Teaching functional programming to first-year students

STEF JOOSTEN (ED.), KLAAS VAN DEN BERG, AND GERRIT VAN DER HOEVEN

University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands
(*e-mail:* joosten@cs.utwente.nl)

Abstract

In the period 1986-1991, experiments have been carried out with an introductory course in computer programming, based on functional programming. Due to thorough educational design and evaluation, a successful course has been developed. This has led to a revision of the computer programming education in the first year of the computer science curriculum at the University of Twente.

This article describes the approach, the aim of the computer programming course, the outline and subject matter of the course, and the evaluation. Educational research has been done to assess the quality of the course.

Contents

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

1 Introduction

There is a growing interest in lazy functional programming languages such as Miranda[1] and Haskell. It is therefore obvious to investigate whether an introductory course in computer programming can be given in a functional programming language. Because these languages are so new, there are only a few places in the world where functional programming is used in this rôle.

Until 1991, the introductory computer programming course at Twente was based on imperative languages, i.e. Pascal and Modula-2. A decision to switch to functional programming was rather drastic, and has been taken with great care. A period of five years has preceded the introduction, in which extensive experimentation and evaluation went together with careful planning and decision making. The functional programming course has been conducted four times in experimental form, with 30 to 40 participants each year. By now, the course had found its definitive form, and was introduced for all computer science students at the start of the 1991/92 curriculum. As a consequence, a large amount of didactic experience has been built up on teaching functional programming as a first language.

In this article we want to motivate the choice for lazy functional programming for the introduction to algorithmic thinking. The new programming course is described briefly. The following questions will be answered:

- What is the aim and the subject matter of the introductory computer programming course?
- Why did we choose this approach?
- Which problems occurred and how did we solve them?

1.1 Motivation

Research on functional programming has been conducted at the University of Twente from 1982 onwards. Part of this research was directed towards using the functional languages in practice (Joosten, 1989). The idea to introduce our own freshmen to computer programming by means of a functional language dates back to 1986. Although many thought of it as unrealistic, we could think of many reasons why this was a good idea. Some years later many of these reasons still stand. We mention the most important ones.

The concept of the algorithm is introduced with a minimum number of distracting elements such as redundant syntax, details about the order of evaluation, and exceptional situations to keep in mind while programming. Much better than in imperative languages, a functional language enables you to denote appropriate abstractions. Clear and concise programs can be written that express the essence of the algorithm, and nothing more. Such properties have created the necessary room in the course to concentrate on design issues rather than on language details.

As imperative programming still dominates this field, we also want to educate our students in an imperative language. We have noted that knowledge of two

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

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

language families at such an early moment improves the attitude of students towards programming languages. Uncritical language adoration makes way for a more objective attitude.

Functional programming offers a suitable starting point for many fields, such as computer algebra, artificial intelligence, formal language theory, specification, etc., and appealing applications are within the reach of students sooner.

1.2 The students

The course is designed for freshmen students in computer science. Most of them are 18-19 years of age. At high school, all students have taken mathematics and physics classes. Few of them have had previous exposure to computer programming, although many have used a computer in one way or another. Since Dutch universities do not have admission examinations, the level of the freshmen cannot be influenced directly by our department.

Most of our students find jobs in business information technology (approx. 50%). The other students find jobs in many different fields, such as process control, science and education, telematics, or research.

2 The computer programming course

In this section we describe the structure and the contents of the computer programming course. After a general introduction, each part is discussed in more detail.

The aim of the course is to introduce students to the concept of the algorithm and data abstraction for the purpose of designing software on a realistic scale. After successful completion of the course, the student must be able to

- design an algorithm solving a practical problem
- prove that an algorithm satisfies its specification
- reproduce and apply a number of standard algorithms (e.g. backtracking, combinatorial algorithms on graphs, sorting)
- design software 'in the large' by means of data abstraction (i.e. modularisation)
- implement separate modules and integrate them with modules built by fellow students to create a correctly functioning system.

