Coverpage

Introducing Scheme in a FP-less Environment

– The Students' Opinions –

Authors: Kris Aerts, Karel De Vlaminck

Address:

Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
Fax: ++ 32 / 16 . 32 . 79 . 97
e-mail : Kris.Aerts@cs.kuleuven.ac.be

Contact Person: Kris Aerts

Introducing Scheme in a FP-less Environment

– The Students' Opinions –

Kris Aerts, Karel De Vlaminck

Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail : Kris.Aerts@cs.kuleuven.ac.be

Abstract. Since 1994 the first programming language taught to students of the faculty of Applied Sciences is *Scheme*. We briefly describe our preference for Scheme and give an overview of the course. The controversial language choice induced a lot of reactions, mainly because of unfamiliarity with the functional programming (FP) paradigm. We are very open to these reactions, and even more, performed two surveys. This article discusses the answers. We believe to be the first to actually concentrate on the students' opinion. This may be of particular interest to any teacher considering the move to Scheme or any other functional language, but also to the general FP community. In the beginning students, especially those with advance knowledge of other programming languages, are quite sceptic and even negative, but the appreciation improves as the course moves on.

Introducing Scheme in a FP-less Environment

– The Students' Opinions –

1 Overview and Scope of the Article

At the university of Leuven *(Vlaanderen - Belgium)*, we made the move from Pascal to Scheme three years ago. We believe that Scheme offers better opportunities to express mathematically correct algorithms. The core language can also be taught very quickly and is very flexible which allows us to teach different styles of programming and focus on software engineering concepts.

These are key issues when choosing for a specific programming language. We have also been continuously concerned about the quality of the course, already resulting in a number of changes, but there's a lot more to it.

People tend to (just) like what they know. Students don't know Scheme, but they do know that it isn't widely used in our country, neither in world wide industry projects, and they even tend to dislike Scheme for it. This unknown - unloved situation forces us to take care that our students are not turned away from the contents of the course, simply because of the use of Scheme. We know objectively that Scheme is a good language, but the subjective feelings of the students may differ.

If we really want functional programming to spread widely, as is the goal of this conference, we have to address this issue, and it is vital for any teacher considering the move to Scheme or any other functional language to be aware of the students' opinions.

We have conducted two surveys on how our students feel about the course in general and Scheme specifically. Some of these questions are only relevant to us, but others are important enough to be communicated on a larger scale. In this stress on the opinion of the students lies the main interest of our article. Various other papers presented the educators' view, both on their particular approach, on consequences for and errors made by the students, and on criteria for the perfect functional language [3, 4, 6, 8, 10, 12], but we believe we're the first to actually investigate primarily the students' opinion. This is of major importance as well, because the language may be extraordinarily powerful, but if it isn't appreciated, it won't be used.

Also interesting is the fact that the course is not taught solely to computer science students. It is given in the (general) first year of *Burgerlijk Ingenieur* ('civil engineering'). Only in the third year a small part (approx. 10 %) chooses for Computer Science, while others prefer mechanics, electronics, mining, architecture, ...

There are still some caveats to the article:

- The results obviously link to the specific approach used in our teaching, which may not be your favourite one (see also infra).
- The results are specifically valid in an environment where Scheme is (relatively) unknown. They would probably look different if the survey was performed at places as MIT or Indiana.
- It is a rather quantitative research. We have focused on questioning many students to get a representative view on their opinions, but did not learn in depth why individual students responded the way they did. For this, one would need a qualitative research in which less students are questioned, but more thoroughly. Based on our teaching experiences we will nevertheless try to answer some of these in depth questions too.

In the rest of this article we first motivate our choice for Scheme and give an overview of the actual course. Then follows the main point of the article: the survey, of which the results are sometimes complemented with my personal experience of guiding last years students using Haskell [7] and Fudgets [2]. We've grouped the questions and answers in a number of categories:

- **What students think about Scheme**
  A controversial choice, but appreciated in the end.
- **How do students rate their implementation capabilities?**
  It decreases as time goes by!
- **The use of computers**
  We don't use computers a lot, but students definitely want more.
- **How do they like the course?**
  Already in real life not everyone likes computers as much as we do. The same thing applies to this computer science course, but in overall the course is at least judged relevant.
- **Preference for specialising in computer science**
  A different kind of questions relates to their interest in the computer science profession.

2 Why We Chose to Go from Pascal to Scheme

The Pascal course we taught in a previous life until 1994 was quite unsatisfactory because we were more concentrating on language syntax than on software methodologies. Moreover the structure of Pascal programs didn't fully reflect the mathematically correct algorithms. We had to explain the algorithm, and then prove that the Pascal program followed this algorithm. This was double work. So we often took care that our algorithm was such that it could be directly used in Pascal.

