A Theory-Based Introductory Programming Course

Michael R. Hansen, Jens Thyge Kristensen and Hans Rischel
Technical University of Denmark, DK-2800 Lyngby
{mrh,jtk,rischel}@it.dtu.dk

Abstract - *This paper presents an introductory programming course designed to teach programming as an intellectual activity. The course emphasizes understandable concepts which can be useful in designing programs, while the oddities of today's technology are considered of secondary importance. An important goal is to fight the trial-and-error approach to programming which is a result of the students battles with horribly designed and documented systems and languages prior to their studies at university. Instead we strive for giving the students a good experience of programming as a systematic, intellectual activity where the solution of a programming problem can be described in an understandable way. The approach is illustrated by an example which is a commented solution of a problem posed to the students in the course.*

Keywords - Introductory programming, curriculum

I. Introduction

We present the rationale behind the introductory programming course in the Informatics Programme at the Technical University of Denmark. The development of this course started in 1992 and has now resulted in the book "*Introduction to Programming using SML*" published by Addison Wesley Longman in 1999 [4].

Students prior experiences with software: Software-controlled systems and programmable computers are found everywhere in our daily life. The TV-set and the video recorder are controlled by programs. Programs can be found on the Web or created using the PC. However, software appears as a mysterious and incomprehensible technology - despite the widespread use.

For example, when adjusting the channels on your TV you will play with the buttons on the remote control until you succeed - you know by experience that it would be waste of time to try to understand the manual. When introduced to programming the high-school students get the same kind of experience: Programming tools and systems are badly documented and even badly designed, so it is much easier to experiment – rather than try to get a systematic understanding.

The core of the problem is that programs and systems are so complicated and strange that analysis is not possible, and the only way to work is in a trial-and-error manner.

Thus, many students have experiences with programming systems which can be summarized as:

- The systems and the programming languages are incomprehensible and their behaviour is unpredictable.
- By enough experiments one can get them to work — but there is no point in trying to get a clear understanding of them.

This negative picture reflects the immature and poor state of todays software technology. In our teaching at the university we have to cope with this fact of life as it is the starting point for our students – but we should strive for an education of our students which could help improving software technology in the future.

The first programming course: When students enter university they have already a lot of experience with programs and programming as described above. They know programming as a purely experimental activity, and they are looking forward to learn more "tricks". They do not expect programming to be a "subject" (like Mathematics or Physics) with theories and concepts of its own, independent of todays technology.

The first course in a larger computer science curriculum should in our opinion fight these (mis)conceptions and teach the students an *intellectual* approach to programming right from the beginning. The challenge is to teach the students to appreciate and strive for elegant programs which are understandable and can be communicated to and discussed with other people. Hence, the emphasis should be on understanding and thinking about how to solve a programming problem by using basic, well-understood concepts.

A means to understand a problem is to build a data model and express the algorithmic idea of the solution in terms of this model. This puts demands on the programming activity and the programming language:

The result of the programming activity must be a program reflecting the problem in the sense that the main concepts from the problem formulation must appear clearly in the program. Thus, the programming language used must provide a collection of constructs, each having a clear and intuitive explanation.

Fulfillment of this demand does not only depend on subjective qualities such as taste and style. It also depends on the ability to create and select appropriate *abstractions* for solving a programming problem. We exemplify this point below in the next section.

Selection of programming language: The challenge in designing such a course is to get the right combination of theory and practice: One should present concepts with a clear and well-understood meaning, but at the same time these concepts should be used in problem solving to achieve succinct and elegant programs and program designs, i.e. theory must be brought to practical use. For this reason we have decided to use Standard ML as it offers the following features:

- The language is very powerful in expressing structured data as well as computations on such data.
- The language is close to common mathematical notation. This means that it is not too hard for students to learn the syntax of programs.
- The language has a complete mathematical semantics. Based on this we give a clear, informal explanation of the programming language constructs, such that the students can predict the behaviour of their programs.
- There is an extensive standard library and the students are taught good habits in using program libraries.