Formal and practical aspects are involved in this. Students must translate a practical problem into an algorithmic notation. At the same time, they must apply formal techniques to prove the correctness of an algorithm and to transform it into an equivalent algorithm. Moreover, students are familiarised with design aspects.

The whole course takes one year and consists of three terms. A term consists of eight weeks of scheduled activities followed by four to five weeks of 'free' time to prepare and take examinations. In this section, the computer programming course is described term-by-term. Subsequently, the instruction material is addressed.

The form of instruction is similar in each of the three terms: lectures (eight weekly sessions of two hours) tutorials (12 sessions of two hours during eight weeks) and practicals (laboratory assignments) (eight weekly sessions of four hours) On average, a student spends about 50 hours on self-study, involving homework, exam

preparation, reading, etc. The practicals are obligatory. A student spends about 125 hours studying in total during each term.

2.1 Functional programming

In the first term the students became acquainted with algorithms expressed in Miranda (Turner, 1986). The subject matter covers most of Bird and Wadler (1988). At the end of this term the students have written many different algorithms in a functional language, the complexity of which is comparable to quicksort, tree traversal, folding with minor pitfalls, and the like. Also, they have designed and built a few larger programs of a more complex nature. Students have shown that they can define one function in several different ways, for example recursively, with list comprehension or with standard functions. Also, students have made several proofs based on structural induction. They have diagnosed errors in given definitions. Finally, they have translated a number of practical problems to suitable data structures with accompanying functions. The remaining skills have been demonstrated in practical work. Most of these skills are tested by means of examinations.

In the tutorials, many small exercises are done to make the theory operational. The tutorials offer a lot of practice in theoretical issues such as proof techniques. Examination results show that students cope with proofs well.

In the laboratory students solve realistic problems. The first sessions comprise small exercises that are intended to familiarise students with the language. These exercises are done individually. Solving 'realistic' problems starts about half way through the first term. From that point students work in pairs. There is supervision (one supervisor per 12 students) to prevent a pair of students from becoming stuck for too long. Otherwise, they can just carry on and solve their own problems.

The first problem solving assignment is one in which students have to design the contents of a file containing information about a given situation. This file is built as a list of n-tuples, and contains (depending on the concrete assignment of each student) family relations, football results, ingredients for cooking, and so on. The students have to write a program to provide answers to questions like which teams have lost a football match at home? Such problems can usually be solved with a 'one-liner' that uses a list comprehension:

$$
\begin{array}{lcl}
lost & :: & [footballresult] \rightarrow [team] \\
lost\ results & = & [home \mid (home, visitor, scoreH, scoreV) \leftarrow results; \\
 & & scoreH < scoreV]
\end{array}
$$

Usually it takes a while for students to discover that the problem can be solved in such a simple way. Each student writes a program to answer approximately four such questions. The student has to create an input file with test input, and make the whole thing work. This assignment is illustrative for the other assignments. Other assignments include a modification of the calendar program from Bird and Wadler (1988) according to a given requirements specification, interactive programming and writing a program to manipulate tree structures. By means of the lab assignments, students develop a reasonable experience in problem solving and programming.

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

Compared with the 'old' curriculum, students solve more problems of a more complicated nature.

The reader may appreciate that we disagree with the popular belief that functional programming would be more theoretical (than imperative programming).

2.2 Imperative programming

In the second term students learn imperative programming. The students must learn to write a good, conventional style imperative program. However, in presenting the material we benefit from the abilities acquired in the first term. One reason to expose students to a second language early is to prevent them from acquiring unmotivated preferences for 'their' language.

We have chosen Modula-2 instead of Pascal, because Modula-2 offers standardised support for modularisation. Abstraction being a major issue in this course, it is desirable to have a language that supports modularity well.

The imperative course must ensure that the skills of students with respect to imperative programming are at least equal (if not better) than the skills of students in the old curriculum. In that sense, this is an ordinary programming course. However, the approach is different because one can take advantage of the functional programming skills acquired so far. For example, recursion is not treated as a separate subject, but is used without introduction. Procedures as parameters are used from the very beginning, because students are used to higher order functions. Function procedures are used frequently. Functions yielding composite types as a result are supported, although this is not a standard Modula-2 facility. Standard operations such as arrays, lists and trees are offered in reusable modules. These operations correspond, as much as possible, with operations already known to the students from the first term. By using the built-in operations, students are trained to solve problems at a higher level of abstraction. They adopt this style in the way in which they define their own functions.