It quickly became clear that functional languages were the way to go. Not only because we use them in our research group and other universities started using them in introductory courses, but mostly because they have little syntax and therefore allow us to quickly introduce the language and concentrate on software engineering principles the rest of the time. Another good aspect is the close relationship between algorithm, specification and implementation. This way, if the algorithm and specification are built, the implementation should not need too much effort.

Anyhow, we all ought to know *why* to choose a functional language. The question is *which one*. We opted for Scheme, mostly because it has dynamic typing. We do not need to explain all about types before we can start writing programs. For simple programs, types are not that necessary. We can quickly and intuitively start programming. We found out that even last year students, who are supposed to have at least some understanding of programming languages and types, have some difficulties at first with the strong typing practised in Haskell. So we try to prevent our students from being exposed too soon to a complete type system. Naturally, we have to admit, and the last year students agree, that strong typing offers important advantages. And as a matter of fact, one of the first changes to the course was the addition of types (but not statically checked).

We also hoped for a positive side effect: keeping the students challenged and motivated. With Pascal we had two distinct groups: those with previous knowledge and those without. The first group had no problems at all, which is good, but they learned very little. Members of the second group often lost their motivation as they saw how the others could solve anything in just a few seconds, whereas they had to suffer hard to get something that *looked* like a solution. Scheme would make sure that the first group feels they can still learn something and that the second groups doesn't feel left alone in the dark with their problems. It puts them more at the same level.

It would also make it easier to learn the first group new ways of thinking. They can be quite stubborn staying with their old – bad – habits when programming in Pascal, but when using Scheme they can't use their old methods and have to resort to the general software engineering principles we're offering them.

To summarise, we preferred Scheme for the following reasons:

- functional language
  - helps to construct mathematically correct program
  - is close to specifications
  - enables inductive programming
- small syntax
- *no* types
- also destructive updates can be modelled
- Scheme puts the students at the same level
- most used functional language in education
- availability of programming environments [1]

3 An Overview of the Course.

The course H006, *Methodiek van de Informatica* (Methodologies of Computer Science) is lectured through a series of approx. 25 *ex cathedra* lectures of 90 minutes for two groups of roughly 200 students. There is a small Pascal part at the end, but in this article we only describe the Scheme related contents.

The first introductory chapter is devoted to the notion of *algorithm*, whereas in the second chapter, titled *Modelling of Processes*, Scheme is presented (`define`, `if`, `recursion`, `let`, substitution model). The definition of procedures is considered as the first important way of abstraction. In the third chapter *Modelling of Information*, `cons` and lists are introduced. We then quickly move to a first glance at Abstract Data Types and give the small classic examples of vector and representation of time.

Efficiency is the central theme of the fourth chapter: from recursive and iterative processes to calculating time efficiency in terms of the O(.) notation.

A further means of generalisation and abstraction are higher order functions. We learn them how to define genuine higher order procedures, and how to use `foldr`, `foldl`, `map` and `filter`. We also give the definitions of these procedures.

There's also a part on sorting algorithms as they are fine examples of the power of a higher order language: one can simply include the sorting criterion as a parameter. Calculating their different time complexities is another interesting aspect.

After this side step we return to Abstract Data Types and give two more elaborate examples: algebraic expressions and sets, implemented as both (un)sorted lists and binary trees. As an introduction to Object Oriented ADT's (implemented as procedures that receive messages), we first discuss the problems associated with dynamic typing. A proposed solution is the use of message passing objects as in *Structure and Interpretation of Computer Programs* [1]. This is exemplified by the redefinition of algebraic expressions, which are implemented as *functional* objects: objects with a state that cannot be destructively updated.

[1] For imperative programming languages even better tools may exist, but in the field of functional programming, Scheme offers the most and best choices.

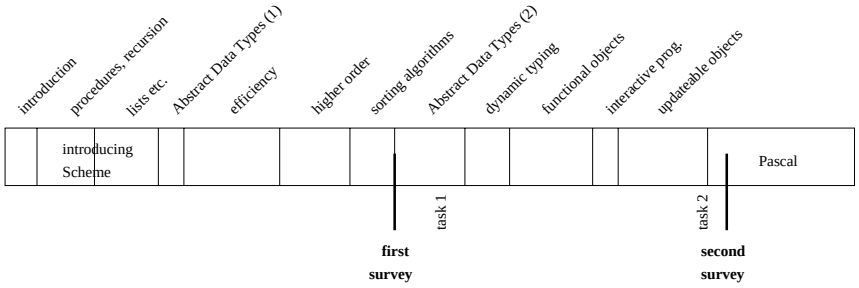

Fig. 1. A rough course time scale

The use of updateable objects (and `set!`) is explained through the modelling of bank accounts. We therefore need to get rid of the simple substitution model presented in the second chapter. We explain the real environmental model of Scheme and the impact of `set!` on it. We stress that this `set!` should only be used when defining updateable objects, and not for plain calculations.