Current technology (C, C++, Java, etc.) are taught in later courses when a conceptual basis has been created.

II. Example

We illustrate our approach by means of an example which is actually a solution to one of the exercises we ask the students to solve. The focus will be on the following issues:

- Modelling the structure of data in the problem by means of (composite) data types.
- Specifying the interface to a program by means of function types.
- Describing a functional break-down by means of the types of the auxiliary functions.

These issues state the kinds of abstractions to be made in order to achieve an elegant and understandable solution of the problem, and the SML language provides a succinct notation for writing them down. Inventing the right abstractions is, of course, a creative process which requires talent and taste.

Sample problem: When a map is coloured, the colours should be chosen so that neighbouring countries get different colours. The problem is to construct a program computing such a colouring. A trivial solution where each country always gets its own colour is not accepted. On the other hand, the solution does not have to be an 'optimal' one.

A colouring problem is shown in Figure 1. It comprises four countries `"a"`, `"b"`, `"c"`, and `"d"`, where the country `"a"` has the neighbouring countries `"b"` and `"d"`, the country `"b"` has the neighbouring country `"a"`, and so on. A solution of this colouring problem is to give one colour to the countries `"a"` and `"c"` and another colour to the countries `"b"` and `"d"`.

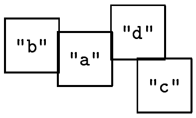

Fig. 1. Colouring problem with 4 countries

In solving this problem we represent a *country* by its name which is a string, whereas the *neighbour relation* is represented by a list of pairs of countries having a common border. For instance, the list:

```
[("a","b"),("c","d"),("d","a")]
```

defines the colouring problem in Figure 1.

The round brackets `(···)` denote tuple construction in SML. A pair is a tuple with two components. The square brackets `[···]` denote list construction, so the above neighbour relation is a list containing three pairs. This list is an example of a composite value. Such values can be entered directly to the SML system.

For convenience we use short country names like `"a"` and `"b"`. A real application of the program would, of course, use real country names like `"Honduras"`, `"El Salvador"` and `"Nicaragua"`, where `"Honduras"` and `"El Salvador"` are neighbours while `"Nicaragua"` and `"El Salvador"` are not.

The problem is to make a program to colour a map for a given neighbour relation.

The solution: To solve this problem, we must make a data model for the relevant concepts mentioned in the problem formulation. This implies that we must define the notions colour and colouring. Furthermore, the problem is so complicated that one need to break it up into several simpler problems.

Data modelling: From the problem formulation we already know that a country is modelled by a string and a neighbour relation is modelled by a list of pairs of countries. This is expressed in SML by *type* declarations:

```
type country      = string
type neighbourRel = (country * country) list
```

Hence, we use types to name important concepts and to express the model for these concepts (cf. Figure 2).

For example, `neighbourRel` is a name for the type `(country * country) list`. This type contains the *tuple* type constructor `*`. If c_1 and c_2 are values of type `country` then the tuple (c_1, c_2) is a value of type `country*country`. The *list* type constructor `list` is used in suffix form, so a value of type `neighbourRel` is a list of pairs of values of type `country`.

Type declaration	Sample value	Meta symbol
`type country      = string`	`"a"`	`c`
`type neighbourRel = (country * country) list`	`[("a","b"),("c","d"),("d","a")]`	`rel`
`type colour       = country list`	`["a","c"]`	`col`
`type colouring    = colour list`	`[["a","c"],["b","d"]]`	`colr`

Fig. 2. Data modelling for map colouring problem

An obvious choice of data representation for the concept *colour* could be a set of fixed names:

```
datatype colour = Red | Blue | ···
```

but that would not be useful for solving the problem. The essence is to express that *countries* have the *same* colour, and it turns out that it is useful to model a colour by a list of the countries having that colour. Continuing the example from the problem formulation, the list:

```
["a","c"]
```

represents a colour where the countries `"a"` and `"c"` have this colour. This would be meaningful for the neighbour relation given above as `"a"` and `"c"` are not neighbours.