As mentioned previously, abstraction is a big issue in the second term. Students must learn to abstract from concrete aspects, and find the right abstraction level to express a problem. They either use or define the proper procedures to reach that level of abstraction. They learn to lift a set of standard operations to a new set of standard operations that allows them to solve their problem adequately. The concept of abstract data type is treated in Miranda and Modula-2 in parallel.

The difference with functional programming is emphasised by reasoning about programs in conventional state semantics. Students are taught to reason about programs in terms of state assertions (Floyd-Hoare logic). They learn to consider the control flow explicitly, and to make decisions about the representation of data. These issues (control flow and data representation) remain implicit in the functional world.

2.3 Programming techniques

In the third term, functional and imperative programming are used in the context of software design. The subject matter in this term consists of two parts. The first

part is a treatment of programming techniques, standard methods and categories of problems. Students learn to use backtracking and branch and bound techniques, pattern recognition and parsing, finite automata in dialogue construction, sorting and shortest path algorithms. For each of these topics the same aspects are treated:

- standard techniques and algorithms
- complexity considerations
- relation between functional and imperative programming
- data abstraction
- documentation.

In the second part, students carry out a programming project. The students are confronted with a system that consists of 10 modules. A prototype system, written in Miranda, is available for experimentation purposes. This system is written entirely in the functional realm. The students learn about the system by studying it, and making their own version of the dialogue specification.

Then they get a partial implementation of the same system, written in Modula-2. Two modules have to be added to this system in order to complete it. Certain data structures are implemented 'invisibly', so students are confronted directly with the consequences of abstract data types.

Finally, the students integrate their own modules with the modules of other students, yielding a complete system written by several people independently. Section 4 contains a detailed description of this assignment.

This approach has advantages over letting students make a full program from scratch. In this situation students have to delve into existing software, which confronts them with important issues like maintainability, the role of specification, etc.

After completion of this third term the student is able to:

- specify a practical problem in the form of an initial algorithm
- transform the initial algorithm to an efficient algorithm
- convert this algorithm to an imperative implementation
- document the design process.

2.4 Instructional material

Much effort has been paid to the development of instructional material. Not only have we looked carefully at textbooks, but we have also paid a lot of attention to other kinds of written material, to support students as well as instructors.

As textbook for the first term we have chosen *An Introduction to Functional Programming* by R. Bird and P. Wadler (Bird and Wadler, 1988). Although we do not advocate it for self-study, this book has about the right mix of practice and theory. We feel it is important to use a textbook that does *not* deal with implementation of functional languages.

An alternative (at the time) would have been Wikström (1987). The latter has a less formal approach, and therefore it has been considered as less suitable for this term. A more recent introductory textbook is by Holyer (1991), and we are aware that a number of books are on the way.

The book written by Koffman (1988) has been chosen as a reference. In the second term, students use lecture notes on programming in Modula-2 (with previous knowledge of functional programming) (Berg, 1992) In the third term we do not use another book, but rely on our own material and the books already mentioned.

A book of exercises has been composed, partly with worked out solutions (op den Akker et al., 1992 a). This material is a student's companion to Bird and Wadler (1988). Scripts have been worked out for all lectures, tutorials and lab sessions, making explicit the aim of each session (op den Akker et al., 1992 b). This is a teacher's manual to the course. The students obtain a copy of the transparencies used in the lectures, serving as a supplementary text. These texts are all in Dutch.

In the first term students program in Miranda (Turner, 1986) in a UNIX environment. The second term the students use Modula-2 in an MS-DOS environment, specifically using TopSpeed Modula-2 (Jensen, 1988). In the third term both language systems are used.