We are not so much interested in specific algorithms. An example of backtracking (eight queens problem) was removed after the first year, because we find the implementation too technical and preferred to stress the concept of Abstract Data Types more. The solution of the towers of Hanoi is still presented, but more as an application of the chapter on interactive programming.

As for the practical side of the course, we make groups of up to 24 students and teach them a (limited) set of 12 exercise sessions of 2 hours each. Until now, we use no computer at nine of these sessions, just pencil and paper. We believe that algorithms can be better developed on paper than on computer. We want to properly teach the software engineering concepts involved in programming and are not satisfied when they have a program, that is, although working, hacked together. It needs to be constructed methodologically and this can be better done on paper.

On the other hand, real implementation work has to be performed in two larger projects that are to be handed in individually or in groups of up to 4 persons.

4 Our Survey

Instead of trying to prove the relevance of our approach by using our own arguments, we decided to let our audience speak, not literally, but by the use of questionnaires. This paper presents the results, mainly from the students' viewpoint and referring to their opinion, but also coupled to our own insights. This is necessary as students will mostly only notice the short term effects (especially when they have not yet been assessed), whereas we may as well be interested in the long term effects.

We have questioned the students twice, to see if there would be an evolution in their answers. The first time was after 8 lessons, when the concept of higher order functions was explained. At that moment they had only attended the very first exercise session and had hardly acquired any Scheme skills.

At the second time we had just started introducing Pascal. The part of Scheme that we use, was completely lectured and the students had attended ten out of twelve exercise sessions. The first (smaller) project was finished for some time and the second task was just handed out. They were supposed to have a lot more hands on experience.

We mostly asked questions regarding their personal opinion. At the first survey they marked their answer on a scale of 1 to 9. The middle answer would be 5. The second set of answers was scanned by the computer. As our computer eye is limited to 6 choices, the scale was restricted to 1 to 6 (middle score now 3.5). If we compare questions asked at both occasions, we will recalculate the results according to the 1-6 scale.

Most of the students that follow our course have little or no previous experience with (other) programming languages. Half of them has no experience at all, while about a quarter has prescience to some larger extent. As these two groups will probably have different opinions, we decided not only to view the group as a whole, but also to split it according to their advance knowledge (really high or none). Another criterion was whether they liked the course a lot or not at all. It will be obvious that the group who likes the courses gave more positive answers, but it's still worth looking at.

5 What Students Think about Scheme

5.1 Our Questions

- In the first questionnaire:
  - To me, the choice of Scheme is *1 = awfully bad, 9 = amazingly great*
  - I consider the use of brackets in Scheme as *1 = perfect, 9 = terrible*
- In the second questionnaire:
  - In Scheme one can do a lot more than I thought at first *(1 = agree, 6 = not agree)*
  - We have seen lots of programming styles (stepwise refinement, Abstract Data Types, Object Orientation (message passing style)). I think this can be done very nicely in Scheme *(1 = agree, 6= not agree)*

The item that strikes the students most in our course is the programming language. We should point out that Scheme is completely unknown to our students. Until now, even most graduated students do not know of this language. So when students asked colleagues who study computer science about Scheme, they got the answer 'Never heard about it'. Some even started thinking that Scheme was something invented by us (which, ofcourse, it is not!). The most frequent complaint is that it's useless to teach a language they won't be using later on.

But as those who complain loudly may not be representative, we had to ask the entire group.

The most striking elements in Scheme is the parenthesis, much to the annoyance of many students. So we expected rather negative answers on both questions, but were convinced that the third question would yield better results. The last question in this category is the most thrilling. We hoped the response would be positive, but had received no indications in whatever direction.

5.2 Some Criticism

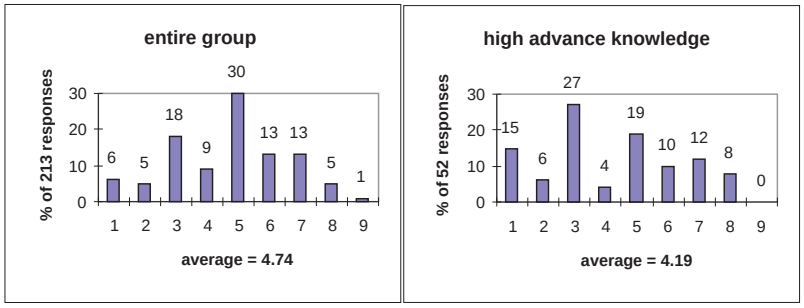

Fig. 2. To me, the choice of Scheme is *1 = awfully bad, 9 = amazingly great*

Despite the distinct number of complaints, the answer to whether they liked our preference for Scheme as an implementation language (Fig. 2), was not that negative. The slightly negative trend was mostly pushed by those with major knowledge of other languages. It is a fact that, if used to an imperative language, one has to make a notable mental switch to be able to program correctly in Scheme. One could argue that their opinion counts double as they are the only ones that know of other possible choices, but we rather believe it's due to prejudice. We have a hard time convincing these people to use Scheme properly. In the beginning, lots of them misuse `define` to write Scheme programs in a Pascal like way. And if they do not know how to program in Scheme, how can they judge it properly?

