An Intelligent E-Learning System for Beginner Programming - Using Analogical Reminder for Error Classification and Explanation

Master Thesis

in Education of Business and Information Systems at the Faculty of Information Systems and Applied Computer Science of the Otto-Friedrich-University Bamberg

Author: Robert Pollack
Supervisor: Prof. Ute Schmid
Abstract

The work explains the design and implementation of STAEREX the Scheme Tutor for Analogical ERror EXplanation. The tutor system is a prototype of an Intelligent Tutoring System that assists a learner during solving programming exercises in the functional programming language SCHEME by displaying an example that has been solved correctly in the past. The work describes important aspects of learning by analogy and learning to program. It analyses existing ITS for programming functional languages. From this considerations the technique of Constraint Based Modelling (CBM) is identified to create the student and the expert model of the prototype. Furthermore the work explains how constraints are use to retrieve an example that is similar to the erroneous code written by the student.
## Contents

1 Introduction .................................................. 4

2 Learning to Program by Analogies .................................. 7
   2.1 Learning by Analogies ........................................ 7
      2.1.1 Characteristics of Analogical Reasoning .............. 7
      2.1.2 Usage of Examples in Learning Programming .......... 9
   2.2 Learning to Program ........................................ 10
      2.2.1 Recursion and Functional Programming .............. 10
      2.2.2 General Aspects of Learning to Program .......... 12
      2.2.3 Learning Functional Programming Languages .......... 13
      2.2.4 Programming Recursion ................................ 14
   2.3 Summary .................................................... 17

3 E-Learning and Intelligent Tutoring Systems .................. 18
   3.1 Components of an ITS ...................................... 18
      3.1.1 The Expert Module ................................... 19
      3.1.2 The Student Module ................................... 20
      3.1.3 The Pedagogical Module ............................... 21
      3.1.4 The Communication Module ............................ 22
   3.2 Examples of Programming Tutors ............................ 23
      3.2.1 The LISP Tutor ....................................... 24
      3.2.2 The ELM Programming Environment .................. 25
      3.2.3 Summary .............................................. 27
   3.3 Constraint Based Tutoring Systems ......................... 27
      3.3.1 Learning from Performance Errors ................... 28
      3.3.2 Constraint Based Modelling .......................... 29
      3.3.3 Example of a CBM Tutor .............................. 31
   3.4 Summary .................................................... 32
4 Conception of the STAEREX system
   4.1 User Interface ........................................... 33
   4.2 Expert Module ........................................... 34
      4.2.1 Syntax Constraints ................................. 35
      4.2.2 Problem Constraints ................................. 36
   4.3 Student Module ........................................... 39
   4.4 Pedagogical module ...................................... 41
      4.4.1 Steps for the conception of e-learning environments ... 41
      4.4.2 Analysis of the Target Audience ..................... 42
      4.4.3 Learning Content and Learning goals .................. 43
      4.4.4 Sequencing of the Course ............................ 44

5 Implementation Aspects .................................. 47
   5.1 Implementation of the Tutor component .................. 47
      5.1.1 The Tutor and the Interpreter Class ................. 47
      5.1.2 Parsing a SchemePair ............................... 50
   5.2 The Client-Server Interface ............................. 53
   5.3 The STAEREX Data Model ................................ 55

6 Summary .................................................. 59

Appendix .................................................. 70
## Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBM</td>
<td>Constraint Based Modelling</td>
</tr>
<tr>
<td>ELM</td>
<td>Episodic Learner Model</td>
</tr>
<tr>
<td>ELM-PE</td>
<td>Episodic Learner Model - Programming Environment</td>
</tr>
<tr>
<td>ITS</td>
<td>Intelligent Tutoring System</td>
</tr>
<tr>
<td>STAEREX</td>
<td>Scheme Tutor for Analogical Error Explanation</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the middle of the last century learning with the help of machines is a topic researched not only in education science but also in computer scientist. Especially in the field of Artificial intelligence the attempt was made to substitute or complement the human tutor by an Intelligent Tutoring System (ITS). Even if the research in the field of e-learning has moved away from this topic to web based learning and interactive online learning environments it is still reasonable to deal with ITS. Today’s learning system provide content in many forms like instructional materials, exercises, forums or chats. Components of a tutoring system can be integrated in such an learning environment. Especially in e-learning settings for learning programming it seems straight forward to provide a software component that assists the student when working on exercises.

An important source of information for solving new programming problems is the usage of previous solved examples. Learners apply analogical reasoning to solve new problems by transferring solution steps from the previous problem to the current problem. Retrieving an example from previous problem solving sessions is the central strategy for assistance of the ITS that is explained in this work. Thus a further motivation to deal with ITS the possibility to get insights into the process of learning and to test and improve theories of cognition and learning [Nwa90, p. 153]. In case of this work the developed system should be a starting point for investigating students behaviour while learning basic programming from examples.

From this consideration the central research question of this work arises:

• How can an ITS for learning a functional programming language be designed and implemented that assists students by displaying an example from a already solved exercise?

To answer this question a prototype for such a system was designed and
implemented. From this main question some other questions arise that have to be regarded before the system can be designed:

- What techniques have been developed so far design and implement tutoring systems?
- What are the important steps in solving programming problems with the help of example?
- How does the retrieving of examples take place?
- How must examples be represented in an ITS to retrieve them for assisting the student?

Because the design and the implementation is an complex and time consuming task their are aspects that are not part of this work. Out of the scope is a detailed evaluation of the system that controls the systems behaviour and the effect that working with the tutor has on the learning outcomes. Because the focus lies on the components that store and retrieve examples other aspects of software design such as an appealing user interface and performance issues are not regarded.

To cover the aspects that are important for the design of the system from the point of view of learning theory and analogical reasoning chapter two deals with these topics. Because a discussion about how people learn is connected with the learning content Chapter 2.1 gives an introduction to basic concepts of the functional programming language Scheme. After that the process of acquiring programming knowledge is investigated with special focus on common misconceptions and errors. The considerations include a section about the concept of recursion because this is a central aspect of functional programming languages. The process of analogical reasoning is the second important part of the chapter. Aim of the chapter is to identify important steps within analogical reasoning to be able to model these steps in the tutoring system.

Chapter 3 deals with the structure and design of ITS. The aim of this chapter is to analyse the general structure of an ITS. The central aspect is to explore how the students and the expert knowledge can be modelled and how this can be done in the (TAEREX system. Therefore some examples of ITS that have proven to work in practice are considered in that chapter. After that important aspects are summarized the will be part of the design of the STAREAX system.

The chapters 4 and 5 deal with the design of the system. The design of the components that were identified in Chapter 3 is discussed. Chapter
5 explains important implementation aspect. Because it is not possible to discuss all aspects, only three important components are regarded: the tutor component, the client-server interface and the data base model.

The last chapter summarizes the conceptions of the prototype and gives an overview on extensions that can be made to transform the prototype to a running system as well as further research questions that are connected with the system.
Chapter 2

Learning to Program by Analogies

This chapter deals with learning by analogies and learning to program and the connection between this two areas. The aim is to get a overview on important issues of learning as a starting point for the conception of an ITS.

2.1 Learning by Analogies

2.1.1 Characteristics of Analogical Reasoning

Learning by analogies is characterised by the usage of a problem solving procedure that has been acquired during solving a previous problem to solve a new unknown problem [And07, p. 299]. The central point of analogical reasoning is the mapping of a base problem to a target problem. The mapping has to be done by using some features of both problems. On the basis of the quality of these features two types of analogies can be distinguished: surface analogies and structural analogies. If we regard the base domain and the target domain as a system of elements that are connected by relations, a surface analogy maps similar characteristics of the elements in the base domain to the target domain but not the relations between the elements. An example of a surface analogy would be ”The surface of the glass table shimmers like water”. The example shows that a surface analogy focus on the features of elements in base and target, here the shimmering surface of the table and water. Structural analogies are characterized by a deep similarity between base and target. This similarity arises from an abstract principle that is shared by base and target [Wen06, p. 407]. In systems thinking that would be represented by the relations of the elements that
<table>
<thead>
<tr>
<th><strong>Base: Solar system</strong></th>
<th><strong>Target: Atom</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The sun attracts planets</td>
<td>The nucleus attracts electrons</td>
</tr>
<tr>
<td>The sun is bigger than the planets</td>
<td>The nucleus is bigger than the electrons</td>
</tr>
<tr>
<td>Planets revolve around the sun</td>
<td>Electrons revolve around the nucleus</td>
</tr>
<tr>
<td>Planets revolve around the sun because the sun is more massive than the planets</td>
<td>Electrons revolve around the nucleus because the nucleus is more massive than the electrons</td>
</tr>
<tr>
<td>The sun is hotter than the planets</td>
<td>No transfer</td>
</tr>
</tbody>
</table>

Table 2.1: Analogy between the solar system and the structure of atoms

form a pattern which is similar in base and target. A famous example would be the Rutherford’s analogy that ”an atom is like the solar system” [Gen83, p. 160]. This analogy is summarized in Table 2.1.

The core of this analogy lies in the relationship of its elements. The planets circulate around the sun like the electrons around the nucleus. The sun is more massive than the planets like the nucleus is more massive than the electrons ([Gen83, p. 160]. As the example shows a analogy between different domains tends to have a low surface similarity where as an within domain analogy has a high surface similarity [Wen06, p.407]. In the context of learning a surface analogy is much more likely to be recognized by students then a structural analogy. After students got a hint that two problems are similar in structure and that they should examine them more closely they are able to use the structural analogy between the two domains. The ability to recognize structural analogies is increasing on the way from novice to expert [Wen06, p.409].

It has already been stated that analogical reasoning includes the mapping between base and target. Furthermore the process of applying an analogy can be broken down into several steps. The steps presented here are a synthesis of how different authors (e.g. [NH91, pp. 398], [May08, p. 329], [Wen06, p.408]) break down the process. They often just use different notations for the single steps. In this work analogical problem solving is seen as a five-step process:

1. **Encoding:** Building a representation of the target area by recognizing characteristics of the new problem.

2. **Retrieval:** Usage of the identified characteristics to retrieve previous solved problems from long term memory.
3. **Mapping:** Recognize problem solving procedures from the base problem that can be applied to the target problem.

4. **Alignment:** The retrieved procedures have to be adopted to the new problem.

5. **Learning:** New problem solving skills are acquired by learning an abstract schemata that can be used for a certain class of problems.

The steps imply that the learner has to be able to discover similarity on its own [Wen06, p.408]. In retrieving a base problems he has to recognize a surface similarity based on the characteristics found during the encoding of the new problem. In the step of mapping the problem a structural similarity has to be identified to extract the needed problem solving procedure from the base problem [And07, p. 307]. When adopting this procedure to the new problem an evaluation on the quality of the analogy takes place. As a result it becomes clear whether the solving procedure can be used on the new problem or not [Wen06, p.408]. The important step is the last one. Only if the student is able to induce an abstract schema from the solved examples and the similarity between them the analogy will be useful for him. Otherwise the analogy will be forgotten after problem solving because of its idiosyncratic and ephemeral nature [WHGP01, p.126].

### 2.1.2 Usage of Examples in Learning Programming

Learning a (functional) programming language is a complex task (see Chapter 2.2). Therefore learning from analogies and examples seems to play an important role in learning to program [PA85, p. 285]. Even if analogies can be very useful there arise some problems. When students use examples in programming they use them as kind of template and modify them. This is a surface strategy. The students compare the example with the new problem and try to fix the difference [KA86, p. 156]. The search for structural relations in the examples does not take place. There is also evidence that in further stages of the learning process analogy plays an less important role. Novices tend to use analogies more often then more experienced learners do [AFS84, p. 122]. This reveals a limitation of analogies to early learning stages.

If examples are used for learning functional programming using recursion it is not enough to simply provide examples to students. Examples only show how a recursion works. They do not show how a recursion can be written to solve a particular problem. To solve this problem an structural model
of recursion in combination with an example can be provide to increase learning outcomes [PA85, p. 285].

To enhance the learning process examples used in learning text should meet some criteria [May08, pp. 329]:

- They should consist of different components such as a problem description, a solution and an explanation of the solution.

- A descriptive example should be combined with a transfer exercise to be useful.

- An well elaborated example works better then a short explanation that lists each solution step.

This points should be taken into account when designing learning text for the tutor system. Also the issue of combining a structured model with a concrete example should be regarded.

### 2.2 Learning to Program

#### 2.2.1 Recursion and Functional Programming

Before the process of learning a programming language is explained a short introduction on functional programming and recursion is given. The concept of recursion is essential for functional programming thus it will be considered first. After that it will be explained in the context of functional programming.

A recursive function is a function that is defined in terms of itself. To illustrate how a recursive function works the example of the factorial function \( n! \) is used. The factorial of number is the product of the predecessors of the number greater null. So the factorial of \( 4! \) is \( 1 \times 2 \times 3 \times 4 \) which equals 24. The recursive definition of the factorial function is:

\[
n! = \begin{cases} 
  1, & \text{for } n = 0 \\
  n \cdot (n-1)!, & \text{for } n > 0 
\end{cases}
\]

The factorial function illustrates two important parts of a recursive function. First there is the recursive case where the recursive call takes place and second the base case where the recursive call ends. The recursive function is solved by processing until the termination case is reached. At this point the recursive call ends but the function has to be evaluated by multiplying the results of each call with each other.

The simplest form of recursion is the tail recursion. When a recursive call is made in a tail recursion, this call is always the last part of the function.
Their is no other evaluation done after the recursive call. That means that after the termination of the recursive calls no other evaluation is applied [GO97, p. 139]. In node tail recursive functions an evaluation of the function has to take place after the termination of recursive calls.