We express this representation by the type declaration:

```
type colour = country list
```

This type declaration captures one of the crucial points: the use of abstractions. Of course, a list of countries is a very abstract notion for a colour. We shall see that this abstraction is useful for solving the problem.

A *colouring* is described by a list of colours:

```
type colouring = colour list
```

Hence, "expanding" the type declarations, a colouring is a list of lists of countries. For example, a colouring for the above neighbour relation is:

```
[["a","c"],["b","d"]]
```

where the countries `"a"` and `"c"` get one colour, while the countries `"b"` and `"d"` get another colour.

This data modelling is subject to certain *invariant* conditions. For example, the value `[["a","b"],["a"]]` of type `colouring` is not a proper representation of a map colouring as it gives two different colours to the country `"a"`. The invariant for values of type `colouring` is: no country may occur in more than one of the colours of the colouring. The solution to the problem should, of course, only produce colourings satisfying this invariant.

The type declarations are collected in Figure 2. The sample value column contains SML values of the indicated types, while the meta symbol column introduces identifiers to be used for parameters of the indicated type, so e.g. `col`, `col1` or the like are used for parameters denoting a `colour`.

Interface specification: The solution of the problem should be a function computing a colouring for any given neighbour relation. We give the name `makeColouring` to this function, and using the types in Figure 2 we get the specification:

```
makeColouring: neighbourRel -> colouring
```

This specification formalizes the problem in a succinct way: We must declare a function `makeColouring` where the argument has type `neighbourRel` while the value has type `colouring`. It also captures the essence that a user of the program need to know, and we say that it specifies the *interface* to the program.

Functional break-down: The next step is to construct a program for the function `makeColouring` specified in the interface. To do so, we need an algorithmic idea. The idea we will pursue here is the following: We start with the empty colouring, i.e. the empty list containing no colours. Then we will gradually extend the actual colouring by adding one country at a time.

We illustrate this algorithmic idea on the above example, where the neighbour relation contains the four countries: `"a"`, `"b"`, `"c"` and `"d"`. Thus, there are four main steps in the algorithm, one for each country:

	country	old colouring	new colouring
1.	`"a"`	`[]`	`[["a"]]`
2.	`"b"`	`[["a"]]`	`[["a"],["b"]]`
3.	`"c"`	`[["a"],["b"]]`	`[["a","c"],["b"]]`
4.	`"d"`	`[["a","c"],["b"]]`	`[["a","c"],["b","d"]]`

Fig. 3. Algorithmic idea

We give a brief comment to each step:

1. The colouring containing no colours is the empty list.
2. The colour `["a"]` is not *extendible* by `"b"` because the countries `"a"` and `"b"` *are neighbours*. Hence the colouring is *extended* by the colour `["b"]`.
3. The colour `["a"]` is *extendible* by `"c"` because `"a"` and `"c"` are not neighbours.
4. The colour `["a","c"]` is not extendible by `"d"` while the colour `["b"]` is extendible by `"d"`.

The task is now to make a program where the main concepts of this algorithmic idea are directly represented. The concepts emphasized in the above comment are:

- Test whether a colour is extendible by a country for a given neighbour relation.
- Test whether two countries are neighbours in a given neighbour relation.
- Extend a colouring by a country for a given neighbour relation.

Function type			Legend
`areNeighbours:`	`country * country * neighbourRel`	`-> bool`	Decides whether two countries are neighbours
`extendible:`	`country * colour * neighbourRel`	`-> bool`	Decides whether a colour can be extended by a country
`extend:`	`country * colouring * neighbourRel`	`-> colouring`	Extends a colouring by an extra country
`countriesOf:`	`neighbourRel`	`-> country list`	Computes a list of countries in a neighbour relation
`colCntrList:`	`country list * neighbourRel`	`-> colouring`	Builds a colouring for the countries in a list of countries

Fig. 4. Functional break-down for map colouring problem

The function specification of each of the main concepts documents the algorithmic idea. These specifications are shown in Figure 4. We have added the specification of a function `countriesOf` for extracting the list of countries occurring in a given neighbour relation and the specification of a function `colCntrList` which gives the colouring for given country list and neighbour relation.

