Otto-Friedrich-Universität Bamberg

Cognitive Science Group

Ausarbeitung

Im Rahmen des Bachelor-Seminars

Thema des Seminars

Zum Thema:

Learning to Program Recursive Functions

Vorgelegt von:

Hieber, Thomas

Betreuer: Emanuel Kitzelmann

Bamberg, SoSe 2008

Abstract

Learning how to programm is the foundation every student of computer science must be able to rely on for his entire career as IT expert. While facing control structures such as conditionals, loops as well as matters of scope like local and global variables the majority of students does not struggle too hard getting to know and using them. This suddenly changes when it comes to the concept of recursion. It is the most powerful concept in programming and thus a crucial technique to understand and use. In reality this concept has turned out to be quite difficult since students cannot grasp this paradigma as easily as the other ones. This paper is going to show the difficulties students have to deal with learning how to program, learning recursion and how machines can learn the same techniques by given examples. An empirical study carried out in 2007 shows exactly, that students find it easier to provide those examples instead of writing a recursive program to solve a certain problem. A detailed insight into the study as well as some threoretical background will be the concern of this paper.

Contents

1 Introduction **1**

2 Learning of recursive programming **1**
2.1 Recursion . . . 1
2.2 Students learning to program . . . 4
2.2.1 Language-independent conceptual bugs . . . 4
2.2.2 Language-dependent conceptual bugs . . . 6
2.3 Students learning recursion . . . 6
2.4 How do machines learn recursive programs? . . . 8
2.4.1 IGOR . . . 9

3 Empirical study **9**
3.1 Sample . . . 10
3.2 Assignments . . . 10
3.3 Procedure . . . 10

4 Results **11**
4.1 Interpreting the results of the study . . . 11
4.1.1 Flatten - exceptionally difficult? . . . 14
4.1.2 Experts Group . . . 14
4.2 Conclusion . . . 15

5 Appendix **17**

List of Figures

1 Fibonacci Numbers (http://mitpress.mit.edu/sicp/chapter1/node13.html) . . . 2
2 Correct Results Overall (without Flatten) . . . 12
3 Helper functions employed (tail-recursion) . . . 13
4 Results Length . . . 17
5 Results Reverse . . . 17
6 Results Oddlength . . . 18
7 Results Flatten . . . 18
8 Tail-Recursive Solutions - 1 . . . 19
9 Tail-Recursive Solutions - 2 . . . 19
10 Time - 1 . . . 20
11 Time - 2 . . . 20

1 Introduction

The ability to write computer programs is an engineering discipline without concrete items. When a machine is put together, workers use different tools for the various operations that need to be performed in order to assemble the bits and pieces. As computer programs are put together, it is only logical for the programmer to use tools of his own. Seldomly a brute force attempt can generate a descent solution for sometimes complex problems. When learning how to program, it is only natural to learn how to use more than just the syntactical features of a language, but far more importantly to learn how to use the constructs offered most effectively. Very early in the course of their studies, students of computer science learn about loops and conditional clauses and about one of their most important application - recursion. Here begin the difficulties for most of the students, since the concept of recursive problem solving is not as familiar to them from everyday life as simple iteration [8]. There are different approaches to the problem, some of them concentrate on the practical side like the 'Fundations of Computer Science Course' [3].

The core of this paper is concerned with the results of an empirical study on the ability of novice and intermediate programmers to formulate recursive programs to solve different tasks. At the same time, the students have to provide example input/output pairs for those imaginary programs. The result of this study is unambiguous - students are struggling with the concept of recursion, even though they are aware of the solution of the problem as the percentage of correct examples for three of four of the assignments shows. We will have a closer look at this phenomenon later on, but first we will have a look at the way programming and recursion is taught to students. In this process students undergo different phases of learning, which in term provide different sources for errors. After that we will switch perspectives taking a look at how machines can learn those same concepts and how they are fit to handle recursion much better than a human could ever do.

2 Learning of recursive programming

2.1 Recursion

> To iterate is human, to recurse, divine.
> (L. Peter Deutsch)

The word *recursion* is derived from latin *recurrere* which means 'run again' or 'run back'. Roberts [10] describes recursion as the procedure of

breaking down a bigger problem into smaller problems, which are structurally identical with the originating one but easier to solve. Recursion in the computational world is an all important concept - many algortihms use its elegance in order to solve complex tasks. Despite its usefulness there is a downside to recursion as well; Different kinds of problems demand different kinds of recursion.For instance some problems cannot completely be solved by a tree recursive algorithm since the tree recursion would quickly bloat the system's memory causing a program to abort. So there must be a possibility next to tree recursion to solve those problems with the help of recursion. For this purpose let us consider the problem of calculating the *fibonacci* numbers. The formal function of this reads as follows:

$$fib(n) = \begin{cases} 1 & if \quad n = 0 \quad or \quad n = 1 \\ fib(n-1) + fib(n-2) & else \end{cases}$$

Now the computation of this formula would build a tree (see Figure 1) where the leaves would finally end up being added together. Every single leaf results from the ultimate call of a recursive function.

