a particular tabular format which, in Miranda, builds on the character picture examples that the students have met. Part way through the exercise some of the problem specifications are changed, including changes in both required layout and some of the original equations and data. In Miranda, modifying the script should be trivial (provided that the students have

been guided to produce a sensible layout function); in the spreadsheet incorrect results may be given unless the students have understood and applied some of the concepts the exercise is trying to highlight. For example:

- If a student has not used relative and absolute addressing correctly in the spreadsheet then moving a few columns and rows may give wrong (but believable) results.
- Some spreadsheets count blank cells as zero and hence missing data can give misleading results.

The main value of the combined spreadsheet and Miranda exercise to illustrate that effective use of business packages requires a disciplined usage not dissimilar to that of a general programming language. It also helps stimulate discussion about the direction in which business packages such as spreadsheets could develop.

4 Reactions of students and staff

This section comments on some of the features that facilitated the introduction of Miranda, and identifies some of the difficulties the students appear to have at an introductory level. The reactions of other staff are also outlined.

4.1 Reactions of students

The first contact a student has with a programming language is not the set of programming concepts supported but the programming environment within which they have to work. The features that strongly influenced our students reactions to Miranda were:

- The ability to evaluate expressions at the Miranda prompt without having to deal explicitly with input and output gives our students at least part of the look and feel of software they meet elsewhere. Combined with the use of an overhead projector connected to display a computer screen, class discussion of programming problems can take a more fruitful role, since the concise nature of most function definitions naturally leads to experimentation.
- We have chosen an editor that can be used in any of the operating environments available to the students. This enables the students to use the same editor whether in mail, Miranda or elsewhere. This is a huge advantage over those packages which force the user to use a different editor (however good it is) or when moving from an MS-DOS to a Unix environment.
- The availability of a complete on-line manual was also seen by the students as a major advantage. Miranda is probably the only software which offers our students such complete documentation. It has to be admitted, though, that a PC business package culture does tend to promote the usage of software without manuals, almost as an evaluation of the software.

These points may seem trivial, but the programming environment has a huge influence on the perception of the programming language by the users.

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

The course recruits 60 students for the first year, and of these about a quarter of the 1991 cohort had had some prior programming experience at school. Of those with prior programming experience most had studied Pascal, with a small number having used BASIC. There seems to be little correlation between previous experience and performance in the first year (and it would require observation of more than three cohorts of students to obtain meaningful statistical data). However, it was noted that students with prior programming experience had problems which were influenced by their previous contact with programming:

- Many of these students felt that the order of definitions and declarations in a Pascal program block is fundamental to programming rather than a feature of Pascal.
- Many, if not all, of these students have organised their programming knowledge around the syntax of the language. This should not be suprising given the way in which programming is often taught (see, for example, Linn and Clancy, 1992). This affects the student's ability to read programs in a functional language as well as write programs. For example, the '::=' sign in Miranda, which is used to introduce an algebraic data type, was often read as 'becomes' by these students, since they mistook it for the assigment operator ':=' in Pascal.
- Many Pascal texts introduce the concept of a list data structure late in the course, along with some often convoluted examples of using pointers. This may influence the students' view of the difficulty of using the list data structure in programs.
- Some of the things they thought were basic principles turn out to be dependent on features of particular languages or representation of data structures. An example of this is in the description of those sorting algorithms which can be designed by a divide and conquer technique such as selection sort, insertion sort, and so on. When the data is represented as a list it is relatively clear that the various sorting algorithms are variants on defining ways of splitting and joining sequences of data. Those students who have previously seen the same sorting algorithm with the data represented as an array will probably have memories of programs with various nested loops that do not immediately reveal the common pattern of divide and conquer.

4.2 Common notational misconceptions by students

Students still have difficulties, of course. Some arise from the nature of the problems or concepts covered; some difficulties are associated with learning a new notation. Recounted here are some of the difficulties commonly encountered by our students which are concerned primarily with notation or style. It may be that these difficulties are common to all novice functional programmers or just to non-computing science students – it would be useful to be able to compare the reactions of different groups of students.