Let `rel0` denote the neighbour relation in Figure 2:

```
val rel0 = [("a","b"),("c","d"),("d","a")]
```

To get a clear understanding of the specified functions, we illustrate the application of each function to some typical arguments.

Examples for `areNeighbours`:

```
areNeighbours("d","c",rel0) = true
areNeighbours("d","b",rel0) = false
```

as `("c","d")` is an element of `rel0` while neither `("d","b")` nor `("b","d")` is an element of `rel0`.

Examples for `extendible`:

```
extendible("d",["a","c"],rel0) = false
extendible("d",["b"],rel0)     = true
```

as `"d"` is a neighbour of `"c"` while `"d"` is not a neighbour of `"b"`.

Example for `extend`:

```
extend("d",[["a","c"],["b"]],rel0)
 = [["a","c"],["b","d"]]
```

as the colour `["a","c"]` is not extendible by `"d"` while the colour `["b"]` is extendible by `"d"`.

Example for `countriesOf`:

```
countriesOf rel0 = ["a","b","c","d"]
```

as the neighbour relation contains these four countries.

Example for `colCntrList`:

```
colCntrList(["a","b","c","d"],rel0)
 = [["a","c"],["b","d"]]
```

because this function performs the four main steps as presented in Figure 3.

Remark: The sequence of the countries in a list is in this example of no importance. For example, the lists `["c","a"]` and `["a","c"]` are two different lists representing the same colour. It is therefore acceptable that an implementation of the specified functions generate other results than the above ones. The solution given in the appendix will actually generate the colouring: `[["c","a"],["b","d"]]` for the neighbour relation `rel0`.

A solution of the colouring problem is obtained by applying the function `colCntrList` to the list of countries in a neighbour relation:

$$\begin{aligned}&\texttt{makeColouring}(\mathit{rel})\\ &\ = \texttt{colCntrList}(\texttt{countriesOf}(\mathit{rel}), \mathit{rel})\end{aligned}$$

so we can solve the map colouring problem using the functions specified in Figure 4. We also say that we have a functional break-down for this problem.

The few specifications given in Figure 4 describes the program design for our solution to the map colouring problem in a brief and useful way. Thus, such specifications constitute useful program documentation, and a description of the functional break-down will be an important part of the documentation for any program design.

In our course we teach students a systematic approach to program documentation using these concepts.

Function declarations: It remains to provide programs for the specified functions. In this section we will focus on one function. The full program is in the appendix. We consider the function:

```
extendible:
   country * colour * neighbourRel -> bool
```

In SML this function is declared by:

```
fun extendible(_,[],_)          = true
  | extendible(c, c1::col, rel) =
      not (areNeighbours(c, c1, rel))
      andalso extendible(c, col, rel)
```

where we assume that the function `areNeighbours` is already declared. The declaration contains two *clauses*.

The first clause

```
extendible(_,[],_) = true
```

expresses that the colour `[]` containing no countries is extendible by any country for any given neighbour relation. The symbol `_` occurring in the clause is a "don't care" symbol for arguments.

The second clause

```
extendible(c, c1::col, rel) =   ···
```

expresses whether or not the colour `c1::col` is extendible by the country `c` for the neighbour relation `rel`. The notion `c1::col` is a means of decomposing a list, where the first element in the list is named `c1` and the rest of the list is named `col`.

The right hand side of the clause says that the colour `c1::col` is extendible by `c` when `c` and `c1` are not neighbours in `rel`, and, furthermore, the colour `col` is extendible by `c` for a given neighbour relation `rel`.

The declaration for `extendible` is an example of a *recursive* function declaration because the identifier `extendible` occurs on the right hand side of the second clause.