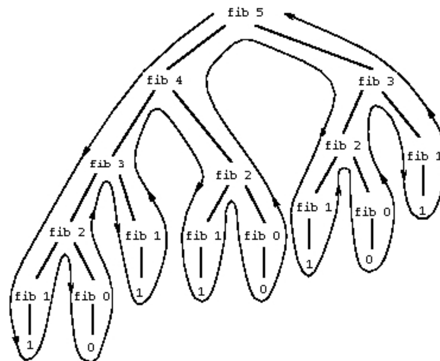


Figure 1: Fibonacci Numbers (`http://mitpress.mit.edu/sicp/chapter1/node13.html`)

A classical approach to implement this function in a functional language like *Scheme (http://www.gnu.org/software/mit-scheme/)* could look like this:

Listing 1: Recursive Fibonacci Numbers in Scheme from `http://en.wikipedia.org/wiki/Recursion_(computer_science)`

```
1
2 ;; n is an integer such that n >= 0
3 (define (fib n)
4   (cond ((= n 0) 0)
5         ((= n 1) 1)
6         (else
7           (+ (fib (- n 1))
8              (fib (- n 2))))))
```

Notice the conditional in the example: if the value of n is 0 or 1, the value is returned and the recursion locally aborted. This is the case in the leaves of Figure 1. This means that every recursive funtion needs a so called 'base case' to which a problem will gradually be reduced to. This may sound familiar to those who know about mathematical induction. The problems which can arise from the way we solved the calculation process in the example can be grasped by returning to Figure 1. For a growing n the amount of leaves to be held in the memory grows rapidly, especially since we employed a double-recursion. Imagine that in order to add all the values returned by the base-case they have to be stored until all the recursive calls have returned a value. Only then the addition of the values is started. This means that a large n will ultimately cause an abortion of the program because it has run out of memory. When we have a look at the results of the empirical study later on, we need to bear in mind that there are two ways to make a recursive call. Either directly in the same function like in our example, or indirectly by another function. A *function a* could call *function b* which itself makes a call on *function a*. This indirect way is used in a different style of recursion, which is more apt as a solution to the fibonacci problem - *tail recursion*.

Let us first have a look at another code example:

Listing 2: Tail Recursive Fibonacci Numbers in Scheme

```
1 (define (fib n)
2  (define (fib-iter n a b)
3    (if (zero? n)
4      a
5      (fib-iter (- n 1) b (+ a b))))
6  (fib-iter n 0 1))
```

We can see, that here two functions are used and the inner function is nothing more than an iteration which adds the values from iteration to iteration until the input of n is zero. So we employed an acumulator variable

and then pass on the result from the current iteration to the next in order to have the result processed *in-situ* instead of first creating the tree and finally doing the simple mathematical addition. This way we do not have to store the sub-results and the outcome is a massive save of memory. If you run these programs for different values of n you will quickly discover the benefit of the second one. We have no abandoned the elegancy of tree recursion for a memory effective tail recursion.

The central statement in this introductory section is that recursion is not all good. It can be very efficient when used properly and in the right context - but it can on the other hand be a cause of nuisance when either used in the wrong context (like in Listing 1) or if implemented with a false base-case which can result in a non-terminating program.

2.2 Students learning to program

When students learn how to programm they do not start with recursion right away. But even without it the road to becoming an experienced programmer consists of different phases in which the learner is prone to different kinds of errors as has been pointed out by Pea [7]. Since we will be interested in the mistakes our test persons may have made in their recursive programs during our study, we will focus on the different 'bugs', Pea has identified in his research on students learning to programm. So this will not be an exhaustive research on how people learn programming, but what classes of errors exist in the process and how they originate. Pea differentiates two categories of bugs: 'Language-independent conceptual bugs' and 'Language-dependent conceptual bugs'.

2.2.1 Language-independent conceptual bugs

To start of with the first category, note that 'While people are intelligent interpreters of conversations, programming languages are not'[7]. This means that humans are capable of discourse and thus being able to come to a mutual conclusion by interpreting a conversation. Since an interpreter or compiler is only bound to strict rules, there can be no such dialog between programming language and programmer. This is the source for three different problems, which are very basic ones and will not contribute a lot to our empirical study (we are not testing 8-year old subjects) but I am going to summarize them quickly. The first is called **Parallelism Bug**, a phenomenon which is common among programming novices. They assume a computer programm being processed parallel, meaning that an interpreter can process more than one line of code at the same time. Another thing students might be stumbling

over are **Intentionality Bugs**, which means that they assume the programm to work intentionally or with foresight, that a programm acts goal oriented like a human being, which a classical computer program will never do. The **Egocentrism Bug** lets a novice assume that a program 'knows' details that are only in the programmer's head, but he/she expects the programm to 'fill in' the for the novice obvious rest. All these three problems share the concept that there is a 'hidden mind in the machine that has intelligent, interpretive powers' [7] but we all know very well (and so did our test persons) that there isn't. One more interesting language-independent issue is that of meta-cognitive skills. Some prominent examples for these problems are the wrong number of iterations in a loop, a false condition to abort a loop loosing track of a variable's value when trying to comprehend a program. Other problems involve the focus on different parts of a program, which sometimes causes the programmer to forget the environment and effects of a local change to some other parts of the program. When the novices have to tackle more complex programs the **Goal/Plan Merging Bugs** begin to play an important role. Based on this is the assumption that a programmer is going to project a problem onto an imaginary program in his mind, thus creating goals and plans how to accomplish them. In order to do so, a goal might have to be split up in subgoals which have to be completed first. So before he begins to implement his solution of the problem, the programmer breaks it down in his mind into various goals and subgoals together with plans to achieve them. Sometimes he can already see the need for optimisation so he merges plans together before the implementation begins, otherwise the merging takes place later on after problems are discovered or some final optimisations are carried out. He then can set out to write a program according to the structure ha has created for himself. The is an intermediate state between the actual, real world problem and the way this problem is described to a programming language. The programmer knows how to translate the problem and so he first has to carry out a 'cognitive-walkthrough' of it. A problem resulting from the **Goal/Plan Merging** can be that a novice merges plans he thinks suitable to being merged overlooking that there are differences for example in the subgoals of these plans. This way two working plans are replaced by one which is not guaranteed to work as intended for every part of the program. It can also happen, that by merging plans some subgoals are omitted which are critical to the main goal or the structure of the program. Since these issues have a heavy impact on the final programs it is important to stress that they appear frequently because programmers tend to make their programs more efficient which often means combining and merging procedures. In his experiments, Pea found out that even novices already 'have a propensity to attempt merging, and moreover, they have a poor track record for being able

to carry it out‘.[7]