Another classification of recursion can be made, too. Recursive functions can be seen as linear if each recursive step is taken by a simple function call. The opposite would be the tree recursion where a complex problem is split up into smaller problems by having more then one recursive call. An example would be the calculation of the Fibonacci sequence:

\[
Fib(n) = \begin{cases} 
0, & \text{for } n = 0 \\
Fib(n - 1) + Fib(n - 2), & \text{for } n > 0 
\end{cases}
\]

The recursive case consists of two recursive calls. When implementing a recursive function a tree recursions could be transformed into a linear recursion by using a second parameter which stores a part of the result and is passed by each recursive call [GO97, p. 141].

To briefly explain the features of a functional language the factorial example is defined as a Scheme function:

\[
(define (fact n) 
  (cond (= n 0) 1
         (> n 0) n * fact (- n 1)
       )
)
\]

The first feature visible is the prefix notation. A function or procedure is generally regarded as a list. The first element of that list is the operator. The following elements are the parameters of the function. The parameters are the input parameters which are transformed by the function to output parameters similar to a mathematical function [Pie10, pp. 6]. The result of the evaluation of the function is the output which is returned to the user. A call to the factorial function and evaluation by a SCHEME interpreter would look as follows:

\[
(fact 4) \Rightarrow 24
\]

Another important feature of functional languages is that they do only consist of a set of basic data types and some primitive function which are used to write more complex functions. Therefore functional languages can be used for teaching programming principles because the syntax students have to learn is small [Bac78, p. 624]. Thus more attention can be laid on higher programming concepts such as recursion.
2.2.2 General Aspects of Learning to Program

The task of programming has three features that are cognitive relevant: language features, design skills and general problem solving skills. Language features can be described as the smallest, non-decomposable elements of a programming language. They are typically introduced by teachers, describing how they work. After that students have to use them in exercises [LD89, p. 59]. The problems occurring in this step are that the student has to understand the syntax of the language feature and that he must be able to put this features together which is a great problem for novice programmers [SS86, p. 624].

On the next level students have to apply design skills that enable them to build a working program from language features. Two skills are developed on that level (cf.[LD89, p. 59]):

- **Usage of Templates**: That represent planes or schemata that describe how to put the language features together to construct complex functions.

- **Procedural skills**: That involve how to split a problem into sub plans and planning how to compose the sub plans to a program, testing the program and reformulating the program if necessary.

Novices and experts differ in their application of procedural skills. Where novices omit the task of planning, experts spend most of their time on planning while working on a programming problem. In the phase of testing experts are better in involving boundary conditions in their tests. For novices testing seems less important than solving the problem [Ebr41, p. 478]. When reformulating a problem novices tend to regard only small parts of their thus applying local fixes without regarding how their programs work as a whole [PSBK89, p. 312].

The last step in acquiring cognitive a skill is gaining problem solving skills. This is characterized as the ability to transfer templates and procedural skills to another, still to learn programming language. This transfer is not limited to programming languages but can also be applied to other formal systems [LD89, p. 61].

Five main areas of difficulties in learning programming can be identified [DB89, pp. 283]. Some of them have already been touched in the description of programming as a cognitive skill:

1. **Orientation**: Students have to get an idea what programming is used for and why it is important for them to study programming.
2. *Notional machine:* Students have to understand how the computer works by creating a model of the computer’s processes.

3. *Notation:* As mentioned before, students have to master the programming language on semantic and syntactic level.

4. *Structures:* They have to acquire templates in the form of schemes and plans.

5. *Pragmatics:* The have to acquire procedural programming skills to be able to specify, develop, test and debug their program.

Mistakes can arise within all of these areas. One important note regarding analogies in programming has to be made. Mistakes can arise from analogy reasoning if students try to transfer the meaning of natural language expressions to the programming language assuming that the computer will know about this expressions as a human would [PSBK89, p.304]. Other sources of error can be a misapplication of an analogy by extracting more structural features from an analogy then needed. An example is the misconception that a variable can store more then one value because of the analogy that a variable is like a box [DB89, pp. 284].

A observation from the studies on novice programmers is that not all students behave in the same way while solving programming tasks. There are two general types of novice programmers: stoppers and movers (cf. [PHHS89, pp. 265]. Stoppers are students that do not know how to solve the problem or which step to take next. In such a case they stop solving the problem admitting that they are not able to solve it. The opposite behaviour can be observed in mover students. They try out new ideas, constantly writing, testing and modifying code. Problems arise when they write and neglect code with out reflecting their work or asking why a certain solution did not work [PHHS89, p. 266]. This type of student is classified as extreme mover. Unfortunately stoppers and extreme movers are most common among novice programmers.

### 2.2.3 Learning Functional Programming Languages

After considering general problems of programming the next section focuses on learning functional programming languages. First a brief look is taken on the advantages that arise from learning a functional programming language. Functional programming language have a high level of abstraction thus the essence of algorithms can be taught with out distracting the students by redundant syntax, exception handling or evaluation order. Thus it is easier
to teach general concepts to students [JBH93, p. 50], [CK04, p. 116] Besides from that functional programming is a good starting point to explore different fields of computer science such as artificial intelligence, computer algebra, formal language theory or others [JBH93, p. 50].

Most of the research on errors and misconception in functional programming is done on LISP. Because Scheme is a dialect of LISP it is reasonable to assume that the errors are quite similar. The following section on functional programming is therefore based on educational studies on LISP. First errors on the syntactical level are regarded. Afterwards errors are discussed that are of semantic nature. On the syntactic level students have problems with learning functions, because for each function the arguments, their type and the type of the returned result has to be learned [WG86, pp. 5]. A second misconception that occurs quite often is connected to the quote macro which forces the compiler not to evaluate an expression. Students tend to use the quote during debugging as a response on evaluation errors [AFS84, pp. 92].

Most compiling errors result from unbalanced parentheses. Further problems arise from the usage of conditionals. Regarding control structures students have problems to distinguish the constructs IF and COND. Also they have problems formulating correct conditions using boolean expressions, especially when the NOT operator is used [Ebr41, pp. 465]. With regard to the later application of control recursive functions, control structures are important in functional languages. Students tend to ignore this importance thus control structures should be taught in detail before introducing recursion [WG86, pp. 6]. On the semantic level the same problems concerning the usage of templates and procedural skills arise as mentioned in the consideration of general programming difficulties. Like in other languages the student has to learn to divide his idea into a sequence of elemental functions and language constructs [WG86, pp. 6]. To solve complex functions this elements have to be put together to a working program. In functional languages complex functions are programmed by using recursion. How this concept is earned is described in the next section.

### 2.2.4 Programming Recursion

For many people recursion is difficult to understand because they do not encounter this concept in their daily live. Therefore the following section will have a brief look at misconceptions of novices connected with recursion.

A common view in literature is that most novices understand recursion as some kind of loop or iterative structure [KA86, p.163],[PA85, p. 285], [BGM90, p. 138]. The main features of such a model can be described as follows [Kah89, p. 209]:
1. Their exists an entry point to the recursion. Constituent parts of the entry point are the function name and a parameter list.

2. In the centre of the loop is an action part which is responsible for manipulating the data.

3. Finally a propagation mechanism is needed which generates new parameters giving them back to the entry point.

This model can work on some types of recursion such as the tail recursion. In the loop model the base case is regarded as stopping condition by the students. For them the evaluation of the function ends at that point. In case of recursions different from tail recursion the base case is the end of the recursive calls but it is not the end of the evaluation of the function. Possible sources of this misconception can be knowledge of iterative processes which are used by students as a starting point to understand recursion [BGM90, p. 138]. Another source can arise from the over generalization of the model of tail recursion, where the base case as stopping condition works fine, to other types of recursion [Seg95, p. 401]. Further more it is possible that students do not have a model of recursion at all [Kah89, p. 222] because the concept is to difficult to understand.

In [HA02, p. 87] some common mistakes in the programming of recursive functions are listed. Often students ignoring the smallest instance of a problem such as boundary values of variables, degenerated data structure elements as e.g. the empty list. In formulating a base case students tend to avoid out-of-range values because they regard them as illegal and thus they do not apply them in their code. The problem with this behaviour increases code complexity which can hinder the learning process. It is possible that students ignore the base case completely which results in recursive functions that have no termination condition at all. Students are not able to identify a base case and write over-complex functions that have several, redundant base cases.

To avoid these problems is possible to provide student with steps they should take when writing recursive functions. Sooriamurthi [Soo01, p. 27] has developed such steps for teaching recursion. The steps are as follows:

1. Search for the base case, the smallest instance of the problem. This requires the student to develop a strategy for recognizing base cases and a procedure do generate a solution for the base case.

2. A simplifying routine has to be applied which reduces to problem size till the base case is reached.
3. A recursive call to the simplified version of the problem has to be implemented

4. The function has to be completed which can be done by finding a answer to the question: What needs to be done to build a answer of the function from the results of the recursive calls?

To enhance the understanding of recursion a model can be represented to the student. A common model is the copies model or little-man-model. Figure 2.1 shows the copies model of the factorial function. In this model each recursive function call is represented as a box. In the first line of a box all variable bindings are noted. The boxes are connected with errors that indicate the recursive call with reduced problem parameters [Wag94, pp. 42]. The model addresses the novice problems already discussed. First it is visible that a problem is only solved in parts within one box and that a smaller part of the problem is passed to a copy of the box. Secondly it is visible that the creation of new boxes ends when the smallest problem case is reached. Thirdly the final evaluation process becomes clear by the arrows that pass the result up to the first box. It can be seen that on this way the final result is evaluated.

![Figure 2.1: Copies model of the factorial function](image-url)
There is support to teach recursion to novice programmers by using conceptual models like the steps presented or the copies model. Which type of model should be used is still in question. Important is that teaching material which explains recursion on the base of a certain model, describes this model and provides a adequate number of examples that deploy an analogy how the model works and that represents the mechanism of recursion [WDB98, p. 295].

2.3 Summary

This chapter has identified steps that have to be taken in analogical reasoning. Not all of these steps can be covered by the tutorial system. The system can identify features of the current problem. These features can be used to retrieve an example the student has solved before. Then steps of adapting the example to the problem and the process of learning are on the side of the student. Another important issue is the way the examples in the instructional texts are formulated. They must consist of the mentioned parts: problem description, solution and a description of solution. Consequently the system must store these parts of each example in the text and the solutions made by the student. The consideration of how people learn to program has revealed that there are many sources of misconceptions and errors like general problems in working with computers, understanding basic concepts like variables, putting programs together from the known syntactic constructs or problems with higher concepts like recursion. A tutor system has to deal if most of these problems. Therefore a way has to be found how errors and misconceptions can be classified. Therefore the next chapter considers how an ITS is implemented to get an overview how these problems can be addressed.
Chapter 3

E-Learning and Intelligent Tutoring Systems

ITS were developed to reach a higher level of individualized instructions in the field of computer-based learning. An ideal ITS is intelligent because it is able to react on the learning progress of the user by communicating with him in an adaptive and flexible way [Kre94, p. 33].

To implement this reactions techniques from the field of Artificial Intelligence are used. The techniques used have the purpose of representing, storing and retrieving knowledge as well as the communication of this knowledge to the user. Furthermore techniques to infer new knowledge or rules from the knowledge base are utilized [Shu96, p. 573]. To get a better idea how a ITS works, its basic components are introduced in the following section. After that two examples of implemented ITS are described. At the end of the chapter the approach of Constraint Based Modelling (CBM) is introduced as an fast way to build an ITS.

3.1 Components of an ITS

Although there are many ITS which vary in their architecture, four basic components can be identified:

- the expert module,
- the students module,
- the pedagogical module,
- and the communication module.
3.1.1 The Expert Module

The expert module consist of rules, facts and procedures that represent the knowledge of the domain which should be taught to the student. It must be specific and detailed and therefore a lot of effort has to be put in exploring the domain and codifying the knowledge [Nwa90, 259]. Besides of simply storing the domain knowledge, the expert module has the task to present the information, to solve exercises in the same context as the student does and compare this solutions to the answer given by the student [CKA97, 861]. In [Kre94, p. 45] the main tasks of the expert module are summarized. It should:

- select the content that is displayed by the communication module,
- select a tutoring strategy depending on the learning process,
- control and adjust the speed of tutoring actions,
- select and generate questions to check the learning progress,
- select and generate constructive feedback,
- provide assistances and additional information to deal with gaps in student’s knowledge,
- take actions to guarantee student’s motivation during instruction.

These tasks should be seen as a range of responsibilities the expert module can have. Different tutoring systems while implement some of the tasks better then others by omitting other tasks at the same time.

Beside the above listed functionalities the central part of the expert module is the domain model. Three types of domain models are distinguished: the black box model, the glass box model and cognitive models [Nka10, p. 18]. Black box models try to solve a problem in a way different from the way a human would solve it. This kind of model is useful to check whether a solution is correct or not. Because the steps of the problem solving within the model are not available or not comprehensive, the model is not able to explain its solution to the student [And88, pp. 26]. In contrast to the unobservable calculations in an black box model, the steps of problem solving can be observed within a glass box model [Nwa90, p. 260]. Systems using this model are build by a cooperation between systems engineer and domain expert. With the help of the domain expert key concepts are identified, formalized and implemented. It has to be pointed out that the process of knowledge acquisition is a time consuming task [And88, pp. 29]. But
the resulting system is more applicable for tutoring because the knowledge base is understandable by humans. Another drawback is, that the reasoning in such a model does not take place in the same way as human reasoning [CKA97, p. 861]. In cognitive modelling the domain knowledge is modelled in such a way, that problem solving can be simulated that is similar to the problem solving process of a human. The result of such a simulation can be easy communicated to the student [And88, 34]. Cognitive modelling is based on a theory of cognition. One famous example is the ACT theory of Anderson [And93], [And96]. In ACT two types of knowledge exist: declarative knowledge which represents factual knowledge and procedural knowledge which represents problem solving skills. Declarative knowledge is represented as chunks, procedural knowledge as production rules. These rules are applied when a goal exists in the memory that corresponds to the conditional part of the rule. Other components of a rule can be the call of a declarative chunk or the setting of a new sub goal [And96, pp. 356]. When encountering a problem that can not be solved by the existing production rules, the learner will recall a similar problem and try to solve the problem by analogy [And93, p. 41]. Based on this theory Anderson built different ITS for example on the field of lisp programming or geometry.