Meaning of functions: We only need a few basic concepts to explain the meaning of function declarations. The key point is to explain the meaning of identifiers during a computation of function values. The notions of *binding* and *environment* are introduced to explain the meaning of identifiers, and the notion of *evaluation* is introduced to explain the step by step computation of function values.

A binding has the form $id \mapsto v$ and it associates the value v with the identifier id. An environment is a collection of bindings like the environment env shown in Figure 5.

The meaning of a function application can be described using step by step *evaluation* of expressions. We use the notation:

$$(exp_1, env_1) \rightsquigarrow (exp_2, env_2)$$

where (exp_i, env_i) is a pair consisting of an expression exp_i and an environment env_i. The symbol $\rightsquigarrow$ reads "evaluates to". We omit the environments when they are not needed.

The function application:

```
extendible("d",["b"],rel0)
```

can now be explained by a sequence of evaluation steps. The evaluation starts in an environment containing a binding of the identifier `rel0`:

$$[\ \texttt{rel0} \mapsto \texttt{[("a","b"),("c","d"),("d","a")]}\ ]$$

The first evaluation step is:

$$\begin{array}{ll} & (\texttt{extendible("d",["b"],rel0)}, [\texttt{rel0} \mapsto \cdots]) \\ \rightsquigarrow & \left(\begin{array}{l}\texttt{not (areNeighbours(c, c1, rel))} \\ \texttt{andalso extendible(c, col, rel)}\end{array}, env\right) \end{array}$$

where env is given in Figure 5. In this step the evaluation uses the second clause of the declaration because the colour `["b"]` is not the empty list. The main part of this step is to build the environment env for the identifiers `c`, `c1`, `col` and `rel` occurring in the second clause.

$$env = \left[\begin{array}{lll} \texttt{c} & \mapsto & \texttt{"d"} \\ \texttt{c1} & \mapsto & \texttt{"b"} \\ \texttt{col} & \mapsto & \texttt{[]} \\ \texttt{rel} & \mapsto & \texttt{[("a","b"),("c","d"),("d","a")]} \end{array}\right]$$

Fig. 5. Environment with bindings of `c`, `c1`, `col` and `rel`.

The further evaluation uses that the left hand side of the above `andalso` expression evaluates to `true`:

$$(\texttt{not (areNeighbours(c, c1, rel))}, env) \rightsquigarrow \texttt{true}$$

as `"d"` and `"b"` are not neighbours in the value:

```
[("a","b"),("c","d"),("d","a")]
```

associated with `rel`. Thus, the evaluation progresses as follows:

$$\begin{array}{ll} & \left(\begin{array}{l}\texttt{not (areNeighbours(c, c1, rel))} \\ \texttt{andalso extendible(c, col, rel)}\end{array}, env\right) \\ \rightsquigarrow & (\texttt{extendible(c, col, rel)}, env) \\ \rightsquigarrow & \texttt{extendible("d",[],[("a","b"),...])} \\ \rightsquigarrow & \texttt{true} \end{array}$$

where the last step uses the first clause of the declaration of `extendible`.

The introduction of the notions of binding, environment and evaluation gives the introductory programming course a theoretical flavour; but the theory is used to give a decent explanation of the meaning of programs.

Other solutions to the map colouring problem: There are many other solutions to the map colouring problem than the one presented here.

An immediate idea is to use a set of countries to model a colour rather than a list of countries, because there is no ordering among the countries in a colour. An advantage of using sets is that the need for invariants disappears. The use of sets makes the solution a little more abstract, but we have experienced that many students can cope with such abstractions.

The most elegant solution we have considered is even more abstract, as it uses library programs for both sets and binary relations. The idea is to represent the neighbour relation by a binary relation, while a colouring is represented by a partition of the countries occurring in the neighbour relation. Although this solution can be elegantly formulated in SML, we have, not surprisingly, experienced that it was too abstract for most of our students on this introductory level.