2.2.2 Language-dependent conceptual bugs

This category mainly consists of two kinds of problematic bugs - the **Knowledge Unavailability** and the **Knowledge Inaccessability**. The first one can easily be described as a general lack of knowledge in one or more components the problem is comprised of. The possibilities range from the knowledge of programming in general to the syntax of specific languages, from knowledge about a part of the problem to knowledge about the problem on the whole. The latter is not a question of lacking knowledge but of retreival. This phenomenon can affect anyone from novice to expert, but it sure is a lot more frequent when a novice tries to access knowledge he has learned recently. One reason for this may be that it has not yet been transfered to the long-term memory, or he may have mixed up some similar parts of the information stored in his memory. The reasons for problems concerning knowledge retreival are of less concern to this paper so the fact that it exists and has been reported [7] shoud be enough for the moment.
So far we have seen that there are various reasons for buggy programs written by novices, sometimes even experts. We will have to bear that in mind when we have a closer look at the results of our empirical study which had the test persons write recursive programs. Since the test persons were novices as well as experts with different levels of experience and practice, especially the problems described in the last subsection will play a role when it comes to interpreting the results.

2.3 Students learning recursion

I have already hinted at the difficulties students experience when first confronted with recursion. There are a few theories about teaching recursion, but in the reasons for the difficulty most authors of those papers agree. Pirolli has pointed out that ‘it seems that the human mind cannot suspend one process, perform a recursive calculation, and then return to the original suspended process‘ [9]. Levy & Lapidot [5] have listed the most general points not forgetting an interesting one for our study - the ‘inability to express a solution recursively‘. This is connected with the observation Pirolli has made and we will notice this very clearly in our study. The result is that the approach of how to teach recursion must be well structured and alert of these facts. While the overall structure varies among the propositions from Kruse [4], Pirolli [9], da Rosa [2] or Levy - they all agree on the necessity of examples. In this context, we get a very thorrough and detailed instruction

from Kruse, while Da Rosa and Levy focus on an empirical approach, the latter emphasizing the student centered approach. Pirolli is concerned with the examples themselves and how they can be used to learn from them. This is a more generic approach and we will return to this when we take a look at the way machines can learn recursive programs from examples. Levy directly contradicts Pirolli concerning the theory that the lack of everyday analogies of recursion does impede the students comprehending the concept. Levy lays the focus on the student, especially the terminology used in order to process the exmaples representing a recursive algorithm. His conclusion is that there must be lots of analogies, since he recorded a 'wide range use of analogies and metaphors' used by the students in order to grasp and explain the procedures. Even though he blames the abstract terminology used in the teaching or recursion - suggesting to combine it with a more informal language used by the students, he cannot escape the fact that the human seems to lack the cognitive capacity to quickly pick up the concept. This is supported by the dialog on *Wholism* and *Reductionism* in [10, p.12]. According to this, the novice is much more focused on the single parts of the problem instead of seeing the 'big picture'. It is also argued that it will always be difficult to abandon *Reductionism* completely, even though experts gradually learn to shift their bias towards *Wholism*. Da Rosa has researched the students' difficulties more closely and singled out different levels of abstraction. The initial example was:

1. *ab* is a word
2. If * is a word then a * a is a word
3. If * is a word then b * b is a word
4. Only the words obtained by application of the above rules a determined number of times are words of the language

The levels of abstraction are:

1. $ab \rightarrow w_n(level1)$
2. $w_{n-1} \rightarrow w_n(level2)$
3. $w_n \rightarrow ab(level3a)$
4. $w_n \rightarrow w_{n-1}(level3b)$

The observations show that level 1 and 2 are the most important ones, since they establish the relation between initial word and any word (level 1) and the relation between any word and the next word (level 2). Da Rosa notes that while students reach level 1 very easily, they struggle with level 2, often confusing the initial element *ab* with the previous element of a word. Once they have reached level 2 they found it easy to invert the concept by constructing an algorithm using the rules learned so far thus reaching level 3b. None of the students would detect the necessity of the base case (level 3a) until they had to apply their algorithm to a given example. Next to the obviuos difficulties with specific parts of the recursive concept (namely level 2), da Rosa states that 'the transformation of instrumental into perceptual knowledge [...] takes place in a slow and hard process which is (almost) never considered in traditional teaching'. His implication on the subject is evident, novice programmers sould be introduced to recursion carefully, giving them time to transform the concept into practical knowledge by exercise and repetition.

