# Scheme Tutor: Notes on Design and Implementation

Sowmya Ramachandran
Department of Computer Sciences,
University of Texas,
Austin, TX 78712

Email : sowmya@cs.utexas.edu

August 30, 1996

## 1 Data Structures

The tutor uses the following data structures:

1. tutorial
2. lesson
3. problem
4. errors
5. explanations

In the following subsections, I describe the data structures and their implementation. These data structures are defined in the files *explanations.lisp* and *problems.lisp*.

### 1.1 Tutorial

- **Abstract**: Collection of *lessons* on topics.
- **Implementation**: List of *lessons*.
- **Access functions**:
  1. (get-lesson-no lesson-no): Access a particular lesson from the tutorial.
  2. dolist, do etc.: Iterate over lessons in a tutorial

  **Note:** Iteration over lessons is implementation dependent, i.e. if the implementation of the tutorial data structure is modified, iterations over the structure would have to be modified as well.

## 1.2 Lesson

- **Abstract**: A collection of *problems* on a topic. A lesson also has a *name* associated with it.
- **Implementation**: a list whose first member is a list of problems, and whose second member is a symbol which is the name of the topic.
- **Access Functions**:

  1. (lesson-name lesson): Returns the name of the lesson.
  2. (lesson problems): Returns the problem set for the topic.
  3. do, dolist: Iterators over the problems in the problem set.

  **Note:** Iteration over the problem-set is implementation dependent. Thus, changing the implementation of this data structure would involve modifying the iterators wherever they are used throughout the program. However, only the implementation of the access functions need to be modified if the data structure is implemented differently.

## 1.3 Problem

- **Abstract**: A specific problem to be presented to the student associated with a collection of possible *errors*.
- **Implementation**: A list, whose first member is a problem specification (a form), and whose second member is a collection of errors.
- **Access Functions**:

  1. (problem-spec problem): Accesses the problem specification.
  2. (problem-errors problem): Accesses the associated errors.

  **Note:** Whenever the implementation of this data structure is modified, the only functions that need to are the access functions.

## 1.4 Errors

- **Abstract**: A collection of erroneous answers associated with error types.
- **Implementation**: An association list of erroneous answers and error types. Error types are implemented as symbols.
- **Access Functions**:

  1. (get-error-type errors answer): Given a collection of errors and an answer, it returns the error type associated with the answer.

**Note:** Each collection of errors is assumed to have an entry for a default error type. This is the error type that gets returned when the student makes an unanticipated error. Here is an example:

```
(define *math-problem1* '((+ 10 26)
  ((36.0 exact-error)
   (* unknown-add-error))
   ))
```

Here, if the student enters an unexpected answer, the error returned would be of type *unknown-add-error*. (The * indicates a default case). The default error type should have an associated explanation (next subsection). Otherwise, the tutor will provide no explanation or help to the student.

**Also Note:** The implementation of problems which illustrate the car/cdr functions is different. For these problems, the errors are generated and recognised dynamically. Since there is no need for pre-defined errors, that part of the data structure is used to specify the expansion of the car/cdr function. Here is an example:

```
(define *l2-problem1* '((cadadr '((now is) (the time)))
                        (car (cdr (car (cdr))))))
```

### 1.5 Explanations

- **Abstract**: A collection of explanations associated with error types.
- **Implementation**: An association list of error messages (explanations) associated with error types. Error messages are represented as strings.
- **Access Functions**:
    1. (get-error-msg error-type): returns the explanation associated with an error type.

## 2 Expanding the Tutor

In this section, I will outline the procedure for extending the tutor to include new problems and topics.

### 2.1 Adding a topic

Before adding a topic, it would be useful to prepare a list of problems relating to the topic, a list of possible errors associated with each problem, identify the error types and explanations for each error type. Once this is done, the topic may be added to the tutor. The files *problems.scm* and *explanations.scm* define the problem sets and explanations, and should provide examples for how these are to be specified. Here is the general outline:

1. For each problem, define a variable that records the problem and the error types associated with each anticipated error. Note that it is necessary to specify an default error type to handle unanticipated errors (section 1.4).

2. Add any new error type to the *explanations* structure, along with the corresponding error message or explanation.

3. Define a variable that collects all these problems in a list. The position of the problems in this list defines the sequence in which they will be presented.

4. Add this variable to the list defined as *tutorial*, making sure to assign a name to the topic. The position of the topic in this list defines the sequence in which the topics will be presented.

## 2.2 Adding a problem

In order to add a problem to a topic,

1. Follow steps 1 and 2 as above.

2. Add the newly defined problem to the list that defines the topic. Take care to position the new problem in the list such that the problems are presented in the desired sequence.

# 3 Relevant Files

All the code for the tutor is distributed among the following files:

1. explanations.scm
2. problems.scm
3. data.scm
4. cadr.scm
5. display.scm
6. shell.scm
7. load.scm

The tutor also uses macros defined in novak/cs304p/init.scm.