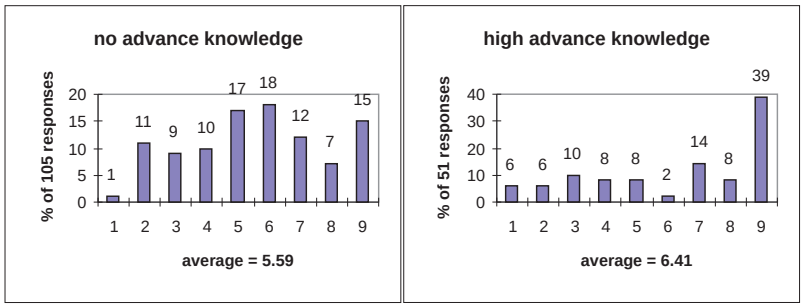

Fig. 3. I consider the use of brackets in Scheme as *1 = perfect, 9 = terrible*

Further exploring their opinion on aspects of Scheme, we especially notice that the use of parentheses is something students don't seem to like and/or understand (Fig. 3). Those without advance knowledge are best off. They are open to its use as a function evaluator, while those with advance knowledge (probably better called prejudice) do not see the light: they clearly dislike them. This is certainly a pity, because we considered it an advantage that everything in Scheme is a S-expression. This makes the language very orthogonal, but students do not seem to get this. They just see a lot of brackets that have to be placed somewhere. Lots of them open a bracket just when they *feel* appropriate, and wait with closing until the very end of the definition. This is clearly very error prone. In a negative mood, we could be tempted to state that we switched from the problems in Pascal with the semicolon, to the parentheses in Scheme. It is definitely true that some students loose the overview, purely due to the brackets.

This may seem like a trivial observation, but if this would be the main reason why students dislike Scheme, we have to take care. The source of their frustration is most likely the fact that parentheses mostly do not have a semantics as strict as in Scheme. The number of parentheses has a big influence on the meaning of the program. Another problem is that brackets are sometimes used as a means to group entities (as in the `let`), but also then, students must be aware that the grouping can just be done in one single way.

Another related problem is the use of lists. The fact that a `cons` can also create a pair, but that some pairs are lists, while others are not, turns out to be very confusing. A good solution might be to avoid the confusion by restricting the use of `cons`. We could redefine `cons` such that it can only be used to construct lists. It is quite feasible to do this such that `cons` signals an error whenever the second argument is not a list.

5.3 Some Good Points

In our second questionnaire, we included the question **'In Scheme one can do a lot more than I thought at first** *(1 = agree, 6 = not agree)'* (Fig. 4). And this question is answered quite positively (a score of 2.95). Those who have advance knowledge and didn't like Scheme at first, agree even more. But the most convinced group is those who like Scheme. They already liked Scheme, but their faith still seems to be growing.

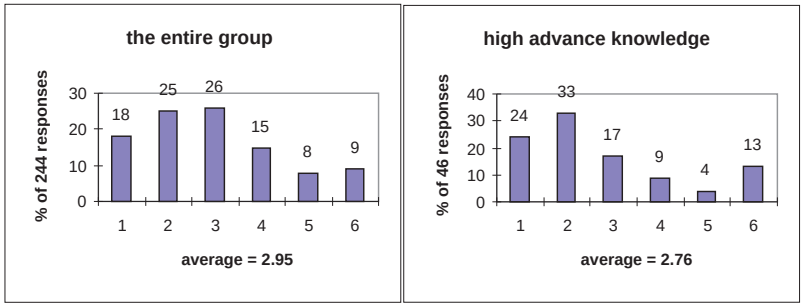

Fig. 4. In Scheme one can do a lot more than I thought at first *(1 = agree, 6 = not agree)*

The question '**We have seen lots of programming styles (stepwise refinement, Abstract Data Types, Object Orientation (message passing style)). I think this can be done very nicely in Scheme** *(1 = agree, 6= not agree)*', scores a 2.80, with identical scores if one knew nothing or a lot of other programming languages. Even those who dislike the course score 3.38, but here the histogram is more evenly spread. On the other hand, those who like it, score an incredible 1.92.

5.4 Discussion of the Results

Although the students are resistant at the beginning, the positive reactions in the second questionnaire indicate that they finally seem to agree that Scheme offers certain advantages. We believe this to be true for all functional languages.

Haskell may even do better, because it needs no or little brackets and has features such as pattern matching that enhance readability. Its list manipulation is far better and the confusion between pairs and lists is removed.