2.4 How do machines learn recursive programs?

Since the learning of recursion is evidently a chore to most students, we might ask ourselves if a machine is capable of helping out? We found out, that one major restriction of the human mind is the lack of 'working memory' to keep track of recursive steps. And memory is for sure something a computer has in abundance. The way a computer learns to program is outlined in detail in [1] and [6]. Pirolli has concentrated on recursion when designing an intelligent tutoring system for learning recursion [8]. His *GRAPES* system is a production system which uses goal/plan analysis in order to produce a set of rules. The rules to build a tail-recursive function in LISP are to be found in [8, p.329]. With the help of given examples the computer will try to apply the rules in order to write a recursive program. When we compare the way a program is created by a machine to the way a human programmer proceeds we find that the process itself does not differ much. Like the student, a computer is given examples and then designs a number of goals and subgoals in order to solve the problem programmatically. The difference is, that a computer does not employ analogy, nor is he able to start out on a single recursive formula, he is depending on a human expert to give him specific examples. We willcome to realise, that even at the very beginning of their education, studens are capable of providing such examples - much more as they are capable of writing the program themselves.

2.4.1 IGOR

IGOR is a system, developed by Ute Schmid and Emanuel Kitzelman of the Cognitive Science Group at the University of Bamberg [1]. By analysing a problem, consisting of a set of input/output examples combined with background information, the system can synthesize the specification generating a recursive program which recursively solves the given problem. This output will in time be exported into the syntax of a functional programming language. Given a formalised specification of a problem, *IGOR* will automate what is so problematic for a human's mind. To describe it in a most superflous way (which should be enough to clearify the contrast to a human) the system builds up a search tree bringing together the 'basic knowledge' provided (or not) with the restrictions or operations allowed. The specification is delivered to *IGOR* in *M*aude [2] and it is nothing more than an abstract data type with one difference. The functions are incomplete and only described by the input/output examples. *IGOR* takes it from there deducting the functions from the examples producing one ore more hypotheses which are possible solutions to the specified problem. So even as you will percieve this system potentially being quite superior when it comes to actually finding a solution by systematically browsing trough a search tree you will also note that it is not as independent as a human. One still has to provide a proper specification otherwise there would not be a satisfying result. This simple roundup about the way *IGOR* works should be enough for our purpose - you can find more information on the department's website and on `http://www.inductive-programming.org/resources/systems/#igor2`.

3 Empirical study

All the evidence we have collected so far leads to only one conclusion - without a change in the teaching strategy students will keep struggling with recursion and even have a hard time as more experienced programmers to dive into the paradigma. We have also seen that a machine could help developing recursive programs when given decent examples. Our study will thus comprise two different tasks - writing the programm and providing a number of examples.

[1] `http://www.uni-bamberg.de/kogsys/`
[2] `http://maude.cs.uiuc.edu/`

3.1 Sample

All of the 30 test persons were undergraduate students of computer science at the University of Bamberg, most of them in their first year. A second group of 5 test persons consisted of 2 post-graduate experts and three more advanced students of computer science and psychology. The group was only formed for time measurement and was not included in the overall result of the study. Nevertheless I am going to draw some comparisons now and then. From now on the former group will be referred to as the *students group*, the latter as the *experts group*. The students group consisted of 28 male and 2 female, the experts of 4 male and 1 female persons. The majority of the students group had heard an introductory lecture on computer science where recursion was introduced with the help of *Scheme*. So for most persons in the students group the memory of recursion and Scheme was still fresh. Albeit both post-graduates in the experts group had not heard this lecture they still had some experience with scheme which was around the same level as the average level in the students group.

3.2 Assignments

As I have already mentioned, the assignments were grouped into two categories. Overall there were four different tasks, which would either have to be solved by writing down a Scheme program, or by giving a restricted number of examples of optimal length/complexity. All the tasks were to be performed on lists. Order and type (example/program) of the tasks were permuted so that an equally distributed amount of programs and examples could be ensured.

1. length (returns a list's length)
2. reverse (returns a list in reverse order)
3. oddlength (returns true if a list contains an odd number of items)
4. flatten (returns a 'flat' list from a list containing nested lists as items)

For the programming, only a restricted set of commands, operations or literals was permitted:

3.3 Procedure

The assignment sheet started off with a cover sheet showing two illustrative examples of possible solutions for both categories of tasks. All the assignments had to be processed sequentially in a limited span of time. After the

define	if	cond
null?	list?	'()
cons	car	cdr
list	append	0
+	1+	#t

Table 1: Permitted Operations for the Programming Task

time had expired the participants were requested to skip to the next assignment. The experts group was the first group to be tested giving them the opportunity to write down the amount of time they used to solve the four problems. After them the students group had to tackle the assignments without recording the processing time.

4 Results

So far I have introduced the concept of recursion, how students and machines can learn it, what types of mistakes a programming novice/expert is likely to make. Furthermore I have outlined the concept, procedures and goal of the empirical study; now it is time to have a closer look at the results which will support the scenario pictured in chapter 2.3.

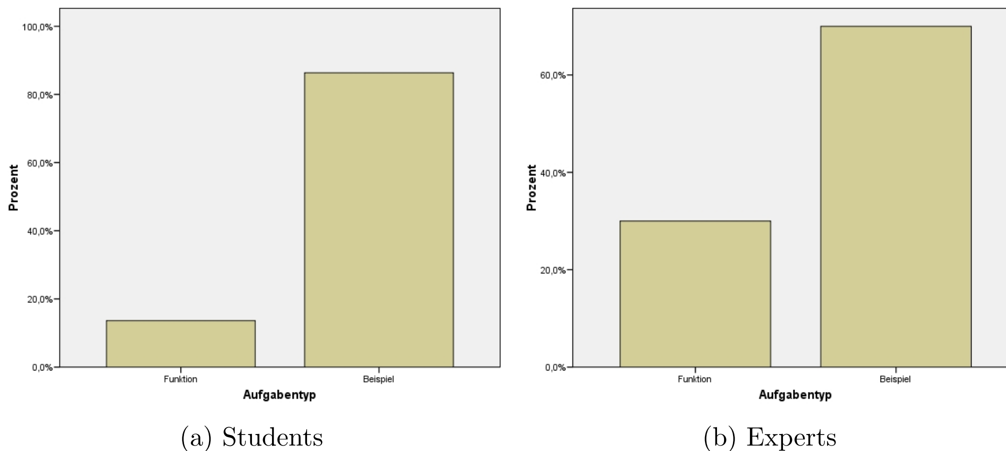
4.1 Interpreting the results of the study