III. Course Contents

The contents of the first semester course covers several aspects:

Fundamental data structures: We introduce the fundamental data structures: numbers, characters, strings, tuples, records, lists, trees, sets, tables, functions, and their applications.

Programming paradigms: The functional paradigm has major attention in the first part of the semester. The imperative programming paradigm has some attention in the last part of the semester. An advantage of SML is that it supports both functional and imperative programming. So we need not consider more than one programming language in the first semester.

Semantical concepts: The notions of binding, environment and evaluation are introduced to explain the meaning of functional programs and the notion of store is introduced to explain the meaning of imperative programs.

Problem solving and program design: We introduce a standard way of documenting a programming solution.

Thus, we introduce some concepts which will exist beyond the next generation of programming languages. Other durable concepts e.g. from object oriented programming languages are introduced in later courses.

IV. Educational Issues

Teaching programming to university freshmen is a challenging task: 1) The students should solve interesting problems using a computer, and 2) Programming must be taught as an intellectual activity.

The first part relates directly to what is achievable using existing programming languages. The second part relates to the students attitude towards the programming activity and programming languages. In this respect it is of great importance to teach basic, well-understood concepts to create a solid foundation for further education in computer science.

The book by Abelson and Sussman [1] uses the Scheme language to present solutions for a large variety of programming problems, and they explain the meaning of programs in terms of a few basic concepts. The main difference to our approach is that Scheme is an untyped language while we have a systematic use of types for modelling and documentation as illustrated above.

Many "customers" of a computer science education ask for the students ability to design programs and concepts rather than proficiency in a particular programming language. Since 1992 we have been giving a course as described in this paper. The reactions we have received so far indicate that we are much closer to reaching this goal with our current course than with the previous Pascal course.

Acknowledgements: We are grateful for comments from F. Nielson, A.P. Ravn and J. Steensgaard-Madsen.

References

[1] Abelson, H., Sussman, J., *Structure and Interpretation of Computer Programs*, MIT Press, 1985.
[2] Tucker, A.B. (Ed.), Computing Curricula 1991, *Communications of the ACM*, 34, June, pp. 69-84, 1991.
[3] Deimel, L.E. (Ed.), *Software Engineering Education*, LNCS, vol. 423, Springer-Verlag, 1990.
[4] Hansen, M.R., Rischel, H., *Introduction to Programming using SML*, Addison-Wesley, 1999.
[5] Sestoft, P., Kristensen, J.T., Ravn, A.P., Rischel, H., From Functional to Imperative Programming, *Les languages applicatifs dans l'enseignement de l'informatique*, pp. 26-33, Actes des 2èmes journées de travail, IFSIC-IRISA, Rennes 1993.

Appendix

```
infix member;
fun _ member []     = false
  | x member (y::ys) = x=y orelse x member ys;

fun insert(x, xs) = if x member xs then xs else x::xs;

fun areNeighbours(c1, c2, rel) =
   (c1,c2) member rel orelse (c2,c1) member rel;

fun extendible as in the main text

fun extend(c, [], _)       = [[c]]
  | extend(c, col::colr, rel) =
     if extendible(c, col, rel) then (c::col)::colr
     else col::extend(c, colr, rel)

fun countriesOf []            = []
  | countriesOf((c1,c2)::rel) =
         insert(c1, insert(c2, countriesOf rel));

fun colCntrList([], _)      = []
  | colCntrList(c::cs, rel) =
   extend(c, colCntrList(cs, rel), rel)

fun makeColouring rel =
   colCntrList(countriesOf rel, rel);
```

Fig. 6. Complete solution to colouring problem

Suppose that the declarations in Figure 6 has been edited into a text file named `colouring.sml`. The coloring of the four countries in Figure 1 can then be obtained by the following dialogue with SML:

```
- use "colouring.sml";     (* compile file *)
- val rel0 = [("a","b"),("c","d"),("d","a")];
- makeColouring rel0;
> [["c", "a"], ["b", "d"]] : string list list
```