3.1.2 The Student Module

The expert module and the student module form the central part of an ITS. The expert module provides the model of an ideal student [CKA97, 861]. In comparison the students module models the imperfect students knowledge. A students module should include all the information which could effect the learning process. It is almost impossible to create such an ideal model because the only input available for many ITS is the information entered by the keyboard [Nwa90, 260]. Accordingly the students module can be seen as abstracted beliefs of the system about the learner [HDJG94, p.4]. To represent the students misconception or errors traditionally two main techniques are applied. The first is the overlay model. This model tries to compare the behaviour of a student with the behaviour of an expert. The difference between those two states can be seen as the skills and knowledge the student has not gained yet [HDJG94, 6]. The drawback of this approach is the assumption, that the learners knowledge is a subset of the expert. Research on novice-expert developing shows that novices act differently the experts [CGF88]. A second approach is to add bug library to the overlay model. Such a model is called perturbation model. It tries to model the student not only with regard to the correct knowledge but additionally with regard to known errors an misconceptions in the domain [HDJG94, pp. 8].
To generate a bug library the following approaches were taken:

- enumerate bugs from empirical studies,
- generate bugs from a cognitive model,
- reconstruct bugs from observed errors.

None of these approaches is free of problems. Enumerating bugs depends on the analysis of a huge amount of empirical data. The effort of a generative model is not lower than the research needed for an enumerated bug library. In the case of reconstructing bugs the problem of misdiagnosis arises [HDJG94, 10].

Another type of student modelling is the learner-based modelling. These models do not model the students knowledge as a subset of the expert knowledge. The focus of learner-based modelling lies in the process of knowledge acquisition because the misconceptions are produced during that process. Problem solving rules which explain the steps taken until a misconception was created by the student, can be generated by utilizing machine learning techniques, [EC93, pp. 235]. To simulate problem solving rules their has to exist a problem space which can be predefined. The problem space is searched to find a path which leads to the incorrect user input. After the path is found machine learning methods can be used to generate paths or rules which reproduce the incorrect answer [Ohl86, p. 300].

In comparison to the bug library information about bugs are not stored in a database but are generated from previous problem solving attempts. This is a great advantage because it is not necessary to build up a bug library. On the other hand learner based modelling is strongly driven by AI techniques and is not grounded on theories of learning and human cognition [Kre94, p. 63].

### 3.1.3 The Pedagogical Module

The pedagogical module is responsible for representing the knowledge that should be taught to the student. In the first part of knowledge presentation the learning content has to be divided into units and put into a sequence. Within the units one way of structuring is to give an overview of the concept and going into detail afterwards. In contrast each concept can be explored deeply from the beginning [Pup92, p. 199]. Very often the concepts are illustrated by examples. If examples are used within the instruction the following basic conditions have to be taken into account [Pup92, p. 200]:
1. The examples must cover the taught concept, but only necessary examples should be shown.

2. Only one new concept per unit should be taught by the examples.

3. All elements of the example must be explained.

4. A restructuring of knowledge within a new unit has to be avoided.

Typically the student is asked to work on exercises after the first presentation of the knowledge. During the process of solving the exercises the module should interact with the student. The question to answer in the development of an ITS is when and how this interaction should take place. Should interaction take place immediately when the student makes an error or should student be able to explore different solution paths [Pup92, p. 200]. In this case the tutor system will start a supportive action when it is ask for it. Other methods which support exploration of the problem by the student is the possibility to monitor the student while he is working on the problem and give assistance when he makes errors. To reduce interruption from the system and allow alternative solution paths the system can intervene on basis of abstract problem states rather then on single user actions [CKA97, 852], [TM92, pp. 115].

If the tutor registers that help is needed different types of intervention are possible. The tutor can show the student a complete sequence of successful problem solving actions or he can guide the learner step by step through a problem solving task. A similar approach is to show the user just a part of a correct solution and let him try to solve the problem with this additional information. Alternatively an easier task can be presented. Another reaction which is important in the context of this work is to present known examples or an example that is opposite to the current task to show a contradiction in the solution. [Lus92, p. 142]

### 3.1.4 The Communication Module

The communication module is the user interface of the system and establish the communication between the tutor and the student (in the sense of giving feedback or providing assistance) and the student and the tutor (in the sense of asking the system for help, or submitting a solution) [Nwa90, p. 262]. There exist different forms of interaction between the learner and the system. These are important for the design of the user interface. Some of these points have been already touched in the previous consideration of the other modules. Schulmeister [Sch07, p. 181] lists four forms of interaction:
• The system is asking questions to the learner, guiding him by those questions in the form of a Socratic dialogue.

• The system does not communicate with the student until he asks for help during a problem solving session.

• The system is active by asking the student to select information and it infers from this informations differences to the expert model.

• The system stays in the background an gives advice only from time to time.

In each scenario of interaction the communication module faces a general problem. It has to translate the language of the student, which represents to some extend the mental model of the student, into the internal language of the system and vice versa. In human to human communication it is possible to bridge such differences in vocabulary by natural language dialogue which uses common background knowledge. This is not possible for an ITS because it has no access to common knowledge. Additional it is not possible so far to realize a natural language dialogue between system and student. That circumstance forces the student to apply the terminology used by the system [Pup92, p. 201].

Because the knowledge and skills acquired during the learning session with the tutoring system have to be transferred into real life problem solving situations, the user interface has to be as similar as possible to the real environment. To reach this a variety of technologies can be used such as texts and graphics, animations complemented by audio as well as embedded videos [Kre94, p. 46]. In the case of a programming tutor the editor used for solving the exercises must provide similar functionality and look-and-feel as the programming editors the student will work with in real problem situations.

### 3.2 Examples of Programming Tutors

The last section introduced the general architecture of ITS. It became clear that there are many possible ways to design the different modules of an ITS. This chapter should give an overview on two important programming tutors which have inspired this work. It will show how to realize the presented modules. Furthermore important aspects should be pointed out that will be relevant for an Scheme tutoring system.
3.2.1 The LISP Tutor

The LISP-Tutor (cf. [AR85]) was developed at Carnegie Mellon University by J.R. Anderson and its group of the Advanced Computer Tutoring Project in the 1980s. The aim was to develop a tutor system for teaching LISP as an entry to the study of artificial intelligence. The LISP tutor should improve the learning effect of classroom teaching by providing a system that acts as close as possible to a human tutor while students are solving programming tasks. The LISP tutor provides a coding environment in form of a structured editor rather than a simple text editor to enhance the structure of programming problems. Whenever the student makes an error or ask for help the tutor gives a feedback that should help the student to move back to the right solution path.

The expert module contains a domain model in the form of an ideal students model. This model is realized as a simulation of a problem solving path an ideal student would take. The GRAPES production system is used to represent the problem solving rules as production rules. A example would be the rule for using the APPEND function which constructs a new list from two single lists:

\[
\text{IF the goal is to combine LIST1 and LIST2 into a single list}
\]
\[
\text{THEN use the function APPEND and set a subgoal to code}
\]
\[
\text{LIST1 and LIST2}
\]

Besides production rules that represent correct problem solving steps the system stores buggy production rules that make it possible to detect errors and misconceptions of the students. The production rules and the buggy rules where created by observing students solving LISP problems and collecting error data (cf. [AFS84], [PA85]).

The students model is created in a process called model tracing. The student is monitored during each key stroke he does. In the background the tutor tries to match the current input with a production rule or a buggy rule. If the input matches a correct rule the tutor stays passive otherwise it starts an intervention. To guide the student to the correct solution the system switches into the planning mode where the student tries to develop an algorithm for the problem. In the planning mode the construction of the new algorithm is assisted by showing the student a similar example that he has already solved. If the student has planned his algorithm but still has problems while coding, the tutor gives hints in the form of short queries that remind him which goal he has to code next or reveals a part of the final solution.
The communication between student and tutor takes place via a structural editor. The editor automatically balances parentheses and provides place holders for the arguments of LISP-function. Anderson argues that this helps the student to focus on the planning of its algorithm and increases learning speed. A more technical argument is that the usage of place holders enable the tutor to recognize on which sub goal the student is working. By that the communication between student and tutor is improved.

The LISP tutor is one successful example for the implementation of an ITS. The evaluations conducted by Anderson show that the tutor is more effective then letting students learn alone. On the other hand it is not better or equal to a human tutor. Two design decisions of the LISP ITS have influenced the conception of the Scheme tutor. First the communication via a structured editor and secondly the planning of the programming task by examining an example. In this context it is important to note that the LISP tutor is criticised for its directive style of forcing the student to a solution path which is defined by the production rules ([Sch07, p. 210], [BC88, p. 391]). Care has to be taken that the usage of a rich editor environment is not used by the system to restrict the student to a certain solution.

3.2.2 The ELM Programming Environment

The ELM-PE (cf. [WM95] and [WM94]) is a tutor similar to the LISP tutor of Anderson. Both have the aim to teach the basics of the LISP programming language and both use cognitive student modelling. As the following sections will show the ELM-PE has a richer communication model and its feedback behaviour is more closer to learning from examples.

The ELM tutor provides a complete programming environment which provides debugging facilities for the student to reflect about their errors. In general the system stays passive until the student asks for help. Immediate assistance is only given in the case of syntax errors. It is important to correct such errors as soon as possible to enhance the learning process [WM95, p. 379]. From the technical point of view it is also better to pass correct code to the diagnose component of the system. To enable the student to explore the system and the learning content more individual without interruptions from the tutor three modes exist: the listener mode, the editor mode and the exercise mode. Each mode provides different support facilities. The tutor stays quiet passive in the listener mode. In the listeners mode the system acts very close to a standard programming environment. Error messages are shown if syntactical wrong code is evaluated. As an addition this error messages are explained to help novice programmers understanding them. Another function is the highlighting of parentheses which should not be confounded with
the parentheses balancing of the Lisp tutor. In the editor mode the system switches to a structured editor. The structural editor ensures that the code stays free of syntax errors. To support problem solving examples from the lecture can be displayed in a separate window. To encourage the student to reflect his errors a debugger with a stepper function is provided by the system. Additionally erroneous code gets highlighted. In the exercise mode the highest level of support is provided. The student selects a problem he wants to solve. The system displays a problem example together with some input-output examples. After the student submits his solution the system test the correctness of the solution by evaluating the student’s function with the input-output pairs. If the function produces the correct output for each pair the system accepts the solution as correct. Otherwise the student is shown which pair does not work and is asked to debug the program. If the student needs further help a cognitive analysis based in the Episodic Learner Model is accomplished.

The ELM is described in [WW87] [WB95] and [Web95]. In the following a short description of the model will be given. Four sources of information are included in creating the model: the task description, the student’s solution, the domain model and the current student model. The task description is related to higher concepts and plans. According to the task description a concept is selected. A concept consists of plans and rules to reach the goal which is specified by the current plan. All rules of a concept are executed until function name, a parameter or a constant is matched. The execution of a rule can create new sub rules or a comparison with the students code. The result of this recursive process is a derivation tree that comprises all rules that explain the students solution. Episodic frames are created from concepts in the tree which contain a sub plan, the rule used to solve the plan and the corresponding part of analysed code. The frames are added to the knowledge base as instances for their concepts. The set of episodic instances forms the episodic learner model. Accordingly an episode consist of a set of frames that is referenced to as case in the case library. One frame identifies the case and indexes all other frames in the episode enabling the reconstruction of the derivation tree and by that the context of plans and rules. The information from episodic instances can be used for tutoring actions such as the reusing of previous explanation or for application of similar plans to the a part of the solution that matches a similar plan. By predicting problem solving behaviour from the derivation tree, useful examples or remindings can be retrieved to assist the learner in problem solving. Analogous examples can be retrieved from the concept hierarchy (cf. [Web95, pp. 170]. The similarity between the current problem or solution and the instances of a concept is calculated by the contrast model (see Chapter 4.3) which enhances
that semantic similarity plays an important role in retrieving an analogous example.

To put it in a nutshell, the ELM-PE avoids some of the drawbacks of the LISP tutor such as the strict control of the solution path. Also, the philosophy of learning from examples is more strongly embedded in the system then in the LISP tutor. With regard to the Scheme tutor, it seems important to enable the student to explore the problem cases as free as possible by providing him the possibility to ask for help if needed. Additionally, an editor should be implemented that realizes useful features such as parentheses highlighting or code patterns without risking the tutor to intervene when using these functionalities.

3.2.3 Summary

The two presented tutors show the capability of the ITS so far developed. The LISP tutor has been used for many years at the Carnegie Mellon University. For the EML-PE, there is a web-based version available called EML-ART (cf. [BSW96]). In summary, the following features have influenced the development of the Scheme tutor:

- The usage of a structured editor which helps the student to avoid syntax errors and unbalanced parentheses.
- Providing help by showing the student an example from a previous problem solving session or lecture which is retrieved from a concept hierarchy by a similarity measure.

Nevertheless, it took many years of research to develop these two systems. The reason is that both tutors need a great knowledge base for simulating the students’ problem solving process. The observation of students’ creation of production rules is time and resource consuming. Even more difficult is the creation of buggy rules because there are so many misconceptions and errors that it seems impossible to model all of them. For this reason, the approach of constraint-based student modeling is explained in the next section.

3.3 Constraint Based Tutoring Systems

This section describes how an ITS can be built using CBM. First, the underlying theory of learning from performance errors is explained. After that, the CBM is regarded. Finally, an example of a CBM-based tutoring system is introduced.
3.3.1 Learning from Performance Errors

The theory of learning from performance errors was proposed by Ohlsson [Ohl96] and tries to explain how people detect and correct their errors during problem solving. The theory applies to the learning of complex skills such as computer programming [Ohl96, p. 242].