The most important result is shown in Figure 2 - we cannot deny that sudents in Bamberg struggle as much as their collegues all over the world in learning recursion. The Pearson chi-square test proves this result to be highly significant:

$$chi^2 = 29.173; \quad df = 1; \quad p < 0.001$$

This statistic would look even worse, if **Flatten** was to be included. As it turned out that this was the most difficult task for all of the participants (including our expert group) we had to take the assignment out of the accumulation. I am going to get back to **Flatten at the end of this chapter** This result supports the theory that the students are not able to find a recursive program to match their understanding of the problem. As they are predominantely able to give correct examples it seems that it is not the problem itself which causes difficulties, but the task of defining a recursive solution for it. There is only one out of the students group who found a

valid recursive program for **Length**, one in both groups combined to solve **Oddlength**, while nobody was able to correctly solve **Reverse** (see Figures 4, 5 and 6).



(a) Students

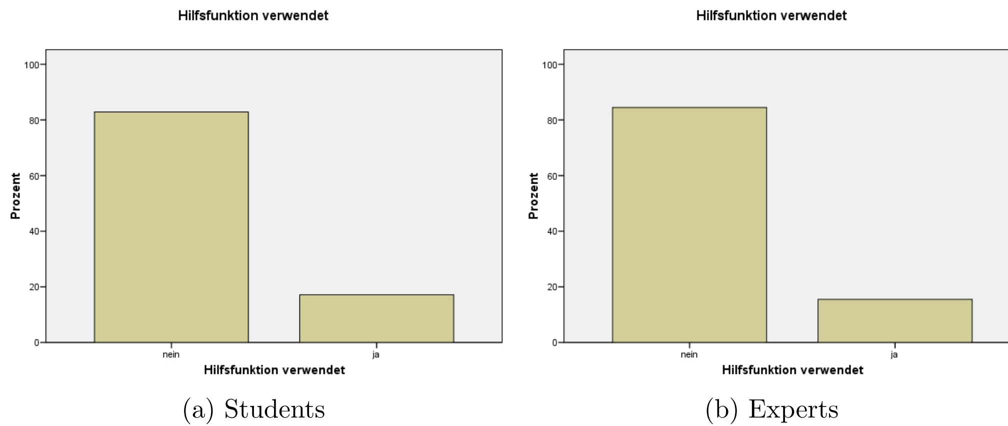
(b) Experts

Figure 2: Correct Results Overall (without Flatten)

When we take a look at the number of correct examples for the three problems (again, **Flatten** is to be neglected) we see the tables turning. With exception of **Reverse**, more than 80% of the test persons provided correct examples. A frequent source of error was students confusing the assignment type, writing a function when thy were supposed to give examples and vice versa. This means that a large majority of the test persons was able to conceptualise the problem, even more importantly, they were able to provide an expected behavior of a program that would solve the problem by giving input/output examples. The meaning for this is as we expected earlier - it is not the task itself which is complicated but the transfer from concept to recursive program. This becomes even more evident by the number of students who tried to solve the problem tail-recursively (see Figures 3, 8 and 9).

Reverse being an exception to this trend cannot directly be linked to the task itself. Since there is no distinct type of error discernible, it migth as well be an accident - the sample size is just not big enough. Still we find the orientation towards a majority of correct examples, even though not as distinct as with the other assignments.

The range of errors among the programming assignments is naturally wider. As we were not after a perfect runnable code example, the syntax of the pro-



(a) Students (b) Experts

Figure 3: Helper functions employed (tail-recursion)

gram was not a criteria of correctness, it was merely recorded. The important factors contributing to a correct solution were:

- correct base-case
- recursive call employed
- correct semantics

The evaluation started with base-case and recursion, making them a prerequisite for correct semantics. Sometimes there was an example of an 'almost' correct solution, which was semantically good, but still not correct on the whole. In our ratings, such an assignment is therefore not considered correct, since there are still some vital parts missing to make an acceptable solution in Listing 3. Note that the trouble lies in line 5, where the call of 'cdr' is redundand. Apart from this the program would work fine, even though it is tail recursive and thus an example for another phenomenon I have already pointed out.

Listing 3: ”Almost” Correct Tail-Recursive Example

```
1
2 (define (length liste) (help liste 0))
3
4 (define (help liste zaehler)
5         (if (null? (cdr liste)) zaehler
6             (help (cdr liste (+ zaehler 1)))
7         )
8 )
```

But there is also a number blank or incomplete solutions which could either mean that the student ran out of time or was not able to conceptualise the problem. But you will have already noticed, that ‘incorrect‘ does not mean the student is not able to formulate recursive programs at all, the reasons are more subtle. In some sheets we found drawings of tree structures, which means that students had to visualise what was difficult for them to grasp, otherwise. This supports Pirollis's claim of the cognitive restriction in connection with recursion (see chapter 2.3).