Many of our students (typically over half each year) would have Advanced level

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

school maths and all would have a good grade at Ordinary level (or GCSE) school maths. However, most of the students identify the term 'function' with 'polynomial' or 'formula' in an engineering maths sense. Hence they have a certain disbelief that functions can be defined to manipulate character pictures or output the critical path for a project network. Associated with the students' view of the concept of a function is the belief that all functions are called f (or possibly g or h) and the argument to a function is called x (or possibly y or z). It is a great pity that some account of the history of mathematical notation is not given along with the mathematical concepts, since this may add some life to the symbols on the page. A little history can (lightheartedly) be made relevant even to business students, since mathematical notation must be one of the few areas which has been almost a genuine free market. We have tried therefore to introduce some historical context. For example, we mention that x, y, z were first used as notation for unknowns by Descartes in the 1600s, and Euler and Lagrange first used f as the name for a function in the 1700s (see Cajori, 1928).

Novices frequently forget that function application is more binding than anything else, left associative and denoted by juxtaposition. This leads to errors such as writing $f\ a : xs$ instead of $f\ (a : xs)$, $f \cdot g\ x$ instead of $(f \cdot g)\ x$ and $f\ g\ x$ instead of $f\ (g\ x)$. Until students are convinced of the usefulness of some higher order functions and partial application of functions they regard the notation as strange. Indeed, most students perceive the brackets and comma in the school notation of $f(x, y)$ as in some sense stopping the function and argument names from bumping into each other.

A further point about the notation for function that some students have raised is why function application is denoted fx rather than xf. The students point out that the diagrammatic representation would show a set containing x on the left and the result of applying f to x as an element of a target set on the right . Similarly, if one uses data flow diagrams to represent the linkage between processes, then the data flow is read from left to right, whereas the resulting function composition is read from right to left. Mathematically trained people would probably not regard such an issue as a problem.

We encourage students to explicitly specify the type of every top level function, even the most trivial ones. This is because until the students have some familiarity with the style of error messages and confidence with the language, they generally do not have a feel for tracking down their errors. Consider the following erroneous definition of a function which a student intends to take a character and return *True* if the character is a lower case letter:

$$lowerletter\ x \quad = \quad \text{'a'} \leqslant x \leqslant \text{'z'}$$

The definition has a simple error of the kind easily made in typing (with fingers) the definition. The definition should have been:

$$lowerletter\ x \quad = \quad \text{'a'} \leqslant x \leqslant \text{'z'}$$

Without an explicit type specification, the compiler will deduce that *lowerletter* has a perfectly valid definition but is of the type $(char \rightarrow char) \rightarrow bool$ instead of the

https://doi.org/10.1017/S0956796800000587 Published online by Cambridge University Press

intended *char* → *bool*. Only when *lowerletter* is used elsewhere in other function definitions will error messages be generated. Novice programmers tend not to realise that it is unreasonable to expect the compiler to track down the true source of error and tend to misinterpret the error messages.

4.3 Reactions of staff

The dissemination of a new language and style of programming beyond computer science departments, which invented the concepts, requires not just the demonstration of the intellectual quality of the ideas, but also socialisation into a culture. Kingston Business School has a tradition of having a more mathematical and computing orientation than many other Business Schools in Britain. This provided the motivation for the investigation and acceptance of a functional language by some staff through small examples relevant to their areas of interest. The readability of small Miranda function definitions which require minimal explanation of syntax greatly helps to demonstrate the usefulness of the language.

Almost inevitably, there are some staff who regard programming as *merely* technical activity to be subsumed and abandoned by the higher echelons of strategic thinkers. To this audience, programming is not a problem because somebody else will always do it. We directed staff with views of this nature to recent articles on software engineering education, and in particular to Dijkstra's 1989 article, 'On the Cruelty of Really Teaching Computing Science'. Especially relevant are the sections of Dijkstra's article where he indicates that such staff may be in danger of defining software engineering as 'how to program if you cannot'.

In the academic year 1990/91 the course received several external reviews, inspections and visits. The external review and the inspection by a computer science HMI (Her Majesty's Inspector of colleges) both accepted the move to the introduction of a functional language. [Note for those not familiar with the British education system: HMIs are part of the quality control process for schools and part of higher education. As an external and independent body of inspectors, their views are influential and determine, amongst other things, funding.]

Some criticism came from those who believed that a lack of an explicit notion of state in pure functional languages meant that such languages should not be used as a teaching vehicle for business students. This was on the grounds that most commercial programming was concerned with persistent storage or interactive I/O, and that this required the manipulation of state. We have used simple examples of interactive programs based on those that can be found in Bird and Wadler (1988) and Holyer (1991) and found them to be accessible to our students. From experience, the difficulties students have tend to be reminiscent of the difficulties they have with understanding lazy I/O in Pascal and, for simple examples, neither harder nor easier. We have encouraged interested staff to read Hudak (1989) and articles he refers to, Thompson (1990) and Grimley (1988), for further material on functional programming and state. There are areas which we will admit to not covering which are important to commercial programming. We make no mention of, for example, non-determinism, concurrency and real-time programming.