3 Evaluations

The development of this course started in August 1986. A five-year plan was made for extensive experimentation, leading to the definitive introduction in 1991. In 1987/88 we started teaching with a group of 24 volunteers, out of some 120 freshmen of the faculty of computer science. We found that this group had scored 10% (on average) higher on the mathematics and physics examinations in high school. Also, these volunteers scored about 10% better than their peers in the other subjects taught in the first year at the university. Apparently, this group was far from representative. Therefore, this course could be used only for trying out the material. Comparative studies could not be done until the following year. In 1988/89 we composed two representative groups totalling 48 students. In 1989/90 we proceeded with two groups (in total 40 arbitrarily chosen students) in this new computer programming course. In the last preliminary year, 1990/91, the course was executed in its definitive form on two groups of 34 students in total. Over the years, the functional programming course has evolved considerably. Student results, appreciation and learning speed have improved considerably. Also, the major part of the old imperative curriculum is covered in the second and third terms of the course. From 1991 onwards, all students (up to 120 per annum) are taking this course.

3.1 Observations

Evaluation of the courses has been performed in close cooperation with the Educational Research Centre of the university. Regular discussions have been held between staff and students, instructors and educational experts, and the people carrying out the actual teaching. After each term, questionnaires have been used to measure the opinions and attitudes of students. In the first year of experimentation (1987/88), exact time measurements were performed to assess the time spent by students. In the other years a detailed estimate of time spent was asked in the questionnaires.

The students judge the course to be not very difficult compared to the other courses in the first year. The rather formal textbook in the English language, which is not native to our (Dutch) students, is not experienced as a problem. The time expenditure is in good agreement with the norm (and is even favourable compared with the programming courses in the old curriculum). In general, the students find the course pleasant and useful.

Another source of information, albeit 'soft', is the experience and impressions of participants (both students and tutors). From the open remarks on the questionnaires, discussions with students and colleagues and the performance of students at examinations, we have become convinced that students can cope with the higher level of abstraction. We think that this is an improvement over the classical programming education. The ability to make a program work by means of trial and error is less useful to students than it used to be.

Since this is a freshman's course, the department is interested to know how this course separates the better students from the poor performers. In the new course, students are selected much more on their ability to make abstractions. In the old course, we have the impression that smart programmers with insufficient abstraction ability would sometimes pass only because they can make programs work.

3.2 Problems

Over the years we have encountered problems that have to do with the way in which functional programming is taught. Such problems were foreseen. In 1986, functional programming was taught only as a facultative subject for students with reasonable experience in imperative programming. Not much instructional material was available in 1986 (cf. Bailes and Salzman, 1989; Savitch, 1989), and similar courses are mostly of a more recent date. So, the course and the material have been developed from scratch. Teaching it to students with no previous exposure to programming was considered risky, because functional programming has a reputation for being difficult. The importance of a freshman course for the entire (4-year) curriculum is such that a lot of time was needed to pilot and introduce the course. This created the opportunity to analyse educational problems properly, and to determine good solutions. Three of these problems are discussed in the following sections.

3.2.1 Priority and associativity

Many problems in understanding Miranda expressions in the first courses were connected with the priority and associativity-rules and the placing of parentheses, especially with the 'invisible' function application operator. Students have a hard time getting used to the way in which operators interact with function application. For example $f.g\ x\ y$ is often read as $((f.g)x)y$, whereas it really means $f.((g\ x)\ y)$.

To solve this problem we have introduced special exercises to train this ability. Furthermore, we use the @-symbol during the first weeks if we need an explicit denotation for function application. This helps when students have problems with the implicit presence of the application operator. For students who grasp the idea

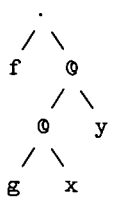

Fig. 1. Syntax tree.

right away, we use normal notations only. Furthermore, we tend to draw syntax trees to make the parsing of an expression explicit.

The role of parentheses is explained in connection with these trees. So, $f.g\ x\ y$ is written as $f.g@x@y$ or with parentheses $f.((g@x)@y)$. The corresponding syntax tree is drawn in Fig. 1.