Another aspect is that programs in Scheme still really look like programs: a bit obfuscated by the use of brackets, and the control structures such as `if`, `cond`, ...that need to be written explicitly. Haskell (just as other modern functional languages) comes closer to programs looking like specifications. Because our strong interest in specifications, this is good. It stresses that we are not interested that much in programming in itself, but more in well thought out development.

Its strong typing on the other hand could make the initial steps more difficult (although one can argue about this), because we would first need to explain the type system. Still, the major reason for not using Haskell would be the fact that destructively updated objects cannot be modelled. A language with little brackets, list comprehensions and the possibility for destructive updating may be the appropriate choice.

6 How do students rate their implementation capabilities?

Questions relating to this were asked on both occasions. Although the curves didn't reveal a lot, we noticed that there was a negative trend. On overall they thought they were worse implementors on the second occasion. This really surprised us a lot, as one would expect this to ameliorate.

After some consideration, we came up with two hypotheses:

- either the complexity of our algorithms grew faster than the students could cope with,
- or they overestimated their capabilities at the first occasion.

The last one may be the most likely. At the time of the first survey they had rarely implemented something useful. The programs they had seen, were fully explained in the course. Due to Scheme's expressive nature, the programs can be easily understood and seem to be very easy. It is most probable that students could not fully appreciate the difficulties involved in programming. They just *thought* they would be able to implement, but programming in Scheme turned out to be more difficult than it seemed.

When we asked them during the exercise sessions why they couldn't implement a certain exercise, students mostly mentioned *I don't know how to start with it.* This was confirmed by the empty sheet of paper they were still facing. This *not being able to start* is more explicit when programming in Scheme than in Pascal. There, students can easily write down a frame of program heading, variable declarations etc. without actually having done any thinking on the solution. Such a *busy doing nothing* phase doesn't exist in Scheme and students face in a more violent way the fact that they actually can't solve the problem. They first have to completely understand the problem and develop a solution for it before they can write any Scheme code. And if they can't find an algorithm, they can't write the program.

As a matter of fact, we have calculated the individual correlations between the questions "**If I understand the algorithm, I can implement it.**" and "**If I understand the problem completely, I can develop the algorithm and the program.**". They are really significant: a whopping high of 0.82 for the group who likes the course, but also 0.77 for those without advance knowledge and 0.68 for the entire group. This indicates that the two problems are very strongly related.

Indeed, if we take a look at the group with prior knowledge of programming, this correlation drops to 0.40. But then we noticed that this is the group who disliked Scheme the most. Their problem may be dual. Firstly, they're probably not used to the functional way of thinking and develop their algorithms in a different way. Secondly, it is even likely that they generally don't think about the algorithm, but just start programming, which is a failure proof method in Scheme. Scheme doesn't seem to pose extra difficulties for the non experienced users, but it does so for the experienced group and that may be why their correlation drops.

Another but related view on this problem can be seen when students *think* they have an algorithm for the solution. When we notice that they cannot write the Scheme program, we ask them to explain what they're trying to program. It turns out that they are unable to formalise their algorithm enough to be able to write it down as a computer program. Computers need very concise programs, and they have problems getting to this level of conciseness. They would suffer from the same pain in any programming language, either Scheme or Pascal.

We have the feeling that they might get closer to something in Pascal. Mostly, part of their ideas is good, but not good enough for Scheme. They may be able to code the good part in Pascal, and sometimes they may even get an executable program by fiddling with it, but we believe that the program will be very error prone, because it is the translation of a badly understood algorithm, simply hacked together by assembling vague ideas. Scheme does not allow such a way of writing programs.

7 The Use of Computers

7.1 The Questions

These questions vary largely in scope, but all relate to moments in which students can/should encounter computers.

- Our use of computers
  - I want more/less demonstrations of working Scheme programs on computer during the lectures.
  - 3 out of 12 exercise sessions are in the computer lab. This is *1 = simply not enough 9 = way too much.*
- The programming tasks
  - I've used a Scheme interpreter at home *(before the first task was handed out).*
  - How do you feel about the programming tasks?
  - Would you still perform the task if you weren't obliged to?
  - How much did you learn from the first project?

7.2 Our Use of Computers

Due to circumstances, we haven't given any computer demonstrations during the lectures this year. But even in normal situations, the amount of demos remains pretty limited. We believe that it is not that interesting to see a Scheme program run: one learns little about the way Scheme works by just looking at a running computer. It's even pretty boring e.g. to see several lists being sorted.

It surprised us that the students don't agree. They clearly want to have demonstrations and the need even grows in time: the first enquiry gave 2.26 (recalculated figure), the second time 2.13 *(1 = a must, 6 = don't want demos).* So we asked a few students what they considered an interesting demo. They don't want a plain run, but would like to see how the different procedures are used during evaluation. So they'd rather have a program with breakpoints, and not just a trace, because that scrolls way too fast and doesn't show code being executed, just entry and exit points.