4.1.1 Flatten - exceptionally difficult?

Initially, the studie's assignment sheet consisted of four tasks. I have already analysed the data seperating **Flatten** from the rest of the assignments. This is because of the poor results including both programming and examples. There were only two students providing examples which were decent enough. It is obvious that students struggled not only with the recursion here, but without the assignment itself, which is of course more difficult than the other ones. The difficulty is that the nesting becomes very complicated very quickly and together with the observations made until now it is no surprise that almost everyone failed (1 correct set of examples by the students group, one by the experts group) no matter what level of expertise they come from. Furthermore they seem to be inable to come up with a structure for the recursive data type ‘nested list‘ when asked to produce input/output examples. The variety of attempts shows the amount of uncertainty which can only be interpreted as a general inability to solve this task in a structurised manner. The number of blank solutions for flatten is another convincing argument for the seperate evaluation of the assignment.

4.1.2 Experts Group

One last thought on the experts group is the average time they took for each assignment. Even though this group was first and foremost an attempt to

find out about the time students would later on need for the various tasks it is still intriguing to have a quick look at their performance. As the sample size is clearly not big enough, I will try not to speak in generalities, but the expectation is that the average time for providing the examples should be smaller than the average time used to write a function. But, as you can see on the example in Figure 10a, one unproportinal value can blur the direction indicated in figure 10b and figure 11a. We might also conclude that the balance between the values of flatten (figure 11b) is an expression for the difficulty of the task, but at the same time we couldn't because of the lack of examples. This does not mean that the analysis of the experts group's performance becomes void, it can be used as a contrast to the novice results, which differ from them as expected (figure 2).

4.2 Conclusion

We have come to see in section 2 that there are explanations for students having difficulties in learning recursion. At the same time it should have become evident, that machines can infer recursive programs from given examples for simple tasks such as those from the studies' assignemnts more reliably. The problems students seem to have are wide spread starting from difficulties with the task itself to a general cognitive restriction to tackle recursive problems effectively. The results of our study support Levy ([5]) who stronlgy suggests a more student-centered approach instead of the classical, more problem centered approach. Also, the findings of da Rosa ([2]) are underlined, as she clearly stated that the transformation from problem solution to a valid formalisation is 'a slow and hard process'. Like Levy, she suggests a change of the teaching approaches - and she may well be right. As there seems to be a cognitive barrier which is very hard to lower, students should be supported in this process with the help of teaching methods which are aware of this fact. So, in order to take the results of our study, which only confirm what we may have guessed before, one step further, we should involve the teaching. As Levy has already introduced his *classification and discussing learning activity (CDA)* it seems like a good starting point to take his theories to the next level. A direct comparison between students learning recursion the 'common' way and a methodology which demands a strong theoretical and practical activity of the student. If Levy and da Rosa are on the right track, the results of such a study would be most intriguing.

References

[1] Allen Cypher. Programming repetitive tasks by demonstration. In *Watch What I Do: Programming by Demonstration*, pages 205–217. The MIT Press, 1993.

[2] Sylvia da Rosa. The learning of recursive algorithms from a psychogenetic perspective. Technical report, Universidad de la República, Monterrey, 2007.

[3] Peter B. Henderson and Francisco J. Romero. Teaching recursion as a problem-solving tool using standard ml. *SIGCSE Bull.*, 21(1):27–31, 1989.

[4] R.L. Kruse. On teaching recursion. *ACM SIGCCE-Bulletin*, 14:92–96, 1982.

[5] Dalit Levy and Tami Lapidot. Recursively speaking: analyzing students' discourse of recursive phenomena. *SIGCSE Bull.*, 32(1):315–319, 2000.

[6] H. Lieberman. Tinker: A programming by demonstration system for beginning programmers. In *Watch What I Do: Programming by Demonstration*. MIT Press, 1993. see `http://www.acypher.com/wwid/Chapters/02Tinker.html`.

[7] R.D. Pea. The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, 9:5–30, 1987.

[8] P.L. Pirolli. A cognitive model and computer tutor for programming recursion. *Human-Computer-Interaction*, 2:319–355, 1986.

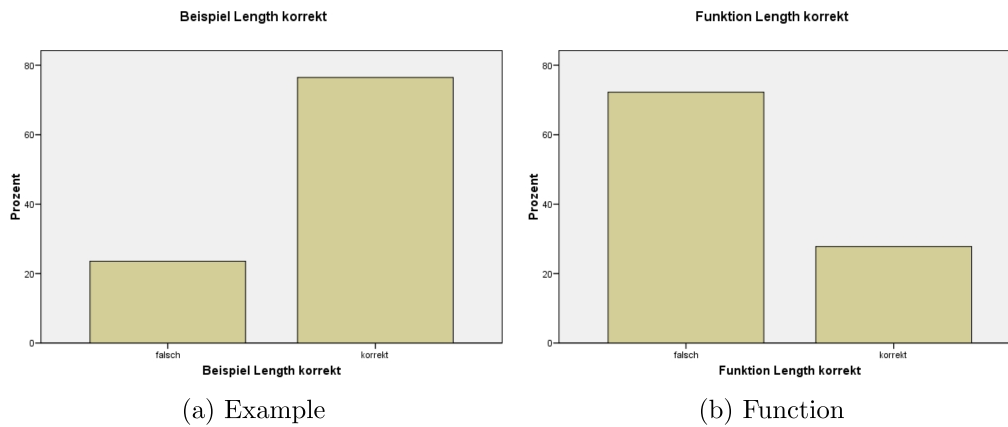
[9] P.L. Pirolli and J.R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 39:240–272, 1985.

[10] E.S. Roberts. *Thinking recursively.* Wiley, 1986.

[11] G. Weber. Episodic learner modeling. *Cognitive Science*, 20:195–236, 1996.