Problem solving is seen as process of stepping from one situation to another. In each situation multiple actions are possible leading to other situations. This task environment can be represented as a situation tree where a node is a situation and the branches are actions transforming one situation into another situation. Within the tree the perceive-decide-act cycle is processed at each node. That means that the learner as to recognize a situation (a node in the tree), decide among the possible actions (decide which branch to take) and apply the chosen action (move to the next node [Ohl96, p. 243].

To selecting the right action practical knowledge is needed. This knowledge states which action has to be taken to reach a goal in current situation. Practical knowledge can be formalizes as production rule:

\[ G, S \rightarrow A \]

Where G is a goal that should be reached and S is the situation in which the goal should be reached. The letter “A” depicts the action that is to be taken to reach the goal. A problem solving method can consists of many of those rules. How can a human detect an error in such a knowledge structure? Error detection takes place by comparing the actual outcome with the expected outcome of the action. This implies that actions can not be stated as correct or incorrect in general, they have to be evaluated according to the current situation. A discrepancy of the current situation from the expected situation is recognized by features of the produced situation which are called error signals [Ohl96, p. 245].

To evaluate the outcome of a declarative knowledge is needed. In Ohlssons theory declarative knowledge has a judging function rather then being a simple representation of facts. A part of the declarative knowledge is encoded in human brain as constraints that describes what ought to be in a certain situation.

Errors are constraint violations and indicate that the action generated by the practical knowledge is incorrect thus the practical knowledge itself is incorrect or incomplete ([Ohl96, p. 247]. If a human applies problem solving on a unknown task, he uses overly generalized rules to solve the task. Errors are a result of such over generalized practical knowledge and are corrected by specializing this knowledge.
Specialization is needed because incorrect rules can stay in memory with correct rules overlaying them [Ohl96, p. 248]. Because all activated rules are considered by decision making and the rule with the highest activation is selected it is possible that an incorrect rule outmatches a correct rule.

To correct the wrong rule in such a way that it never applies to the current situation or similar situations three steps have to be taken:

1. **Blame assignment**: To identify the rule that has to be revised, the action which was responsible for the error must be identified.

2. **Error attribution**: In the next step the situation has to be analysed. A situation consist of features. The aim is to find the feature that in combination with the action caused the wrong outcome.

3. **Rule revision**: The rule is specialized in such a way that it is used only in such situations that do not result in the erroneous outcome. Ohlsson states that the situation part of the rule has to be constrained to a state which excludes the current state [Ohl96, p. 248].

The process of blame assignment and error attribution is a complex reasoning task for the individual because one has to explain to oneself where and why the error occurred. If the student does not have enough declarative knowledge it is not possible for him to detect his errors. At this point a human tutor or an ITS can intervene to point out the error and start the reasoning process on the side of the learner [Mit10, p. 66].

### 3.3.2 Constraint Based Modelling

The constraint based modelling approach was developed to overcome some of the drawbacks of the cognitive modelling approach such as the need of a runnable students model or the creation of a bug library which both require large scale empirical studies [MKM03, pp. 313]. Ohlsson ([Ohl94]) who developed the approach argues that knowledge in the student model must be relevant for tutoring actions and that (buggy) knowledge which is not resulting in a pedagogical advise is not useful at all. Furthermore the space of possible misconceptions and errors is too big so that abstraction to pedagogically relevant equivalences classes has to be made [Ohl94, p. 174]. Therefore the domain knowledge should be considered as a set of constraints that describe correct solutions. By that the constraints represent fundamental principles that are not allowed to be violated by a solution. Equivalence classes are formed by each constraint, dividing possible solution in a set of solutions that full fill a constraint and those that do not [Ohl94, p. 177].
A constraint is expressed as an ordered pair $\langle C_R, C_S \rangle$ in which $C_R$ is the relevance condition which identifies the problem state in which the constraint has to be applied and where $C_S$ is the satisfaction condition which specifies the conditions under which the current state is valid [OR91, p. 119]. The natural language pattern for a constraint would look as follows:

\[
\text{IF } \langle \text{relevance condition} \rangle \text{ is true,}
\]
\[
\text{THEN } \langle \text{satisfaction condition} \rangle \text{ had better also be true.}
\]

Even if the natural language representation of a constraint looks similar to a production rule they are quite different. Production rules form the base for running a simulation of the problem solving by generating new action on execution. A constraint has a control function by checking whether a solution state is valid for the domain or not [Mit10, p. 65]. This corresponds to the theory of learning from performance errors where constraints realize the evaluative character of declarative knowledge.

In CBM it is not important to know which concrete misconception was responsible for a wrong solution. Important is to know which constraint failed in evaluating the solution. Feedback can be given explaining the student which domain principle his solution has violated. Furthermore a feedback message can be attached to each constraint. CBM enables the student to explore the solution space because there is no restriction to a solution path. All solutions fulfilling the constraints are correct [Mit10, p. 65]. The challenge is to select constraints that represent pedagogically significant states [Mar99, p. 32].

Even if CBM provides a modelling approach which enables a less time and effort consuming development of ITS [MKM03, p. 321] there are three disadvantages which have to be taken into account when building a tutoring system on the base of CBM (cf. [Ohl94, p. 1187]):

1. There can exist domains where it is not possible to identify attributes of problem states which provide adequate pedagogical information.

2. It is possible that the set of constraint is not specific enough thus some incorrect solutions are not violating the constraints.

3. Constraints can be ill defined, providing not the right level of abstraction. Even worse it is possible that the do not form relevant equivalence classes at all.
3.3.3 Example of a CBM Tutor

The following chapter will briefly describe the SQL-Tutor (cf.[MO99]) as an example of a CBM tutor. To get an idea how a CBM could be implemented the three main components are described: the communication interface, the knowledge base and the students modeller and the pedagogical module.

The user interface is separated into three parts. The upper part displays the problem description. In the central part the user can enter his solution. This part is already structured according to the possible components of a select-from-where SQL statement. In the lowest section the schema of the current database is displayed. Descriptions for SQL statements, tables and attributes can be accessed by the user. The tutor is also checking the syntax on a low level enabling the student to focus on high-level features of the problem. The tutor stays passive until the student submits its solution [MO99, p.243].

After submitting the solution is passed to the students modeller. The student modeller checks the solution of the student for correctness. The modeller manages two structures: the relevance network and the satisfaction network. Each network consists of input, output and test nodes. The student’s solution and the ideal solution a propagated through the relevance network to generate a list of constraints that describe the current problem solving situation. In a second step the satisfaction conditions are checked. A violated constraint is recorded. The set of violated constraints forms the student model. Additionally the history of each constraint is saved to the model [MO99, pp. 246].

Constraints are stored in the knowledge base. The version of the SQL-Tutor described here contains 406 constraints. The source for those constraints where an analysis of the target domain and a collection of incorrect students solutions. Constraints are coded as LISP expressions. Two types of constraints are distinguished:

- **Syntax constraint:** These are constraints that describe the syntactic correctness of a query. Syntax constraint contain a verbal description that is used in the feedback.

- **Semantic constraint:** These are constraints that cover the meaning of the elements in a query. They are more complex then syntax constraints.

The high number of constraints is explained by their high modularity. This enables the designers of the system to attach feedback messages to each constraint and improve student modelling and instruction [MO99, pp.244].
The pedagogical module is responsible for selecting exercises and the generation of feedback. Because a solution can violate more than one constraint, the pedagogical module checks in the constraint history which of the violated constraints was violated most in earlier problem solving sessions. The feedback for this constraint is displayed to the student. According to the number of solution attempts the feedback varies in detail. There are five levels of detail: right/wrong feedback message, error flags informing the student in which clause the error occurred, hints providing more information about the error, a partial solution or a complete solution [MO99, pp. 246].

3.4 Summary

From the considerations of different tutor systems that use different techniques the following features of the STAEREX system can be derived:

- The system is designed according to the four-components architecture.
- The system uses a structured editor as problem solving environment.
- The tutoring process is controlled by the learner. He asks for help only when needed.
- Learning from examples is the main feature of the system.
- The tutor models knowledge by usage of CBM. Additionally constraints should be used for retrieving examples.
Chapter 4

Conception of the STAEREX system

In the following chapter the conception of the STAEREX system is explained. The considerations are structured according to the four modules of an ITS that have been discussed in the previous chapter. The chapter describes the system on a conceptional level. Implementation aspects are regarded in chapter 5.

4.1 User Interface

The user interface of the STAEREX system is determined by its nature as a web application. Thus the student will communicate with system via a web browser that displays html pages. The creation of a nice looking web interface that is oriented on the principles of human computer interaction is out of the scope of this work. The focus lies on the interaction with the system during the problem solving process. Figure 4.1 illustrates the conceptual design of the page displayed to the user when working on a problem. The central component is a structured editor which is used to enter code. The editor should provide the following functionality:

- A highlighting of parentheses the avoid errors resulting from unbalanced parentheses.
- A code completion function which displays scheme keywords when a letter is typed.
- Highlighting of important key words.
The student should interact with the system by using two buttons: "Run" and "Ask". The Run-Button is used to evaluate the code written by the student with regard to syntax a functionality by testing it on a predefined list of input-output pairs. The Ask-Button starts the retrieval of an example that should help the student to find the correct solution. Their also has to exist an output window which displays messages from the system. The window displays only messages that arise from the evaluation after pressing the run-button. These messages can include the descriptions of errors while interpreting the code or a information that the code failed to produce an expected result in the input-output test. To display an example a special area on the rightmost part of the screen is used. There the name of the example, the problem description, input-output pairs and the solution for the example is displayed.

4.2 Expert Module

The task of the expert model is to find out what is wrong in the students solution. Therefore it checks the solution against the domain model which is modelled by constraints (see Chapter 3.2). Similar to the SQL-tutor two types of constraints can be distinguished. In the STAEREX system these
two types are refereed to as syntax constraints and problem constraints.

4.2.1 Syntax Constraints

Syntax constraints model the syntactical features of the scheme programming language. A syntax constraint is defined for each function that can be used in Scheme. To implement all possible syntax constraint for the scheme language is out of the scope of this work. Thus only syntax constraint where implemented that where needed for the prototype implementation of the system. From the language constructs taught in the modules of the course the following constraints can be derived:

- A constraint for using basic arithmetic operators.
- A constraint for defining function
- A constraint for the call of user defined functions
- A constraint for using predicates.
- Constraints for the conditional operators if and cond.
- Constraints for the list processing functions car and cdr.

The design and implementation of a constraint will be discussed on the example of the scheme expression `define`. The remaining constraints are explained in the appendix. To create a constraint for scheme expression the general syntax of the expression is regarded. For define the general syntax would be as follows:

\[
\text{(define (ируют (function name ) (parameterlist ) (function body))}
\]

From this definition one can see that the define expression consists of two parts, a head and a body. This can be formulated as the first constraint for the define expression:

*Define Constraint 1:*

**IF** the S-Expression starts with `define`

**THEN** their have to exist two arguments a function head an a body.

To complete the set of constraints for define the two identified parts have to be investigated. In case of the function definition one can state that a function definition must consist of a function name and at least one parameter.
In other words it must have at least two elements. The resulting constraint is the following:

Define Constraint 2:

IF the S-Expression starts with define
THEN the function definition has to contain two arguments.

In the function body the application logic of the function has to be defined. This includes the usage of other Scheme expression. The evaluation of these expressions has to be done by other constraints. Thus in the define constraint it is enough to state that the body has to contain some Scheme expression:

Define Constraint 3:

IF the S-Expression starts with define
THEN their has to exist a S-Expression in the body

The identified constraints have to be evaluated one after another starting with Constraint 1 which checks for the existence of the required elements. After that Constraint 2 and 3 are regarded which check each required element (the function definition and the body). The next point to consider is which action should be taken when one of the constraints fails. This is a pedagogical consideration because it has to be guessed which wrong or incomplete knowledge of the student caused the fail of a constraint. Because the expert model is responsible for identifying students error it does not take any corrective action it rather passes a indication what is wrong in the students solution to the tutor component. For the define constraint and all other constraints one indication is obvious. If the constraint fails the student did not understand the used language construct. This holds for the discussed constraints 1 and 2. In case of the function body the situation can be different. The function body is the part of the function definition where actual problem solving takes place. If a student leaves this part empty or does not provide any Scheme code their one can guess that he is not able to solve the problem. Thus in case of constraint 3 the indication that a action has to be taken which is aimed to support the student in problem understanding.

4.2.2 Problem Constraints

Problem constrains try to describe the way in which a certain problem can be solved by a Scheme program. By assigning a problem constraint to an example the example is classified as belonging to a certain type of problem. If a problem constraint fails an example can be retrieved from this
class of problems. The crucial point in formulating problem constraints is to build up a system of problems that is described by the constraints. In every programming language their exist programming patterns that could be used for formulating problem constraints. But these patterns are influenced by technical considerations. Therefore a pedagogical motivated approach is adopted to formulate problem constraints. Additionally the used approach should not be overspecialised so that the sets of problems become too small to assign a reasonable amount of examples to them. A attempt to classify functional programming problems that meet the mentioned condition and is furthermore aimed to introductory functional programming was proposed by Soloway [Sol85]. He describes three types of plans that are used in list processing:

- **Predicate Problems:** Problems that check a list for the fulfilment of a certain condition. The result of the evaluation of the function is either true or false. An example would be a function that checks if a list contains a number.

- **Builder Problems:** Problems that perform some operation on one or more input list and return a modified list as a result. An example would be a function that deletes a given element from the list.

- **Selector Problems:** Problems that return a element of a list depending on some condition. An example would be a function that returns the first number in a list.