As already mentioned earlier on, we prefer not to program computers extensively. It is not a course on programming, but on methodology. Practical issues are also involved. During the exercise sessions, we usually rehearse the theoretical concepts to be used. We experienced that it is difficult to get the students' attention during the computer sessions. That is why we are reluctant to schedule more of them.

Another problem is that we want them to implement *several* programs. When implementing on PC, they rather make sure that their *first* program actually runs, before continuing with the rest. Having a program run is important, but because of our limited amount of exercise sessions, we do not necessarily want all the gory details. Instead of having them struggling with syntax details (mostly bracket related), we want the overall structure.

Despite the fact that we get to do more exercises during the paper sessions, the students definitely want more computer sessions. 81 % want it real hard (Fig. 5). They find that computer sessions are more efficient. Perhaps they can only finish one or two exercises, but then they really do understand them. And there's also the important psychological bonus of *"Hey, look what I can do!"*, which is lost on paper.

7.3 The Programming Tasks

But their whish for more computer exercises contrasts with our desire to concentrate on software engineering concepts. We don't want to stress the programming details in the exercise sessions, but rather the methodology.

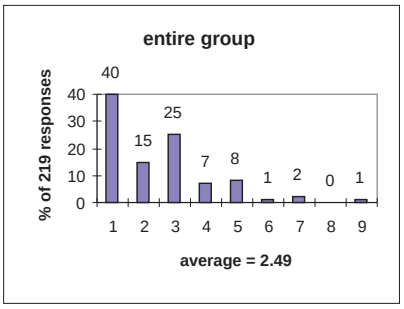

Fig. 5. 3 out of 12 exercise sessions are in the computer lab. This is *1 = simply not enough 9 = way too much.*

On the other hand, we agree that being able to really implement something working is an important issue, but we prefer to let them experiment on their own. It is a well known fact that programming can only be learned by extensive experimenting.

For this purpose, we include two mandatory projects to be made on computer. The results don't count for the examination, but it gives them the opportunity to learn to develop bigger applications. Not all students take care of this as seriously as they should, but those who do, generally get higher grades. It's not important that they score well in the project, just that they actually make the effort, and not simply copy the result from a colleague.

To help them finish the project, we give them the opportunity to have a Scheme interpreter at home. Before the task is handed out, some of them already use it voluntarily. If *1 = never* and *9 = all the time*, about one third answers from 6 to 9. The amount of advance knowledge doesn't influence this, and even in the group of people who love the course, a rather large number seldom experiments at home.

On the matter of projects: they really feel comfortable about them (Fig. 6). And an average score of 6.59 on another question indicates that they would still participate in the projects, even if they were no longer mandatory. This further stresses their interest in real programming.

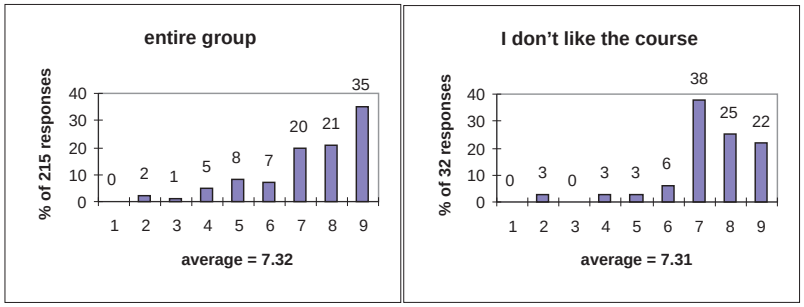

Fig. 6. How do you feel about the programming tasks? *(1 = negative, 9 = positive)*

7.4 Discussion

Although educators do not agree on whether it is better to do exercises on paper or using a computer [5], the students themselves clearly rebuked our approach to sparsely use computers.

And perhaps this is good. We may rightfully concentrate on the software engineering principles, but as it is the students' first computer science course, they should get some real hands-on experience. They can only understand and appreciate what is involved in (good) software development if they went through the (painful) process of programming a computer. This necessarily includes paying full attention to the details and precisely expressing the algorithm in the programming language. As long as we take care that they don't get overwhelmed by the details, they will certainly learn that the software engineering principles help in controlling the complexity of software development.

But please take note that students are very critical when it comes to programming environments. In the first year we used SCM [9] for DOS, which is just a recompilation of the Unix version and therefore has a very basic command line and needs a separate editor. The students were far from enthusiastic, but as we moved to EdScheme for Windows [11] (a very user friendly tool, that helps in parentheses matching, indentation, key words colouring, ...) their appreciation for Scheme more than doubled and they became a lot more eager to actually experiment.

This may sound like a very trivial fact, but it is important to stress that students tend to confuse between the language and the tool. If the tool is bad, they think the language is bad. This is confirmed by the fact that many questions and/or complaints about Scheme are actually about (the use of) EdScheme.