3.2.2 Type expressions

In the beginning we had much trouble with Miranda's types. Students made many mistakes, both in the laboratory and on paper. There are several aspects of the errors made with type expressions. Before considering solutions to this problem, we have made an inventory of the mistakes. The following categories of mistakes were identified:

Understanding given type expressions

- The function arrow is given the same associativity as the function application: $a \to b \to c$ is read as $((a \to b) \to c)$
- The main structure of the type expression in not recognised: e.g. $a \to b \to c$ is interpreted as the type of a 3-argument function.

Giving the type of a specific function

- No parentheses are placed around arguments that are functions
- Functions with more than one argument are not recognised
- The result type is replaced by some type expression of the RHS of the definition or omitted
- Too many restrictions are placed on types, e.g. all types are *num*.
- Too few restrictions are placed on types, e.g. all types are polymorphic type variables and not bound to specific type.

Miscellaneous errors

- Errors in understanding type error messages are mostly due to a wrong interpretation of terms used in these messages: cannot unify, cannot apply, cannot identify. Frequently, students do not use the actual content of the error message, but solely the indication of the place where something is wrong.

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

- Errors due to naming conventions, such as *xs* and *ys* for lists. Several students think that the computer derives the type [*] based on these names.
- Type expressions are mixed with ordinary expressions, like this for example: *last* [*] = *head* (*reverse* [*])

The problems with types were also clearly visible in the evaluation results in the first two years of experimentation (Berg and Dilot, 1989). 'Giving the type of a function' was among the first three subjects in the list of ten most difficult issues.

Major adjustments of the course have taken place based on these observations. Firstly, we relaxed the requirement that a student should be able to *derive* the type from an expression. Now we require that the students can write (as opposed to derive) the type of their own definitions. In order to give the necessary practice and to advocate good programming style, we insist that the type is given explicitly with every definition. Interpreting error messages remains a problem. Phrases like cannot unify, cannot apply and cannot identify are explained in an introductory practical assignment, which helps a little. As a result of all these measures, the topic of typing has disappeared from the top ten of difficult issues.

3.2.3 Computational model

In the experimental phase of the courses, the students received the functional and imperative programming courses in parallel. Interference of both courses has been observed, especially in the case of the computational model. The computational model for functional programming is based on rewriting and lazy evaluation for imperative programs on memory states and state transitions. Some errors occurred because students used the imperative model in the functional programming domain:

1. Some students thought that the definitions in the script should have a particular order: "otherwise the value of a variable is not known".
2. They assumed changes in the value of variables by function application; e.g. taking the tail of a list *ys* would change the list, in other words they expected the effect of an assignment *ys* := *tail* (*ys*).
3. Some felt the need to store intermediate results, otherwise these results would be lost, e.g. they wanted to save the original list before calculating the last element with *last xs* = *head* (*reverse xs*).

Apparently, some of the misconceptions are induced by imperative language use in the functional domain, e.g. names like take, drop, remove, filter could imply some (side-) effects on the argument of the function. This interference nearly completely disappeared after the imperative programming course has been placed after the functional course.

3.3 Functional versus imperative programming

One educational experiment has been conducted which is of particular interest: an experimental comparison of the problem solving abilities of students who have

been first exposed to computer programming by means of Miranda versus those who followed conventional programming education based on Modula-2. Since the functional course was given to a part of the total first-year population, we had an ideal opportunity to do comparative research (Berg et al., 1989).

In the experimental design two equivalent groups of first year students in computing science (equivalent on mathematics and physics grades) received different treatment; they took either the functional programming course or the imperative programming course. Two experimental conditions were provided: time pressure and no time pressure. Their abilities were tested before the course and after the course. In the post-test they received a number of assignments on different aspects of programming. These tests differed only in the programming language used. The experimental design is given in Table 1.

Table 1. *Experiment design 1*

	Treatment			Time pressure	No pressure
Group 1	Pretest	Functional	Post-test1	n=15	n=14
Group 2	Pretest	Imperative	Post-test2	n=11	n=10

Several aspects of the programming ability of students have been tested in the given assignments. No significant differences have been found in both conditions on the following abilities: to specify a function, to write comments to a function, to write the type of a function, to identify semantic equivalence between different program constructs, to use structured data types.