In each problem class the distinction between OR- and AND-problems can be made. In the case of an OR-problem the list is processed until a certain condition is meet. In the case of the AND-problem the whole list must be proceeded until a result is returned. Consider predicate problem of the function that checks if a list contains a number. That would be a OR-Problem because the processing stops with the occurrence of the first number. An equivalent AND-problem would be a function that checks if all elements of a list are numbers. In that case the list has to be processed to the end before a result can be returned.

In the prototype of the STAERE system four types of problem constraints are used:

- Both types of predicate constraints: OR-Predicate-Constraint and AND-Predicate-Constraint.

- The OR-Selector-Predicate-Constraint.
- The AND-Selector-Predicate-Constraint on numerical lists.

The last constraint is chosen as a special form of the AND-Selector problems. This is due to reduce complexity in implementing the tutor system. Adding pure AND-Selector problems and builder problems would require the additional syntax constraints for list constructing problems. That effort was out of the scope of this work and is a task for further improvement of the tutor component and the course material.

A fifth class of problems is used by the systems for problems that do not require recursion. Examples not using recursion are called basic problems.

To formulate constraints for the four problem types a closure look at the way how recursive problems are build are taken. Their are three main parts of a recursive statement [Pir86, p. 329]:

- Their has to be a termination case where the recursive call ends. This is also called base case in the previous consideration of recursion.
- A recursive call where the function calls itself with modified parameters that represent a reduction of the problem towards the termination case.
- A recursive relation which defines how the result of the function call is constructed from recursive calls. This is the part of the function where the actual processing is done.

When formulating a problem constraint these three points have to be regarded. Each part of the recursive function has to be analysed. Result of the analysis should be a constraint for each part that is covering each example in the problem class. The formulation of a problem constraint is shown using the predicate problem. An overview on the other problem constraints is given in the appendix.

An OR-Predicate Problem is a problem where the task is to check if a certain condition is fulfilled for a list. This condition is always aimed on one element of the list e.g. check if the list contains a number. First we consider the required recursive call for the function. To reduce the problem complexity we call the function on the rest of the list. That is the first condition that has to be meet to fulfill the constraint. Now we can consider when the recursive call should end. This should happen at the end of the list which is indicated by the empty list. When the empty list is reached that means that no element in the list meets the condition thus false has to be returned. By this we have the second condition. The recursive relation for Or-Predicate problems is that their has to exist on a conditional part checking for the first element of the list against some condition. This condition is problem specific and
is not defined in the constraint. The conditional has to return true if the condition is meet. In the following the three aspects can be summarized in a constraint:

\[
\text{IF} \quad \text{the problem is a OR-predicate problem,} \\
\text{THEN} \quad \text{their has to be a test for the empty list returning false,} \\
\text{AND} \quad \text{their has to be a recursive call on the rest of the list,} \\
\text{AND} \quad \text{their has to be a test on the first element of the list returning true.}
\]

The constraint for an AND-predicate Problem looks quite similar. Different are the conditions for the termination condition and the recursive relation. In such problems each element must be checked for a problem specific condition. This condition has to return false indicating that the current element harms the checking condition and returns false. For example a function checking if a list contains only numbers has to return false when encountering an element that is a character. Consequently the termination condition checks if the list is empty and returns true indicating that all elements meet the condition. The AND-Predicate-Constraint looks as follows:

\[
\text{IF} \quad \text{the problem is an AND-predicate problem,} \\
\text{THEN} \quad \text{their has to be a test for the empty list returning true,} \\
\text{AND} \quad \text{their has to be a recursive call on the rest of the list,} \\
\text{AND} \quad \text{their has to be a test on the first element of the list returning false.}
\]

Like in the case of syntax constraints the expert module does not take any action when a constraint fails. It returns a indication to the tutor component that it should retrieve an example from the problem class that the constraint checked for.

### 4.3 Student Module

As discussed in the previous chapter the students module is the place where the knowledge of the student is modelled. Because the main aspect of the STAEREX system is the retrieval of examples which should help the student to solve a problem, the information stored in the students model is a list of examples the student has seen in the course texts and solved on his own. An example in the students model consists of the following components:

- The name of the example.
• The problem class the example belongs to.
• A description of the problem that has to be solved by the example code.
• A set of input-output values that illustrate the desired behaviour of the example code.
• A Scheme function definition that solves the described problem by transforming the given input values to output values.
• A list of syntax constraints for the language constructs used in the code.

While the student is working on one example it is possible that he is asking for help more than ones. It is assumed that such behaviour means that the retrieved example is of no use for the student and he wants to get another example. The students model maintenance a list of already seen examples to ensure that no example is displayed twice in a problem solving session.

The retrieval of an example works according to the following algorithm:

**Algorithm 1 Retrieve example for a constraint**

\[
\text{ListExamples} \leftarrow \text{load examples known by the student} \\
\text{ListSeen} \leftarrow \text{all examples shown to student in the session} \\
\text{Indication} \leftarrow \text{indication which concept was not understood by the student} \\
\text{RetrievedExamples} \leftarrow 0 \\
\text{while} \ \text{ListExamples} \ \text{has more elements} \ \text{do} \\
\quad \text{Example} \leftarrow \text{next example from ListExamples} \\
\quad \text{Constraints} \leftarrow \text{syntax constraint list from Example} \\
\quad \text{if} \ \text{Constraints} \ \equiv \ \text{Indication} \ \text{then} \\
\qquad \text{RetrievedExamples} \leftarrow \text{RetrievedExamples} + \text{Example} \\
\quad \text{end if} \\
\text{end while} \\
\text{Remove all examples in ListSeen from RetrievedExamples} \\
\text{return \ first element from RetrievedExamples}
\]

Besides the two lists, the model needs an indication to retrieve an example. This indication comes from the expert model as the result of a failed constraint. To retrieve an example one has to iterate over the examples the student has studied so far. An example that has attached the constraint that is indicated by the indication is added as a possible result. After all examples has been examined for their constraints the examples which have already
been seen by the student in the current problem solving session are removed from the result list. After that the result list is sorted and the example on the top of the list is returned as the retrieved example. The algorithm is identical for syntax and problem constraints. The difference is the way the result list is sorted. In case that a syntax constraint was violated the sorting takes place according to the number of syntax constraints of each example. After sorting the example with the smallest amount of constraints will be added to the top. This ensures that the example is shown to the student that is closest to his current misunderstanding. A failed syntax constraint indicates missing or wrong knowledge about a language construct. Thus a example should be shown that addresses this language construct and does not include any other language constructs.

In the case that a example should be retrieved that is similar to the current problem the situation is different. To understand how to solve a problem the student has to understand how different language constructs work together. Thus from the examples that have the same problem class as the indication indicates this example has to be retrieved that is most similar to the solution submitted by the student. The similarity is evaluated on the base of the syntax constraint in the example and the submitted code. To evaluate the similarity the contrast model is used:

$$S(C, E) = F(C \cap E, C - E, B - E)$$

The similarity of the code and the example is calculated subtracting the numbers of constraints that are only in the code and the number of constraints that are only in the example from the number of constraints that are in the example and the code.

### 4.4 Pedagogical module

The pedagogical module manages the sequence of learning units presented to the student. In the prototype of the STAEREX system this sequence is fixed. So the module describes a fixed sequence of learning modules or learning units and a fixed sequence of exercises that have to be solved in each unit.

#### 4.4.1 Steps for the conception of e-learning environments

When designing the sequence of an e-learning system different steps have to be taken. The steps undertaken in the conception of the pedagogical mod-
ule are derived from the steps described in Kerres(1998), Niegemann(2001), Issing(1997). The steps undertaken are shown in Figure 4.2

![Figure 4.2: Steps in designing learning content](image)

The first step is the analysis of the target audience. This can be done on the base of socio demographic characteristics, previous knowledge, learning motivation or personal attitude to e-learning [Ker98, p. 149]. In most cases a detailed analysis of the target audience is difficult because because an e-learning course is often aimed on a heterogeneous group of learners [Ker98, p. 141]. In the second step the knowledge that should be taught is structured. From the analysis of the target audience learning objectives are identified and assigned to the structured knowledge. In the third step the knowledge has to be split up and but into a sequence of learning units. In this step the question has to be answered if there should be a sequence at all or if the learner is free to select his next learning item [Ker99, p. 12].

In the following the conception of the pedagogical module is explained by exploring the mentioned steps.

### 4.4.2 Analysis of the Target Audience

Table 4.2 summarizes the analysis of the target audience. Because it is difficult to make assumptions about socio demographic characteristics of the future learners this characteristic was replaced by the notion group of learners. The group of learners will be undergraduate students of computer science. It is assumed that the course will take place in the second or third semester. Thus students will have some basic understanding of information processing by machines and programming. It is not possible to make an assumption about the learning motivation of the students. Motivation can be rather different in the learning group. It can range from high personal interest in studying functional programming to simple collecting necessary credit points. It is assumed that the students will not have much experience with an e-learning setting. They will be used to classroom lessons and lectures. The time used for the course can be stated to be one semester while a parallel lecture on functional programming is visited. Besides that lecture it is planned that students invest two to four hours per week to work on exercises with the STAEREX system.
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group of learners</td>
<td>undergraduate students</td>
</tr>
<tr>
<td>Previous knowledge</td>
<td>Basic computer knowledge. Maybe knowledge of a first programming knowledge</td>
</tr>
<tr>
<td>Learning motivation</td>
<td>no statement bout the motivation possible</td>
</tr>
<tr>
<td>Learning habits</td>
<td>classroom learning, institution bound</td>
</tr>
<tr>
<td>Learning time</td>
<td>one semester, 2-4 hours per week</td>
</tr>
</tbody>
</table>

Table 4.2: Characteristics of the target audience

4.4.3 Learning Content and Learning goals

After the characteristics of the learners are analysed an research on possible learning content was undertaken. Therefore two box on Scheme programming where analysed and compared with each other ([Smi88] and [SF89]). From this analysis a set of topics is derived that should be covered by an introductory course of scheme programming. The following topics where identified:

1. An introduction to Scheme and functional programming including the history of functional programming and its differences to other languages.

2. Scheme basics such as important language elements (e.g. atoms, lists, S-expressions) and standart arithmetic operations

3. Definition of variables and functions.

4. Introduction to list processing by introducing operators on lists (car, cdr, operators for joining lists).

5. Conditional execution by using if, cond and case, boolean operators and predicates.

6. Repetitive execution by recursion.

After this content can be structured into learning units and one has to identify learning goals. On popular taxonomy was contributed by [BEF\textsuperscript{+}56] The taxonomy differentiates three types of learning goals:

- **Cognitive**: Aimed on acquiring declarative and procedural knowledge as well as rules and problem solving procedures
• Affective: Are aimed on changing attitudes, interests and values of a student

• Psychomotoric: Aimed on the ability to physically use and manipulate instruments

For this work the acquisition of cognitive problem solving skill stays in the focus. Therefore cognitive learning goals are formulated for the STAEREX system. The different levels of those learning goals are: knowledge, comprehension, application, analysis, synthesis and evaluation. For the study of basic Scheme programming the following cognitive learning goals are formulated:

• Having knowledge about basic operations in Scheme.
• Having knowledge about list processing operation in Scheme.
• Applying knowledge about Scheme functions to solve simple problems.
• Having knowledge about different types of recursion and how they can be implemented in Scheme to process lists.
• Applying recursive concepts to new problems which are similar to previous problems.
• Being able to analyse problems and to solve them using the acquired concepts about Scheme and recursion.

4.4.4 Sequencing of the Course

The identified learning goals indicate a structure that has to be followed when the aims should be reached. First the student needs knowledge about the basic Scheme language constructs. Then he has to apply that knowledge to deepen it and to be able to solve different problems with the basic Scheme operators. After that the concept of recursion can be introduced. The students should learn how recursion is implemented in Scheme. After that they should be able to transfer the learned procedures to new problems. According to this general structures the mentioned content is accumulated into five units respectively learning modules:

The modules are organized in such a way that the student gets the knowledge he needs to define his first Scheme function on its own as soon as possible. By this the interactive process between learner and the tutor system can start early. It is assumed that when the student starts to solve exercises the motivation increases thus this process should start as early as possible.
The instructional text are similar in each model. They describe the concept that should be acquired using concrete examples. After that the student has to solve a fixed set of exercises before he can continue with the next module. The instructional text, the examples used in each unit and a description of each example can be found in the appendix.

<table>
<thead>
<tr>
<th>Learning Module</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 1: Introduction to Scheme</td>
<td>• An brief introduction to the history of Scheme.</td>
</tr>
<tr>
<td></td>
<td>• Introduction to the special feature of a functional language (prefix notation and the evaluation of expressions).</td>
</tr>
<tr>
<td>Module 2: Scheme Basics</td>
<td>• Usage of the basic operators addition, subtraction, multiplication and division.</td>
</tr>
<tr>
<td></td>
<td>• Definition of own functions using the basic operators.</td>
</tr>
<tr>
<td>Module 3: Conditional Statements</td>
<td>• Usage of the conditional statements if and cond.</td>
</tr>
<tr>
<td></td>
<td>• Usage of operators for comparison such as greater, smaller, equal.</td>
</tr>
<tr>
<td></td>
<td>• Usage of predicates.</td>
</tr>
</tbody>
</table>
| Module 4: List Processing I | • Usage of the list processing operators car and cdr.  
|                            | • Explanation the concept of recursion in general.  
|                            | • Explanation of recursion on predicate problems.  |
| Module 5: List Processing II| • Explanation of recursion on selector problems.  
|                            | • Pointing out the difference between selector and predicate problems.  |

Table 4.3: Learning modules of the STAEREX system
Chapter 5

Implementation Aspects

After the different modules have been designed on conceptional level important aspects of the implementation of the system are regarded. Because the system is a prototype the focus lies on functionalities that are important to know to extend the system. To get an overview on the way the system works it is describes how the tutor component deals with user actions. A second important aspect is the design of the Client-Server interface. Last but not least it is important to explain how informations are stored in the STAEREX data base because adding new data like exercises, examples and solutions is essential when extending the system.