A major issue for staff is access to the literature on issues such as functional programming and state and descriptions of teaching experiences. The dissemination of example applications in many areas will be particularly important, since by this means the community of programmers who see the virtues of a functional style may be enlarged.

Conclusions

Those readers who are surprised to learn that non-specialist students are using a functional language should ask themselves what future they see for functional programming in general. I believe that the reasons for promoting a functional style and functional languages propounded, for example, by Turner (1982) or Hughes (1989) should apply to any person who comes into contact with programming. The context within which we teach, and the examples we use, may have different flavours, for different students, but the primary concepts should be the same. Ideas of abstraction, modularity of design, problem and data structuring are relevant to any student who has a need to formulate solutions to problems. In order to illustrate this we need notations that are sufficiently powerful to enable students to express solutions to problems that they perceive as relevant, yet simple enough to highlight the programming concepts that will be of value in whatever computing environment they find themselves.

Issues which we feel will affect the spread of functional programming, especially outside computing science departments, will include the range of computing environments available; a broader range of textbooks and a range of contact points for the dissemination of teaching experiences. The computing environment is important for many reasons, but for novice non-specialist users, the first impressions of the look and feel of how software can be used will be formed by comparison with the typical range of PC packages rather than by reference to how they might have programmed in Pascal.

In the relatively short time we have been using Miranda, we feel that we are able to reveal basic principles and encourage good programming practice more effectively than when using other languages or packages. It is certainly the case that we can get students to implement algorithms used in other areas of the course that could not easily have been achieved by other means. Our experience is that by using Miranda we have opened up the possibility for non-specialists of investigating computing concepts and powerful ideas.

Acknowledgements

I am grateful for discussions with Dan Russell, Richard Ennals and David Miles. I also thank the referees, who provided helpful comments. Many people in the functional programming community have helped us with knowledge of functional programming and support for our course; in particular I would like to thank Simon Peyton Jones who first introduced me to the subject.

References

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice-Hall.
Cajori, F. 1928. *A History of Mathematical Notations*. The Open Court Publishing Company, LaSalle, Il (in two volumes).
Dijkstra, E. W. 1989. On the cruelty of really teaching computing science. *Communications of the ACM*, **32** (12): December, pp. 1398–1404.
Financial Times, 1992. *Software at Work*. 19 March.
Grimley, A. 1988. *Natural Constructions in Functional Programming*. University of Kent at Canterbury, UK, Computing Laboratory Report No 53.
Harrison, R. 1989. *Abstract Data Types in Modula-2*. Wiley.
Holyer, I. 1991. *Functional Programming with Miranda*. Pitman.
Horowitz, E. and Sahni, S. 1984. *Fundamentals of Data Structures in Pascal*. Computer Science Press.
Hudak, P. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21** (3): September, pp. 359–411.
Hughes, J. 1989. Why functional programming matters. *The Computer Journal*, **32** (4): April, pp. 98–107.
Linn, M. C. and Clancy, M. J. 1992. The case for case studies of programming problems. *Communications of the ACM*, **35** (3): March, pp. 121–132.
Macro, A. and Buxton, J. 1987. *The Craft of Software Engineering*. Addison-Wesley.
Papert, 1980. *Mindstorms*. Harvester Press.
Reade, C. 1989. *Elements of Functional Programming*. Addison-Wesley.
Taha, H. A. 1992. *Operations Research: An Introduction*. Maxwell Macmillan.
Thompson, S. 1990. Interactive functional programs: A method and a formal semantics. In Turner, D. (ed), *Research Topics in Functional Programming*. Addison-Wesley.
Turner, D. A. 1982. Recursion equations as a programming language. In Darlington, J. and Henderson, P. (eds), *Functional Programming and its Applications*. Cambridge University Press.
Wadler, P. 1987. A critique of Abelson and Sussman or why calculating is better than scheming. *SIGPLAN Notices*, **22** (3): March.
Wikstrom, A. 1987. *Functional Programming Using Standard ML*. Prentice-Hall.
Wilkes, M. 1989. *Operational Research: Analysis and Applications*. McGraw-Hill.