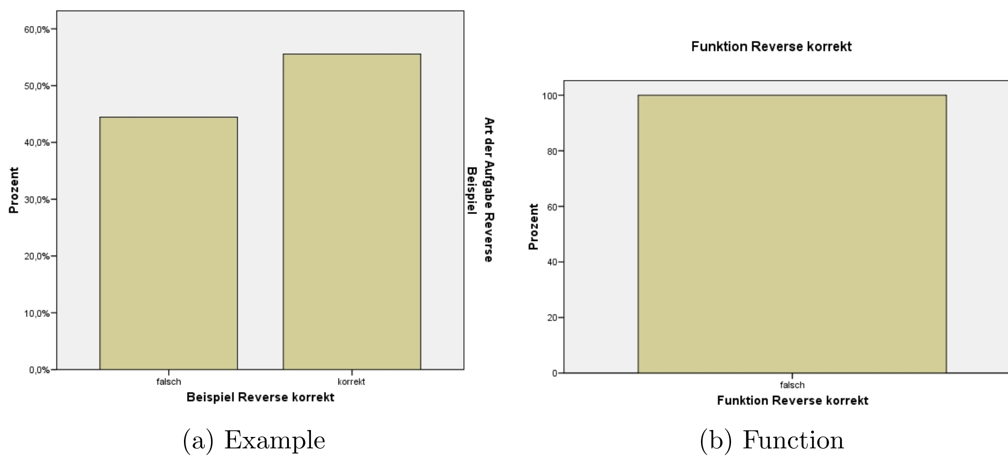
[12] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP.* Addison-Wesley Publishing Company, 1984.