We agree with their concern for a good programming environment. It may have nothing to do with the language itself, but it is an important aspect that cannot be underestimated. After all, if the students can write their programs quickly and nicely, but only feed it to the computer through major efforts, what's the point?

So the morale of this section is that anyone considering to start teaching a functional language has to take care that it is complimented with a suitable programming environment, and that, especially in an introductory course, a substantial part of the exercises is scheduled on computers.

8 How Do Students Like the Course?

8.1 Questions and Answers

Whereas we looked into their love for Scheme earlier on, we also asked whether or not they like the course (Fig. 7). During the first survey only few people disliked the course, and a fair number liked it a lot, but the overall curve was Gaussian (again) with a slight advantage for the positive side. The second survey revealed that a number of people with a neutral opinion moved to the negative side. We see no clear reason for that. It may be influenced by the different lectors: as the normal lector was ill during the first nine weeks of the semester, another person replaced him. The students viewed his lessons as more interesting (4.04 instead of 4.85 (1 = swift, 6 = boring)). Especially those with advance knowledge preferred the part with the first lector.

The other question of the second survey that fits in this section is more specialised towards the contents of the course: **'The software engineering concepts in this course are fascinating** *(1 = agree, 6 = disagree)*'. It gave no really different result. The average (3.50) is now exactly in the middle. The peak just moved a bit in the positive sense. Those with advance knowledge like the concepts even more. They seem to understand that they can be used in any language.

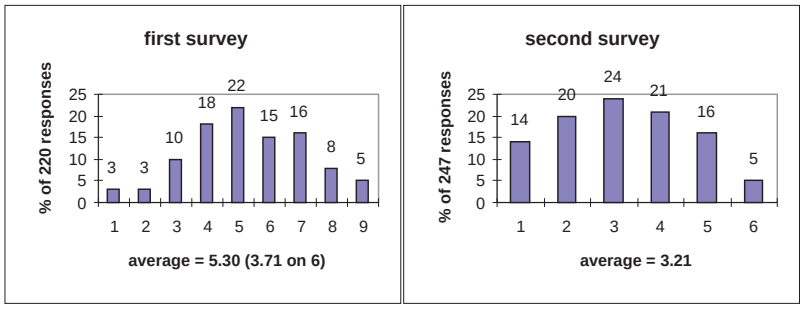

Fig. 7. I *(1 = hate, 9 = love)* this course

Up to the crucial question: **'Do you think the contents (as far as known) makes sense?** *(1 = not at all, 6 = 100%)*' (Fig. 8).

The results were identical on both occasions, but clearly point in the good direction. There's no Gauss curve here, but a steady ascend with a peak just below the top score. The average is clearly positive: 5.23 (max. 9) or 3.67 (max. 6).

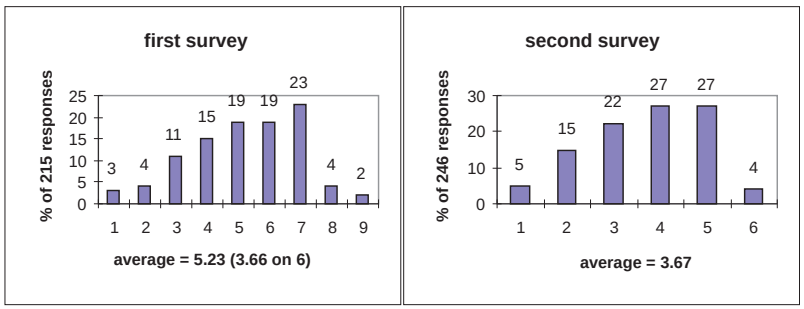

Fig. 8. Do you think the contents (as far as known) makes sense? *(1 = not at all, 6 or 9 = 100%)*

8.2 Discussion

The observation that students prefer some lecturers over others may not be surprising, but is important. Teachers still in doubt about moving to Scheme or another language, should keep in mind that simply the decision of moving from one language to another will not substantially influence the fascination for the course. Other efforts have to be made.

For example, when we used Scheme for the first time, we introduced lists very late (after higher order functions). At that time, we had lots of complaints of the kind '*In Scheme one can only work with numbers. Nothing interesting can be solved.*' As a result we now introduce lists a lot earlier, immediately after recursion. Since then we no longer heard such complaints, but we decided to explicitly ask the question during our first survey. The answers were evenly spread. A conclusion cannot be safely drawn, but at least the initial – wrong – impression is gone.

Related to our concern to keep the interest in our course high, we wondered if some graphical applications would help. The survey revealed that no high demand exists. Nevertheless, those who do not like

the course, would like some. As we should try to convince them too (otherwise, convincing the convinced is not hard), it may be appropriate to include some more graphics, for example through the use of the turtle graphics included in EdScheme.