Two assignments were offered with one condition only (no time pressure). The first of these assignments (1) comprised the modification of an existing program. The second assignment (2) was the design and implementation of a new program for a given specification. The experimental design is given in Table 2.

Table 2. *Experiment design 2*

	Treatment		Assignment 1	Assignment 2
Group 1	Pretest	Functional	n=11	n=8
Group 2	Pretest	Imperative	n=9	n=10

For the modification assignment, the following four quantities were determined: the number of new local functions, the number of new global functions, the number of modified functions, and the percentage of students who modified the main function. The results are shown in Table 3.

For the design and implementation assignment, a program call graph has been derived for each solution. The following five quantities were determined: the number of user defined functions in the graph, the number of levels in the graph (transformed

Table 3. *Results: modification assignment.*

Group (*n*)	FP (11) Mean	IP (9) Mean	F	Sign.
# local new functions	0.4	0.0	4.6	0.045
# new global functions	1.5	0.3	16.6	0.001
# modified functions	0.1	0.8	17.0	0.001
modification main function	82%	22%	9.9%	0.006

to a tree by removing recursive calls), the maximum number of functions directly called by another function, the number of functions identified in the design, the coverage, i.e. the percentage of design functions recognizable in the implementation. The results are shown in Table 4. There was no significant difference found between the two groups on the maximum number of functions directly called by another function and not on the number of functions identified in the design.

Table 4. *Results: design and implementation assignment*

Group (*n*)	FP (8) Mean	IP (10) Mean	F	Sign.
# user defined functions	6.9	2.6	16.4	0.001
# levels in program graph	1.8	0.7	10.3	0.005
coverage design/program	92%	57%	3.9%	0.064

From the results for the modification assignment it can be concluded that students in group 1 (the functional programmers) introduced significantly more new functions to accomplish the required modification than students in group 2 (the imperative programmers). The latter realise the required new functionality by changing the existing program at the lowest level code. At the main level, this change is less frequently visible for these students than for students in group 1.

From the results for design and implementation assignments it can be concluded that the correspondence between design and program is significantly higher for students in group 1 than in group 2. Students in group 1 use more levels of abstractions with more functions in their programs than students in group 2.

Although it is rather subjective to derive the quality of programs from the criteria used above, it could be argued that the results of these experiments give evidence that students in the functional programming group produce programs with a better structure than students in the imperative programming group.

4 Programming project

In this section we describe a programming project that is conducted at the end of the third term. It serves the purpose of illustrating the type of assignments students do.

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

It allows the reader to form an idea of the level obtained at the end of the first year. The description starts with a discussion about the educational and organisational aspects of the assignment. After that we give some technical details.

4.1 Organisation

In the second half of the third term, students work on a larger assignment. Each student spends 16 hours in the laboratory on this assignment, divided into four sessions of four hours each. The work is done in pairs. During the first session, the students are confronted with a Miranda prototype of the system. In the end, this system will be built by those students in Modula-2. They are supposed to experiment with the prototype, to carefully analyse its behaviour, and to present the results of their analysis in the form of an external specification of the system.

The next two sessions, eight hours in total, are devoted to the implementation of parts of the system. The students will have to integrate their work with the work of others, so they realise the importance of sticking to the specification, test thoroughly and remain on schedule. The external specification is used as a starting point. Students do not use the external specification they built themselves, for that was handed in earlier. Instead, they all use the same specification provided by the supervisor. This annihilates the risk of delay for students who have had trouble making the specifications. In this way, students skip part of the design trajectory. What they are supposed to do here is just to fill in the design. This requires a passive understanding of the structure of the system; the skills required to come to a satisfactory system design by themselves are not taught nor trained in this course.

The final session is for the integration of the system parts. Four pairs of students will now merge their material into one complete and working system.

4.2 Railway information system

The students work on a restricted railway information system. In this section we give an overview of the system. We hope to give the reader a feeling for the kind of application we are talking about. This description is not intended to be complete.