5.1 Implementation of the Tutor component

The tutor component consists of three classes which control and perform the actions to guide the student. The first class is the SchemeTutorServlet which evaluates what action was performed by the user. This action can either be submitting code or asking for help. When code is submitted the servlet calls the SchemeInterpreter class to perform the evaluation. If the student asks for help the SchemeTutor class is responsible for analysing the solution and returning an example.

5.1.1 The Tutor and the Interpreter Class

The SchemeInterpreter class evaluates the code of the student. The class diagram for the interpreter is shown in Figure 5.1. The class implements two important methods: interpret() and checkOnIOExamples(). The first method uses the JScheme interpreter to check the syntactic correctness of

\footnote{see Chapter 5.1.2}
the student’s solution. If the check fails the attribute feedback is filled with the message returned from the JScheme interpreter.

```
<table>
<thead>
<tr>
<th>SchemeInterpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>code: String</td>
</tr>
<tr>
<td>feedback: String</td>
</tr>
<tr>
<td>sp: SchemePair</td>
</tr>
<tr>
<td>jScheme: JScheme</td>
</tr>
<tr>
<td>parsedCode: ParsedSchemeCode</td>
</tr>
<tr>
<td>interpret(): boolean</td>
</tr>
<tr>
<td>checkIOExamples (testData: List&lt;IORelation&gt;): boolean</td>
</tr>
</tbody>
</table>
```

Figure 5.1: The class SchemeInterpreter

The second method checks syntactical correct code on a list of input-output examples. This is done to ensure that the submitted solution works correctly. If this method returns false the attribute feedback is filled with a message that indicates on which input-output example the submitted function calculated the wrong result. The result of the interpretation is returned to the SchemeTutorServlets which passes the result of the evaluation to the user.

The SchemeTutor class manages the analysis of the submitted code. The class diagram is shown in Figure 5.2. The processing is done in the method giveHelp(). Within that method the code is parsed. After that the tutor calls an instance of the SchemeDomainModel. Then the methods checkSyntaxConstraint and checkProblemConstrasts are called. If one of these methods returns false, that means that a constraint failed. For that case the SchemeDomainModel holds a indication which is passed by the SchemeTutor to the StudentModel to retrieve an example.

The indication is an enumeration called FailIndications. This enumeration assigns to each indication a short description. These descriptions are used in the methods checkSyntaxConstraint and checkProblemConstrasts to check for the different constraints. The descriptions are also stored in the data base to identify the constraints. When adding a new constraint the connection between FailIndications, the checkConstraint methods and the data base has to be regarded. A description for adding new constraints can be found in the appendix.
Figure 5.2: The class SchemeTutor
5.1.2 Parsing a SchemePair

The SchemeTutor and the SchemeInterpreter class make use of the JScheme interpreter. JScheme is a Scheme interpreter which is freely available on the web. This interpreter was used because it is easy to use and well documented. Thus it was easy to integrate it to the system. The JScheme interpreter returns as result a object of the class SchemePair. A SchemePair consists of a first element and a rest which both can be SchemePairs, Symbols or the empty list represented as "()". Figure 5.3 illustrates the tree like structure of this object.

The parsing of SchemePairs is done by the class SchemeCodeParser. The class diagram in Figure 5.4 shows the important classes that are related to the parser class. The result of the parsing process is a ParsedSchemeCode object. This objects consist of the function name and the parameters of the function defined by the parsed code. Additional it holds a list of AnnotatedSchemePairs. This class wraps a SchemePair by adding an information which scheme expression is represented by the SchemePair.

The parsing is done in the method parse(). Algorithm 2 shows how a SchemePair is parsed by an instance of the SchemeCodeParser. The algorithm navigates to the tree structure of SchemePairs and adds to a result list each pair that has a symbol as its first element.

---

2The source code and the documentation are available under http://jscheme.sourceforge.net/jscheme/main.html
Algorithm 2 Parsing of a SchemePair object

ListPairs ← add SchemePair
ListResult as list of AnnotatedSchemePair

while ListPairs ≠ empty do
    Pair ← first element from ListPairs
    Remove Pair from ListPairs
    if Pair.first() is S-Expression and Pair.rest() = () then
        Pair ← Pair.first()
    end if
    if Pair.first() is a SchemePair then
        Add Pair.first() to the top of ListPairs
        if Pair.rest() is a SchemePair then
            Add Pair.rest() as second element to ListPairs
        end if
    end if
    else
        if Pair.first() is a Symbol then
            Add Pair.first() to ListResult
        end if
        if Pair.rest() is a SchemePair then
            Add Pair.rest() to the top of ListPairs
        end if
    end if
end while
return ListResult
In other words the result list consists of instances of the class \textit{AnnotatedSchemePair}. To move through the tree a list \textit{ListPair} holds the \textit{SchemePairs} that are subelements of this node. At each node several checks have to be performed:

1. The first element of the \textit{SchemePair} can be a \textit{SchemePair} itself and the rest can be empty. Then the first element has to be investigated. Thus it is regarded in the further processing.

2. The first element is a \textit{SchemePair} and the rest is not empty, then the first element is added to \textit{ListPair} as first element of the list. If the rest is a \textit{SchemePair} it is added to \textit{ListPair} as second element.

3. If the first element of the \textit{SchemePair} is a Symbol then a leaf of the tree is reached and an \textit{AnnotatedSchemePair} is created from the Symbol and the \textit{SchemePair}. If the rest of the \textit{SchemePair} is a \textit{SchemePair} itself it is added to \textit{ListPair} as first element.

To summarize the algorithm it can be stated that the structure of the \textit{SchemePair} tree structure is mapped to a list by navigating through the tree by a left-most, depth-first strategy.
5.2 The Client-Server Interface

The STAEREX prototype is implemented as web application. The system is accessed via a web browser. The processing is done on a Apache Tomcat server. To generate dynamic web content java servlets and Java Server Pages (JSP) are used. For displaying dynamic content on the JSP, Java Beans are utilized which are defined in the session scope of a request. The general process for the communication between client and server is illustrated in Figure 5.5. The client is a web browser which displays a JSP page. When the user clicks on a link or a button in the JSP a request is send to the server. On the server the request is processed by a servlet which interacts with other components of the STAEREX system. After the processing a response is send to the browser which includes a JSP and object in the session.

Within the STAEREX system there are three occasions when the client has to communicate with the server:

1. During logging in to the system.
2. While browsing through the instructional text.
3. While solving exercises.

When the user access the system the content of the file index.jsp is displayed. This page acts as the login screen. After entering user name and

---

3 http://tomcat.apache.org

---

53
password the user submits the information to the LoginServlet. This servlet checks whether the user exists in the database or not. If he does not exist the user name and the password is saved and the page firstwelcome.jsp is generated and returned to the browser. In case that the user already exists in the database the password is checked. If the password is wrong the user is redirected to the index.jsp to enter the password again. If the password is correct the page welcome.jsp is send to the browser. Additionally a session is started to identify the user. The following objects are added to the session scope to be accessible for the further usage of the system:

- A **StudentModelBean** which holds a list of the ids of exercises that have been solved by the student so far.
- An **UserBean** which holds the login information of the student.
- An instance of the **StudentModel** of the user.
- An **ExampleBean** which references the next exercise that should be solved by the student.
- A **ModulBean** that refernces the current module the student is working on.

The information stored by these objects are needed to generate the welcome page. On the welcome page the user is asked to continue the module he was working on when he logged out in the previous session or to solve a exercise from that module. Clicking on the link which leads to the next learning module initializes the communication with the ModulServlet. This servlet selects the JSP which corresponds to the learning modul that was referenced by the link the user clicked on. Furthermore it adds all examples that are used in the text of the module to the students model assuming that the student will study the instructional text before he calls the first exercise. The JSP returned to the browser is dynamic only with regard to a link which lead to the first exercise that should be solved by the student and the information from the session. Their exists a JSP for each learning module which is selected by the servlet.

When the students calls for an exercise the request is processed by the OpenExerciseServlet which adds the exercise to the ExampleBean of the session and returns the page exercise.jsp. This page contains a applet for entering the code and the exercise specific information from the ExampleBean. The applet is a structured editor which implements important aspects such as syntax highlighting, parentheses highlighting and code completion. To
implement this feature the RSyntaxArea\(^4\) is used which supports to develop programming editors. For more information on the component and how it is used the author refers to the web page of the project. The applet is embedded in a html-form that has two buttons: Run and Ask. When one of the buttons is pressed the code is read from the applet by using JavaScript and is submitted to the SchemeTutorServlet. If the user pressed the Run-button the following pages can be returned to the browser:

- **Exercise.jsp**: The user has submitted a wrong solution. A message will be displayed in the output window of the structured editor.

- **RightNextEx.jsp**: The user submitted a correct solution which is displayed on the page together with a link to the next example.

- **RightNextMod.jsp**: This page is displayed if the solution was correct and the exercise was the last of the learning module. A link to the next learning module is displayed.

- **Finished.jsp**: This page is displayed if the solution was wrong and all possible examples have already been shown to the student. The page displays the master solution.

After the communication between the client and the server as well as the internal processes of the tutor component have been described the last important component left is the data base. The model of the STAERE 5 data base is described in the following section.

### 5.3 The STAERE 5 Data Model

When developing the data model a concept has to be developed which information have to be stored by the system and how. There are several kinds of data that have been identified within the conception of the modules:

- **Examples**: Each example and exercise has to be stored.

- **Solutions**: The correct solutions committed by the student have to be stored as well as a master solution for each example.

- **Constraints**: For each solution the syntax constraints have to be stored. Problem constraints are stored for each example.

\(^4\)for source code and documentation see http://fifesoft.com/rsyntaxtextarea
- **Student**: Information to identify each student have to be collected. These has to be at least a user name and a password for logging in to the system.

- **Modules**: The lecturing content and the exercises used in each module has to be stored.

Theses information will be stored in different ways in the system. Because STAEREX is designed as a web application the instructional text will be stored within JSP files. The definition of the constraints is represented by Java classes. All other information will be stored in a data base. The data model will be explained using Structured Entity Relationship Diagrams (SERM). More information about this modelling method can be found in [FS08]. In the following important entities and their relation to other entities are regarded.

![Figure 5.6: Relationship between Student, Solution and Exercise](image)

First the relationship between student, exercise and solution is explained (see Figure 5.6). Their first generalisation that is made is that an exercises and an example are regarded as the same entity because they have the same components. Each exercise has one or more solutions. This is because an exercise can be solved in different ways by students. Furthermore an exercises must always has a solution and a solution must always be associated with an exercise. By that it is ensured that each exercise will have a master solution. Students and exercises have two kinds of relations an direct and an indirect relationship. The indirect relation models the fact that an exercise can have more solutions which are associated with different students. When a student commits an correct solution to the system this solution is stored and associated with the student. This is modelled by the the relationship type Sol_Stud. Their exists a student named expert in the system. This
user is associated with the master solution of an exercise. The direct relation between student and exercise is modelled by the relationship type Stud_ Ex. This relation is used to store which exercises have been successfully solved by the student.

![Figure 5.7: Relationship between Constraint, Solution and Exercise](image)

The second important issue is how constraints are associated with the solutions and the exercises. The SERM shown in Figure 5.7 models the distinction between problem and syntax constraint. This is because the two types of constraints are associated with different entities. A problem constraint is related to an exercise. One problem constraint can be associated with different exercises. A syntax constraint is related to one or more solutions by the relationship-type Sol_Const. By this the fact is modelled that a solution can has one or more syntax constraints and that a syntax constraint is associated with different solutions.

The third important issue is the instructional sequencing (see Figure 5.8). Therefore an entity Module exists which stores the modules associated with a number that indicates when the module should be shown to the student. The association of exercises and modules is done by the entity-relation type Sequence. This entity stores which exercises is used in a module. Furthermore it points out which exercises are used as examples within the module. Finally the Sequence stores in which sequence the student has to solve the examples.
Figure 5.8: Relationship Module and Exercise
Chapter 6

Summary

In this work the conception and the prototype of an intelligent tutoring system for introductory SCHEME programming was presented. The system is inspired by the ELM-PE system and the LISP-tutors. It adopts the idea of learning from examples from these systems but moves learning by analogy to the centre of the tutorial strategy. The system applies Constraint Based Modelling for representing the domain and the students knowledge. It extends the CBM approach by a mechanism for retrieving examples on the base of syntactic and semantic constraints. The implemented prototype covers instructional texts and examples on basic scheme operators and list processing. It implements the core components for representing the SCHEME-domain as constraints for example retrieval.

In the following some suggestions should be made how the system can be extended and which further research is needed. The fist issue can be summarized by extending the prototype to a fully running tutoring system. To do so the following steps have to be taken:

- The user interface should be redesigned according to principles of human-computer interaction.
- More exercises have to be assigned to existing units.
- New learning units have to be entered such as using operations that construct new lists from existing lists
- Instructional texts should be reviewed. Especially they should contain more graphics and maybe some Web 2.0 content like videos
- To motivate the student to reflect his solution, it should be possible to enter some descriptive text for a solution after a problem was solved correctly.
• If the STAEREX system should be used in practice it has to be embedded into another learning environment such as e-learning course or a lecture examples then text must be aligned to the content of that learning environment.

• When the system is used in practice the performance of solution evaluation and example retrieving has to be improved.

• In distant further a authoring component is needed for easily adding new learning content exercises ans constraints.

A second issue is the evaluation that has to be performed of the system because so far only the single components are tested:

• It should be evaluated in small experiments if students think that the displayed examples are useful at all.