If the direct question to the relevance of the contents is answered positively, there's another indication to prove this. We also asked how much they would remember after 1 and 5 years. There was no immediate interesting answer to this, but it struck us that those with advance knowledge think they will remember *a lot* more. We interpret this positively because we believe that they can see the difference between specific techniques and general principles. If there's anything to be remembered, it's the principles. So if they think there's a lot to be remembered, there must have been many principles learnt and this is what we wanted.

9 Preference for Specialising in Computer Science

A matter not really related to simply teaching a course, but still very relevant, is the following. Our students are very heterogeneous: they still have to choose an option: they can go into mining, electronics, computer science, ...

Some may already know what to choose and others may only be sure not to choose CS. Anyhow, for most of the students this course meant the first contact with real computer science, not just programming, playing games, word processing and web browsing.

Perhaps it learned them that computer science is more than just bit manipulations and indeed a very interesting art. Or perhaps to the contrary, only now they see that programming computers is a hard working profession and realise that it's not as fun as blasting enemies in the games gallery.

It may be vital for our department at the university, but more importantly for our economy, to raise the amount of students choosing for CS, because in Belgium we are facing a huge shortage of computer science graduates (at the moment approx. 7000), and I'm sure this equally applies to various other countries.

Our first questionnaire included two questions about the influence of this course on their preference for computer science.

- **Did your interest in the option computer science (1 = grow, 9 = shrink) ?**
  One third had no opinion, but for the rest we have a bit more interest losers than interest winners.
- **'If this course gives a correct taste of the computer science option, I will choose 1 = computer science, 9 = no computers.'**
  This scored 6.34. At that time we were unable to attract anyone without previous knowledge of computer languages. Luckily those who love the course, will mostly choose computer science.

These results may not look so good, but should be a bit relaxed. They were to be expected, as also in real life only a small minority eventually chooses CS. The survey just confirms this. Besides, it should not be the only purpose of a course to attract students. If anything, it should give them a right view on the problems involved and solutions proposed. We'd rather have students that know what is awaiting them, than those that are simply attracted by job opportunities.

10 Conclusion

As long as Scheme is not widely used in Belgium, both in educational and commercial projects, our use of Scheme as the first programming language will be controversial. Not because we allegedly don't do valuable things with it, but simply because the language won't be used later on. The students who are actually exposed to this strange thingie called Scheme, appreciate it in the end. We think that it is a powerful language in which a lot of software engineering concepts can be modelled and most of the students finally agree.

Despite the fact that a lot of students will continue to be confused by the brackets of Scheme and the problems related with dynamic typing, we will continue to use Scheme the following years, mostly because dynamic updating of objects is possible and the absence of types partly liberating. The presence of a user friendly environment is another important advantage.

The students who had advance knowledge of other computer languages resisted the most in the beginning, but even this group can be persuaded of the assets of our approach. The group of final year students who use Haskell is the most convinced of the power of functional programming, but then, they really got to taste it to its full extent. The problem with a first year course may be that it is just a first course: the examples and tasks are not big enough to fully appreciate the power of well thought out development.

Although not everyone is fully satisfied about the course and the language, we have certainly been able to convince most of our students. We therefore believe our approach and our preference for a functional language to be quite successful.

Acknowledgements

We would like to thank the students for filling in the questionnaires and giving very interesting extra remarks, Margot and the people of DUO for processing the questionnaires and Bern, Peter, Wim and Yvan for proof reading.

References

1. H. Abelson, G.J. Sussman: *Structure and Interpretation of Computer Programs* MIT Press, (1985).
2. M. Carlsson, Th. Hallgren, Th.: *Fudgets - A Graphical User Interface in a Lazy Functional Language.* Proceedings of FPCA, (1993) 321–330.
3. C. Clack, C. Myers: *The Dys-functional Student*, Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).
4. A. Davidson: *Teaching C after Miranda*, Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).
5. S. Fincher (co-ordinator), S. Thompson, P. Molyneux: *Teaching Functional Programming: Opportunities and Difficulties*, CSDN one-day workshop, Kingston University (10th September 1996), report at `http://snipe.ukc.ac.uk/CSDN/conference/96/Report.html`
6. P. H. Hartel, B. van Es, D. Tromp *Basic Proof Skills of Computer Science Students* Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).
7. P. Hudak, S. Peyton Jones, P. Wadler (editors): *Report on the Programming Language Haskell (version 1.3)*, Technical Report Yale University/Glasgow University (1995)
8. J.-P. Jacquot, J. Guyard: *Requirements for an Ideal First Language*, Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).
9. A. Jaffer: SCM 4e.1, Implementation of Scheme that conforms to R4RS and is available for many platforms.
10. E. T. Keravnou: *Introducing Computer Science Undergraduates to Principles of Programming through a Functional Language*, Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).
11. E. Martin (editor): *The WinScheme editor, the manual for Edscheme* Schemers Inc. (1994)
12. S. Thompson, S. Hill: *Functional Programming through the Curriculum*, Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995).