The railway information system computes the price of the cheapest ticket for a given journey, taking into account possible reduced fares for reduction-pass holders, group tickets, etc. The system has two important aspects. One is the correct functionality: it should collect the proper data on a journey, and correctly compute the price of the ticket from these data and the information it has stored on the costs of various kinds of tickets. The other aspect is its user interface. It should facilitate the presentation of data on a journey, and handle errors in the input in a clear and understandable way. Any user should be able to consult the system without much explanation.

The two main requirements of the external specification are that it defines the functional behaviour of the system, and that it defines the form and the nature of the interaction between user and system. The functional behaviour is described by means of abstract data types. Students have to realise which operations are

necessary and worry about the exact content of these operations. There is a close relation between the abstract data types in the Miranda program and the modules in the Modula-2 implementation. The interaction between the user and the system is described by regular expressions over some alphabet of events. Both elements in this specification lead to a precise formulation of pre- and post-conditions, which is useful when designing the Modula-2 code.

To conclude this account of the railway information system, we present the main piece of functional program: the system function. This code was made by a student by way of specifying the interactive behaviour of the system at the global level. Students are expected to produce such code in the external specification they make in the first session of this assignment.

```
ticketPriceSys  ::  aTable → aTicketStream → aPriceStream
ticketPriceSys ticketPriceTable
                =   map (ticketPrice ticketPriceTable)
ticketPrice     ::  aTable → aTicket → aPrice
ticketPrice table ticket
 =  bp,          if n = 1
 =  min np gp,   otherwise
  where
    bp   =  basePrice table
              dist
              (sinOrRet ticket)
              (class ticket)
              (fulOrRed ticket)
    dist =  distance table (dep ticket) (dest ticket)
    np   =  bp * n
    gp   =  groupPrice table n
    n    =  numberP b
```

The presentation of these functions presupposes the introduction of types and functions, which can be done at an abstract level. Somewhere the specification must show that the following (abstract) types and functions are involved:

```
aTable
aTicket
aPrice
aNumberOfP
aDistance
aStation
aTicketStream  ==  [aTicket]
aPriceStream   ==  [aPrice]

aWay           ::=  Single  |  Return
aClass         ::=  First   |  Second
aFare          ::=  Full    |  Reduced
```

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

```
dep          ::  aTicket → aStation
dest         ::  aTicket → aStation
sinOrRet     ::  aTicket → aWay
class        ::  aTicket → aClass
fulOrRed     ::  aTicket → aFare
numberOfP    ::  aTicket → aNumberOfP
distance     ::  aTable → aStation → aStation → aDistance
basePrice    ::  aTable → aDistance → aWay → aClass → aFare
                 → aPrice
groupPrice   ::  aTable → aNumberOfP → aPrice
```

4.3 Experience

In itself, the assignment is not a difficult one. Most of the students will succeed in the integration of their own part with those of their fellow students. However, it turns out to be very illuminating in several aspects. It confronts students with their own mistakes, their lack of thorough testing, the problems caused by ill-structured code, and so on. It clearly shows the necessity to stick to specifications if you want your part of the system to cooperate with other parts. It shows that it is most useful to test parts of the system separately and thoroughly before they are put together. And finally, it confronts students with the problems of version management: it happens more than once that they start integrating versions of modules which are not the final ones, e.g. because they contain material which was put there solely for the purpose of testing.

It is worthwhile to observe the students as they work. Some of them sit down at the keyboard and do 'trial and error' development. Others sit down and think everything through, starting from the (given) Miranda prototype and the specification down to the Modula-2 code. During the integration session, the modules produced by the former students usually contain the problems, whereas the modules of the 'thinkers' often operate flawlessly.

The role of functional programming in this assignment is restricted. It is true that in this assignment, for the first time, students see a larger piece of software which performs a useful function, and which is written in Miranda. But they do not themselves develop a program of comparable size in Miranda. The skills in functional programming are used to capture the essence of the functional behaviour of a system.