• It can be evaluated which conditions influence the learning from analogy. Is it more help full to display a list of examples from which the student should choose? Or is it helpful to highlight the crucial part of an example on which the student should focus?

• If the tutor system is used in a lecture it can be evaluated if their was any advantage for student when using the system.

By this it becomes visible that the provided concept and system gives a starting point for further extensions of the system and research in the field of learning programming and analogical reasoning.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Copies model of the factorial function</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Conceptual design of the exercise solving environment</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Steps in designing learning content</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>The class <code>SchemeInterpreter</code></td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>The class <code>SchemeTutor</code></td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>Tree structure of a <code>SchemePair</code> object</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>The class <code>SchemeInterpreter</code></td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>General process of communication between client and server</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>Relationship between Student, Solution and Exercise</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>Relationship between Constraint, Solution and Exercise</td>
<td>57</td>
</tr>
<tr>
<td>5.8</td>
<td>Relationship Module and Exercise</td>
<td>58</td>
</tr>
<tr>
<td>1</td>
<td>Login Page</td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>The Editor</td>
<td>72</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Analogy between the solar system and the structure of atoms . 8
4.2 Characteristics of the target audience . . . . . . . . . . . . . . . . . 43
4.3 Learning modules of the STAEREX system . . . . . . . . . . . . 46
List of Algorithms

1  Retrieve example for a constraint  . . . . . . . . . . . . . . . . . . . . . . . 40
2  Parsing of a SchemePair object  . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Bibliography


[SS86] Spohrer, James C.; Soloway, Elliot: Novice mistakes: are the folk wisdoms correct? In: *Communications of the ACM* 29 (1986), Nr. 7, S. 625–632


Appendix

First steps using the prototype

To access the STAEREX prototype it is necessary to be in the network of the University of Bamberg or to have established a VPN connection to the university’s network. If this precondition is fulfilled the prototype is accessible via:

http://chios.cogsys.wiai.uni-bamberg.de:8081/STAEREX/

![Figure 1: Login Page](image)

Welcome to the “Scheme Tutor for Analogic Error Retrieval and Explanation” (STAEREX). To use the system it is necessary to login with a username and a password. If you used the system before then use the username and password you created during the first login. If you use the system for the first time, simply enter any username and password. This is necessary to track your progress during the usage of the system.

When entering the address you will be redirected to the login page. There you have to enter your user name and a password. If you login for the first
time an account will be created.

After the login you will be directed to the welcome screen of the system. This screen has two areas:

1. The main window in the center of the screen, where the system is welcoming you. On this window you can see a link which will take you to the module that you have to study next. If you already started to work with the system an additional link to the next exercise you have to work on will be displayed.

2. On the left side of the screen an additional window is placed for logging out from the system.

![Figure 2: The Editor](image)

When you click on the link which takes you to the next module the instructional text of this module is shown. Read the text and the examples carefully. From that moment the system will assume that you are familiar with the examples and concepts explained in the text. At the button of the page is a link for starting the first exercise.

In the exercise window the exercise text is displayed together with some input-output examples that illustrate the behaviour of the function you have
to write. For writing the function on editor is embedded on the page. The
editor consists of the following parts.

- The upper part of the editor is the the input window. Here you can
  enter your code.

- The lower part of the window is the output window. Here the messages
  from the system are displayed in case that you made an error in your code.

Below the window there are two buttons that start evaluation and tutoring
process:

- The Run Button: Use this button to test if your solution is correct. If
  it is correct the system will tell you by redirecting you to a new page
  which will provide a link to the next exercise or module

- Use that button to ask the system to retrieve an example that should
  help you solving the problem. The example will be displayed on the
  left side of the screen.

How to: Adding new content

Adding a new exercise.

To enter new examples a form can be used. The system must run on a server
to access the form. The url of the page containing the form is:

```
hostname : port/STAEREX/JSP/eingabeBeispiele.jsp
```

When submitting the fully filled form the following data are added to the
data base:

- The data of the exercise is added to the table `schemexercise`.
- The relation between the student and the exercise is established in the
  table `studexample`
- The solution for the exercise is inserted to the `solution` table.
- The relation between the student and the solution is established in the
  table `solutionstudent`
- The relation between the solution and the exercise is established in the
  table `exercisesolution`
• The relation between the solution and the syntax constraints is established in the table `solutionconstraint`.

After that some additional information have to be entered to the database manually either by using the mysql console or a graphical user interface like phpmyadmin.

The table `exsequence` has to be updated. The following information have to be added to the table:

• **modul_id:** The id of the module the exercise belongs to.

• **ex_id:** The exercise id of the exercise that was inserted. This is the id generated by the system when the exercise was inserted to the `scheme-exercise` table.

• **prio:** The priority of the example indicating when it should be solved by the student.

• **training:** If the exercise is used as an example in the module this value has to be 1. If it is an exercise that should be solved by the student this value has to be 0.

**Adding a new module**

When adding a new module two things have to be done:

1. The table `modul` of the database has to be updated.

2. A new JSP has to be created for the module.

Before the JSP file can be created the database table for modules has to be updated using the following SQL statement:

```sql
insert into modul (description) values (<Modul_Name>);
```

The system generates an id for the added module. Note this id because it is needed for naming your JSP file.

For creating the JSP file there exist a template file for modules which should be used for new modules. There is a reference in the file where the new content should be added. The file can be found in the projects directory under the following path:

`STAEREX/WebContent/JSP/Lectures/template.jsp`
The name of the JSP file has to be created according to the following convention:

\(<\text{Modul}\_\text{Number}>\text{<Modul}\_\text{Name}>.jsp\)

The <Modul_Number> is the id which was generated when the <Modul_Name> was inserted to the modul table.

**Adding New Constraints**

To add new constraints the following steps have to be taken:

1. Register the constraint in the data base.
2. Implement a class for the constraint.
3. Add an indication for the constraint.
4. Adding the constraint to the SchemeDomainModel class.

To register the constraint in the database the following SQL-statement is used:

```
insert into schemeconstraint (description)
    values (<Constraint_Name>);
```

After that you have to implement the class for the constraint. If it is a syntax constraint it must extend the abstract class `SchemeSyntaxConstraint`, if it is a problem constraint it has to extend the class `SchemeProblemConstraint`. You have to implement all abstract methods for the constraints to work properly. The classes can be found in the package com.staerex.constraints.

After that you have to edit the enumeration `FailIndications` from the package com.staerex.helper. You have to add an indication for the constraint and a description using the constraint name you have added to the database.

After that you have to update the class `SchemeDomainModel` depending on the constraint the method `checkForSyntaxConstraint` respectively `checkForProblemConstraint` has to be changed. Their you have to add your constraint to the central if-control structure. Just have a look how the if-statement looks for the other constraints. The class lies in the package com.staerex.tutor.

Finishing the last step you have successfully added a new constraint to the system.
Instructional Text

Basic Operators and Defining Functions

To use basic arithmetic operators we have to know how to call a function in SCHEME. For example if we want to add the numbers 4 and 5 we have to type the following:

```
[1] > (+ 4 5);
[2] >> 9
```

In the first line we can see the call of the function + with the two arguments 4 and 5. An function call has the form of a list containing the elements + 4 5. The SCHEME interpreter knows that + is a function and evaluates the proceeding arguments according to the internal method for +.

It is possible to combine different operators. Look at the following expression:

```
[1] > (+ (* 2 3) 5);
[2] >> 11
```

In this example we have a nested structure of two different operators + and *. To visualize the evaluation of the expression we can use a function tree:

The root element is the +. First the left side of the tree is evaluated. So we take the left branch of the tree and find the * which is the operator for multiplication. In the next step the arguments of the times operator are evaluated. Because they are numbers they are simply multiplied. The result of the multiplication is 6. So we know that the fist argument of the addition is 6. The second argument of the addition is a number thus 5 is added to 6 which gives the final result of 11.

To summarize the usage of basic operators we can formulate the following structure of a function call in SCHEME:

```
(<function name><argument>...)
```

In our examples the function name were the plus (+) and times (*) operator. Arguments in the examples have been numbers like 4 and 5 in (+45) or further function calls like the argument (*23) in the second example.

Defining our own functions
If we want to write our own function we can use the keyword \textit{define}. The general structure of define is the following:

$$(\textit{define} \ (<\textit{function}\_\textit{name}> <\textit{argument}>...) (<\textit{function}\_\textit{body} >))$$

To explain the structure we will have a look at a concrete example:

\begin{verbatim}
( define (add1 n)
     (+ 1 n)
 )
\end{verbatim}

A user can use the function in the following way:

\begin{verbatim}
> (add1 9)
>> 10
> (add1 -1)
>> 0
\end{verbatim}

The function add1 calculates the successor of a number. It gets one argument which is denoted by \textit{a}. The argument \textit{a} is a variable which is used in the function body for calculation. The function simply adds one to the number given by the user as argument to the function.

It is possible to write functions that have more than two arguments. For example consider a function that calculates the surface area of a rectangle.

\begin{verbatim}
( define (area a b)
     (* a b)
 )
\end{verbatim}

Here the two function arguments represent the length of the two sides of a rectangle which are multiplied in the function body.
Using conditional statements.

If we want to write more complex functions we often encounter points where a decision has to be made depending on the value of an user input. Have a look at the following example of a function which tells us if a number is positive or negative:

```lisp
(define (arsign n)
  (if (< n 0) "negative" "positive")
)

> (arsign 3)
>> positive
> (arsign −1)
>> negative
> (arsign 0)
>> positive
```

The function uses the IF-statement to determine whether a number is greater or smaller then zero. The general structure of an IF-statement is as follows:

```
(if(< test condition >) < true case > else case )
```

The test condition is a expression that returns true or false. In the example the test checks whether the argument n is smaller then null. If this is true the function returns "positive" otherwise "negative". Other operators for comparison of numbers are the following:

- `< n a` Is n smaller then a?
- `> n a` Is n greater then a?
- `<= n a` Is n smaller or equal to a?
- `>= n a` Is n greater or equal to a?
- `= n a` Is n equal a?

The IF-Statement should be used when their exist two alternatives which should be regarded. If we have more then one condition that should be tested the COND-statement is used. The general structure of COND is as follows:

```
(cond < condition1 > < consequent1 > < condition2 > < consequent2 >
   ...else < alternative >
```

78
Consider the *arsign* example. If we want to add third output for the case that the input is equal to null:

\[
\text{(define (arsign n)}
\begin{align*}
&\text{(cond ((< n 0) "negative") } \\
&\text{((> n 0) "positive") } \\
&\text{(else "zero")})
\end{align*}
\text{)}
\]

In the COND-statement we have two conditions that are checked. First it is checked if the number n is greater than zero. In that case the function returns the message "positive". In the second case it is checked whether the number n is smaller than 0. In that case the function returns "negative". IF both conditions are not true then the else case is evaluated and returns "zero".

To regard some other alternatives to formulate conditions regard the following example that returns the type of its argument:

\[
\text{(define (typeof n)}
\begin{align*}
&\text{(cond ((null? n) "null") } \\
&\text{((number? n) "number") } \\
&\text{((symbol? n) "symbol") } \\
&\text{(else "unknown type")})
\end{align*}
\text{)}
\]

The predicates for testing the types can be summarized as follows:

\[
\begin{align*}
\text{(null? n) & Is n null or the empty list? } \\
\text{(number? n) & Is n a number? } \\
\text{(symbol? n) & Is n a symbol?}
\end{align*}
\]

If we want to create a condition which checks for equality we can use the =-operator for numbers or the equal? operator for other data types. An example is the following function that checks if a letter is a vowel.

\[
\text{(define (vowel? n)}
\begin{align*}
&\text{(cond ((equal? n "a") #t))}
\end{align*}
\text{)}
\]
((equal? n "e") #t)
((equal? n "i") #t)
((equal? n "o") #t)
((equal? n "u") #t)
(else #f)
List processing I

After considering operations on numbers let’s have a look on the processing of lists. Lists are data structures that can contain different values. For example the list `(1 2 3 4)` contains the numbers 1, 2, 3 and 4. The empty list is denoted by `()`. Another example would be a shopping list like:

- bananas
- potatoes
- milk
- pizza

That can be expressed in Scheme as: `(bananas potatoes milk pizza)`

For the beginning there are two basic operators which can be used on a list. First the `car`-operator which returns the first element of a list and the `cdr`-operator which returns the rest of a list.

```
[1]> (car '(bananas potatoes milk pizza))
>> bananas
[2]>
[3]>
[4]> (cdr '(bananas potatoes milk pizza))
>> (potatoes milk pizza)
```

Applying `car` on the shooting list returns the first element which is "bananas". Applying `cdr` on the same list returns the rest of the list "(potatoes milk pizza)" without the first element.

Let us look at the following examples:

```
[1]> (car '())
[2]>
[3]>
[4]> (cdr '(pizza))
```

If we apply `car` on the empty list we get a result the empty list. If we apply `cdr` on a list with one element the empty list is returned.

**Iteration through lists.**

To see how the functions `car` and `cdr` can be applied in list processing let us look at following problem. The task is to write a function that checks if a list contains a number.
To solve this (and the following) problems the concept of recursion is used. Recursion means that a function is calling itself on a smaller subset on the problem. In the example the function `hasnumber?` is called with rest of the list passed by the user. To produce the desired output the function checks if the first argument of the list, which is extracted with `car` is a number. Another condition is needed to stop the recursive call of the function which is reached when the first element of the list is the empty list. In that case the function returns false. The behaviour can be summarized as follows:

- Check if the first element of the list is the empty list. If yes then return false, if not check the next condition.
- Check if the first element of the list is a number. If yes then return true, if not check the else.
- Call the function on the rest of the list, to check if a number is contained in the rest of the list.