5 Appendix



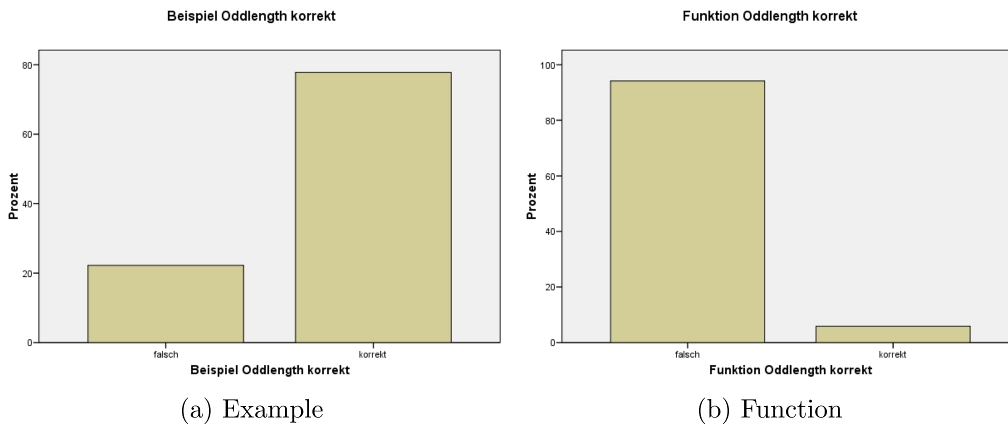
(a) Example (b) Function

Figure 4: Results Length



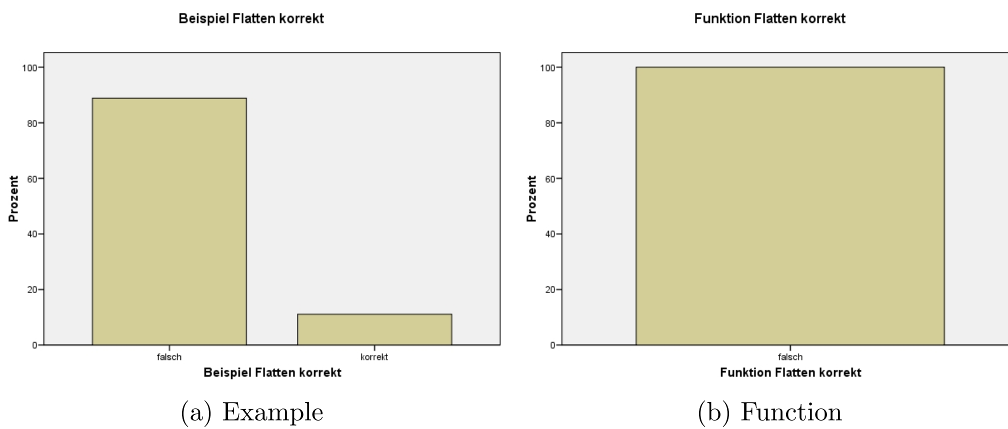
(a) Example (b) Function

Figure 5: Results Reverse



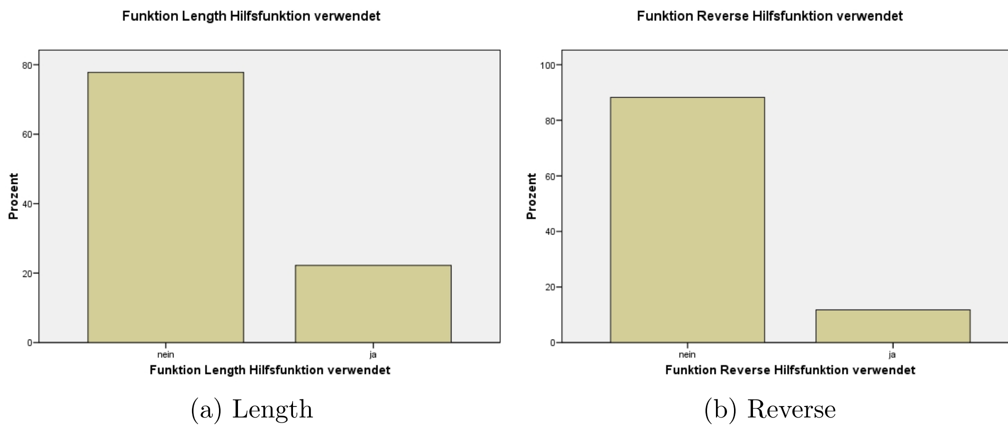
(a) Example (b) Function

Figure 6: Results Oddlength



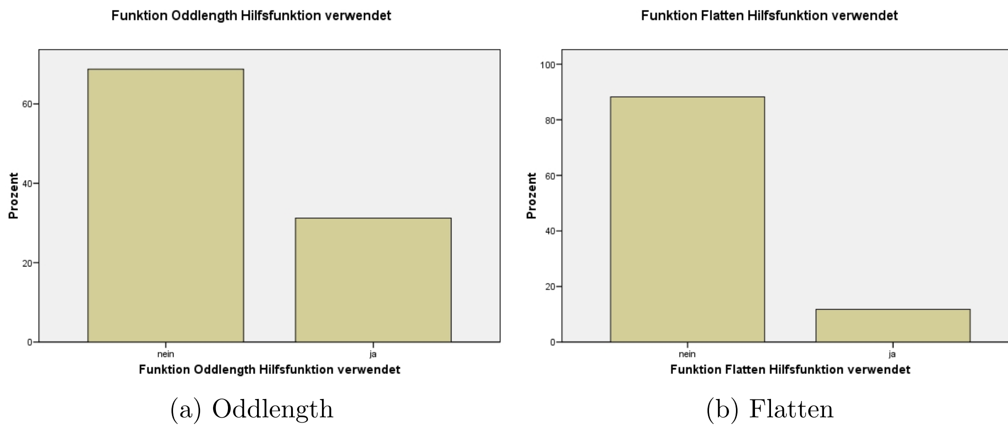
(a) Example (b) Function

Figure 7: Results Flatten



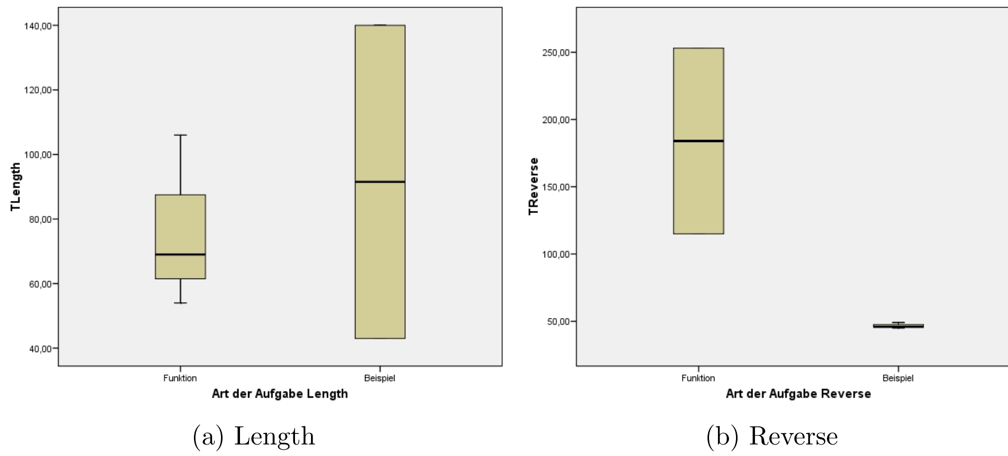
(a) Length (b) Reverse

Figure 8: Tail-Recursive Solutions - 1



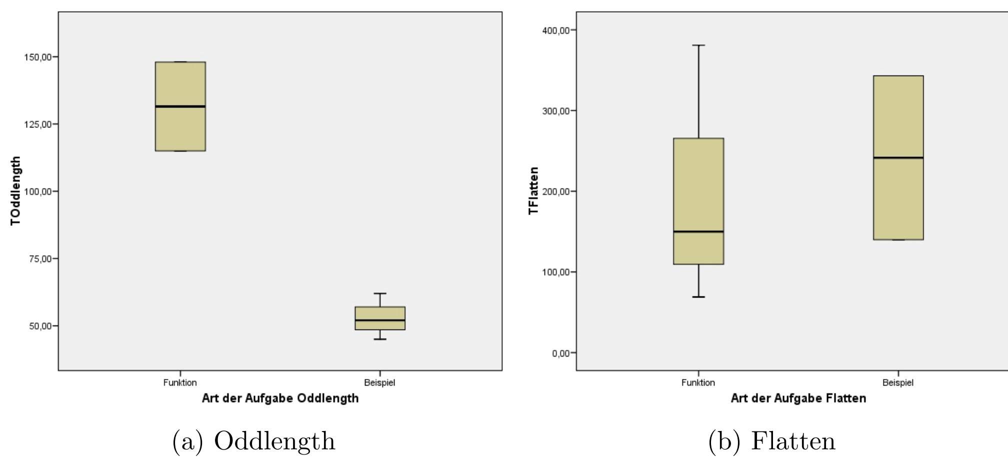
(a) Oddlength (b) Flatten

Figure 9: Tail-Recursive Solutions - 2



(a) Length (b) Reverse

Figure 10: Time - 1



(a) Oddlength (b) Flatten

Figure 11: Time - 2