4.4 Role of functional programming

There are two basically different ways of using a functional program as an intermediate step towards an efficient imperative implementation. One way is by doing program transformations, and the other way is by programming and justification. The first 'method' contains the following steps:

1. To write the functional program

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press

2. To transform this program by means of correctness preserving steps until the program is fully tail recursive without using intricacies.
3. To rewrite the result (mechanically) into an imperative language.

The second way is much more informal. A functional program is written and used as a formal specification. The imperative program is developed 'as usual', in which the experience of making the specification makes the big difference in the quality of the resulting product. Proof techniques must be used to get an a *posteriori* justification for the program.

We made the choice for the second method on a rather practical basis: the transformational approach requires a greater skill and education than the second approach. We have educated students in program transformations to a level where they can make proofs. There is no room in the first year program to enhance these skills further to a level in which transformational programming becomes feasible. In the current situation, these skills do not belong in the first year. This motivates our choice for a 'limited' importance of functional programming in the design of software.

Students who have built (their share of) the railway information system report that they appreciate what they have learnt: to capture the functionality of a system in a concise functional specification that fits on the back of a business-card. At the same time they find it useful to have the experience of successfully integrating their work with the work of so many other students. Students appreciate the value of modules and abstract data types in software design. This is an issue that is taught better by experience than in the classroom.

5 Conclusions

The design and implementation of a new computer programming course was completed successfully within the original time constraints.

Based on research done in this period, we conclude that the quality of the introductory computer programming course has improved. The students learn to handle abstraction as a design tool and are able to describe their problem formally. The skills in formal manipulations have improved. Students solve more problems that are also more challenging. Because imperative programming is still taught, no problems need to be expected with respect to the connection to other (existing) parts of the curriculum. Passing or failing of students depends more on their abstraction skills and less on their coding abilities. Appreciation of students is very high.

References

Akker, R. op den (ed.), Hoeven, G. van der, Joosten, S., and Seters, H. van. 1992. *Functioneel Programmeren, studentenhandleiding [Functional Programming, student's manual]*, University of Twente, Enschede.

Akker, R. op den (ed.), Hoeven, G. van der, Joosten, S., and Seters, H. van. 1992. *Functioneel Programmeren, docentenmateriaal [Functional Programming, teacher's manual]*, University of Twente, Enschede.

Bailes, P. A. and Salzman, E. J. 1989. A Proposal for a Bachelor's Degree Program in Software Engineering, Software Engineering Education. Volume 376 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 90-108.

Berg, K. G. van den and Pilot, A. 1989. *Functioneel programmeren 1987/88. Evaluatie van een onderwijsexperiment [Functional Programming 1987/88. Evaluation of an educational experiment]*, University of Twente, Enschede, Memorandum INF-89-6.

Berg, K. G. van den, Massink, M. and Pilot, A. 1989. *Experimentele vergelijking van het leren programming ondersteund door een functionele versus een imperatieve programmeertaal. [Experimental comparison of programming education supported by a functional versus an imperative programming language*, Talk]. Educational Research Days, Leiden.

Berg, K. G. van den. 1992. *Imperatief Programmeren*, (in Dutch). Lecture notes, University of Twente, Enschede.

Bird, R. S. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice Hall.

Holyer, I. 1991. *Functional Programming with Miranda*. Pitman.

Jensen & Partners International. 1988. *TopSpeed(TM) Modula-2*.

Joosten, S. M. M. 1989. *The use of functional programming in software development*. Dissertation, Faculty of Computer Science, University of Twente.

Koffman, E. B. 1988. *Problem Solving and Structured Programming in Modula-2*. Addison-Wesley.

Savitch, W. J. 1989. Functional Programming in Pascal? *Journal of Pascal, Ada & Modula-2* 8 (5): 35-41.

Turner, D. A. 1986. An Overview of Miranda. *Sigplan Notices* 21 (12): 158-166.

Wikström, Å. 1987. *Functional Programming using Standard ML*. Prentice Hall.

https://doi.org/10.1017/S0956796800000599 Published online by Cambridge University Press