A similar problem is to check if all elements of a list are numbers:

```scheme
(define (allnumber? list)
  (cond ((null? list) #t)
        ((number? (car list)) (hasnumber? (cdr list)))
        (else #f))
)
```

82
The conditional structure looks like this:

- If the list is empty then all elements have been tested and no element which is not a number has been found. Return true.

- If the first element of the list is not a number then everything is ok and the rest of the list is processed by calling allnumber? on the rest of the list.

- When the Else-Case is reached that means that the first element of the list is not a number thus return false.

As the two examples show it is important to ask the following question while programming a recursive function.

- How can the problem be reduced? Which arguments have to be passed to the recursive call?

- What processing should be done by the function? In our examples the processing was to return true or false.

- What is the stopping condition?
The functions regarded in the last module did return an answer in the form of true or false. Now we want to regard problems where list elements are returned. The first example is called \textit{first\_greater}:

\begin{verbatim}
(define (first_greater t list)
  (cond
    ((null? (car list)) #f)
    ((< t (car list)) (car list))
    (else (first_greater t (cdr list))))
)
\end{verbatim}

\begin{verbatim}
> (first_greater 5 '(2 -3 4 1 6 8))
>> 6
> (first_greater -2 '(-5 -3 -1 6))
>> -1
\end{verbatim}

The function has two parameters the first is a number and the second is a list. The function returns the first value in the list that is greater than the first argument.

The structure of the conditional expression can be described in the following way:

1. Check if the first element of the list is the empty list. If yes then return false, if not check the next condition.

2. Check if the first element of the list is greater than the first parameter. If yes then return the first element of the list, if not check the else case.

3. Call the function on the rest of the list, to check if there is a number greater then the first parameter in the rest of the list.

The example is similar to the \textit{hasNumber?} example from the previous module. But it differs to it in two ways:

- In the second condition the first element of the list is compared with the first parameter. If this check is true the first element of the list is returned.
• When calling the *first-greater* example recursively two parameters have to be passed, the first parameter that should be compared and the rest of the list. The first parameter stays the same for all recursive calls but the list has to be reduced by using cdr.

Like in the the last module there are other problems thinkable which have to iterate over the whole list before a result is returned. Consider the following example:

```lisp
(define (sumlist list)
  (if (null? (car list)) 0
      (+ (car list)
          (sumlist (cdr list))))
)

> (sumlist '(1 2 3))
>> 6
> (sumlist '(-1 2 3))
>> 5
```

This example differs in such a way from the other examples that values have to be summed up after the recursive call is finished. The following graphic illustrate how the final result is evaluated for the example `(sumlist '(1 2 3)).

Each box in the picture represents a recursive call. When the last box is reached the list is the empty list thus the if condition in the function `(null? (car list)) becomes true and 0 is returned by the last box. This 0 is passed up to the box which calculates `sumlist '(3). The result of this box can now be calculated as `3 + 0 = 3. This result is passed up to the next box. This processes continues until the first box is reached. Now the overall result can be returned which is 6.
Constraints

Syntax Constraints

Define Constraint
Template for the Scheme Expression:
\[ \text{define (<function name>)(<parameterlist>) (function body)} \]

Derived constraint:
IF the S-Expression starts with define,
THEN their have to exist two arguments a function head an a body,
AND the function definition has to contain two arguments,
AND their has to exist a S-Expression in the body.
INDICATION Show a define-example.

Basic Operator Constraint
Template for the Scheme Expression:
\[ (<operator>(<parameterlist>) ) \]

Derived constraint:
IF the S-Expression starts with a basic operator,
THEN the number of parameters must be greater or equal to two.
INDICATION Show a basic-operator-example.

Predicate Constraint
Template for the Scheme Expression:
\[ (<Predicate?>(<parameterlist>) ) \]

Derived constraint:
IF the S-Expression starts with a predicate,
THEN the expression must have at least one argument.
INDICATION Show a predicate-example.

If Constraint
Template for the Scheme Expression:
\[ \text{if (<condition>) <true-case><else-case>} \]

Derived constraint 1:
IF the S-Expression starts with if ,
THEN the condition has to be surrounded by parentheses,
AND the condition must perform a test,
AND the true case and the else case must be surrounded by parentheses,
OR the true case and the else case must start with ’ or ” or # .
INDICATION  Show an if-example.

Derived constraint 2:
IF  the S-Expression starts with if,
THEN  the true case is not allowed to be empty.
INDICATION  Show an example similar to the current problem.

Condition Constraint
Template for the Scheme Expression:

\[
\text{(cond}\ <\ \text{condition1}\ >
<\ \text{consequence1}> <\ \text{condition2}> \\
<\ \text{consequence2}> \ldots \ \text{else}\ <\ \text{alternative}>
\]

Derived constraint 1:
IF  the S-Expression starts with cond,
THEN  the condition has to be surrounded by parentheses,
AND  the condition must perform a test,
AND  each consequence argument must be surrounded by parentheses,
OR  each consequence argument must start with ’ or ” or # .
INDICATION  Show a cond-example.

Derived constraint 2:
IF  the S-Expression starts with cond,
THEN  the consequence arguments are not allowed to be empty.
INDICATION  Show an example similar to the current problem.

Car Constraint
Template for the Scheme Expression:
\[
\text{car}\ <\ \text{list}> )
\]

Derived constraint:
IF  the S-Expression starts with car,
THEN  the argument after car is surrounded by parentheses,
OR  the argument after car does not contain spaces.
INDICATION  Show a car-example.

Cdr Constraint
Template for the Scheme Expression:
\[
\text{cdr}\ <\ \text{list}> )
\]
Derived constraint:
IF the S-Expression starts with cdr,
THEN the argument after car is surrounded by parentheses,
OR the argument after car does not contain spaces.
INDICATION Show a cdr-example.

User Function Constraint
Template for the Scheme Expression:
| <function name><parameterlist>) )

Derived constraint:
IF the S-Expression starts with an identifier of a user defined function,
THEN the number of parameters in the parameter list must match the
number of parameters defined in the function definition.
INDICATION Show example of a function that has one or more arguments.

Problem Constraints

OR-Predicate Problem Constraint
IF the problem is a OR-predicate problem,
THEN their has to be a test for the empty list returning false,
AND their has to be a recursive call on the rest of the list,
AND their has to be a test on the first element of the list returning true.
INDICATION Show OR-predicate problem example.

AND-Predicate Problem Constraint
IF the problem is a AND-predicate problem,
THEN their has to be a test for the empty list returning true,
AND their has to be a recursive call on the rest of the list,
AND their has to be a test on the first element of the list returning false.
INDICATION Show AND-predicate problem example.

OR-Selector Problem Constraint
IF the problem is a OR-Selector problem,
THEN their has to be a test for the empty list returning false,
AND their has to be a recursive call on the rest of the list,
AND their has to be a test on the first element of the list returning the first element of the list.
INDICATION  Show OR-Selector problem example.

AND-Selector On Numerical List Problem Constraint

IF the problem is a AND-Selector problem,
THEN their has to be a test for the empty list returning one or zero depending on the action taken in the recursive call,
AND their has to be a recursive call using a build operator (plus or times),
AND the recursive call is applied on the rest of the list.

INDICATION  Show AND-Selector problem example.
List of Exercises

Successor of a number (Basic Problem)
Write a function, that calculates the successor of an integer. *Hint:* The successor is calculated by adding one to the given number.

```
(define (add1 i) (+ i 1))
```

Predecessor of a number (Basic Problem)
Write a function, that calculates the predecessor of an integer. *Hint:* The predecessor is calculated by subtracting one from the given number.

```
(define (sub1 i) (- a 1))
```

Surface of a square (Basic Problem)
Write a function that calculates the surface area of a square given the length of the side "a" as an integer.

```
(define (areaSquare a) (* a a))
```

Surface of a rectangle (Basic Problem)
Write a function, that calculates the surface area of a rectangle, given the length of the sides "a" and "b" of the rectangle as integers. Assume that the two values are provided as positive numbers.

```
(define (areaRect a b) (* a b))
```

Surface of a triangle (Basic Problem)
Write a function, that calculates the surface area of a triangle, given the length of the side "a" and the corresponding height "h" as integers. Assume that the two values are provided as positive numbers.

```
(define (areaTri a h) (* 0.5 a h))
```

Surface of a geometric figure (Basic Problem)
Write a function that calculates the surface area of a geometric figure. The first parameter is the type of the geometric (square, rectangle, or triangle). If the first parameter has any other value, an error message should be displayed. The second and third parameter are used for calculation:
- **Square**: Only the second parameter is used for calculation. The third parameter is ignored.
- **Rectangle**: The two parameters represent the two sides of the rectangle.
- **Triangle**: The two parameters represent a side of the triangle and the corresponding height.

```
define (area f a b)
  (cond
    ((equal? f "square") (* a a))
    ((equal? f "rectangle") (* a b))
    ((equal? f "triangle") (* 0.5 a b))
    (else "Error Message")
  )
```

**Type of an argument (Basic Problem)**
Write a function that returns the type of its argument. The function should check whether the argument is a number, a symbol or null. If the argument does not match to one of these types, the function should return the short message "Other type".

```
define (type-of a)
  (cond
    ((null? item) "null")
    ((number? item) "number")
    ((symbol? item) "symbol")
    (else "Other type")
  )
```

**Has Temperature? (Basic Problem)**
Write a function that takes as his argument a number. The number represents the temperature of a person and should return true if the person has a temperature and false otherwise.

```
define (hasTemp? t) (if (< t 38) #f #t)
```

**Arithmetic sign (Basic Problem)**
Write a function that has one parameter. The function should return "positive" if the parameter is greater or equal to zero and "negative" if the parameter is smaller then zero.
\[(\text{define (arsign } i) (\text{if} (< i 0) "negative" "positive")))\]

**Arithmetic sign with zero (Basic Problem)**
Write a function that has one parameter. The function should evaluate if a number is positive or negative. If the number is equal to 0, the function should return "zero".

\[(\text{define (arsign0 } i)\]
\[\text{(cond}\]
\[\text{((< i 0) "positive")}\]
\[\text{((> i 0) "negative")}\]
\[\text{(else "zero")}\]
\[\text{)}\]

**Capital of a country (Basic Problem)**
Write a function that returns the capital of the three countries Germany, France, Poland. If the user enters another country the function should respond with: "I do not know!".

\[(\text{define (capital s)}\]
\[\text{(cond}\]
\[\text{((equal? s "Berlin"))}\]
\[\text{((equal? s "Paris"))}\]
\[\text{((equal? s "Warsaw"))}\]
\[\text{(else "I do not know")}\]
\[\text{)}\]

**Second (Basic Problem)**
Write a function that returns the second argument of a list.

\[(\text{define (second list)} (\text{car (cdr list))})\]

**Third (Basic Problem)**
Write a function that returns the third argument of a list.

\[(\text{define (third list)} (\text{car (cdr (cdr list))}))\]
Has a list a number? (Or-Predicate Problem)
Write a function that checks whether a list contains a number or not. If the
list contains a number the function returns true otherwise false.

```
(define (hasNumber? list)
  (cond
   ((null? list)#f)
   ((number? list)#t)
   (else (hasNumber? list)(cdr list))
  )
)
```

List of Numbers (And-Predicate Problem)
Write a function that check if a list contains only numbers. The function
returns true if the list contains only numbers otherwise it returns false. For
simplification the function returns true if the argument is the empty list.

```
(define (allNumber? list)
  (cond
   ((number? car (list))
    (allNumber? (cdr (list))))
   ((null? list)#t)
   (else #f))
)
```

The member function (Or-Predicate Problem)
Write a function that expects two parameters the first. The function should
check if the first parameter is a member of the list given by the second
parameter.

```
(define (member? t list)
  (cond
   ((null? car (list))#f)
   ((equal? t (car list))true)
   (else (member? t (cdr list)))
  )
)
```

List of symbols (And-Predicate Problem)
Write a function that checks if a list consists only of symbols. As a simplifi-
cation passing the empty list to the function returns true.
(define (allSymbols? list)
  (cond
   ((symbol? (car list))
    (allSymbols? (cdr list)))
   ((null? (car list)) #t)
   (else (allSymbols? (cdr list)))
  ))

Is list increasing? (And-Predicate Problem)
Write a function that checks if a list of numbers is in increasing order.

(define (increasing? list)
  (cond
   ((null? (cdr list)) #t)
   ((< (car list) (car (cdr list)))
    (else (increasing? (cdr list)))
   ))
)

First greater (Or-Selector Problem)
Write a function that has two parameters. The function should return the value of a numeric list (second parameter) which is greater then the first parameter. If there is no greater value in the list the function returns false.

(define (firstGreater t list)
  (cond
   ((null? (car list)) #f)
   ((< t (car list) (car list))
    (else (firstGreater t (cdr list)))
   ))
)

First smaller (Or-Selector Problem)
Write a function that has two parameters. The function should return the value of a numeric list (second parameter) which is smaller then the first parameter. If there is no smaller value in the list the function returns false.

(define (firstGreater t list)
  (cond
   (else...
First sublist (Or-Selector Problem)
Write a function that returns the first sub list from a list. If the list does not contain a sub list the function returns false.

```
(define (firstSublist list)
  (cond
    ((null? (car list)) #f)
    ((list? (car list))(car list))
    (else firstSublist (cdr list)))
)
```

Sum of a list (And-Selector Problem on Numerical List)
Write a function that calculates the sum of a numeric list.

```
(define (sum-list list)
  (if
    (null? (car list)) 0
    (+ (car list) (sum-list (cdr list))))
)
```

Length of a list (And-Selector Problem on Numerical List)
Write a function that returns the length of a list. Only top level elements should be counted. Nested elements are regarded as one element.

```
(define (myLength list)
  (if
    (null? (cdr list)) 0
    (+ 1 (myLength (cdr list))))
)
```
Erklärung
Ich erkläre hiermit gemäß 17 Abs. 2 APO, dass ich vorstehende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

05.11.2007

Unterschrift